Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Комп'ютерних наук і технологій
(повне найменування факультету)

Комп'ютерні системи та мережі
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
бакалаврський
(ступінь вищої освіти)

на тему: РОЗРОБКА ВЕБПЛАТФОРМИ ДЛЯ КООРДИНАЦІЇ ВОЛОНТЕРСЬКОЇ ДІЯЛЬНОСТІ

Виконав(ла): студент(ка) 4 курсу,
групи КНТз-513сп

Спеціальності
123 Комп'ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп'ютерна інженерія
(назва освітньої програми (спеціалізації)

ГРЕБНЯК М.Ю.
(ПРІЗВИЩЕ та ініціали)

Керівник ЗЕЛЕНЬОВА І.Я.
(ПРІЗВИЩЕ та ініціали)
Рецензент РОМАНОВСЬКИЙ О. В.
(ПРІЗВИЩЕ та ініціали)

2026

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
Кафедра комп'ютерних систем та мереж
Ступінь вищої освіти бакалаврський
Спеціальність 123 комп'ютерна інженерія
(код і найменування)
Освітня програма (спеціалізація) комп'ютерна інженерія
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри КУДЕРМЕТОВ Р.К.
_____
«____»_____2026 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ГРЕБНЯКА Максима Сергійовича
(ПРІЗВИЩЕ, ім'я, по батькові)

Тема проєкту (роботи) Розробка вебплатформи для координації волонтерської діяльності
керівник проєкту (роботи) к.т.н., доцент, ЗЕЛЕНЬОВА І.Я.,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від « 14 » квітня 2026 року № 160
2. Строк подання студентом проєкту (роботи) 01.06.2026 р.
3. Вихідні дані до проєкту (роботи) опис предметної області, технології розробки клієнтської та серверної частини

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) Аналіз технологій розробки клієнт-серверних систем;
2) Розробка серверної частини системи;
3) Розробка клієнтської частини системи;
4) Випробування системи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	ЗЕЛЕНЬОВА І.Я.		
Нормоконтроль	ГОЛУБ Т.В.		

7. Дата видачі завдання «14» квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	**Примітка**
1	Аналіз предметної області та аналогів	до 18.04.2026	
2	Визначення та аналіз вимог до системи	до 21.04.2026	
3	Проєктування бази даних системи	до 25.04.2026	
4	Проєктування архітектури сервера	до 28.04.2026	
5	Розробка серверної частини	до 10.05.2026	
6	Проєктування клієнтської частини	до 14.05.2026	
7	Розробка користувацького інтерфейсу	до 22.05.2026	
8	Випробування комп’ютерної системи	до 24.05.2026	
9	Оформлення пояснювальної записки	до 26.05.2026	
10	Проходження нормоконтролю	до 30.05.2026	
11	Перевірка на наявність академічного плагіату	до 01.06.2026	
12	Проходження рецензування	до 02.06.2026	

Студент(ка) _____ Максим ГРЕБНЯК
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Ірина ЗЕЛЕНЬОВА
(підпис) (Ім’я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 96 с., 29 рис., 3 табл., 3 дод., 20 джерел.

REACT, REST API, POSTGRESQL, PRISMA ORM, ВЕБПЛАТФОРМА, ВОЛОНТЕРСЬКА ДІЯЛЬНІСТЬ, ІНФОРМАЦІЙНА СИСТЕМА, СЕРВЕР

Предмет розробки – клієнт-серверна інформаційна система для координації волонтерської діяльності.

Мета роботи – підвищення ефективності координації волонтерської діяльності шляхом розробки вебплатформи з централізованою системою керування заявками, користувачами та волонтерськими завданнями.

Новизна роботи – поєднання клієнтської частини на основі React та серверного застосунку Node.js із використанням REST API для організації роботи веб-платформи координації волонтерської діяльності.

Практична цінність роботи полягає у можливості використання розробленого програмного забезпечення для організації взаємодії між координаторами та волонтерами, автоматизації обробки заявок та централізованого керування волонтерською діяльністю.

В межах роботи виконано аналіз предметної області, сформовано вимоги до системи та обґрунтовано вибір клієнт-серверної архітектури. Реалізовано серверну частину із використанням Node.js та Express із взаємодією через REST API та передаванням даних у форматі JSON, а також клієнтську частину у вигляді вебзастосунку на основі React з компонентною структурою, маршрутизацією сторінок та динамічним оновленням інтерфейсу без повного перезавантаження сторінки. Проведено випробування користувацького інтерфейсу та аналіз продуктивності системи під час виконання типових сценаріїв роботи користувача.

ЗМІСТ

ВСТУП

Сучасний розвиток інформаційних технологій змінив підходи до організації взаємодії між людьми, до обробки інформації та координації спільної діяльності. Особливо помітно ці зміни проявляються у таких сферах, де необхідно забезпечити швидкий обмін інформацією між великою кількістю учасників. Багато сфер суспільного життя зараз потребують оперативного реагування на зміну потреб та мають у собі централізоване керування процесами.

Однією з таких сфер є волонтерська діяльність, значення якої останніми роками суттєво зросло у зв’язку з повномасштабним вторгненням росії до нашої країни [1].

Волонтерські організації та ініціативи виконують важливу роль у суспільстві: надання гуманітарної допомоги, підтримку населення та організацію взаємодії між людьми, які потребують допомоги, і тими, хто готовий її надавати [1].

У більшості випадків координація волонтерської діяльності виконується за допомогою месенджерів, соціальних мереж або окремих електронних таблиць. Такий підхід дозволяє швидко організувати взаємодію між учасниками, однак при збільшенні кількості заявок, волонтерів та координаторів виникають труднощі з централізованим керуванням інформацією. Частина заявок дублюється, окремі повідомлення можуть втрачатися серед великої кількості чатів, а контроль виконання завдань стає менш ефективним. Крім того, використання декількох різних засобів комунікації одночасно створює додаткове навантаження на координаторів та збільшує час обробки заявок.

Існуючі програмні рішення для організації волонтерської діяльності часто орієнтовані на досить вузький набір функцій, або на використання складних систем керування, що потребують додаткового адміністрування та технічної підтримки. Частина платформ має перевантажений інтерфейс і надлишкову кількість функцій, які ускладнюють роботу користувачів, а деякі системи не

забезпечують достатньої швидкодії та зручності використання на мобільних пристроях. В результаті багато волонтерських організацій продовжують використовувати розрізнені інструменти комунікації замість єдиної централізованої системи.

В таких умовах виникає необхідність створення вебплатформи, яка дозволить об’єднати основні процеси координації волонтерської діяльності в межах одного програмного середовища. Використання веб-технологій дозволяє забезпечити доступ до системи з різних пристроїв без необхідності встановлення додаткового програмного забезпечення, а також спрощує процес супроводу та оновлення програмного продукту [2]. Крім того, веб-платформа дозволяє централізовано зберігати інформацію про заявки, волонтерів, координаторів та результати виконання завдань.

Під час розробки подібних систем важливе значення мають не лише функціональні можливості, але й питання продуктивності, стабільності роботи та безпеки обробки даних [3]. Система повинна забезпечувати одночасну роботу великої кількості користувачів, підтримувати швидкий обмін інформацією між клієнтською та серверною частинами, а також гарантувати коректне зберігання даних та розмежування прав доступу між різними категоріями користувачів. Окрему увагу необхідно приділяти проєктуванню архітектури програмного забезпечення, структурі бази даних та організації механізмів авторизації користувачів.

Предметом розробки є вебплатформа для координації волонтерської діяльності. Метою роботи є підвищення ефективності координації волонтерської діяльності шляхом розробки вебплатформи з централізованою системою керування заявками, користувачами та волонтерськими завданнями.

У першому розділі виконано аналіз предметної області та існуючих програмних рішень для організації волонтерської діяльності, визначено функціональні та нефункціональні вимоги до вебплатформи, а також обґрунтовано вибір основних технологій реалізації системи.

У другому розділі наведено проєктування вебплатформи, структуру

клієнтської та серверної частин системи, організацію бази даних та архітектуру взаємодії між компонентами програмного забезпечення.

В третьому розділі описано реалізацію програмного забезпечення, структуру основних модулів системи, механізми авторизації користувачів та роботу із заявками та волонтерськими завданнями.

В четвертому розділі проведено тестування веб-платформи, перевірку працездатності системи та аналіз результатів роботи програмного забезпечення під час виконання типових сценаріїв використання.

Новизна роботи полягає у розробці вебплатформи для координації волонтерської діяльності з централізованою системою керування заявками, ролями користувачів та засобами моніторингу виконання завдань.

Практична цінність роботи полягає у можливості використання розробленого програмного забезпечення для організації взаємодії між координаторами та волонтерами, автоматизації обробки заявок та централізованого керування волонтерською діяльністю.

1 АНАЛІЗ ТЕХНОЛОГІЙ РОЗРОБКИ КЛІЄНТ-СЕРВЕРНИХ СИСТЕМ

1.1 Призначення та функціональні вимоги до системи

Волонтерська діяльність передбачає постійну взаємодію між великою кількістю учасників: координаторами, волонтерами, організаціями та людьми, які потребують допомоги. Основною задачею координаторів є швидке отримання інформації про нові потреби, розподіл завдань між волонтерами та контроль виконання заявок. У більшості випадків така взаємодія здійснюється через месенджери, соціальні мережі або окремі електронні таблиці, що створює значні труднощі при збільшенні кількості учасників та обсягу інформації [2].

Використання великої кількості чатів ускладнює пошук актуальних заявок та контроль їх виконання. Частина повідомлень може губитися серед іншої інформації, а дублювання заявок призводить до неефективного використання ресурсів.

Аналіз існуючих вебресурсів та програмних платформ показує, що більшість із них або мають вузьку спеціалізацію, або містять надлишковий функціонал, який ускладнює роботу користувачів. Частина систем орієнтована переважно на ведення внутрішньої документації, а не на оперативну взаємодію між учасниками волонтерської діяльності. У результаті багато організацій змушені використовувати декілька різних сервісів одночасно, що створює додаткові складнощі під час роботи.

У зв’язку з цим виникає необхідність створення єдиної вебплатформи, яка дозволить централізовано організувати процес координації волонтерської діяльності, автоматизувати роботу із заявками та спростити взаємодію між користувачами системи.

Призначенням розроблюваної веб-платформи є забезпечення зручного середовища для координації волонтерської діяльності, створення та обробки заявок, взаємодії між координаторами і волонтерами, а також контролю виконання поставлених завдань. Система повинна забезпечувати централізоване

зберігання інформації та надавати користувачам швидкий доступ до актуальних даних через вебінтерфейс.

На основі призначення системи формуються наступні функціональні вимоги, які описані нижче.

Функціональні вимоги до користувальницької авторизації. Система повинна підтримувати реєстрацію та авторизацію користувачів із розмежуванням прав доступу відповідно до ролей. Передбачається використання ролей адміністратора, координатора та волонтера. Після успішної авторизації користувач повинен отримувати доступ лише до тих функцій системи, які відповідають його ролі.

Функціональні вимоги до роботи із заявками. Координатор повинен мати можливість створювати нові заявки, редагувати їх та змінювати статус виконання. Для кожної заявки повинні зберігатися назва, опис, дата створення, пріоритет, поточний статус та інформація про відповідального волонтера. Система повинна забезпечувати швидкий пошук та фільтрацію заявок за різними параметрами.

Функціональні вимоги до взаємодії між користувачами. Вебплатформа повинна забезпечувати можливість призначення волонтерів на виконання окремих завдань. Координатор повинен бачити поточний стан виконання заявок та мати можливість оперативно змінювати розподіл завдань між користувачами системи.

Функціональні вимоги до моніторингу виконання завдань. Система повинна підтримувати зміну статусів заявок, наприклад: «нова», «в роботі», «виконана» або «скасована». Це дозволяє координаторам контролювати поточний стан виконання завдань та швидко отримувати інформацію про незавершені заявки.

Функціональні вимоги до клієнт-серверної взаємодії. Вебплатформа повинна забезпечувати обмін даними між клієнтською та серверною частинами системи у режимі реального часу. При зверненні користувача до сервера має виконуватись отримання актуальної інформації про заявки, користувачів та результати виконання завдань. Серверна частина повинна коректно обробляти одночасні запити від декількох користувачів.

Функціональні вимоги до адміністрування системи. Адміністратор повинен

мати можливість керувати обліковими записами користувачів, змінювати ролі, блокувати доступ до системи та контролювати загальний стан вебплатформи. Адміністративні функції не повинні бути доступні звичайним користувачам.

Функціональні вимоги до збереження даних. Вся інформація про користувачів, заявки та результати виконання завдань повинна централізовано зберігатися у базі даних. Система повинна забезпечувати коректне оновлення та збереження інформації без втрати даних під час одночасної роботи декількох користувачів.

Функціональні вимоги до масштабованості системи. Архітектура веб-платформи повинна передбачати можливість подальшого розширення функціональності без необхідності повної переробки програмного забезпечення. Зокрема, система має підтримувати можливість додавання нових модулів, категорій заявок або додаткових ролей користувачів.

Сформульовані функціональні вимоги визначають основні задачі, які повинна вирішувати вебплатформа, та створюють основу для подальшого проєктування архітектури системи, структури бази даних і механізмів взаємодії між компонентами програмного забезпечення.

1.2 Нефункціональні вимоги до системи

Крім функціональних можливостей, важливе значення для вебплатформи координації волонтерської діяльності мають також і нефункціональні характеристики системи [3].

Саме вони визначають стабільність роботи програмного забезпечення, швидкодію, зручність використання та можливість подальшого розвитку системи.

Для вебплатформи, яка передбачає одночасну роботу декількох категорій користувачів та постійний обмін даними між клієнтською і серверною частинами, нефункціональні вимоги безпосередньо впливають на ефективність роботи всієї

системи.

Вимоги до продуктивності та швидкодії. Вебплатформа повинна забезпечувати швидку обробку запитів користувачів та мінімальний час відгуку під час роботи із заявками. При відкритті сторінок, перегляді списку завдань або зміні статусів система не повинна створювати помітних затримок для користувача. Серверна частина має підтримувати одночасну роботу декількох користувачів без зниження продуктивності. Особливу увагу необхідно приділити оптимізації роботи із базою даних та зменшенню кількості зайвих запитів до сервера.

Вимоги до надійності та стабільності роботи. Система повинна забезпечувати стабільну роботу навіть при великій кількості активних користувачів або тимчасових помилках мережевого з’єднання. У випадку виникнення помилок веб-платформа не повинна втрачати дані користувачів або завершувати роботу аварійно. Усі дії, пов’язані зі створенням та оновленням заявок, повинні коректно зберігатися у базі даних.

Вимоги до безпеки системи. Веб-платформа повинна забезпечувати захист облікових записів користувачів та розмежування прав доступу відповідно до ролей. Дані авторизації мають передаватися та зберігатися у захищеному вигляді. Система повинна обмежувати доступ до адміністративних функцій для звичайних користувачів та запобігати несанкціонованій зміні інформації. Також необхідно передбачити перевірку вхідних даних для зменшення ризику помилок або атак на серверну частину системи.

Вимоги до масштабованості. Архітектура програмного забезпечення повинна передбачати можливість подальшого розширення функціональності системи без повної переробки програмного коду. Вебплатформа має підтримувати можливість збільшення кількості користувачів, додавання нових модулів або розширення структури бази даних без суттєвих змін базової архітектури системи.

Вимоги до сумісності та доступності. Система повинна коректно працювати у сучасних веббраузерах та підтримувати використання на різних типах пристроїв. Інтерфейс вебплатформи має адаптуватися до різних розмірів екранів

та забезпечувати однакову логіку взаємодії як на персональних комп’ютерах, так і на мобільних пристроях.

Вимоги до зручності використання. Інтерфейс системи має бути зрозумілим для користувачів без необхідності тривалого навчання. Основні функції вебплатформи мають бути доступні за мінімальної кількості дій з боку користувача. Навігація між сторінками повинна бути логічною та не створювати труднощів під час роботи із заявками або перегляду інформації.

Вимоги до підтримуваності програмного забезпечення. Програмний код веб-платформи повинен мати модульну структуру та бути організованим таким чином, щоб спростити подальше внесення змін або виправлення помилок. Використання окремих модулів для клієнтської та серверної частин дозволяє зменшити зв’язність між компонентами системи та полегшує супровід програмного забезпечення.

Вимоги до збереження та цілісності даних. База даних системи повинна забезпечувати коректне зберігання інформації про користувачів, заявки та результати виконання завдань. Під час одночасної роботи декількох користувачів система не повинна допускати втрати або пошкодження даних. Усі зміни інформації мають виконуватися контрольовано та коректно відображатися у веб-платформі.

Сформульовані нефункціональні вимоги визначають основні якісні характеристики вебплатформи та створюють основу для подальшого вибору архітектури системи, технологій реалізації та способів організації взаємодії між компонентами програмного забезпечення [3, 4].

1.3 Вибір технологій реалізації системи

Для реалізації веб-платформи координації волонтерської діяльності необхідно обрати технології, які зможуть забезпечити стабільну клієнт-серверну

взаємодію, зручну роботу користувача з вебінтерфейсом, надійне зберігання даних та можливість подальшого розширення системи [5]. Оскільки платформа передбачає одночасну роботу кількох категорій користувачів, зокрема адміністратора, координатора та волонтера, обрані технології мають підтримувати розмежування доступу, обробку запитів, збереження стану заявок і швидке оновлення інформації на клієнтській стороні.

На першому етапі було розглянуто загальний підхід до побудови системи. Для такої задачі найбільш доцільною є клієнт-серверна архітектура, за якої клієнтська частина відповідає за відображення інтерфейсу та взаємодію з користувачем, а серверна частина виконує обробку запитів, роботу з базою даних, перевірку прав доступу та формування відповідей. Такий підхід дозволяє розділити логіку представлення даних і логіку їх обробки, що спрощує підтримку системи та її подальший розвиток.

Для реалізації клієнтської частини було розглянуто декілька варіантів: використання звичайних HTML, CSS і JavaScript без фреймворків, застосування бібліотеки React, фреймворку Vue.js або Angular. Найпростішим варіантом є створення інтерфейсу на основі HTML, CSS і JavaScript без додаткових інструментів. Такий варіант підходить для невеликих статичних сайтів, однак для веб-платформи з авторизацією, особистими кабінетами, списками заявок, фільтрацією та зміною статусів він швидко стає незручним. При збільшенні кількості сторінок і сценаріїв взаємодії код ускладнюється, зростає ризик дублювання логіки та помилок при оновленні інтерфейсу.

Angular є потужним фреймворком для створення великих вебзастосунків. Він має розвинену структуру, власні засоби маршрутизації, роботи з формами та обробки даних. Водночас для даної дипломної роботи Angular є трохи надлишковим, тому що потребує складнішого налаштування, має вищий поріг входження та передбачає суворішу архітектуру проєкту. Для веб-платформи середнього рівня складності це може збільшити час розробки без практичної переваги.

Vue.js також є зручним інструментом для створення динамічних

інтерфейсів. Він простіший за Angular і добре підходить для поступового впровадження у вебпроєкти. Проте в межах даної роботи оптимально використати React, оскільки він має велику кількість готових компонентів, зручну роботу зі станом інтерфейсу та добре підходить для створення односторінкових вебзастосунків.

Для клієнтської частини веб-платформи було обрано React [5]. Його основною перевагою є компонентний підхід до побудови інтерфейсу. Кожен елемент системи може бути реалізований як окремий компонент: форма авторизації, список заявок, картка волонтерського завдання, панель координатора, кнопки зміни статусу, фільтри та елементи навігації. Це дозволяє уникнути дублювання коду та спростити подальше внесення змін. Наприклад, якщо потрібно змінити вигляд картки заявки, достатньо змінити один компонент, а не редагувати декілька сторінок окремо.

Ще однією важливою перевагою React є можливість швидкого оновлення інтерфейсу без повного перезавантаження сторінки. Для платформи координації волонтерської діяльності це важливо, оскільки користувач повинен бачити актуальні статуси заявок, зміни в списках завдань та результати призначення волонтерів без зайвих переходів між сторінками. Такий підхід робить взаємодію з системою більш зручною і наближеною до роботи повноцінного застосунку.

Для серверної частини було розглянуто декілька технологічних варіантів: PHP, Python із фреймворком Django або Flask, Java Spring Boot та Node.js із Express [6, 7]. PHP залишається поширеним засобом розробки вебсистем, але для даної роботи він менш зручний через потребу окремо організовувати сучасну структуру API, авторизацію та взаємодію з динамічним клієнтським застосунком. Django є потужним фреймворком і містить багато готових механізмів, але для невеликої веб-платформи він може бути занадто важким. Flask, навпаки, легший, однак потребує самостійного підключення багатьох компонентів.

Java Spring Boot є надійним рішенням для промислових серверних систем, але його використання в межах бакалаврської роботи може ускладнити розробку зза великої кількості конфігурацій, складнішу структуру проєкту та більший обсяг

допоміжного коду.

Для серверної частини було обрано Node.js у поєднанні з Express. Node.js дозволяє використовувати JavaScript на сервері, що спрощує узгодження клієнтської та серверної частин системи [6, 7]. Це зручно, оскільки одна мова використовується і для побудови інтерфейсу, і для обробки серверної логіки. Express забезпечує просту організацію маршрутів, обробку HTTP-запитів, підключення middleware та створення REST API.

Серверна частина веб-платформи повинна виконувати декілька основних задач: приймати запити від клієнтської частини, перевіряти права користувачів, працювати з базою даних, повертати структуровані відповіді та обробляти помилки. Для цього використання REST API є доцільним, оскільки такий підхід дозволяє чітко розділити клієнтську і серверну логіку. Наприклад, клієнтська частина надсилає запит на отримання списку заявок, а сервер повертає відповідь у форматі JSON. Аналогічно виконуються створення нової заявки, зміна її статусу, призначення волонтера або оновлення даних користувача [8-13].

Для передавання даних між клієнтом і сервером обрано формат JSON [13, 14]. Він є зручним для вебзастосунків, добре підтримується JavaScript і дозволяє передавати структуровану інформацію у компактному вигляді. Для даної системи у JSON-форматі можуть передаватися дані про користувачів, заявки, статуси, ролі, категорії допомоги та результати виконання завдань.

Окрему увагу необхідно приділити вибору бази даних. Для вебплатформи координації волонтерської діяльності дані мають чітку структуру. У системі передбачається зберігання користувачів, ролей, заявок, статусів, категорій допомоги та зв’язків між координаторами і волонтерами. Такі дані мають логічні зв’язки між собою, тому для їх зберігання треба використовувати реляційну базу даних.

Серед можливих варіантів було розглянуто MySQL, MariaDB та PostgreSQL. MySQL і MariaDB є поширеними рішеннями для веброзробки, мають просте налаштування та достатню продуктивність для більшості типових систем. Водночас PostgreSQL має більш широкі можливості роботи зі складними

запитами, обмеженнями цілісності, транзакціями та типами даних. Для системи, де важливо зберігати коректні зв’язки між користувачами, заявками та статусами, PostgreSQL є більш придатним рішенням.

Для реалізації бази даних вебплатформи було обрано PostgreSQL [8-10]. Вона дозволяє створити нормалізовану структуру даних, забезпечити цілісність записів та коректну роботу зі зв’язаними таблицями. Наприклад, заявка повинна бути пов’язана з користувачем, який її створив, координатором або волонтером, який її виконує, а також зі статусом і категорією допомоги. Реляційна модель дозволяє описати такі зв’язки явно та зменшити ризик появи неузгоджених даних.

Для взаємодії серверної частини з базою даних може використовуватися ORM-бібліотека, наприклад Sequelize або Prisma. ORM дозволяє працювати з таблицями бази даних через програмні моделі, що спрощує написання запитів і зменшує кількість помилок при роботі з SQL. В межах даного проекту доцільним є використання Prisma, оскільки вона має зручну схему опису моделей, підтримує міграції бази даних і добре інтегрується з Node.js [11-17].

Для організації авторизації користувачів передбачається використання JWT-токенів [15]. Після успішного входу користувача в систему сервер формує токен, у якому зберігається службова інформація про користувача та його роль. Надалі цей токен передається разом із запитами до сервера і використовується для перевірки прав доступу. Такий підхід дозволяє реалізувати розмежування доступу між адміністратором, координатором та волонтером.

Паролі користувачів не повинні зберігатися у відкритому вигляді [16]. Для цього на серверній стороні необхідно застосувати хешування паролів, наприклад за допомогою бібліотеки bcrypt. Це дозволяє зменшити ризик компрометації облікових записів у випадку несанкціонованого доступу до бази даних. Для веб-платформи, яка працює з персональними даними користувачів, такий механізм є обов’язковим елементом базової безпеки.

Для розробки програмного забезпечення було обрано середовище Visual Studio Code [18]. Воно підтримує роботу з JavaScript, React, Node.js, Express та PostgreSQL, має інтегрований термінал, засоби налагодження і велику кількість

розширень. Це дозволяє виконувати розробку клієнтської та серверної частин в межах одного середовища.

Порівняння основних технологій для реалізації веб-платформи наведено в таблиці 1.1.

Таблиця 1.1 – Порівнянні технологій реалізацій веб-платформ

Критерій	React + Node.js + Express	Angular + Spring Boot	Vue.js + Django
Складність розробки	середня	висока	середня
Швидкість створення прототипу	висока	середня	висока
Єдина мова для клієнта і сервера	так	ні	ні
Підтримка REST API	висока	висока	висока
Масштабованість	висока	висока	середня
Кількість допоміжного коду	помірна	висока	середня
Зручність підтримки	висока	середня	середня

З наведеного порівняння видно, що використання React, Node.js та Express є доцільним для даної роботи, оскільки такий стек забезпечує достатню швидкодію, просту організацію клієнт-серверної взаємодії та зручність розробки. При цьому система залишається придатною для подальшого розвитку: до неї можна додати нові ролі користувачів, модуль сповіщень, розширену аналітику заявок або інтеграцію з іншими сервісами.

Таким чином, для реалізації вебплатформи координації волонтерської діяльності обрано клієнтську частину на основі React, серверну частину на основі Node.js та Express, базу даних PostgreSQL, формат обміну даними JSON, механізм авторизації на основі JWT та середовище розробки Visual Studio Code. Обраний набір технологій дозволяє створити клієнт-серверну вебсистему, яка відповідає функціональним і нефункціональним вимогам, має зрозумілу архітектуру та може

бути розширена у майбутньому.

1.4 Вибір бази даних

Одним із важливих етапів розробки веб-платформи є вибір системи керування базами даних. Саме база даних забезпечує централізоване збереження інформації про користувачів, заявки, статуси виконання завдань та результати роботи системи [11, 19]. Від правильного вибору бази даних залежить швидкодія веб-платформи, стабільність роботи при одночасному доступі декількох користувачів та можливість подальшого розширення функціональності системи.

Для вебплатформи координації волонтерської діяльності передбачається постійна робота з великою кількістю взаємопов’язаних даних. У системі повинна зберігатися інформація про користувачів, їх ролі, створені заявки, категорії допомоги, статуси виконання завдань та призначення волонтерів. Між цими даними існують логічні зв’язки, тому структура сховища повинна забезпечувати підтримку реляційної моделі даних та коректну роботу із взаємозалежними таблицями.

На початковому етапі було розглянуто декілька популярних систем керування базами даних: MySQL, MariaDB, MongoDB та PostgreSQL.

MySQL є однією з найбільш поширених реляційних баз даних у сфері веброзробки. Вона має просте налаштування, підтримує SQL-запити та добре інтегрується з більшістю серверних технологій. Основною перевагою MySQL є висока швидкість виконання типових операцій та значна кількість готових засобів адміністрування. Проте під час роботи зі складними зв’язками між таблицями та великою кількістю одночасних операцій PostgreSQL забезпечує ширші можливості контролю цілісності даних.

MariaDB є розвитком MySQL та підтримує сумісність із більшістю її механізмів. Вона має хорошу продуктивність та активно використовується у

вебпроєктах різного рівня складності. Водночас функціонально MariaDB у межах даної роботи не надає суттєвих переваг порівняно з PostgreSQL.

Окремо було розглянуто MongoDB - документоорієнтовану нереляційну базу даних. Основною перевагою MongoDB є гнучка структура документів та можливість швидкої роботи з неструктурованими даними. Такий підхід добре підходить для систем, у яких структура інформації часто змінюється або не має жорстких зв'язків між об'єктами. Однак для веб-платформи координації волонтерської діяльності дані мають чітко визначену структуру та логічні залежності між користувачами, заявками й статусами. У такому випадку використання реляційної бази даних є більш доцільним.

Після аналізу доступних варіантів для реалізації системи було обрано PostgreSQL [8]. Дана система керування базами даних підтримує реляційну модель, забезпечує роботу зі складними SQL-запитами та дозволяє організувати надійне зберігання взаємопов'язаних даних.

Однією з основних переваг PostgreSQL є підтримка механізмів забезпечення цілісності даних. Наприклад, система дозволяє створювати зовнішні ключі між таблицями, що запобігає появі некоректних або неузгоджених записів. Для веб-платформи це особливо важливо, оскільки кожна заявка повинна бути пов'язана з користувачем, який її створив, категорією допомоги та поточним статусом виконання.

Ще однією перевагою PostgreSQL є підтримка транзакцій. Це дозволяє виконувати декілька взаємопов'язаних операцій як єдину дію. Наприклад, при зміні статусу заявки та одночасному призначенні волонтера система повинна або успішно виконати обидві операції, або скасувати зміни у випадку помилки. Використання транзакцій дозволяє уникнути появи частково змінених даних у базі [8, 19].

Важливе значення має також підтримка одночасної роботи декількох користувачів. Оскільки веб-платформа передбачає взаємодію координаторів, волонтерів та адміністратора, база даних повинна коректно обробляти паралельні запити без втрати або пошкодження інформації. PostgreSQL підтримує механізми

блокування та керування конкурентним доступом до даних, що забезпечує стабільну роботу системи при одночасному виконанні великої кількості операцій.

Для вебплатформи також важливо забезпечити можливість подальшого розширення структури бази даних. У майбутньому до системи можуть бути додані нові модулі, категорії заявок, система повідомлень або розширена аналітика. PostgreSQL дозволяє змінювати структуру таблиць та додавати нові зв’язки між об’єктами без необхідності повної переробки бази даних.

Під час проєктування структури бази даних передбачається створення декількох основних таблиць:

– таблиці користувачів;

– таблиці ролей;

– таблиці заявок;

– таблиці статусів виконання;

– таблиці категорій допомоги;

– таблиці призначення волонтерів на завдання.

Між таблицями будуть організовані зв’язки типу «один до багатьох» та «багато до одного». Наприклад, один користувач може створювати декілька заявок, а кожна заявка може мати лише одного відповідального волонтера на певному етапі виконання.

Для взаємодії серверної частини з PostgreSQL доцільно використовувати ORM-засоби. ORM дозволяє працювати з базою даних через програмні моделі без необхідності постійного написання SQL-запитів вручну. Це спрощує розробку серверної частини, робить код більш зрозумілим та зменшує ймовірність помилок під час роботи з даними.

Таким чином, використання PostgreSQL для реалізації веб-платформи координації волонтерської діяльності є доцільним завдяки підтримці реляційної моделі даних, механізмів забезпечення цілісності інформації, транзакцій та одночасної роботи великої кількості користувачів. Обрана система керування базами даних дозволяє забезпечити стабільну роботу веб-платформи та створює основу для подальшого розвитку програмного забезпечення.

1.5 Висновки по розділу 1

У першому розділі дипломної роботи було проведено аналіз особливостей побудови клієнт-серверних систем та визначено основні вимоги до вебплатформи координації волонтерської діяльності. У результаті аналізу предметної області встановлено, що використання окремих месенджерів, соціальних мереж та електронних таблиць ускладнює централізоване керування заявками, контроль виконання завдань та взаємодію між координаторами і волонтерами.

Під час формування функціональних вимог визначено основні можливості системи, серед яких реєстрація та авторизація користувачів, створення та обробка заявок, розподіл завдань між волонтерами, зміна статусів виконання та адміністрування вебплатформи. Також були визначені нефункціональні вимоги до системи, пов’язані зі швидкодією, стабільністю роботи, безпекою, масштабованістю та зручністю використання веб-платформи.

У ході аналізу технологій реалізації системи було розглянуто різні підходи до побудови клієнтської та серверної частин вебзастосунку. Для реалізації клієнтської частини обрано бібліотеку React, що дозволяє створювати динамічний користувацький інтерфейс та забезпечує компонентний підхід до побудови системи. Для серверної частини обрано Node.js та Express, які забезпечують зручну організацію REST API та взаємодію між клієнтом і сервером.

Під час вибору системи керування базами даних було проаналізовано особливості реляційних та нереляційних підходів до зберігання інформації. Для реалізації веб-платформи обрано PostgreSQL, оскільки дана система забезпечує підтримку реляційної моделі даних, механізмів цілісності, транзакцій та одночасної роботи декількох користувачів.

Таким чином, у результаті виконаного аналізу було визначено основні технології та програмні засоби, які доцільно використовувати для реалізації вебплатформи координації волонтерської діяльності. Отримані результати створюють основу для подальшого проєктування архітектури системи (рис. 1.1),

структури бази даних та реалізації програмного забезпечення.

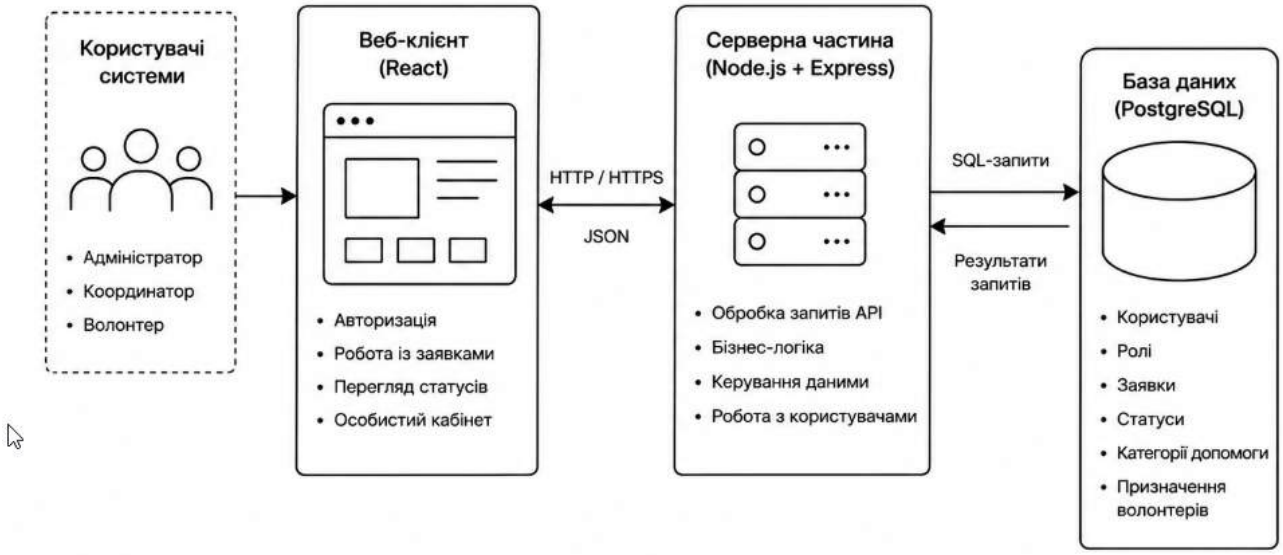

Рисунок 1.1 - Концептуальна архітектура клієнт-серверної системи вебплатформи координації волонтерської діяльності

2 РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ СИСТЕМИ

2.1 Проєктування серверної архітектури

Серверна частина веб-платформи координації волонтерської діяльності проєктується як окремий застосунок, який відповідає за обробку запитів клієнтської частини, роботу з базою даних, авторизацію користувачів та виконання основної бізнес-логіки системи [10, 11]. Такий підхід дозволяє відокремити інтерфейс користувача від логіки обробки даних і зробити систему більш керованою під час подальшого супроводу [6, 7].

Серверна частина системи реалізується на основі Node.js та Express [6, 7, 12]. Node.js забезпечує виконання серверного JavaScript-коду, а Express використовується для побудови REST API, через яке клієнтська частина

взаємодіє із сервером [12]. Клієнт надсилає HTTP-запити до сервера, сервер обробляє ці запити, звертається до бази даних PostgreSQL та повертає відповідь у форматі JSON [12, 18].

Серверний проєкт веб-платформи має модульну структуру [5, 11]. Це означає, що окремі частини системи відповідають за різні задачі: маршрутизацію запитів, авторизацію, роботу з користувачами, роботу із заявками, перевірку прав доступу та взаємодію з базою даних. Такий підхід спрощує розробку, тому що зміни в одному модулі не потребують повного переписування всієї серверної частини [5, 11].

Структура серверного проєкту веб-платформи наведена на рисунку 2.1.

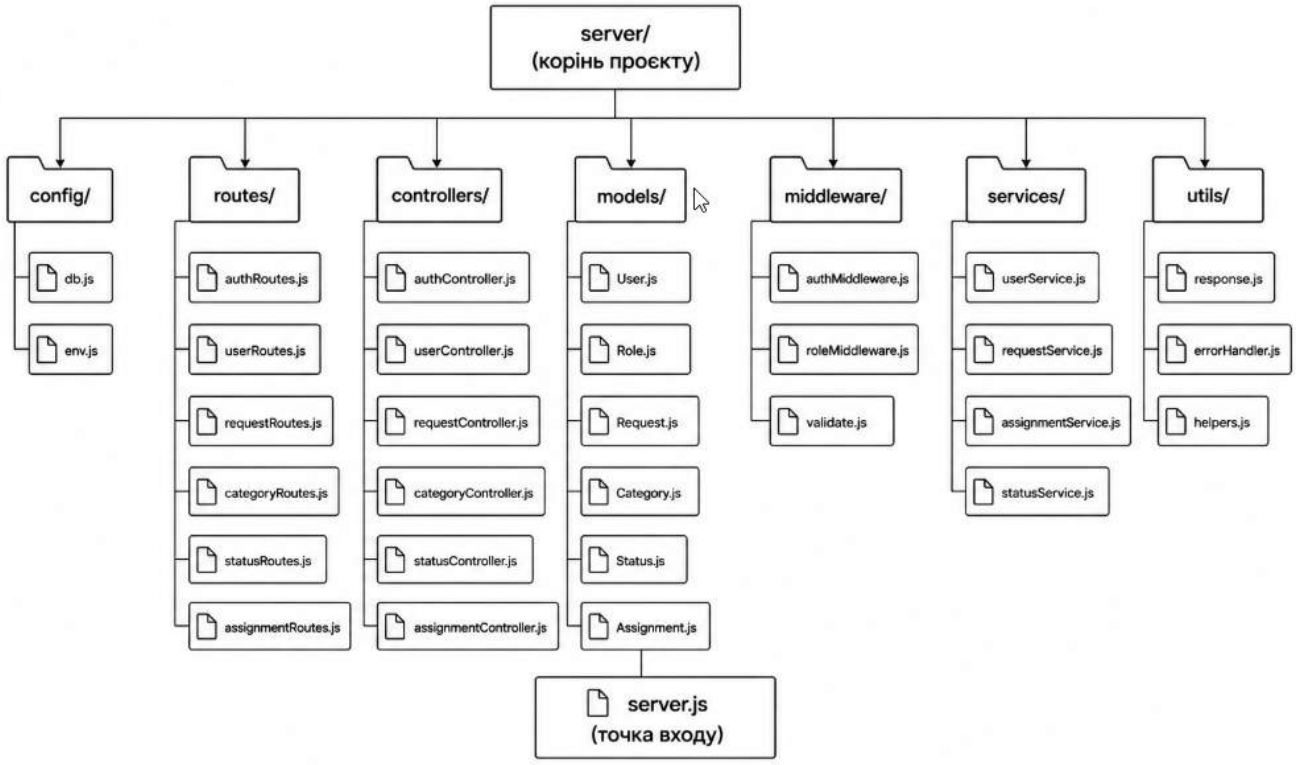

Рисунок 2.1 – Стуктура серверного проєкту веб-платформи координації волонтерської діяльності

До складу серверної частини входять такі основні модулі:

– server.js — точка входу у серверний застосунок та ініціалізація роботи сервера;

– routes — модуль маршрутизації запитів клієнтської частини;

– controllers — модуль обробки HTTP-запитів та формування відповідей;

– services — модуль реалізації прикладної логіки роботи веб-платформи;

– middleware — проміжні функції перевірки доступу та обробки запитів;

– config — модуль конфігурації серверного середовища;

– db.js — модуль встановлення з’єднання із базою даних через Prisma ORM.

Під час запуску застосунку виконується ініціалізація сервера, підключення маршрутів та налаштування механізму взаємодії із базою даних. Після отримання HTTP-запиту сервер визначає відповідний маршрут, передає керування контролеру та виконує необхідну обробку даних із використанням сервісного рівня.

Файл server.js виконує роль початкової точки запуску серверної частини. У ньому ініціалізується Express-застосунок, підключаються основні модулі, задається порт сервера та виконується запуск прослуховування вхідних запитів. Також у цьому файлі підключаються middleware для обробки JSON-даних, налаштування CORS та маршрути системи.

Модуль routes відповідає за маршрутизацію запитів. У ньому визначається, які URL-адреси використовуються для роботи з авторизацією, користувачами, заявками та статусами. Наприклад, запит на створення нової заявки передається до відповідного контролера заявок, а запит на авторизацію користувача — до контролера автентифікації.

Модуль controllers містить основну логіку обробки запитів. Контролери отримують дані від клієнтської частини, перевіряють їх коректність, викликають потрібні службові функції та формують відповідь для клієнта. Наприклад, requestController відповідає за створення, редагування, перегляд і зміну статусу волонтерських заявок.

Окреме значення має модуль middleware. Він використовується для перевірки JWT-токена, визначення ролі користувача та обмеження доступу до окремих функцій системи. Наприклад, звичайний волонтер не повинен мати доступу до адміністративного керування користувачами, а координатор повинен

мати можливість працювати із заявками, але не змінювати системні налаштування.

Модуль services містить службову логіку, яка не повинна напряму розміщуватися у контролерах. Це дозволяє не перевантажувати контролери зайвим кодом. Наприклад, у сервісному модулі може виконуватися перевірка можливості призначення волонтера на заявку, зміна статусу завдання або формування списку активних заявок для координатора.

Модуль config використовується для збереження налаштувань підключення до бази даних PostgreSQL, параметрів сервера та службових змінних середовища. Винесення таких параметрів в окремий модуль спрощує налаштування системи при розгортанні на іншому комп’ютері або сервері.

Серверна частина взаємодіє з базою даних через окремий рівень доступу до даних. Це дозволяє відокремити SQL-запити або ORM-моделі від контролерів і зменшити залежність між окремими частинами програми. У результаті серверний код стає більш зрозумілим і придатним до подальшого розширення.

Основна логіка роботи сервера полягає в тому, що клієнтська частина надсилає запит до певного маршруту API. Після цього сервер перевіряє права користувача, обробляє отримані дані, звертається до бази даних і повертає результат у форматі JSON. Такий формат є зручним для веб-платформи, оскільки легко обробляється клієнтською частиною на React.

Наприклад, при створенні нової заявки координатор заповнює форму у вебінтерфейсі. Клієнтська частина передає ці дані на сервер, сервер перевіряє токен користувача, визначає його роль, перевіряє коректність заповнених полів і після цього створює новий запис у базі даних. У відповідь клієнт отримує повідомлення про успішне створення заявки або інформацію про помилку.

Така архітектура дозволяє забезпечити розмежування відповідальностей між частинами системи. Клієнтська частина відповідає за відображення інформації та взаємодію з користувачем, серверна частина — за обробку даних, перевірку доступу та взаємодію з базою даних. Це відповідає загальним принципам побудови клієнт-серверних комп’ютерних систем.

Перевагою запропонованої серверної архітектури є можливість подальшого розвитку системи. У майбутньому до серверної частини можна додати модуль повідомлень, журнал подій, аналітику виконання заявок або інтеграцію з зовнішніми сервісами. При цьому базова структура проєкту не потребуватиме повної переробки, оскільки нові можливості можуть бути реалізовані як окремі модулі.

Таким чином, серверна частина веб-платформи координації волонтерської діяльності проєктується як модульний Node.js-застосунок із використанням Express, REST API, JWT-авторизації та бази даних PostgreSQL. Обрана структура забезпечує чітке розділення функцій, спрощує супровід програмного коду та створює основу для подальшого розширення системи.

2.2 Апаратні вимоги до сервера

Для стабільної роботи вебплатформи координації волонтерської діяльності необхідно визначити мінімальні апаратні вимоги до серверної частини системи. Саме сервер виконує обробку запитів користувачів, взаємодіє з базою даних, забезпечує роботу системи авторизації та передає дані клієнтській частині вебзастосунку [2, 3].

Під час визначення апаратних вимог враховувалось, що система орієнтована на використання невеликими або середніми волонтерськими організаціями, де кількість одночасно активних користувачів не є надто великою. Тому для роботи вебплатформи не потребується дорогого високопродуктивного серверного обладнання. Водночас сервер повинен забезпечувати стабільну роботу бази даних, коректну обробку HTTP-запитів та підтримувати одночасне підключення декількох користувачів [6, 8].

Основне навантаження на сервер створюють:

– обробка запитів клієнтської частини;

– взаємодія з базою даних PostgreSQL;

– перевірка JWT-токенів авторизації;

– робота із заявками та користувачами;

– формування відповідей у форматі JSON [14, 15].

Для розгортання серверної частини вебплатформи достатньо персонального комп’ютера або сервера середнього рівня продуктивності. Мінімальні апаратні вимоги до сервера наведені у таблиці 2.1.

Таблиця 2.1 – Мінімальні апаратні вимоги до сервера

Компонент	Мінімальні характеристики
процесор	Intel Core i3 / AMD Ryzen 3
Оперативна пам’ять	4 Гб
Рекомендований обсяг ОЗП	8 Гб
Накопичувач	SSD від 120 ГБ
Мережева карта	Ethernet 100 Мбіт/с
Операційна система	Linux Ubuntu Server

Використання SSD-накопичувача дозволяє пришвидшити роботу бази даних та зменшити час доступу до інформації. Під час роботи вебплатформи сервер постійно виконує операції читання та запису даних, тому швидкість роботи накопичувача безпосередньо впливає на швидкодію системи.

Оперативна пам’ять використовується для роботи Node.js-сервера, PostgreSQL та кешування службових даних [6, 8]. Для невеликої кількості користувачів достатньо 4 ГБ оперативної пам’яті, однак для стабільнішої роботи системи доцільно використовувати 8 ГБ або більше.

Процесор сервера повинен забезпечувати обробку декількох одночасних запитів до вебплатформи. Оскільки система не виконує ресурсоємних обчислень або обробки мультимедійних даних, використання дорогих серверних процесорів не є необхідним. Для роботи системи достатньо сучасного двоядерного або чотириядерного процесора середнього рівня продуктивності.

Для серверної частини системи доцільно використовувати операційну систему Linux Ubuntu Server [2]. Дана операційна система забезпечує стабільну роботу серверних застосунків, підтримує Node.js, PostgreSQL та необхідні засоби адміністрування. Крім того, Linux дозволяє ефективно використовувати апаратні ресурси та спрощує розгортання вебзастосунків.

Важливе значення для стабільної роботи системи має мережеве підключення сервера. Оскільки взаємодія між клієнтською та серверною частинами виконується через мережу Інтернет, нестабільне з’єднання може призводити до збільшення часу відповіді або тимчасової недоступності веб-платформи.

Під час подальшого розвитку системи або збільшення кількості користувачів серверна частина повинна підтримувати можливість масштабування [11]. Це може бути реалізовано шляхом збільшення обсягу оперативної пам’яті, використання продуктивнішого процесора або перенесення веб-платформи на хмарний сервер.

Таким чином, для роботи веб-платформи координації волонтерської діяльності достатньо серверного обладнання середнього рівня продуктивності. Визначені апаратні вимоги забезпечують стабільну роботу серверної частини, підтримку взаємодії між користувачами та коректне функціонування бази даних системи.

2.3 Проєктування бази даних

База даних вебплатформи координації волонтерської діяльності (рис.2.2) призначена для централізованого зберігання інформації про користувачів, волонтерські заявки, категорії допомоги, статуси виконання, призначення волонтерів, повідомлення та історію змін у системі [11, 19]. Модель даних побудована відповідно до реляційного підходу з дотриманням принципів

нормалізації, що дозволяє уникнути дублювання інформації, забезпечити цілісність даних і спростити подальше розширення функціональності системи [11, 19].

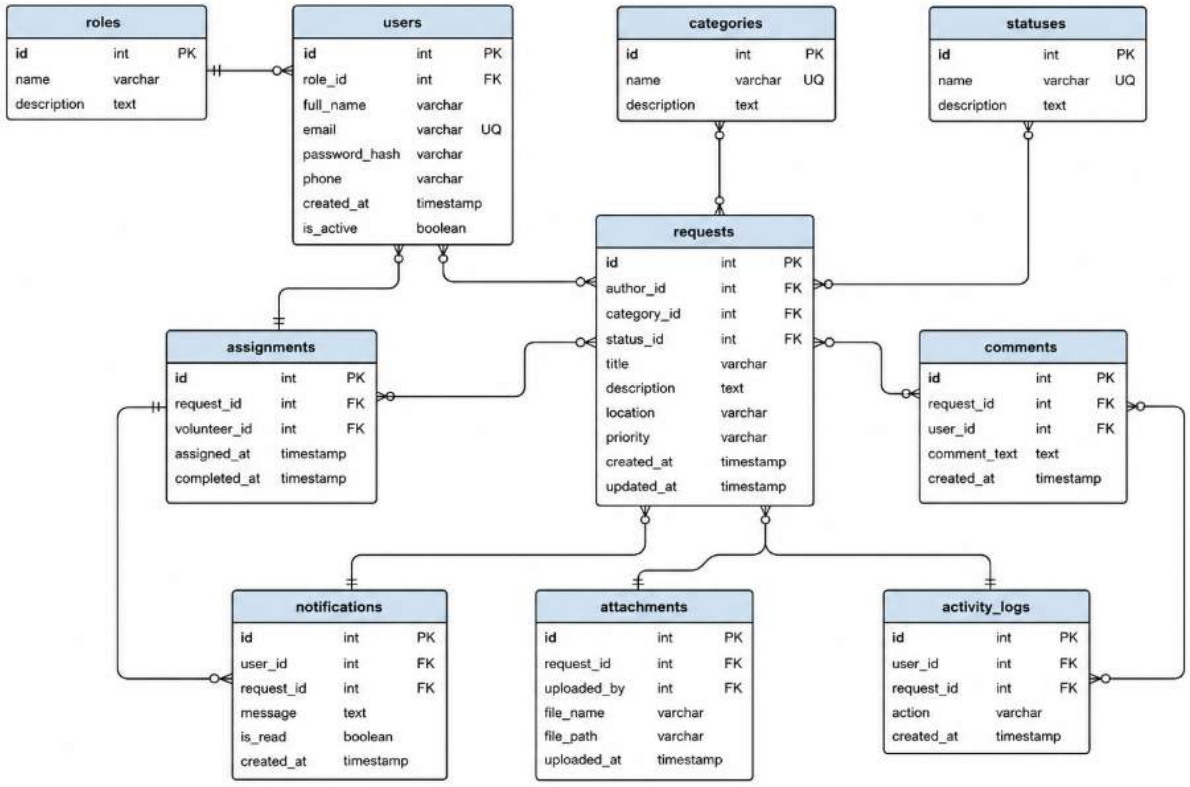

Рисунок 2.2 – Схема бази даних вебплатформи координації волонтерської діяльності

У базі даних зберігається інформація про основних учасників системи: адміністраторів, координаторів і волонтерів. Окремо виділено сутності для опису ролей користувачів, волонтерських заявок, категорій допомоги та статусів виконання. Для фіксації призначення волонтерів на конкретні заявки використовується окрема зв’язкова сутність. Також у структурі бази даних передбачені сутності для зберігання коментарів, повідомлень, вкладень до заявок та журналу змін. Такий набір сутностей дозволяє реалізувати основний функціонал вебплатформи і водночас залишає можливість подальшого розвитку системи без повної зміни структури бази даних.

Сутність Roles (лістинг 2.1) призначена для зберігання ролей користувачів

системи. В межах веб-платформи передбачено три основні ролі: адміністратор, координатор і волонтер. Виділення ролей в окрему сутність дозволяє реалізувати розмежування прав доступу та не дублювати текстові значення ролей у таблиці користувачів.

Лістинг 2.1 – DBML-опис сутності Roles

```
Table roles {
id int [pk, increment]
name varchar [not null, unique]
description text
}
```

Сутність Users (лістинг 2.2) призначена для зберігання інформації про користувачів веб-платформи. Вона містить ім’я користувача, адресу електронної пошти, пароль у захищеному вигляді, номер телефону та посилання на роль користувача. Саме ця сутність використовується під час авторизації, перевірки прав доступу та відображення інформації про учасників волонтерської діяльності.

Лістинг 2.2 – DBML-опис сутності Users

```
Table users {
id int [pk, increment]
role_id int [not null, ref: > roles.id]
full_name varchar [not null]
email varchar [not null, unique]
password_hash varchar [not null]
phone varchar
created_at timestamp
is_active boolean
}
```

Сутність Categories (лістинг 2.3) використовується для зберігання категорій волонтерської допомоги. До таких категорій можуть належати гуманітарна допомога, транспорт, медична підтримка, евакуація, побутові потреби або організаційні питання. Окрема сутність категорій дозволяє спростити пошук і фільтрацію заявок у системі.

Лістинг 2.3 – DBML-опис сутності Categories

```
Table categories {
id int [pk, increment]
name varchar [not null, unique]
description text
}
```

Сутність Statuses (лістинг 2.4) призначена для зберігання можливих статусів виконання заявок. Наприклад, заявка може перебувати у статусі «нова», «в роботі», «виконана» або «скасована». Винесення статусів в окрему сутність дає змогу централізовано керувати станами заявок і уникнути помилок при їх повторному використанні.

Лістинг 2.4 – DBML-опис сутності Statuses

```
Table statuses {
id int [pk, increment]
name varchar [not null, unique]
description text
}
```

Сутність Requests (лістинг 2.5) є центральною сутністю бази даних, оскільки саме вона описує волонтерські заявки. У ній зберігається назва заявки, детальний опис потреби, адреса або місце виконання, дата створення, пріоритет, автор заявки, категорія допомоги та поточний статус. На основі цієї сутності формується основна частина функціоналу веб-платформи.

Лістинг 2.5 – DBML-опис сутності Requests

```
Table requests {
id int [pk, increment]
author_id int [not null, ref: > users.id]
category_id int [not null, ref: > categories.id]
status_id int [not null, ref: > statuses.id]
title varchar [not null]
description text [not null]
location varchar
priority varchar
created_at timestamp
updated_at timestamp
}
```

Сутність Assignments (лістинг 2.6) реалізує зв’язок між волонтерами та заявками. Вона дозволяє фіксувати, який саме волонтер призначений на виконання конкретної заявки. Такий підхід є зручним, оскільки одна заявка може потребувати участі декількох волонтерів, а один волонтер може виконувати різні заявки у різний час.

Лістинг 2.6 – DBML-опис сутності Assignments

```
Table assignments {
id int [pk, increment]
request_id int [not null, ref: > requests.id]
volunteer_id int [not null, ref: > users.id]
assigned_at timestamp
completed_at timestamp
}
```

Сутність Comments (лістинг 2.7) призначена для зберігання коментарів до заявок. Коментарі можуть використовуватися координаторами та волонтерами для уточнення деталей виконання завдання, передачі проміжної інформації або фіксації результату роботи. Винесення коментарів в окрему сутність дозволяє не перевантажувати основну таблицю заявок.

Лістинг 2.7 – DBML-опис сутності Comments

```
Table comments {
id int [pk, increment]
request_id int [not null, ref: > requests.id]
user_id int [not null, ref: > users.id]
comment_text text [not null]
created_at timestamp
}
```

Сутність Notifications (лістинг 2.8) використовується для зберігання системних повідомлень користувачів. Повідомлення можуть формуватися при створенні нової заявки, зміні її статусу, призначенні волонтера або додаванні коментаря. Така сутність дозволяє реалізувати інформування користувачів про важливі зміни в системі.

Лістинг 2.8 – DBML-опис сутності Notifications

```
Table notifications {
id int [pk, increment]
user_id int [not null, ref: > users.id]
request_id int [ref: > requests.id]
message text [not null]
is_read boolean
created_at timestamp
}
```

Сутність Attachments (лістинг 2.9) призначена для зберігання інформації про файли, пов’язані із заявками. Це можуть бути фотографії, документи або інші матеріали, які уточнюють зміст заявки. У самій базі даних доцільно зберігати не файл повністю, а шлях до нього або посилання на місце зберігання.

Лістинг 2.9 – DBML-опис сутності Attachments

```
Table attachments {
id int [pk, increment]
request_id int [not null, ref: > requests.id]
uploaded_by int [not null, ref: > users.id]
file_name varchar [not null]
file_path varchar [not null]
uploaded_at timestamp
}
```

Сутність ActivityLogs (лістинг 2.10) використовується для зберігання історії дій користувачів у системі. Вона дозволяє фіксувати створення заявки, зміну статусу, призначення волонтера, додавання коментаря або інші важливі події. Наявність журналу дій підвищує прозорість роботи системи та спрощує контроль за змінами.

Лістинг 2.10 – DBML-опис сутності ActivityLogs

```
Table activity_logs {
id int [pk, increment]
user_id int [not null, ref: > users.id]
request_id int [ref: > requests.id]
action varchar [not null]
created_at timestamp
}
```

Основним зв’язком у базі даних є зв’язок між користувачами та заявками. Один користувач може створити декілька заявок, але кожна заявка має одного автора. Аналогічно одна категорія може бути пов’язана з багатьма заявками, проте кожна заявка належить лише до однієї категорії. Статус також використовується багатьма заявками, але для кожної конкретної заявки зберігається тільки один поточний статус.

Зв’язок між волонтерами та заявками реалізовано через сутність Assignments. Вона фактично виконує роль проміжної таблиці між Users і Requests. Завдяки цьому система може зберігати не лише сам факт призначення волонтера, але й дату призначення та дату завершення виконання завдання.

Коментарі, повідомлення, вкладення та журнал дій пов’язані із заявками через зовнішні ключі [11, 19]. Це дозволяє зберігати всю додаткову інформацію окремо, але при цьому швидко отримувати її для конкретної заявки. Така структура не перевантажує основну сутність Requests і робить модель даних більш гнучкою.

Таким чином, спроєктована база даних забезпечує зберігання всіх основних даних, необхідних для роботи вебплатформи координації волонтерської діяльності. Запропонована структура дозволяє реалізувати авторизацію користувачів, розмежування ролей, роботу із заявками, призначення волонтерів, коментування, повідомлення та контроль історії змін у системі.

2.4 Реалізація серверних модулів

Серверна частина веб-платформи координації волонтерської діяльності реалізується як окремий застосунок на платформі Node.js із використанням фреймворку Express [6, 7]. Вона відповідає за приймання запитів від клієнтської частини, перевірку користувачів, роботу з базою даних PostgreSQL, обробку

заявок та формування відповідей у форматі JSON [8, 14].

Загальна логіка роботи серверної частини побудована за класичною схемою REST API [13]. Клієнтська частина надсилає HTTP-запит до певного маршруту, після чого сервер перевіряє авторизацію, передає дані до контролера, виконує необхідну бізнес-логіку в сервісному модулі та звертається до бази даних через Prisma ORM [17]. Після виконання операції сервер повертає клієнту відповідь із відповідним HTTP-статусом.

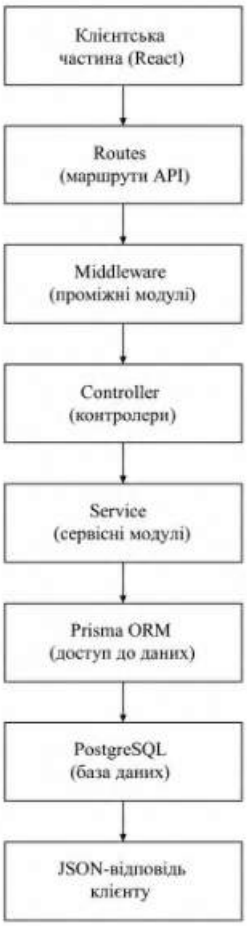

Рисунок 2.3 – Схема проходження запиту через серверні модулі

У серверному проєкті основна логіка розподілена між кількома групами модулів [10]. Каталог routes містить маршрути API, каталог controllers відповідає за обробку запитів, каталог services містить службову логіку, middleware використовується для перевірки авторизації та ролей, а config містить налаштування підключення до бази даних. Такий поділ дозволяє не змішувати

всю логіку в одному файлі та полегшує подальше внесення змін [10].

Файл server.js є точкою входу в серверний застосунок. У ньому створюється Express-застосунок, підключаються маршрути, middleware для обробки JSON-даних та глобальний обробник помилок [7].

Лістинг 2.11 – Код точки входу в серверний застосунок

```
const express = require('express');
const cors = require('cors');
const authRoutes = require('./src/routes/authRoutes');
const userRoutes = require('./src/routes/userRoutes');
const requestRoutes = require('./src/routes/requestRoutes');
const categoryRoutes = require('./src/routes/categoryRoutes');
const statusRoutes = require('./src/routes/statusRoutes');
const errorHandler = require('./src/utils/errorHandler');
const app = express();
app.use(cors());
app.use(express.json());
app.use('/api/auth', authRoutes);
app.use('/api/users', userRoutes);
app.use('/api/requests', requestRoutes);
app.use('/api/categories', categoryRoutes);
app.use('/api/statuses', statusRoutes);
app.use(errorHandler);
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
console.log(`Server started on port ${PORT}`);
});
```

Підключення до бази даних винесено в окремий файл db.js. Це потрібно для того, щоб усі модулі серверної частини використовували один спільний механізм доступу до PostgreSQL. Для роботи з базою даних використовується Prisma ORM, яка дозволяє виконувати операції з таблицями через програмні моделі.

Лістинг 2.12 – Підключення Prisma ORM до бази даних

```
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();
module.exports = prisma;
```

Модуль авторизації реалізує перевірку електронної пошти та пароля користувача [16]. Пароль не зберігається у відкритому вигляді, тому під час входу в систему введене значення порівнюється з хешем, який зберігається в базі даних.

Якщо перевірка проходить успішно, сервер формує JWT-токен [15].

Лістинг 2.13 – Фрагмент контролера авторизації

```
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const prisma = require('../config/db');
async function login(req, res, next) {
try {
const { email, password } = req.body;
const user = await prisma.users.findUnique({
where: { email },
include: { role: true }
});
if (!user) {
return res.status(401).json({
message: 'Невірна електронна пошта або пароль'
});
}
const validPassword = await bcrypt.compare(
password,
user.password_hash
);
if (!validPassword) {
return res.status(401).json({
message: 'Невірна електронна пошта або пароль'
});
}
const token = jwt.sign(
{
id: user.id,
role: user.role.name
},
process.env.JWT_SECRET,
{ expiresIn: '24h' }
);
res.json({
token,
user: {
id: user.id,
fullName: user.full_name,
email: user.email,
role: user.role.name
}
});
} catch (error) {
next(error);
}
}
module.exports = { login };
```

Після авторизації клієнтська частина передає JWT-токен у заголовку кожного захищеного запиту [15]. Для перевірки токена використовується middleware authMiddleware.js. Якщо токен відсутній або неправильний, сервер не дозволяє виконати запит [15].

REST API у серверній частині використовується як основний механізм взаємодії між клієнтською та серверною частинами. GET-запити застосовуються для отримання даних із сервера, POST-запити - для створення нових записів, PATCH-запити - для часткового оновлення інформації, а DELETE-запити можуть використовуватися для видалення або деактивації записів. Такий підхід дозволяє чітко розділити операції над даними та зробити взаємодію між компонентами системи зрозумілою.

JWT використовується для підтвердження авторизованого доступу користувача до захищених маршрутів API. Після входу в систему сервер формує токен, у якому зберігається ідентифікатор користувача та його роль. Надалі клієнтська частина передає цей токен у заголовку запиту. Сервер перевіряє токен перед виконанням захищених операцій.

Prisma ORM використовується як проміжний рівень між серверним кодом і базою даних PostgreSQL [17]. Завдяки цьому серверні модулі працюють із даними через програмні моделі, а не через ручне написання SQL-запитів у кожному контролері. Це спрощує структуру коду та зменшує кількість помилок під час роботи з базою даних.

Перевірка вхідних даних виконується перед передаванням запиту до основної логіки системи. Наприклад, під час створення заявки перевіряється наявність назви, опису, категорії та статусу. Якщо обов’язкові поля не заповнені, сервер повертає помилку з кодом 400 і не створює некоректний запис у базі даних.

Лістинг 2.14 – Middleware перевірки JWT-токена

```
 const jwt = require('jsonwebtoken');
function authMiddleware(req, res, next) {
const authHeader = req.headers.authorization;
```

```
if (!authHeader) {
return res.status(401).json({
message: 'Токен авторизації відсутній'
});
}
const token = authHeader.split(' ')[1];
try {
const decoded = jwt.verify(token, process.env.JWT_SECRET);
req.user = decoded;
next();
} catch (error) {
return res.status(401).json({
message: 'Недійсний токен авторизації'
});
}
}
module.exports = authMiddleware;
```

Для обмеження доступу до окремих функцій використовується middleware перевірки ролей. Наприклад, створення заявок і призначення волонтерів доступні координатору та адміністратору, а звичайний волонтер не повинен мати доступу до адміністративних операцій.

Лістинг 2.15 – Middleware перевірки ролі користувача

```
function roleMiddleware(allowedRoles) {
return function (req, res, next) {
if (!req.user || !allowedRoles.includes(req.user.role)) {
return res.status(403).json({
message: 'Недостатньо прав для виконання операції'
});
}
next();
};
}
module.exports = roleMiddleware;
```

Для роботи із заявками використовується окремий контролер `requestController.js`. Він приймає запити від клієнтської частини та передає їх у сервісний модуль. Контролер не виконує складну логіку самостійно, а лише координує отримання даних, виклик потрібної функції та повернення результату клієнту.

Лістинг 2.16 – Фрагмент контролера роботи із заявками

```
const requestService = require('../services/requestService');
async function getAllRequests(req, res, next) {
try {
const requests = await requestService.getAllRequests();
res.json(requests);
} catch (error) {
next(error);
}
}
async function createRequest(req, res, next) {
try {
const request = await requestService.createRequest(
req.body,
req.user.id
);
res.status(201).json(request);
} catch (error) {
next(error);
}
}
async function updateRequestStatus(req, res, next) {
try {
const { id } = req.params;
const { statusId } = req.body;
const request = await requestService.updateStatus(id, statusId);
res.json(request);
} catch (error) {
next(error);
}
}
module.exports = {
getAllRequests,
createRequest,
updateRequestStatus
};
```

Основна логіка роботи із заявками винесена до сервісу `requestService.js`. У цьому модулі виконується вибірка заявок із пов’язаними сутностями, створення нових записів та оновлення статусів. Такий підхід робить контролери коротшими, а бізнес-логіку — більш зрозумілою.

Лістинг 2.17 – Фрагмент сервісу роботи із заявками

```
const prisma = require('../config/db');
async function getAllRequests() {
return prisma.requests.findMany({
include: {
```

```
author: true,
category: true,
status: true,
assignments: {
include: {
volunteer: true
}}},
orderBy: {
created_at: 'desc'
}});
}
async function createRequest(data, authorId) {
return prisma.requests.create({
data: {
title: data.title,
description: data.description,
location: data.location,
priority: data.priority,
author_id: authorId,
category_id: Number(data.categoryId),
status_id: Number(data.statusId),
created_at: new Date()
}});
}
async function updateStatus(requestId, statusId) {
return prisma.requests.update({
where: {
id: Number(requestId)
},
data: {
status_id: Number(statusId),
updated_at: new Date()
}});
}
module.exports = {
getAllRequests,
createRequest,
updateStatus
};
```

Окремо реалізовано сервіс призначення волонтерів на заявки. Призначення не зберігається безпосередньо в таблиці заявок, а фіксується через сутність `assignments`. Це дозволяє зберігати історію призначень і не перевантажувати основну таблицю заявок.

Лістинг 2.18 – Фрагмент сервісу призначення волонтера

```
const prisma = require('../config/db');
async function assignVolunteer(requestId, volunteerId) {
```

```
return prisma.assignments.create({
data: {
request_id: Number(requestId),
volunteer_id: Number(volunteerId),
assigned_at: new Date()
}});
}
async function completeAssignment(assignmentId) {
return prisma.assignments.update({
where: {
id: Number(assignmentId)
},
data: {
completed_at: new Date()
}});
}
module.exports = {
assignVolunteer,
completeAssignment
};
```

Маршрути для роботи із заявками зберігаються у файлі `requestRoutes.js`. У ньому визначаються HTTP-методи, адреси API та middleware, які повинні бути виконані перед передаванням запиту до контролера. GET-запит використовується для отримання даних, POST — для створення нової заявки, PATCH — для часткового оновлення запису.

Лістинг 2.19 – Маршрути для роботи із заявками

```
const express = require('express');
const router = express.Router();
const authMiddleware = require('../middleware/authMiddleware');
const roleMiddleware = require('../middleware/roleMiddleware');
const requestController =
require('../controllers/requestController');
router.get(
'/',
authMiddleware,
requestController.getAllRequests
);
router.post(
'/',
authMiddleware,
roleMiddleware(['admin', 'coordinator']),
requestController.createRequest
);
router.patch(
'/:id/status',
```

```
authMiddleware,
roleMiddleware(['admin', 'coordinator']),
requestController.updateRequestStatus
);
module.exports = router;
```

Для перевірки вхідних даних використовується окремий middleware [15]. Наприклад, під час створення заявки необхідно перевірити наявність назви, опису, категорії та статусу. Це дозволяє не передавати до бази даних неповні або некоректні значення.

Лістинг 2.20 – Middleware перевірки даних заявки

```
function validateRequest(req, res, next) {
const { title, description, categoryId, statusId } = req.body;
if (!title || !description || !categoryId || !statusId) {
return res.status(400).json({
message: 'Не заповнені обов’язкові поля заявки'
});
}
next();
}
module.exports = validateRequest;
```

У серверній частині також передбачено централізовану обробку помилок. Якщо помилка виникає в контролері або сервісному модулі, вона передається до `errorHandler.js`. Це дозволяє не дублювати однаковий код обробки помилок у кожному контролері.

Лістинг 2.21 – Глобальний обробник помилок

```
function errorHandler(error, req, res, next) {
console.error(error);
res.status(500).json({
message: 'Внутрішня помилка сервера',
details: error.message
});
}
module.exports = errorHandler;
```

Для клієнтської частини важливо, щоб сервер повертав не тільки дані, але й зрозумілі HTTP-статуси. Наприклад, код 200 використовується при успішному

отриманні даних, 201 - при створенні нового запису, 400 - при помилці у вхідних даних, 401 - при відсутності авторизації, 403 - при недостатніх правах доступу, а 500 - при внутрішній помилці сервера. Завдяки цьому клієнтська частина може правильно реагувати на результат виконання запиту.

Реалізована серверна частина не відповідає за відображення сторінок користувачу. Її задача полягає у формуванні структурованих даних для клієнтського застосунку. Наприклад, при запиті списку заявок сервер повертає масив JSON-об’єктів, у яких міститься назва заявки, опис, категорія, статус, автор і призначені волонтери. Клієнтська частина на React уже самостійно відображає ці дані в інтерфейсі.

Таким чином, серверні модулі веб-платформи забезпечують авторизацію користувачів, перевірку ролей, роботу із заявками, призначення волонтерів, взаємодію з базою даних PostgreSQL та повернення результатів у форматі JSON. Модульна структура серверної частини спрощує супровід програмного забезпечення і дозволяє надалі розширювати систему без повної зміни її архітектури.

Після ініціалізації серверної частини та підключення основних модулів реалізується механізм обробки запитів клієнтської частини. Для взаємодії між компонентами системи використовується REST API, що дозволяє виконувати передачу структурованих даних між клієнтом і сервером незалежно від внутрішньої реалізації окремих модулів.

Для кожної функціональної області веб-платформи передбачено окремий набір маршрутів. Такий підхід дозволяє розділити логіку обробки різних типів даних та спростити супровід програмного забезпечення.

Основні маршрути серверної частини наведено у таблиці 2.2.

Після отримання HTTP-запиту сервер виконує перевірку коректності отриманих параметрів та визначає доступність відповідної операції для поточного користувача. У випадку успішного проходження перевірок виконується звернення до сервісного рівня та формування відповіді у форматі JSON.

Таблиця 2.2 – Основні маршрути REST API серверної частини

Маршрут	Призначення
/auth	Виконує операції реєстрації та авторизації користувачів
/users	Забезпечує отримання та зміну інформації про користувачів
/requests	Виконує створення, перегляд та зміну заявок
/assignments	Забезпечує роботу з призначенням волонтерів
/comments	Виконує створення та отримання коментарів
/notifications	Забезпечує відображення повідомлень користувача

Для обмеження доступу до окремих функцій системи використовується механізм розмежування прав користувачів відповідно до ролей. Адміністративні функції доступні лише користувачам із відповідними повноваженнями, тоді як звичайні користувачі працюють лише з дозволеним набором операцій.

Така організація серверної частини дозволяє забезпечити незалежність клієнтського застосунку від внутрішньої структури системи, спростити розширення функціональності та зменшити кількість змін під час подальшої модифікації веб-платформи.

2.5 Висновки по розділу 2

У другому розділі виконано розробку серверної частини вебплатформи координації волонтерської діяльності та визначено основні принципи організації її роботи.

На початковому етапі було сформовано серверну архітектуру системи та визначено склад основних компонентів, необхідних для забезпечення взаємодії між клієнтською частиною, сервером і базою даних. Обрана архітектура побудована на використанні платформи Node.js та фреймворку Express із реалізацією взаємодії між компонентами через REST API.

В межах розділу визначено мінімальні програмно-апаратні вимоги до серверного середовища та обґрунтовано вибір конфігурації, достатньої для підтримки роботи веб-платформи та обробки користувацьких запитів.

Окрему увагу приділено проєктуванню структури бази даних. Сформована модель даних забезпечує зберігання інформації про користувачів, заявки, призначення волонтерів, повідомлення, коментарі та інші службові дані, необхідні для функціонування системи.

Також виконано опис реалізації серверних модулів та організації обробки запитів із використанням маршрутизації, контролерів, сервісного рівня та механізму доступу до бази даних через Prisma ORM.

Отримані результати створюють основу для подальшої реалізації клієнтської частини веб-платформи та забезпечують можливість організації повноцінної взаємодії користувачів із серверним застосунком.

3 РОЗРОБКА КЛІЄНТСЬКОЇ ЧАСТИНИ СИСТЕМИ

3.1 Проєктування клієнтської архітектури

Клієнтська частина веб-платформи координації волонтерської діяльності призначена для безпосередньої взаємодії користувача із системою [2, 10]. Саме через неї користувач виконує авторизацію, переглядає заявки, створює нові звернення, змінює статуси завдань або отримує інформацію відповідно до своєї ролі.

Клієнтська частина проєктується як вебзастосунок на основі React [5]. Використання компонентного підходу дозволяє розділити інтерфейс на окремі логічні частини: сторінки авторизації, панель користувача, список заявок, форму створення заявки, елементи фільтрації та службові повідомлення. Така організація спрощує розробку, оскільки кожен компонент відповідає за власну частину інтерфейсу і може змінюватися окремо від інших.

Основним завданням клієнтської архітектури є зручне відображення даних, отриманих із серверної частини, та передавання дій користувача на сервер через REST API [13]. Клієнтський застосунок не зберігає основні дані системи постійно, а отримує їх із сервера у відповідь на відповідні запити. Це дозволяє підтримувати актуальність інформації про заявки, користувачів, статуси та призначення волонтерів.

Під час проєктування клієнтської частини враховувалося розмежування прав доступу між різними категоріями користувачів. Адміністратор повинен мати доступ до керування користувачами та загального контролю системи. Координатор працює із заявками, призначеннями та статусами виконання. Волонтер отримує доступ до призначених йому завдань та може переглядати необхідну інформацію для їх виконання.

Загальна структура клієнтської частини складається з таких основних елементів:

– сторінка авторизації користувача;

– головна панель після входу в систему;

– сторінка перегляду заявок;

– форма створення або редагування заявки;

– сторінка перегляду детальної інформації про заявку;

– елементи керування статусами та призначеннями;

– службові повідомлення про результат виконання дій.

Взаємодія користувача із системою починається зі сторінки авторизації. Після успішного входу клієнтська частина отримує інформацію про роль користувача і на її основі визначає, які елементи інтерфейсу потрібно відобразити. Це дозволяє не перевантажувати користувача функціями, які йому не потрібні або недоступні.

Для роботи із заявками клієнтська частина надсилає запити до серверної частини, отримує відповідь у форматі JSON та відображає дані у вигляді списків, карток або форм. При створенні або оновленні заявки дані з форми передаються на сервер, де проходять перевірку та зберігаються у базі даних.

Окрему увагу під час проєктування було приділено простоті навігації. Користувач повинен швидко переходити між основними сторінками системи, бачити актуальний стан заявок і розуміти, які дії доступні на поточному екрані. Такий підхід особливо важливий для волонтерської діяльності, де швидкість обробки інформації має практичне значення [10].

Таким чином, клієнтська архітектура веб-платформи побудована як набір взаємопов’язаних React-компонентів, які забезпечують відображення даних, взаємодію з користувачем та обмін інформацією із серверною частиною через REST API. Обрана структура дозволяє реалізувати основні сценарії роботи системи та залишає можливість подальшого розширення інтерфейсу без повної перебудови клієнтської частини.

Для уточнення сценаріїв взаємодії користувачів із клієнтською частиною системи доцільно побудувати use case діаграму. Вона відображає основні дії, які можуть виконувати користувачі веб-платформи залежно від їхньої ролі. Оскільки система передбачає розмежування прав доступу, на діаграмі виділено три основні ролі: адміністратора, координатора та волонтера.

Рисунок 3.1 – Use case діаграма клієнтської частини веб-платформи

Наведена діаграма показує, що всі користувачі починають роботу із авторизації в системі. Подальший набір доступних дій залежить від ролі користувача. Адміністратор виконує керування обліковими записами та контролює загальний стан системи. Координатор працює із заявками, змінює їхній статус і призначає волонтерів. Волонтер переглядає призначені йому завдання, ознайомлюється з деталями заявки та може додавати коментарі щодо виконання роботи.

Структура клієнтського інтерфейсу веб-платформи будується на основі окремих сторінок і багаторазово використовуваних React-компонентів. Такий підхід дозволяє розділити інтерфейс на логічні частини, спростити навігацію між екранами та забезпечити різний набір доступних дій залежно від ролі користувача.

Основними екранами клієнтської частини є:

– сторінка авторизації користувача;

– головна панель користувача після входу в систему;

– сторінка перегляду заявок;

– сторінка створення нової заявки;

– сторінка детального перегляду заявки;

– сторінка керування призначеннями волонтерів;

– сторінка повідомлень;

– адміністративна сторінка керування користувачами.

Для уточнення структури клієнтського інтерфейсу було побудовано низькодеталізований прототип основних сторінок веб-платформи (рис.3.2). Wireframe дозволяє показати розташування базових елементів інтерфейсу без прив’язки до остаточного графічного оформлення.

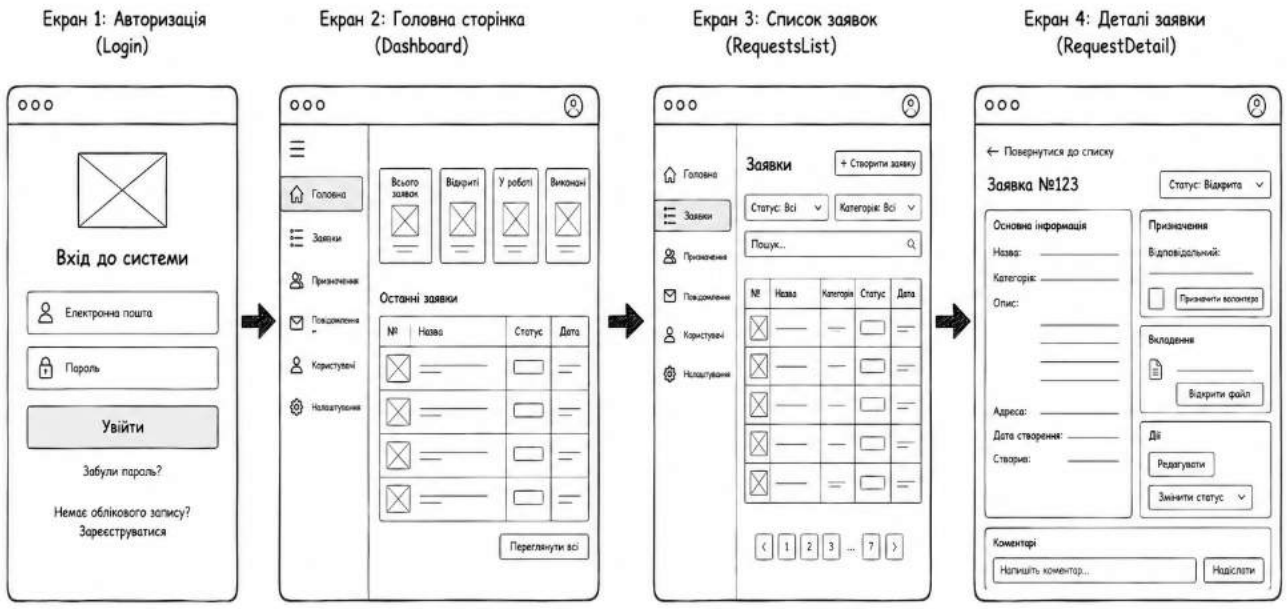

Рисунок 3.2 – Wireframe клієнтської частини веб-платформи

На рисунку 3.2 показано основні екрани клієнтської частини системи. Сторінка авторизації забезпечує вхід користувача до системи. Після авторизації користувач переходить до головної панелі, де відображається коротка інформація про заявки та доступні дії. Окремо передбачено сторінку перегляду заявок із можливістю фільтрації та сторінку детального перегляду заявки, де користувач може ознайомитися з описом, статусом і призначеним виконавцем.

Наведений wireframe відображає загальну логіку організації клієнтської частини та послідовність переходів між основними екранами веб-платформи. На відміну від остаточного інтерфейсу, прототип не деталізує оформлення та візуальний стиль, а використовується для визначення складу сторінок і основних елементів взаємодії.

Основними компонентами клієнтської частини є сторінка авторизації, головна сторінка користувача, модуль роботи із заявками та сторінка перегляду детальної інформації про вибране звернення.

Сторінка авторизації забезпечує початкову взаємодію користувача із системою та виконує перевірку введених облікових даних. Після успішного входу

користувач переходить до головної сторінки, де відображається перелік доступних функцій відповідно до його ролі.

Основна частина роботи користувача виконується через сторінку перегляду заявок. У цьому модулі реалізується відображення списку звернень, пошук та перехід до детального перегляду.

Сторінка детального перегляду заявки використовується для відображення повної інформації про звернення, поточного стану виконання та виконання доступних дій залежно від прав доступу користувача.

Сукупність описаних сторінок формує мінімально необхідну структуру клієнтської частини веб-платформи та забезпечує реалізацію основних сценаріїв взаємодії із системою (рис.3.3).

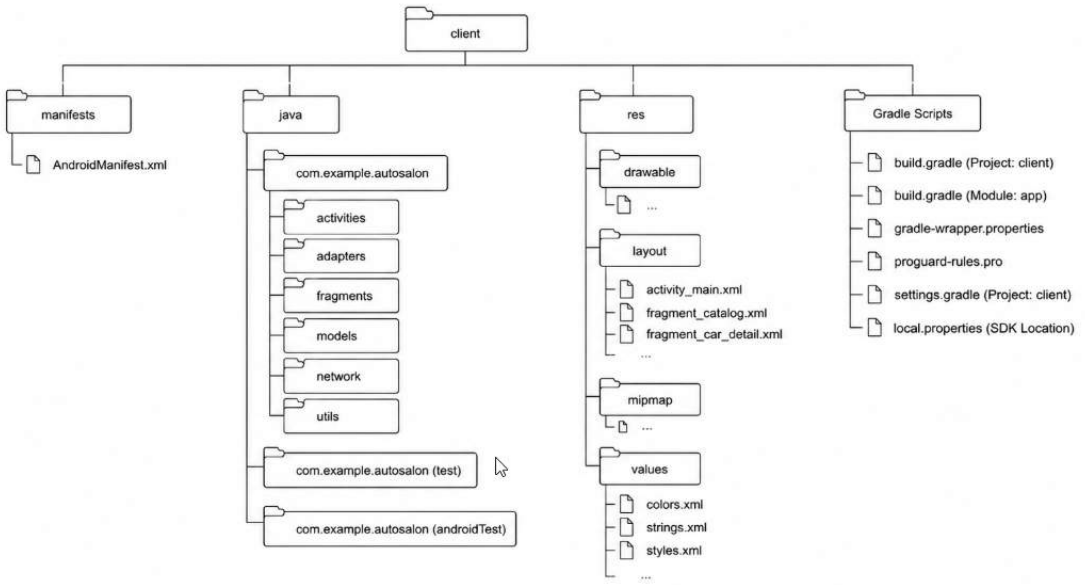

Рисунок 3.3 – Структура проєкту клієнтського застосунку

3.2 Програмно-апаратні вимоги до клієнта

Клієнтська частина веб-платформи не висуває високих вимог до апаратного забезпечення користувача, оскільки основна обробка даних виконується на

серверній стороні [13]. На стороні клієнта здійснюється формування інтерфейсу, передавання запитів до сервера та відображення отриманих результатів [5, 13].

Для стабільної роботи системи достатньо використання персонального комп’ютера, ноутбука, планшета або сучасного смартфона із доступом до мережі Інтернет. Мінімально достатнім обсягом оперативної пам’яті можна вважати 4 ГБ, що забезпечує коректну роботу браузера та відображення інтерфейсу. Для комфортної роботи при одночасному використанні декількох вкладок браузера доцільно використовувати пристрої з 8 ГБ оперативної пам’яті або більше.

З точки зору обчислювальних ресурсів достатньо сучасного двоядерного процесора середнього рівня або еквівалентного рішення мобільного класу. Оскільки клієнтська частина не виконує локального аналізу великих обсягів даних і не використовує ресурсоємні алгоритми обробки, застосування високопродуктивного обладнання не є необхідним.

Для коректної роботи веб-платформи користувач повинен використовувати сучасний браузер із підтримкою JavaScript та актуальних вебстандартів [12]. Робота системи передбачається у браузерах Google Chrome, Mozilla Firefox, Microsoft Edge або аналогічних рішеннях із підтримкою сучасних механізмів роботи вебзастосунків [12].

Також важливою умовою використання системи є стабільне мережеве підключення. Передавання інформації між клієнтською та серверною частинами здійснюється у форматі JSON через HTTP-запити, тому навіть звичайного широкосмугового або мобільного доступу до мережі достатньо для виконання типових операцій — авторизації, перегляду заявок, створення звернень та оновлення даних.

Для комфортної роботи з інтерфейсом рекомендується використовувати пристрої з роздільною здатністю екрана не нижче 1366×768 пікселів. Разом із тим клієнтська частина повинна коректно працювати і на пристроях із меншою шириною екрана за рахунок адаптивного відображення елементів інтерфейсу.

Таким чином, запропонована клієнтська частина не потребує спеціалізованого обладнання та може використовуватися на більшості сучасних пристроїв без додаткових вимог до програмного середовища.

3.3 Реалізація клієнтських модулів

Після завершення етапу проєктування виконується реалізація клієнтської частини вебплатформи. Клієнтський застосунок побудований із використанням React та реалізований як набір взаємопов’язаних компонентів, що забезпечують відображення даних, навігацію між сторінками та взаємодію із серверною частиною [5].

Функціональність клієнтської частини поділено між окремими сторінками та допоміжними компонентами. Основними елементами реалізації є сторінка авторизації, головна сторінка користувача, модуль роботи із заявками та сторінка детального перегляду інформації [5].

Після відкриття вебплатформи користувач переходить до сторінки авторизації, де вводить облікові дані для подальшої перевірки на сервері. У випадку успішної авторизації виконується перехід до головної сторінки застосунку.

Отримання та передавання інформації між клієнтською і серверною частинами виконується через REST API [13]. Дані передаються у форматі JSON та використовуються для оновлення стану інтерфейсу без повного перезавантаження сторінки [14].

Окремим елементом реалізації клієнтської частини є організація взаємодії із серверною частиною системи. Клієнтський застосунок не виконує локального зберігання основних даних та отримує необхідну інформацію під час роботи через REST API [13].

Після виконання дій користувача формується HTTP-запит до серверної частини [13]. Сервер виконує обробку запиту, звертається до бази даних та повертає результат у форматі JSON [8, 14]. Отримані дані використовуються для оновлення інтерфейсу без повного перезавантаження сторінки [5].

Схему клієнт-серверної взаємодії вебплатформи наведено на рисунку 3.4.

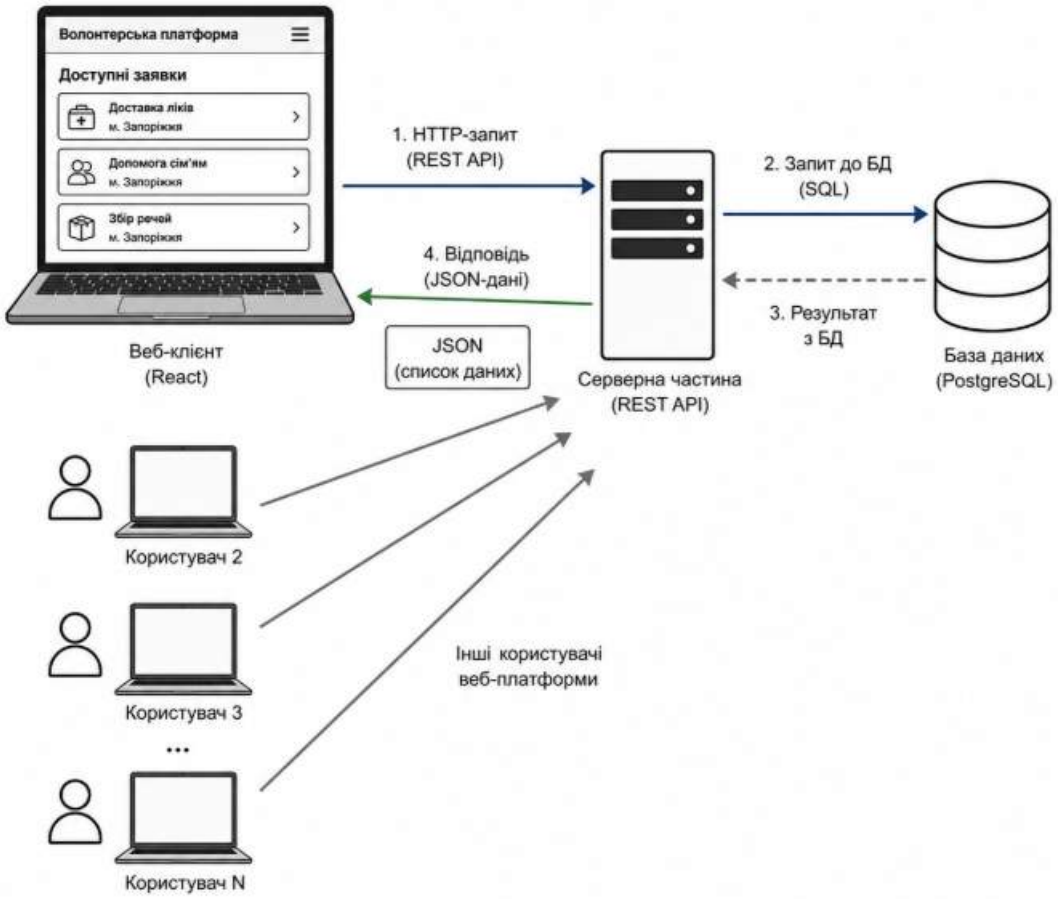

Рисунок 3.4 – Схема клієнт-серверної взаємодії веб-платформи

Наведена схема демонструє загальний принцип взаємодії клієнтської та серверної частин вебплатформи під час виконання основних сценаріїв роботи користувача. Передавання інформації між компонентами системи здійснюється через HTTP-запити із використанням REST API, а результати обробки повертаються клієнтській частині у форматі JSON.

Реалізація клієнтської частини виконується у вигляді набору окремих функціональних модулів, кожний з яких відповідає за певний сценарій взаємодії користувача із системою.

Початковим модулем є сторінка авторизації, яка забезпечує введення та перевірку облікових даних користувача. Після успішного проходження перевірки виконується перехід до основного інтерфейсу веб-платформи.

Подальша робота здійснюється через модуль перегляду заявок, який забезпечує отримання інформації із серверної частини, відображення списку доступних звернень та відкриття детальної інформації про обраний запис.

Для організації взаємодії із серверною частиною використовується окремий сервіс доступу до даних, через який виконується формування запитів та обробка отриманих відповідей.

Навігація між сторінками реалізована засобами маршрутизації клієнтського застосунку. Це дозволяє виконувати переходи між екранами без повного перезавантаження сторінки та забезпечує безперервність роботи користувача із системою.

Фрагмент реалізації маршрутизації наведено у лістингу 3.1.

Лістинг 3.1 – Фрагмент реалізації маршрутизації клієнтської частини

```
<Routes>
<Route        path="/"        element={<LoginPage        />}        />
<Route    path="/dashboard"    element={<DashboardPage    />}    />
<Route     path="/requests"    element={<RequestsPage     />}    />
<Route   path="/request/:id"   element={<RequestDetailPage   />}   />
</Routes>
```

Наведений фрагмент демонструє організацію переходів між основними сторінками веб-платформи та забезпечує формування єдиного механізму навігації всередині клієнтської частини системи.

3.4 Висновки по розділу 3

У цьому розділі було виконано проєктування та опис реалізації клієнтської частини веб-платформи для координації волонтерської діяльності. У процесі роботи визначено основні сценарії взаємодії користувача із системою та сформовано структуру клієнтського застосунку.

Для опису роботи користувача побудовано use case діаграму та розроблено структуру інтерфейсу веб-платформи. Також визначено склад основних сторінок застосунку та їх взаємозв’язок під час виконання типових сценаріїв роботи.

В межах розділу описано програмно-апаратні вимоги до клієнтської частини та наведено підхід до організації навігації між сторінками застосунку. Окремо розглянуто принцип взаємодії клієнтської та серверної частин системи під час обміну даними.

Результатом виконання розділу стало формування клієнтської частини веб-платформи та визначення принципів її подальшої інтеграції із іншими компонентами системи.

4 ВИПРОБУВАННЯ СИСТЕМИ

4.1 Випробування користувацького інтерфейсу

Після завершення реалізації вебплатформи було виконано випробування користувацького інтерфейсу. Основною метою перевірки було визначення коректності відображення сторінок системи, роботи навігації, форм введення даних та взаємодії клієнтської частини із сервером [4].

Під час випробування перевірялися основні сценарії роботи користувача: вхід до системи, перехід до головної панелі, перегляд списку заявок, відкриття детальної інформації про заявку, створення нового звернення та зміна статусу

виконання. Окремо перевірялося, чи коректно інтерфейс реагує на дії користувача та чи відображає повідомлення про успішне виконання операцій або помилки [4].

На першому етапі було перевірено сторінку авторизації. Користувач вводить адресу електронної пошти та пароль, після чого клієнтська частина надсилає запит до серверної частини для перевірки облікових даних [13, 15]. У разі успішної авторизації виконується перехід до головної панелі вебплатформи.

Для ілюстрації роботи сторінки авторизації наведено рисунок 4.1.

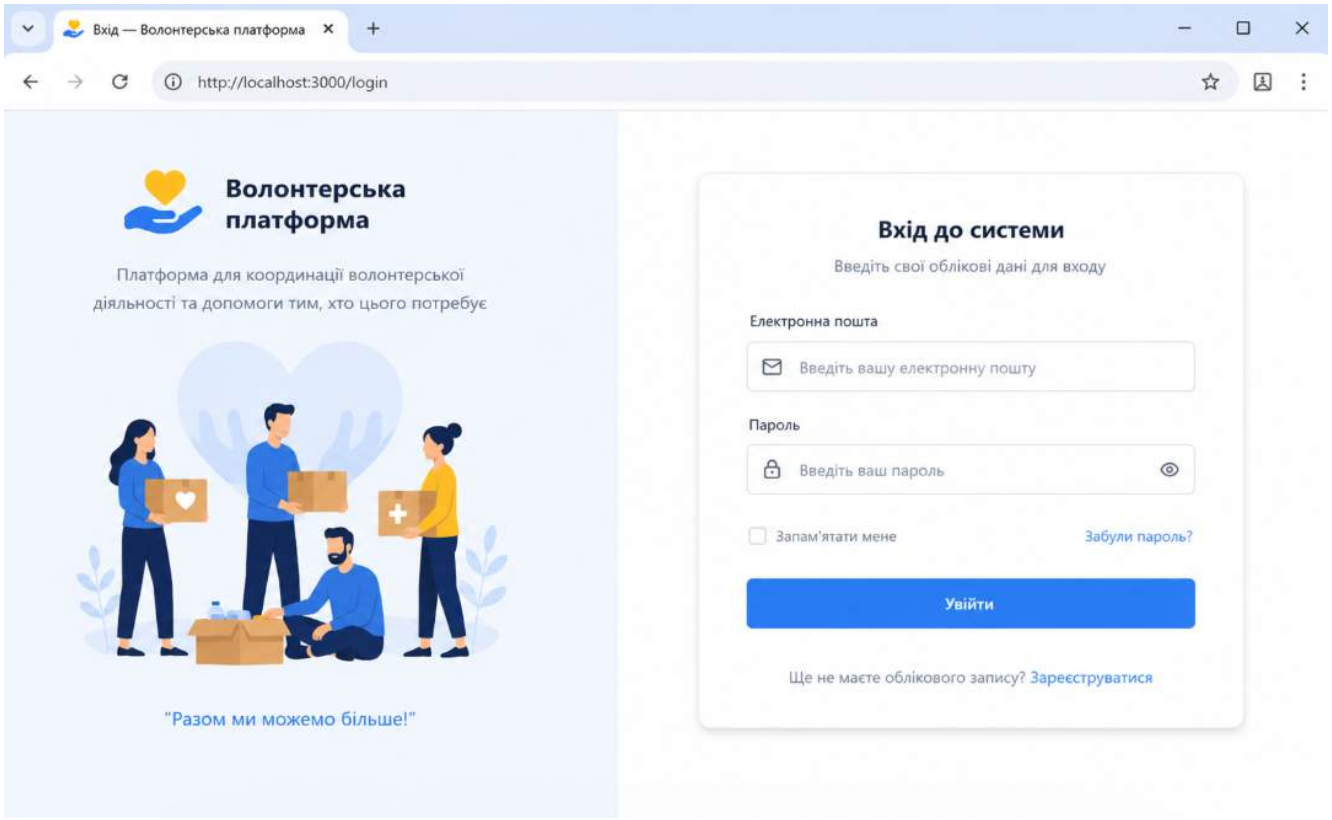

Рисунок 4.1 – Сторінка авторизації вебплатформи

Після успішного проходження авторизації користувач отримує доступ до головної панелі вебплатформи. Головна сторінка використовується як основний робочий інтерфейс системи та забезпечує доступ до основних функцій залежно від ролі користувача.

На головній панелі відображаються доступні розділи системи, інформація про поточний стан заявок та елементи навігації між сторінками. Інтерфейс побудований таким чином, щоб користувач міг швидко перейти до перегляду

списку заявок, створення нового звернення або перегляду інформації про виконання завдань.

Під час випробування перевірялася коректність переходу після авторизації, відображення елементів інтерфейсу та доступність функцій відповідно до ролі користувача. Окремо оцінювалася робота навігації між сторінками без необхідності повторного завантаження веб-застосунку.

Приклад головної панелі користувача наведено на рисунку 4.2.

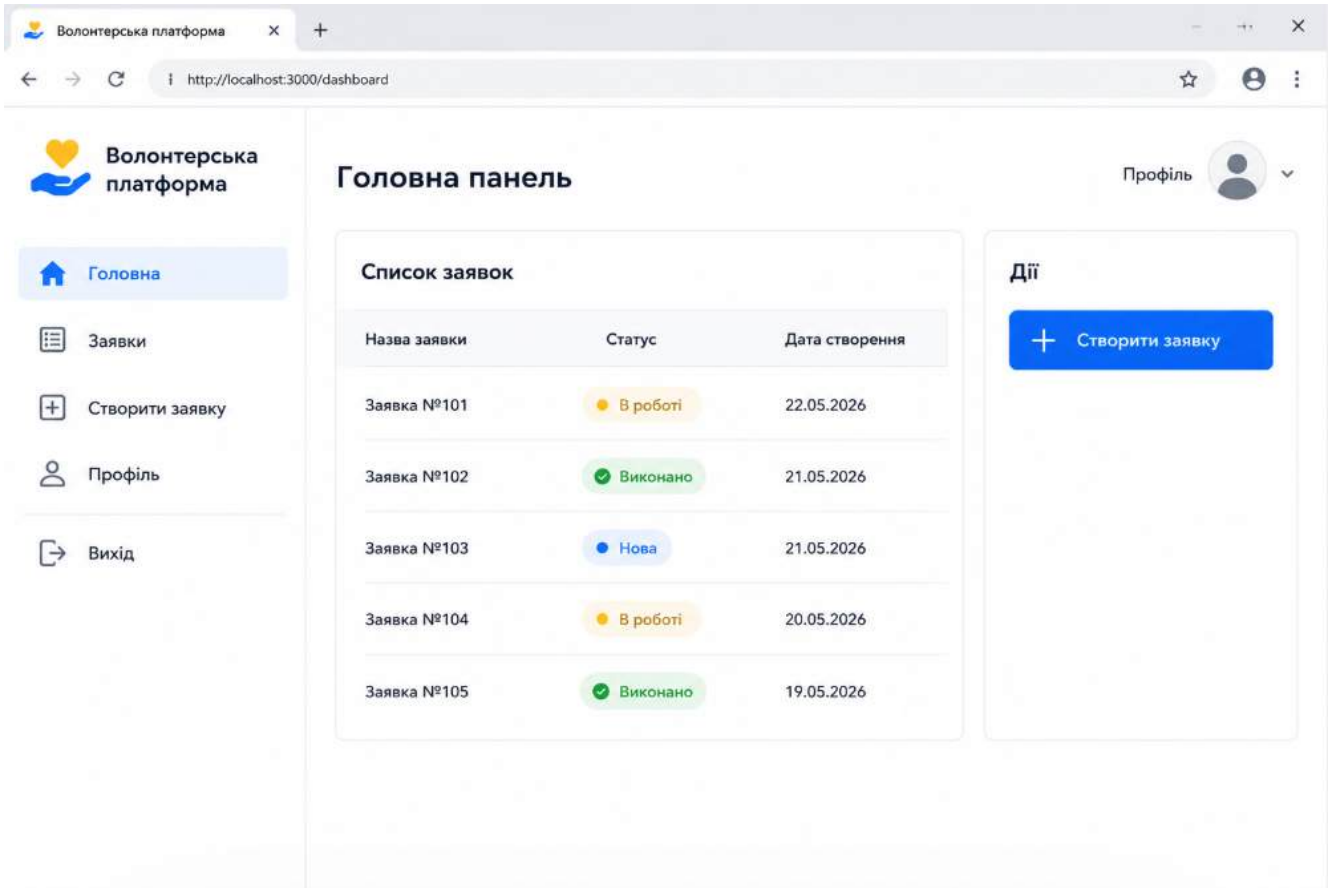

Рисунок 4.2 – Головна панель користувача після входу в систему

Після переходу до головної панелі користувач отримує доступ до перегляду та керування заявками через відповідний розділ веб-платформи.

Сторінка перегляду заявок використовується для відображення інформації про поточні звернення та забезпечує швидкий доступ до операцій роботи із заявками. Для кожного запису відображається назва заявки, її поточний статус та дата створення.

Під час випробування перевірялася коректність завантаження списку заявок після відкриття сторінки, правильність відображення отриманих із серверної частини даних та робота переходу до детального перегляду окремого запису.

Окремо перевірялося оновлення інформації після зміни стану заявки та коректність відображення результатів без повторного входу користувача до системи.

Також оцінювалася робота навігації між сторінками та реакція інтерфейсу під час виконання типових дій користувача.

Приклад сторінки перегляду списку заявок наведено на рисунку 4.3.

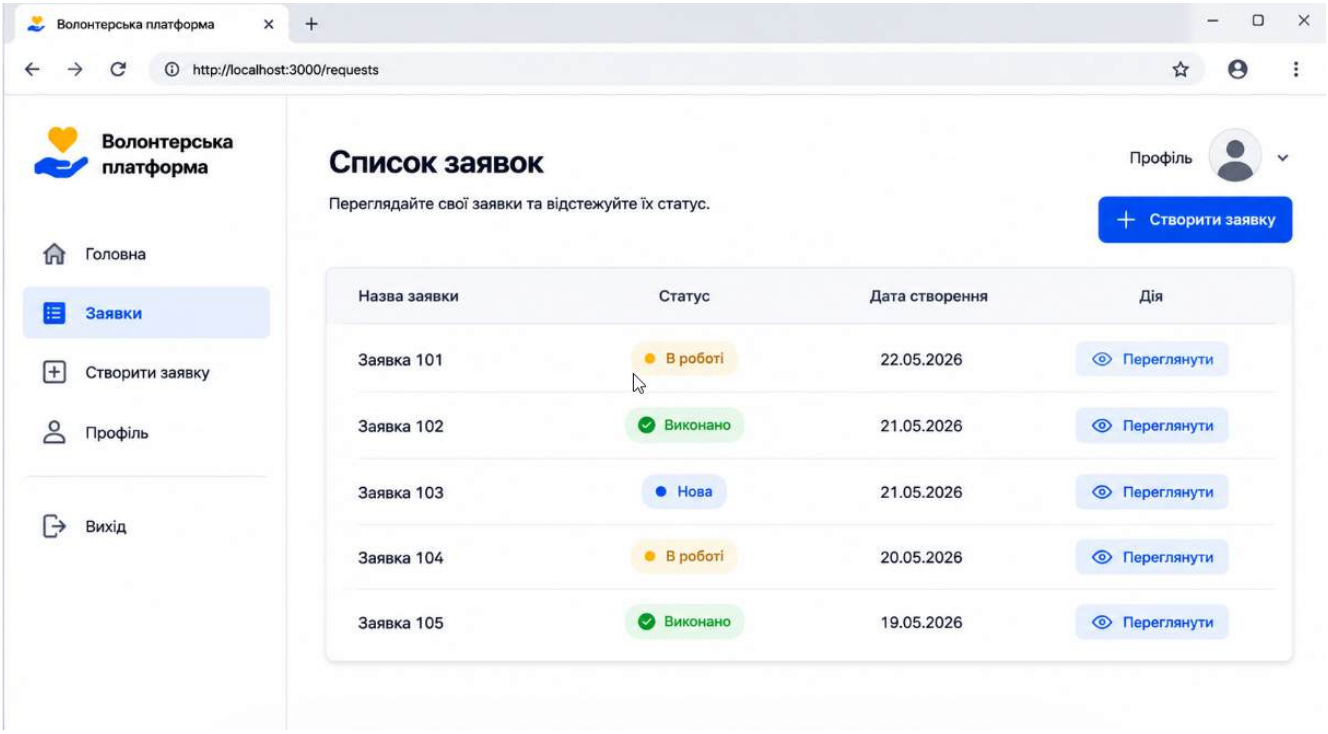

Рисунок 4.3 – Сторінка перегляду списку заявок

Після вибору окремого запису зі списку користувач переходить до сторінки детального перегляду заявки. Цей етап випробування використовувався для перевірки коректності відкриття окремого запису, відображення повної інформації та роботи переходів між сторінками системи.

На сторінці детального перегляду користувач отримує доступ до розширених відомостей про заявку. Відображаються основні характеристики

звернення, поточний стан виконання та інформація, необхідна для подальшої роботи із записом.

Під час перевірки особлива увага приділялася правильності передавання даних між сторінкою списку заявок і сторінкою перегляду, коректності відображення інформації після відкриття запису та роботі навігації під час повернення до попереднього екрана.

Окремо перевірялося відображення зміненого стану заявки після виконання дій користувача та коректність оновлення інтерфейсу після повторного відкриття сторінки.

За результатами випробування встановлено, що перехід до детального перегляду виконується без помилок, а відображення інформації відповідає структурі даних системи.

Приклад сторінки детального перегляду заявки наведено на рисунку 4.4.

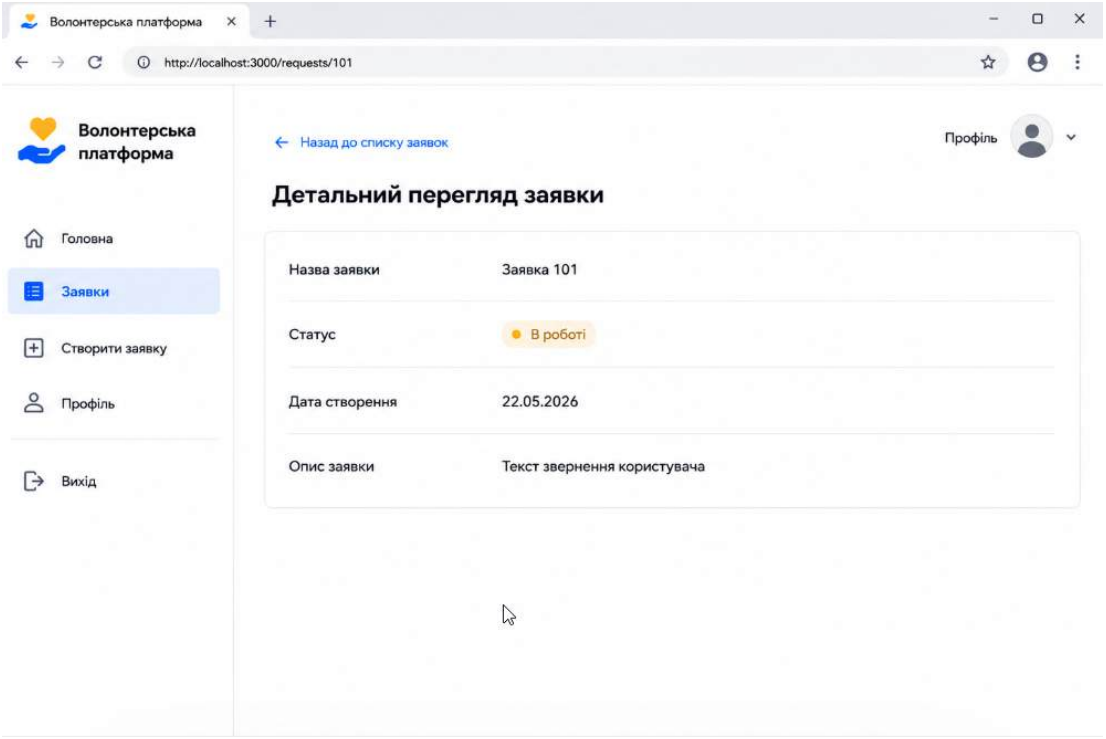

Рисунок 4.4 – Сторінка детального перегляду заявки

Під час випробування сторінки детального перегляду було підтверджено коректність відкриття окремої заявки після вибору запису зі списку та правильність відображення інформації, отриманої із серверної частини системи.

Перевірка показала, що навігація між сторінками виконується без помилок, а перехід між головною панеллю, списком заявок та сторінкою детального перегляду не потребує повторного завантаження застосунку. Дані після відкриття сторінки відображаються у повному обсязі та відповідають структурі, передбаченій під час проєктування клієнтської частини.

Окремо було перевірено коректність відображення статусу заявки та збереження переходів між сторінками під час повторного відкриття записів. Під час виконання типових сценаріїв використання помилок інтерфейсу або порушень відображення виявлено не було.

Таким чином, результати випробування підтвердили працездатність користувацького інтерфейсу та можливість виконання основних операцій роботи із заявками через веб-платформу.

4.2 Аналіз продуктивності системи

Після перевірки роботи користувацького інтерфейсу було виконано аналіз поведінки веб-платформи під час виконання основних сценаріїв використання.

Оскільки розроблена система реалізована за клієнт-серверним принципом, на швидкість роботи впливають як процеси обробки запитів на серверній стороні, так і відображення отриманої інформації у клієнтському інтерфейсі [13]. Основними сценаріями використання, що розглядалися під час аналізу, були авторизація користувача, відкриття головної панелі, перегляд списку заявок та перехід до сторінки детального перегляду.

Під час виконання зазначених дій перевірялася загальна працездатність системи, послідовність переходів між сторінками та коректність відображення інформації після отримання відповіді від серверної частини [4].

Особлива увага приділялася роботі механізму взаємодії між клієнтською та серверною частинами, оскільки саме цей процес забезпечує отримання актуальних даних та їх подальше відображення користувачеві без необхідності повторного відкриття застосунку [13, 14].

Проведений аналіз показав можливість виконання основних сценаріїв роботи із системою та підтвердив коректність організації взаємодії між її компонентами.

4.3 Висновки по розділу 4

В четвертому розділі виконано випробування розробленої веб-платформи для координації волонтерської діяльності та перевірено роботу основних сценаріїв використання системи.

В межах випробування користувацького інтерфейсу було послідовно перевірено процес авторизації, відкриття головної панелі користувача, перегляд списку заявок та перехід до сторінки детального перегляду. Перевірка підтвердила коректність переходів між сторінками та відображення інформації відповідно до реалізованої логіки роботи застосунку.

Також виконано аналіз роботи системи під час взаємодії клієнтської та серверної частин. Розглянуто виконання типових дій користувача та особливості передавання даних між компонентами веб-платформи під час роботи із заявками.

За результатами проведеного випробування встановлено, що реалізована система забезпечує виконання передбачених функцій та підтримує основні сценарії взаємодії користувачів із веб-платформою.

ВИСНОВКИ

В межах дипломної роботи виконано розробку вебплатформи для координації волонтерської діяльності, призначеної для організації взаємодії між користувачами системи, централізованої роботи із заявками та підтримки процесу розподілу й контролю виконання завдань.

Під час виконання роботи проведено аналіз предметної області та розглянуто особливості організації волонтерської діяльності в умовах використання сучасних інформаційних технологій. Проаналізовано існуючі підходи до організації взаємодії між учасниками процесу та визначено основні обмеження використання розрізнених засобів комунікації під час роботи із заявками. На основі проведеного аналізу сформульовано функціональні та нефункціональні вимоги до розроблюваної системи.

В процесі проєктування визначено архітектуру програмного забезпечення, побудовано структуру взаємодії між клієнтською та серверною частинами системи, спроєктовано організацію збереження даних та механізми обміну інформацією між компонентами вебплатформи.

В межах реалізації створено серверну частину системи, що забезпечує обробку запитів користувачів, роботу з інформацією про заявки та взаємодію з базою даних. Також реалізовано клієнтську частину у вигляді вебінтерфейсу, який забезпечує виконання основних сценаріїв роботи користувача, включаючи авторизацію, перегляд списку заявок, отримання детальної інформації та навігацію між сторінками системи.

Під час завершального етапу роботи виконано випробування розробленої вебплатформи та перевірено коректність реалізації основних функцій системи. Проведено перевірку роботи користувацького інтерфейсу та розглянуто особливості взаємодії між клієнтською і серверною частинами під час виконання типових сценаріїв використання.

Отримані результати підтвердили можливість використання розробленої вебплатформи як єдиного програмного середовища для організації роботи із заявками та підтримки процесів координації волонтерської діяльності.

Практична цінність роботи полягає у можливості подальшого розвитку та використання запропонованого програмного рішення для автоматизації окремих процесів організації волонтерської діяльності та централізованого керування інформацією в межах єдиної системи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1.Про волонтерську діяльність : Закон України від 19.04.2011 № 3236-VI. URL: https://zakon.rada.gov.ua/laws/show/3236-17 (дата звернення: 25.05.2026).

2.Sommerville I. Software Engineering. 11th ed. Harlow : Pearson Education Limited, 2020. 811 p.

3.Pressman R. S., Maxim B. R. Software Engineering: A Practitioner's Approach. 9th ed. New York : McGraw-Hill Education, 2019. 992 p.

4.ISO/IEC 25010:2011. Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models. Geneva : International Organization for Standardization, 2011.

5.React Documentation. URL: https://react.dev/ (дата звернення: 25.05.2026).

6.Node.js Documentation. URL: https://nodejs.org/docs/latest/api/ (дата звернення: 25.05.2026).

7.Express.js Documentation. URL: https://expressjs.com/ (дата звернення: 25.05.2026).

8.PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 25.05.2026).

9.Авраменко В. С., Авраменко А. С. Проектування інформаційних систем : навч. посіб. Черкаси : Черкаський національний університет ім. Б. Хмельницького, 2017. 434 с.

10.Корнієнко Б. Я. Компоненти програмної інженерії. Архітектура програмного забезпечення : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2023.

11.Харів Н. О. Бази даних та інформаційні системи : навч. посіб. Рівне : НУВГП, 2018. 127 с.

12.Босько В. В. Web-програмування : навч. посіб. Кропивницький : ЦНТУ, 2021.

13.Fielding R. T. Architectural Styles and the Design of Network-Based Software Architectures *:* doctoral dissertation. Irvine : University of California, 2000.

14.Bray T. The JavaScript Object Notation (JSON) Data Interchange Format *:* RFC 8259. Internet Engineering Task Force, 2017.

15.Jones M., Bradley J., Sakimura N. JSON Web Token (JWT) *:* RFC 7519. Internet Engineering Task Force, 2015.

16.OWASP Foundation. Authentication Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html (дата звернення: 25.05.2026).

17.Prisma ORM Documentation. URL: https://www.prisma.io/docs (дата звернення: 25.05.2026).

18.Visual Studio Code Documentation. URL: https://code.visualstudio.com/docs (дата звернення: 25.05.2026).

19.Elmasri R., Navathe S. B. Fundamentals of Database Systems. 7th ed. Boston : Pearson, 2016. 1272 p.

20.Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston : Addison-Wesley, 1994. 395 p.

ДОДАТОК А
ЛІСТИНГИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Лістинг А.1 – DBML-опис сутності Roles

```
Table roles {
id int [pk, increment]
name varchar [not null, unique]
description text
}
```

Лістинг А.2 – DBML-опис сутності Users

```
Table users {
id int [pk, increment]
role_id int [not null, ref: > roles.id]
full_name varchar [not null]
email varchar [not null, unique]
password_hash varchar [not null]
phone varchar
created_at timestamp
is_active boolean
}
```

Лістинг А.3 – DBML-опис сутності Categories

```
Table categories {
id int [pk, increment]
name varchar [not null, unique]
description text
}
```

Лістинг А.4 – DBML-опис сутності Statuses

```
Table statuses {
id int [pk, increment]
name varchar [not null, unique]
description text
}
```

Лістинг А.5 – DBML-опис сутності Requests

```
Table requests {
id int [pk, increment]
author_id int [not null, ref: > users.id]
category_id int [not null, ref: > categories.id]
status_id int [not null, ref: > statuses.id]
```

```
title varchar [not null]
description text [not null]
location varchar
priority varchar
created_at timestamp
updated_at timestamp
}
```

Лістинг А.6 – DBML-опис сутності Assignments

```
Table assignments {
id int [pk, increment]
request_id int [not null, ref: > requests.id]
volunteer_id int [not null, ref: > users.id]
assigned_at timestamp
completed_at timestamp
}
```

Лістинг А.7 – DBML-опис сутності Comments

```
Table comments {
id int [pk, increment]
request_id int [not null, ref: > requests.id]
user_id int [not null, ref: > users.id]
comment_text text [not null]
created_at timestamp
}
```

Лістинг А.8 – DBML-опис сутності Notifications

```
Table notifications {
id int [pk, increment]
user_id int [not null, ref: > users.id]
request_id int [ref: > requests.id]
message text [not null]
is_read boolean
created_at timestamp
}
```

Лістинг А.9 – DBML-опис сутності Attachments

```
Table attachments {
id int [pk, increment]
request_id int [not null, ref: > requests.id]
uploaded_by int [not null, ref: > users.id]
file_name varchar [not null]
file_path varchar [not null]
uploaded_at timestamp
}
```

Лістинг А.10 – DBML-опис сутності ActivityLogs

```
Table activity_logs {
id int [pk, increment]
user_id int [not null, ref: > users.id]
request_id int [ref: > requests.id]
action varchar [not null]
created_at timestamp
}
```

Лістинг А.11 – Код точки входу в серверний застосунок

```
const express = require('express');
const cors = require('cors');
const authRoutes = require('./src/routes/authRoutes');
const userRoutes = require('./src/routes/userRoutes');
const requestRoutes = require('./src/routes/requestRoutes');
const categoryRoutes = require('./src/routes/categoryRoutes');
const statusRoutes = require('./src/routes/statusRoutes');
const errorHandler = require('./src/utils/errorHandler');
const app = express();
app.use(cors());
app.use(express.json());
app.use('/api/auth', authRoutes);
app.use('/api/users', userRoutes);
app.use('/api/requests', requestRoutes);
app.use('/api/categories', categoryRoutes);
app.use('/api/statuses', statusRoutes);
app.use(errorHandler);
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
console.log(`Server started on port ${PORT}`);
});
```

Лістинг А.12 – Підключення Prisma ORM до бази даних

```
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();
module.exports = prisma;
```

Лістинг А.13 – Фрагмент контролера авторизації

```
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const prisma = require('../config/db');
async function login(req, res, next) {
try {
const { email, password } = req.body;
const user = await prisma.users.findUnique({
where: { email },
include: { role: true }
});
```

```
if (!user) {
return res.status(401).json({
message: 'Невірна електронна пошта або пароль'
});
}
const validPassword = await bcrypt.compare(
password,
user.password_hash
);
if (!validPassword) {
return res.status(401).json({
message: 'Невірна електронна пошта або пароль'
});
}
const token = jwt.sign(
{
id: user.id,
role: user.role.name
},
process.env.JWT_SECRET,
{ expiresIn: '24h' }
);
res.json({
token,
user: {
id: user.id,
fullName: user.full_name,
email: user.email,
role: user.role.name
}
});
} catch (error) {
next(error);
}
}
module.exports = { login };
```

Лістинг А.14 – Middleware перевірки JWT-токена

```
const jwt = require('jsonwebtoken');
function authMiddleware(req, res, next) {
const authHeader = req.headers.authorization;
if (!authHeader) {
return res.status(401).json({
message: 'Токен авторизації відсутній'
});
}
const token = authHeader.split(' ')[1];
try {
const decoded = jwt.verify(token, process.env.JWT_SECRET);
req.user = decoded;
next();
} catch (error) {
```

```
return res.status(401).json({
message: 'Недійсний токен авторизації'
});
}
}
module.exports = authMiddleware;
```

Лістинг А.15 – Middleware перевірки ролі користувача

```
function roleMiddleware(allowedRoles) {
return function (req, res, next) {
if (!req.user || !allowedRoles.includes(req.user.role)) {
return res.status(403).json({
message: 'Недостатньо прав для виконання операції'
});
}
next();
};
}
module.exports = roleMiddleware;
```

Лістинг А.16 – Фрагмент контролера роботи із заявками

```
const requestService = require('../services/requestService');
async function getAllRequests(req, res, next) {
try {
const requests = await requestService.getAllRequests();
res.json(requests);
} catch (error) {
next(error);
}
}
async function createRequest(req, res, next) {
try {
const request = await requestService.createRequest(
req.body,
req.user.id
);
res.status(201).json(request);
} catch (error) {
next(error);
}
}
async function updateRequestStatus(req, res, next) {
try {
const { id } = req.params;
const { statusId } = req.body;
const request = await requestService.updateStatus(id, statusId);
res.json(request);
} catch (error) {
next(error);
}
```

```
}
module.exports = {
getAllRequests,
createRequest,
updateRequestStatus
};
```

Лістинг А.17 – Фрагмент сервісу роботи із заявками

```
const prisma = require('../config/db');
async function getAllRequests() {
return prisma.requests.findMany({
include: {
author: true,
category: true,
status: true,
assignments: {
include: {
volunteer: true
}
}
},
orderBy: {
created_at: 'desc'
}
});
}
async function createRequest(data, authorId) {
return prisma.requests.create({
data: {
title: data.title,
description: data.description,
location: data.location,
priority: data.priority,
author_id: authorId,
category_id: Number(data.categoryId),
status_id: Number(data.statusId),
created_at: new Date()
}
});
}
async function updateStatus(requestId, statusId) {
return prisma.requests.update({
where: {
id: Number(requestId)
},
data: {
status_id: Number(statusId),
updated_at: new Date()
}
});
}
module.exports = {
```

```
getAllRequests,
createRequest,
updateStatus
};
```

Лістинг А.18 – Фрагмент сервісу призначення волонтера

```
const prisma = require('../config/db');
async function assignVolunteer(requestId, volunteerId) {
return prisma.assignments.create({
data: {
request_id: Number(requestId),
volunteer_id: Number(volunteerId),
assigned_at: new Date()
}
});
}
async function completeAssignment(assignmentId) {
return prisma.assignments.update({
where: {
id: Number(assignmentId)
},
data: {
completed_at: new Date()
}
});
}
module.exports = {
assignVolunteer,
completeAssignment
};
```

Лістинг А.19 – Маршрути для роботи із заявками

```
const express = require('express');
const router = express.Router();
const authMiddleware = require('../middleware/authMiddleware');
const roleMiddleware = require('../middleware/roleMiddleware');
const requestController =
require('../controllers/requestController');
router.get(
'/',
authMiddleware,
requestController.getAllRequests
);
router.post(
'/',
authMiddleware,
roleMiddleware(['admin', 'coordinator']),
requestController.createRequest
);
router.patch(
```

```
'/:id/status',
authMiddleware,
roleMiddleware(['admin', 'coordinator']),
requestController.updateRequestStatus
);
module.exports = router;
```

Лістинг А.20 – Middleware перевірки даних заявки

```
function validateRequest(req, res, next) {
const { title, description, categoryId, statusId } = req.body;
if (!title || !description || !categoryId || !statusId) {
return res.status(400).json({
message: 'Не заповнені обов'язкові поля заявки'
});
}
next();
}
module.exports = validateRequest;
```

Лістинг А.21 – Глобальний обробник помилок

```
function errorHandler(error, req, res, next) {
console.error(error);
res.status(500).json({
message: 'Внутрішня помилка сервера',
details: error.message
});
}
module.exports = errorHandler;
```

Лістинг А.22 – Фрагмент реалізації маршрутизації клієнтської частини

```
<Routes>
<Route        path="/"        element={<LoginPage        />}        />
<Route    path="/dashboard"    element={<DashboardPage    />}    />
<Route     path="/requests"     element={<RequestsPage     />}     />
<Route   path="/request/:id"   element={<RequestDetailPage   />}   />
</Routes>
```

ДОДАТОК Б
ЛІСТИНГИ КЛІЄНТСЬКОЇ СИСТЕМИ

Лістинг Б.1 – Файл App.jsx

```
import { BrowserRouter, Routes, Route, Navigate } from "react-
router-dom";
import LoginPage from "./pages/LoginPage";
import DashboardPage from "./pages/DashboardPage";
import RequestsPage from "./pages/RequestsPage";
import RequestDetailPage from "./pages/RequestDetailPage";
import RequestFormPage from "./pages/RequestFormPage";
import Layout from "./components/Layout";
import "./styles.css";
function App() {
const token = localStorage.getItem("token");
return (
<BrowserRouter>
<Routes>
<Route path="/login" element={<LoginPage />} />
<Route
path="/"
element={token ? <Layout /> : <Navigate to="/login" replace />}
>
<Route index element={<DashboardPage />} />
<Route path="requests" element={<RequestsPage />} />
<Route path="requests/new" element={<RequestFormPage />} />
<Route path="requests/:id" element={<RequestDetailPage />} />
</Route>
<Route path="*" element={<Navigate to="/" replace />} />
</Routes>
</BrowserRouter>
);
}
export default App;
```

Лістинг Б.2 – Файл api.js

```
const API_URL = "http://localhost:5000/api";
export async function apiRequest(path, method = "GET", body = null)
{
const token = localStorage.getItem("token");
const options = {
method,
headers: {
"Content-Type": "application/json",
},
};
if (token) {
```

```
options.headers.Authorization = `Bearer ${token}`;
}
if (body) {
options.body = JSON.stringify(body);
}
const response = await fetch(`${API_URL}${path}`, options);
if (!response.ok) {
const error = await response.json().catch(() => ({}));
throw new Error(error.message || "Помилка виконання запиту");
}
return response.json();
}
```

Лістинг Б.3 – Файл LoginPage.jsx

```
import { useState } from "react";
import { useNavigate } from "react-router-dom";
import { apiRequest } from "../api";
function LoginPage() {
const navigate = useNavigate();
const [email, setEmail] = useState("");
const [password, setPassword] = useState("");
const [error, setError] = useState("");
const handleSubmit = async (event) => {
event.preventDefault();
setError("");
try {
const data = await apiRequest("/auth/login", "POST", {
email,
password,
});
localStorage.setItem("token", data.token);
localStorage.setItem("user", JSON.stringify(data.user));
navigate("/");
} catch (err) {
setError("Невірний логін або пароль");
}
};
return (
<div className="login-page">
<form className="login-form" onSubmit={handleSubmit}>
<h1>Вхід до системи</h1>
<p>Веб-платформа координації волонтерської діяльності</p>
{error && <div className="error-message">{error}</div>}
<label>Електронна пошта</label>
<input
type="email"
value={email}
placeholder="user@example.com"
onChange={(event) =>
setEmail(event.target.value)}
required
/>
```

```
<label>Пароль</label>
<input
type="password"
value={password}
placeholder="Введіть пароль"
onChange={(event) => setPassword(event.target.value)}
required
/>
<button type="submit">Увійти</button>
</form>
</div>
);
}
export default LoginPage;
```

Лістинг Б.4 – Файл Layout.jsx

```
import { Outlet, NavLink, useNavigate } from "react-router-dom";
function Layout() {
const navigate = useNavigate();
const user = JSON.parse(localStorage.getItem("user") || "{}");
const handleLogout = () => {
localStorage.removeItem("token");
localStorage.removeItem("user");
navigate("/login");
};
return (
<div className="app-layout">
<aside className="sidebar">
<h2>VolunteerHub</h2>
<nav>
<NavLink to="/">Панель</NavLink>
<NavLink to="/requests">Заявки</NavLink>
<NavLink to="/requests/new">Створити заявку</NavLink>
</nav>
<div className="user-block">
<strong>{user.name || "Користувач"}</strong>
<span>{user.role || "роль не визначена"}</span>
<button onClick={handleLogout}>Вийти</button>
</div>
</aside>
<main className="content">
<Outlet />
</main>
</div>
);
}
export default Layout;
```

Лістинг Б.5 – Файл DashboardPage.jsx

```
import { useEffect, useState } from "react";
import { apiRequest } from "../api";
function DashboardPage() {
const [stats, setStats] = useState({
total: 0,
newRequests: 0,
inProgress: 0,
completed: 0,
});
useEffect(() => {
async function loadStats() {
try {
const data = await apiRequest("/requests/statistics");
setStats(data);
} catch (error) {
console.error(error.message);
}
}
loadStats();
}, []);
return (
<section>
<h1>Панель користувача</h1>
<p className="page-description">
На цій сторінці відображається загальний стан заявок у системі.
</p>
<div className="stats-grid">
<div className="stat-card">
<span>Усього заявок</span>
<strong>{stats.total}</strong>
</div>
<div className="stat-card">
<span>Нові</span>
<strong>{stats.newRequests}</strong>
</div>
<div className="stat-card">
<span>У роботі</span>
<strong>{stats.inProgress}</strong>
</div>
<div className="stat-card">
<span>Виконані</span>
<strong>{stats.completed}</strong>
</div>
</div>
</section>
);
}
export default DashboardPage;
```

Лістинг Б.6– Файл RequestDetailPage.jsx

```
import { useEffect, useState } from "react";
import { useParams } from "react-router-dom";
import { apiRequest } from "../api";
function RequestDetailPage() {
const { id } = useParams();
const [request, setRequest] = useState(null);
const [status, setStatus] = useState("");
useEffect(() => {
async function loadRequest() {
try {
const data = await apiRequest(`/requests/${id}`);
setRequest(data);
setStatus(data.status);
} catch (error) {
console.error(error.message);
}
}
loadRequest();
}, [id]);
const handleStatusChange = async () => {
try {
const updated = await apiRequest(`/requests/${id}/status`, "PATCH",
{
status,
});
setRequest(updated);
} catch (error) {
console.error(error.message);
}
};
if (!request) {
return <p>Завантаження заявки...</p>;
}
return (
<section>
<h1>{request.title}</h1>
<div className="detail-card">
<p>
<strong>Опис:</strong> {request.description}
</p>
<p>
<strong>Категорія:</strong> {request.category?.name || "Не вказано"}
</p>
<p>
<strong>Пріоритет:</strong> {request.priority}
</p>
<p>
<strong>Відповідальний волонтер:</strong>{" "}
{request.volunteer?.name || "Не призначено"}
</p>
<p>
```

```
<strong>Дата створення:</strong>{" "}
{new Date(request.createdAt).toLocaleDateString("uk-UA")}
</p>
<div className="status-control">
<label>Статус заявки</label>
<select
value={status}
onChange={(event) => setStatus(event.target.value)}
>
<option value="new">Нова</option>
<option value="in_progress">У роботі</option>
<option value="completed">Виконана</option>
<option value="cancelled">Скасована</option>
</select>
<button onClick={handleStatusChange}>Зберегти статус</button>
</div>
</div>
</section>
);
}
export default RequestDetailPage;
```

Лістинг Б.7 – Файл RequestFormPage.jsx

```
import { useState } from "react";
import { useNavigate } from "react-router-dom";
import { apiRequest } from "../api";
function RequestFormPage() {
const navigate = useNavigate();
const [form, setForm] = useState({
title: "",
description: "",
categoryId: "",
priority: "normal",
});
const handleChange = (event) => {
const { name, value } = event.target;
setForm((prevForm) => ({
...prevForm,
[name]: value,
}));
};
const handleSubmit = async (event) => {
event.preventDefault();
try {
await apiRequest("/requests", "POST", form);
navigate("/requests");
} catch (error) {
console.error(error.message);
}
};
return (
<section>
```

```
<h1>Створення заявки</h1>
<p className="page-description">
Форма використовується координатором для додавання нової волонтерської заявки до системи.
</p>
<form className="request-form" onSubmit={handleSubmit}>
<label>Назва заявки</label>
<input
type="text"
name="title"
value={form.title}
onChange={handleChange}
required
/>
<label>Опис заявки</label>
<textarea
name="description"
value={form.description}
onChange={handleChange}
rows="5"
required
/>
<label>Категорія допомоги</label>
<select
name="categoryId"
value={form.categoryId}
onChange={handleChange}
required
>
<option value="">Оберіть категорію</option>
<option value="1">Гуманітарна допомога</option>
<option value="2">Транспорт</option>
<option value="3">Медична допомога</option>
<option value="4">Побутова допомога</option>
</select>
<label>Пріоритет</label>
<select name="priority" value={form.priority}
onChange={handleChange}>
<option value="low">Низький</option>
<option value="normal">Звичайний</option>
<option value="high">Високий</option>
</select>
<button type="submit">Створити заявку</button>
</form>
</section>
);
}
export default RequestFormPage;
```

Лістинг Б.8 – Файл styles.css

```
body {
margin: 0;
font-family: Arial, sans-serif;
background: #f4f6f8;
color: #222;
}
.login-page {
min-height: 100vh;
display: flex;
align-items: center;
justify-content: center;
}
.login-form {
width: 360px;
padding: 32px;
background: #fff;
border-radius: 12px;
box-shadow: 0 4px 18px rgba(0, 0, 0, 0.08);
}
.login-form h1 {
margin-bottom: 8px;
}
.login-form p {
margin-bottom: 24px;
color: #666;
}
label {
display: block;
margin-top: 16px;
margin-bottom: 6px;
font-weight: 600;
}
input,
select,
textarea {
width: 100%;
box-sizing: border-box;
padding: 10px 12px;
border: 1px solid #ccd2d8;
border-radius: 8px;
font-size: 14px;
}
button,
.primary-link {
display: inline-block;
margin-top: 20px;
padding: 10px 16px;
border: none;
border-radius: 8px;
background: #1f6feb;
color: #fff;
```

```
text-decoration: none;
cursor: pointer;
}
.app-layout {
display: flex;
min-height: 100vh;
}
.sidebar {
width: 240px;
padding: 24px;
background: #1f2937;
color: #fff;
}
.sidebar h2 {
margin-top: 0;
}
.sidebar nav {
display: flex;
flex-direction: column;
gap: 12px;
margin-top: 24px;
}
.sidebar a {
color: #dbeafe;
text-decoration: none;
}
.sidebar a.active {
font-weight: bold;
color: #fff;
}
.user-block {
margin-top: 40px;
display: flex;
flex-direction: column;
gap: 6px;
}
.content {
flex: 1;
padding: 32px;
}
.page-header {
display: flex;
align-items: center;
justify-content: space-between;
}
.page-description {
color: #666;
}
.stats-grid {
display: grid;
grid-template-columns: repeat(4, 1fr);
gap: 20px;
margin-top: 24px;
```

```
}
.stat-card,
.request-card,
.detail-card,
.request-form {
background: #fff;
padding: 20px;
border-radius: 12px;
box-shadow: 0 2px 12px rgba(0, 0, 0, 0.06);
}
.stat-card span {
color: #666;
}
.stat-card strong {
display: block;
margin-top: 8px;
font-size: 28px;
}
.filters {
display: flex;
gap: 16px;
margin: 24px 0;
}
.request-list {
display: flex;
flex-direction: column;
gap: 16px;
}
.request-card {
display: flex;
justify-content: space-between;
gap: 20px;
}
.request-card h3 {
margin-top: 0;
}
.request-meta {
display: flex;
flex-direction: column;
gap: 8px;
}
.status {
padding: 6px 10px;
border-radius: 20px;
background: #e5e7eb;
font-size: 13px;
}
.status-new {
background: #dbeafe;
}
.status-in_progress {
background: #fef3c7;
}
```

```
.status-completed {
background: #dcfce7;
}
.status-cancelled {
background: #fee2e2;
}
.status-control {
margin-top: 24px;
}
.empty-message,
.error-message {
color: #b91c1c;
}
```

ДОДАТОК В
СЛАЙДИ ПРЕЗЕНТАЦІЇ

Рисунок В.1 – Слайд 1

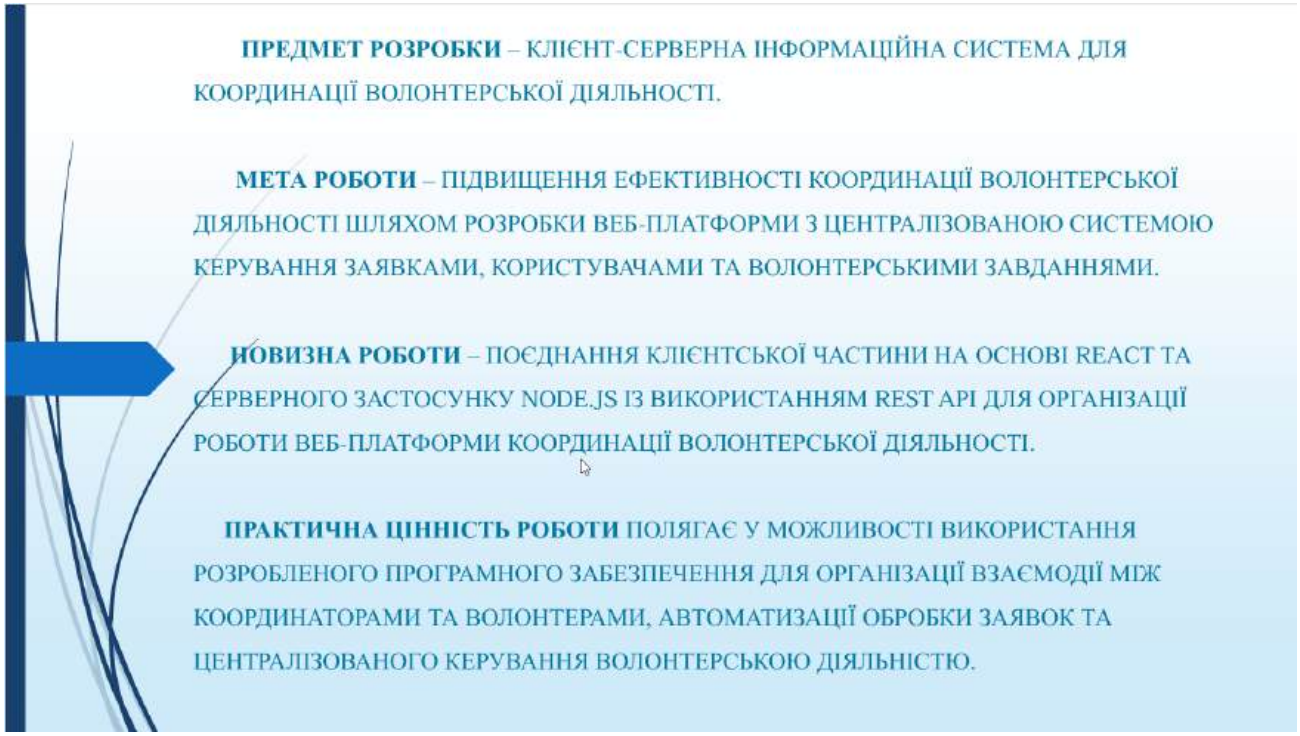

Рисунок В.2 – Слайд 2

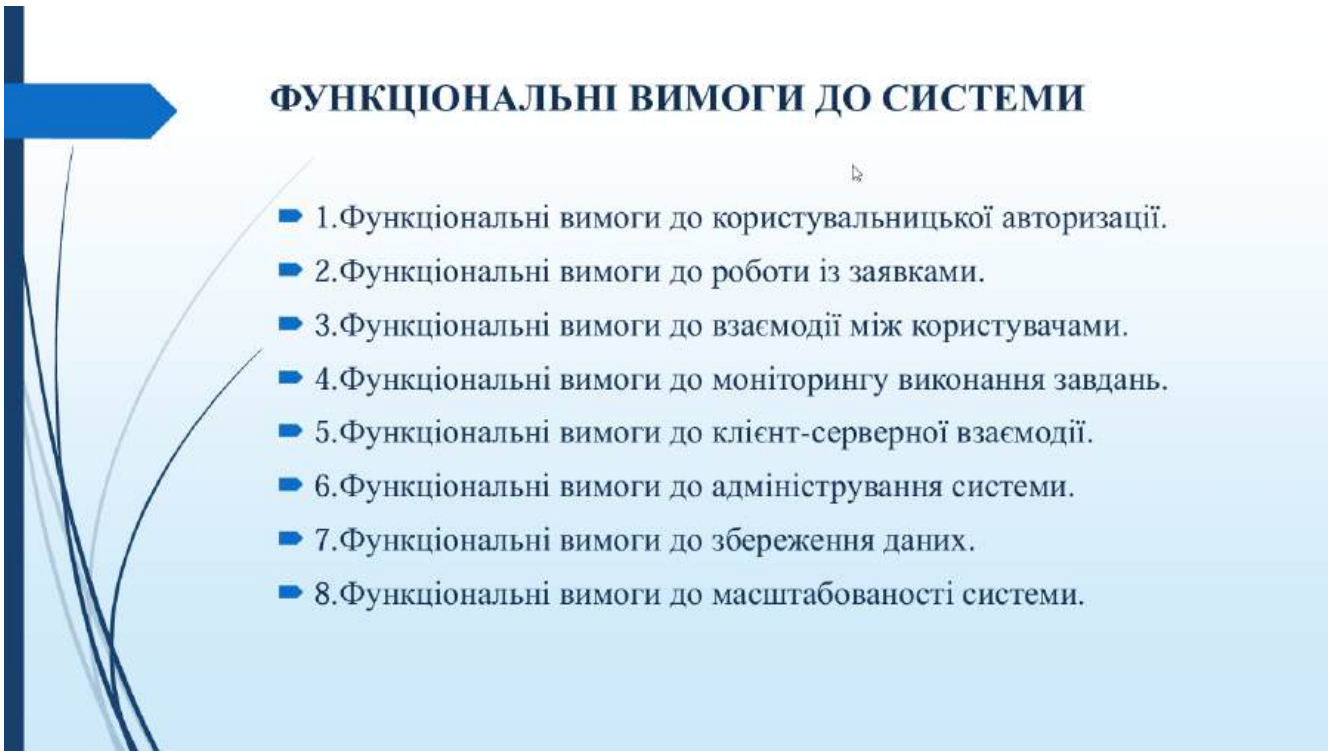

Рисунок В.3 – Слайд 3

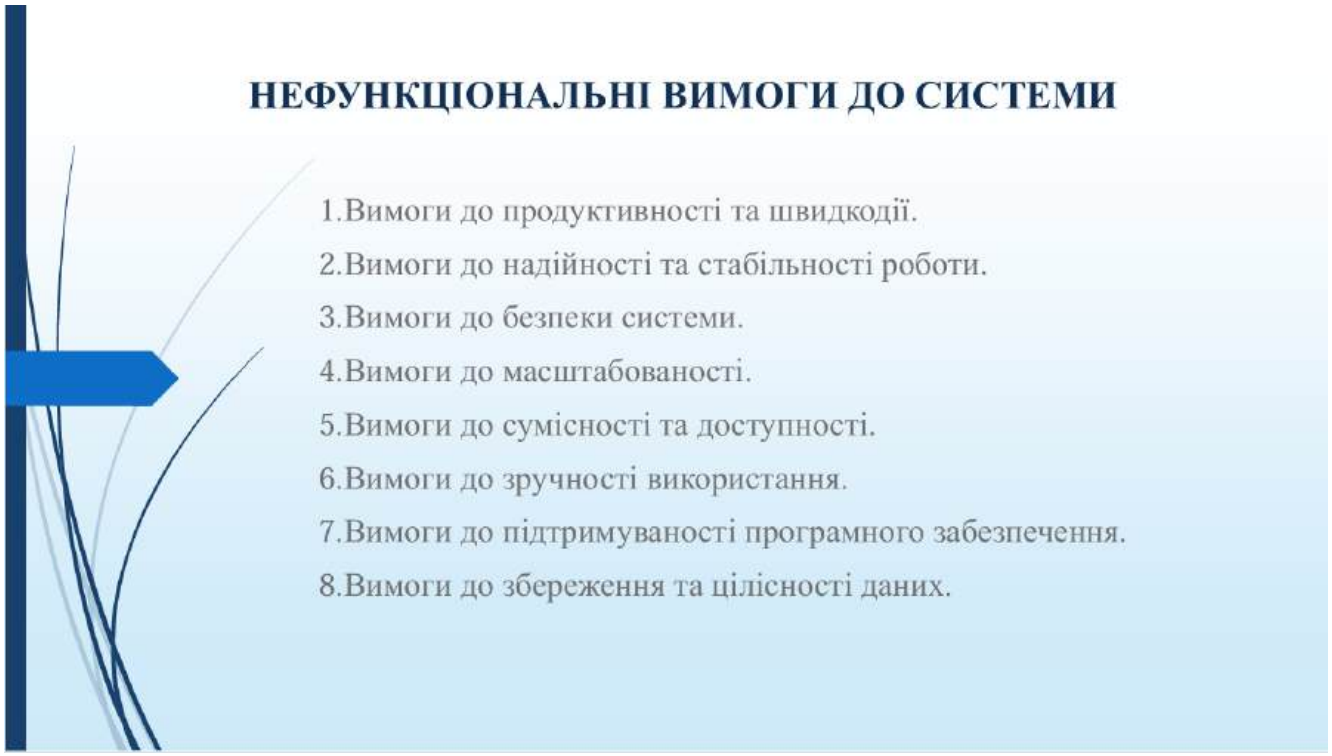

Рисунок В.4 – Слайд 4

ПОРІВНЯННЯ ТЕХНОЛОГІЙ РЕАЛІЗАЦІЙ ВЕБ-ПЛАТФОРМ

Критерій	React + Node.js + Express	Angular + Spring Boot	Vue.js + Django
Складність розробки	середня	висока	середня
Швидкість створення прототипу	висока	середня	висока
Єдина мова для клієнта і сервера	так	ні	ні
Підтримка REST API	висока	висока	висока
Масштабованість	висока	висока	середня
Кількість допоміжного коду	помірна	висока	середня
Зручність підтримки	висока	середня	середня

Рисунок В.5 – Слайд 5

СТРУКТУРА СЕРВЕРНОГО ПРОЄКТУ ВЕБ-ПЛАТФОРМИ КООРДИНАЦІЇ ВОЛОНТЕРСЬКОЇ ДІЯЛЬНОСТІ

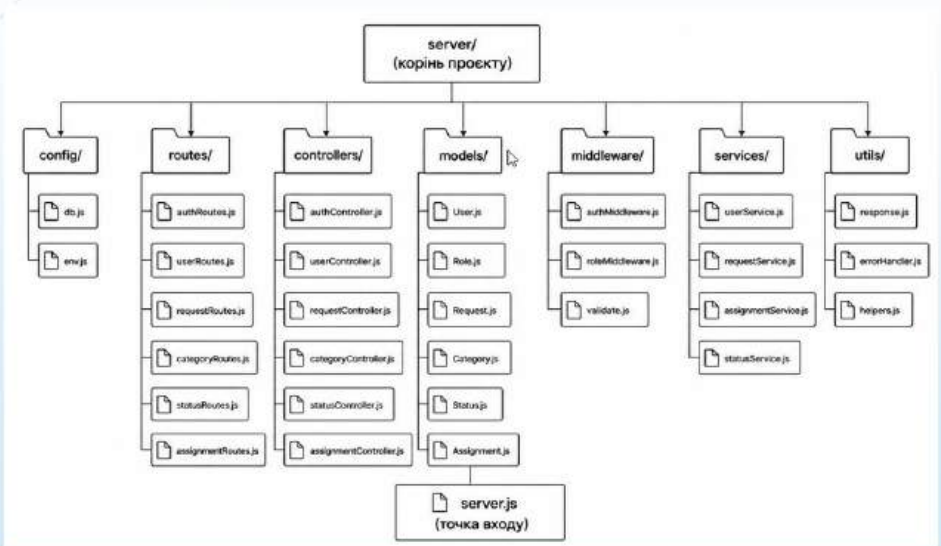

Рисунок В.6 – Слайд 6

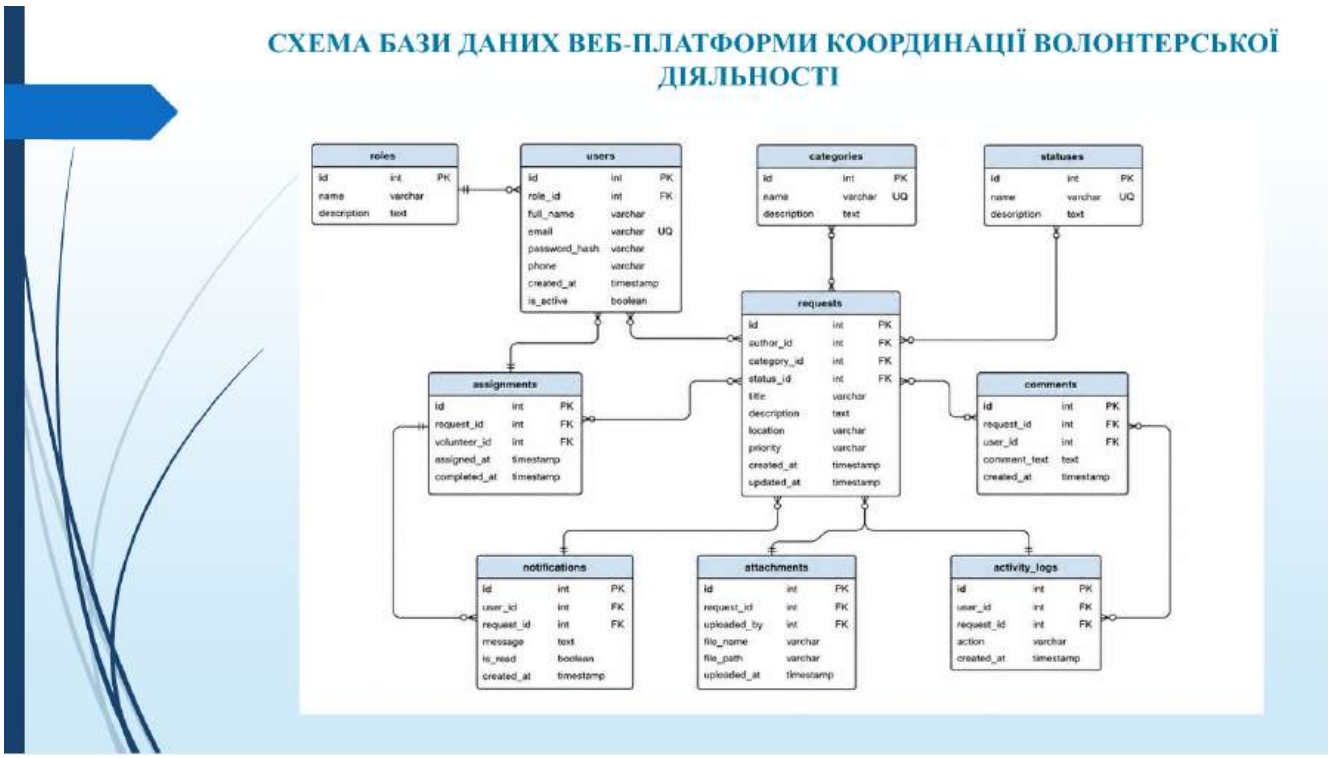

Рисунок В.7 – Слайд 7

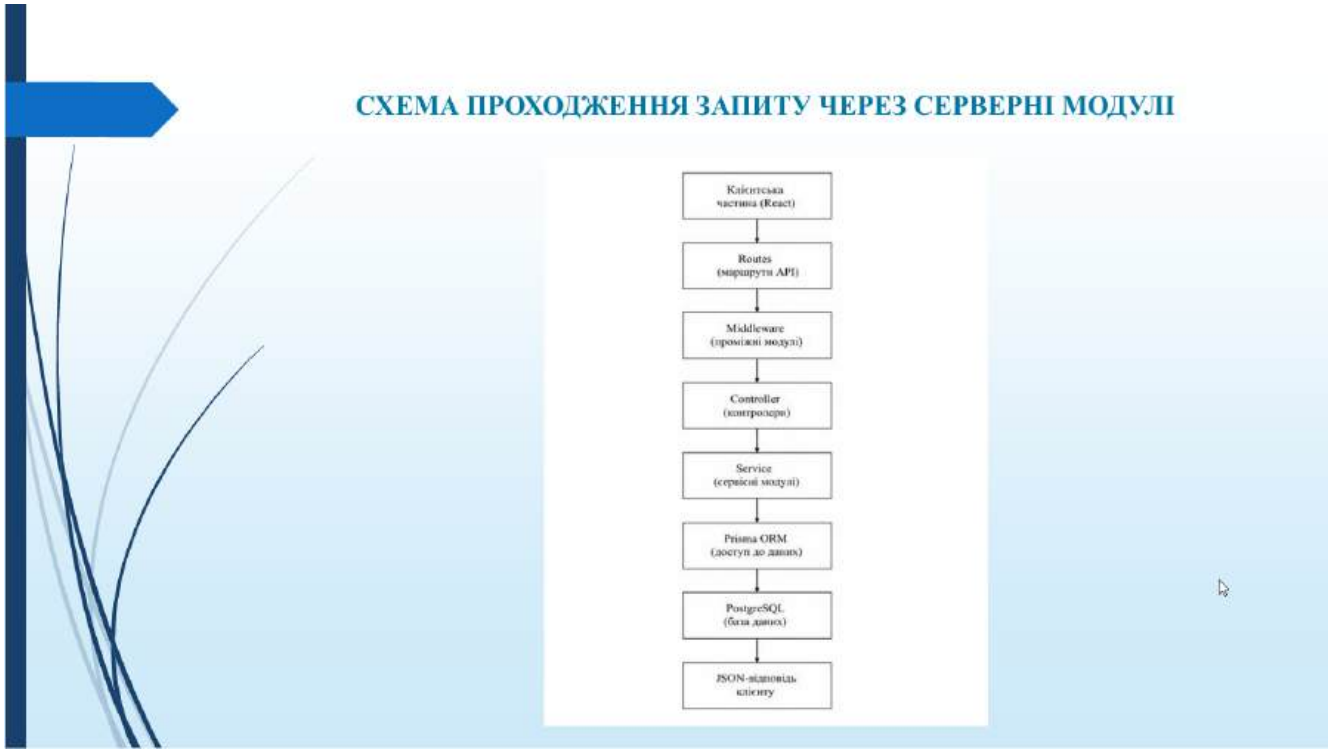

Рисунок В.8 – Слайд 8

ОСНОВНІ МАРШРУТИ REST API СЕРВЕРНОЇ ЧАСТИНИ

Маршрут	Призначення
/auth	Виконує операції реєстрації та авторизації користувачів
/users	Забезпечує отримання та зміну інформації про користувачів
/requests	Виконує створення, перегляд та зміну заявок
/assignments	Забезпечує роботу з призначенням волонтерів
/comments	Виконує створення та отримання коментарів
/notifications	Забезпечує відображення повідомлень користувача

Рисунок В.9 – Слайд 9

USE CASE ДІАГРАМА КЛІЄНТСЬКОЇ ЧАСТИНИ ВЕБ-ПЛАТФОРМИ

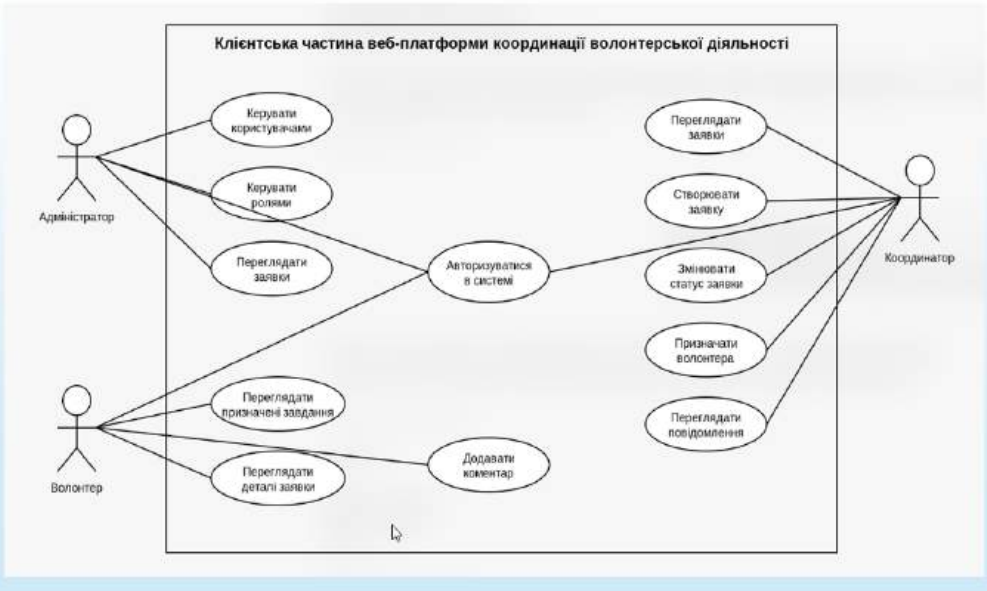

Рисунок В.10 – Слайд 10

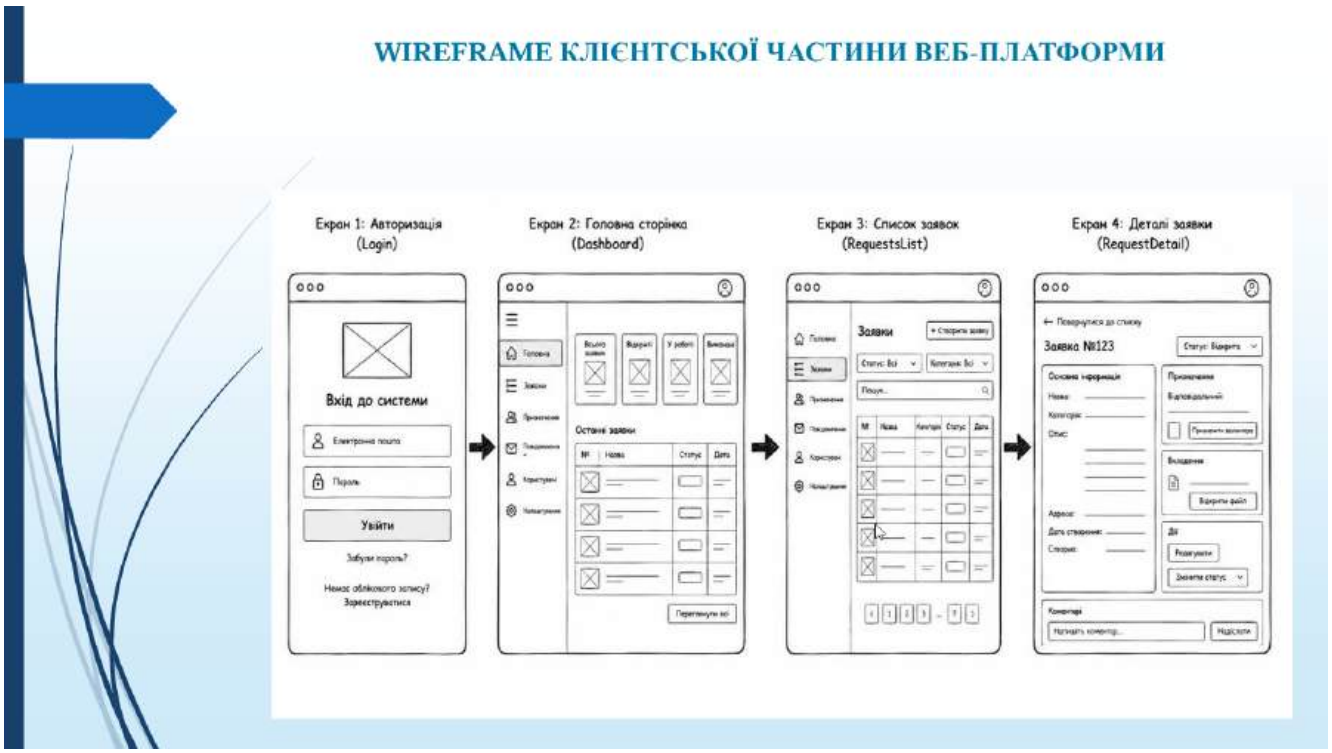

Рисунок В.11 – Слайд 11

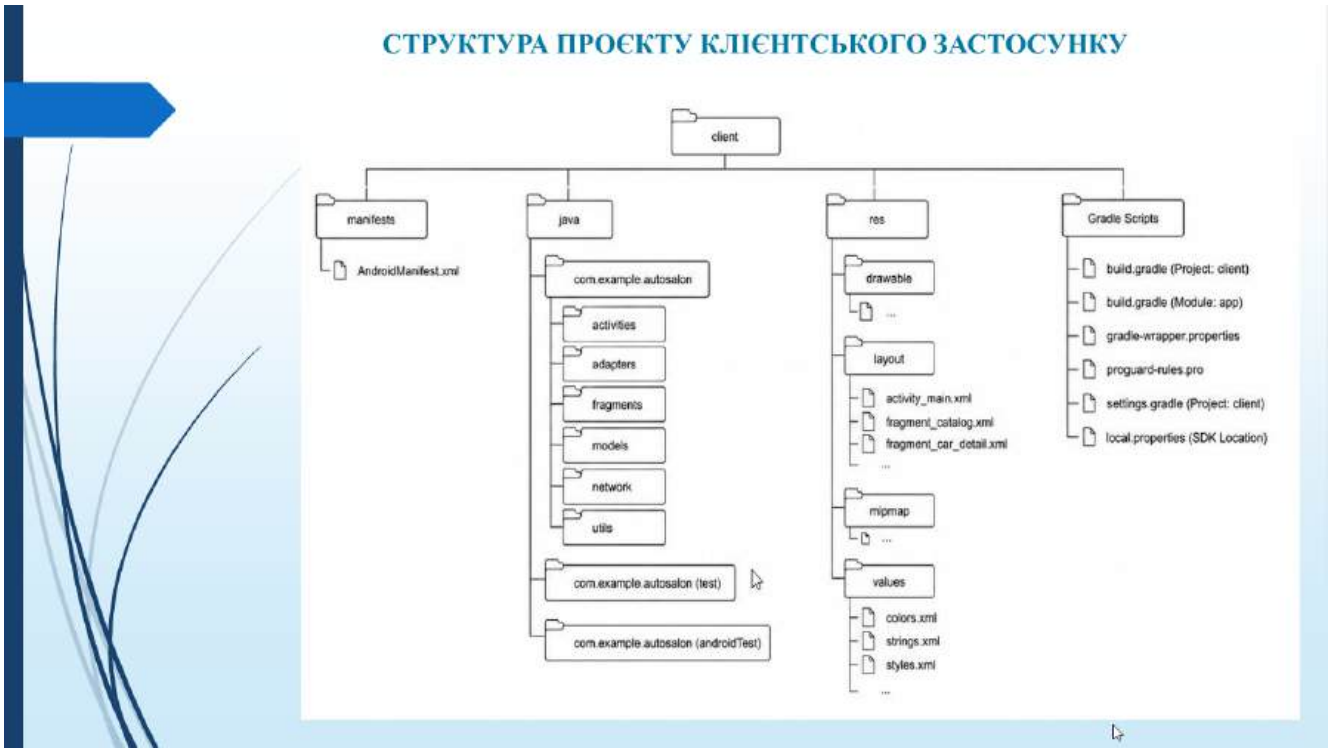

Рисунок В.12 – Слайд 12

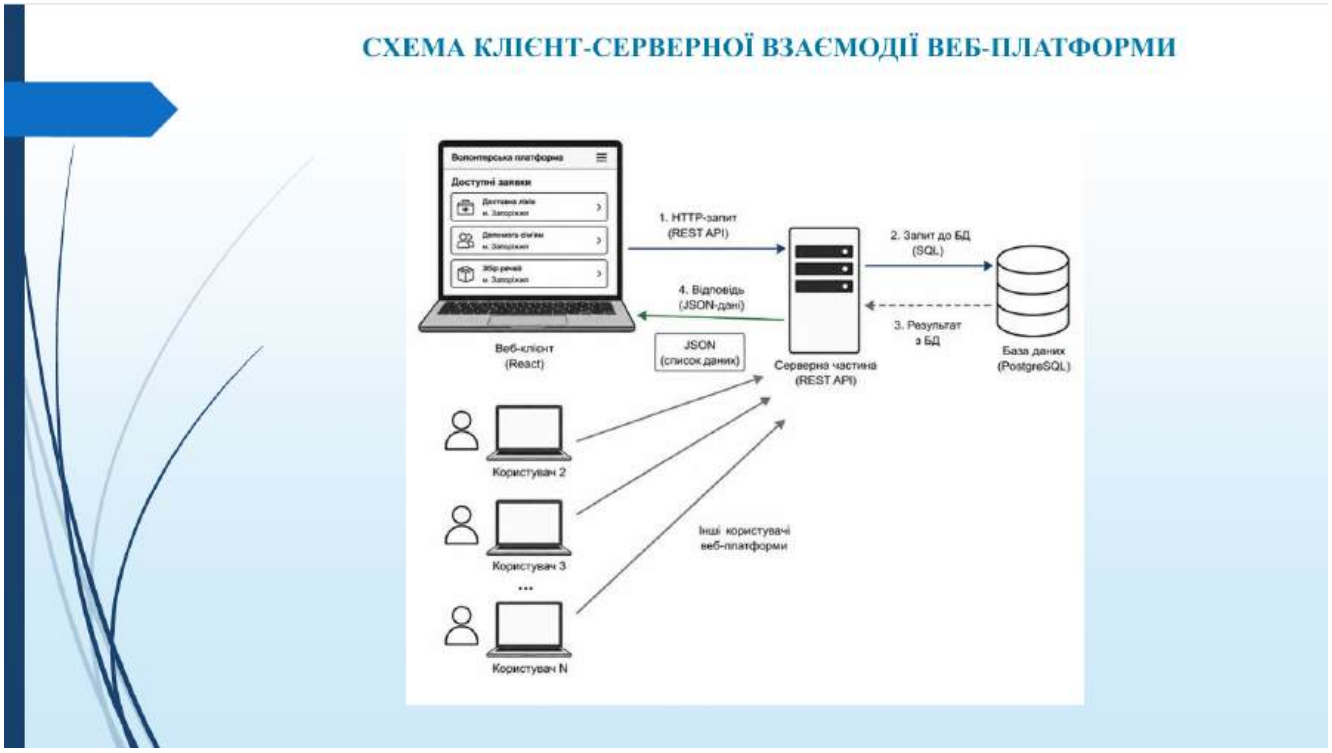

Рисунок В.13 – Слайд 13

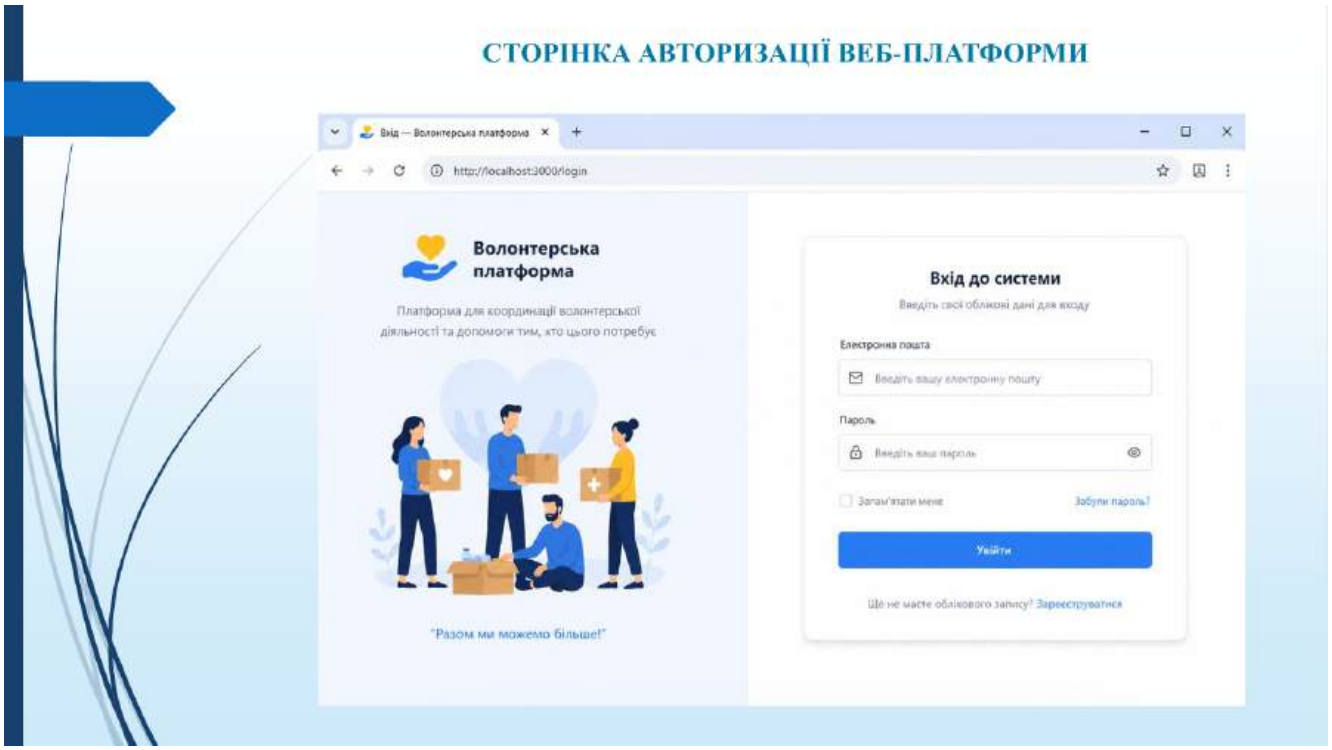

Рисунок В.14 – Слайд 14

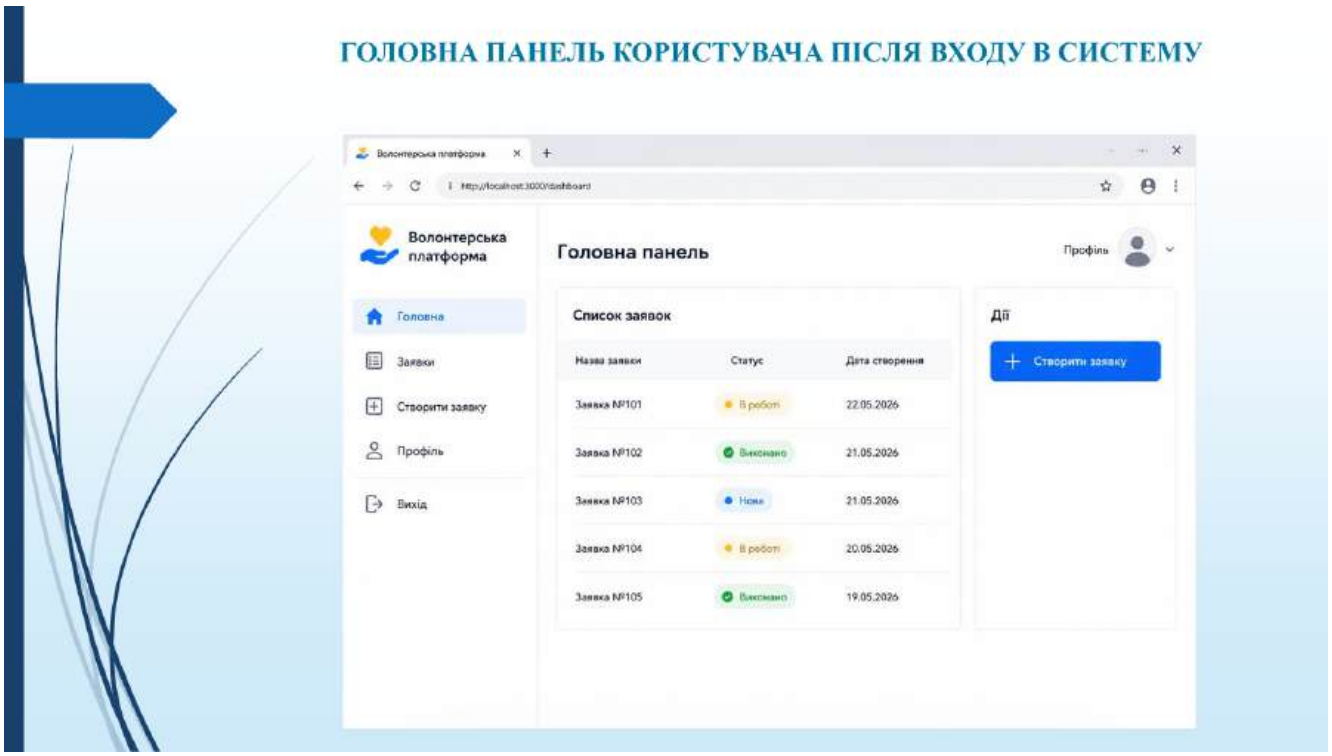

Рисунок В.15 – Слайд 15

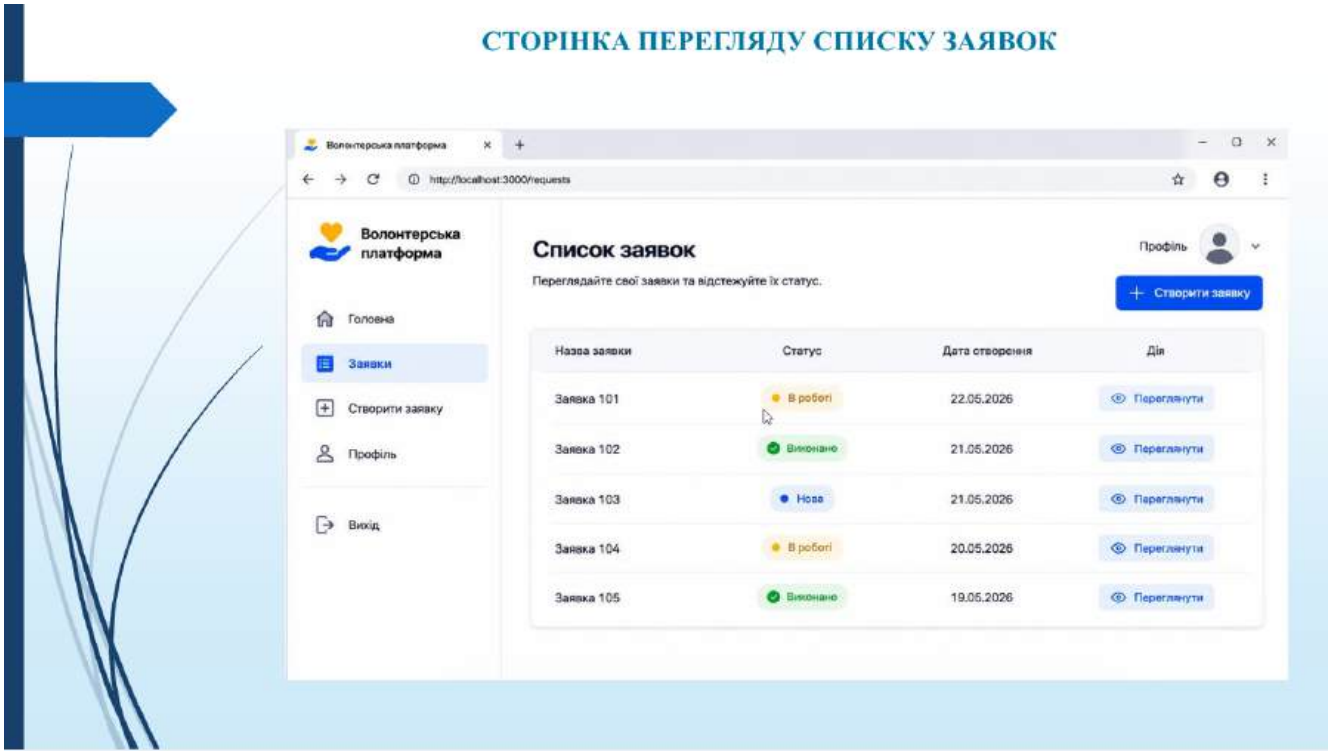

Рисунок В.16 – Слайд 16

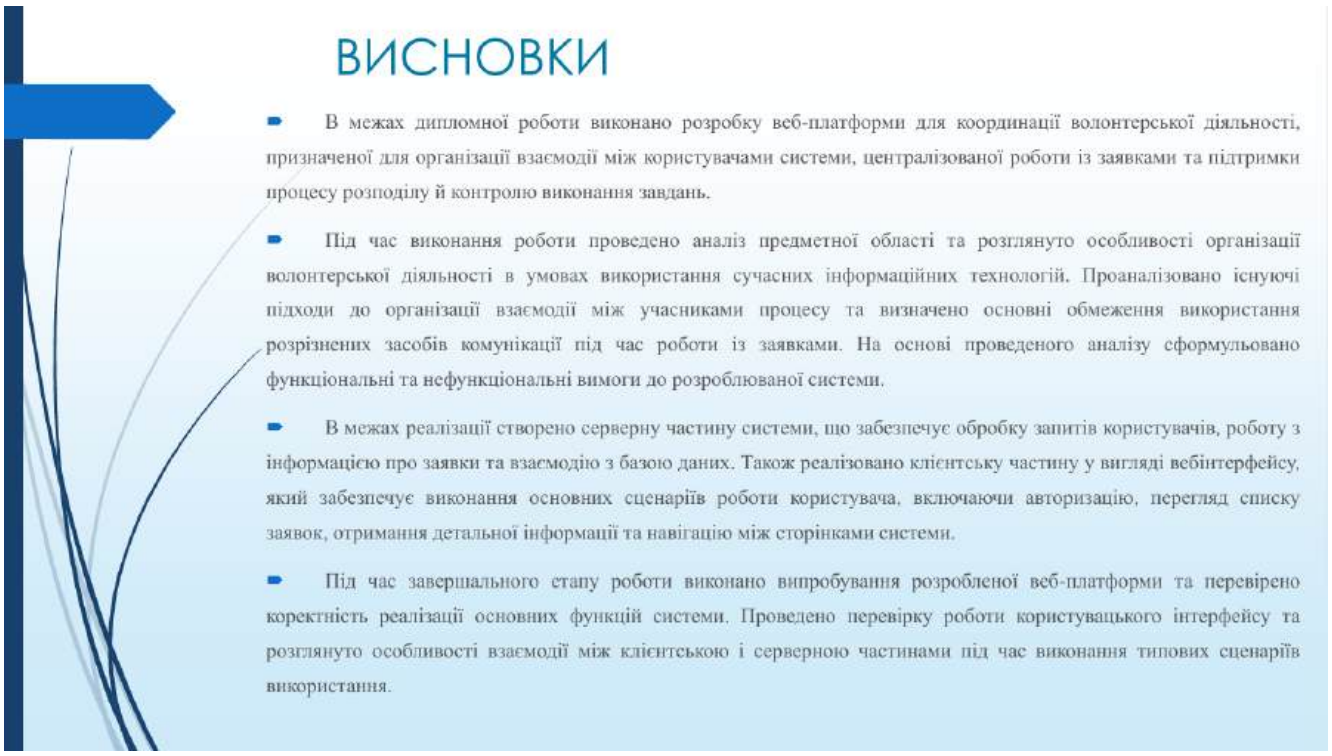

Рисунок В.17 – Слайд 17