НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
(повна назва інституту/факультету)

Кафедра інформатики та програмної інженерії
(повна назва кафедри)

«До захисту допущено»
Завідувач кафедри
_____ Едуард ЖАРІКОВ
(підпис) (ім'я прізвище)
"___"_____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення інформаційних систем»

спеціальності «121 Інженерія програмного забезпечення»

на тему: Вебзастосунок для для оптимізації електронних черг та прийому

Виконав студент IV курсу, групи ІП-13
(шифр групи)
Петров Ігор Ярославович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник ст.викл., д-р філософії, Стельмах О.П. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант ст.викл., Вітковська І.І. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент доцент, к.т.н., Сперкач М.О. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Київ – 2025

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 Інженерія програмного забезпечення

Освітньо-професійна програма – Інженерія програмного забезпечення інформаційних систем

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Едуард ЖАРІКОВ
(підпис) (ім'я прізвище)
"___"_____2025 р.

ЗАВДАННЯ
на дипломний проєкт студенту

Петрову Ігорю Ярославовичу
(прізвище, ім'я, по батькові)

1. Тема проєкту Вебзастосунок для для оптимізації електронних черг та прийому

керівник проєкту Стельмах Олександр Петрович, д-р філософії, ст.викл.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «23» травня 2025 р. №1705-с

2. Термін подання студентом проєкту «16» червня 2025 року

3. Вихідні дані до проєкту: технічне завдання

4. Зміст пояснювальної записки
1) Передпроєктне обстеження предметної області: постановка завдання дипломного проєктування, аналіз предметної області, аналіз існуючих рішень, аналіз та моделювання бізнес-процесів.
2) Розроблення вимог до програмного забезпечення: варіанти використання програмного забезпечення, розроблення функціональних вимог, розроблення нефункціональних вимог, аналіз системних вимог, аналіз економічних показників програмного забезпечення, постановка завдання на розробку програмного забезпечення.
3) Конструювання та розроблення програмного забезпечення: архітектура програмного забезпечення, архітектурні рішення та обґрунтування вибору засобів розробки, конструювання програмного забезпечення, аналіз безпеки даних.
4) Аналіз якості та тестування програмного забезпечення: аналіз якості ПЗ, опис

процесів тестування, опис контрольного прикладу.
5) Розгортання та супровід програмного забезпечення: розгортання програмного забезпечення, супровід програмного забезпечення.

5. Перелік графічного матеріалу
1) Схема структурна варіантів використань
2) Схема структурна компонентів програмного забезпечення
3) Креслення вигляду екранних форм

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання «15» березня 2025 року

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1	Вивчення рекомендованої літератури	15.03.2025	
2	Аналіз існуючих методів розв’язання задачі	21.04.2025	
3	Постановка та формалізація задачі	22.04.2025	
4	Розробка інформаційного забезпечення	23.04.2025	
5	Алгоритмізація задачі	28.04.2024	
6	Обґрунтування вибору використаних технічних засобів	02.05.2025	
7	Розробка програмного забезпечення	10.05.2025	
8	Налагодження програми	13.05.2025	
9	Виконання графічних документів	27.05.2025	
10	Оформлення пояснювальної записки	01.06.2025	
11	Подання ДП на попередній захист	02.06.2025	
12	Подання ДП рецензенту	10.06.2025	
13	Подання ДП на основний захист	16.06.2025	

Студент _____ Ігор ПЕТРОВ
(підпис) (ініціали, прізвище)

Керівник _____ Олександр СТЕЛЬМАХ
(підпис) (ініціали, прізвище)

АНОТАЦІЯ

Пояснювальна записка дипломного проєкту складається з п’яти розділів, містить 65 таблиць, 40 рисунків та 15 джерел – загалом 101 сторінка.

Дипломний проєкт присвячений розробці вебзастосунку для організації електронних черг та порядку прийому для підтримки діяльності організацій.

Мета розробки – удосконалення процесів взаємодії між клієнтами та організаціями шляхом впровадження механізмів керування живими чергами і чергами за попереднім записом.

У першому розділі сформульовано завдання дипломного проєктування, виконано аналіз предметної області, розглянуто існуючі рішення та проведено моделювання бізнес-процесів.

У другому розділі визначено вимоги до програмного забезпечення: описано варіанти використання ПЗ, функціональні й нефункціональні вимоги, проаналізовано системні потреби та економічні показники, а також сформульовано завдання на розробку.

Третій розділ присвячено конструюванню та безпосередній розробці ПЗ: викладено його архітектуру, обґрунтовано вибір засобів розробки, описано етапи конструювання і здійснено аналіз безпеки даних.

У четвертому розділі проведено оцінку якості та тестування ПЗ: встановлено критерії якості, описано процеси тестування і наведено контрольний приклад.

П’ятий розділ охоплює розгортання та супровід програмного забезпечення: розглянуто процедури впровадження ПЗ і заходи з його подальшої підтримки.

КЛЮЧОВІ СЛОВА: ВЕБЗАСТОСУНОК, VIRTUAL QUEUING, DYNAMIC QUEUE CONTROL, SERVICE MANAGEMENT, RESOURCE OPTIMIZATION.

ABSTRACT

The explanatory note of the diploma project consists of five sections, contains 65 tables, 40 figures and 15 sources – in total 101 pages.

The diploma project is dedicated to the development of a web application for organizing electronic queues and appointment scheduling to support the operations of organizations.

The purpose of the development is to improve the processes of interaction between clients and organizations by introducing mechanisms for managing live queues and appointment-based queues.

In the first chapter, the objectives of the diploma project are formulated, the domain is analyzed, existing solutions are reviewed, and business process modeling is performed.

In the second chapter, the software requirements are defined: use cases of the software are described, functional and non-functional requirements are outlined, system needs and economic indicators are analyzed, and development tasks are formulated.

In the third chapter, the software's design and implementation are covered: its architecture is presented, the choice of development tools is justified, the design stages are described, and a data security analysis is performed.

In the fourth chapter, the software's quality evaluation and testing are conducted: quality criteria are established, testing processes are described, and a control example is provided.

In the fifth chapter, deployment and maintenance of the software are covered: the procedures for implementing the software and the measures for its ongoing support are reviewed.

KEYWORDS: WEB APPLICATION, VIRTUAL QUEUING, DYNAMIC QUEUE CONTROL, SERVICE MANAGEMENT, RESOURCE OPTIMIZATION.

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4		Завдання на дипломний проєкт	2	
2	А4	КПІ.ІП-1328.045440.01.91	Технічне завдання	23	
3	А4	КПІ.ІП-1328.045440.02.81	Пояснювальна записка	101	
4	А4	КПІ.ІП-1328.045440.03.12	Текст програми	35	
5	А4	КПІ.ІП-1328.045440.04.51	Програма та методика тестування	6	
6	А4	КПІ.ІП-1328.045440.05.34	Керівництво користувача	17	
7	А3	КПІ.ІП-1328.045440.06.99	Графічний матеріал	4	

Змін.	Арк.	№ докум.	Підп.	Дата	КПІ.ІП-1328.045440.00.90		
Розроб.		Петров І.Я.			Відомість дипломного проєкту	Літ.	Аркуш 1 / Аркушів
Перевір.		Стельмах О.П.				КПІ ім. Ігоря Сікорського ФІОТ каф. ІПІ гр. ІП-13	
Н.контр.		Вітковська І.І.					
Затв.		Жаріков Е.В.					

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”
Завідувач кафедри
_____ Едуард ЖАРІКОВ
“___” _____ 2025 р.

ВЕБЗАСТОСУНОК ДЛЯ ОПТИМІЗАЦІЇ ЕЛЕКТРОННИХ ЧЕРГ ТА ПРИЙОМУ

Технічне завдання

КПІ.ІП-1328.045440.01.91

“ПОГОДЖЕНО”
Керівник проєкту:
_____ Олександр СТЕЛЬМАХ

Нормоконтроль:
_____ Ірина ВІТКОВСЬКА

Виконавець:
_____ Ігор ПЕТРОВ

Київ – 2025

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: Вебзастосунок для оптимізації електронних черг та прийому.

Галузь застосування:

Наведене технічне завдання поширюється на розробку вебзастосунку E-queue, котрий використовується для створення та управління віртуальними чергами та призначений для використання організаціями, що обслуговують своїх клієнтів офлайн та їх клієнтами.

2 ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки E-queue є завдання на дипломне проєктування, затверджене кафедрою інформатики та програмної інженерії Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3 ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для автоматизації процесів електронного прийому клієнтів, включно з формуванням і керуванням чергами в режимі реального часу та бронюванням послуг із можливістю вказувати необхідні документи.

Мета розробки – удосконалення процесів взаємодії між клієнтами та організаціями шляхом впровадження механізмів керування живими чергами і чергами за попереднім записом.

4 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Вимоги до функціональних характеристик

Програмне забезпечення повинно забезпечувати виконання наступних основних функції:

4.1.1 Користувацького інтерфейсу

– прототип екранної форми посадкової сторінки (рисунок 4.1);

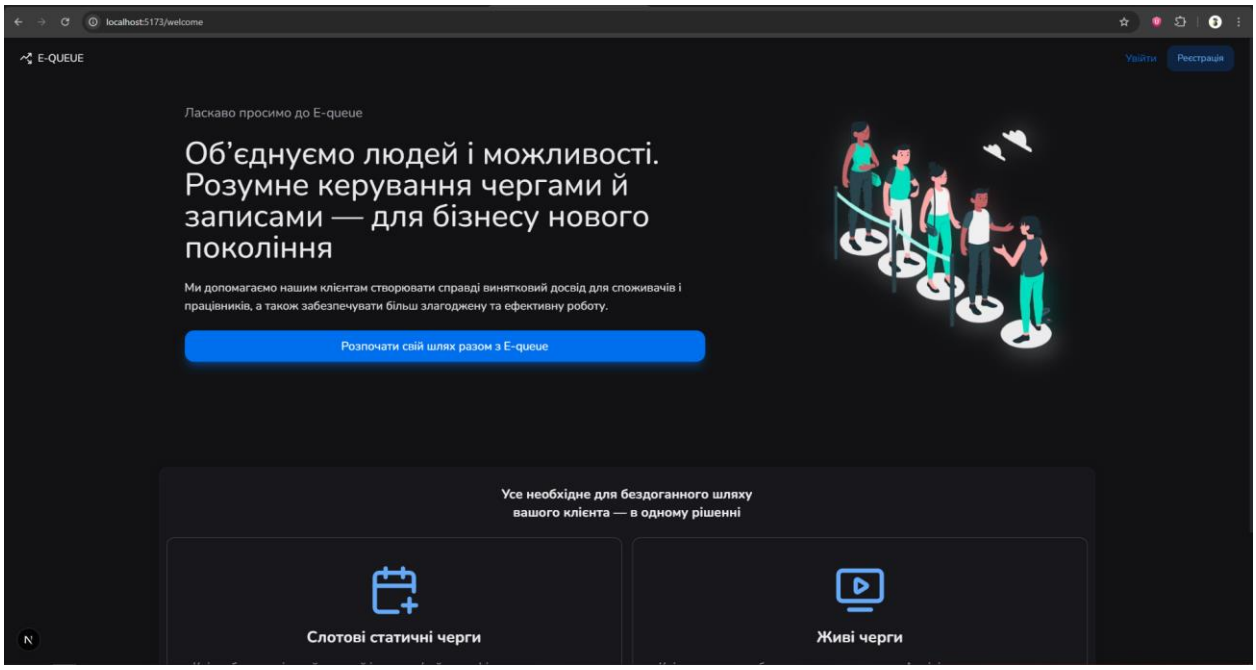


Рисунок 4.1 – Прототип посадкової сторінки

– прототип екранної форми авторизації (рисунок 4.2);

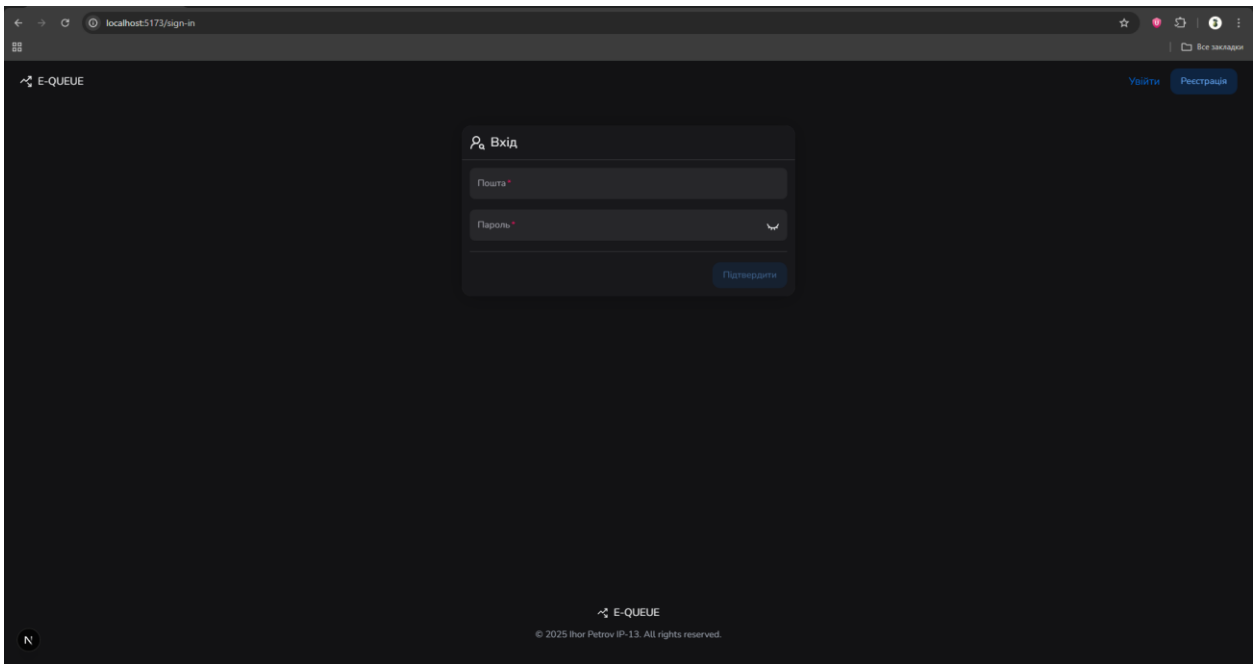


Рисунок 4.2 – Прототип сторінки авторизації

– прототип екранної форми реєстрації користувача (рисунок 4.3);

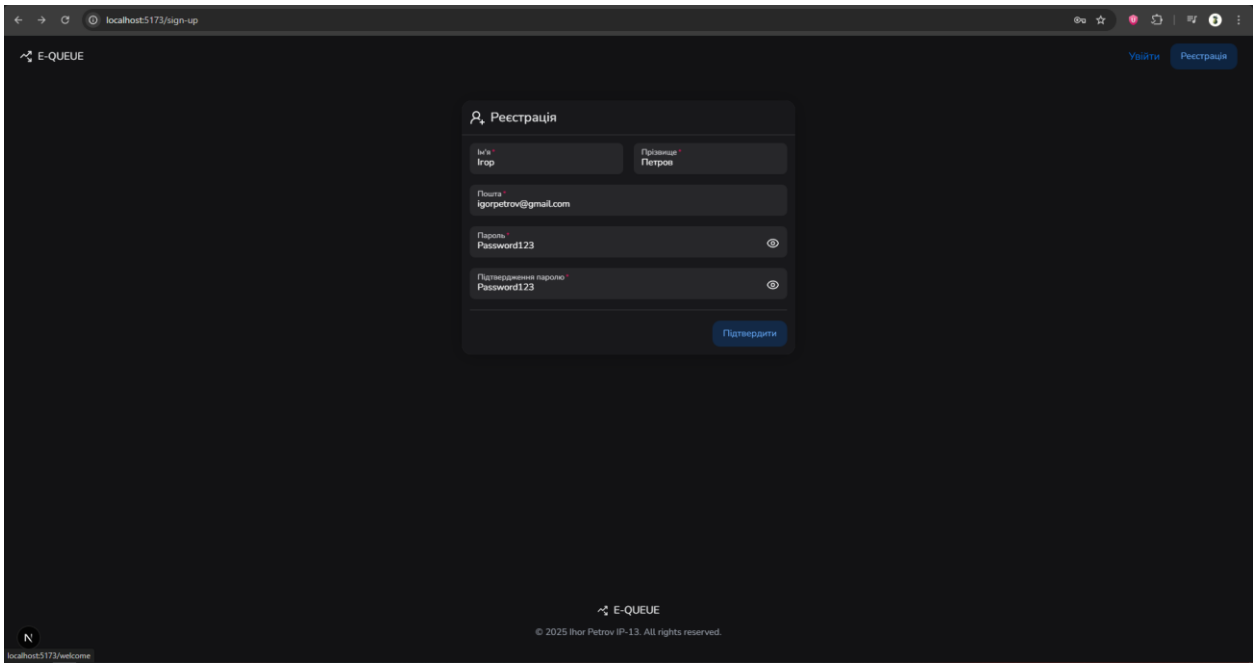


Рисунок 4.3 – Прототип сторінки реєстрації

– прототип екранної форми створення нової організації (рисунок 4.4);

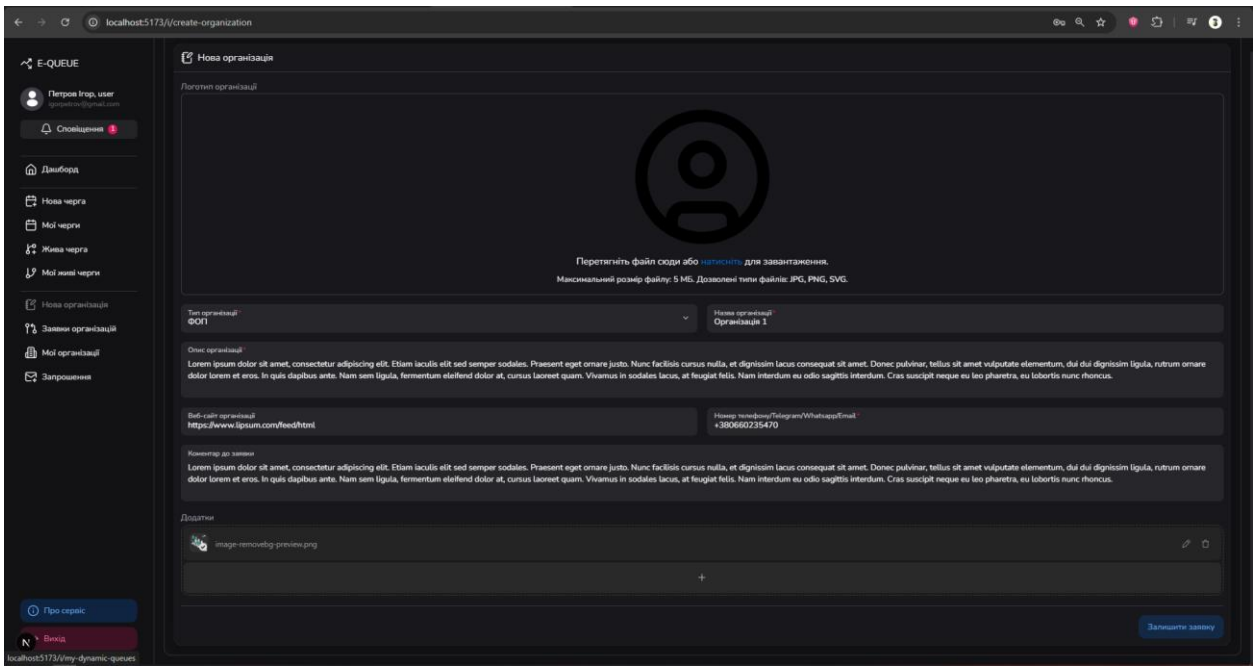


Рисунок 4.4 – Прототип сторінки створення організації

– прототип екранної форми заявок на організацію (рисунок 4.5);

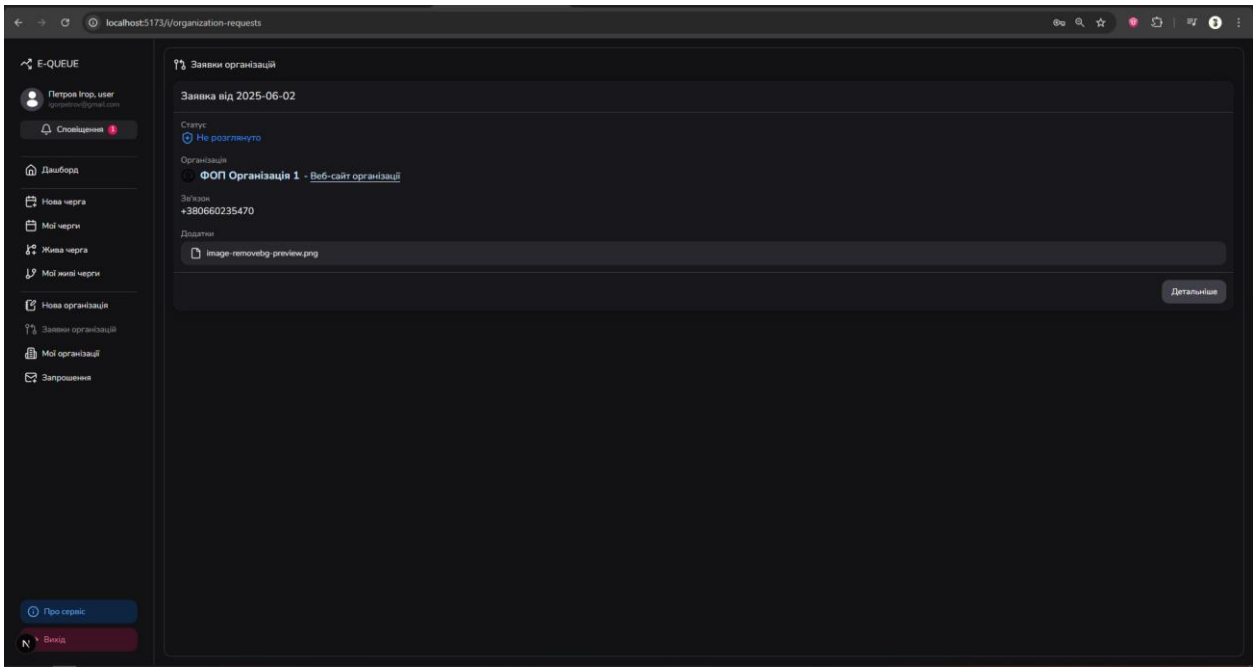


Рисунок 4.5 – Прототип сторінки заявок на організацію

– прототип екранної форми детальної інформації заявки на організацію (рисунок 4.6);

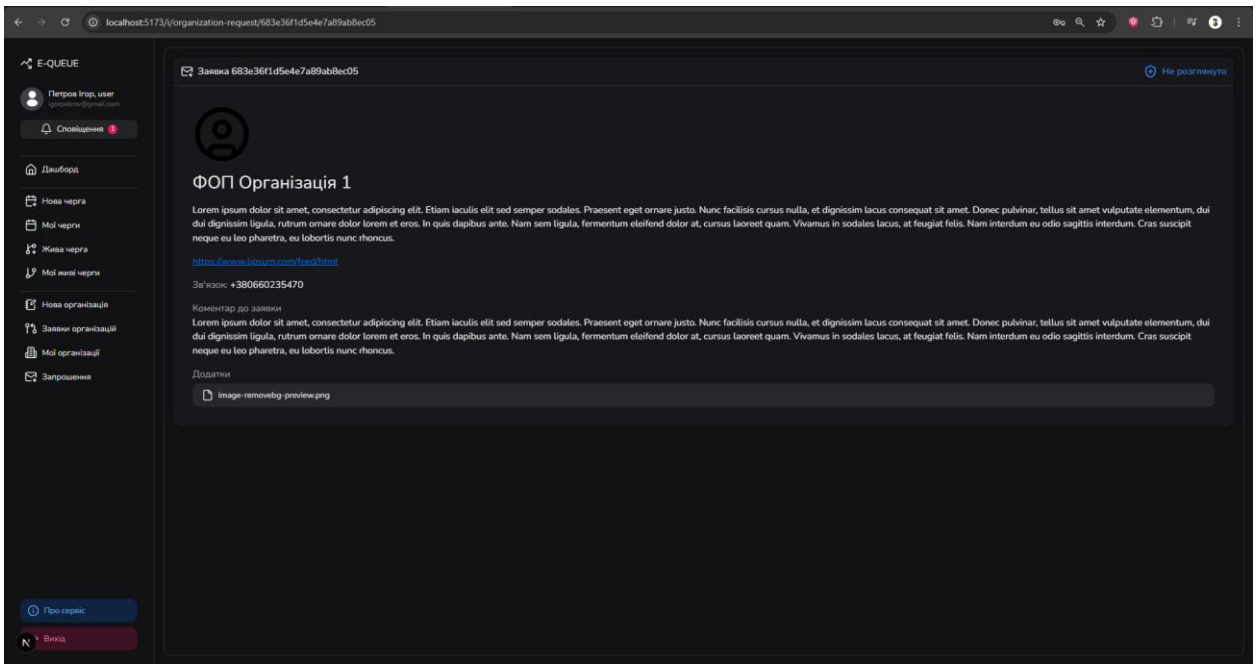


Рисунок 4.6 – Прототип сторінки детальної інформації заявки на організацію

– прототип екранної форми чату з адміністрацією (рисунок 4.7);

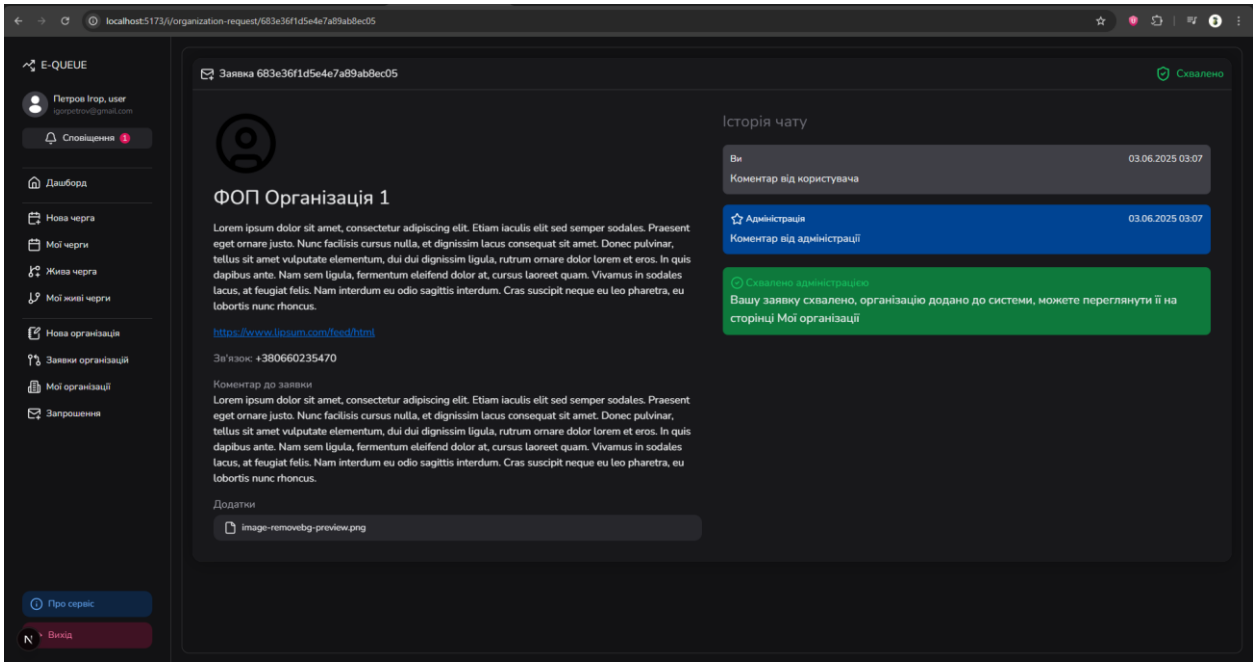


Рисунок 4.7 – Прототип чату з адміністрацією

– прототип екранної форми організацій користувача (рисунок 4.8);

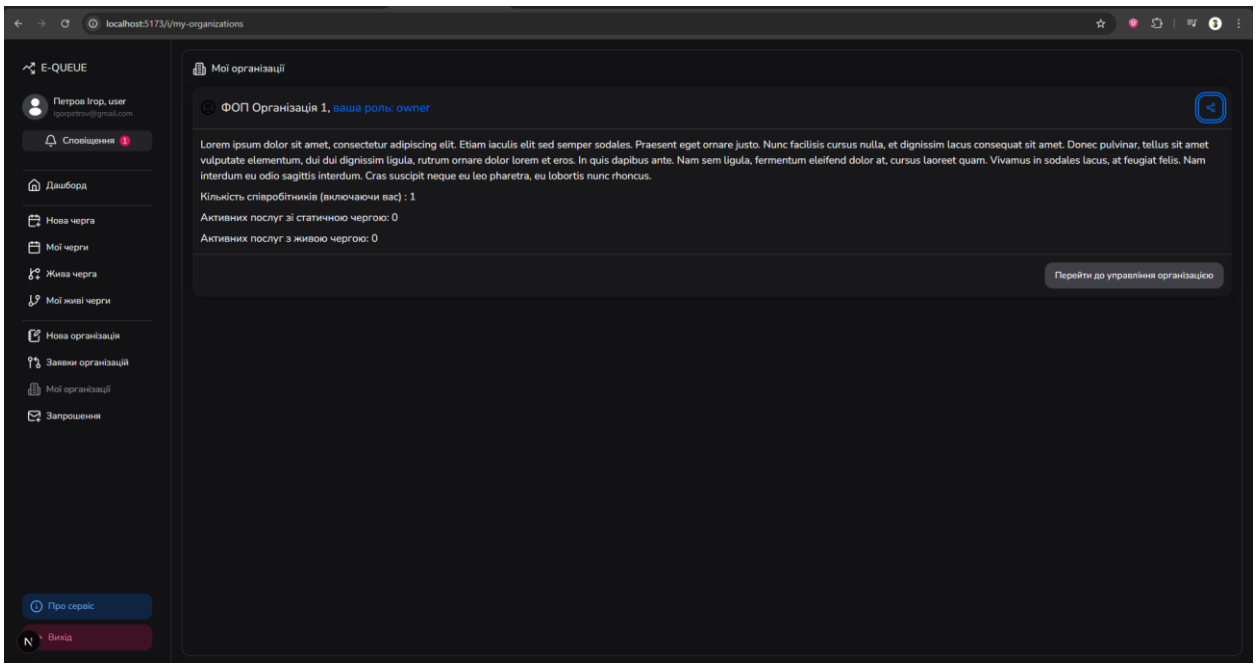


Рисунок 4.8 – Прототип сторінки організацій користувача

– прототип екранної форми створення статичної черги (рисунок 4.9);

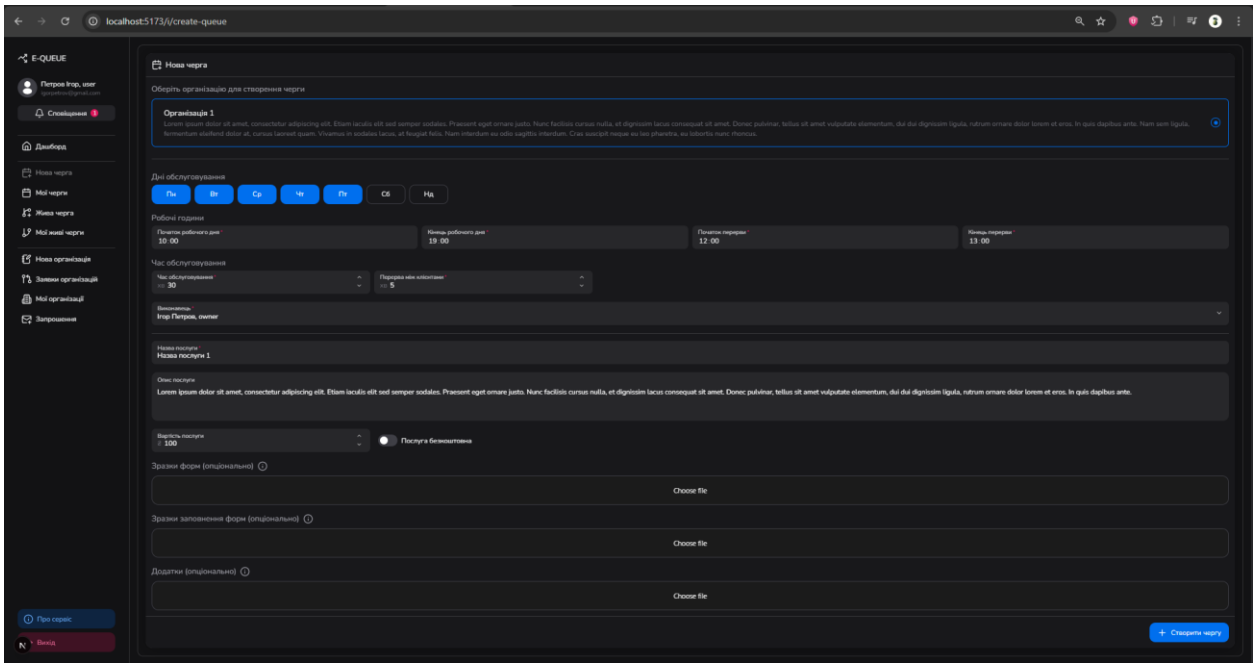


Рисунок 4.9 – Прототип сторінки створення статичної черги

– прототип екранної форми переліку статичних черг, де користувач є виконавцем (рисунок 4.10);

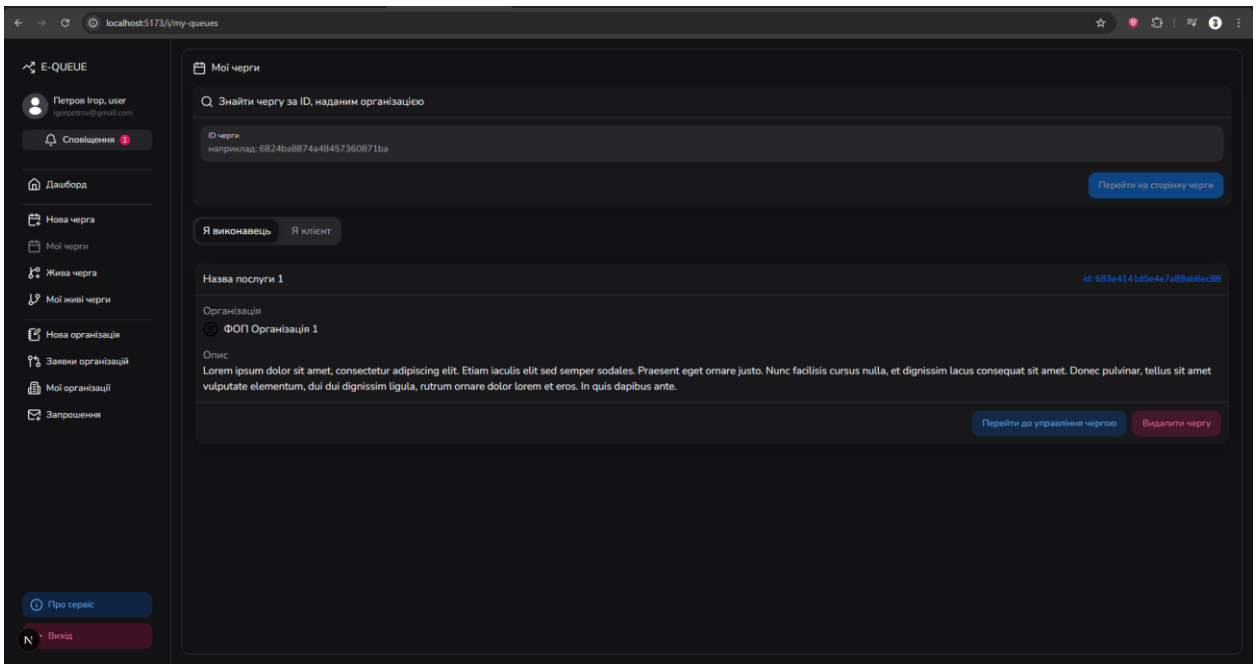


Рисунок 4.10 – Прототип сторінки переліку статичних черг користувача, де він є виконавцем

– прототип екранної форми детальної інформації статичної черги з точки зору клієнта (рисунок 4.11);

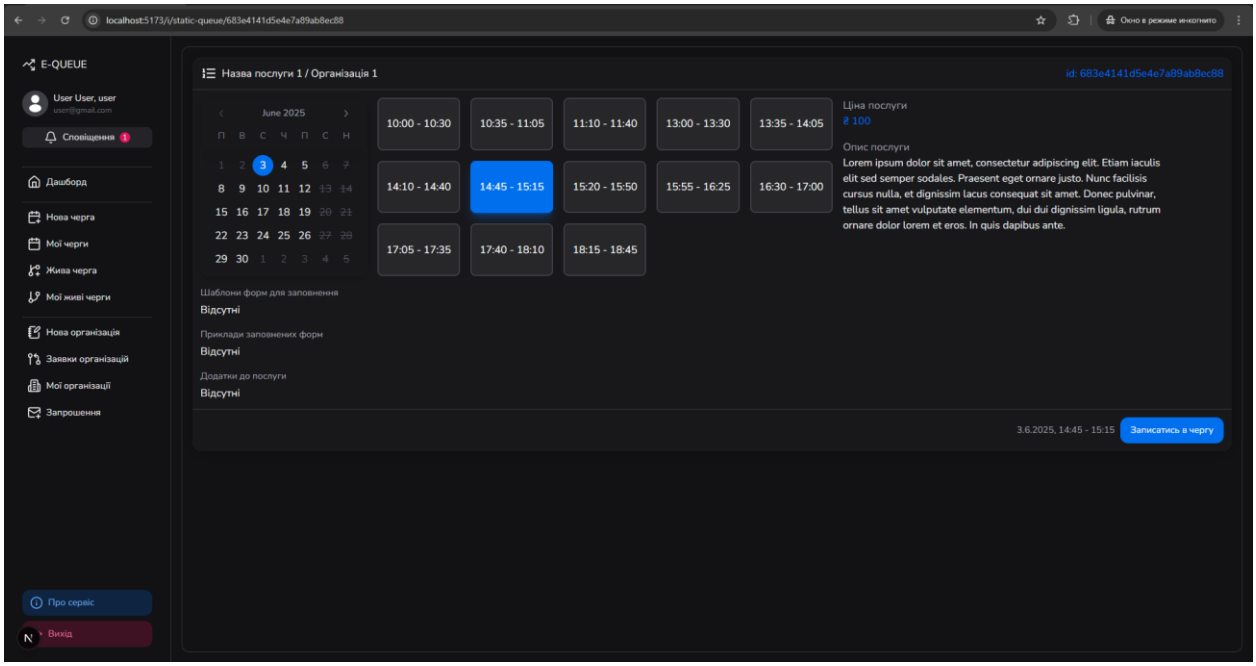


Рисунок 4.11 – Прототип сторінки детальної інформації статичної черги з точки зору клієнта

– прототип екранної форми переліку статичних черг, де користувач є клієнтом (рисунок 4.12);

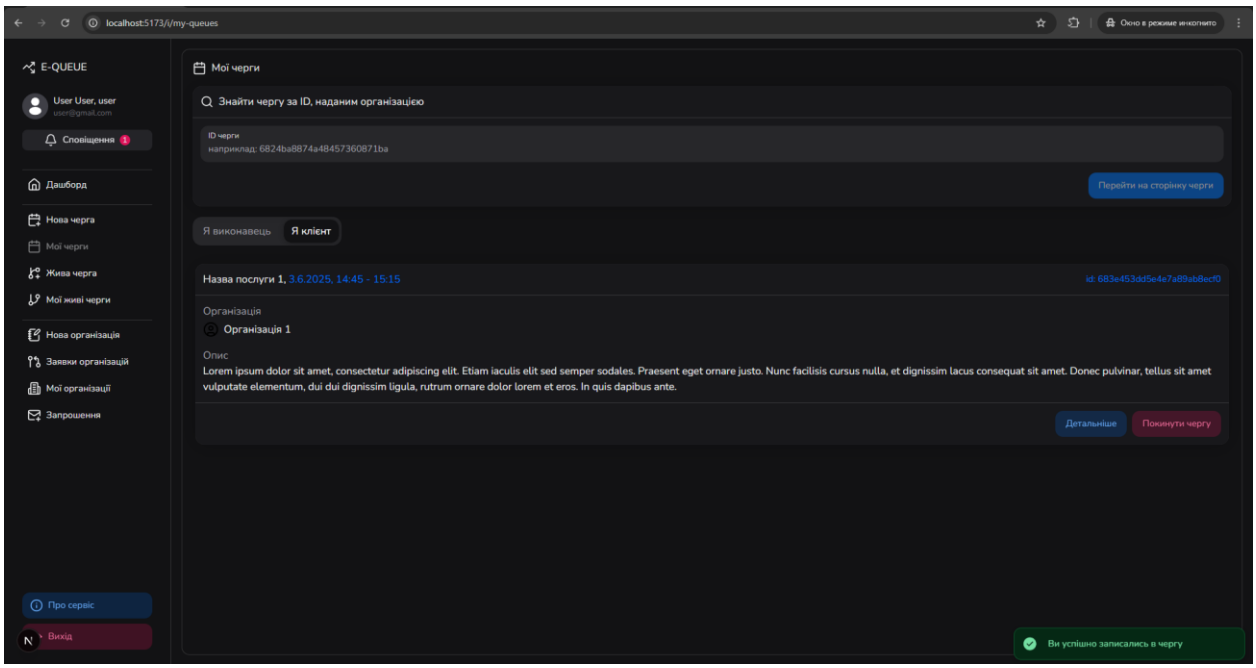


Рисунок 4.12 – Прототип сторінки переліку статичних черг, де користувач є клієнтом

– прототип екранної форми детальної інформації статичної черги з точки зору виконавця (рисунок 4.13);

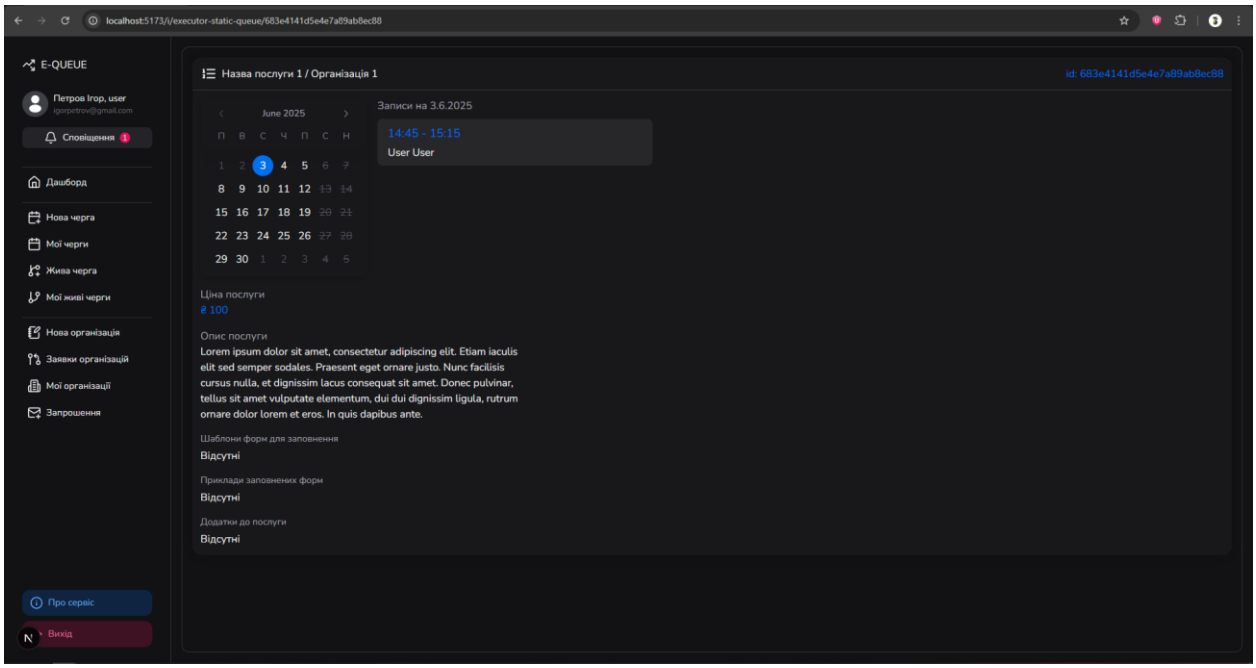


Рисунок 4.13 – Прототип сторінки управління статичною чергою

– прототип екранної форми створення живої черги (рисунок 4.14);

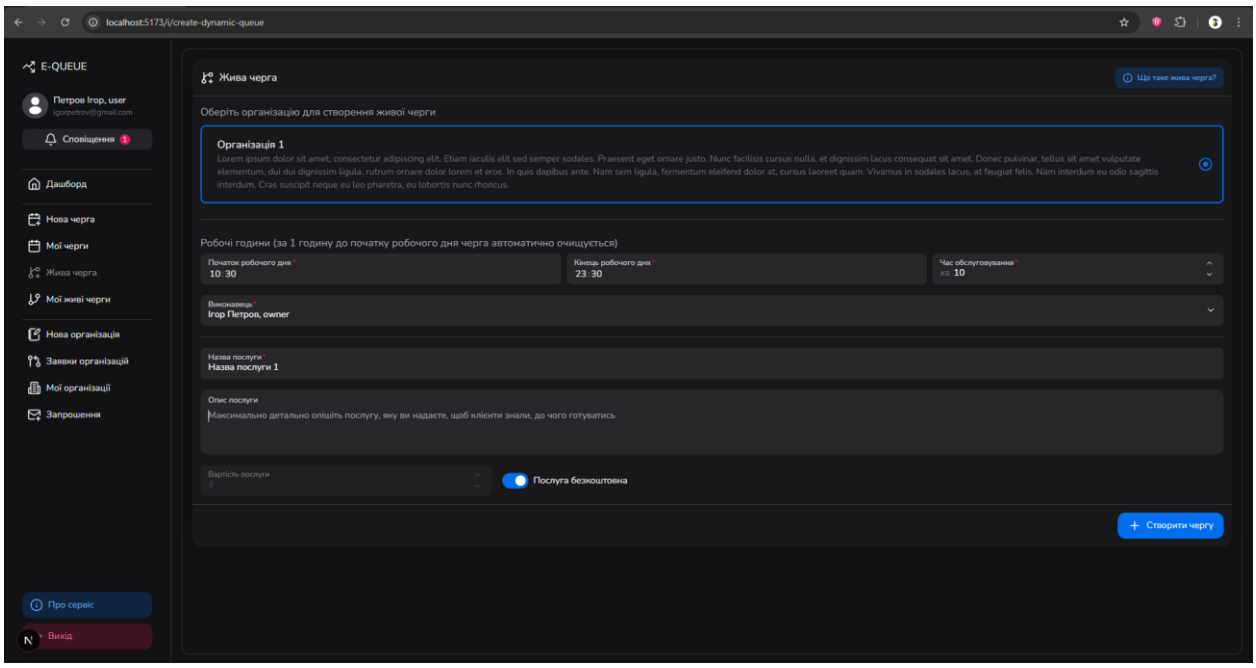


Рисунок 4.14 – Прототип сторінки створення живої черги

– прототип екранної форми переліку живих черг, де користувач є виконавцем (рисунок 4.14);

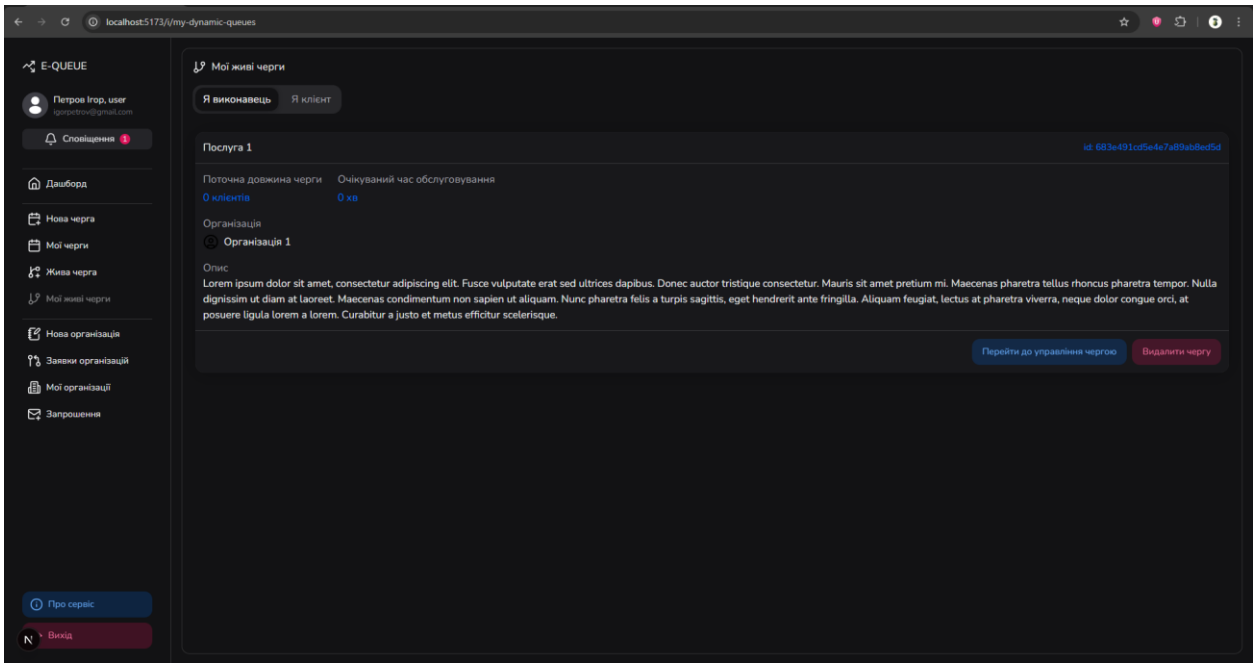


Рисунок 4.14 – Прототип сторінки переліку живих черг користувача, де він є виконавцем

– прототип екранної форми детальної інформації живої черги з точки зору виконавця (рисунок 4.15);

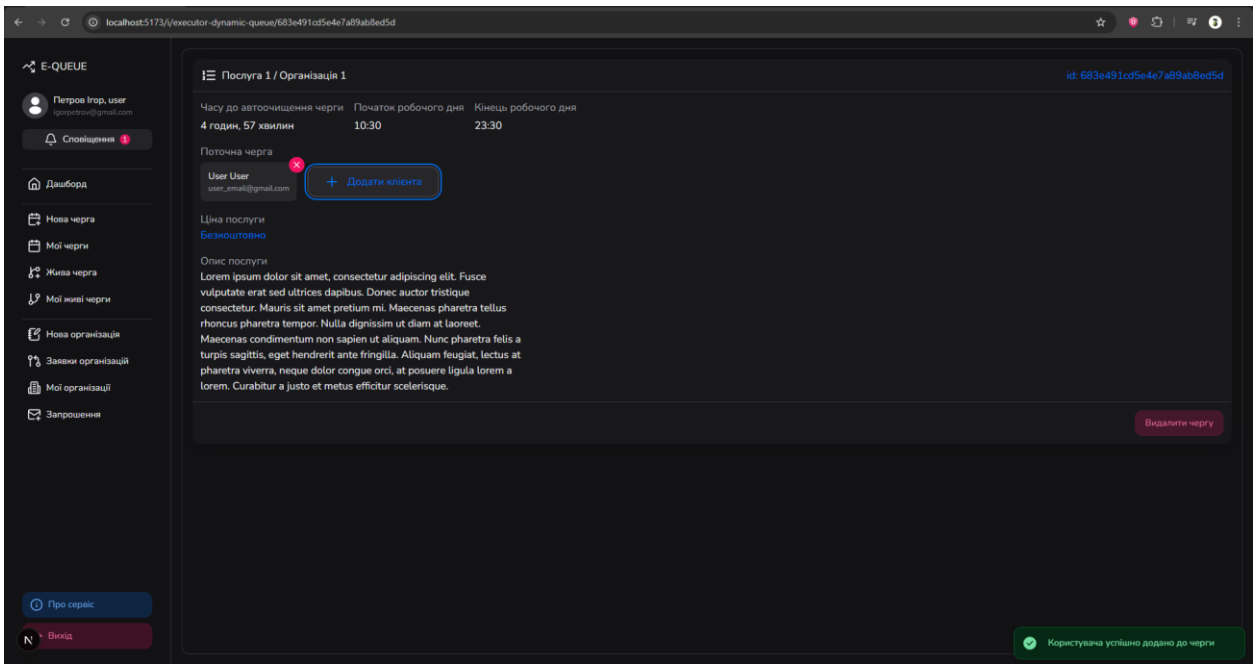


Рисунок 4.15 – Прототип сторінки управління живою чергою

– прототип екранної форми переліку живих черг, де користувач є клієнтом (рисунок 4.16);

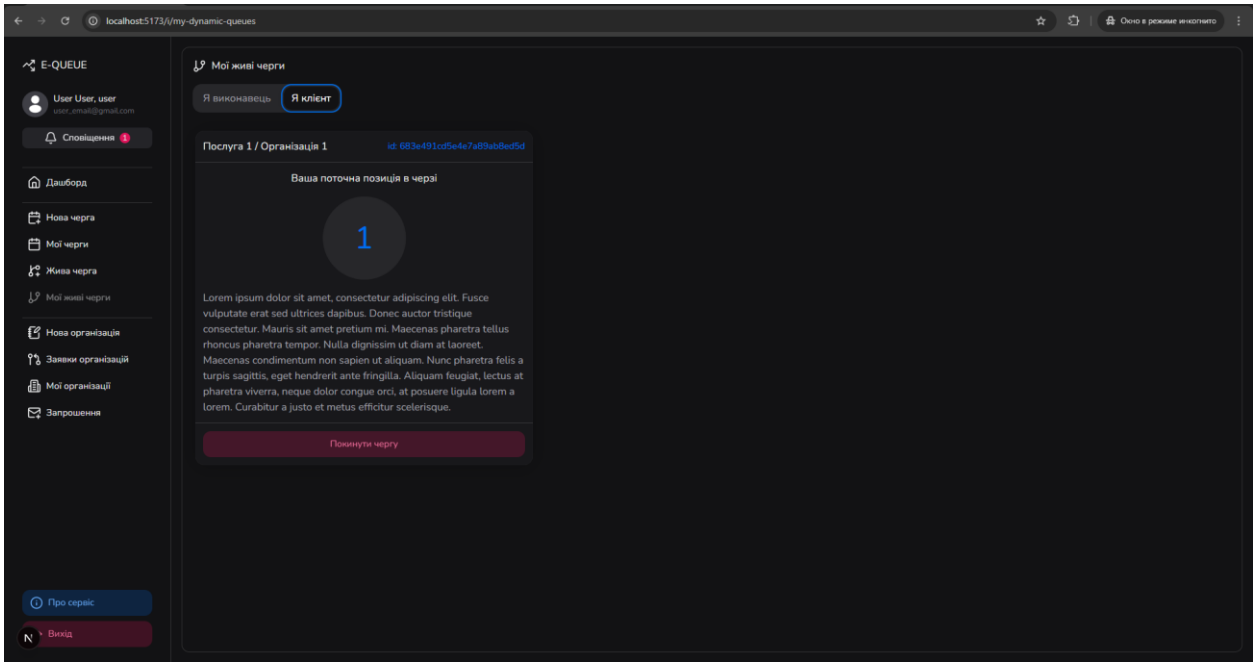


Рисунок 4.16 – Прототип сторінки записів користувача в живі черги

– прототип екранної форми дашборду (рисунок 4.17) ;

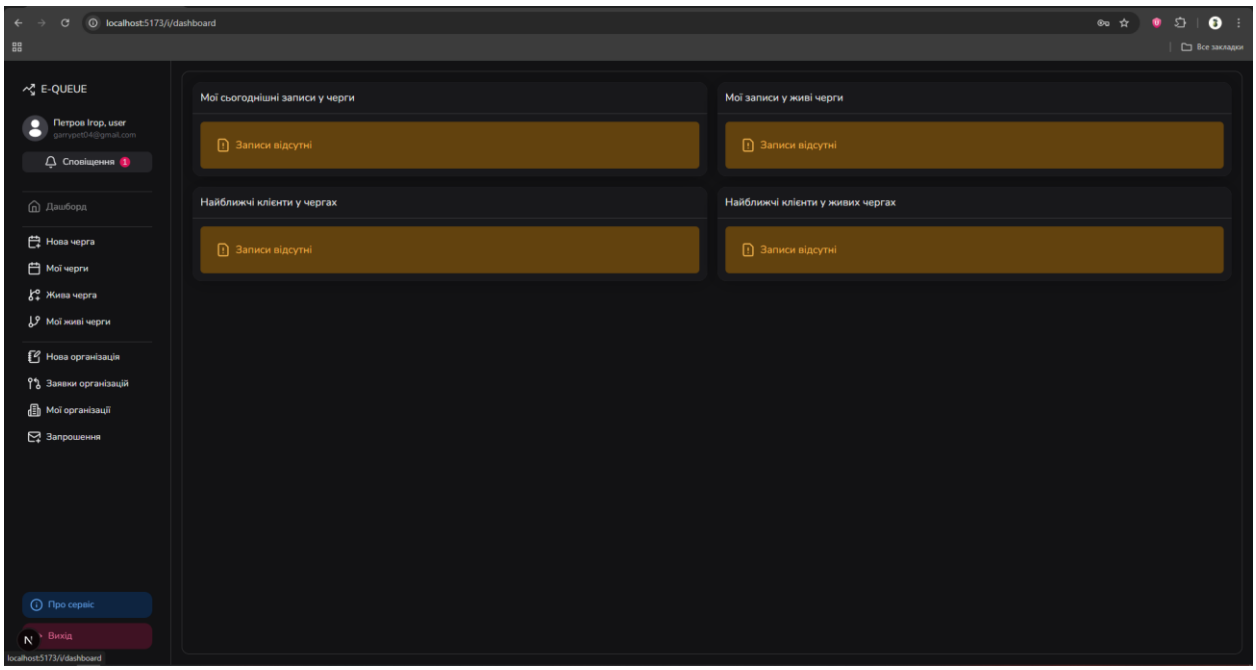


Рисунок 4.17 – Прототип сторінки дашборду

– прототип екранної форми особистого профіля (рисунок 4.18).

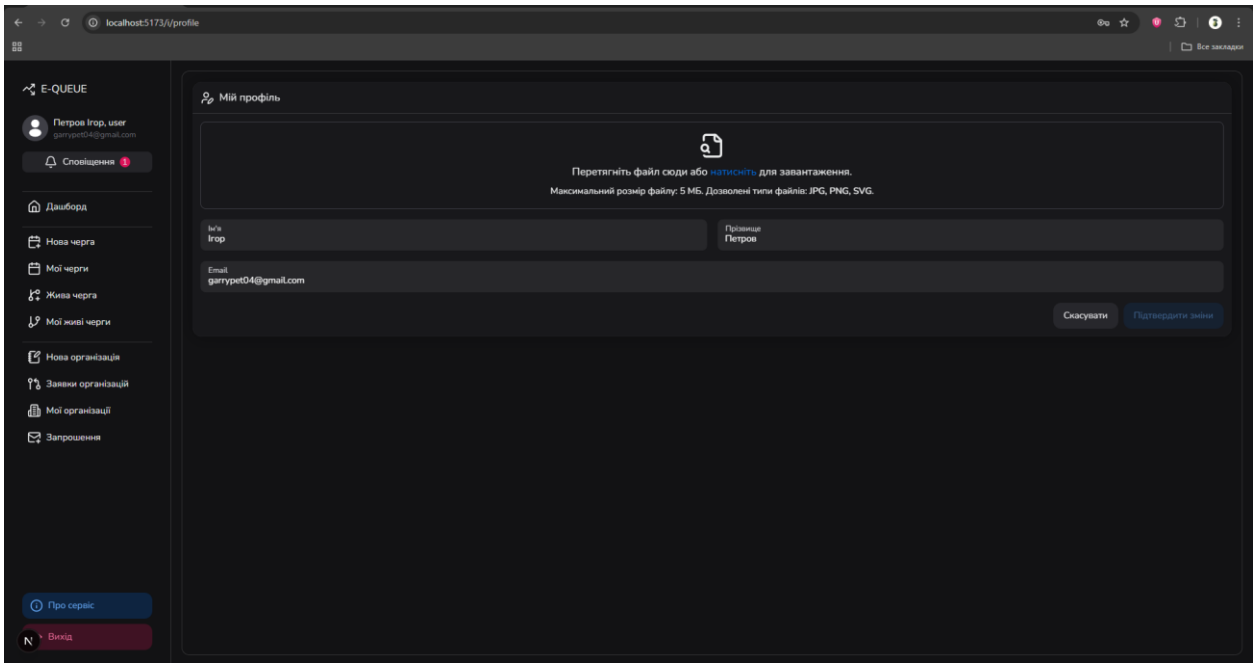


Рисунок 4.18 – Прототип сторінки особистого профіля

4.1.2 Авторизація та управління обліковим записом:

4.1.2.1 Для неавторизованого користувача:

– авторизація в системі;

– реєстрація в системі.

4.1.2.2 Для користувача:

- авторизація в системі;
- вихід з акаунту;
- редагування облікового запису.

4.1.2.3 Для адміністратора:

- авторизація в системі.

4.1.3 Система заявок організацій:

4.1.3.1 Для користувача:

- створення заявки;
- перегляд статусу зявки;
- чат з адміністрацією.

4.1.3.2 Для адміністратора:

- схвалення заявки;
- відхилення заявки;
- чат з користувачем.

4.1.4 Управління організаціями:

4.1.4.1 Для користувача:

- редагування організації;
- видалення організації;
- запрошення до організації.

4.1.5 Статичні черги:

4.1.5.1 Для користувача:

- створення статичних черг;
- редагування статичних черг;
- видалення статичних черг;
- реєстрація в статичних чергах;
- видалення власного запису у статичну чергу.

4.1.6 Живі черги:

4.1.6.1 Для користувача:

– створення живих черг;
– видалення живих черг;
– реєстрація клієнтів у живу чергу;
– видалення клієнтів з живої черги.

4.2 Вимоги до надійності

Передбачити контроль введення інформації та захист від некоректних дій користувача. Забезпечити цілісність інформації в базі даних.

4.3 Умови експлуатації

Умови експлуатації згідно СанПін 2.2.2.542 – 96.

4.3.1 Вид обслуговування

Вимоги до виду обслуговування не висуваються.

4.3.2 Обслуговуючий персонал

Вимоги до обслуговуючого персоналу не висуваються.

4.4 Вимоги до складу і параметрів технічних засобів

Програмне забезпечення повинно функціонувати на IBM-сумісних персональних комп‘ютерах.

Мінімальна конфігурація технічних засобів:

– тип процесору: Intel Core i3;
– об‘єм ОЗП: 4 Гб;
– підключення до мережі Інтернет зі швидкістю від 5 мегабіт.

Рекомендована конфігурація технічних засобів:

– тип процесору: Intel Core i5;
– об‘єм ОЗП: 16 Гб;

– підключення до мережі Інтернет зі швидкістю від 100 мегабіт.

4.5 Вимоги до інформаційної та програмної сумісності

Програмне забезпечення повинно працювати під управлінням операційних систем сімейства WIN32 (Windows XP, Windows NT і т.д.) або Unix.

4.5.1 Вимоги до вхідних даних

Вимоги до вхідних даних не висувається.

4.5.2 Вимоги до вихідних даних

Вимоги до вихідних даних не висувається.

4.5.3 Вимоги до мови розробки

Розробку виконати на мові програмування TypeScript.

4.5.4 Вимоги до середовища розробки

Розробку виконати за допомогою середовища Visual Studio Code.

4.5.5 Вимоги до представленню вихідних кодів

Вихідний код програми має бути представлений у вигляді репозиторію на GitHub.

4.6 Вимоги до маркування та пакування

Вимоги до маркування та пакування не висуваються.

4.7 Вимоги до транспортування та зберігання

Вимоги до транспортування та зберігання не висуваються.

4.8 Спеціальні вимоги

Згенерувати інсталяційну версію програмного забезпечення.

5 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

5.1 Попередній склад програмної документації

У склад супроводжувальної документації повинні входити наступні документи на аркушах формату А4:

– пояснювальна записка;
– технічне завдання;
– текст програми;
– програма та методика тестування;
– керівництво користувача.

Графічна частина повинна бути виконана на аркушах формату А3 та містити наступні документи:

– схема структурна варіантів використання;
– схема структурна компонентів програмного забезпечення;
– креслення вигляду екранних форм.

5.2 Спеціальні вимоги до програмної документації

Програмні модулі, котрі розробляються, повинні бути задокументовані, тобто тексти програм повинні містити всі необхідні коментарі.

6 СТАДІЇ І ЕТАПИ РОЗРОБКИ

№	Назва етапу	Строк	Звітність
1.	Вивчення літератури за тематикою проєкту	15.03	
2.	Розробка технічного завдання	19.04	Технічне завдання
3.	Аналіз вимог та уточнення специфікацій	21.04	Специфікації програмного забезпечення
4.	Проєктування структури програмного забезпечення, проєктування компонентів	23.04	Схема структурна програмного забезпечення та специфікація компонентів (діаграма класів, схема алгоритму)
5.	Програмна реалізація програмного забезпечення	10.05	Тексти програмного забезпечення
6.	Тестування програмного забезпечення	13.05	Тести, результати тестування
7.	Розробка матеріалів текстової частини проєкту	19.05	Пояснювальна записка
8.	Розробка матеріалів графічної частини проєкту	21.05	Графічний матеріал проєкту
9.	Оформлення технічної документації проєкту	01.06	Технічна документація

7 ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ

Тестування розробленого програмного продукту виконується відповідно до “Програми та методики тестування”.

Пояснювальна записка
до дипломного проєкту

на тему: **Вебзастосунок для оптимізації електронних черг та прийому**

КПІ.ІП-1328.045440.02.81

Київ – 2025

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

IDE	– Integrated Development Environment – інтегроване середовище розробки.
API	– Application programming interface, прикладний програмний Інтерфейс.
SDK	– Software development kit.
IT	– Інформаційні технології.
ER	– Entity-Relation diagram.
ОС	– Операційна система.
БД	– База даних.
QMS	– Quality Management System – система управління якістю
JWT	– JSON Web Token – формат токенів для аутентифікації та обміну даними
B2B	– Бізнес для бізнесу. Застосунки створені для бізнесу.
SMS	– Short Message Service, послуга обміну (передачі і прийому) короткими текстовими повідомленнями
SAAS	– Software as a service, модель використання застосунку, замовники платять не за володіння програмами як такими, а за їх використання.
FIFO	– First In, First out.
HTTPS	– HyperText Transfer Protocol Secure, вказує, що протокол HTTP має використовуватися, але з додатковим шаром шифрування.
SPA	– Single Page Application.
RPC	– Remote Procedure Call, дозволяє виконувати функції на сервері з клієнта, наче вони локальні.
BSON	– Binary JavaScript Object Notation, надмножина JSON у бінарному форматі.
CRUD	– Чотири основні операції REST, створення, видалення, запис, редагування.
REST	– Representational State Transfer, архітектурний стиль взаємодії компонентів розподіленої програми у мережі.
JSON	– JavaScript Object Notation, текстовий формат даних, що зчитується людиною та машиною
CORS	– Cross-Origin Resource Sharing, механізм захисту сервера від несанкціонованого доступу.

ВСТУП

Сучасні компанії часто стикаються з проблемою неефективної організації і управління онлайн чергами через запізнення клієнтів, невідповідності реального часу надання послуги і прописаному у записі, некоректному розподілу клієнтів між штатом, нечітку інформацію про послугу для клієнтів, тощо. Це призводить до іноді значних затримок у прийомі клієнтів, перевантаження штату організацій і, як наслідок, незадоволення клієнтів, або навіть їх повну відмову від користування послугами організації. Необхідно впровадити рішення, яке оптимізує розподіл ресурсів, покращить швидкість роботи, спростить менеджмент електронними чергами самим організаціям і при цьому надасть зручність користування сервісу кінцевим клієнтам.

Мета розробки – удосконалення процесів взаємодії між клієнтами та організаціями шляхом впровадження механізмів керування живими чергами і чергами за попереднім записом. Застосунок має дозволити організаціям ефективно управляти робочим часом виконавців і їх послугами, надати клієнтам максимально повну інформацію про послугу, шляхом додавання у вимоги до надання послуги документів, фото, текстового опису.

Клієнти організації мають отримати всю детальну інформацію, що надана організацією і можуть слідкувати за порядком прийому, щоб мати на увазі, якщо черга змістилась через запізнення/передчасний прийом. Також необхідна адмінська панель у веб форматі, де адміністрація зможе підтверджувати або відхиляти запити організації на створення.

1 ПЕРЕДПРОЄКТНЕ ОБСТЕЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка завдання дипломного проєктування

Дипломне проєктування передбачає виконання наступних завдань:

– аналіз предметної області та опис її ключових бізнес-процесів, визначення загального завдання розробки у рамках ДП;

– аналіз існуючих рішень (як програмних продуктів, так і алгоритмічних чи технічних рішень) обраного завдання розробки у рамках ДП;

– аналіз та моделювання бізнес-процесів;

– розроблення функціональних, нефункціональних та системних вимог до програмного забезпечення;

– аналіз економічних показників програмного забезпечення ДП;

– постановка завдання на розробку програмного забезпечення ДП;

– розроблення архітектури програмного забезпечення;

– розроблення архітектурних рішень та обґрунтування вибору засобів розробки програмного забезпечення;

– конструювання та розроблення програмного забезпечення;

– аналіз безпеки даних програмного забезпечення;

– аналіз якості та тестування програмного забезпечення;

– розгортання та супровід програмного забезпечення;

– створення супроводжувальної документації до розробленого програмного забезпечення.

1.2 Аналіз предметної області

Електронна система керування чергами (QMS) — це цифрове рішення, що дає змогу зменшувати час очікування та оптимізувати потоки клієнтів у сервісних організаціях (аеропорти, банки, лікарні, тощо) за допомогою інтеграції терміналів видачі талонів та/або мобільних та веб-додатків.

Серед напрямів розвитку можна виділити:

- Черги з отриманням талону в кіосках. Це жива черга, доступ до якої надається клієнтам по приходу у відділення. Гарно підходить для пунктів видачі та державних установ, де час надання послуги несуттєвий, або не може бути визначеним заздалегіть.
- Віртуальні черги, які дозволяють клієнтам записатись заздалегіть на конкретний час та очікувати своєї черги за границями пункту прийому.
- Омніканальні рішення, що об’єднують голосові виклики, SMS, чат-боти та веб-інтерфейси в єдину систему обслуговування, підтримуючи повноцінну взаємодію з клієнтом онлайн та офлайн.

Для ведення черги також часто використовують загальні рішення, на кшталт Google Spreadsheets[1], Excel[2] або Teams Queue[3] для конкретної послуги, що не буде повторюватись у подальшому.

Типовий бізнес-процес включає реєстрацію клієнта онлайн або на терміналі, видачу талона з інформацією про позицію в черзі або показ його позиції онлайн, очікування з можливістю віддаленого моніторингу статусу та в кінці кінців безпосереднє обслуговування працівником сервісу.

Незважаючи на очевидні переваги, багато існуючих рішень мають недоліки. Системи не надають необхідної інформації про умови надання послуги, що може спричиняти конфуз на етапі обслуговування і призводить до втрати потенційних клієнтів. При використанні віртуальної і живої черги одночасно, може бути проблема неспівпадінь, коли клієнти приходять на заброньований для них час, а перед ними ще доволі об’ємна жива черга, що створює іноді значні затримки в обслуговуванні.

В рамках дипломного проєкту було обрано створити комплексне рішення, яке покриває потребу в одночасному створенні живих черг, коли клієнти самі приходять на запис у відділення обслуговування та статичних черг, коли клієнти записуються заздалегіть через веб-інтерфейс. Також необхідним є створення власної адмінської панелі, де адміни зможуть вести діалог з потенційними організаціями, запитуючи у них уточнюючі дані, щоб

переконатись, що організація дійсна і пропонує реаліні послуги. Також необхідна система ролей всередині самої організації, щоб надавати більше можливостей менеджерам та навпаки обмежувати права простих виконавців.

1.3 Аналіз існуючих рішень

Проаналізуємо відоме на сьогодні алгоритмічне забезпечення у даній області та технічні рішення, що допоможуть у реалізації «E-queue». Далі будуть розглянуті готові програмні рішення, допоміжні програмні засоби та засоби розробки.

1.3.1 Аналіз відомих програмних продуктів

У сучасних реаліях автоматизація й оптимізація процесів обслуговування клієнтів стає вирішальним фактором конкурентоспроможності. Особливо це стосується сфер, де черговість обслуговування, гнучке управління ролями й швидкий зворотний зв’язок мають критичне значення – наприклад, у закладах державного, освітнього чи комерційного сектору. Тому існує немало B2B рішень, які покривають такі потреби.

Одними з таких додатків є Qmatic[4] і Waitwhile[5].

1. Qmatic

Qmatic позиціонує себе як глобальний лідер у сфері управління клієнтським шляхом, обробляючи понад два мільярди взаємодій щорічно в більш ніж 65 000 інсталяцій по всьому світу. Платформа представлена двома основними рішеннями: Experience Cloud (SaaS), яке пропонує швидке розгортання, масштабованість на вимогу та модель оплати за кількість обслугованих клієнтів. Qmatic охоплює ключові сценарії: онлайн-запис і планування прийому, віртуальні черги та самообслуговування через кіоски. Приклад функціоналу з точки зору клієнту (запис на послугу) зображено на рисунку 1.1.

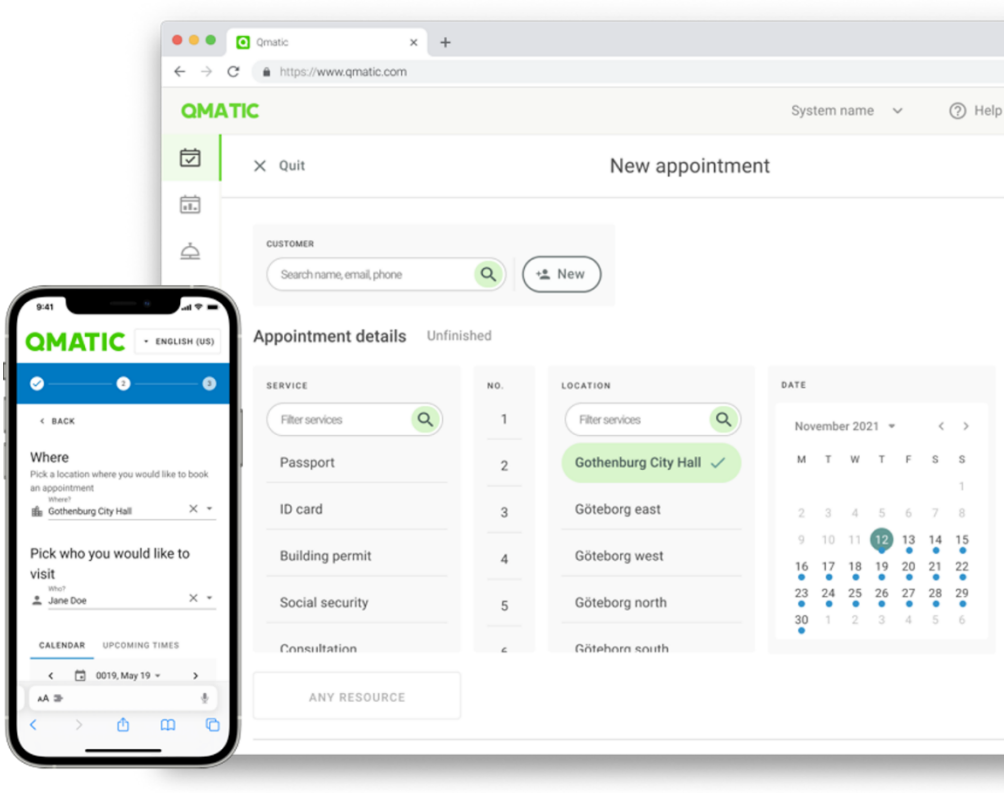


Рисунок 1.1 – Веб-платформа Qmatic та її мобільна версія

2. Waitwhile

Waitwhile – це хмарне рішення для керування чергами та записами, яке використовується в понад 50 000 точках по всьому світу. Платформа об’єднує використовує систему онлайн-бронювань, дозволяючи клієнтам приєднуватися до віртуальної черги через веб-посилання. Приклад реєстрації на послугу, як клієнта зображено на рисунках 1.2, 1.3 та 1.4.

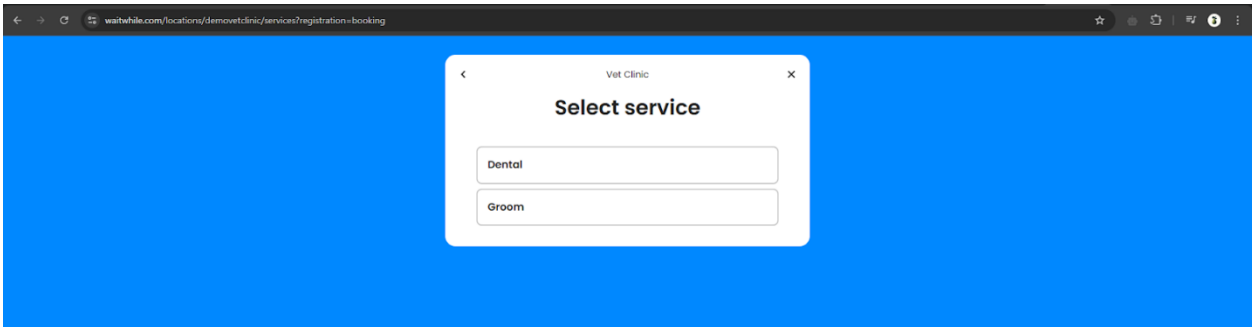


Рисунок 1.2 – Процес реєстрації на послугу. Етап вибору послуги

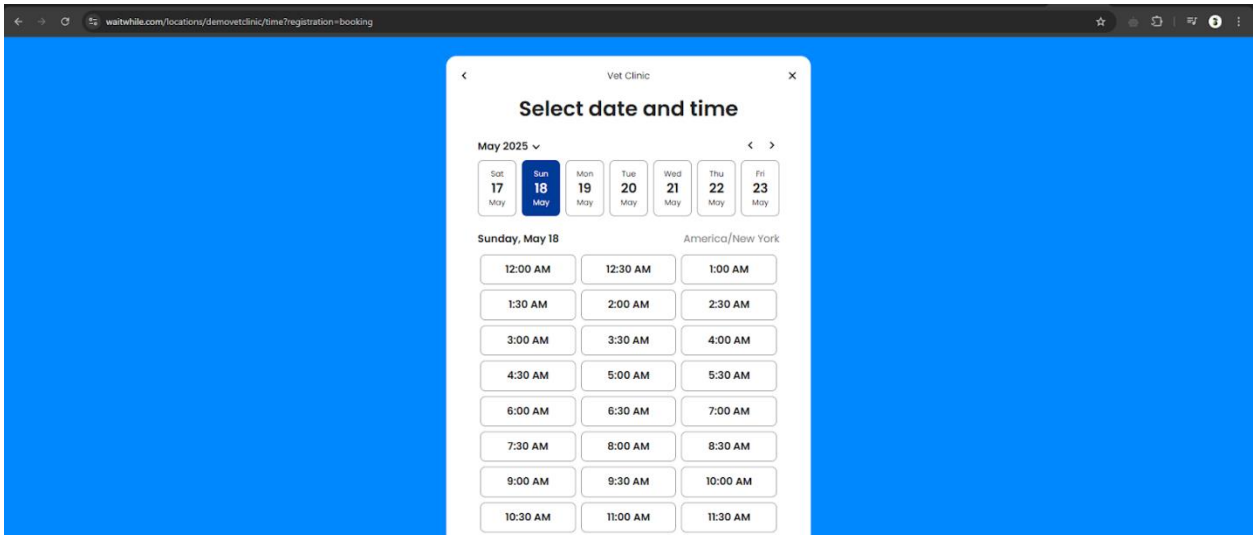


Рисунок 1.3 – Процес реєстрації на послугу. Етап вибору часу

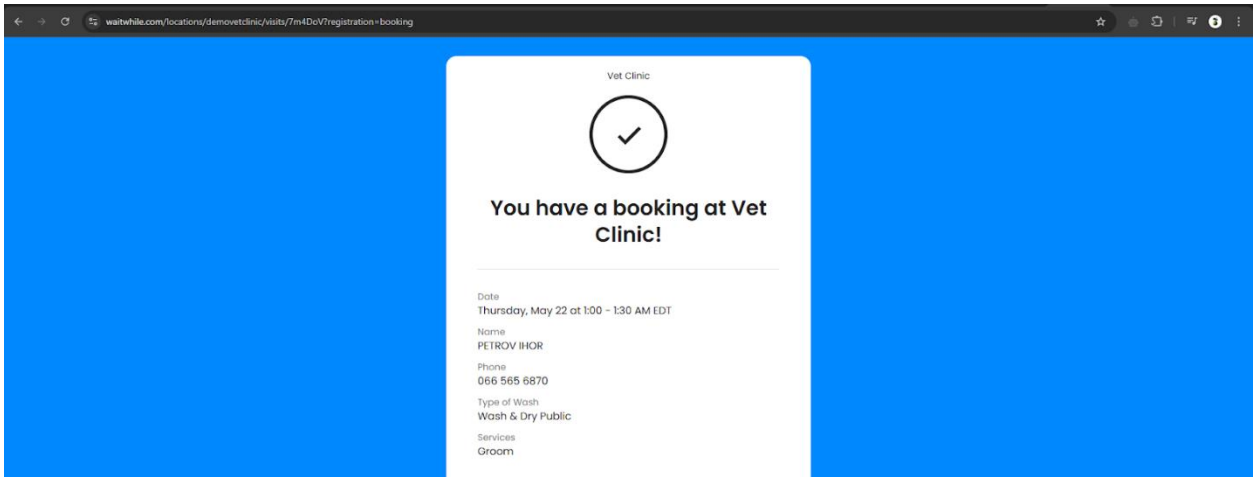


Рисунок 1.4 – Процес реєстрації на послугу. Фіналізація, отримання інформації про успішний запис

На рисунку 1.5 зображена таблиця з клієнтами з точки зору організації.

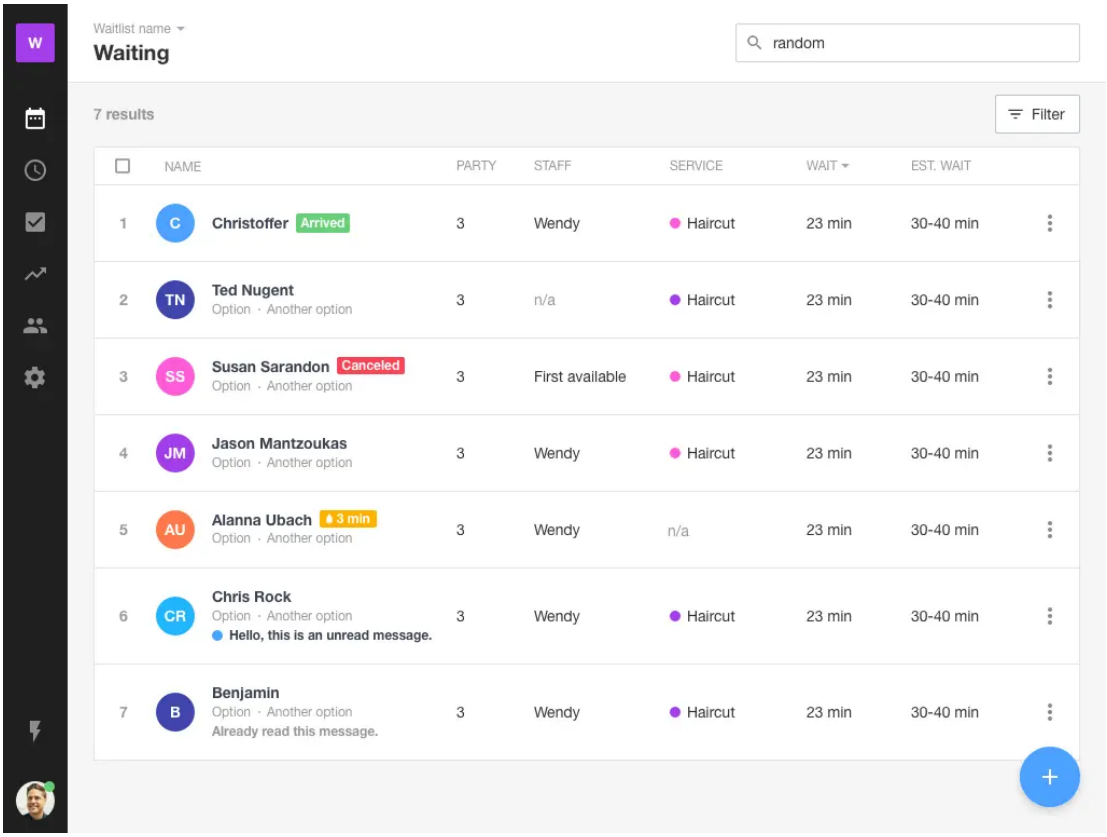


Рисунок 1.5 – Дашборд Waitwhile з чергою клієнтів

Для порівняння проєкту з аналогами можна скористатись таблицею 1.1.

Таблиця 1.3 – Порівняння з аналогами

Функціонал	E-queue	Waitwhile	Qmatic	Пояснення
Створення декількох організації	+	+	-	Qmatic орієнтований виключно на роботу з чергами — організації створюються офлайн, без клієнтської форми
Керування ролями та правами всередині організації	+	-	+	Waitwhile надає можливість лише бути оператором у конкретній черзі.

Продовження таблиці 1.3.

Прикріпленн документів/медіа до статичних/динамічних черг	+	-	-	Платформи не підтримують такий функціонал, у Waitwhile максимум можна додати декілька додаткових полів для реєстрації у черзі.
Створення статичних черг, запис клієнтів і моніторинг теперішньої позиції	+	+	+ (з обмеженнями)	Waitwhile підтримує створення статичних черг і букінг конкретного часу, також дозволяє реєструватись у них самим користувачам. Qmatic же пропонує адміністрації організації реєструвати користувачів, після чого видає талони онлайн, за яким клієнт
Створення живих черг і моніторинг теперішньої позиції	+	-	+	Qmatic частково підтримує живі черги через видачу талонів у кіосках.

Продовження таблиці 1.3.

Запити на приєднання до штату організації	+	-	-	Waitwhile і Qmatic: не підтримують такий функціонал. Необхідно ділитись посиланням окремо від платформи, або шукати організацію вручну.

1.3.2 Аналіз відомих алгоритмічних та технічних рішень

Для онлайн-черг найчастіше використовуються такі базові підходи до організації самого процесу очікування у черзі:

- FIFO-черга (First-In, First-Out). Гарантує чесну обробку запитів у порядку надходження. Підходить для сценаріїв із відносно однорідними операціями, проте не враховує тривалість обслуговування.
- Пріоритетна черга. Кожен елемент чергши отримує вагу (наприклад, на основі типу послуги або статусу користувача). Система обробляє спочатку запити з вищим пріоритетом, поступово спускаючись по пріоритету вниз. Забезпечує гарне обслуговування критичних випадків, але є дуже чутливою і може призводити до відмов у разі багатьох послуг з низьким пріоритетом або недостатній кількості даних для виставлення коректного пріоритету.
- Призначення часових слотів. При використанні такого алгоритму організація заздалегіть планує часовий розподіл між клієнтами, користувачі резервують конкретний інтервал часу, а система демонтструє завантаженість для виконавців і клієнтів. З мінусів такого підходу можна виділити лише необхідність додаткової логіки для перенесень і скасувань.

Для дипломного проєкту можна виділити комбінацію FIFO-черг для організації швидких живих черг, коли часові проміжки не мають заданого параметру або є незначними та Time-slot assignment для організації статичних черг, щоб користувачі могли заздалегідь планувати час початку і кінця прийому.

Також для створення надійного і зручного веб-додатку важливо ретельно підійти до вибору загальної архітектури та ключових технологій.

З архітектурних підходів можна виділити:

- Моноліт, забезпечує швидкий старт і спрощену розробку, але сильно обмежує можливості масштабування, а також створює потенційні секьюрні діри.
- Клієнт-сервер, додає гнучкості при масштабуванні та незалежного розгортання, проте потребує більше зусиль на написання окремого клієнту і серверу, можуть бути потенційні неспівпадіння типів даних при використанні різних мов для написання клієнту і серверу.
- Мікросервісна архітектура, забезпечує максимально гнучке і незалежне управління серверною стороною, може бути корисною при використанні одного бекенду для багатьох клієнтів, однак потребує складної оркестрації і супроводу, в деякий випадках може бути зайвим ускладненням без реальної користі.

Для дипломного проєкту краще вибрати архітектуру «Клієнт-сервер» через баланс швидкості розробки, легкість у подоланні недоліків і розгортанні та якості фінального продукту, мікросервісна архітектура є зайвим ускладненням проєкту, а моноліт занадто ненадійний і несе потенційні проблеми, які перешкодять маштабованості продукту.

1.4 Аналіз та моделювання бізнес-процесів

Для моделювання бізнес-процесів використовуються BPMN моделі. Модель бізнес-процесу створення організації зображена у графічних

матеріалах, креслення 1. Моделі створення статичної черги і запису зображені на рисунках 1.7 і 1.8.

Опис моделі бізнес-процесу створення створення організації:

– Користувач авторизується і переходить на сторінку «Створити організацію».

– Користувач заповнює всі обов’язкові поля форми (назва, опис тощо) і натискає «Підтвердити».

– На клієнті відбувається валідація полів форми.

– Якщо валідація не пройдена, відображається повідомлення про помилку.

– Якщо валідація успішна, відображається індикатор завантаження і надсилається запит на сервер.

– Сервер приймає запит, очищає дані від зайвих пробілів і переновсів і створює новий запис у таблиці заявок зі статусом «Не розглянуто».

– Сервер повертає клієнту відповідь про успішне створення заявки.

– Клієнт прибирає індикатор завантаження та перенаправляє користувача на сторінку «Мої заявки».

– Клієнт із сторінки «Мої заявки» надсилає запит на сервер за переліком заявок поточного користувача.

– Сервер шукає в базі даних усі записи-заявки, пов’язані з цим користувачем, і повертає їх.

– Клієнт відображає список заявок із їх статусами («Не розглянуто», «В обробці», «Схвалено»).

– Адміністратор авторизується і заходить на сторінку «Вільні заявки» (статус «Не розглянуто»).

– Клієнт надсилає серверу запит на перелік заявок зі статусом «Новий».

– Сервер виконує запит на заявки зі статусом «Не розглянуто» і повертає список.

– Клієнт відображає адміністратору картки заявок із основною інформацією (назва організації, хто подав, дата).

– Адміністратор натискає кнопку «Взяти в обробку» на конкретній картці заявки.

– На клієнті кнопка стає неактивною і з’являється індикатор завантаження.

– Клієнт надсилає запит на сервер із ID заявки та новим статусом «В обробці».

– Сервер знаходить відповідний запис у таблиці заявок і оновлює поле status на «В обробці», записує time-stamp.

– Сервер надсилає повідомлення користувачу про оновлення статусу заявки на «В обробці».

– Сервер повертає відповідь про успішне оновлення, клієнт прибирає індикатор завантаження.

– Адміністратор натискає на картку заявки або кнопку «Переглянути», щоб отримати детальну інформацію.

– Клієнт надсилає запит на сервер із ID заявки для отримання повної інформації.

– Сервер шукає у базі запис із цим ID і повертає всі поля заявки.

– Клієнт відображає сторінку із детальною інформацією про заявку.

– Усередині модального вікна адміністратор натискає «Схвалити заявку».

– На екрані відображається поле для введення коментаря до рішення.

– Адміністратор вводить коментар і натискає кнопку «Підтвердити».

– Клієнт показує індикатор завантаження і надсилає запит на сервер з параметрами: ID заявки, status = «Схвалено», comment = «…».

– Сервер знаходить заявку в таблиці, оновлює status на «Схвалено», зберігає коментар і timestamp.

– Сервер створює нову організацію у відповідній таблиці (якщо передбачено бізнес-логікою).

– Сервер надсилає повідомлення користувачу про успішне схвалення та створення організації.

– Сервер повертає клієнту підтвердження успіху, клієнт прибирає індикатор завантаження.

– Користувач отримує сповіщення про оновлення статусу заявки на «В обробці» і пізніше на «Схвалено».

– У базі даних оновлюються записи у таблиці requests зі статусами та коментарями, а за потреби створюється запис нового запису у таблиці organizations.

– Користувач у розділі «Мої заявки» бачить оновлений статус заявки «Схвалено» і, за необхідності, отримує доступ до новоствореної організації.

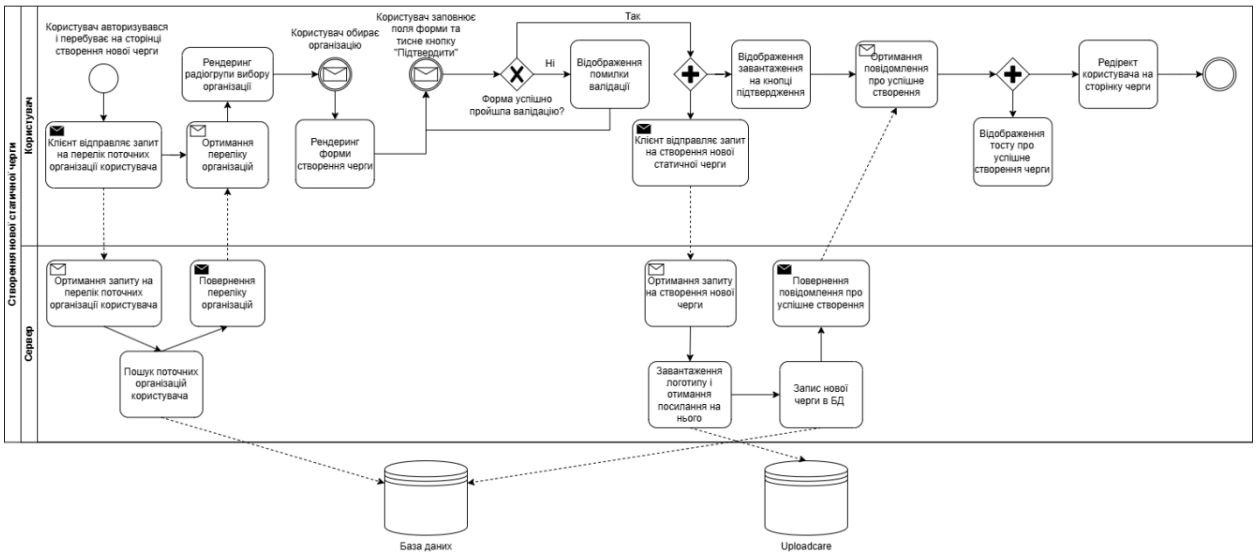


Рисунок 1.7 – Модель бізнес-процесу створення статичної черги

Опис моделі бізнес-процесу створення статичної черги:

– Користувач авторизується і переходить на сторінку створення нової черги веб-застосунку.

– Клієнт надсилає запит на перелік доступних організацій, в яких користувач є учасником.

– Система обробляє запит, знаходить поточні організації користувача в базі даних та повертає їх перелік.

– Система відображає радіогрупу для вибору організації.

– Користувач обирає організацію з наданого переліку.

– Відбувається рендеринг форми створення нової черги.

– Користувач заповнює поля форми (опис черги, слоти часу тощо) та натискає кнопку «Підтвердити».

– Система виконує валідацію введених даних.

– Якщо валідація не пройдена — показується повідомлення про помилку.

– Якщо форма успішно пройшла валідацію:

– Відображається кнопка підтвердження, користувач натискає її.

– Відображається індикатор завантаження.

– Клієнт надсилає запит на створення нової черги.

– Сервер отримує запит, виконує завантаження логотипу черги в Uploadcare та отримує посилання на нього.

– Після цього відбувається запис нової черги до бази даних.

– Сервер повертає повідомлення про успішне створення черги.

– Користувач отримує повідомлення та відображення тосту про успішне створення.

– Відбувається редирект користувача на сторінку черги.

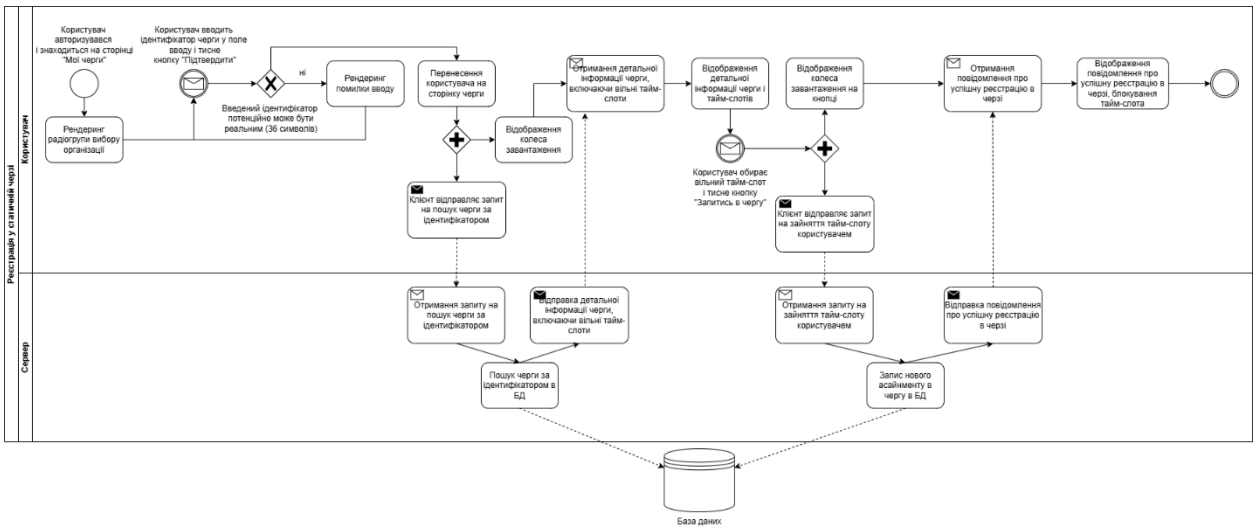


Рисунок 1.8 – Модель бізнес-процесу реєстрації у статичній черзі

Опис моделі бізнес-процесу реєстрації у статичній черзі:

– Користувач авторизується та знаходиться на сторінці «Мої черги» веб-застосунку.

– Користувач вводить ідентифікатор черги у відповідне поле та натискає кнопку «Підтвердити».

– Система перевіряє, чи введений ідентифікатор потенційно валідний (36 символів).

– Якщо ідентифікатор некоректний, відображається повідомлення про помилку введення, і процес зупиняється.

– Якщо ідентифікатор коректний, користувача перенаправляють на сторінку черги.

– Клієнт надсилає запит на пошук черги за ідентифікатором.

– Сервер отримує запит, виконує пошук черги у базі даних та відправляє детальну інформацію про чергу, включаючи вільні тайм-слоти.

– Користувачу відображається детальна інформація про чергу та доступні тайм-слоти.

– Користувач обирає вільний тайм-слот і натискає кнопку «Записатись в чергу».

– Клієнт надсилає запит на зайняття обраного тайм-слоту.

– Сервер отримує запит, створює новий запис (асайнмент) у базі даних.

– Система надсилає користувачу повідомлення про успішну реєстрацію в черзі.

– Користувачу відображається повідомлення про успішну реєстрацію, і обраний тайм-слот блокується.

Висновки до розділу

Було проведено обстеження предметної області, що дозволило чітко окреслити завдання дипломного проєктування, виявити ключові процеси та обґрунтувати необхідність розробки веб-застосунку для оптимізації електронних черг. Спочатку сформульовано головну мету: підвищити ефективність обслуговування клієнтів та спростити менеджмент для організацій шляхом створення єдиного цифрового рішення. Далі здійснено детальний аналіз предметної області, у якому розглянуто сучасні підходи до організації живих і віртуальних черг, а також оцінено можливості омніканальних сервісів, що поєднують термінали, мобільні й веб-інтерфейси.

Наступним етапом був огляд існуючих рішень: порівняно функціонал комерційних продуктів Qmatic і Waitwhile, визначено їх сильні та слабкі сторони й виявлено прогалини, зокрема відсутність можливості прикріплення документів, відсутність живих і статичних черги в одному рішенні та інші. З огляду на це обґрунтовано доцільність створення «E-queue» як платформи, що поєднує переваги обох підходів та закриває виявлені потреби користувачів. Також проаналізовано алгоритмічні рішення і вибрано комбінацію декількох підходів для підтримки як живих, так і статичних черг.

Також було сформовано моделювання основних бізнес-процесів у нотації BPMN, зокрема основні процеси створення організації й формування черг. Це моделювання візуалізувало послідовність дій користувачів і адміністрації, включаючи валідацію даних, зміну статусів заявок та сповіщення. Отримані результати заклали міцну основу для подальшого

розроблення вимог, технічних рішень та архітектури системи, забезпечивши цілісне розуміння контексту та завдань майбутнього програмного продукту.

2 РОЗРОБЛЕННЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Варіанти використання програмного забезпечення

Діаграма варіантів використання з ключовими акторами (гість, користувач, адмін) та основним функціоналом для створення живих і статичних черг винесена у графічні матеріали, креслення 2.

В таблицях 2.1–2.38 наведено опис варіантів використання програмного забезпечення.

Таблиця 2.1 – Варіант використання UC-01

Use case name	Перегляд посадкової сторінки
Use case ID	UC-01
Goals	Надати можливість користувачу відкрити сторінку
Actors	Гість (незареєстрований користувач)
Trigger	Користувач вперше зайшов у застосунок і ще не встиг зареєструватись
Pre-conditions	Користувач не авторизований
Flow of Events	Користувач заходить на сторінку, верстка посадкової сторінки підвантажується і він бачить посадкову сторінку.
Extension	-
Post-Condition	Користуч побачив посадкову сторінку

Таблиця 2.2 – Варіант використання UC-02

Use case name	Авторизуватись
Use case ID	UC-02
Goals	Надати зареєстрованому користувачу доступ до внутрішньої системи
Actors	Гість (зареєстрований користувач)
Trigger	Користувач бажає увійти до системи

Продовження таблиці 2.2.

Pre-conditions	Користувач вже зареєстрований, але не увійшов у систему.
Flow of Events	Користувач відкриває додаток, заходить на сторінку Авторизації, вводить пошту та пароль, натискає кнопку «Увійти».
Extension	Якщо користувача з такою поштою не існує, то форма поверне помилку і користувача не пропустить далі. Якщо користувач з такою поштою існує, але пароль неправильний, форма також поверне помилку.
Post-Condition	Користувач авторизований і перенаправлений у внутрішню систему, він не може перейти на сторінку авторизації/реєстрації, поки не вийшов з системи

Таблиця 2.3 – Варіант використання UC-03

Use case name	Зареєструватись
Use case ID	UC-03
Goals	Надати незареєстрованому користувачу доступ до внутрішньої системи
Actors	Гість (незареєстрований користувач)
Trigger	Користувач бажає зареєструватись на платформі
Pre-conditions	Користувач ще не зареєстрований на пошту, з якої планує зареєструватись.
Flow of Events	Користувач відкриває додаток, заходить на сторінку Реєстрації, вводить ім’я та прізвище, пошту, пароль та підтвердження паролю на натискає кнопку «Зареєструватись».
Extension	Якщо користувач з такою поштою вже існує, то форма поверне помилку і користувача не пропустить далі. Якщо користувач неправильно ввів підтвеодження паролю, форма також поверне помилку.
Post-Condition	Користувач зареєстрований і перенаправлений у внутрішню систему, він не може перейти на сторінку авторизації/реєстрації, поки не вийшов з системи.

Таблиця 2.4 – Варіант використання UC-04

Use case name	Вийти з акаунту
Use case ID	UC-04
Goals	Надати зареєстрованому користувачу можливість вийти з акаунту
Actors	Зареєстрований користувач
Trigger	Користувач бажає вийти зі свого акаунту
Pre-conditions	Користувач авторизувався і зайшов на платформу
Flow of Events	Користувач натискає на кнопку «Вийти» у панелі управління. Показується модальне вікно з підтвердженням виходу і попередженням про наслідки. Користувач натискає на кнопку «Вийти» у модальному вікні.
Extension	У модальному вікні також є кнопка «Скасувати», натиснувши на яку користувач може відмінити вихід з системи.
Post-Condition	Користувач вийшов з системи і перенаправлений назад на сторінку авторизації, він не може перейти на до внутрішнього застосунку, поки не пройде авторизацію.

Таблиця 2.5 – Варіант використання UC-05

Use case name	Створити заявку на організацію
Use case ID	UC-05
Goals	Користувач успішно створив заявку на створення організації
Actors	Зареєстрований користувач
Trigger	Користувач бажає створити нову організацію
Pre-conditions	Користувач авторизувався і зайшов на платформу
Flow of Events	Користувач натискає на кнопку «Нова організація» у панелі управління. Користувача редіректить на сторінку з формою для створення організіції, він заповнює повну інформацію у заявці та тисне на кнопку «Залишити заявку».
Extension	Якщо якісь з обов’язкових полів не були заповнені, то форма покаже помилку і не буде відправлена.

Продовження таблиці 2.5.

Post-Condition	Користувач залишив заявку, яка наразі має статус «Не розглянуто» і був перенаправлений на сторінку «Мої заявки».

Таблиця 2.6 – Варіант використання UC-06

Use case name	Переглянути власні заявки на організації
Use case ID	UC-06
Goals	Надати можливість користувачу переглядати власні заявки на організації
Actors	Зареєстрований користувач
Trigger	Користувач бажає переглянути заявки на організації і їх статус
Pre-conditions	Користувач авторизувався і зайшов на платформу
Flow of Events	Користувач натискає на кнопку «Мої заявки» у панелі управління. Користувача перенаправляє на сторінку заявок.
Extension	-
Post-Condition	Користувач бачить перелік своїх заявок у вигляді карток, з основною інформацією.

Таблиця 2.7 – Варіант використання UC-07

Use case name	Переглянути деталі заявки
Use case ID	UC-07
Goals	Надати можливість користувачу переглядати деталі конкретної заявки на організацію
Actors	Зареєстрований користувач
Trigger	Користувач бажає переглянути детальну інформацію заявки на організацію.
Pre-conditions	Користувач залишив заявку на організацію на сторінці «Нова організація», знаходиться на сторінці «Мої організації» або перейшов по прямому посиланню.

Продовження таблиці 2.7.

Flow of Events	Користувач натискає на кнопку «Детальніше» на картці заявки, яку бажає переглянути на сторінці «Мої організації». Користувача перенаправляє на сторінку конкретної заявки, де він може побачити детальну інформацію по своїй заявці і її статус, а також чат з адміністрацією.
Extension	-
Post-Condition	Користувач бачить сторінку конкретної заявки, де він може переглянути детальну інформацію по своїй заявці і її статус, а також чат з адміністрацією.

Таблиця 2.8 – Варіант використання UC-08

Use case name	Написати адміністрації в чаті заявки
Use case ID	UC-08
Goals	Надати можливість користувачу спілкуватись в чаті стосовно його заявки з адміністрацією.
Actors	Зареєстрований користувач
Trigger	Користувач бажає написати повідомлення в чаті заявки зі зверненням до адміністрації платформи.
Pre-conditions	Користувач залишив заявку на організацію на сторінці «Нова організація», адміністрація взяла її в роботу, користувач знаходиться на сторінці конкретної заявки.
Flow of Events	Користувач залишає коментар у полі вводу і натискає кнопку «Відправити».
Extension	Якщо користувач нічого не написав, то кнопка буде заблокована
Post-Condition	Користувач бачить своє повідомлення в чаті, воно також відображається у адміністратора платформи, який взяв цю заявку в роботу.

Таблиця 2.9 – Варіант використання UC-09

Use case name	Переглянути організації
Use case ID	UC-09

Продовження таблиці 2.9.

Goals	Надати можливість користувачу переглядати свої організації
Actors	Зареєстрований користувач
Trigger	Користувач бажає переглянути свої організації.
Pre-conditions	Користувач залишив заявку на організацію на сторінці «Нова організація», адміністрація схвалила її, створивши тим самим організацію в системі. Також організація може бути відображена, якщо користувача запросили до неї і він прийняв запрошення.
Flow of Events	Користувач натискає на кнопку «Мої організації» у панелі управління. Користувача перенаправляє на сторінку організацій, у яких він є частиною штату.
Extension	-
Post-Condition	Користувач бачить свої організації і коротку інформацію по ним.

Таблиця 2.10 – Варіант використання UC-10

Use case name	Запросити в організацію
Use case ID	UC-10
Goals	Надати можливість користувачу запросити інших в організацію
Actors	Зареєстрований користувач
Trigger	Користувач бажає запросити іншого користувача в організацію.
Pre-conditions	Користувач є у штаті організації і має адмінські права в ній, або є її власником. Користувач знаходиться на сторінці «Мої організації».
Flow of Events	Користувач натискає на кнопку «Запросити» на картці організації зі сторінки «Мої організації». Відкривається модальне вікно, де він вводить пошту запрошуваного, роль, на яку його запрошують, та коментар до запрошення. Користувач натискає на кнопку «Відправити».

Продовження таблиці 2.10.

Extension	Якщо користувач не ввів якесь з полів, то форма покаже помилку і не відправиться, у модальному вікні є кнопка «Скасувати», при натискані на яку форма також не відправиться.
Post-Condition	У запрошеного користувача з’являється нове запрошення на сторінці «Запрошення»

Таблиця 2.11 – Варіант використання UC-11

Use case name	Редагувати дані організації
Use case ID	UC-11
Goals	Надати користувачам можливість редагувати дані організацій
Actors	Зареєстрований користувач
Trigger	Користувач хоче змінити інформацію про організацію.
Pre-conditions	Користувач є у штаті організації і має адмінські права в ній, або є її власником. Користувач знаходиться на сторінці «Мої організації».
Flow of Events	Користувач натискає на кнопку «Детальніше» на картці організації зі сторінки «Мої організації». Його перекидує на сторінку організації. Користувач натискає на кнопку «Редагувати» і вводить нову інформацію про організацію (назва, опис, логотип і т. д.). Користувач нажимає на кнопку «Зберегти». Система зберігає нові дані.
Extension	Якщо користувач після натискання на кнопку редагувати не змінив поля, то кнопка «Зберегти» буде неактивною, також проводиться валідація полів за такими ж правилами, як при створенні.
Post-Condition	Дані організації успішно оновлено.

Таблиця 2.12 – Варіант використання UC-12

Use case name	Переглянути сторінку статичних черг
Use case ID	UC-12

Продовження таблиці 2.12.

Goals	Надати користувачам можливість переглядати сторінку статичних черг
Actors	Зареєстрований користувач
Trigger	Користувач хоче переглянути сторінку статичних черг.
Pre-conditions	Користувач зареєструвався на платформі.
Flow of Events	Користувач натискає на кнопку «Мої черги» на панелі управління у застосунку. Користувача перекидує на сторінку статичних черг.
Extension	-
Post-Condition	Користувач бачить сторінку статичних черг

Таблиця 2.13 – Варіант використання UC-13

Use case name	Переглянути статичні черги, як клієнт
Use case ID	UC-13
Goals	Надати користувачам можливість переглядати статичні черги, де він зареєстрований, як клієнт
Actors	Зареєстрований користувач
Trigger	Користувач хоче переглянути статичні черги, де він зареєстрований, як клієнт.
Pre-conditions	Користувач зареєстрований, як клієнт хоча б в одній черзі і знаходиться на сторінці «Мої черги».
Flow of Events	Користувач натискає на таб «Я клієнт» на сторінці «Мої черги». Система відображає перелік карток з чергами, у які записаний користувач.
Extension	-
Post-Condition	Користувач бачить перед собою перелік карток з короткою інформацією про черги, у яких він зареєстрований, як клієнт.

Таблиця 2.14 – Варіант використання UC-14

Use case name	Переглянути деталі черги і позицію в ній
Use case ID	UC-14
Goals	Надати користувачам можливість переглядати деталі черги і їх позицію, де вони зареєстровані, як клієнти.
Actors	Зареєстрований користувач
Trigger	Користувач хоче переглянути детальну інформацію про позицію в та деталі прийому в черзі, де він зареєструвався заздалегіть.
Pre-conditions	Користувач зареєстрований, як клієнт хоча б в одній черзі і знаходиться на сторінці «Мої черги». Користувач знаходиться на табі «Я клієнт».
Flow of Events	Користувач натискає кнопку «Детальніше» на картці черги, деталі якої хоче переглянути. Користувача перенапрвляє на сторінку конкретної черги.
Extension	-
Post-Condition	Користувач бачить перед собою детальну інформацію черги і свою позицію в ній.

Таблиця 2.15 – Варіант використання UC-15

Use case name	Знайти статичну чергу за ідентифікатором
Use case ID	UC-15
Goals	Надати користувачам можливість знаходити статичні черги за ідентифікаторами, який їм надає організація
Actors	Зареєстрований користувач
Trigger	Користувач хоче знайти статичну чергу за ідентифікатором, що йому надала організація.
Pre-conditions	Користувач зареєстрований, знаходиться на сторінці «Мої черги».
Flow of Events	Користувач натискає кнопку «Знайти за ідентифікатором» на сторінці «Мої черги», вводить наданий ідентифікатор в модальному вікні, що відкрилось та натискає кнопку «Перейти». Користувача перенапрвляє на сторінку черги.

Продовження таблиці 2.15.

Extension	Якщо чергу за цим ідентифікатором не знайдено, то на сторінці черги буде відображено помилку.
Post-Condition	Користувач бачить перед собою сторінку черги.

Таблиця 2.16 – Варіант використання UC-16

Use case name	Зареєструватись в чергу
Use case ID	UC-16
Goals	Надати користувачам можливість реєструватись у статичні черги самостійно
Actors	Зареєстрований користувач
Trigger	Користувач хоче зареєструватись в статичну чергу.
Pre-conditions	Користувач зареєстрований, знаходиться на сторінці черги, куди перейшов за ідентифікатором, прямим посиланням або через сторінку організації.
Flow of Events	Користувач обирає слот дати та часу, на який бажає бути зареєстрованим та натискає кнопку «Зареєструватись».
Extension	-
Post-Condition	Користувач бачить свою позицію в черзі. Черга відображається у загальному переліку на сторінці «Мої черги», у табі «Я клієнт»

Таблиця 2.17 – Варіант використання UC-17

Use case name	Переглянути статичні черги, як виконавець
Use case ID	UC-17
Goals	Надати користувачам можливість переглядати статичні черги, де вони зареєстровані, як виконавці
Actors	Зареєстрований користувач
Trigger	Користувач хоче переглянути перелік статичних черг, де він зареєстрований, як виконавець.
Pre-conditions	Користувач зареєстрований, знаходиться на сторінці «Мої черги».

Продовження таблиці 2.17.

Flow of Events	Користувач переходить на таб «Я виконавець» на сторінці «Мої черги».
Extension	-
Post-Condition	Користувач бачить перед собою сторінку з переліком карток з чергами, де він зареєстрований, як виконавець з короткою інформацією про чергу.

Таблиця 2.18 – Варіант використання UC-18

Use case name	Переглянути деталі черги і саму чергу клієнтів
Use case ID	UC-18
Goals	Надати користувачам можливість переглядати чергу клієнтів і деталі в статичних чергах, де вони зареєстровані, як виконавці.
Actors	Зареєстрований користувач
Trigger	Користувач хоче переглянути деталі і порядок статичної черги, де він зареєстрований, як виконавець.
Pre-conditions	Користувач зареєстрований, знаходиться на сторінці «Мої черги», на табі «Я виконавець».
Flow of Events	Користувач натискає кнопку «Детальніше» на картці черги, деталі якої бажає переглянути. Користувача перенаправляє на сторінку черги.
Extension	-
Post-Condition	Користувач бачить перед собою сторінку черги з деталями і чергу прийому.

Таблиця 2.19 – Варіант використання UC-19

Use case name	Переглянути запрошення в організації
Use case ID	UC-19
Goals	Надати користувачам можливість переглядати запрошення в організації.
Actors	Зареєстрований користувач

Продовження таблиці 2.19.

Trigger	Користувач хоче переглянути запрошення від організацій стати частиною штату.
Pre-conditions	Користувач зареєстрований
Flow of Events	Користувач натискає кнопку «Запрошення» на панелі управління у додатку, його перенаправляє на сторінку запрошень від організацій.
Extension	-
Post-Condition	Користувач бачить перелік запрошень від організацій.

Таблиця 2.20 – Варіант використання UC-20

Use case name	Прийняти запрошення
Use case ID	UC-20
Goals	Надати користувачам можливість приймати запрошення в організації.
Actors	Зареєстрований користувач
Trigger	Користувач хоче прийняти запрошення від організацій стати частиною штату.
Pre-conditions	Користувач зареєстрований та має хоча б одне запрошення від організації стати частиною штату. Користувач знаходиться на сторінці «Запрошення».
Flow of Events	Користувач натискає кнопку «Прийняти запрошення» на картці запрошення на сторінці «Запрошення». Користувача перекидує на сторінку організації.
Extension	-
Post-Condition	Користувача додано до організації з правами згідно з запрошенням.

Таблиця 2.21 – Варіант використання UC-21

Use case name	Переглянути профіль
Use case ID	UC-21
Goals	Надати користувачам можливість переглядати свій профіль.
Actors	Зареєстрований користувач
Trigger	Користувач хоче переглянути свій профіль.
Pre-conditions	Користувач зареєстрований.
Flow of Events	Користувач натискає кнопку з його аватаром, ім’ям та прізвищем на панелі управління. Користувача перекидує на сторінку «Профіль».
Extension	-
Post-Condition	Користувач бачить дані свого профілю на сторінці.

Таблиця 2.22 – Варіант використання UC-22

Use case name	Редагувати профіль
Use case ID	UC-22
Goals	Надати користувачам можливість редагувати свій профіль.
Actors	Зареєстрований користувач
Trigger	Користувач хоче переглянути відредагувати свій профіль.
Pre-conditions	Користувач зареєстрований. Користувач знаходиться на сторінці «Профіль».
Flow of Events	Користувач змінює дані профілю у формі редагування на сторінці та тисне кнопку «Підтвердити». Користувача перенаправляє назад по історії браузера.
Extension	Якщо нові дані профіля некоректні, користувач не зможе підтвердити форму, кнопка «Підтвердити» буде заблокована.
Post-Condition	Користувач бачить оновлені дані профілю на панелі управління та на сторінці «Профіль».

Таблиця 2.23 – Варіант використання UC-23

Use case name	Переглянути сторінку живих черг
Use case ID	UC-23
Goals	Надати користувачам можливість переглядати сторінку живих черг.
Actors	Зареєстрований користувач
Trigger	Користувач хоче переглянути сторінку живих черг.
Pre-conditions	Користувач зареєструвався на платформі.
Flow of Events	Користувач натискає на кнопку «Живі черги» на панелі управління у застосунку. Користувача перекидує на сторінку живих черг.
Extension	-
Post-Condition	Користувач бачить сторінку живих черг

Таблиця 2.24 – Варіант використання UC-24

Use case name	Переглянути живі черги, як виконавець
Use case ID	UC-24
Goals	Надати користувачам можливість переглядати живі черги, де вони зареєстровані, як виконавці
Actors	Зареєстрований користувач
Trigger	Користувач хоче переглянути перелік живих черг, де він зареєстрований, як виконавець.
Pre-conditions	Користувач зареєстрований, знаходиться на сторінці «Живі черги».
Flow of Events	Користувач переходить на таб «Я виконавець» на сторінці «Живі черги».
Extension	-

Продовження таблиці 2.24.

Post-Condition	Користувач бачить перед собою сторінку з переліком карток з чергами, де він зареєстрований, як виконавець з короткою інформацією про чергу.

Таблиця 2.25 – Варіант використання UC-25

Use case name	Додати клієнта до черги
Use case ID	UC-25
Goals	Надати користувачам можливість додавати клієнтів до живих черг, де вони зареєстровані, як виконавці
Actors	Зареєстрований користувач
Trigger	Користувач хоче додати до живої черги, в якій він є виковцем, нового клієнта.
Pre-conditions	Користувач зареєстрований, знаходиться на сторінці конкретної живої черги, де він зареєстрований, як виконавець.
Flow of Events	Користувач натискає на кнопку «Додати клієнта до черги». Користувач заповнює невелику форму, де вказує електронну пошту користувача, якого необхідно додати до черги. Користувач натискає кнопку «Підтвердити».
Extension	Якщо користувача з такою електронною поштою не існує у системі, то користувач побачить помилку додавання клієнту до черги.
Post-Condition	Клієнта додано на останню позицію у черзі. Позиція живої черги відображається у клієнта в додатку.

Таблиця 2.26 – Варіант використання UC-26

Use case name	Переглянути деталі живої черги і позиції клієнтів
Use case ID	UC-26
Goals	Надати користувачам можливість переглядати деталі і позиції клієнтів у живій черзі, де користувачі зареєстровані, як виконавці.
Actors	Зареєстрований користувач

Продовження таблиці 2.26.

Trigger	Користувач хоче переглянути деталі живої черги і позиції клієнтів у черзі, де він є виконавцем .
Pre-conditions	Користувач знаходиться на табі «Я виконавець» на сторінці «Живі черги».
Flow of Events	Користувач натискає на кнопку «Детальніше» на картці живої черги, деталі якої бажає переглянути. Користувача перекидує на сторінку конкретної живої черги.
Extension	-
Post-Condition	Користувач бачить деталі черги і позиції клієнтів на сторінці.

Таблиця 2.27 – Варіант використання UC-27

Use case name	Переглянути живі черги, як клієнт
Use case ID	UC-27
Goals	Надати користувачам можливість переглядати живі черги, у яких вони зареєстровані, як клієнти.
Actors	Зареєстрований користувач
Trigger	Користувач хоче переглянути живі черги, у яких він зареєстрований, як клієнт.
Pre-conditions	Користувач знаходиться на сторінці «Живі черги».
Flow of Events	Користувач натискає на кнопку «Детальніше» на картці живої черги, деталі якої бажає переглянути. Користувача перекидує на сторінку конкретної живої черги.
Extension	-
Post-Condition	Користувач бачить деталі черги і позиції клієнтів на сторінці.

Таблиця 2.28 – Варіант використання UC-28

Use case name	Переглянути деталі живої черги і позицію в ній
Use case ID	UC-28
Goals	Надати користувачам можливість переглядати деталі живої черги і їх позицію, де вони зареєстровані, як клієнти.
Actors	Зареєстрований користувач
Trigger	Користувач хоче переглянути детальну інформацію про позицію та деталі прийому в живій черзі.
Pre-conditions	Користувач зареєстрований, як клієнт хоча б в одній живій черзі і знаходиться на сторінці «Живі черги». Користувач знаходиться на табі «Я клієнт».
Flow of Events	Користувач натискає кнопку «Детальніше» на картці черги, деталі якої хоче переглянути. Користувача перенапрвляє на сторінку конкретної черги.
Extension	-
Post-Condition	Користувач бачить перед собою детальну інформацію черги і свою позицію в ній.

Таблиця 2.29 – Варіант використання UC-29

Use case name	Створити нову статичну чергу
Use case ID	UC-29
Goals	Надати користувачам можливість створювати статичні черги.
Actors	Зареєстрований користувач
Trigger	Користувач хоче створити нову статичну чергу.
Pre-conditions	Користувач зареєстрований. Користувач є в штаті хоча б одної організації.
Flow of Events	Користувач натискає кнопку «Нова черга» на панелі управління. Користувача перенапрвляє на форму створення черги, де він обирає організацію, де має бути створена черга, додає виконавців, розписує деталі обслуговування.
Extension	Якщо якісь з обов’язкових полів не заповнені, то форма поверне помилку і черга не буде створена.

Продовження таблиці 2.29.

Post-Condition	Нову чергу створено в рамках обраної організації, виконавців додано згідно заповненої форми.

Таблиця 2.30 – Варіант використання UC-30

Use case name	Створити нову живу чергу
Use case ID	UC-30
Goals	Надати користувачам можливість створювати живі черги.
Actors	Зареєстрований користувач
Trigger	Користувач хоче створити нову живу чергу.
Pre-conditions	Користувач зареєстрований. Користувач є в штаті хоча б одної організації.
Flow of Events	Користувач натискає кнопку «Нова жива черга» на панелі управління. Користувача перенапрвляє на форму створення черги, де він обирає організацію, де має бути створена черга, додає виконавців, розписує деталі обслуговування.
Extension	Якщо якісь з обов’язкових полів не заповнені, то форма поверне помилку і черга не буде створена.
Post-Condition	Нову живу чергу створено в рамках обраної організації, виконавців додано згідно заповненої форми.

Таблиця 2.30 – Варіант використання UC-30

Use case name	Створити нову живу чергу
Use case ID	UC-30
Goals	Надати користувачам можливість створювати живі черги.
Actors	Зареєстрований користувач
Trigger	Користувач хоче створити нову живу чергу.
Pre-conditions	Користувач зареєстрований. Користувач є в штаті хоча б одної організації.
Flow of Events	Користувач натискає кнопку «Нова жива черга» на панелі управління. Користувача перенапрвляє на форму створення черги, де він обирає організацію, де має бути створена черга, додає виконавців, розписує деталі обслуговування.

Продовження таблиці 2.30.

Extension	Якщо якісь з обов’язкових полів не заповнені, то форма поверне помилку і черга не буде створена.
Post-Condition	Нову живу чергу створено в рамках обраної організації, виконавців додано згідно заповненої форми.

Таблиця 2.31 – Варіант використання UC-31

Use case name	Авторизуватись, як адмін
Use case ID	UC-31
Goals	Надати адмінам можливість авторизовуватись за виданими системним адміністратором кредами.
Actors	Адмін
Trigger	Адмін хоче авторизуватись у адмінській панелі.
Pre-conditions	Системний адміністратор створив користувача з роллю адміна і передав пошту і пароль адміну. Адмін ще не авторизований і знаходиться на сторінці «Авторизація».
Flow of Events	Адмін вводить пошту і пароль, що видані системним адміністратором і тисне на кнопку «Авторизуватись». Адміна перенаправляє у внутрішню систему.
Extension	Якщо якісь з обов’язкових полів не заповнені, або пошта чи пароль неправильні, то форма поверне помилку.
Post-Condition	Адмін авторизований і перенаправлений у внутрішню систему, він не може перейти на сторінку авторизації, поки не вийшов з системи

Таблиця 2.32 – Варіант використання UC-32

Use case name	Переглянути список нових заявок на організації
Use case ID	UC-32
Goals	Надати адмінам можливість переглядати заявки, що ще не були взяті в обробку іншими адмінами.
Actors	Адмін
Trigger	Адмін хоче переглянути нові заявки на організації.

Продовження таблиці 2.32.

Pre-conditions	Адмін авторизований.
Flow of Events	Адмін натискає на кнопку «Доступні заявки» на панелі управління. Адміна перенаправляє на сторінку доступних заявок.
Extension	-
Post-Condition	Адмін бачить перелік карток з доступними заявками з їх коротким описом.

Таблиця 2.33 – Варіант використання UC-33

Use case name	Взяти заявку в обробку
Use case ID	UC-33
Goals	Надати адмінам можливість брати заявки в обробку.
Actors	Адмін
Trigger	Адмін хоче переглянути хоче взяти заявку в обробку.
Pre-conditions	Адмін авторизований. Адмін знаходиться на сторінці «Доступні заявки».
Flow of Events	Адмін натискає на кнопку «Взяти в обробку» на картці заявки, яку хоче взяти в обробку. Адміна перенаправляє на сторінку заявки з детальним описом та чатом з клієнтом.
Extension	-
Post-Condition	Статус заявки змінено на «В обробці». Заявка зникає з переліку доступних. Адмін має доступ до функцій комунікації з користувачем.

Таблиця 2.34 – Варіант використання UC-34

Use case name	Переглянути список заявок на організації в обробці
Use case ID	UC-34
Goals	Надати адмінам можливість переглядати заявки, що були взяті ними в обробку.

Продовження таблиці 2.34.

Actors	Адмін
Trigger	Адмін хоче переглянути заявки в обробці.
Pre-conditions	Адмін авторизований.
Flow of Events	Адмін натискає на кнопку «Обробка заявок» на панелі управління. Адміна перенаправляє на сторінку заявок, які він оброблює.
Extension	-
Post-Condition	Адмін бачить перелік карток з заявками в обробці з їх коротким описом.

Таблиця 2.35 – Варіант використання UC-35

Use case name	Переглянути детальну інформацію заявки
Use case ID	UC-35
Goals	Надати можливість адмінам переглядати деталі конкретної заявки на організацію.
Actors	Адмін
Trigger	Адмін бажає переглянути детальну інформацію заявки на організацію.
Pre-conditions	Адмін знаходиться на сторінці «Обробка заявок» або «Доступні заявки».
Flow of Events	Адмін натискає на кнопку «Детальніше» на картці заявки, яку бажає переглянути. Адміна перенаправляє на сторінку конкретної заявки, де він може побачити детальну інформацію по заявці і її статус.
Extension	-
Post-Condition	Адмін бачить сторінку конкретної заявки, де він може переглянути детальну інформацію по заявці і її статус.

Таблиця 2.36 – Варіант використання UC-36

Use case name	Схвалити заявку
Use case ID	UC-36
Goals	Надати можливість адмінам схвалювати заявки.
Actors	Адмін
Trigger	Адмін бажає схвалити заявку.
Pre-conditions	Адмін знаходиться на сторінці конкретної заявки. Адмін взяв заявку в обробку.
Flow of Events	Адмін натискає на кнопку «Схвалити заявку» на сторінці заявки. Відкривається модальне вікно, де адмін може написати коментар до схвалення заявки. Адмін натискає на кнопку «Схвалити» у модальному вікні.
Extension	-
Post-Condition	Статус заявки змінено на «Схвалено», організацію створено, користувача додано, як власника до організації.

Таблиця 2.37 – Варіант використання UC-37

Use case name	Відхилити заявку
Use case ID	UC-37
Goals	Надати можливість адмінам відхиляти заявки.
Actors	Адмін
Trigger	Адмін бажає відхилити заявку.
Pre-conditions	Адмін знаходиться на сторінці конкретної заявки. Адмін взяв заявку в обробку.
Flow of Events	Адмін натискає на кнопку «Відхилити заявку» на сторінці заявки. Відкривається модальне вікно, де адмін може написати коментар з причиною відхилення. Адмін натискає на кнопку «Відхилити» у модальному вікні.
Extension	-

Продовження таблиці 2.37.

Post-Condition	Статус заявки змінено на «Відхилено».

Таблиця 2.38 – Варіант використання UC-38

Use case name	Написати користувачу в чаті заявки
Use case ID	UC-38
Goals	Надати можливість адміну спілкуватись з користувачем стосовно заявки.
Actors	Адмін
Trigger	Адміністратор бажає написати повідомлення в чаті заявки зі зверненням до користувача.
Pre-conditions	Адмін взяв заявку в обробку, адмін знаходиться на сторінці конкреної заявки.
Flow of Events	Адмін залишає коментар у полі вводу і натискає кнопку «Відправити».
Extension	Якщо адмін нічого не написав, то кнопка буде заблокована
Post-Condition	Адмін бачить своє повідомлення в чаті, воно також відображається у користувача, що створив заявку.

2.2 Розроблення функціональних вимог

Програмне забезпечення розділене на модулі. Кожен модуль має свій певний набір функцій. В таблиці 2.39 наведено загальну модель вимог, а в таблиці 2.40 наведений опис функціональних вимог до програмного забезпечення. Матрицю трасування вимог можна побачити на рисунку 2.2.

Таблиця 2.39 – Загальна модель вимог

№	Назва	ID Вимоги	Пріоритети	Ризики
1	Перегляд посадкової сторінки.	FR-1	Низький	Низький
2	Обліковий запис користувача.	FR-2	Високий	Високий
2.1	Реєстрація користувача.	FR-3	Високий	Високий

Продовження таблиці 2.39.

2.2	Авторизація користувача.	FR-4	Високий	Високий
2.3	Вихід із акаунту.	FR-5	Середній	Низький
2.4	Редагування профілю	FR-6	Середній	Низький
3	Система заявок організацій	FR-7	Високий	Високий
3.1	Створення заявки	FR-8	Високий	Високий
3.2	Схвалення заявки	FR-9	Високий	Високий
3.3	Відхилення заявки	FR-10	Високий	Високий
3.4	Чат адміністрації і користувача	FR-11	Середній	Низький
4	Менеджмент організацій	FR-12	Високий	Високий
4.1	Редагування організації	FR-13	Середній	Низький
4.2	Додавання співробітників до штату	FR-14	Високий	Високий
5	Черги	FR-15	Високий	Високий
5.1	Створення, видалення і редагування статичних черг	FR-16	Високий	Високий
5.2	Створення і видалення живих черг	FR-17	Високий	Високий
5.3	Додавання клієнтів до живих черг виконавцями	FR-18	Високий	Високий
5.4	Самостійне додавання клієнтів до статичних черг	FR-19	Високий	Високий

Таблиця 2.40 – Перелік функціональних вимог

Назва	Опис
FR-1	Перегляд посадкової сторінки. Неавторизований користувач бачить посадкову сторінку. На ній же можна обрати можливість зареєструватися або авторизуватися.

Продовження таблиці 2.40.

FR-2	Обліковий запис користувача. Надається функціонал для управління обліковим записом, включаючи реєстрацію, авторизацію, вихід та редагування профілю.
FR-7	Система заявок організацій. Користувачі мають можливість створювати заявки на нові організації, а адміністрація підтверджувати, створюючи тим самим нові організації на платформі, або відхиляти їх. Також присутній мінімальний функціонал для комунікації адмін – користувач всередині самої платформи.
FR-12	Менеджмент організацій. Користувачі мають можливість управляти своїми організаціями, включаючи редагування деталей про організацію, менеджмент персоналу шляхом запрошення до організації або виключення з неї.
FR-15	Черги. Користувачі мають змогу створювати та редагувати живі/статичні черги всередині своїх організацій, додаючи до них необхідні подробиці обслуговування. Користувачі можуть записуватись в статичні черги самостійно за допомогою пошуку через ідентифікатор черги, в живі черги виконавець черги сам заносить своїх клієнтів.

	FR-1	FR-2	FR-3	FR-4	FR-5	FR-6	FR-7	FR-8	FR-9	FR-10	FR-11	FR-12	FR-13	FR-14	FR-15	FR-16	FR-17	FR-18	FR-19
UC-01	+																		
UC-02				+															
UC-03			+																
UC-04					+														
UC-05								+											
UC-06							+												
UC-07							+												
UC-08											+								
UC-09												+							
UC-10														+					
UC-11													+						
UC-12															+				
UC-13															+				
UC-14															+				
UC-15															+				
UC-16																			+
UC-17															+				
UC-18															+				
UC-19														+					
UC-20														+					
UC-21		+																	
UC-22						+													
UC-23															+				
UC-24															+				
UC-25																		+	
UC-26															+				
UC-27															+				
UC-28															+				
UC-29																+			
UC-30																	+		
UC-31				+															
UC-32							+												
UC-33							+												
UC-34							+												
UC-35							+												
UC-36									+										
UC-37										+									
UC-38											+								

Рисунок 2.2 – Матриця трасування вимог

2.3 Розроблення нефункціональних вимог

Передбачено такі ключові нефункціональні вимоги:

Безпека. Уся передача даних між клієнтом і сервером відбувається через HTTPS. Для авторизації застосовується пара JWT – access token із терміном дії 15 хвилин та refresh token із терміном 7 днів. Коли access token закінчується, клієнт автоматично надсилає refresh token, отримує новий access token і продовжує роботу без повторного входу. Якщо ж термін дії обох токенів вичерпано, користувач має повторно увійти. Система базово захищена від поширених атак. Next.js за замовчуванням екранує всі вставлені у JSX рядкові значення. Це означає, що якщо у змінній буде небезпечний HTML, він відобразиться як текст, а не виконуватиметься. Для базового захисту від атаки на сервер застовується Zod[6] для валідації всіх вхідних даних. Mongoose[7] захищає від NoSQL-ін’єкцій за допомогою параметризованого формування запитів значення поля передаються як параметр, а не як частина сирого рядка запиту. Замість того, щоб «рядково» складати фільтр, Mongoose розділяє структуру запиту і самі змінні.

Користувацький досвід. Структура сайту має бути простою: не більше трьох рівнів вкладеності, щоб користувач міг дістатися будь-якої сторінки за максимум три кліки. Перехід між сторінками відбувається без повного перезавантаження, а якщо запит на сервер триває довше ніж кілька сотень мілісекунд, показується прелоудер, щоб дати зрозуміти, що дані завантажуються. Кожна дія користувача, що змінює стан системи, супроводжується тост-повідомленням, яке автоматично зникає через кілька секунд, а оновлення змінного контенту відбувається, забезпечуючи актуальність інформації без ручного перезавантаження.

Платформенна сумісність. Система повинна коректно працювати в браузерах із підтримкою ECMAScript 2015 (ES6), зокрема в Chrome починаючи з версії 51, Firefox з версії 54, Edge з версії 15, Safari з версії 10.1, Opera з версії 38, оскільки всі необхідні поліфіли розраховані саме на цей рівень стандарту.

Масштабованість. Система, організована як монорепозиторій на Turborepo[8], що забезпечує можливість незалежного масштабування сервісів без потреби переукладання всього проєкту: кожен сервіс знаходиться в спільній екосистемі, при цьому зберігаючи незалежність одне від одного, що дозволяє деплоїти й додавати інстанси окремо одне від одного. Turborepo-кешування артефактів і інкрементальне складання скорочують час збірки при розширенні коду. У разі подальшого створення дзеркал для більшого просування сервісу, можливе використання вже готових пакетів і API.

2.4 Аналіз системних вимог

Веб-застосунок «E-queue» призначений для оптимізації електронних черг та управління прийомом клієнтів, тому його компоненти мають працювати на максимально широкому колі пристроїв і забезпечувати стабільну продуктивність як на стороні користувача, так і в серверній інфраструктурі. З одного боку, клієнтська частина повинна коректно відображатися і реагувати на дії у будь-якому сучасному браузері; з іншого —

серверні сервіси мають витримувати одночасні запити в режимі реального часу, зберігаючи цілісність даних і забезпечуючи швидку обробку операцій.

Клієнтська частина «E-queue» побудована як SPA на базі Next.js[9] із підтримкою WebSocket-підключень. Для коректної роботи достатньо будь-якого браузера, що підтримує ECMAScript 2015+, Service Workers та WebSocket API (наприклад, Chrome 90+, Firefox 88+, Edge 90+, Safari 14+). Мінімальні апаратні параметри на рівні клієнта — одноядерний процесор з тактовою частотою від 1,8 ГГц та 2 ГБ оперативної пам’яті, а також стійке інтернет-з’єднання зі швидкістю від 5 Мбіт/с для зручного підвантаження інтерфейсних ресурсів. Для комфортної роботи рекомендується обладнання з двоядерним CPU, 4 ГБ RAM і каналом 20 Мбіт/с, що знизить затримки при оновленні позиції в черзі та переході між сторінками.

2.5 Аналіз економічних показників програмного забезпечення

Для оцінки вартості розробки веб-застосунку «E-queue» використано метод Use Case Points. У ролі акторів виступають відвідувач черги з вагою 3 бали та адміністратор із вагою 2 бали, що дає Unadjusted Actor Weight на рівні 5 балів. Загальна кількість прецедентів використання становить 19, серед яких виділено сім базових сценаріїв із невеликою кількістю транзакцій, вісім кейсів середньої складності та чотири високоскладні сценарії, що сумарно формує Unadjusted Use Case Weight у 175 балів і безкоригувальний показник UUCP = 180 Use Case Points.

З урахуванням технічного коефіцієнта складності (TCF = 0,85), який враховує особливості алгоритмів динамічного та статичного розподілу черг, та екологічного коефіцієнта середовища (EF = 1,00) скоригована величина Use Case Points становить 153. При Productivity Factor на рівні 18 людино-годин на одну UCP загальний обсяг робіт оцінюється в приблизно 2 754 людино-години.

За погодинної ставки 20 € це дає орієнтовну собівартість проєкту близько 55 080 €.

2.6 Постановка завдання на розробку програмного забезпечення

Проблема, що розглядається, полягає в недостатній ефективності існуючих систем електронних черг: відсутність чіткого інформування клієнтів про зміни в порядку обслуговування та затримки через невідповідність реального часу заявленому (наприклад, клієнти приходять на заброньований слот, але перед ними ще залишається велика жива черга). Це призводить до зниження задоволеності користувачів і втрати потенційних клієнтів.

Метою розробки є створення веб-застосунку «E-queue», який дозволить організаціям гнучко керувати своїми службами прийому, формувати й редагувати власні черги (динамічні та статичні), а клієнтам — відстежувати свою позицію в режимі реального часу. Застосунок міститиме єдину платформу для реєстрації організацій, управління персоналом і їх ролями, налаштування переліку послуг із можливістю прикріплення текстових описів, документів, фото та відео, а також інструменти адміністрування заявок організацій.

Реалізація передбачає розробку клієнтського інтерфейсу на базі Next.js із підтримкою WebSocket для миттєвого оновлення станів черги та серверу на NestJS[10], який оброблятиме як REST-запити, так і події реального часу.

Висновки до розділу

У розділі було детально описано функціональні вимоги до системи «E-queue» на основі аналізу бізнес-процесів і користувацьких сценаріїв. Було сформульовано ключові Use Case, серед яких створення та редагування організацій, управління динамічними й статичними чергами, ролями й сервісами, а також адміністрування заявок. На їхній базі виокремлено 19 функціональних вимог, які покривають усі визначені процеси та забезпечують прозору взаємодію між користувачами і системою.

Далі розроблено загальну модель вимог у нотації UML, що структурує відношення між актором, діями й даними, та виконано матрицю трасування,

яка підтвердила повне покриття вимог Use Case. Це дозволило виявити й усунути потенційні прогалини, а також заклало міцний фундамент для подальшого технічного проєктування архітектури та інтерфейсних компонентів. В результаті реалізовано чітку й зрозумілу основу для переходу до розробки детальної технічної та програмної реалізації.

За результатами розділу сформовано технічне завдання на розробку програмного забезпечення.

3 КОНСТРУЮВАННЯ ТА РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Архітектура програмного забезпечення

Загальна архітектура застосунку побудована за клієнт-серверним принципом: браузер взаємодіє з сервером через HTTP-запити, отримуючи готові HTML-сторінки або JSON-дані. Кодова база побудована як модульний моноліт, у якому весь код розміщено в єдиному репозиторії під управлінням Turborepo. Це дозволяє зберегти простоту розгортання та спільну інфраструктуру для клієнту й серверу, водночас чітко розділяючи доменні області на незалежні модулі. У корені репозиторію знаходяться два основні застосунки: веб-інтерфейс, реалізований на Next.js, і серверна частина, побудована на NestJS. Сторонній спільний функціонал — типи TypeScript[11], утиліти для роботи з API, загальні константи і UI-компоненти — виноситься в окремі пакети на рівні репозиторію, що забезпечує повторне використання коду у серверній і клієнтській частинах та підтримку єдиного стилю.

Серверна частина організована за традиційним підходом NestJS: кожна сутність представлена власним незалежним модулем із наборами контролерів, сервісів і схем Mongoose для взаємодії з MongoDB[12]. Кожен модуль чітко визначає свої провайдери (сервіси), контролери та ті провайдери, які він експортує іншим модулям, що забезпечує ізольованість внутрішньої логіки й можливість повторного використання компонентів. Комунікація між клієнтом і сервером здійснюється через REST-інтерфейс, що дозволяє легко додавати нові ендпоінти та розгортати додаткові модулі без зміни базової інфраструктури, а також дає більш зручний для документації і взаємодії інтерфейс порівняно з RPC чи GraphQL[13].

Архітектура інжекції залежностей у NestJS базується на єдиному контейнері провайдерів, який автоматично створює та передає потрібні сервіси через параметри конструктора. Будь-який сервіс, позначений як провайдер, реєструється в модулі – усі залежності вирішуються під час

створення контролера або іншого сервісу, тому розробнику не потрібно вручну створювати об’єкти чи слідкувати за їхнім життєвим циклом. За замовчуванням кожен провайдер існує як сінглтон, але за потреби можна задати область дії «на запит» або «транзієнтно», щоб отримувати новий екземпляр при кожному виклику.

При масштабуванні або розширенні функціоналу новий модуль просто імпортує необхідні модулі, отримуючи доступ до всіх їхніх експортованих провайдерів. Оскільки залежності працюють через абстракції, а не через жорсткі реалізації, можна легко підміняти сервіси під час тестування або змінювати спосіб збереження даних (наприклад, замінити MongoDB на інше сховище), не змінюючи код клієнтів. Такий підхід забезпечує високу модульність, простоту тестування й можливість поступово нарощувати складність системи без переписування вже існуючих підсистем.

Доступ до даних реалізовано через MongoDB із використанням бібліотеки Mongoose, що автоматично керує схемами й валідацією документів. У кожному модулі, який працює з базою, через DI-контейнер NestJS підключаються відповідні моделі Mongoose, що дозволяє сервісам оперувати колекціями без додаткового налаштування клієнта. Модуль аутентифікації працює на основі JWT: під час логіну генерується токен із інформацією про ідентифікатор користувача та його ролі, а для захисту маршрутів використовуються вбудовані Guards, які автоматично перевіряють валідність токена та дістають дані про користувача з бази.

Для графічного представлення архітектурк додатку було візуалізовано на трьох рівнях діаграм C4, наведених на рисунках 3.1-3.3.

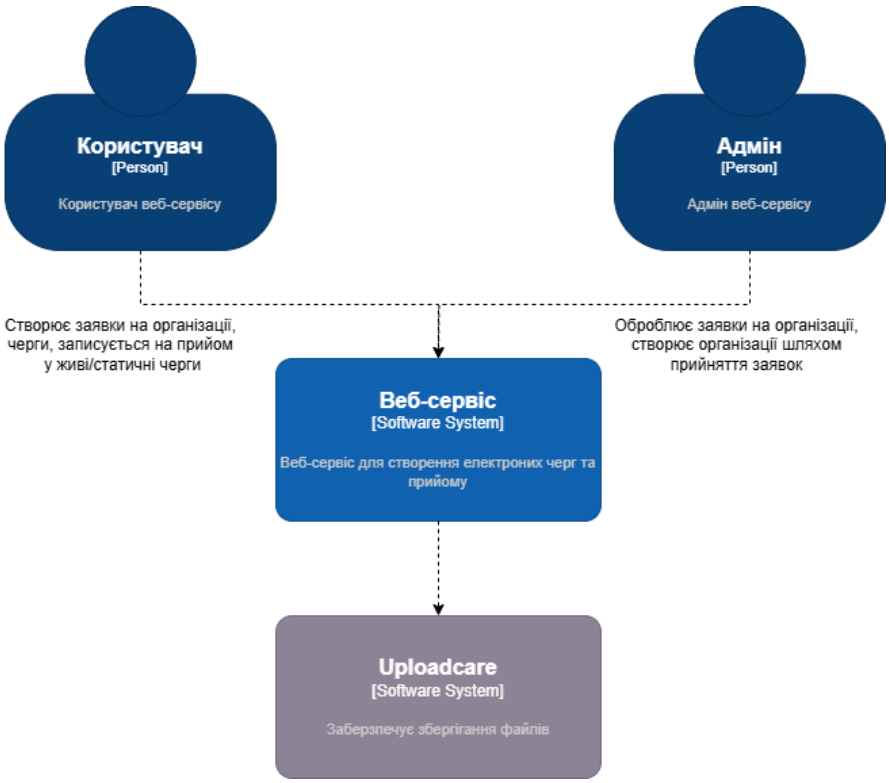


Рисунок 3.1 – Діаграма 1 рівня C4

На діаграмі першого рівня зображено контекстну модель системи «E-queue» і її ключові зовнішні зв’язки. У центрі стоїть сам веб-сервіс, призначений для створення електронних черг і обробки заявок. Зліва від нього розташований актор «Користувач», який через браузер взаємодіє із системою, надсилаючи заявки на створення організацій, формуючи черги (живі чи статичні) та записуючись на прийом. Праворуч показаний актор «Адмін», що володіє підвищеними правами: він переглядає надіслані користувачами заявки, приймає або відхиляє їх і таким чином створює організації в системі. Також використовується зовнішній сервіс Uploadcare для зберігання й обслуговування файлів (документів, зображень), що прикріплюються до записів про послуги та організації.

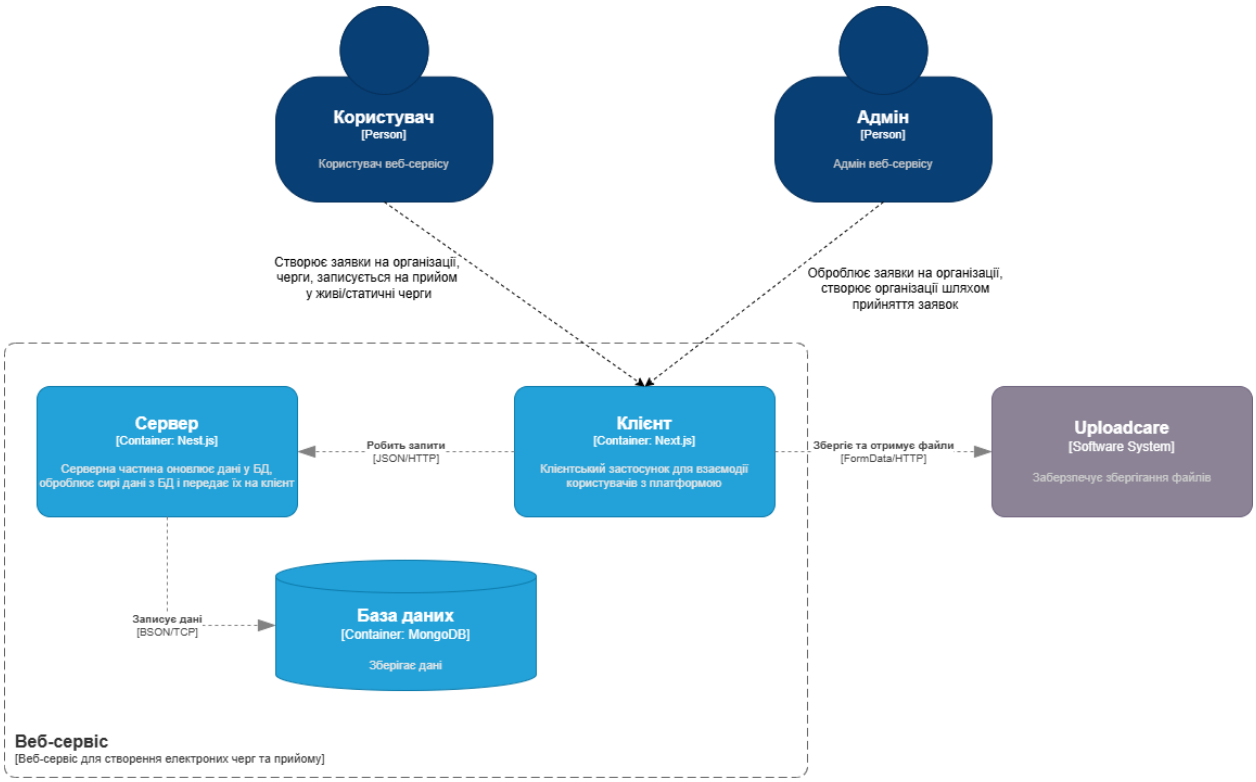


Рисунок 3.2 – Діаграма 2 рівня C4

На діаграмі другого рівня показано розбиття сервісу на чотири ключові складові й як вони взаємодіють із зовнішнім середовищем. Виділено три контейнери: клієнтська частина, сервер та база даних.

Клієнт - Next.js-застосунок. Саме через нього обидва актори — користувач і адміністратор — взаємодіють із платформою: він відповідає за відображення форм реєстрації організацій та черг, побудову інтерфейсу запису в живі або статичні черги та надсилання всіх запитів на сервер у форматі JSON по HTTP. Цей контейнер може працювати як у режимі серверного рендерингу, так і як односторінковий застосунок, що дає гнучкість у масштабуванні та кешуванні.

Сервер реалізовано на NestJS. Він приймає всі клієнтські запити, проводить валідацію й авторизацію (JWT), обробляє бізнес-логіку (створення й обробка заявок, управління ролями, формування черг) і взаємодіє з базою даних. Для зберігання файлів, прикріплених до організацій чи послуг, сервер формує FormData-запити до зовнішнього сервісу Uploadcare[14] (по HTTP), а

для зчитування й запису сутностей користується Mongoose, що обмінюється з MongoDB у форматі BSON через TCP.

База даних - MongoDB. Саме тут зберігаються всі документи: користувачі, організації, статуси заявок, налаштування черг та метадані прикріплених файлів. Завдяки схемам Mongoose забезпечується структурована валідація на рівні бази, а реплікація й резервне копіювання гарантують надійність даних.

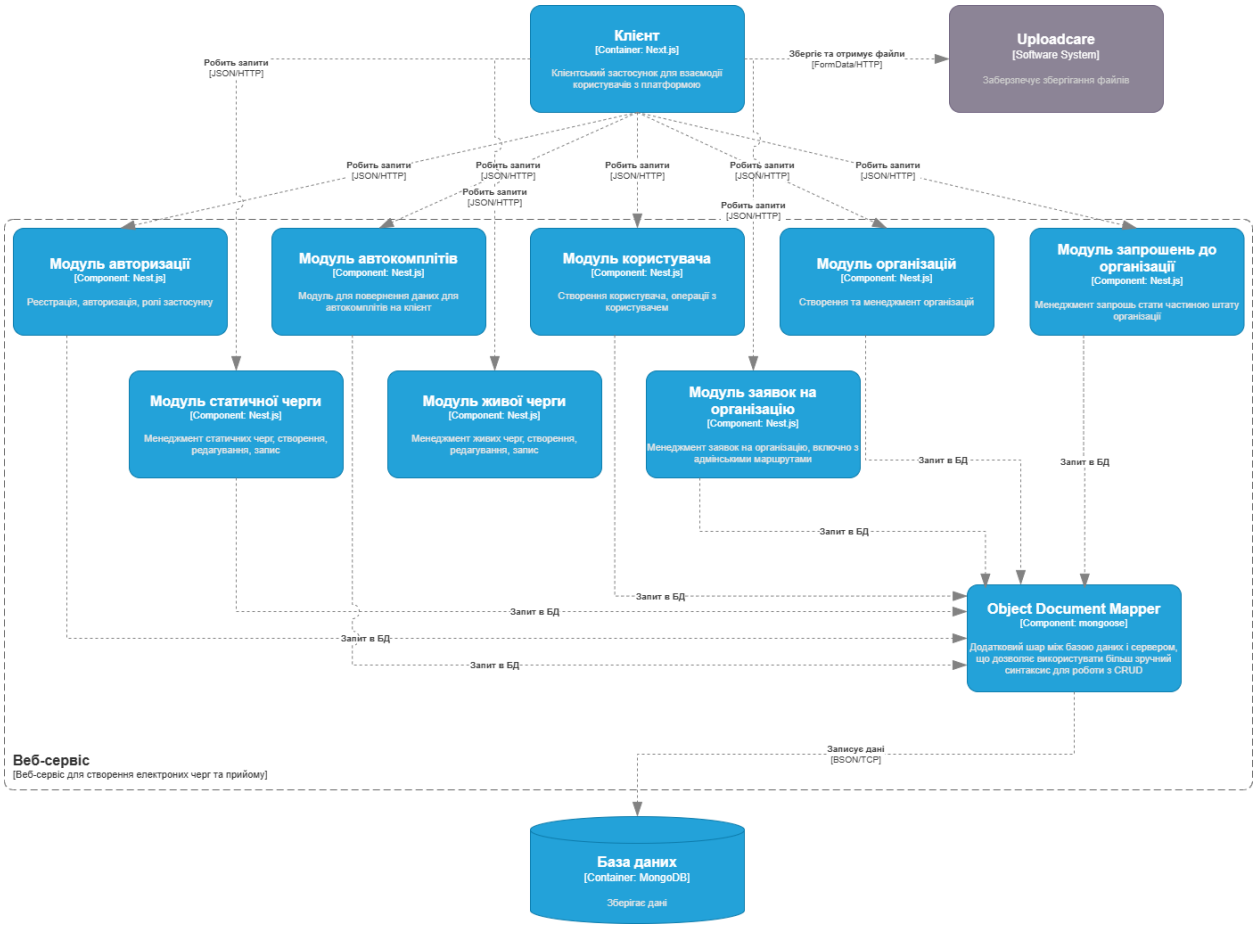


Рисунок 3.3 – Діаграма 3 рівня С4

На цій діаграмі показано, як на серверній стороні організовано модулі, що реалізують бізнес-функції «E-queue».

Усі клієнтські запити у форматі JSON/HTTP надходять з Next.js-застосунку безпосередньо до контролерів відповідних модулів сервера. Модуль авторизації відповідає за реєстрацію користувачів, перевірку прав (JWT) та розподіл ролей. Модуль автокомплітів повертає дані для підказок у формі введення (наприклад, тип організації, бажаний тип зв’язку). Модуль

користувача відповідний за створення і оновлення профілів користувачів, модуль організацій забезпечує повний цикл CRUD-операцій над організаціями, а модуль запрошень до організації обробляє запити про приєднання працівників до штату.

Для роботи з чергами виділено два окремі компоненти: модуль статичної черги керує списками очікування з фіксованим часом прийому, а модуль живої черги реалізує FIFO-чергу. Окремий модуль заявок на організацію інкапсулює процес подачі та адмінської обробки заявок, об'єднуючи користувцьки та адмінські маршрути.

Усі ці серверні модулі звертаються до єдиного шару Object Document Mapper (Mongoose), який перетворює методи CRUD у BSON-запити по TCP до MongoDB. Завдяки такому розподілу відповідальності кожний компонент чітко фокусується на власній предметній області, а Mongoose-ODM гарантує узгодженість даних у базі, а також допомагає більше адаптувати використання MongoDB під NestJS.

3.2 Архітектурні рішення та обґрунтування вибору засобів розробки

У системі «E-queue» обрано власну модель аутентифікації з використанням JWT і зберіганням облікових записів у базі даних. Кожен користувач при реєстрації передає дані, які верифікуються на сервері, а після успішного логіну формується токен із закодованим ідентифікатором і ролями. Такий підхід дозволяє відмовитися від зовнішніх SSO-провайдерів, зберігаючи повний контроль над одним з найбільш важливих безпекових аспектів, а також спрощує масштабування системи в рамках монолітного бекенду без необхідності додаткових інтеграцій із зовнішніми сервісами авторизації. Інтеграції з іншими системами зведено до мінімуму: обмін сповіщеннями та документами відбувається через внутрішні REST-ендпоінти, доступні за HTTPS із взаємною аутентифікацією через JWT. Безпека гарантується використанням стандартних засобів захисту HTTP-запитів (HTTPS, CORS-політики).

У якості основного сховища обрано базу даних MongoDB. Архітектура MongoDB дозволяє швидко додавати нові поля без необхідності складних міграцій і має високу продуктивність при індексуванні великих колекцій. MongoDB використовує документно-орієнтований підхід, що природно відображає JSON-структури сутностей, покращуючи тим самим Developer Experience і знижуючи ризики помилки при розробці, а також забезпечує просту і безшовну інтеграцію з Node.js/NestJS через Mongoose для швидкої розробки CRUD-логіки.

Технологічний стек об’єднує React з Next.js на клієнті для забезпечення серверного рендерингу та SEO-оптимізації, а на сервері використано NestJS через його модульну структуру, гарне поєднання Node.js з клієнтом в рамках монорепозиторія, підтримку TypeScript і вбудовані механізми валідації та обробки помилок. Обидві частини розгорнуті в єдиному монорепозиторії Turborepo, що уніфікує конфігурації, прискорює і кешує збірку, надає більшу структурованість коду за допомогою системи пакетів, універсальних часток коду, наприклад: API, роутинг, UI-бібліотека, утиліти, константи, тощо, які пишуться розробником і використовуються всередині окремих незалежних частинах додатку (наприклад клієнт і сервер). TypeScript обрано як єдину мову для всього стеку задля типобезпечності й покращення Developer Experience.

У якості IDE обрано Visual Studio Code[15] з набором розширень для ESLint[16], Prettier[17] та Debugger for Chrome для дебагінгу і відображення помилок до збірки. У порівнянні з альтернативами (WebStorm[18]), Visual Studio Code легший, швидший і більш гнучкий, що дозволяє підлаштувати його під конкретний проєкт.

3.3 Конструювання програмного забезпечення

У серверній частині «E-queue» реалізовано власні модулі обробки черг, які комбінують класичний алгоритм FIFO для живих черг і тайм-слотне призначення для статичних черг.

Структури даних у бекенді визначені через Mongoose-схеми та відповідають колекціям MongoDB. Кожна сутність має набір атрибутів із суворими валідаційними правилами. Концептуальна модель бази даних передбачає зв’язки через референси — користувачі реферують до організацій, а квитки до конкретної черги. Логічна схема охоплює індекси за полями статусу та часу створення для пришвидшення вибірок, а фізична модель оптимізована під горизонтальне масштабування шардінгом.

Клієнт побудовано на Next.js із TypeScript, де кожен модуль сторінки відповідає за окремий функціонал: реєстрація, адміністрування організацій, перегляд черг і т. д. Компоненти оформлені за допомогою React Hooks і контексту для керування станом, а API-запити виконуються через Axios із централізованою обробкою помилок. Усі збірки та тестування організовані в монорепозиторії на Turborepo: фронтенд і бекенд мають спільні пакети з типами, константами й утилітами.

3.3.1 Опис структури бази даних

В якості системи управління базами даних використовується MongoDB. Опис документів бази даних наведено у таблицях 3.1-3.14. Модель бази даних наведена на рисунку 3.1.

Таблиця 3.1 – Опис таблиці organization_invites

Назва поля	Тип даних	Опис
_id	ObjectId	Унікальний ідентифікатор запрошення
user_id	ObjectId	Посилання на користувача, якого запрошено в організацію
organization_id	ObjectId	Посилання на організацію, в яку запрошується користувач
invitation_role	String	Роль, на яку запрошують користувача

Продовження таблиці 3.1.

user_email	String	Електронна пошта користувача

Таблиця 3.2 – Опис таблиці organization_requests

Назва поля	Тип даних	Опис
_id	ObjectId	Унікальний ідентифікатор запиту
organization_logo	String	URL-посилання на логотип організації
organization_type_id	ObjectId	Посилання на інший документ, тип організації
organization_title	String	Назва організації
desired_connection	String	Бажаний спосіб комунікації, через який адміністрація зв’яжеться з організацію
organization_description	String	Опис організації
organization_website	String	URL-посилання на сайт організації
attachments	String[]	Масив URL-посилань на файли або зображення, прикріплені до заявки
user_id	ObjectId	Посилання на користувача, який залишив заявку
status	String	Статус запиту
admin_id	ObjectId	Посилання на адміна, що обробляє запит

Продовження таблиці 3.2.

comments	String[]	Масив повідомлень у чаті, між заявником і адміністратором
rejection_comment	String	Коментар у випадку відмови
approval_comment	String	Коментар у випадку підтвердження
createdAt	Date	Час створення запису
updatedAt	Date	Час оновлення запису

Таблиця 3.3 – Опис таблиці organizations

Назва поля	Тип даних	Опис
_id	ObjectId	Унікальний ідентифікатор організації
organization_logo	String	URL-посилання на логотип організації
organization_type_id	ObjectId	Посилання на тип організації
organization_title	String	Назва організації
organization_description	String	Опис організації
organization_website	String	URL-посилання на сайт організації
members	ObjectId[]	Масив посилань на співробітників організації
static_queues	ObjectId[]	Масив посилань на статичні черги організації

Продовження таблиці 3.3.

dynamic_queues	ObjectId[]	Масив посилань на динамічні черги організації

Таблиця 3.4 – Опис таблиці user_organizations

Назва поля	Тип даних	Опис
_id	ObjectId	Унікальний ідентифікатор таблиці
user_id	ObjectId	Посилання на користувача
organization_id	ObjectId	Посилання на організацію
organization_role	String	Роль організації

Таблиця 3.5 – Опис таблиці users

Назва поля	Тип даних	Опис
_id	ObjectId	Унікальний ідентифікатор користувача
name	String	Ім’я користувача
surname	String	Прізвище користувача
email	String	Електронна пошта користувача
password	String	Хешований пароль користувача
role	String	Роль користувача

Таблиця 3.6 – Опис таблиці static_queues

Назва поля	Тип даних	Опис
_id	ObjectId	Унікальний ідентифікатор статичної черги
title	String	Назва послуги для черги
organization_id	ObjectId	Посилання на організацію, за якою закріплено чергу
days_of_service	String[]	Робочі дні, у які доступні слоти для черги
work_start_time	String	Час початку робочого дня
work_end_time	String	Час кінця робочого дня
work_break_start_time	String	Час початку перерви у робочий день
work_break_end_time	String	Час кінця перерви у робочий день
work_time_estimation	Number	Очікуваний час обробки одного клієнта
break_time_estimation	Number	Очікуваний час перерви між клієнтами
executor	ObjectId	Посилання на виконавця послуги, що обслуговує чергу
price	Number	Ціна послуги у гривнях

Продовження таблиці 3.6.

description	String	Опис послуги, на яку надається черга
forms_examples	String[]	Масив URL на приклади форм, що мають бути заповнені перед наданням послуги
forms_completed_examples	String[]	Масив URL на приклади заповнених форм, що мають бути заповнені клієнтом перед наданням послуги
attachments	String[]	Масив URL з додатками до черги
appointments	ObjectId[]	Масив посилань на записи на прийом клієнтів

Таблиця 3.7 – Опис таблиці dynamic_queues

Назва поля	Тип даних	Опис
_id	ObjectId	Унікальний ідентифікатор динамічної черги
title	String	Назва черги
organization_id	String	Посилання на організацію, за якою закріплено чергу
work_start_time	String	Час початку робочого дня
work_end_time	String	Час кінця робочого дня

Продовження таблиці 3.7.

break_time_estimation	Number	Очікуваний час перерви між клієнтами
executor	ObjectId	Посилання на виконавця послуги, що обслуговує чергу
description	String	Опис послуги, на яку надається черга
price	Number	Ціна послуги у гривнях
appointments	ObjectId[]	Масив посилань на користувачів, що зареєстровані в черзі

Таблиця 3.8 – Опис таблиці appointments

Назва поля	Тип даних	Опис
_id	ObjectId	Унікальний ідентифікатор запису
user_id	ObjectId	Посилання на користувача, що записався
queue_id	ObjectId	Посилання на статичну чергу, за якою закріплений запис
date	Date	Дата, на яку зроблено запис
start_time	String	Очікуваний початок обслуговування
end_time	String	Очікуваний кінець обслуговування

Таблиця 3.9 – Опис таблиці organization_types

Назва поля	Тип даних	Опис
_id	ObjectId	Унікальний ідентифікатор типу організації
title	String	Тип організацій

Модель бази даних з представленням взаємодії документів винесена у графічні матеріали, креслення 3.

Опис утиліт, бібліотек та іншого стороннього програмного забезпечення, що використовується у розробці наведено в таблиці 3.10.

Таблиця 3.10 – Опис утиліт

№ п/п	Назва утиліти	Опис застосування
1	Visual Studio Code	Використовується як основна IDE для розробки обох частин застосунку (frontend і backend). Завдяки великій кількості розширень
2	Postman	Застосовується для тестування REST-ендпоінтів серверної частини.
3	Mongoose	Забезпечує зручну роботу з MongoDB у Node.js, дозволяючи створювати схеми, валідувати та маніпулювати даними як з об'єктами.
4	NestJs	Каркас у стилі «модуль–контролер–сервіс» для організації серверної логіки, валідації через Zod і захисту маршрутів за допомогою JWT.

Продовження таблиці 3.10.

5	Next.js	Застосовується для побудови React-інтерфейсу, file-based routing сторінок і можливого використання API Routes.
6	Zod	Схема-валідатор для опису форм і DTO, що використовується і на сервері (Pipe у NestJS), і в React-формах для перевірки даних перед відправкою.
7	Axios	HTTP-клієнт на для виконання запитів до серверу NestJS, з інтерсепторами, які автоматично додають JWT у заголовки.
8	HeroUI	Набір готових, адаптивних UI-компонентів, створених на основі Tailwind CSS, що пришвидшує розробку інтерфейсів.
9	Tailwind CSS	CSS-фреймворк з утилітарним підходом, де кожен клас — це окрема властивість стилю, що дозволяє швидко компонувати дизайн.
10	React Query	Бібліотека для запитів і кешування асинхронних даних на клієнті.
11	JWT	Механізм генерації та перевірки токенів для аутентифікації користувачів, який забезпечує безпечний доступ до захищених ендпоінтів.
12	argon2	Алгоритм хешування паролів на сервері, який гарантує безпечне збереження й перевірку облікових даних користувачів.
13	husky	Налаштування Git-хуків, які запускають ESLint, Prettier перед комітом чи пушем.

Продовження таблиці 3.10.

14	react-hook-form	Бібліотека для створення форм на клієнті, керування станом і валідації React-форм із мінімальною кількістю ререндерів, інтегрується із Zod.
15	framer-motion	Інструмент для додавання плавних анімацій у React-компоненти.

3.4 Аналіз безпеки даних

Вразливості в коді та залежностях проєкту «E-queue» регулярно виявляються й усуваються ще на етапі розробки. Усі npm-пакети автоматично перевіряються при кожному оновленні через npm audit, а знайдені критичні вразливості блокуються до встановлення безпечних версій.

Статичний аналіз коду здійснюється за допомогою ESLint-плагінів: вони виявляють ризикові конструкції JavaScript/TypeScript, потенційні витоки даних і невідповідності стилю, які можуть призвести до помилок. Динамічне тестування REST-інтерфейсів організовано через Postman Collections, що імітують як коректні, так і навмисно некоректні запити для перевірки захисту від CSRF, XSS та NoSQL-ін’єкцій.

Одним із ключових векторів загроз є несанкціонований доступ до облікових записів, що може виникнути через використання викрадених JWT-токенів, брутфорс-атаки або випадкове витікання секретів. Для мінімізації цього ризику в «E-queue» на боці сервера реалізовано автентифікацію через JWT-токени з обмеженим терміном дії, підписані секретним ключем, який жодного разу не зберігається в системі контролю версій, а лише у змінних оточення.

Паролі користувачів та інші критичні дані, що не потребують відображення на клієнті, зберігаються в хешованому вигляді із застосуванням argon2.

Завантажені користувачами файли зберігаються в Uploadcare із обмеженими правами доступу. Посилання на файли мають додатковий хеш, що унеможливлює знайдення посилання на файл шляхом перебору.

Оскільки сервіс наразі не розгортається в публічному хмарному середовищі та не контейнеризується, основну увагу приділено безпеці серверної інфраструктури. Всі з’єднання з API захищені TLS-шифруванням, а доступ до портів обмежується локальною мережею та файрволом.

Висновки до розділу

Було реалізовано безпосередню розробку програмного забезпечення системи «E-queue», що спрямована на оптимізацію електронних черг в організаціях. Робота розпочалась із побудови загальної архітектури проєкту. Обґрунтовано вибір архітектурного підходу — клієнт-серверної моделі, яка поєднує в собі гнучкість у масштабуванні, зручність у розгортанні та ефективність комунікації між компонентами системи. Для реалізації клієнтської частини було обрано Next.js, що дозволило реалізувати сучасний і зручний інтерфейс, адаптований під різні пристрої. Серверну частину реалізовано з використанням NestJS, що забезпечує модульність, підтримку REST API та WebSocket, необхідних для обробки подій у реальному часі.

У процесі конструювання було детально опрацьовано структуру бази даних: визначено ключові сутності (користувач, організація, черга, запис тощо), їх атрибути та взаємозв’язки. Описано ER-діаграму, що відображає логіку взаємодії між елементами системи. Також було розроблено механізми управління ролями користувачів, що дозволяють реалізувати контроль доступу до функціоналу системи на основі ролей (власник, менеджер, виконавець, клієнт).

Особливу увагу приділено питанням безпеки — реалізовано механізми авторизації та автентифікації з використанням JWT, обмеження доступу до приватних маршрутів, а також системи перевірки прав доступу. Розглянуто й заходи щодо захисту бази даних і мережевої взаємодії.

Таким чином, продемонстроване завершення фази активної розробки системи, поєднуючи теоретичні напрацювання попередніх розділів із їх практичною реалізацією у вигляді функціонального програмного забезпечення. Це створює ґрунт для подальших етапів — тестування, оцінки якості, розгортання та експлуатації.

4 АНАЛІЗ ЯКОСТІ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Аналіз якості ПЗ

Для забезпечення високої якості програмного забезпечення було проведено автоматизований аналіз коду за допомогою платформи DeepSource. Цей інструмент дозволяє виявляти потенційні помилки, проблеми з продуктивністю, антипатерни, недоліки стилю, проблеми з безпекою та інші важливі аспекти, що можуть вплинути на надійність та підтримуваність проєкту. На рисунку 4.1 наведено результати останнього аналізу:

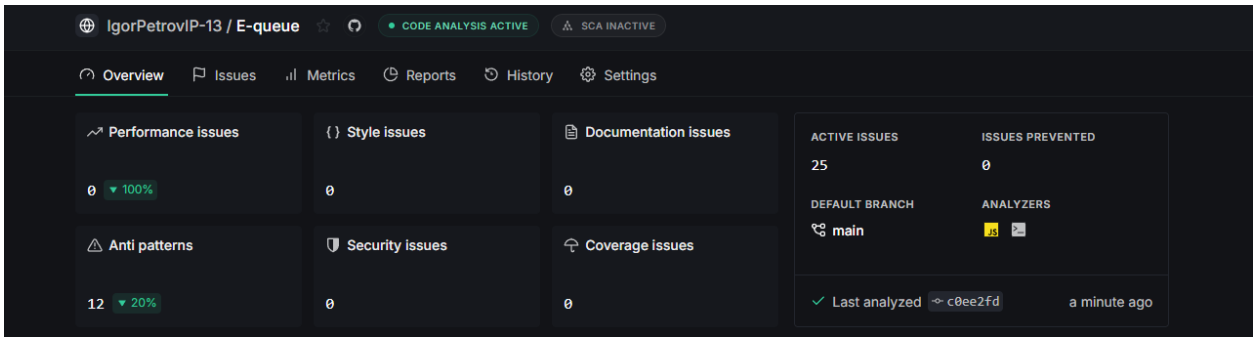


Рисунок 4.1 – Результати аналізу коду через DeepSource

За результатами останнього сканування, було виявлено 25 активних проблем, з яких найбільшу частину становлять антипатерни — 12 випадків. Це на 20% менше, ніж під час попередньої перевірки, що свідчить про позитивну динаміку. Усі інші категорії проблем, включно з помилками продуктивності, стилю, безпеки, документації та покриття, показали нульовий результат, тобто жодної проблеми в цих напрямках не зафіксовано. Зокрема, повністю відсутні проблеми з продуктивністю, що на 100% краще порівняно з попередніми перевірками. Жодна з проблем не була автоматично попереджена, проте це може бути пов’язано з характером змін або конфігурацією аналізаторів. Аналіз проводився для з використанням аналізаторів JavaScript для основної частини коду та Shell для інфраструктурних файлів.

Загалом результати свідчать про досить високу якість коду в проєкті. При подальшому розвитку проєкту основна увага має бути зосереджена на

усуненні антипатернів, які потенційно можуть ускладнити підтримку або негативно вплинути на масштабованість застосунку.

4.2 Опис процесів тестування

Тестування виконується згідно документу «Програма та методика тестування».

Було виконане мануальне тестування програмного забезпечення, опис відповідних тестів наведено у таблицях 4.1 – 4.12.

Таблиця 4.1 – Тест 1.1 Реєстрація користувача

Початковий стан системи	Користувач знаходиться на сторінці реєстрації.
Вхідні дані	Ім’я, прізвище, електронна пошта, пароль, підтвердження паролю.
Опис проведення тесту	Введення вхідних даних у відповідні поля, ім’я та прізвище містять мінімум по 3 символи, електронна пошта має патерн @email.com, пароль і підтвердження паролю повністю співпадають та містять щонайменше по 6 символів. Натискання кнопки підтвердження.
Очікуваний результат	Створено обліковий запис користувача, користувача перенаправлено у внутрішню систему, показано повідомлення-привітання. Заблоковано вхід на сторінку реєстрації і авторизації.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.2 – Тест 1.2 Авторизація користувача

Початковий стан системи	Користувач знаходиться на сторінці авторизації.
Вхідні дані	електронна пошта, пароль.

Продовження таблиці 4.2.

Опис проведення тесту	Введення пошти і паролю, які попередньо використовувались при реєстрації, у відповідні поля, натискаскання кнопки підтвердження авторизації.
Очікуваний результат	Авторизація проходить успішно, користувача перенаправлено у внутрішню систему, показано повідомлення-привітання. Заблоковано вхід на сторінку реєстрації і авторизації.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.3 – Тест 1.3 Вихід з облікового запису

Початковий стан системи	Користувач знаходиться у внутрішній системі, користувач зареєструвався та є авторизованим.
Вхідні дані	-
Опис проведення тесту	Натискання кнопки виходу з системи у панелі управління, після якого відкривається діалогове вікно з підтвердженням виходу. Натискання кнопки підтвердження виходу.
Очікуваний результат	Вихід проходить успішно, користувача перенаправлено на посадкову сторінку. Розблоковано вхід на сторінку реєстрації і авторизації.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.4 – Тест 1.4 Створення заявки на організацію

Початковий стан системи	Користувач знаходиться на сторінці створення заявки на організацію у внутрішній системі, користувач зареєструвався та є авторизованим.

Продовження таблиці 4.4.

Вхідні дані	Логотип організації у форматі зображення, назва організації, опис організації, посилання на сторінку організації, контактні дані організації, коментар до заявки, файли додатків.
Опис проведення тесту	Додання логотипу організації і додатків шляхом вибору файлів локального пристрою, введення усіх текстових даних у відповідні поля, вибір типу організації (ФОП, ТОВ та інші) у полі вводу з автозаповненням. Натискання кнопки підтвердження.
Очікуваний результат	Перенесення на сторінку заявок, відображення нової заявки на ній, повідомлення про успішне створення заявки.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.5 – Тест 1.5 Створення заявки на організацію

Початковий стан системи	Користувач знаходиться на сторінці створення заявки на організацію у внутрішній системі, користувач зареєструвався та є авторизованим.
Вхідні дані	Логотип організації у форматі зображення, назва організації, опис організації, посилання на сторінку організації, контактні дані організації, коментар до заявки, файли додатків.
Опис проведення тесту	Додання логотипу організації і додатків шляхом вибору файлів локального пристрою, введення усіх текстових даних у відповідні поля, вибір типу організації (ФОП, ТОВ та інші) у полі вводу з автозаповненням. Натискання кнопки підтвердження.

Продовження таблиці 4.5.

Очікуваний результат	Перенесення на сторінку заявок, відображення нової заявки на ній, повідомлення про успішне створення заявки.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.6 – Тест 1.6 Взяття заявки користувача в обробку

Початковий стан системи	Адмін знаходиться на сторінці доступних до обробки заявок на організацію у внутрішній системі, адмін авторизований.
Вхідні дані	-
Опис проведення тесту	Натискання на кнопку взяття в обробку на картці обраної заявки користувача.
Очікуваний результат	Заявку схвалено, статус заявки у БД змінено, користувач, що залишив заявку, отримує сповіщення про оновлення статусу заявки. У користувача і адміна, що взяв заявку в обробку, розблокується чат для спілкування.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.7 – Тест 1.7 Схвалення заявки користувача

Початковий стан системи	Адмін знаходиться на сторінці заявки. Адмін попередньо взяв заявку в обробку.
Вхідні дані	Коментар до схвалення

Продовження таблиці 4.7.

Опис проведення тесту	Натискання на кнопку схвалення на сторінці взятої в обробку заявки. Після відкриття модального вікна, введення коментаря до схвалення у відповідне поле вводу, натискання кнопки підтвердження.
Очікуваний результат	Перенесення на сторінку досту, відображення повідомлення про успішне схвалення. Користувач бачить сповіщення про успішне схвалення і може переглянути схвальний коментар. Створено нову організацію в системі.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.8 – Тест 1.8 Створення статичної черги

Початковий стан системи	Користувач знаходиться на сторінці створення черги. Користувач зареєструвався та авторизований, має хоча б одну організацію.
Вхідні дані	Початок і кінець робочого дня, початок і кінець перерви у робочий день, час обслуговування одного клієнта, часовий проміжок між клієнтами, назва та опис послуги, вартість послуги, зразки необхідних до заповнення клієнтами форм для надання послуги, файлові додатки.

Продовження таблиці 4.8.

Опис проведення тесту	Обирання організації у якій буде створено чергу з переліку організацій, у яких знаходиться користувач. Вибір днів обслуговування, наприклад: Пн, Вт. Завантаження додатків і зразків форм, шляхом вибору файлів локального пристрою. Вибір виконавця черги з переліку співробітників організації. Заповнення текстових полів вхідними даними. Натискання кнопки підтвердження.
Очікуваний результат	В системі створено статичну чергу. Перенесення користувача на сторінку черги, відображення повідомлення про успішне створення черги.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.9 – Тест 1.9 Створення живої черги

Початковий стан системи	Користувач знаходиться на сторінці створення живої черги. Користувач зареєструвався та авторизований, має хоча б одну організацію.
Вхідні дані	Початок і кінець робочого дня, назва та опис послуги, вартість послуги.
Опис проведення тесту	Вибір виконавця черги з переліку співробітників організації. Заповнення текстових полів вхідними даними. Натискання кнопки підтвердження.
Очікуваний результат	В системі створено живу чергу. Перенесення користувача на сторінку живої черги, відображення повідомлення про успішне створення живої черги.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.10 – Тест 1.10 Реєстрація користувача у статичній черзі

Початковий стан системи	Користувач знаходиться на сторінці пошуку черг. Користувач зареєструвався та авторизований. Організація надала ідентифікатор черги користувачу поза платформою.
Вхідні дані	Ідентифікатор черги
Опис проведення тесту	Введення ідентифікатора черги у відповідне поле вводу на сторінці. Натиснення кнопки підтвердження. Відбувається перенесення користувача на відповідну сторінку черги. Вибір дати та вибір часу з доступних тайм-слотів. Натиснення кнопки підтвердження.
Очікуваний результат	Створення нового запису у чергу, що відображається у виконавця і клієнта організації. Блокування обраного користувачем тайм-слоту. Відображення повідомлення про успішну реєстрацію у черзі.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.11 – Тест 1.11 Реєстрація клієнта у живій черзі

Початковий стан системи	Користувач знаходиться на сторінці черги, де є виконавцем. Користувач зареєструвався та авторизований.
Вхідні дані	Електронна пошта клієнта
Опис проведення тесту	Натиснення кнопки додавання клієнта до черги. Відкривається модальне вікно. Введення пошти клієнта в відповідне поле вводу, натиснення кнопки підтвердження.

Продовження таблиці 4.11.

Очікуваний результат	Створення нового запису у живу чергу, клієнта додано на останню позицію в черзі, що відображається у виконавця і клієнта організації. Відображення повідомлення про успішну реєстрацію клієнта у черзі.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.12 – Тест 1.12 Запрошення в організацію

Початковий стан системи	Користувач знаходиться на сторінці своїх організацій. Користувач має хоча б одну організацію, де він є власником або абміністратором. Користувач зареєструвався та авторизований.
Вхідні дані	Електронна пошта запрошуваного
Опис проведення тесту	Натиснення кнопки запрошення в організацію. Відкривається модальне вікно. Введення пошти запрошуваного в відповідне поле вводу, натиснення кнопки прийняття.
Очікуваний результат	Створення нового запрошення в організцію, що відображається у запрошуваного.
Фактичний результат	Збігається з очікуваним.

4.3 Опис контрольного прикладу

При першому заході на платформу, користувач ще не має облікового запису і, відповідно, бачить посадкову сторінку, де може коротко прочитати про платформу.

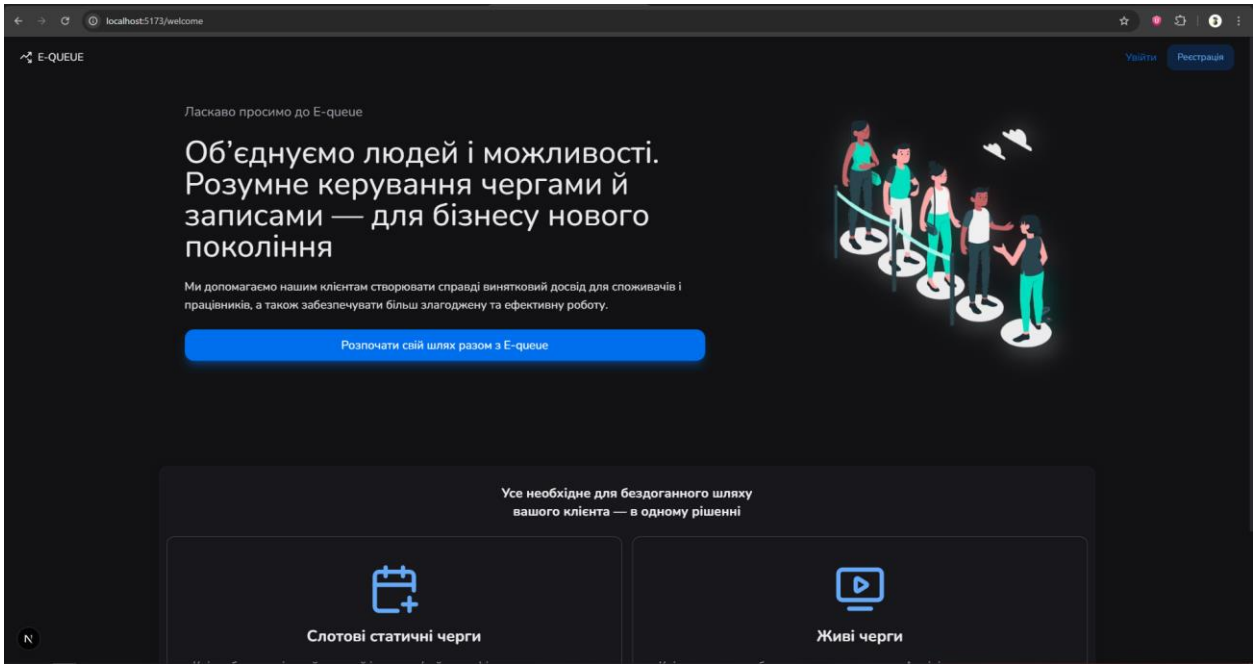


Рисунок 4.2 – Посадкова сторінка

Для подальшої взаємодії необхідно зареєструватись, для цього є кнопка реєстрації у навігаційній панелі. При натисканні на неї користувача перекидує на сторінку реєстрації. В формі на сторінці необхідно ввести: ім'я, прізвище, електронну пошту, пароль та підтвердження паролю. Ім'я та прізвище містять мають містити мінімум по 3 символи, електронна пошта має патерн @email.com, пароль і підтвердження паролю повністю співпадають та містять по 6 символів. Підтверджуємо реєстрацію.

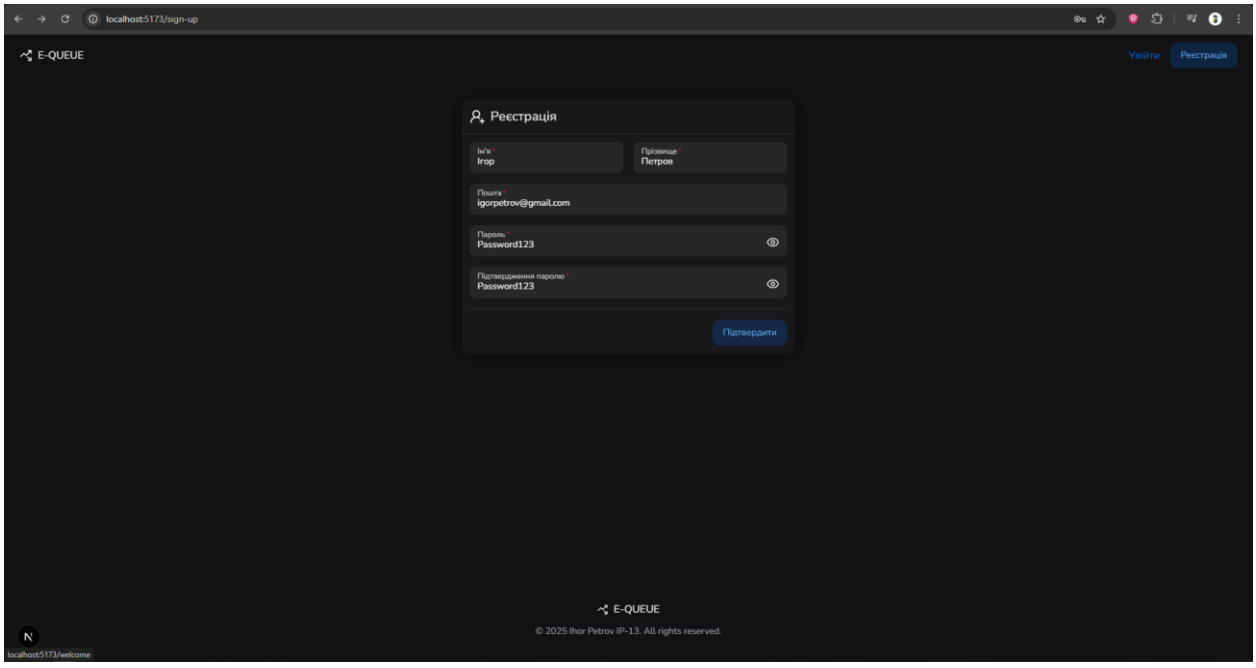


Рисунок 4.3 – Реєстраційна форма

Після входу на платформу користувач може створити організацію. Для необхідно зайти на сторінку створення організації. На самій сторінці знаходиться велика форма, де можена вказати дані заявки на організацію. Частина цих даних далі буде використана для створення самої організації, зокрема назва, опис, логотип.

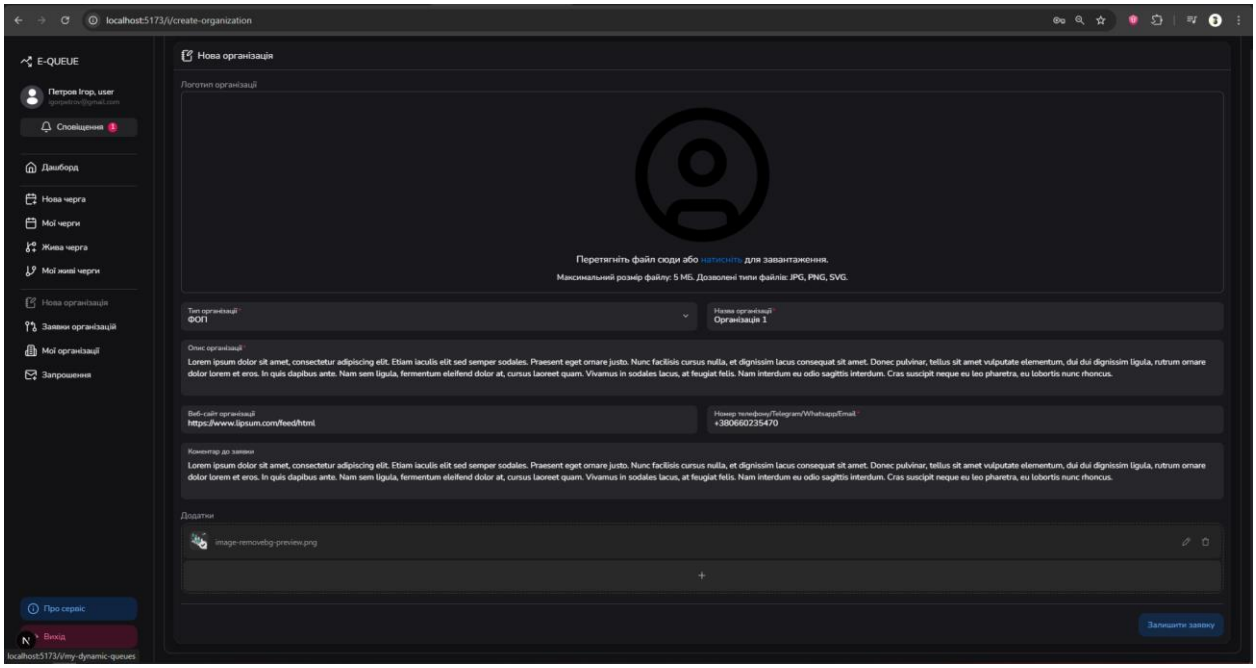


Рисунок 4.4 – Форма створення заявки на організацію

Після підтвердження форми, користувача перенаправляє на сторінку його поточних заявок на організацію. Тут він може побачити її поточний статус, дату створення.

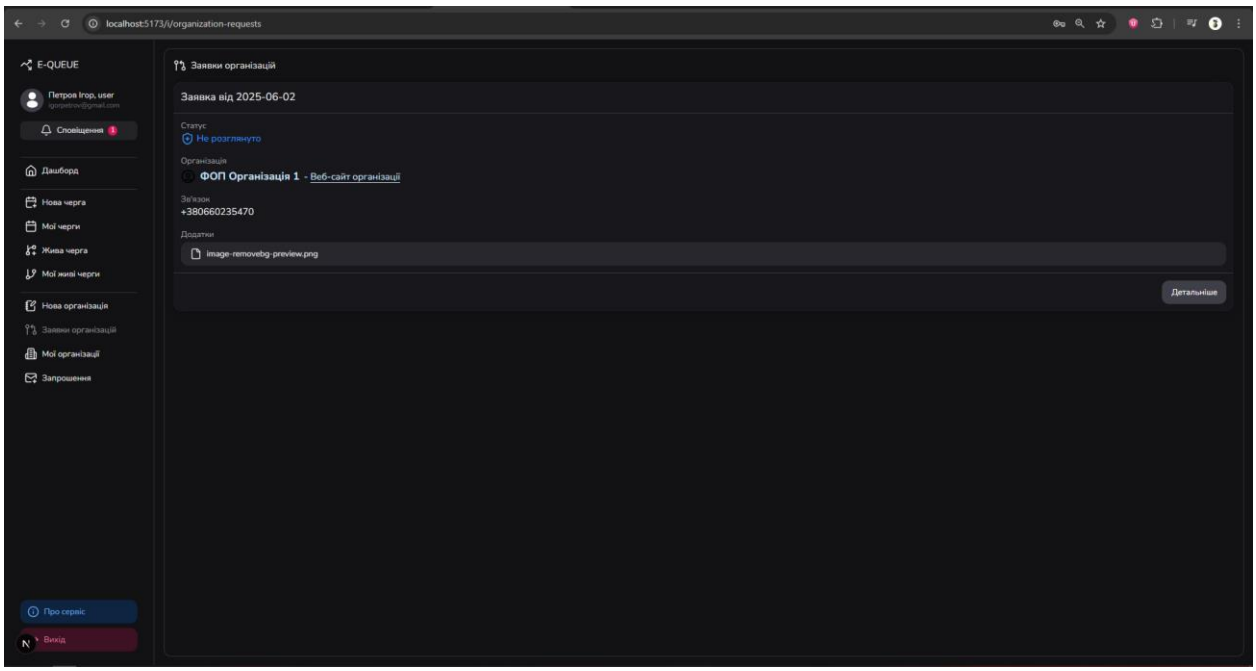


Рисунок 4.5 – Сторінка заявок користувача

При натисканні кнопки «Детальніше», можна відкрити деталі заявки, де ще раз переглянути заповнені при створенні дані.

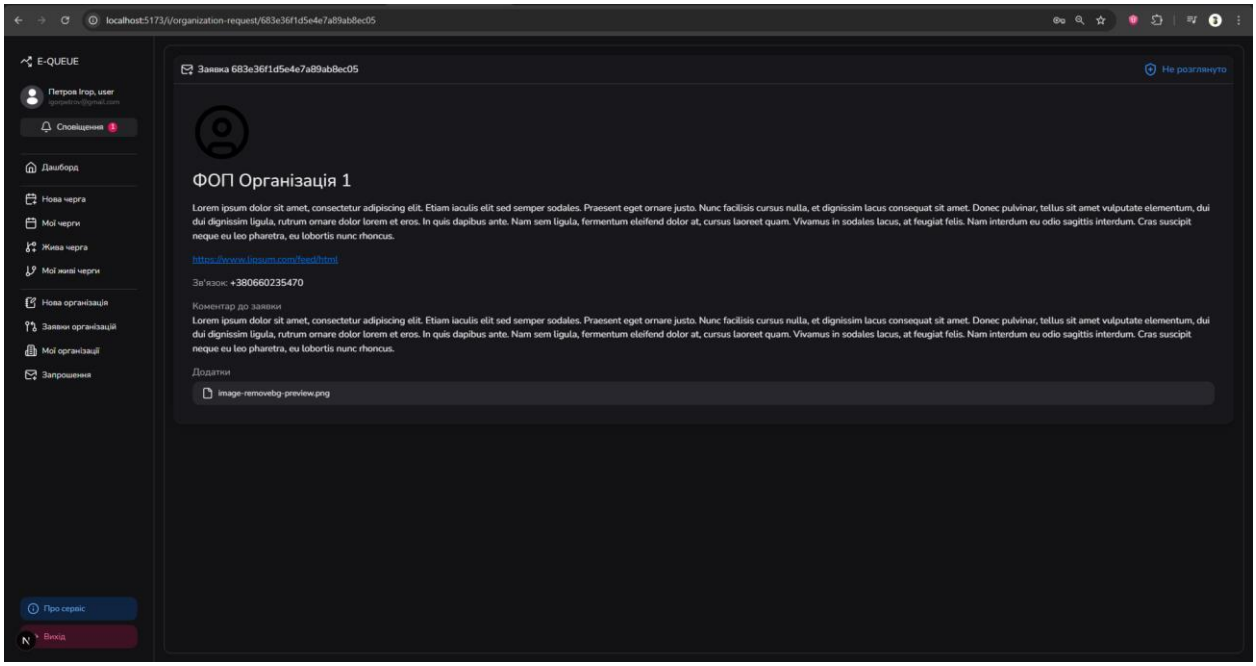


Рисунок 4.6 – Детальна сторінка заявки користувача

Для продовження необхідні дії від адміністрації застосунку. Адміністратор має бути заздалегіть зареєстрований і доданий в базу даних. Адміністратор також має пройти процес авторизації, однак не має можливості зареєструватись напряму через платформу.

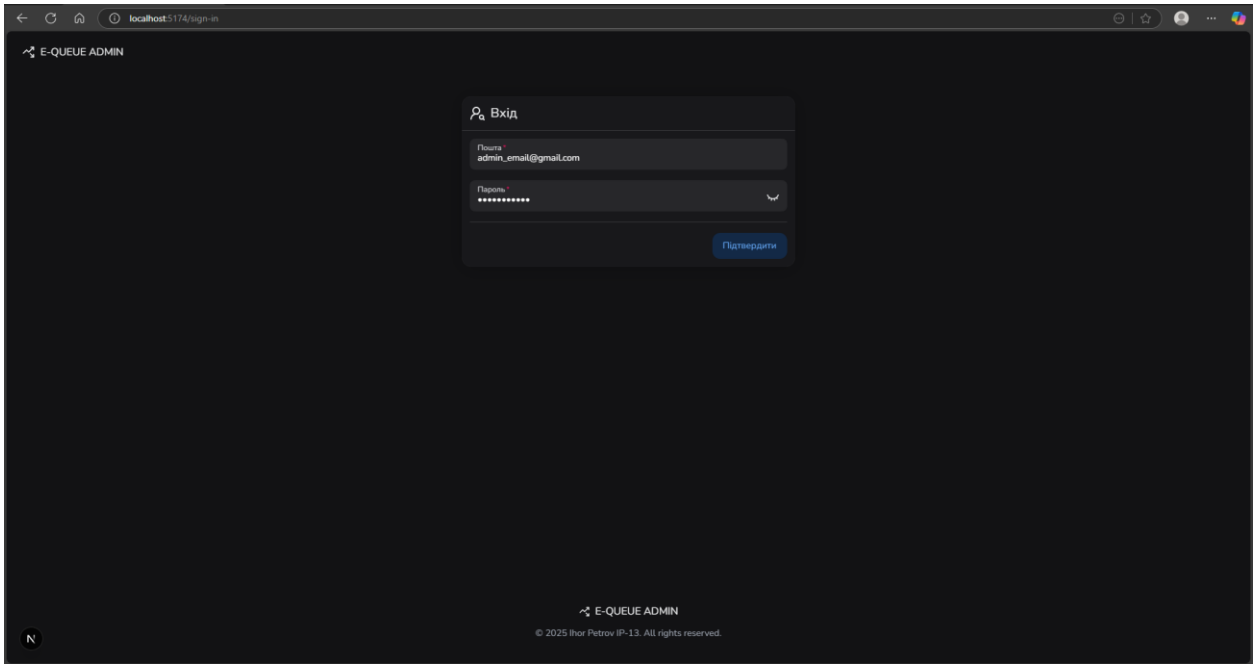


Рисунок 4.7 – Авторизація адміністратора

Після проходження авторизації адміністратор може побачити заявку користувача на сторінці доступних для обробки заявок. У адміністратора вже є можливість взяти заявку в обробку, після цього вона зникне зі сторінки доступних заявок і буде відображатись лише у адміністратора, що взяв заявку в обробку на сторінці заявок в обробці і користувача, що залишив заявку.

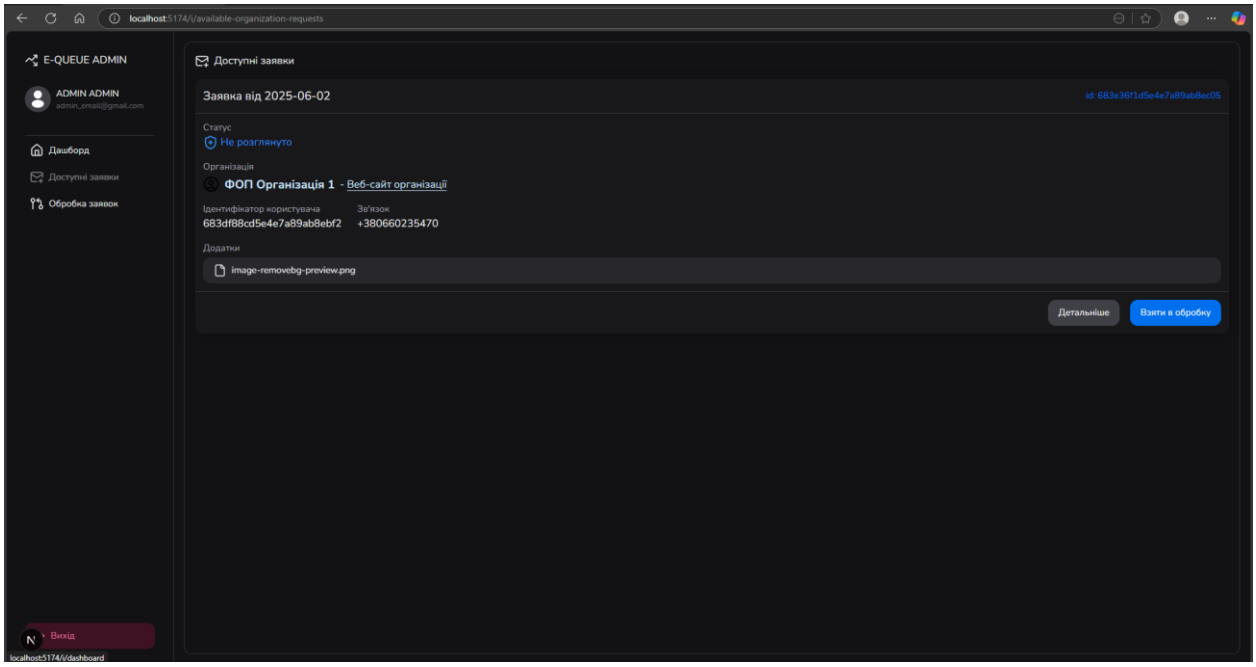


Рисунок 4.8 – Сторінка доступних заявок

Після взяття в обробку заявка змінила статус і адміністратор з заявником отримують додаткові можливості. Розблокується чат для взаємодії, адміністратор отримує можливість схвалити або відхилити заявку.

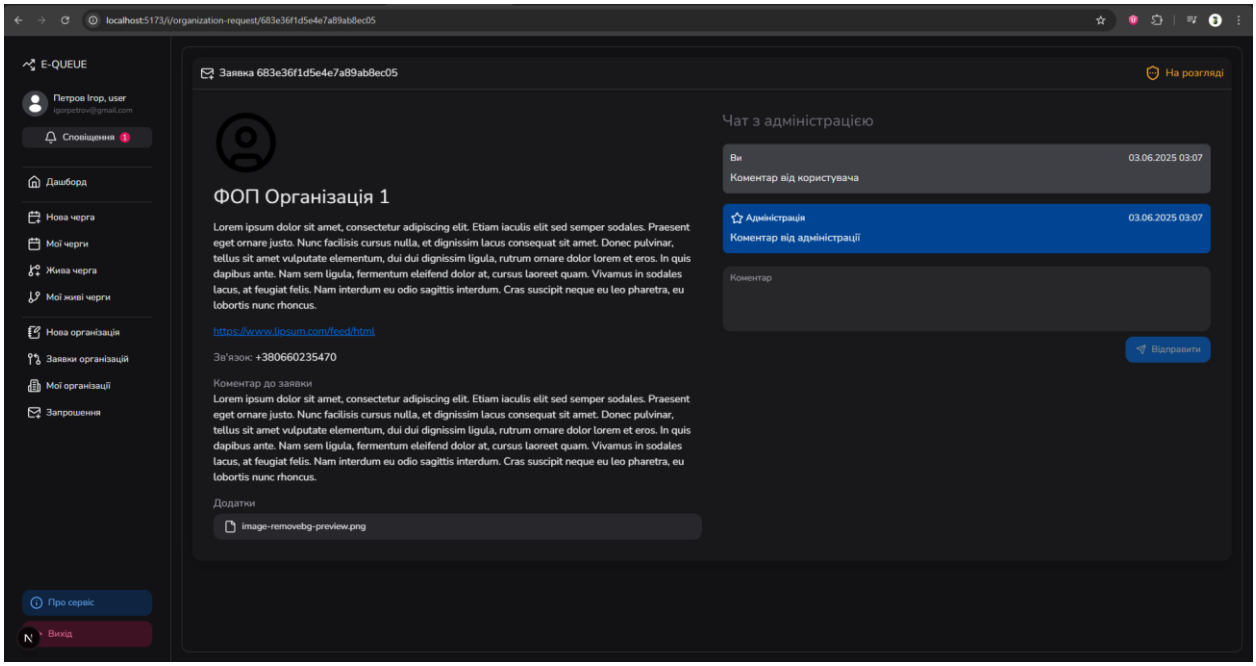


Рисунок 4.9 – Детальна сторінка заявки в обробці у користувача

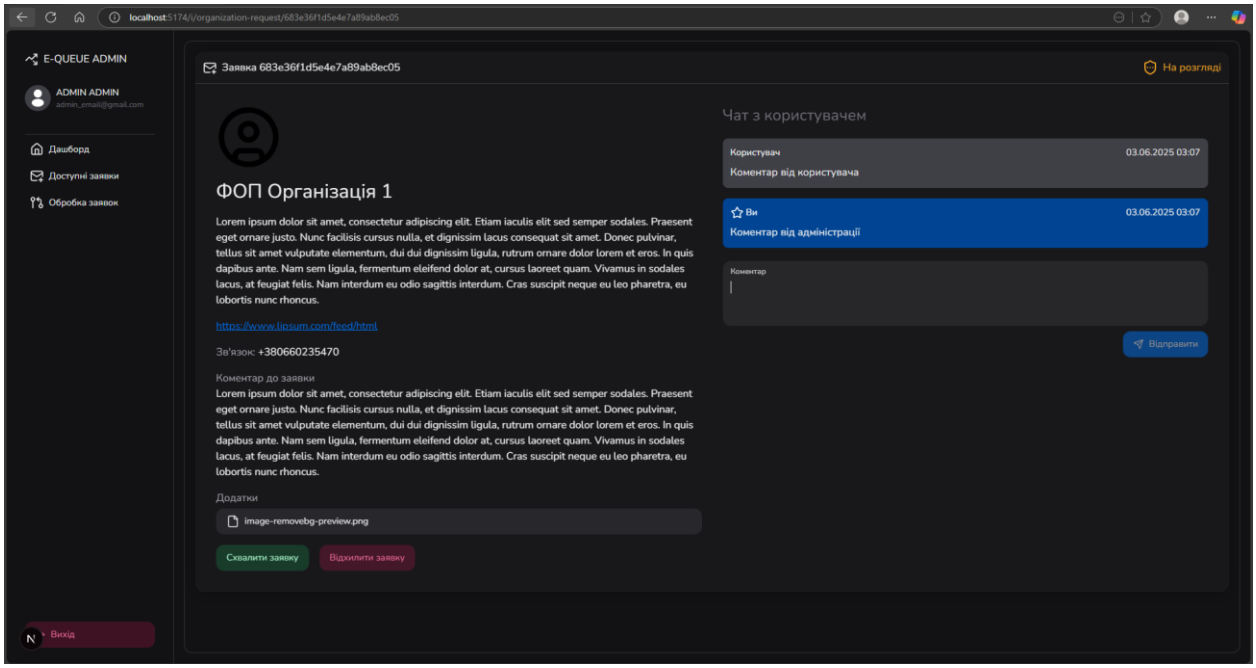


Рисунок 4.10 – Детальна сторінка заявки в обробці у адміністратора

Коли адміністратор отримав всі необхідні деталі та поспілкувався з користувачем, він може схвалити заявку. Для цього необхідно натиснути кнопку схвалення заявки та залишити коментар про схвалення.

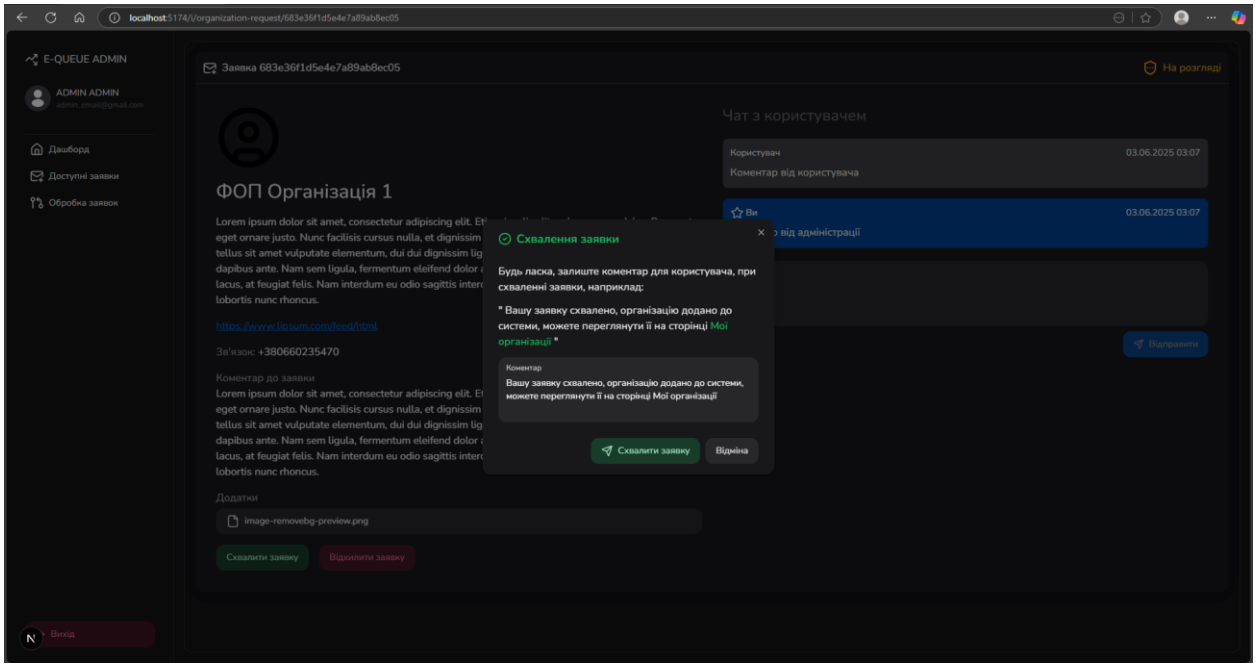


Рисунок 4.11 – Схвалення заявки адміністратором

Після схвалення заявки система створює нову організацію за даними, які користувач ввів при створенні заявки. Статус заявки змінюється, чат з

адміністрацією блокується, користувач може побачити коментар до схвалення.

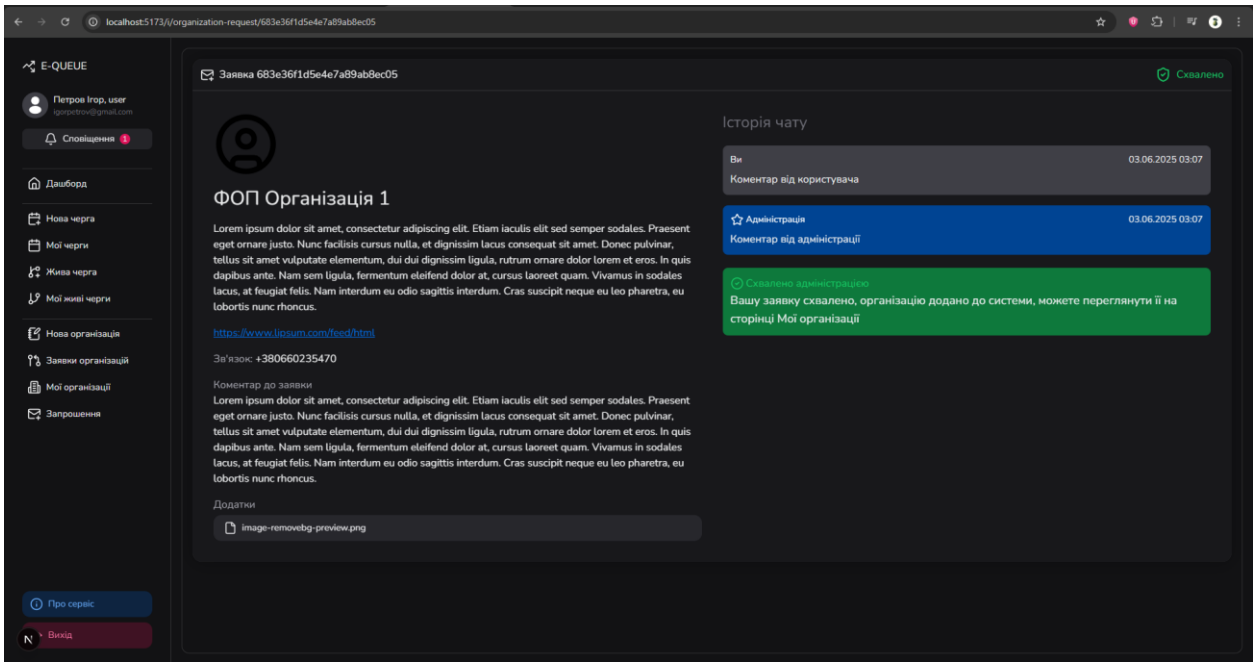


Рисунок 4.12 – Схвалена адміністратором заявка

Тепер користувач може побачити свою новостворену організацію на сторінці організацій з короткою інформацією про організацію, поточну кількість створених черг і співробітників.

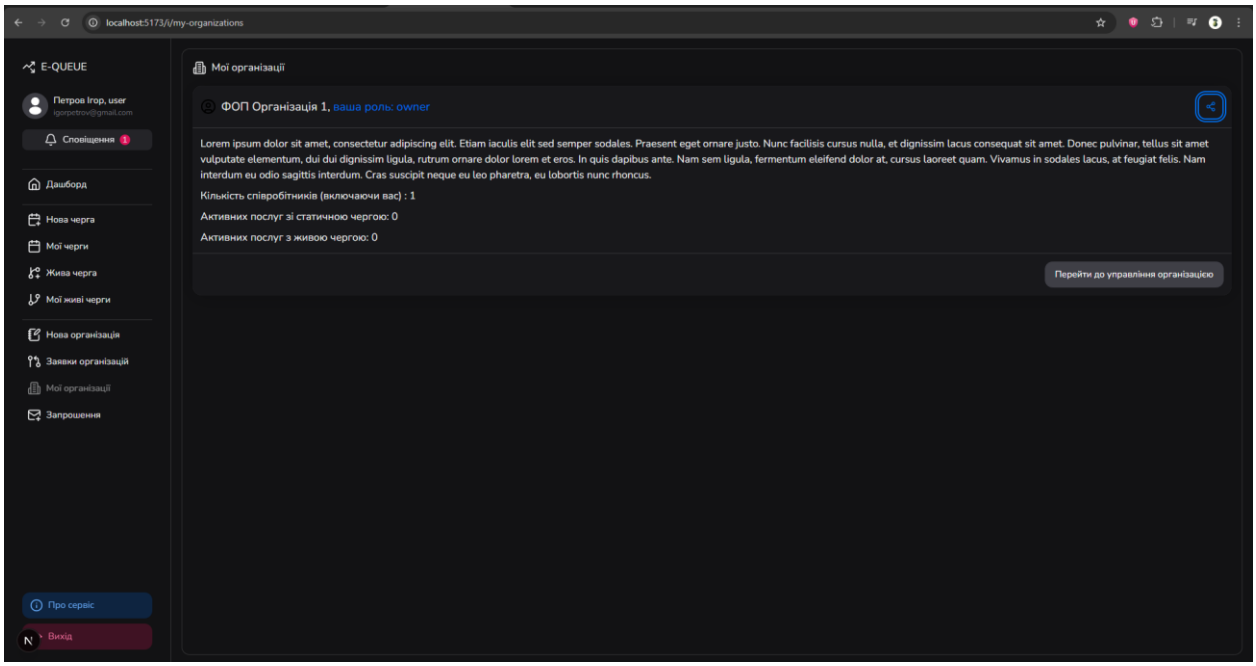


Рисунок 4.13 – Сторінка організацій користувача

Тепер користувач має можливість створити першу статичну чергу для обслуговування клієнтів. Для цього користувач має зайти на сторінку

створення черг. Перш за все користувач обирає організацію, у якій буде створено чергу, також користувач має вибрати дні, у які буде проходити послуга та заповнити усі поля, що відмічені червоною зірочкою. За потреби можна додати файли в додатки, зразки необхідних до заповнення перед прийомом форм та зразки вже заповнених форм для більшого ознайомлення кінцевого клієнта з послугою.

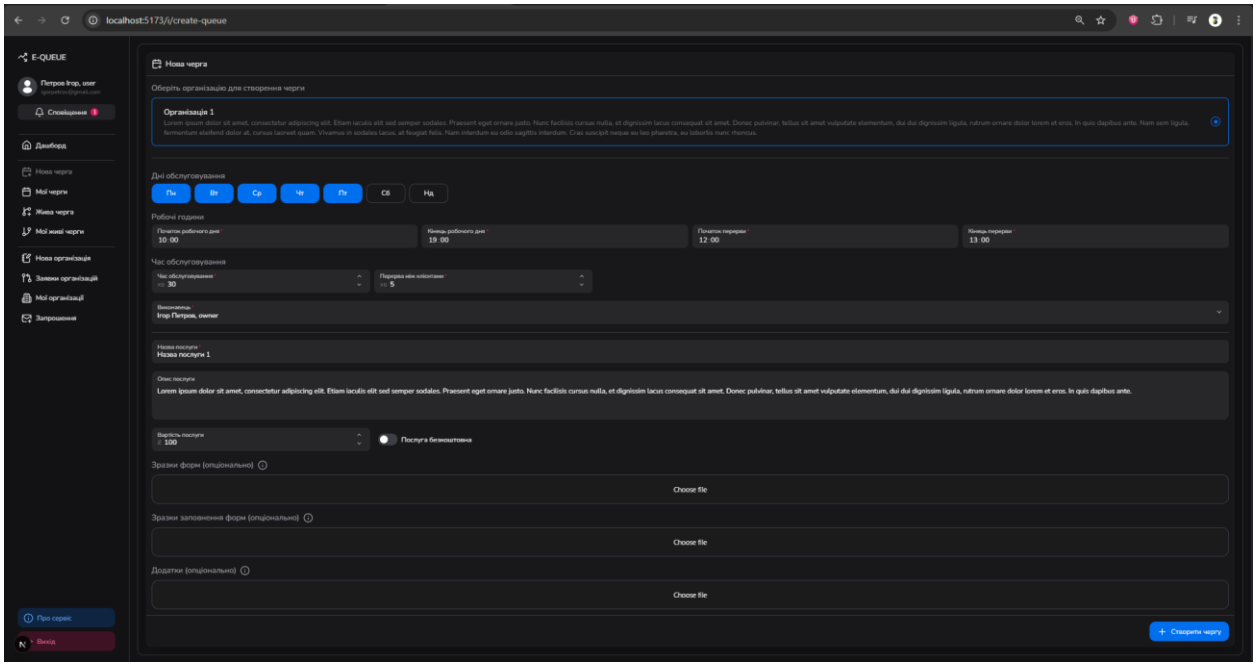


Рисунок 4.14 – Сторінка створення нової статичної черги

Якщо користувач вказав себе у якості виконавця, то новостворена черга збуде відображатись на сторінці черг користувача, де він є виконавцем, у вигляді картки з коротким описом.

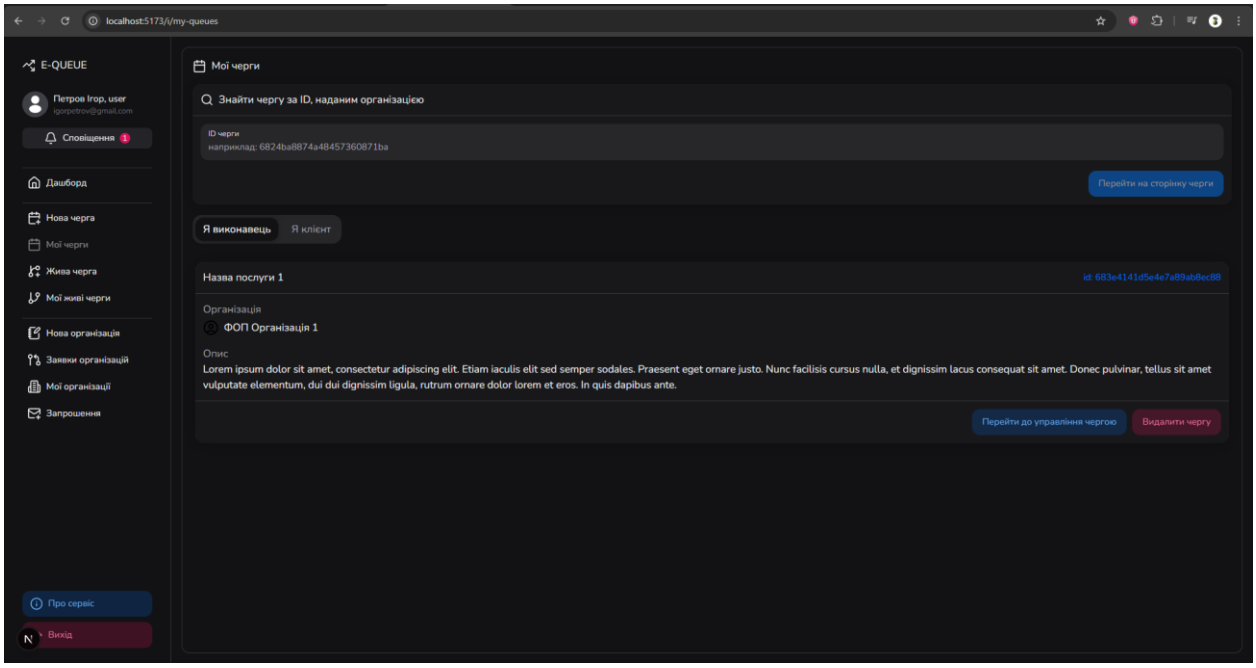


Рисунок 4.15 – Сторінка статичних черг користувача

Для залучення клієнтів організація може поділитись ідентифікатором черги зручним для них способом поза платформою. Тепер інший користувач, клієнт організації може знайти чергу за ідентифікатором, наданим організацією. Для цього він має ввести ідентифікатор у пошук за ідентифікатором та натиснути кнопку підтвердження.

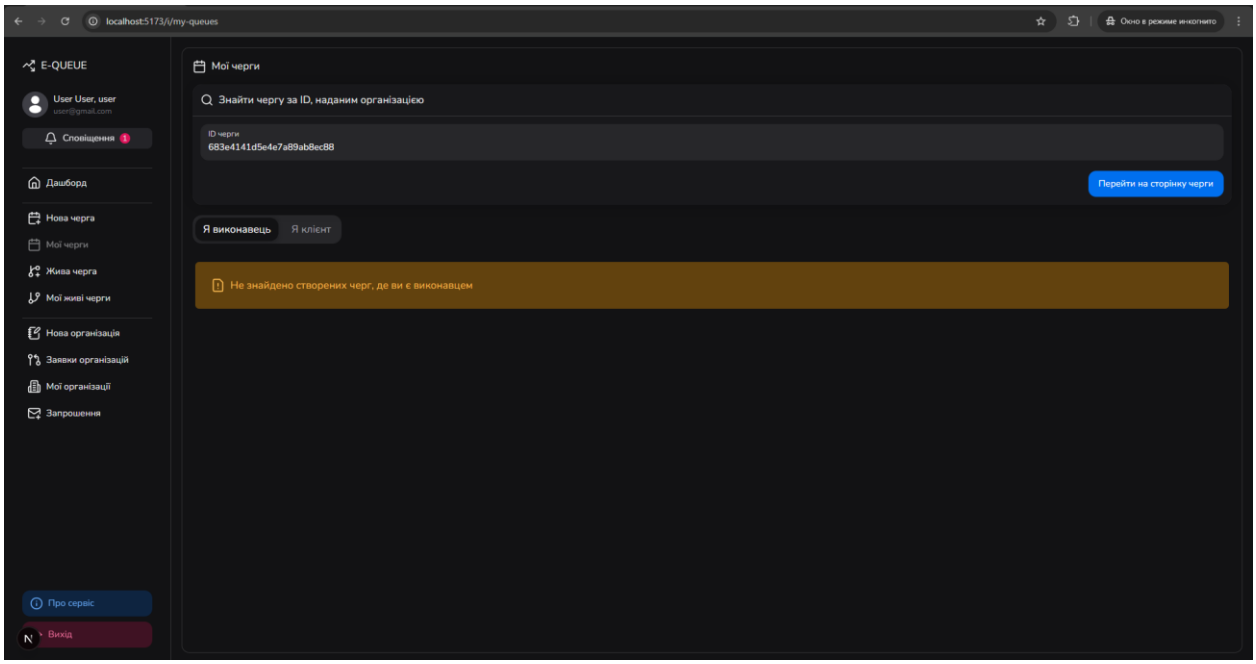


Рисунок 4.16 – Пошук черги за ідентифікатором

Якщо ідентифікатор коректний, то користувача перекине на сторінку черги, де він може обрати підходящі для нього дату для нього тайм-слот. Тайм-

слоти формуються на основі часу початку і кінця роботи, а також перерв, вказаних організацією при створенні черги.

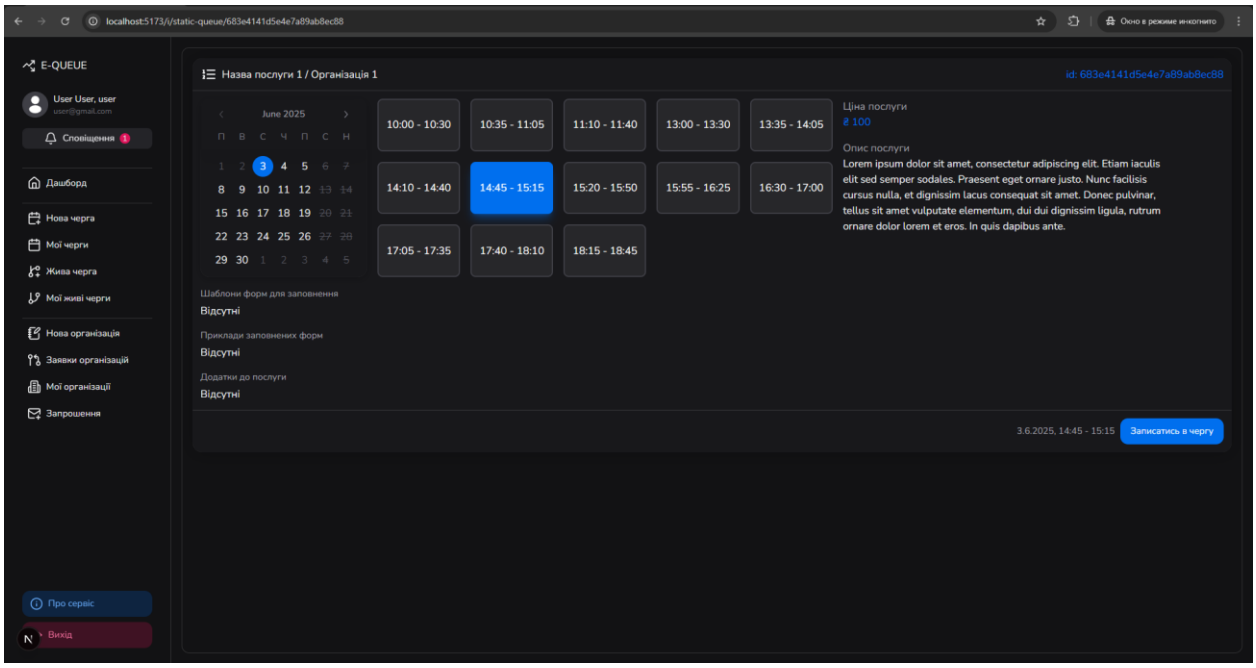


Рисунок 4.17 – Запис клієнта в чергу

Після реєстрації в черзі відповідний слот блокується на запис для інших клієнтів, а користувач бачить запис на сторінці черг, де він є клієнтом.

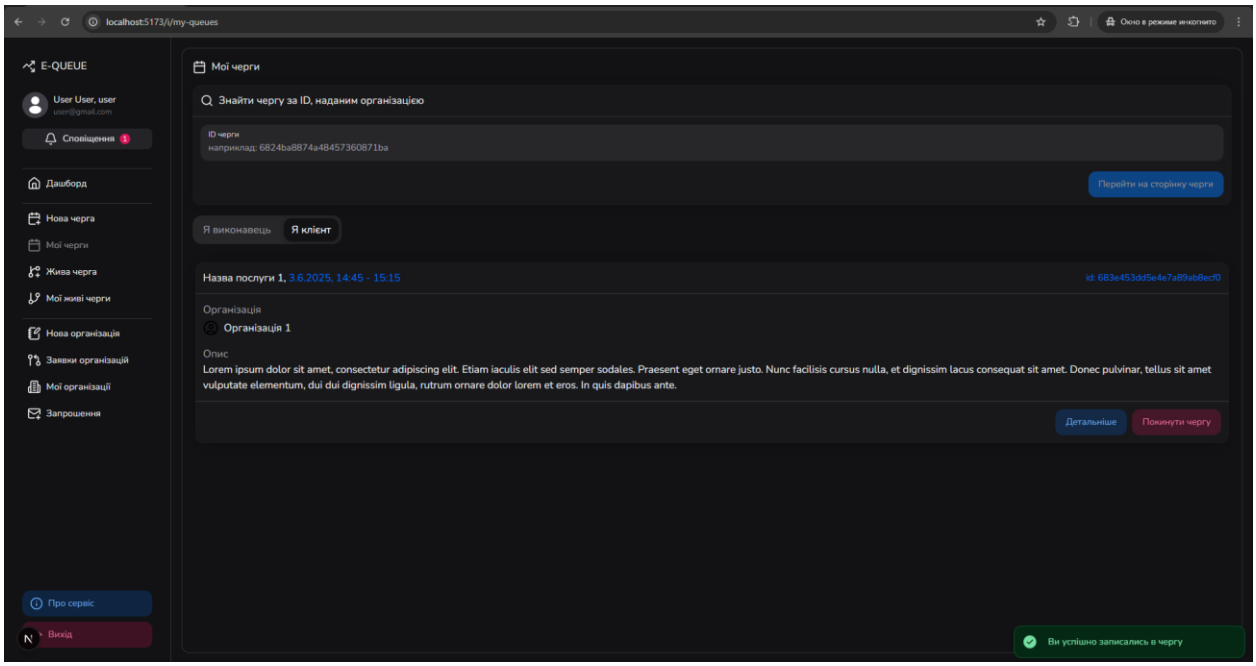


Рисунок 4.18 – Новий запис користувача в чергу

Виконавець також бачить новий запис в чергу на сторінці управління чергою.

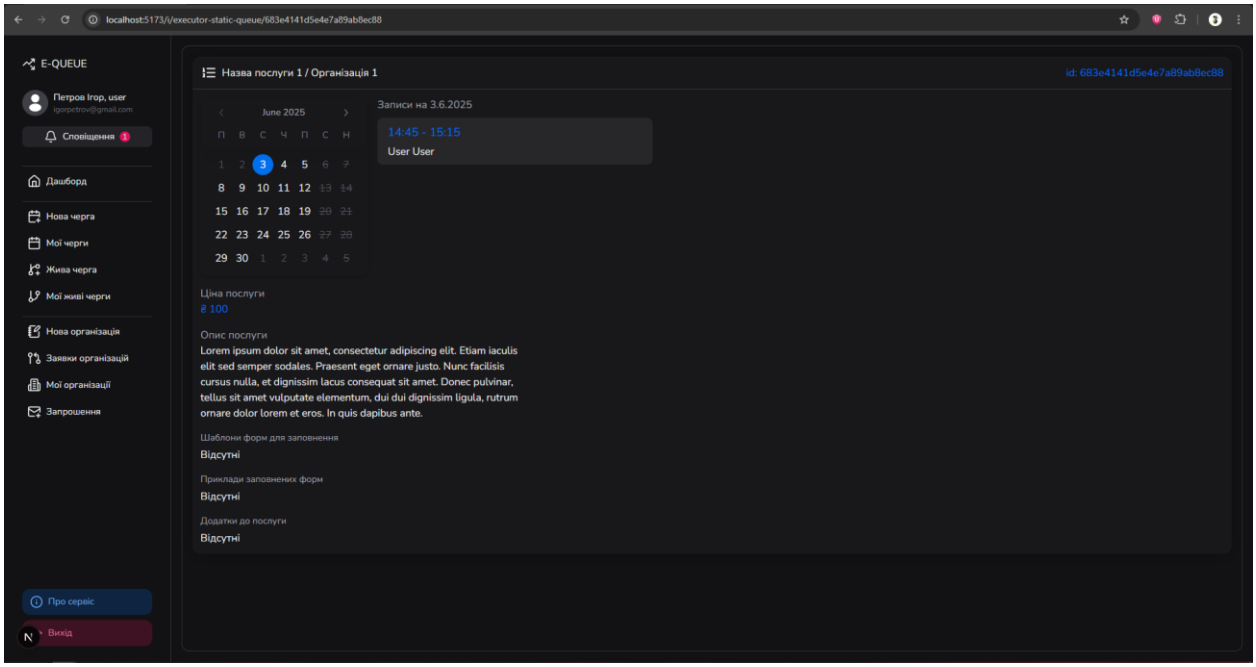


Рисунок 4.19 – Новий запис користувача в чергу на сторінці управління чергою

Окрім статичних черг з тайм-слотами користувачі в організаціях можуть також створювати менш залежні від часу живі черги, призначені для швидкого запису без прямої взаємодії клієнтів з платформою. Для цього потрібно зайти на сторінку створення живих черг. Сама форма схожа на створення статичної черги, окрім файлових додатків і днів обслуговування. Жива черга працює в рамках одного дня і клієнти не готуються до прийому заздалегіть.

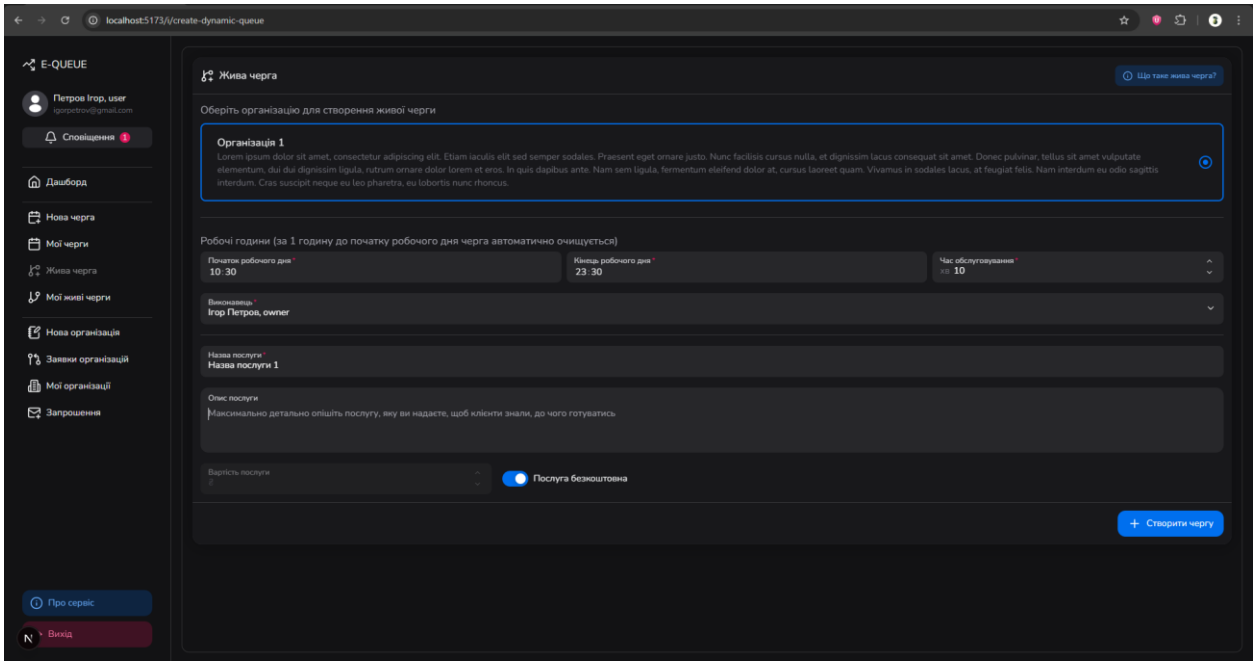


Рисунок 4.20 – Створення живої черги

Після створення живої черги, якщо користувач вказав себе у якості виконавця, він може побачити картку нової черги на сторінці живих черг, де він вказаний, як виконавець.

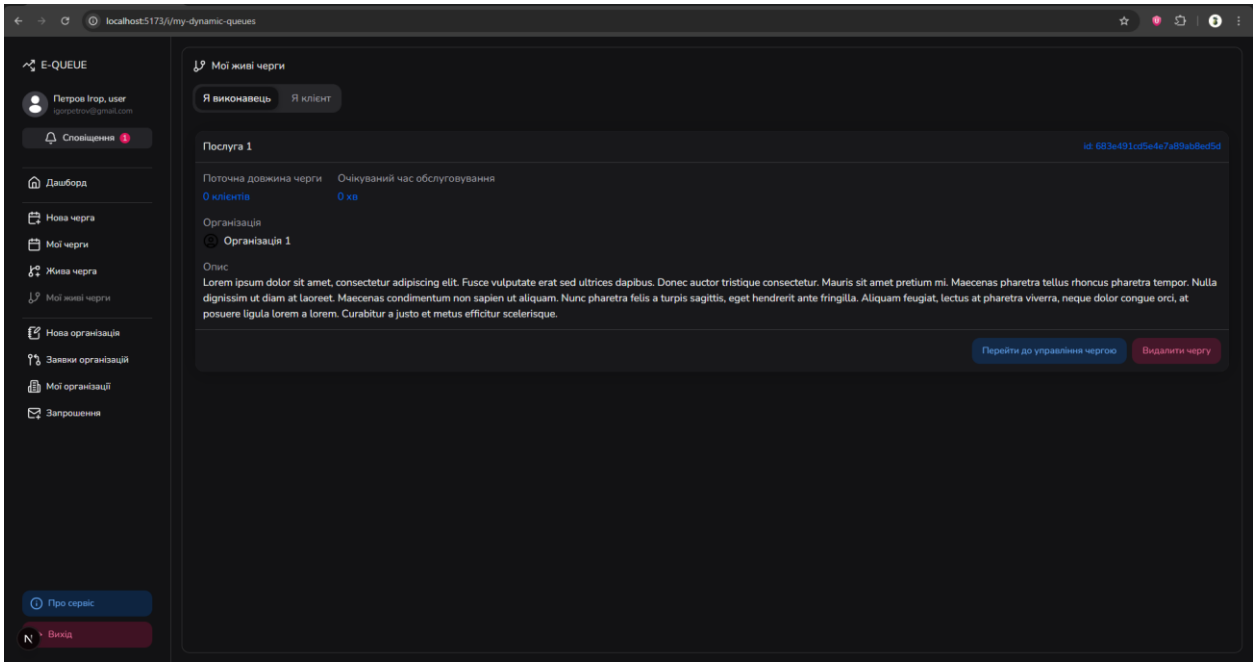


Рисунок 4.21 – Сторінка живих черг, де користувач вказаний, як виконавець

Для додавання клієнтів, користувач має перейти до управління чергою, натиснути кнопку «Додати клієнта» і ввести пошту клієнта у відповідному полі вводу у попапі та підтвердити реєстрацію клієнта. Клієнта буде додано на останню позицію в черзі і він буде відображений.

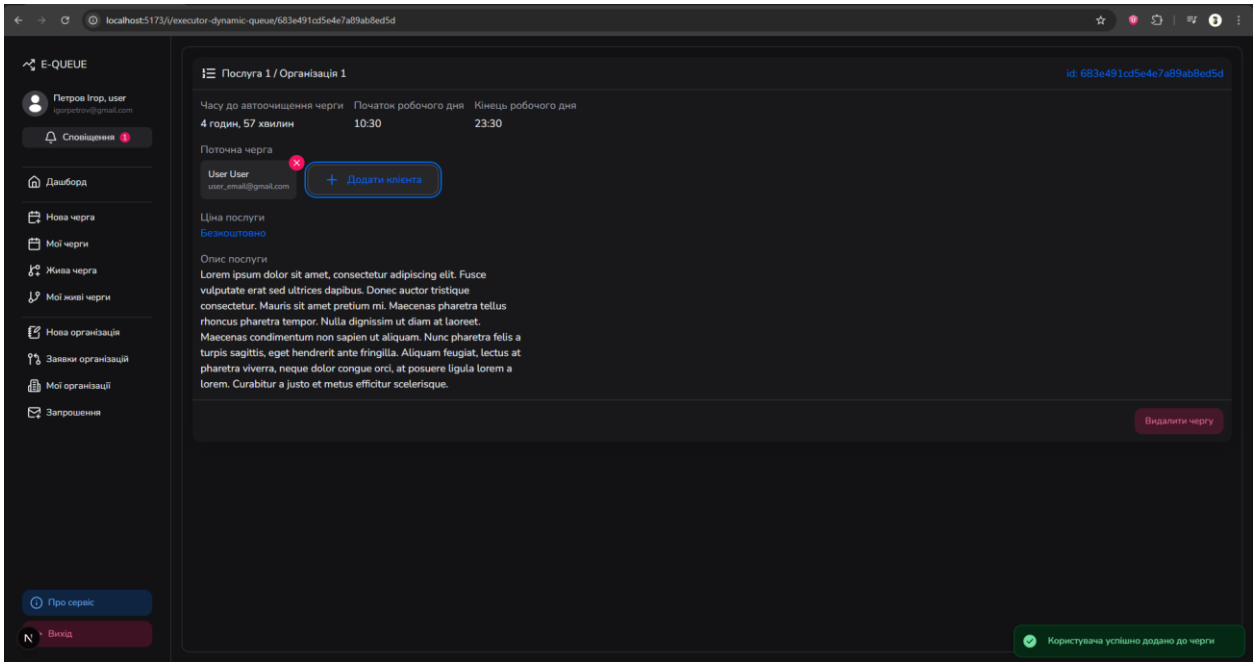


Рисунок 4.22 – Сторінка управління живою чергою

Сам клієнт також бачить цей запис у себе, у вигляді простої картки на сторінці живих черг, де він вказаний, як клієнт. На самій картці клієнт також бачить свою поточну позицію в черзі і короткий опис послуги.

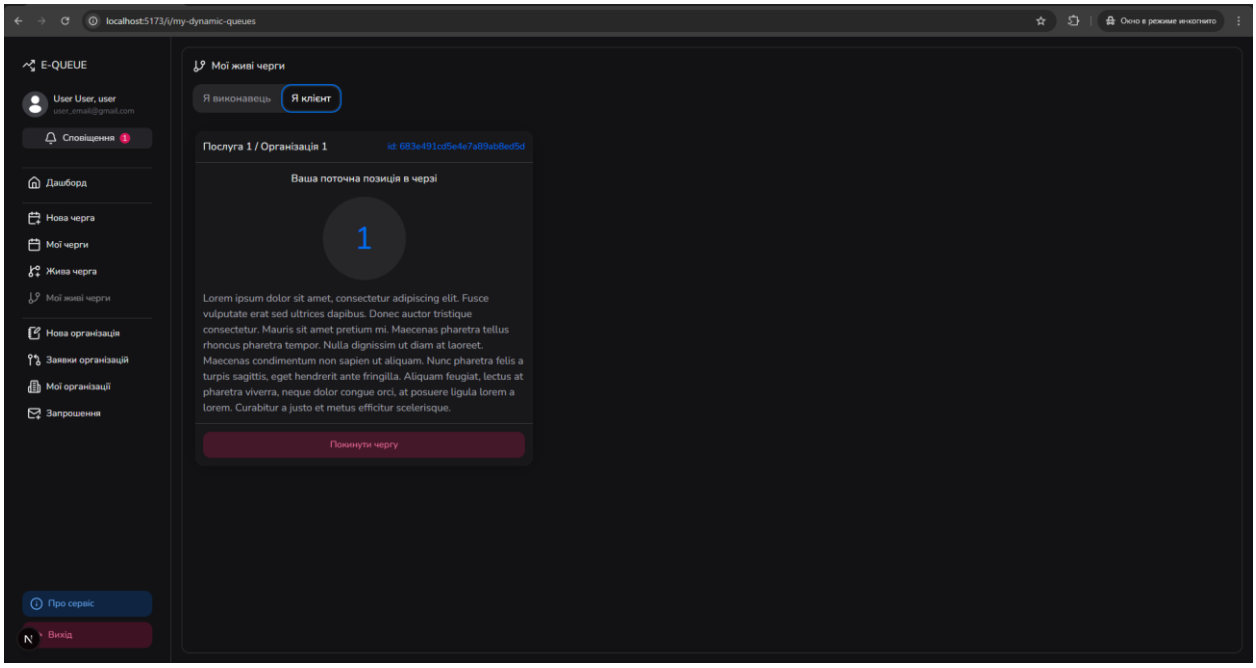


Рисунок 4.23 – Сторінка живих черг, де користувач є клієнтом

Висновки до розділу

Було проведено комплексний аналіз якості програмного забезпечення та здійснено тестування розробленої системи «E-queue». Зокрема, при вивченні якості коду за допомогою платформи DeepSource було виявлено 25 активних проблем, з яких 12 належать до категорії антипатернів. При цьому відсутність проблем продуктивності, стилю, безпеки та документації вказує на достатньо високий рівень підтримуваності та надійності проєкту в цілому . За результатами порівняння з попереднім аналізом рівень антипатернів скоротився на 20 %, що свідчить про поступове поліпшення якості коду.

Щодо процесу тестування, було виконано повний цикл мануального тестування відповідно до «Програми та методики тестування». Наведені у таблицях сценарії охоплюють ключові функціональні блоки системи: реєстрацію та авторизацію користувача, управління чергами, роботу з ролями та іншими сервісами. Усі фактичні результати тестів збіглися з очікуваними,

що підтверджує коректність реалізації основного функціоналу та його відповідність вимогам замовника . Це свідчить про стабільність системи в умовах описаних сценаріїв використання.

Загалом, проведена аналітика якості та результати тестування підтверджують, що система «E-queue» відповідає встановленим критеріям надійності та готова до наступних етапів — розгортання та експлуатації.

5 РОЗГОРТАННЯ ТА СУПРОВІД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Розгортання програмного забезпечення

Спершу необхідно створити кластер у хмарній базі даних MongoDB Atlas. Після реєстрації облікового запису на сайті Atlas потрібно перейти до створення нового кластера, обрати потрібний тип (може бути безкоштовний кластер для тестових цілей) і налаштувати доступ через мережеві правила. У розділі Network Access додаємо IP-адресу свого сервера чи встановлюємо доступ із будь-якої адреси, якщо це робиться у тестовому середовищі. Далі у розділі Database Access створюється користувач із правами читання та запису, після чого копіюється рядок підключення (URI) до кластера. Це значення буде потрібне під час налаштування серверної частини для встановлення зв’язку з базою. Панель управління Atlas з вже створеним кластером на рисунку 5.1.

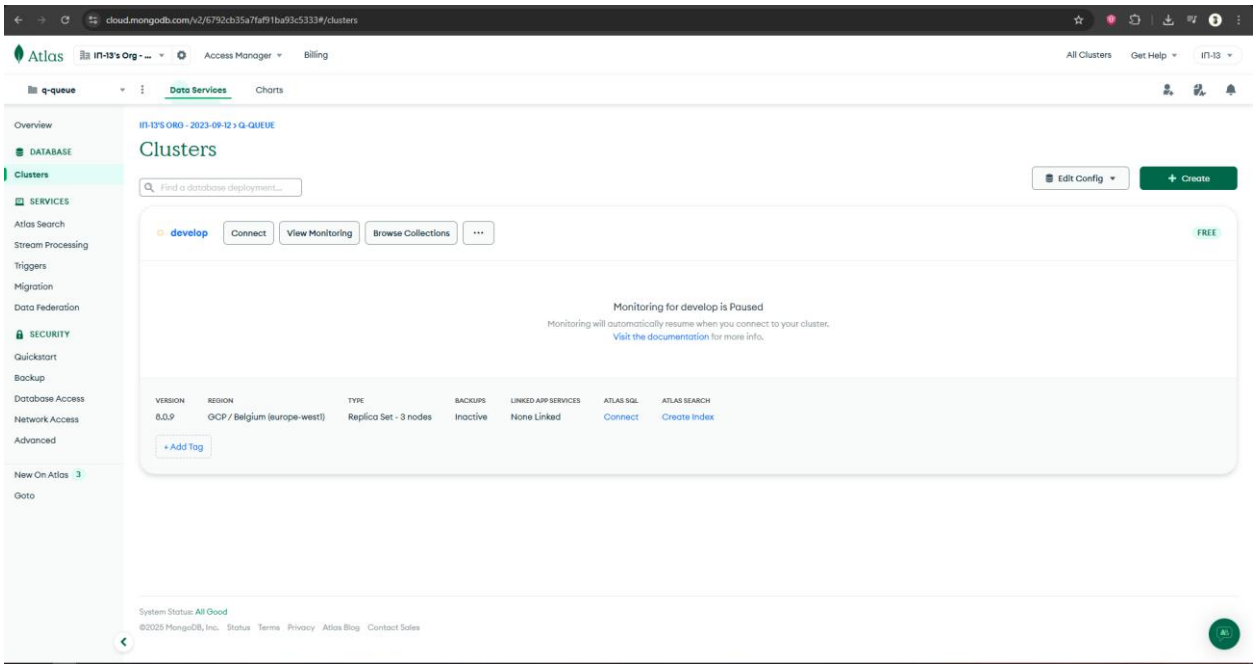


Рисунок 5.1 – Кластер MongoDB Atlas

Для збереження файлів застосунок вікористовує Uploadcare. Необхідно створити обліковий запис на їхньому офіційному сайті. Після реєстрації ви потрапляєте до панелі керування, де зліва або у верхньому меню доступні налаштування проєкту, серед яких розміщена інформація про публічний і приватний ключі (Public Key та Secret Key). Публічний ключ – це саме той

ідентифікатор, який вказується у клієнтській частині коду для завантаження файлів. Він не є секретом, тому його можна зберігати у фронтенд-застосунку, але бажано передавати через змінну середовища або конфігурацію хостингу, щоб легко змінювати його без прямого редагування коду.

Отримання публічного ключа відбувається досить просто – після входу до акаунта у розділі налаштування проекту потрібно знайти блок із написом «Public Key». У деяких випадках спочатку потрібно створити новий проєкт або додати новий ключ, натиснувши відповідну кнопку, а потім система згенерує пару ключів.

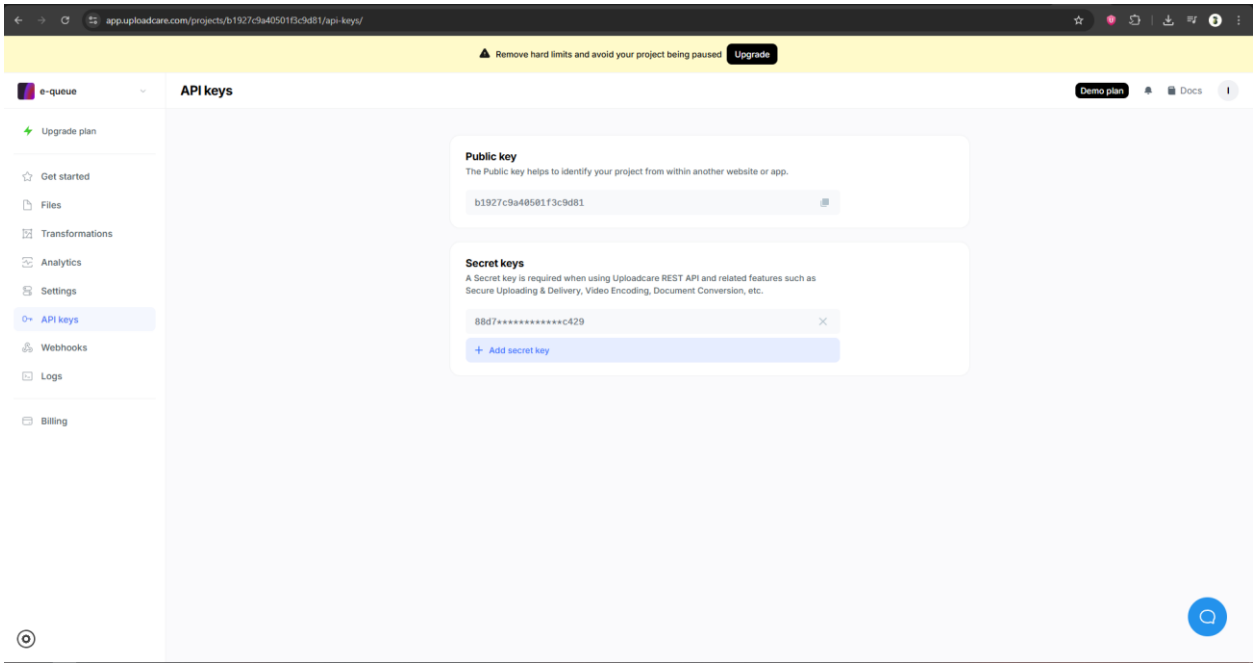


Рисунок 5.2 – Отримання публічного ключа Uploadcare

Далі необхідно створити файли оточення у відповідних проєктах монорепозиторію. Перелік необхідних змінних оточення та шлях до них на рисунках 5.3 та 5.4.

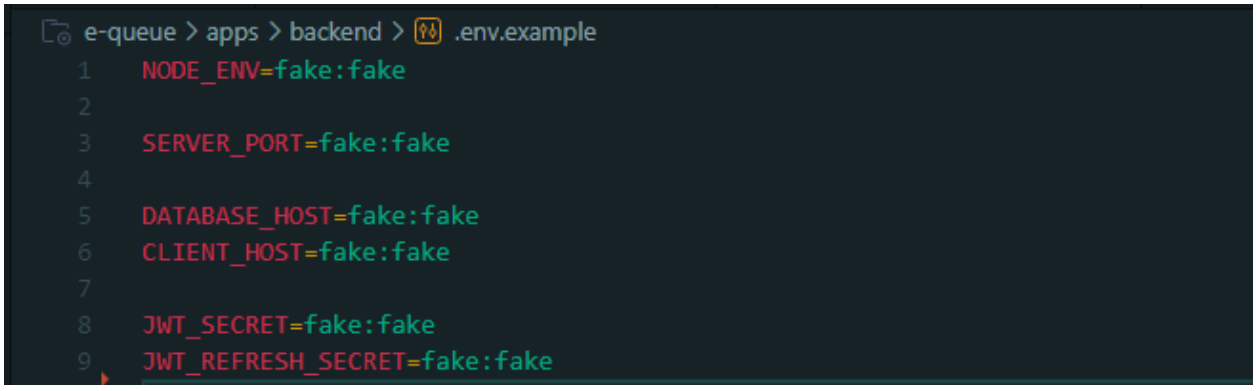
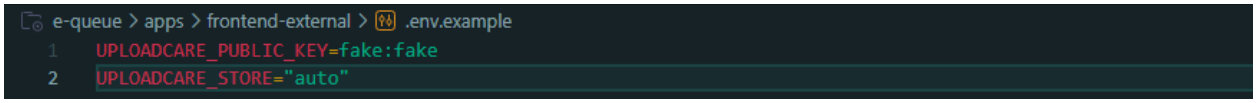


Рисунок 5.2 – Файл змінних оточення для серверної частини



Що стосується клієнтської частини, кожен застосунок реєструється окремо в хостингу Vercel. Після підключення репозиторія GitHubi вибору гілки, усі необхідні змінні середовища налаштовуються в розділі Environment Variables. Для обох клієнтів у налаштуваннях Vercel вказують команду збірки через Turborepo, а також зберігають стандартні вихідні папки Next.js, які Vercel зазвичай виявляє автоматично.

Для серверної частини на Nest.js обирається хостинг Render. Після входу в обліковий запис Render потрібно створити новий Web Service і підключити до нього Git-репозиторій, вибравши потрібну гілку. У розділі налаштувань Build Command задається запуск Turborepo лише для серверного проєкту (turbo run build --filter=backend), а Start Command вказується як запуск зкомпільованого файлу з папки dist (node apps/server/dist/main.js). Далі в розділі Environment можна вказати секрети, що вже були створені заздалегіть. Ці змінні забезпечують безпечне підключення до бази даних та коректну роботу сервера.

5.2 Супровід програмного забезпечення

Інструкція користувача наведена в окремому документі «Керівництво користувача».

Супровід програмного забезпечення починається з організації роботи над кодом у GitHub. Нові виправлення або функціональні доповнення реалізуються у відокремлених гілках, де розробник локально перевіряє зміни: запускає тести, вручну перевіряє основні сценарії і впевнено налаштовує змінні середовища. Після цього створюється pull request, у якому описані мета та деталі оновлення. Після усунення потенційних конфліктів гілок, pull request зливається в головну гілку.

Після розгортання слід приділити увагу перевірці стабільності: перевірити журнали помилок, проглянути звіти помилок у користувачів і щоденно моніторити основні метрики. Якщо з’являються помилки, швидко створюється нова гілка з патчем, проходить локальне тестування, pull request і злиття в main, після чого знову вручну відбувається збірка та запуск оновленої версії. Документація оновлюється паралельно із кодом, щоб будь-які зміни в API або налаштуваннях середовища були завжди задокументовані. Такий підхід із регулярними pull request і ручним деплоєм дозволяє тримати код у GitHub актуальним і водночас контролювати кожне розгортання без автоматичних пайплайнів.

Висновки до розділу

Описано та показано, як система «E-queue» переходить у робочий режим: серверна частина на Nest.js зв’язується з хмарною базою даних MongoDB Atlas через збережений у середовищі URI, а два клієнтські Next.js-застосунки отримують необхідні ключі та URL сервера (включно з публічним ключем Uploadcare) через змінні оточення. Клієнт розгортається на Vercel, де після кожного пушу у вибрану гілку автоматично запускається збірка через Turborepo, а бекенд — на Render із налаштуванням Build та Start команд. Завдяки такій архітектурі всі складові проєкту швидко піднімаються в хмарі та коректно обмінюються даними без додаткової контейнеризації.

Супровід ПЗ організовано через GitHub-процес: зміни вносять у відокремлені гілки, протестовують локально, роблять pull request, а після рев’ю відповідальна особа вручну оновлює проєкти у Vercel і Render. Такий підхід забезпечує контроль якості кожного оновлення, дає змогу швидко випускати патчі та оновлювати документацію. У підсумку система залишається синхронною, а користувачі завжди отримують актуальну версію застосунку.

ВИСНОВКИ

У процесі реалізації дипломного проєкту «E-queue» була розроблена повноцінна клієнт-серверна система для управління електронними чергами, що враховує всі поставлені вимоги. Зокрема, проведено аналіз бізнес-процесів, сформульовано ключові функціональні вимоги — реєстрацію організацій та користувачів, керування динамічними й статичними чергами, розподіл ролей і можливість налаштовувати часові слоти. Розроблено UML-моделі й матрицю трасування, що підтверджують відповідність моделі Use Case всім виявленим процесам і дозволяють мінімізувати ризик пропуску важливих функцій . Архітектура застосунку базується на React/Next.js у ролі SPA-клієнта з WebSocket для миттєвого оновлення станів черги та на NestJS для обробки REST-запитів і подій у реальному часі. Дані зберігаються в MongoDB Atlas, файли користувачів та медіа розміщуються через Uploadcare, а безпечну аутентифікацію забезпечують JWT із хешуванням паролів Argon2 .

Питання безпеки вирішено впровадженням TLS-шифрування для всіх API-запитів, обмеженням доступу до серверних портів через файрвол, регулярним статичним аналізом коду з ESLint і перевіркою залежностей за допомогою npm audit. Це дозволило запобігти загрозам CSRF, XSS та NoSQL-ін’єкціям і мінімізувати ризики несанкціонованого доступу до облікових записів.

У процесі тестування виконано повний цикл мануального тестування ключових функціональних блоків — реєстрації, авторизації, управління чергами, ролями й сервісами, після чого підтверджено коректну роботу всіх основних сценаріїв і стабільну обробку одночасних WebSocket-з’єднань без втрати продуктивності. Розгортання клієнтської частини автоматизовано через Vercel, а серверної — через Render, що забезпечує швидке й безперебійне оновлення системи; супровід здійснюється через GitHub-процес із pull request та ручним рев’ю, що гарантує своєчасне виявлення й виправлення помилок .

Розроблений застосунок повністю відповідає поставленим завданням: він скорочує час очікування клієнтів, підвищує прозорість взаємодії між відвідувачами та адміністраторами й забезпечує оптимальне використання ресурсів організацій. Функціональність «E-queue» перевершує наявні рішення (Waitwhile, Qmatic) завдяки можливості прикріплення документів і медіа до записів, гнучкому керуванню ролями та поданню запитів на приєднання до штату організації.

Незважаючи на повне задоволення початкових вимог, подальший розвиток системи є доцільним. Насамперед бажано створити мобільний додаток на React Native для надання клієнтам можливості отримувати пуш-повідомлення про зміни в черзі й забезпечити комфортний UX на невеликих екранах. Додатково варто інтегрувати SMS- та Telegram-бота для нагадувань і підтвердження запису, що зменшить навантаження на адміністраторів. Запровадження аналітичних дашбордів у адміністративній панелі дозволить керівникам контролювати середній час очікування, кількість звернень і популярність послуг для оптимального розподілу ресурсів. Крім того, введення можливостей онлайн-оплати (Stripe, LiqPay) підвищить відсоток реалізації записів і зменшить «прогули» клієнтів. Реалізація багатомовного інтерфейсу (i18n) дасть змогу виходити на нові ринки, а впровадження гнучкої системи ролей і прав (RBAC) забезпечить делегування адміністрування різним менеджерам у межах однієї організації. Нарешті, для забезпечення масштабованості за високого навантаження варто розглянути перехід на мікросервісну архітектуру та використання Kubernetes, щоб окремо масштабувати WebSocket-сервіс, аналітику та інші ресурсоємні компоненти. Таким чином, «E-queue» в нинішньому вигляді ефективно вирішує поставлену задачу, а реалізація зазначених напрямків розвитку зробить його ще більш гнучким, масштабованим і привабливим для широкого кола організацій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Google Spreadsheets. *Google Workspace*. URL: https://workspace.google.com/products/sheets (дата звернення 10.05.2025).

2) How to Create a Ticketing System with Microsoft Excel? *Wrangle.* URL: https://www.wrangle.io/post/how-to-create-a-ticketing-system-with-microsoft-excel (дата звернення 10.05.2025).

3) Create a Call queue in Microsoft Teams. *Microsoft Learn*. URL: https://learn.microsoft.com/en-us/microsoftteams/create-a-phone-system-call-queue (дата звернення 10.05.2025).

4) Queue management solutions in Qmatic. *Qmatic*. URL: https://www.qmatic.com/solutions/queue-management (дата звернення 10.05.2025).

5) Queue management for frictionless customer experiences. *Waitwhile*. URL: https://waitwhile.com/solutions/queue-management-system (дата звернення 10.05.2025).

6) Zod Intro. *Zod*. URL: https://zod.dev (дата звернення 10.05.2025).

7) Elegant MongoDB object modeling for Node.js. *Mongoose*. URL: https://mongoosejs.com/ (дата звернення 10.05.2025).

8) Introduction to Turborepo. *Vercel Turborepo*. URL: https://turborepo.com/docs (дата звернення 10.05.2025).

9) The React Framework for the Web. *Vercel NextJs*. URL: https://nextjs.org/ (дата звернення 10.05.2025).

10) NestJS - A progressive Node.js framework. *NestJs*. URL: https://nestjs.com/ (дата звернення 10.05.2025).

11) TypeScript is JavaScript with syntax for types. *TypeScript*. URL: https://www.typescriptlang.org/ (дата звернення 10.05.2025).

12) Getting Started With MongoDB & Mongoose. *MongoDB*. URL: https://www.mongodb.com/developer/languages/javascript/getting-started-with-mongodb-and-mongoose/ (дата звернення 10.05.2025).

13) A query language for your API. *GraphQl*. URL: https://graphql.org/ (дата звернення 10.05.2025).

14) Uploadcare: File uploading, processing & delivery for web. *Uploadcare*. URL: https://uploadcare.com/products/file-uploader/ (дата звернення 10.05.2025).

15) Your code editor. Redefined with AI. *Microsoft VisualStudio*. URL: https://code.visualstudio.com/ (дата звернення 10.05.2025).

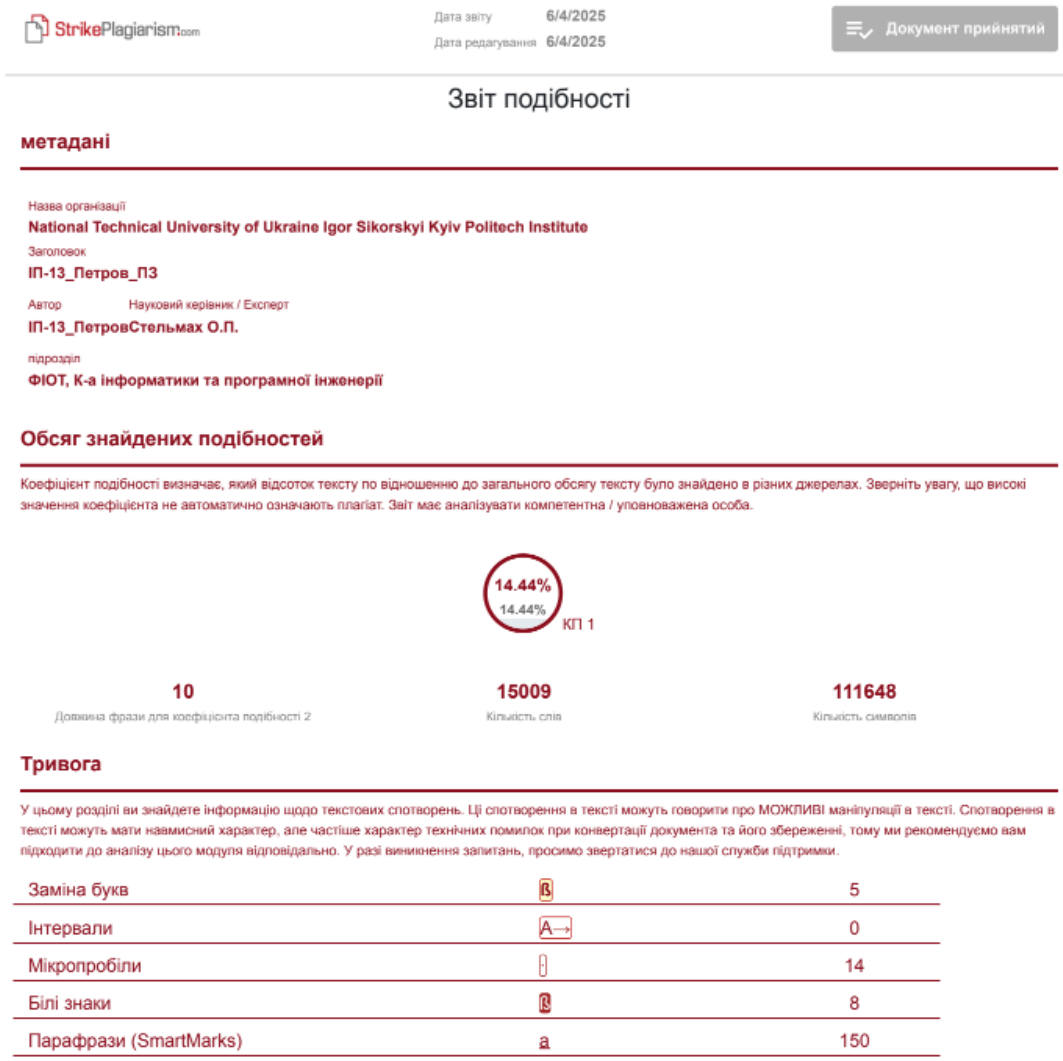
16) Find and fix problems in your JavaScript code. *Eslint*. URL: https://eslint.org/ (дата звернення 10.05.2025).

17) Opinionated code formatter – Prettier. *Prettier*. URL: https://prettier.io/docs/ (дата звернення 10.05.2025).

18) Make Your Software Vision Reality. *JetBrains WebStorm*. URL: https://www.jetbrains.com/ (дата звернення 10.05.2025).

ДОДАТКИ

ДОДАТОК А Звіт подібності



StrikePlagiarism.com

Дата звіту 6/4/2025
Дата редагування 6/4/2025

Документ прийнятий

Звіт подібності

метадані

Назва організації
National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute
Заголовок
ІП-13_Петров_ПЗ
Автор Науковий керівник / Експерт
ІП-13_ПетровСтельмах О.П.
підрозділ
ФІОТ, К-а інформатики та програмної інженерії

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

14.44%
14.44%
КП 1

10	15009	111648
Довжина фрази для коефіцієнта подібності 2	Кількість слів	Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	ß	5
Інтервали	A→	0
Мікропробіли	·	14
Білі знаки	¶	8
Парафрази (SmartMarks)	a	150

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”
Завідувач кафедри
_____ Едуард ЖАРІКОВ
“___” _____ 2025 р.

ВЕБЗАСТОСУНОК ДЛЯ ОПТИМІЗАЦІЇ ЕЛЕКТРОННИХ ЧЕРГ ТА ПРИЙОМУ

Текст програми

КПІ.ІП-1328.045440.03.12

“ПОГОДЖЕНО”
Керівник проєкту:
_____ Олександр СТЕЛЬМАХ

Нормоконтроль:
_____ Ірина ВІТКОВСЬКА

Виконавець:
_____ Ігор ПЕТРОВ

Київ – 2025

Посилання на репозиторій з повним текстом програмного коду
https://github.com/IgorPetrovIP-13/E-queue

Файл accessToken.strategy.ts
Реалізація функціональної задачі «Авторизація та реєстрація користувача»

```
import { Injectable } from "@nestjs/common";
import { PassportStrategy } from "@nestjs/passport";
import { ExtractJwt, Strategy } from "passport-jwt";
import { ConfigService } from "../../core/config/config.service";
import { JwtPayload } from "@repo/core/types/jwt-payload.types";

@Injectable()
export class AccessTokenStrategy extends PassportStrategy(Strategy, "jwt") {
  constructor(private readonly configService: ConfigService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: configService.jwtSecret
    });
  }

  validate(payload: JwtPayload) {
    return payload;
  }
}
```

Файл refreshToken.strategy.ts
Реалізація функціональної задачі «Авторизація та реєстрація користувача»

```
import { PassportStrategy } from "@nestjs/passport";
import { ExtractJwt, Strategy } from "passport-jwt";
import { Request } from "express";
import { Injectable } from "@nestjs/common";
import { ConfigService } from "../../core/config/config.service";
import { Tokens } from "@repo/core/enums/tokens";

@Injectable()
export class RefreshTokenStrategy extends PassportStrategy(
  Strategy,
  "jwt-refresh"
) {
  constructor(private readonly configService: ConfigService) {
    super({
      jwtFromRequest: ExtractJwt.fromExtractors([
        (req: Request) => req.cookies[Tokens.REFRESH_TOKEN]
      ]),
      secretOrKey: configService.jwtRefreshSecret
    });
  }

  validate(payload: any) {
    return payload;
  }
}
```

Файл auth.controller.ts
Реалізація функціональної задачі «Авторизація та реєстрація користувача»

```
import {
  Body,
  Controller,
  Post,
  Put,
  Req,
  Res,
  UseGuards
} from "@nestjs/common";
import { Request, Response } from "express";
import { AccessTokenGuard } from "../core/common/guards/accessToken.guard";
import { RefreshTokenGuard } from "../core/common/guards/refreshToken.guard";
import {
  createUserValidationSchema,
  CreateUserDTO
} from "../../app/user/user.dto";
import { AuthService } from "./auth.service";
import { ZodValidationPipe } from "../core/common/pipes/zod-validation.pipe";
import { Tokens, TokensExpiration } from "@repo/core/enums/tokens";
import { ConfigService } from "../core/config/config.service";
import { signInValidationSchema } from
"@repo/api/services/auth/auth.validation";
import { ISignInReq } from "@repo/api/services/auth/auth.types";

@Controller("auth")
export class AuthController {
  constructor(
    private authService: AuthService,
    private configService: ConfigService
  ) {}

  @Post("signup")
  async signUp(
    @Body(new ZodValidationPipe(createUserValidationSchema))
    body: CreateUserDTO,
    @Res({ passthrough: true }) res: Response
  ) {
    const { user, tokens } = await this.authService.signUp(body);
    this.setAuthCookies(res, tokens.refreshToken);
    return { user, accessToken: tokens.accessToken };
  }

  @Put("signin")
  async signIn(
    @Body(new ZodValidationPipe(signInValidationSchema)) body: ISignInReq,
    @Res({ passthrough: true }) res: Response
  ) {
    const { user, tokens } = await this.authService.signIn(body);
    this.setAuthCookies(res, tokens.refreshToken);
    return { user, accessToken: tokens.accessToken };
  }

  @Put("signin-admin")
  async signInAdmin(
    @Body(new ZodValidationPipe(signInValidationSchema)) body: ISignInReq,
    @Res({ passthrough: true }) res: Response
  ) {
    const { user, tokens } = await this.authService.signInAdmin(body);
    this.setAuthCookies(res, tokens.refreshToken);
    return { user, accessToken: tokens.accessToken };
  }

  @UseGuards(RefreshTokenGuard)
```

```
  @Put("refresh")
  async refresh(
    @Req() req: Request,
    @Res({ passthrough: true }) res: Response
  ) {
    const refreshToken = req.cookies[Tokens.REFRESH_TOKEN];
    const tokens = await this.authService.refresh(refreshToken);
    this.setAuthCookies(res, tokens.refreshToken);
    return { accessToken: tokens.accessToken };
  }

  @UseGuards(AccessTokenGuard)
  @Put("logout")
  logout(@Req() req: Request, @Res({ passthrough: true }) res: Response) {
    res.clearCookie(Tokens.REFRESH_TOKEN);
  }

  private setAuthCookies(res: Response, refreshToken: string) {
    res.cookie(Tokens.REFRESH_TOKEN, refreshToken, {
      httpOnly: true,
      secure: true,
      sameSite: true,
      maxAge: TokensExpiration.REFRESH_TOKEN,
      path: "/"
    });
  }
}
```

Файл auth.service.ts

Реалізація функціональної задачі «Авторизація та реєстрація користувача»

```
import {
  BadRequestException,
  ForbiddenException,
  Injectable
} from "@nestjs/common";
import { UserService } from "../user/user.service";
import * as argon2 from "argon2";
import { JwtService } from "@nestjs/jwt";
import { ConfigService } from "../core/config/config.service";
import { CreateUserDTO } from "../../app/user/user.dto";
import { UserWithTokens } from "./typedefs";
import { ISignInReq } from "@repo/api/services/auth/auth.types";
import { Roles } from "@repo/core/enums/roles";

@Injectable()
export class AuthService {
  constructor(
    private readonly userService: UserService,
    private readonly jwtService: JwtService,
    private readonly configService: ConfigService
  ) {}

  async signUp(data: CreateUserDTO): Promise<UserWithTokens> {
    const userExists = await this.userService.findByEmail(data.email);
    if (userExists) {
      throw new BadRequestException(
        "Користувач вже існує, спробуйте інший email"
      );
    }

    const hash = await this.hashData(data.password);
    const newUser = await this.userService.create({ ...data, password: hash });
```

```
    const tokens = await this.getTokens(
      newUser._id,
      newUser.email,
      newUser.role
    );
    return { user: newUser, tokens };
  }

  async signIn(data: ISignInReq): Promise<UserWithTokens> {
    const user = await this.userService.findByEmail(data.email);
    if (!user) throw new BadRequestException("Такого користувача не існує");
    const passwordMatches = await argon2.verify(user.password, data.password);
    if (!passwordMatches) throw new BadRequestException("Неправильний пароль");
    const tokens = await this.getTokens(user._id, user.email, user.role);
    return { user, tokens };
  }

  async signInAdmin(data: ISignInReq): Promise<UserWithTokens> {
    const user = await this.userService.findByEmail(data.email);
    if (!user) throw new BadRequestException("Такого користувача не існує");
    const passwordMatches = await argon2.verify(user.password, data.password);
    if (!passwordMatches) throw new BadRequestException("Неправильний пароль");
    if (user.role !== Roles.ADMIN)
      throw new ForbiddenException("Ви не маєте доступу до цієї дії");
    const tokens = await this.getTokens(user._id, user.email, user.role);
    return { user, tokens };
  }

  async refresh(refreshToken: string) {
    try {
      const payload = this.jwtService.verify(refreshToken, {
        secret: this.configService.jwtRefreshSecret
      });

      const tokens = await this.getTokens(
        payload.sub,
        payload.email,
        payload.role
      );
      return tokens;
    } catch {
      throw new ForbiddenException("Невалідний рефреш токен");
    }
  }

  private async getTokens(userId: string, email: string, role: string) {
    const [accessToken, refreshToken] = await Promise.all([
      this.jwtService.signAsync(
        { sub: userId, email, role },
        { secret: this.configService.jwtSecret, expiresIn: "15m" }
      ),
      this.jwtService.signAsync(
        { sub: userId, email, role },
        { secret: this.configService.jwtRefreshSecret, expiresIn: "7d" }
      )
    ]);

    return { accessToken, refreshToken };
  }

  private hashData(data: string) {
    return argon2.hash(data);
  }
}
```

Файл profile.controller.ts
Реалізація функціональної задачі «Обліковий запис користувача»

```
import {
  Body,
  Controller,
  Delete,
  Get,
  HttpCode,
  HttpStatus,
  Put,
  Req,
  Res,
  UseGuards
} from "@nestjs/common";
import { AccessTokenGuard } from "../core/common/guards/accessToken.guard";
import { ProfileService } from "./profile.service";
import { Response } from "express";
import { ZodValidationPipe } from "../core/common/pipes/zod-validation.pipe";
import { IUserRequest } from "@repo/core/types/user-request.types";
import { Tokens } from "@repo/core/enums/tokens";
import { updateProfileValidationSchema } from
"@repo/api/services/profile/profile.validation";
import { IUpdateProfileReq } from "@repo/api/services/profile/profile.types";

@UseGuards(AccessTokenGuard)
@Controller("profile")
export class ProfileController {
  constructor(private readonly profileService: ProfileService) {}

  @Get()
  get(@Req() req: IUserRequest) {
    return this.profileService.get(req.user.sub);
  }

  @Delete()
  @HttpCode(HttpStatus.NO_CONTENT)
  delete(@Req() req: IUserRequest, @Res() res: Response) {
    res.clearCookie(Tokens.REFRESH_TOKEN);
    return this.profileService.delete(req.user.sub);
  }

  @Put()
  update(
    @Req() req: IUserRequest,
    @Body(new ZodValidationPipe(updateProfileValidationSchema))
    data: IUpdateProfileReq
  ) {
    return this.profileService.update(req.user.sub, data);
  }
}
```

Файл profile.service.ts
Реалізація функціональної задачі «Обліковий запис користувача»

```
import { Injectable } from "@nestjs/common";
import { IUpdateProfileReq } from "@repo/api/services/profile/profile.types";
import { Types } from "mongoose";
import { UserService } from "../user/user.service";

@Injectable()
export class ProfileService {
```

```
  constructor(private readonly userService: UserService) {}

  async get(userId: Types.ObjectId) {
    const { password, ...payload } = await this.userService.findById(userId);
    return payload;
  }

  async update(userId: Types.ObjectId, data: IUpdateProfileReq) {
    return await this.userService.update(userId, data);
  }

  async delete(userId: Types.ObjectId) {
    return await this.userService.delete(userId);
  }
}
```

Файл useProfile.ts
Реалізація функціональної задачі «Обліковий запис користувача»

```
import { QueryCache, useQuery } from "@tanstack/react-query";
import { profileService } from "@repo/api/services/profile/profile.service";
import { IGetProfileRes as Profile } from
"@repo/api/services/profile/profile.types";

const queryCache = new QueryCache();

export function useProfile() {
  const cachedProfile = queryCache.find({ queryKey: ["profile"] })?.state
    .data as Profile | undefined;

  const { data, isLoading } = useQuery<Profile>({
    queryKey: ["profile"],
    queryFn: () => profileService.getProfile(),
    initialData: () => cachedProfile
  });

  return { data, isLoading };
}
```

Файл organization_request.controller.ts
Реалізація функціональної задачі «Система заявок організацій»

```
import {
  Body,
  Controller,
  Get,
  Param,
  Post,
  Put,
  Req,
  UseGuards
} from "@nestjs/common";
import { AccessTokenGuard } from "../core/common/guards/accessToken.guard";
import { OrganizationRequestService } from "./organization_request.service";
import { ZodValidationPipe } from "../core/common/pipes/zod-validation.pipe";
import { IUserRequest } from "@repo/core/types/user-request.types";
import { createOrganizationRequestValidationSchema } from
"@repo/api/services/organization-request/organization-request.validation";
import { ICreateOrganizationRequestReq } from "@repo/api/services/organization-
request/organization-request.types";
import { RolesGuard } from "../core/common/guards/roles.guard";
```

```
import { Roles } from "@repo/core/enums/roles";
import { RolesRequired } from "../core/common/decorators/roles.decorator";
import { Types } from "mongoose";

@UseGuards(AccessTokenGuard)
@Controller("organization-requests")
export class OrganizationRequestController {
  constructor(
    private readonly organizationRequestService: OrganizationRequestService
  ) {}

  @Post()
  async create(
    @Body(new ZodValidationPipe(createOrganizationRequestValidationSchema))
    data: ICreateOrganizationRequestReq,
    @Req() req: IUserRequest
  ) {
    const user_id = req.user.sub;
    return this.organizationRequestService.create({ user_id, ...data });
  }

  @Get("my-requests")
  async findMyRequests(@Req() req: IUserRequest) {
    return this.organizationRequestService.findMyRequests(req.user.sub);
  }

  @Get("available-requests")
  @RolesRequired(Roles.ADMIN)
  @UseGuards(RolesGuard)
  async findAvailableRequests() {
    return this.organizationRequestService.findAvailableRequests();
  }

  @Get("requests-under-my-review")
  @RolesRequired(Roles.ADMIN)
  @UseGuards(RolesGuard)
  async findRequestsUnderMyReview(@Req() req: IUserRequest) {
    return this.organizationRequestService.findRequestsUnderMyReview(
      req.user.sub
    );
  }

  @Get(":id")
  async findById(@Param("id") id: Types.ObjectId) {
    return this.organizationRequestService.findById(id);
  }

        @Put(":id/approve")
        @RolesRequired(Roles.ADMIN)
        @UseGuards(RolesGuard)
        async approve(
                @Param("id") id: Types.ObjectId,
                @Body() body: { comment: string },
        ) {
                return              this.organizationRequestService.approve(id,
body.comment);
        }

        @Put(":id/reject")
        @RolesRequired(Roles.ADMIN)
        @UseGuards(RolesGuard)
        async reject(
                @Param("id") id: Types.ObjectId,
                @Body() body: { comment: string },
```

```
        ) {
                return                this.organizationRequestService.reject(id,
body.comment);
        }

  @Get(":id/status")
  @RolesRequired(Roles.ADMIN)
  @UseGuards(RolesGuard)
  async findStatusById(@Param("id") id: Types.ObjectId) {
    return this.organizationRequestService.findStatusById(id);
  }

  @Put(":id/leave-comment")
  async leaveComment(
    @Param("id") id: Types.ObjectId,
    @Body() body: { comment: string },
    @Req() req: IUserRequest
  ) {
    return this.organizationRequestService.leaveComment(
      id,
      req.user.role,
      body.comment
    );
  }

  @Put(":id/take-into-processing")
  @RolesRequired(Roles.ADMIN)
  @UseGuards(RolesGuard)
  async takeIntoProcessing(
    @Param("id") id: Types.ObjectId,
    @Req() req: IUserRequest
  ) {
    return              this.organizationRequestService.takeIntoProcessing(id,
req.user.sub);
  }
}
```

Файл organization_request.service.ts

Реалізація функціональної задачі «Система заявок організацій»

```
import { Injectable } from "@nestjs/common";
import         {         OrganizationRequestRepostitory         }         from
"./organization_request.repository";
import { Types } from "mongoose";
import { OrganizationRequest } from "./organization_request.schema";
import       {       ICreateOrganizationRequestWithUserId       }       from
"./organization_request.types";
import { Role } from "@repo/core/enums/roles";
import { EventEmitter2 } from "@nestjs/event-emitter";
import { SocketEvents } from "@repo/core/constants/socket-events";
import { OrganizationService } from "../organization/organization.service";

@Injectable()
export class OrganizationRequestService {
  constructor(
    private              readonly              organizationRequestRepository:
OrganizationRequestRepostitory,
    private readonly organizationService: OrganizationService,
    private readonly events: EventEmitter2
  ) {}

  async create(
    data: ICreateOrganizationRequestWithUserId
```

```
  ): Promise<OrganizationRequest> {
    return await this.organizationRequestRepository.create(data);
  }

  async findMyRequests(userId: Types.ObjectId): Promise<OrganizationRequest[]>
{
    return await this.organizationRequestRepository.findByUserId(userId);
  }

  async findAvailableRequests(): Promise<OrganizationRequest[]> {
    return await this.organizationRequestRepository.findAvailableRequests();
  }

  async findById(id: Types.ObjectId): Promise<OrganizationRequest> {
    return await this.organizationRequestRepository.findById(id);
  }

  async leaveComment(id: Types.ObjectId, role: Role, comment: string) {
    const data = await this.organizationRequestRepository.leaveComment(
      id,
      role,
      comment
    );

    this.events.emit(SocketEvents.ORGANIZATION_REQUEST_CHAT_COMMENT, {
      room: `org_request_${id}`,
      message: data
    });

    return data;
  }

  async findStatusById(id: Types.ObjectId) {
    return await this.organizationRequestRepository.findStatusById(id);
  }

  async takeIntoProcessing(id: Types.ObjectId, userId: Types.ObjectId) {
    return await this.organizationRequestRepository.takeIntoProcessing(
      id,
      userId
    );
  }

  async findRequestsUnderMyReview(
    userId: Types.ObjectId
  ): Promise<OrganizationRequest[]> {
    return await this.organizationRequestRepository.findRequestsUnderMyReview(
      userId
    );
  }

  async approve(id: Types.ObjectId, comment: string) {
    const data = await this.organizationRequestRepository.approve(id, comment);

    const dataForOrg = {
      user_id: data.user_id,
      organization_logo: data.organization_logo,
      organization_type_id: data.organization_type_id,
      organization_title: data.organization_title,
      organization_description: data.organization_description,
      organization_website: data.organization_website as string,
    };
    await this.organizationService.create(dataForOrg);
```

```
    return data;
  }

  async reject(id: Types.ObjectId, comment: string) {
    const data = await this.organizationRequestRepository.reject(id, comment);
    return data;
  }
}
```

Файл organization_request.repository.ts
Реалізація функціональної задачі «Система заявок організацій»

```
import { Injectable, Logger } from "@nestjs/common";
import { InjectModel } from "@nestjs/mongoose";
import {
  OrganizationRequest,
  OrganizationRequestCollection
} from "./organization_request.schema";
import { Model, Types } from "mongoose";
import       {       ICreateOrganizationRequestWithUserId       }       from
"./organization_request.types";
import {
  OrganizationRequestStatus,
  OrganizationRequestStatuses
} from "@repo/core/enums/org-request-status";
import { Role, Roles } from "@repo/core/enums/roles";

@Injectable()
export class OrganizationRequestRepostitory {
  constructor(
    @InjectModel(OrganizationRequestCollection)
    private readonly organizationRequest: Model<OrganizationRequest>
  ) {}

  async create(
    data: ICreateOrganizationRequestWithUserId
  ): Promise<OrganizationRequest> {
    const organizationRequest = new this.organizationRequest(data);
    return organizationRequest.save();
  }

  async findByUserId(userId: Types.ObjectId): Promise<OrganizationRequest[]> {
    return this.organizationRequest
      .find({ user_id: userId })
      .populate("organization_type_id", "title")
      .exec();
  }

  async findAvailableRequests(): Promise<OrganizationRequest[]> {
    return this.organizationRequest
      .find({ status: OrganizationRequestStatuses.NOT_CHECKED })
      .populate("organization_type_id", "title")
      .exec();
  }

  async findById(id: Types.ObjectId): Promise<OrganizationRequest> {
    const organizationRequest = await this.organizationRequest
      .findById(id)
      .populate("organization_type_id", "title")
      .exec();

    if (!organizationRequest) {
      throw new Error("Organization request not found");
```

```
    }

    return organizationRequest;
  }

  async leaveComment(
    id: Types.ObjectId,
    userRole: Role,
    comment: string
  ): Promise<OrganizationRequest["comments"][number]> {
    const organizationRequest = await this.organizationRequest
      .findById(id)
      .exec();

    if (!organizationRequest) {
      throw new Error("Organization request not found");
    }

    const newComment = {
      comment,
      createdAt: new Date().toISOString(),
      isAdmin: userRole === Roles.ADMIN
    };

    organizationRequest.comments.push(newComment);
    await organizationRequest.save();

    return newComment;
  }

  async takeIntoProcessing(
    id: Types.ObjectId,
    userId: Types.ObjectId
  ): Promise<OrganizationRequest> {
    const updatedData = await this.organizationRequest.findByIdAndUpdate(id, {
      status: OrganizationRequestStatuses.PENDING,
      admin_id: userId
    });

    if (!updatedData) {
      throw new Error("Organization request not found");
    }

    return updatedData;
  }

  async findStatusById(id: Types.ObjectId): Promise<OrganizationRequestStatus>
{
    const organizationRequest = await this.organizationRequest
      .findById(id)
      .select("status")
      .exec();

    if (!organizationRequest) {
      throw new Error("Organization request not found");
    }

    return organizationRequest.status;
  }

  async findRequestsUnderMyReview(
    userId: Types.ObjectId
  ): Promise<OrganizationRequest[]> {
    return this.organizationRequest
```

```
      .find({ admin_id: userId, status: OrganizationRequestStatuses.PENDING })
      .populate("organization_type_id", "title")
      .exec();
  }

  async approve(
    id: Types.ObjectId,
    comment: string
  ): Promise<OrganizationRequest> {
    const updatedData = await this.organizationRequest.findByIdAndUpdate(
      id,
      {
        status: OrganizationRequestStatuses.APPROVED,
        approval_comment: comment
      },
      { new: true }
    );
    if (!updatedData) {
      throw new Error("Organization request not found");
    }
    return updatedData;
  }

  async reject(
    id: Types.ObjectId,
    comment: string
  ): Promise<OrganizationRequest> {
    const updatedData = await this.organizationRequest.findByIdAndUpdate(
      id,
      {
        status: OrganizationRequestStatuses.REJECTED,
        rejection_comment: comment
      },
      { new: true }
    );
    if (!updatedData) {
      throw new Error("Organization request not found");
    }
    return updatedData;
  }
}
```

Файл gateway.ts

Реалізація функціональної задачі «Система заявок організацій»

```
import { Logger } from "@nestjs/common";
import {
  ConnectedSocket,
  MessageBody,
  OnGatewayConnection,
  OnGatewayDisconnect,
  OnGatewayInit,
  SubscribeMessage,
  WebSocketGateway,
  WebSocketServer
} from "@nestjs/websockets";
import { Server, Socket } from "socket.io";
import { OnEvent } from '@nestjs/event-emitter';
import { IGeneralNotification, SocketEvents } from
"@repo/core/constants/socket-events";
import { IComment } from
"../../organization_request/organization_request.types";
import { Types } from "mongoose";
```

```
@WebSocketGateway({
  namespace: "/",
  cors: { origin: true, credentials: true }
})
export class Gateway
  implements OnGatewayInit, OnGatewayConnection, OnGatewayDisconnect
{
  @WebSocketServer() server: Server;

  afterInit(server: Server) {
    Logger.log("WebSocket server initialized");
  }

  handleConnection(client: Socket) {
    Logger.log(`Client connected: ${client.id}`);
  }

  handleDisconnect(client: Socket) {
    Logger.log(`Client disconnected: ${client.id}`);
  }

  @SubscribeMessage("subscribe")
  handleSubscribe(
    @ConnectedSocket() client: Socket,
    @MessageBody() { room }: { room: string }
  ) {
    client.join(room);
  }

  @SubscribeMessage("unsubscribe")
  handleUnsubscribe(
    @ConnectedSocket() client: Socket,
    @MessageBody() { room }: { room: string }
  ) {
    client.leave(room);
  }

  @OnEvent(SocketEvents.ORGANIZATION_REQUEST_CHAT_COMMENT)
  handleMessageNew(payload: { room: string; message: IComment }) {

this.server.to(payload.room).emit(SocketEvents.ORGANIZATION_REQUEST_CHAT_COMM
ENT, payload.message);
  }

        @OnEvent(SocketEvents.GENERAL_NOTIFICATION)
        handleGeneralNotification(payload: { userId: Types.ObjectId; message:
IGeneralNotification }) {

        this.server.to(`general_notification/${payload.userId.toString()}`).em
it(SocketEvents.GENERAL_NOTIFICATION, payload.message);
        }
}
```

Файл organization.controller.ts

Реалізація функціональної задачі «Управління організаціями»

```
import { Controller, Get, Put, Req, UseGuards } from "@nestjs/common";
import { AccessTokenGuard } from "../core/common/guards/accessToken.guard";
import { OrganizationService } from "./organization.service";
import { IUserRequest } from "@repo/core/types/user-request.types";

@UseGuards(AccessTokenGuard)
@Controller("organizations")
```

```
export class OrganizationController {
  constructor(private readonly organizationService: OrganizationService) {}

        @Get("my-organizations")
        async findMyOrganizations(@Req() req: IUserRequest) {
                const user_id = req.user.sub;
                return this.organizationService.findByUserId(user_id);
        }

        @Put()
        async updateOrganization(
                @Req() req: IUserRequest
        ) {
                const user_id = req.user.sub;
                const organization_id = req.body.organization_id;
                const data = req.body.data;

                return this.organizationService.updateOrganization(
                        user_id,
                        organization_id,
                        data
                );
        }
}
```

Файл organization.service.ts

Реалізація функціональної задачі «Управління організаціями»

```
import { Injectable, Logger } from "@nestjs/common";
import { Organization } from "./organization.schema";
import { ObjectId } from "mongodb";
import { OrganizationRoles } from "@repo/core/enums/org-roles";
import { OrganizationRepository } from "./organization.repository";
import { Types } from "mongoose";
import { CreateOrganizationDTO } from "./typedefs";
import          {          UserOrganizationService          }          from
"../user_organizations/user_organizations.service";

@Injectable()
export class OrganizationService {
  constructor(
    private readonly organizationRepository: OrganizationRepository,
    private readonly userOrganizationService: UserOrganizationService
  ) {}

  async create(data: CreateOrganizationDTO): Promise<Organization> {
    const objectId = new ObjectId();

    const userOwner = await this.userOrganizationService.create({
      user_id: data.user_id,
      organization_id: objectId,
      organization_role: OrganizationRoles.OWNER
    });

    const dataWithId = {
      _id: objectId,
      organization_logo: data.organization_logo,
      organization_type_id: data.organization_type_id,
      organization_title: data.organization_title,
      organization_description: data.organization_description,
      organization_website: data.organization_website,
      members: [userOwner._id as unknown as Types.ObjectId],
      static_queues: [],
      dynamic_queues: []
```

```
    };

    return await this.organizationRepository.create(dataWithId);
  }

  async findByUserId(user_id: Types.ObjectId): Promise<Organization[]> {
    const userOrgs = await this.userOrganizationService.findByUserId(user_id);
    const orgIds = userOrgs.map(org => org.organization_id);

    return await this.organizationRepository.findByIds(orgIds);
  }

  async addQueue(organization_id: string, queue_id: string) {
    return await this.organizationRepository.addQueue(
      organization_id,
      queue_id
    );
  }

  async addDynamicQueue(organization_id: string, dynamic_queue_id: string) {
    return await this.organizationRepository.addDynamicQueue(
      organization_id,
      dynamic_queue_id
    );
  }

  async deleteDynamicQueue(organization_id: string, dynamic_queue_id: string)
{
    return await this.organizationRepository.deleteDynamicQueue(
      organization_id,
      dynamic_queue_id
    );
  }
}
```

Файл organization.repository.ts

Реалізація функціональної задачі «Управління організаціями»

```
import { Injectable } from "@nestjs/common";
import { Organization, OrganizationCollection } from "./organization.schema";
import { Model, Types } from "mongoose";
import { InjectModel } from "@nestjs/mongoose";

@Injectable()
export class OrganizationRepository {
  constructor(
    @InjectModel(OrganizationCollection)
    private readonly organizationModel: Model<Organization>
  ) {}

  async create(data: Partial<Organization>): Promise<Organization> {
    const organization = new this.organizationModel(data);
    return organization.save();
  }

  async findByIds(ids: Types.ObjectId[]): Promise<Organization[]> {
    return this.organizationModel
      .find({ _id: { $in: ids } })
      .populate("organization_type_id")
      .populate({
        path: "members",
        populate: {
          path: "user_id"
```

```
        }
      })
      .lean()
      .exec();
  }

  async addQueue(id: string, queueId: string) {
    return this.organizationModel.findByIdAndUpdate(
      id,
      { $addToSet: { static_queues: queueId } },
      { new: true }
    );
  }

  async addDynamicQueue(id: string, dynamicQueueId: string) {
    return this.organizationModel.findByIdAndUpdate(
      id,
      { $addToSet: { dynamic_queues: dynamicQueueId } },
      { new: true }
    );
  }

       async deleteDynamicQueue(id: string, dynamicQueueId: string) {
              return this.organizationModel.findByIdAndUpdate(
                     id,
                     { $pull: { dynamic_queues: dynamicQueueId } },
                     { new: true }
              );
       }
}
```

Файл organization_invite.controller.ts
Реалізація функціональної задачі «Управління організаціями»

```
import { Body, Controller, Get, Post, Req, UseGuards } from '@nestjs/common';
import { OrganizationInviteService } from './organization_invite.service';
import          {          IInviteOrganizationReq          }          from
'@repo/api/services/organization_invite/organization-invite.types';
import { IUserRequest } from '@repo/core/types/user-request.types';
import { AccessTokenGuard } from '../core/common/guards/accessToken.guard';

@UseGuards(AccessTokenGuard)
@Controller('organization-invites')
export class OrganizationInviteController {
        constructor(private        readonly        organizationInviteService:
OrganizationInviteService) {}

  @Get("my-invites")
  async findMyInvites(@Req() req: IUserRequest) {
    return this.organizationInviteService.findMyInvites(req.user.email);
  }

  @Post("send-invite")
    async   sendInvite(@Body()   body:   IInviteOrganizationReq,   @Req()   req:
IUserRequest) {
    return this.organizationInviteService.sendInvite(req.user.sub, body);
  }
}
```

Файл organization_invite.service.ts
Реалізація функціональної задачі «Управління організаціями»

```
import { Injectable } from "@nestjs/common";
```

```
import { IInviteOrganizationReq } from
"@repo/api/services/organization_invite/organization-invite.types";
import { Types } from "mongoose";
import { OrganizationInviteRepository } from
"./organization_invite.repository";

@Injectable()
export class OrganizationInviteService {
  constructor(
    private readonly organizationInviteRepository:
OrganizationInviteRepository
  ) {}

  async sendInvite(user_id: Types.ObjectId, body: IInviteOrganizationReq) {
    const newData = {
      user_id: user_id,
      organization_id: body.organization_id,
      invitation_comment: body.invitation_comment,
      ivitation_role: body.ivitation_role,
      user_email: body.user_email
    };

    return this.organizationInviteRepository.sendInvite(newData);
  }

       async findMyInvites(user_email: string) {
              return
this.organizationInviteRepository.findMyInvites(user_email);
       }
}
```

Файл user_organizations.service.ts

Реалізація функціональної задачі «Управління організаціями»

```
import { Injectable, InternalServerErrorException } from "@nestjs/common";
import { UserOrganizationRepository } from "./user_organizations.repository";
import { CreateUserOrganizationDTO } from "./typedefs";
import { Types } from "mongoose";
import { UserService } from "../user/user.service";

@Injectable()
export class UserOrganizationService {
  constructor(
    private readonly userOrganizationRepository: UserOrganizationRepository,
    private readonly userService: UserService
  ) {}

  async create(data: CreateUserOrganizationDTO) {
    try {
                    await this.userService.findById(data.user_id);
      return await this.userOrganizationRepository.create(data);
    } catch (error) {
      throw new InternalServerErrorException(error);
    }
  }

  async findByUserId(user_id: Types.ObjectId) {
    try {
      return await this.userOrganizationRepository.findByUserId(user_id);
    } catch (error) {
      throw new InternalServerErrorException(error);
    }
  }
}
```

Файл user_organizations.repository.ts
Реалізація функціональної задачі «Управління організаціями»

```
import { Injectable } from "@nestjs/common";
import { InjectModel } from "@nestjs/mongoose";
import { Model, Types } from "mongoose";
import { CreateUserOrganizationDTO } from "./typedefs";
import {
  UserOrganization,
  UserOrganizationsCollection
} from "./user_organizations.schema";

@Injectable()
export class UserOrganizationRepository {
  constructor(
    @InjectModel(UserOrganizationsCollection)
    private readonly userOrganizationModel: Model<UserOrganization>
  ) {}

  async create(data: CreateUserOrganizationDTO) {
    const createdUserOrganization = new this.userOrganizationModel(data);
    return createdUserOrganization.save();
  }

  async findByUserId(user_id: Types.ObjectId) {
    return this.userOrganizationModel
      .find({ user_id: user_id })
      .lean()
      .exec();
  }
}
```

Файл static_queue.controller.ts
Реалізація функціональної задачі «Черги»

```
import { Body, Controller, Get, Param, Post, Req, UseGuards } from
"@nestjs/common";
import { StaticQueueService } from "./static_queue.service";
import { ICreateStaticQueueReq } from "@repo/api/services/static-queue/static-
queue.types";
import { createStaticQueueSchema } from "@repo/api/services/static-
queue/static-queue.validation";
import { ZodValidationPipe } from "../core/common/pipes/zod-validation.pipe";
import { IUserRequest } from "@repo/core/types/user-request.types";
import { AccessTokenGuard } from "../core/common/guards/accessToken.guard";

@UseGuards(AccessTokenGuard)
@Controller("static-queue")
export class StaticQueueController {
  constructor(private readonly staticQueueService: StaticQueueService) {}

  @Post()
  async                                          createStaticQueue(@Body(new
ZodValidationPipe(createStaticQueueSchema)) body: ICreateStaticQueueReq) {
    return this.staticQueueService.createStaticQueue(body);
  }

        @Get("as-executor")
        async getStaticQueueAsExecutor(@Req() req: IUserRequest) {
                return
this.staticQueueService.getStaticQueueAsExecutor(req.user.sub);
        }
```

```
        @Get("as-user/:id")
        async getStaticQueueAsUser(@Param("id") id: string) {
                return this.staticQueueService.getStaticQueueAsUser(id);
        }
}
```

Файл static_queue.service.ts
Реалізація функціональної задачі «Черги»

```
import { Injectable } from "@nestjs/common";
import { StaticQueueRepository } from "./static_queue.repository";
import { ICreateStaticQueueReq } from "@repo/api/services/static-queue/static-
queue.types";
import { OrganizationService } from "../organization/organization.service";
import { Types } from "mongoose";

@Injectable()
export class StaticQueueService {
  constructor(
    private readonly staticQueueRepository: StaticQueueRepository,
    private readonly organizationService: OrganizationService
  ) {}

  async createStaticQueue(data: ICreateStaticQueueReq) {
    const staticQueue =
      await this.staticQueueRepository.createStaticQueue(data);
    const staticQueueId = staticQueue._id.toString();

    await this.organizationService.addQueue(
      data.organization_id,
      staticQueueId
    );

    return staticQueue;
  }

  async getStaticQueueAsExecutor(userId: Types.ObjectId) {
    return this.staticQueueRepository.getStaticQueueAsExecutor(userId);
  }

  async getStaticQueueAsUser(id: string) {
    return this.staticQueueRepository.getStaticQueueAsUser(id);
  }

  async addAppointmentToStaticQueue(
    staticQueueId: string,
    appointmentId: string
  ) {
    return this.staticQueueRepository.addAppointmentToStaticQueue(
      staticQueueId,
      appointmentId
    );
  }
}
```

Файл static_queue.repository.ts
Реалізація функціональної задачі «Черги»

```
import { Injectable } from "@nestjs/common";
import { InjectModel } from "@nestjs/mongoose";
import { StaticQueue, StaticQueueCollection } from "./static_queue.schema";
import { Model, Types } from "mongoose";
```

```
import { ICreateStaticQueueReq } from "@repo/api/services/static-queue/static-
queue.types";

@Injectable()
export class StaticQueueRepository {
  constructor(
    @InjectModel(StaticQueueCollection)
    private readonly staticQueueModel: Model<StaticQueue>
  ) {}

  async createStaticQueue(data: ICreateStaticQueueReq) {
    const staticQueue = new this.staticQueueModel(data);
    return staticQueue.save();
  }

  async getStaticQueueAsExecutor(user_id: Types.ObjectId) {
    return this.staticQueueModel
      .find({ executor: user_id })
      .populate({
        path: "organization_id",
        populate: {
          path: "organization_type_id"
        }
      })
      .exec();
  }

  async getStaticQueueAsUser(id: string) {
    return this.staticQueueModel
      .findOne({ _id: id })
      .populate("organization_id")
      .populate({ path: "appointments", populate: { path: "user_id" } })
      .exec();
  }

  async addAppointmentToStaticQueue(
    staticQueueId: string,
    appointmentId: string
  ) {
    return this.staticQueueModel.findByIdAndUpdate(staticQueueId, {
      $addToSet: { appointments: appointmentId }
    });
  }
}
```

Файл appointments.controller.ts
Реалізація функціональної задачі «Черги»

```
import { Controller, Get, Post, Body, Req, UseGuards } from "@nestjs/common";
import { AppointmentsService } from "./appointments.service";
import { ICreateAppointmentReq } from
"@repo/api/services/appointment/appointment.types";
import { IUserRequest } from "@repo/core/types/user-request.types";
import { AccessTokenGuard } from "../core/common/guards/accessToken.guard";

@UseGuards(AccessTokenGuard)
@Controller("appointments")
export class AppointmentsController {
  constructor(private readonly appointmentsService: AppointmentsService) {}

  @Post()
  async createAppointment(
    @Body() body: ICreateAppointmentReq,
    @Req() req: IUserRequest
```

```
  ) {
    return this.appointmentsService.createAppointment(body, req.user.sub);
  }

  @Get("as-user")
  async getAppointmentsAsUser(@Req() req: IUserRequest) {
    return this.appointmentsService.getAppointmentsAsUser(req.user.sub);
  }

  @Get("my-today-appointments")
  async getMyTodayAppointments(@Req() req: IUserRequest) {
    return this.appointmentsService.getMyTodayAppointments(req.user.sub);
  }
}
```

Файл appointments.service.ts

Реалізація функціональної задачі «Черги»

```
import { Injectable } from "@nestjs/common";
import { ICreateAppointmentReq } from "@repo/api/services/appointment/appointment.types";
import { Types } from "mongoose";
import { AppointmentsRepository } from "./appointments.repository";
import { StaticQueueService } from "../static_queue/static_queue.service";

@Injectable()
export class AppointmentsService {
  constructor(
    private readonly appointmentsRepository: AppointmentsRepository,
    private readonly staticQueueService: StaticQueueService
  ) {}

  async createAppointment(data: ICreateAppointmentReq, userId: Types.ObjectId) {
    const dataWithUserId = {
      ...data,
      user_id: userId
    };

    const createdAppointemnt = await this.appointmentsRepository.createAppointment(dataWithUserId);

                await this.staticQueueService.addAppointmentToStaticQueue(
                        data.queue_id,
                        createdAppointemnt._id.toString()
                );

                return createdAppointemnt;
  }

        async getAppointmentsAsUser(userId: Types.ObjectId) {
                return this.appointmentsRepository.getAppointmentsAsUser(userId);
        }

        async getMyTodayAppointments(userId: Types.ObjectId) {
                const rawData = await this.appointmentsRepository.getMyTodayAppointments(userId);

                return rawData.map((appointment) => {
                        return {
                                _id: appointment._id.toString(),
                                time: `${appointment.start_time} - ${appointment.end_time}`,
```

```
                                title: appointment.queue_id.title,
                                organization_title:
appointment.queue_id.organization_id.organization_title,
                            }
                    })
            }
}
```

Файл appointments.repository.ts
Реалізація функціональної задачі «Черги»

```
import { Injectable } from "@nestjs/common";
import { ICreateAppointment } from "./typedefs";
import { InjectModel } from "@nestjs/mongoose";
import { Appointment, AppointmentsCollection } from "./appointments.schema";
import { Model, Types } from "mongoose";

@Injectable()
export class AppointmentsRepository {
  constructor(
    @InjectModel(AppointmentsCollection)
    private readonly appointment: Model<Appointment>
  ) {}

  async createAppointment(data: ICreateAppointment) {
    const appointment = new this.appointment(data);
    return appointment.save();
  }

  async getAppointmentsAsUser(userId: Types.ObjectId) {
    return this.appointment.find({ user_id: userId }).populate({
      path: "queue_id",
      populate: {
        path: "organization_id"
      }
    });
  }

  async getMyTodayAppointments(userId: Types.ObjectId) {
    const today = new Date();
    const yyyy = today.getFullYear();
    const mm = String(today.getMonth() + 1);
    const dd = String(today.getDate());
    const todayString = `${dd}.${mm}.${yyyy}`;

    return this.appointment
      .find({
        user_id: userId,
        date: todayString
      })
      .populate({
        path: "queue_id",
        populate: {
          path: "organization_id"
        }
      })
      .lean() as any;
  }
}
```

Файл SlotPicker.tsx
Реалізація функціональної задачі «Черги»

```
"use client";

import { useMemo, useState } from "react";
import { Button } from "@heroui/react";

type Service = {
  work_start_time: string;
  work_end_time: string;
  work_break_start_time: string;
  work_break_end_time: string;
  work_time_estimation: number;
  break_time_estimation: number;
};

interface BusySlot {
  date: string;
  start_time: string;
  end_time: string;
}

interface SlotsTableProps {
  service: Service;
  onSelect: (slot: string) => void;
  date: string;
  busySlots: BusySlot[];
}

export default function SlotsTable({ service, onSelect, date, busySlots }:
SlotsTableProps) {
  const [activeSlot, setActiveSlot] = useState<string | null>(null);

  const slots = useMemo(() => {
    const toMinutes = (t: string) => {
      const [h, m] = t.split(":").map(Number);
      return h * 60 + m;
    };
    const formatTime = (mins: number) => {
      const date = new Date();

      date.setHours(0, mins);

      return date.toLocaleTimeString([], { hour: "2-digit", minute: "2-digit"
});
    };
    const labels: string[] = [];

    const overlaps = (start1: number, end1: number, start2: number, end2:
number) => {
      return Math.max(start1, start2) < Math.min(end1, end2);
    };

    const isBusy = (start: number, end: number) => {
      return busySlots.some(slot =>
        slot.date === date &&
        overlaps(start,            end,            toMinutes(slot.start_time),
toMinutes(slot.end_time))
      );
    };

    const gen = (from: number, to: number) => {
      let cur = from;
      while (cur + service.work_time_estimation <= to) {
        const end = cur + service.work_time_estimation;
        const label = `${formatTime(cur)} - ${formatTime(end)}`;
```

```
        labels.push(JSON.stringify({
          label,
          start: cur,
          end,
          disabled: isBusy(cur, end),
        }));

        cur += service.work_time_estimation + service.break_time_estimation;
      }
    };

    const start = toMinutes(service.work_start_time);
    const bs = toMinutes(service.work_break_start_time);
    const be = toMinutes(service.work_break_end_time);
    const end = toMinutes(service.work_end_time);

    gen(start, bs);
    gen(be, end);

    return labels.map(str => JSON.parse(str));
  }, [service, busySlots, date]);

  return (
    <div className="grid grid-cols-5 gap-4">
      {slots.map(({ label, disabled }: any) => (
        <Button
          key={label}
          className="text-md flex justify-center items-center h-full"
          color={activeSlot === label ? "primary" : "default"}
          isDisabled={disabled}
          size="sm"
          variant={activeSlot === label ? "shadow" : "faded"}
          onPress={() => {
            onSelect(label);
            setActiveSlot(label);
          }}
        >
          {label}
        </Button>
      ))}
    </div>
  );
}
```

Файл dynamic_queues.controller.ts
Реалізація функціональної задачі «Черги»

```
import {
  Body,
  Controller,
  Delete,
  Get,
  Param,
  Post,
  Put,
  Req,
  UseGuards
} from "@nestjs/common";
import { DynamicQueuesService } from "./dynamic_queues.service";
import { ICreateStaticQueueReq } from "@repo/api/services/static-queue/static-
queue.types";
import { ZodValidationPipe } from "../core/common/pipes/zod-validation.pipe";
```

```
import { createDynamicQueueSchema } from "@repo/api/services/dynamic-
queue/dynamic-queue.validation";
import { IUserRequest } from "@repo/core/types/user-request.types";
import { AccessTokenGuard } from "../core/common/guards/accessToken.guard";
import { Types } from "mongoose";

@UseGuards(AccessTokenGuard)
@Controller("dynamic-queues")
export class DynamicQueuesController {
  constructor(private readonly dynamicQueuesService: DynamicQueuesService) {}

  @Post()
  async createStaticQueue(
    @Body(new ZodValidationPipe(createDynamicQueueSchema))
    body: ICreateStaticQueueReq
  ) {
    return this.dynamicQueuesService.create(body);
  }

  @Get("as-executor")
  async getDynamicQueuesAsExecutor(@Req() req: IUserRequest) {
    return this.dynamicQueuesService.getDynamicQueuesAsExecutor(req.user.sub);
  }

  @Get("as-client")
  async getDynamicQueuesAsClient(@Req() req: IUserRequest) {
    return this.dynamicQueuesService.getDynamicQueuesAsClient(req.user.sub);
  }

  @Get("as-executor/:id")
  async getDynamicQueueAsExecutor(@Param("id") id: string) {
    return this.dynamicQueuesService.getDynamicQueueAsExecutor(id);
  }

  @Get("as-client/:id")
  async getDynamicQueueAsClient(@Param("id") id: string) {
    return this.dynamicQueuesService.getDynamicQueueAsClient(id);
  }

  @Put("add-appointment/:id")
  async addAppointmentToDynamicQueue(
    @Param("id") id: string,
    @Body("email") email: string
  ) {
    return this.dynamicQueuesService.addAppointmentToDynamicQueue(id, email);
  }

  @Put("delete-appointment/:id")
  async deleteAppointmentFromDynamicQueue(
    @Param("id") id: string,
    @Body("userId") userId: Types.ObjectId
  ) {
    return this.dynamicQueuesService.deleteAppointmentFromDynamicQueue(
      id,
      userId
    );
  }

  @Put("delete-my-appointment/:id")
  async deleteMyAppointmentFromDynamicQueue(
    @Param("id") id: string,
    @Req() req: IUserRequest
  ) {
    return this.dynamicQueuesService.deleteMyAppointmentFromDynamicQueue(
```

```
      id,
      req.user.sub
    );
  }

        @Delete(":id")
        async deleteDynamicQueue(@Param("id") id: Types.ObjectId) {
                return this.dynamicQueuesService.deleteDynamicQueue(id);
        }
}
```

Файл dynamic_queues.service.ts
Реалізація функціональної задачі «Черги»

```
import {
  Injectable,
  InternalServerErrorException,
  Logger
} from "@nestjs/common";
import { ICreateDynamicQueueReq } from "@repo/api/services/dynamic-
queue/dynamic-queue.types";
import { DynamicQueueRepository } from "./dynamic_queue.repository";
import { OrganizationService } from "../organization/organization.service";
import { Types } from "mongoose";
import { UserService } from "../user/user.service";

@Injectable()
export class DynamicQueuesService {
  constructor(
    private readonly dynamicQueueRepository: DynamicQueueRepository,
    private readonly organizationService: OrganizationService,
    private readonly userService: UserService
  ) {}

  async create(data: ICreateDynamicQueueReq) {
    const dynamicQueue =
      await this.dynamicQueueRepository.createDynamicQueue(data);
    const dynamicQueueId = dynamicQueue._id.toString();

    await this.organizationService.addDynamicQueue(
      data.organization_id,
      dynamicQueueId
    );

    return dynamicQueue;
  }

  async getDynamicQueuesAsExecutor(userId: Types.ObjectId) {
    return this.dynamicQueueRepository.getDynamicQueuesAsExecutor(userId);
  }

  async getDynamicQueuesAsClient(userId: Types.ObjectId) {
    return this.dynamicQueueRepository.getDynamicQueuesAsClient(userId);
  }

  async getDynamicQueueAsExecutor(id: string) {
    return this.dynamicQueueRepository.getById(id);
  }

  async getDynamicQueueAsClient(id: string) {
    return this.dynamicQueueRepository.getById(id);
  }

  async addAppointmentToDynamicQueue(id: string, userEmail: string) {
```

```
    const user = await this.userService.findByEmail(userEmail);

    if (!user) {
      throw new InternalServerErrorException(
        "Користувача з таким email не знайдено"
      );
    }

    await this.dynamicQueueRepository.addAppointmentToDynamicQueue(
      id,
      user._id as unknown as Types.ObjectId
    );

    return user;
  }

  async deleteAppointmentFromDynamicQueue(id: string, userId: Types.ObjectId)
{
    const user = await this.userService.findById(userId);

    await this.dynamicQueueRepository.deleteAppointmentFromDynamicQueue(
      id,
      userId
    );

    return user;
  }

  async deleteMyAppointmentFromDynamicQueue(
    id: string,
    userId: Types.ObjectId
  ) {
    const data =
      await this.dynamicQueueRepository.deleteAppointmentFromDynamicQueue(
        id,
        userId
      );

    return data;
  }

  async deleteDynamicQueue(id: Types.ObjectId) {
    const deletedQueue =
      await this.dynamicQueueRepository.deleteDynamicQueue(id);

    if (deletedQueue) {
      await this.organizationService.deleteDynamicQueue(
        deletedQueue.organization_id.toString(),
        deletedQueue._id.toString()
      );
      return deletedQueue;
    }
  }
}
```

Файл dynamic_queues.repository.ts
Реалізація функціональної задачі «Черги»

```
import {
  BadRequestException,
  Injectable,
  InternalServerErrorException,
  Logger
} from "@nestjs/common";
```

```
import { InjectModel } from "@nestjs/mongoose";
import { Model, Types } from "mongoose";
import { DynamicQueue, DynamicQueueCollection } from "./dynamic_queue.schema";

@Injectable()
export class DynamicQueueRepository {
  constructor(
    @InjectModel(DynamicQueueCollection)
    private readonly dynamicQueueModel: Model<DynamicQueue>
  ) {}

  async createDynamicQueue(data: any) {
    const dynamicQueue = new this.dynamicQueueModel(data);
    return dynamicQueue.save();
  }

  async getDynamicQueuesAsExecutor(executorId: Types.ObjectId) {
    return this.dynamicQueueModel
      .find({ executor: executorId })
      .populate("executor")
      .populate("organization_id")
      .populate("appointments");
  }

  async getDynamicQueuesAsClient(clientId: Types.ObjectId) {
    return this.dynamicQueueModel
      .find({ appointments: clientId })
      .populate("executor")
      .populate("organization_id")
      .populate("appointments");
  }

  async getById(id: string) {
    return this.dynamicQueueModel
      .findById(id)
      .populate("executor")
      .populate("organization_id")
      .populate("appointments");
  }

  async addAppointmentToDynamicQueue(id: string, userId: Types.ObjectId) {
    const dynamicQueue = await this.dynamicQueueModel.findById(id);
    if (!dynamicQueue) {
      throw new InternalServerErrorException("Dynamic queue not found");
    }

    Logger.log(dynamicQueue.appointments);

    if (dynamicQueue.appointments.includes(userId)) {
      throw new InternalServerErrorException(
        "Користувач вже записаний у цю чергу"
      );
    }

    dynamicQueue.appointments.push(userId);
    return await dynamicQueue.save();
  }

  async deleteAppointmentFromDynamicQueue(id: string, userId: Types.ObjectId)
{
    const dynamicQueue = await this.dynamicQueueModel.findById(id);
    if (!dynamicQueue) {
      throw new InternalServerErrorException("Dynamic queue not found");
    }
```

```
    if (!dynamicQueue.appointments.includes(userId)) {
      throw new BadRequestException("Користувач не записаний у цю чергу");
    }

    dynamicQueue.appointments = dynamicQueue.appointments.filter(
      appointment => appointment.toString() !== userId.toString()
    );

    return await dynamicQueue.save();
  }

        async deleteDynamicQueue(id: Types.ObjectId) {
                const             dynamicQueue             =             await
this.dynamicQueueModel.findByIdAndDelete(id);
                if (!dynamicQueue) {
                        throw  new  InternalServerErrorException("Dynamic  queue
not found");
                }
                return dynamicQueue;
        }
}
```

Файл ExecutorDynamicQueue.tsx
Реалізація функціональної задачі «Черги»

```
"use client";

import { CardBody, CardFooter, CardHeader } from "@heroui/card";
import {
  Button,
  Divider,
  Input,
  Modal,
  ModalBody,
  ModalContent,
  ModalFooter,
  ModalHeader,
  Spinner,
  Tooltip,
  useDisclosure
} from "@heroui/react";
import { dynamicQueueService } from "@repo/api/services/dynamic-queue/dynamic-
queue.service";
import UiNoContent from "@repo/ui/components/uiNoContent";
import { ListOrdered, Plus, X } from "lucide-react";
import { useParams } from "next/navigation";
import { useEffect, useState } from "react";

import { useAddToQueueMutation } from "./hooks/useAddToQueueMutation";
import         {         useRemoveUserFromQueueMutation         }         from
"./hooks/useRemoveUserFromQueueMutation";
import { useDeleteDynamicQueueMutation } from "./hooks/useDeleteDynamicQueue";

export default function ExecutorDynamicQueue() {
  const [data, setData] = useState<any>(null);
  const [loading, setLoading] = useState<boolean>(true);
  const [userEmail, setUserEmail] = useState<string>("");

  const { isOpen, onOpenChange, onOpen, onClose } = useDisclosure();

  const { id } = useParams();

  useEffect(() => {
```

```
  let isMounted = true;

  async function fetchData(id: string) {
    try {
      const rawData = await dynamicQueueService.getAsExecutorById(id);

      rawData.appointments.reverse();

      if (isMounted) {
        setData(rawData);
        setLoading(false);
      }
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
    } catch (e) {
      if (isMounted) setLoading(false);
    }
  }

  if (id) {
    fetchData(id as string);
  }

  return () => {
    isMounted = false;
  };
}, [id]);

function formatTime(time: string): string {
  const [h, m] = time.split(":").map(part => part.padStart(2, "0"));

  return `${h}:${m}`;
}

function getTimeUntilEvent(time: string): string {
  const [h, m] = time.split(":").map(Number);
  const now = new Date();

  let eventTime = new Date();

  eventTime.setHours(h, m, 0, 0);
  if (eventTime <= now) {
    eventTime.setDate(eventTime.getDate() + 1);
  }

  let targetTime = new Date(eventTime.getTime() - 60 * 60 * 1000);

  if (targetTime <= now) {
    eventTime.setDate(eventTime.getDate() + 1);
    targetTime = new Date(eventTime.getTime() - 60 * 60 * 1000);
  }

  const diffMs = targetTime.getTime() - now.getTime();
  const diffMinutes = Math.floor(diffMs / 60000);
  const hours = Math.floor(diffMinutes / 60);
  const minutes = diffMinutes % 60;

  return `${hours} годин, ${minutes} хвилин`;
}

const onSuccessAddCallback = (user: any) => {
  setUserEmail("");
  onClose();
  setData(prevData => {
    if (!prevData) return prevData;
```

```
      return {
        ...prevData,
        appointments: [...prevData.appointments, user]
      };
    });
  };

  const onSuccessDeleteCallback = (user: any) => {
    setData(prevData => {
      if (!prevData) return prevData;

      return {
        ...prevData,
        appointments: prevData.appointments.filter(
          (appointment: { _id: any }) => appointment._id !== user._id
        )
      };
    });
    onClose();
  };

  const { mutate, isPending } = useAddToQueueMutation(onSuccessAddCallback);
  const { mutate: deleteMutate } = useRemoveUserFromQueueMutation(
    onSuccessDeleteCallback
  );
  const { mutate: deleteDynamicQueue } = useDeleteDynamicQueueMutation();

  if (loading) {
    return (
      <div className="w-full h-[400px] flex justify-center items-center">
        <Spinner size="lg" />
      </div>
    );
  }

  if (!data) {
    return (
      <CardBody>
        <UiNoContent text="Не знайдено чергу за цим ідентифікатором, можливо
вона була видалена" />
      </CardBody>
    );
  }

  return (
    <>
      <CardHeader className="flex justify-between items-center">
        <div className="flex items-center gap-2">
          <ListOrdered size={25} />
          <h1 className="text-md">
            {data.title} / {data.organization_id.organization_title}
          </h1>
        </div>
        <p className="text-primary">id: {data._id}</p>
      </CardHeader>
      <Divider />
      <CardBody className="flex flex-col gap-4">
        <div className="flex gap-4">
          <div className="flex flex-col gap-1">
            <p className="text-default-500">Часу до автоочищення черги</p>
            <p>{getTimeUntilEvent(data.work_start_time)}</p>
          </div>
          <div className="flex flex-col gap-1">
```

```
            <p className="text-default-500">Початок робочого дня</p>
            <p>{formatTime(data.work_start_time)}</p>
          </div>
          <div className="flex flex-col gap-1">
            <p className="text-default-500">Кінець робочого дня</p>
            {formatTime(data.work_end_time)}
          </div>
        </div>
        <div className="flex flex-col gap-1">
          <p className="text-default-500">Поточна черга</p>
          <div className="flex gap-4 items-center flex-wrap">
            {data.appointments.map((appointment, index) => (
              <div
                key={index}
                className="relative bg-default-100 p-3 rounded-lg"
              >
                <p className="text-sm">
                  {appointment.name} {appointment.surname}
                </p>
                <p              className="text-default-500              text-
xs">{appointment.email}</p>
                <Tooltip content="Видалити клієнта з черги">
                  <Button
                    isIconOnly
                    className="absolute -right-3 -top-2 rounded-full w-6 h-6
min-w-6"
                    color="danger"
                    size="sm"
                    variant="solid"
                    onPress={() =>
                      deleteMutate({
                        id: id as string,
                        userId: appointment._id
                      })
                    }
                  >
                    <X size={16} />
                  </Button>
                </Tooltip>
              </div>
            ))}
            <Button
              color="primary"
              size="lg"
              startContent={<Plus />}
              variant="faded"
              onPress={onOpen}
            >
              Додати клієнта
            </Button>
          </div>
        </div>
        <div className="flex flex-col gap-4 max-w-[500px]">
          <div>
            <p className="text-default-500">Ціна послуги</p>
            <p className="text-primary">
              {data.price ? `₴ ${data.price}` : "Безкоштовно"}
            </p>
          </div>
          <div>
            <p className="text-default-500">Опис послуги</p>
            <p>{data.description}</p>
          </div>
        </div>
```

```
        </CardBody>
        <Divider />
        <CardFooter className="flex justify-end">
          <Button
            color="danger"
            variant="flat"
            onPress={() => {
              deleteDynamicQueue({ id: id as string });
            }}
          >
            Видалити чергу
          </Button>
        </CardFooter>

        <Modal
          isOpen={isOpen}
          onOpenChange={onOpenChange}
        >
          <ModalContent>
            {onClose => {
              return (
                <>
                  <ModalHeader className="text-primary">
                    <div className="flex items-center gap-2">
                      <Plus size={20} />
                      <h2>Додати клієнта до черги</h2>
                    </div>
                  </ModalHeader>
                  <ModalBody>
                    <Input
                      label="Email клієнта"
                      type="email"
                      validate={value => {
                        if (!value) {
                          return "Будь ласка, введіть email клієнта";
                        }
                        if (!/\S+@\S+\.\S+/.test(value)) {
                          return "Будь ласка, введіть коректний email";
                        }

                        return true;
                      }}
                      value={userEmail}
                      onChange={e => setUserEmail(e.target.value)}
                    />
                    <p className="text-default-500 text-sm">
                      Клієнт має бути зареєстрованим на платформі, щоб додати
його
                      до черги. У разі успішного додавання, клієнта буде додано
на
                      останню позицію черги.
                    </p>
                  </ModalBody>
                  <ModalFooter>
                    <Button
                      color="primary"
                      isDisabled={!userEmail}
                      isLoading={isPending}
                      startContent={<Plus size={20} />}
                      variant="flat"
                      onPress={() => {
                        mutate({ id: id as string, email: userEmail });
                      }}
                    >
```

```
                    Додати
                  </Button>
                  <Button
                    color="danger"
                    variant="flat"
                    onPress={onClose}
                  >
                    Закрити
                  </Button>
                </ModalFooter>
              </>
            );
          }}
        </ModalContent>
      </Modal>
    </>
  );
}
```

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”
Завідувач кафедри
_____ Едуард ЖАРІКОВ
“___” _____ 2025 р.

ВЕБЗАСТОСУНОК ДЛЯ ОПТИМІЗАЦІЇ ЕЛЕКТРОННИХ ЧЕРГ ТА ПРИЙОМУ

Програма та методика тестування

КПІ.ІП-1328.045440.04.51

“ПОГОДЖЕНО”
Керівник проєкту:
_____ Олександр СТЕЛЬМАХ

Нормоконтроль:
_____ Ірина ВІТКОВСЬКА

Виконавець:
_____ Ігор ПЕТРОВ

Київ – 2025

ЗМІСТ

1 ОБ’ЄКТ ВИПРОБУВАНЬ

Об’єктом випробування є вебзастосунок «E-queue» – система для оптимізації електронних черг і прийому клієнтів. Програмне забезпечення розроблено для роботи у середовищі веб-браузерів (Google Chrome, Mozilla Firefox, Safari, Microsoft Edge). Застосунок реалізовано за архітектурою «клієнт-сервер» із використанням технологій Next.js, NestJS, Docker, що забезпечує надійну взаємодію між фронтендом і бекендом. Під час випробування система перевіряється в умовах реального використання її основних функцій: створення організацій, управління чергами, реєстрація клієнтів, моніторинг черг та обробка заявок адміністрацією.

2 МЕТА ТЕСТУВАННЯ

Метою тестування є наступне:

- перевірка правильності роботи програмного забезпечення відповідно до функціональних вимог;
- перевірка збереження даних;
- перевірка сумісності вебзастосунка з останніми версіями сучасних браузерів (Chrome, Opera, Firefox);
- перевірка сумісності застосунку з різними операційними системами (Windows, MacOS);
- знаходження проблем, помилок і недоліків з метою їх усунення;
- перевірка зручності графічного інтерфейсу;
- перевірка виконання функціональних вимог.

3 МЕТОДИ ТЕСТУВАННЯ

Для тестування програмного забезпечення використовуються такі методи:

- статичне тестування – перевіряється програма разом з усією документацією, яка аналізується на предмет дотримання стандартів програмування;
- функціональне тестування – полягає у перевірці відповідності реальної поведінки програмного забезпечення очікуваній;
- системне тестування – перевіряється усе програмне забезпечення в цілому;
- мануальне тестування – тестування без використання автоматизації, тест-кейси пише особа, що тестує програмне забезпечення;
- тестування «чорної скриньки» – об’єктом тестування тут є функції присутні у програмі. Перевіряється коректність вихідних даних при заданих вхідних;

4 ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ

Тестування виконується мануально Під час проведення тестування будуть використовуватись наступні допоміжні засоби:

- React Profiler;
- Postman;
- Chrome DevTools;
- DeepSource;

Порядок проведення тестування буде наступним:

- проведення тестування для перевірки відповідності функціональних можливостей заявленим вимогам;
- тестування використання інтерфейсу, зокрема логічності навігації, доступності ключових функцій та загального користувацького досвіду;
- тестування коректності рендеру у відповідності до очікувань;
- тестування запитів у відокремленому від клієнта середовищі з коректними і некоректними вхідними даними. Перевірка обробки помилок сервером;
- тестування коректності роботи форм з валідними і не валідними даними;
- тестування зручності використання;

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”
Завідувач кафедри
_____ Едуард ЖАРІКОВ
“___” _____ 2025 р.

ВЕБЗАСТОСУНОК ДЛЯ ОПТИМІЗАЦІЇ ЕЛЕКТРОННИХ ЧЕРГ ТА ПРИЙОМУ

Керівництво користувача

КПІ.ІП-1328.045440.05.34

“ПОГОДЖЕНО”
Керівник проєкту:
_____ Олександр СТЕЛЬМАХ

Нормоконтроль:
_____ Ірина ВІТКОВСЬКА

Виконавець:
_____ Ігор ПЕТРОВ

Київ – 2025

ЗМІСТ

1 ПРИЗНАЧЕННЯ ПРОГРАМИ

«E-queue» – це вебзастосунок для оптимізації електронних черг та управління прийомом клієнтів. Функціронал зосереджений на віртуалізації живих черг і черг з попереднім записом шляхом забезпечення управління електронними організаціями і їх штатом, обома видами черг для організацій, наданні детального опису та форм реєстрації у черги для клієнтів.

Кодова база організована як монорепозиторій під керуванням Turborepo. У корені репозиторію розташовано два основні застосунки — клієнтську частину на Next.js і серверну на NestJS, а також окремі пакети для спільного функціоналу (типи TypeScript, утиліти для API, загальні константи та UI-компоненти), що забезпечує повторне використання коду й єдиний стиль у всіх модулях .

Після створення акаунту чи входу користувач отримує доступ до повного функціоналу облікового запису: він може вказати й змінити особисті дані (ім’я, прізвище, електронну пошту), безпечно вийти з сесії та повернутися знову авторизованим у будь-який момент.

Модуль заявок на створення організації дозволяє користувачу подати детальну заявку із назвою, описом і логотипом установи, після чого адміністратор системи може схвалити запит, автоматично створивши організацію, або відхилити його з коментарем. Крім того, інтегрований чат забезпечує обмін повідомленнями між користувачем і адміном прямо в інтерфейсі.

Після того як організація сформована, користувач із роллю власника або адміністратора отримує інструменти для керування нею: він може редагувати її профіль (дані про назву, контактні дані, опис), запрошувати нових співробітників через email-запити та при необхідності вилучати членів команди.

Система черг реалізована як набір статичних і «живих» черг: у статичних чергах клієнти самостійно записуються через пошук за унікальним ідентифікатором і обирають часовий слот, сформований на підставі графіка

роботи та перерв, тоді як у «живих» чергах виконавці додають клієнтів вручну. Крім створення та видалення черг, доступне редагування їх параметрів і перегляд поточного стану всіх черг у межах однієї організації.

2 ПІДГОТОВКА ДО РОБОТИ З ПРОГРАМНИМ ЗАБЕЗПЕЧЕННЯМ

2.1 Системні вимоги для коректної роботи

Мінімальна конфігурація технічних засобів:

– тип процесору: Intel Core i3;
– об‘єм ОЗП: 4 Гб;
– підключення до мережі Інтернет зі швидкістю від 5 мегабіт;
– внутрішня пам’ять: не менше 100 МБ вільної пам’яті;

Рекомендована конфігурація технічних засобів:

– тип процесору: Intel Core i5;
– об‘єм ОЗП: 16 Гб;
– підключення до мережі Інтернет зі швидкістю від 100 мегабіт;
– внутрішня пам’ять: не менше 100 МБ вільної пам’яті;

2.2 Завантаження застосунку

На даний момент застосунок можна відкрити локально. Для цього спершу необхідно завантажити репозиторій на пристрій, з якого буде виконуватись запуск. В корені репозиторію необхідно прописати команди: pnpm install та pnpm run dev в консолі в корені проекту. Це ініціалізує запуск застосунку.

2.3 Перевірка коректної роботи

По завершенню встановлення додатка у консолі повинне відобразитись посилання на локально розгорнутий сервер. Після натискання на посилання повинна відобразитись початкова сторінка застосунку у браузері.

3 ВИКОНАННЯ ПРОГРАМИ

При першому відкритті застосунку користувач бачить посадкову сторінку, на якій може переглянути короткий опис функціоналу і призначення застосунку (рисунок 3.1). Також звідси можна одразу перейти на сторіну авторизації і реєстрації.

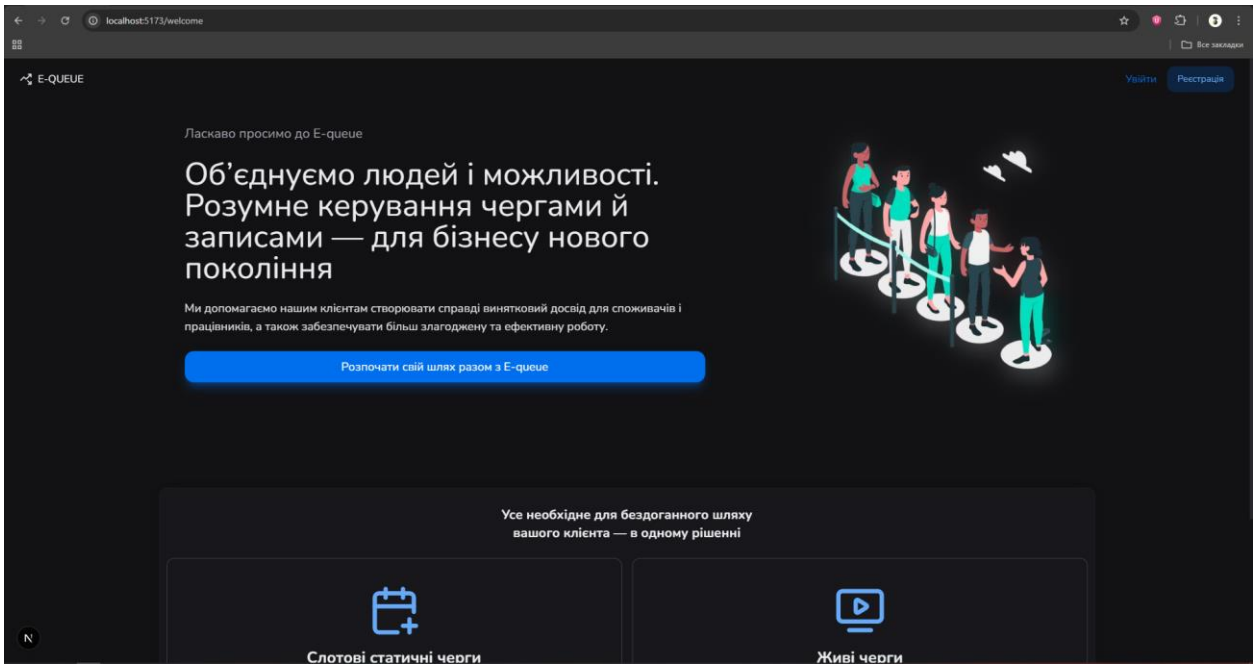


Рисунок 3.1 – Сторінка створення паролю

Для подальшої взаємодії необхідно зареєструватись, для цього є кнопка реєстрації у навігаційній панелі. При натисканні на неї користувача перекидує на сторінку реєстрації. В формі на сторінці необхідно ввести: ім'я, прізвище, електронну пошту, пароль та підтвердження паролю. Підтверджуємо реєстрацію (рисунок 3.2).

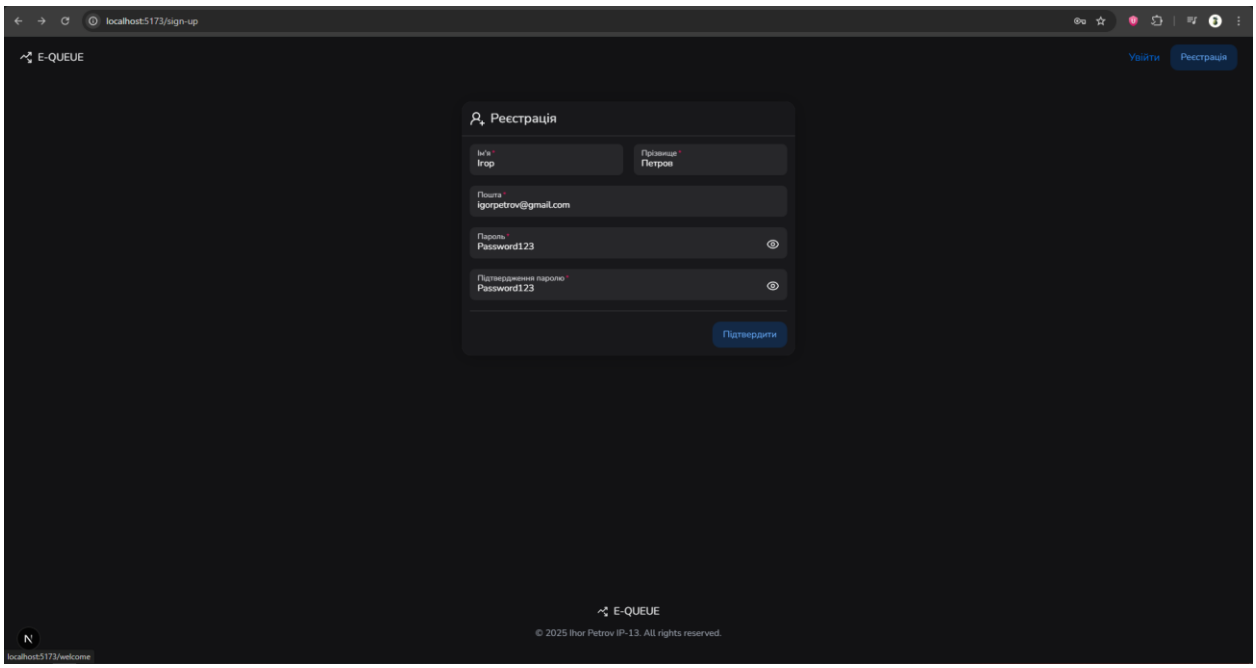


Рисунок 3.2 – Сторінка реєстрації

Якщо акаунт вже існує, то передбачена можливість авторизації, для цього необхідно натиснути кнопку авторизації у навігаційній панелі. При натисканні на неї перекидує на сторінку авторизації. В формі на сторінці необхідно ввести: електронну пошту та. Підтверджуємо авторизацію (рисунок 3.3).

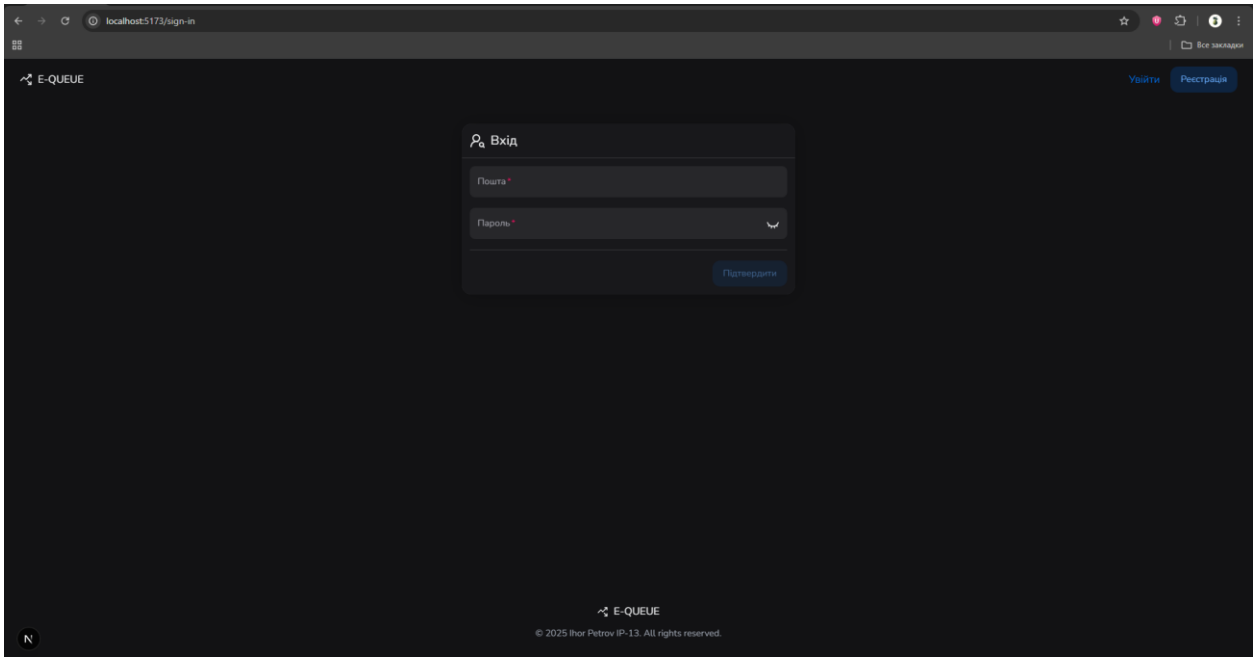


Рисунок 3.3 – Сторінка авторизації

Після входу на платформу можна побачити дашборд, у якому можна швидко переглянути останні записи в черги та найближчих клієнтів (за наявності) (рисунок 3.4).

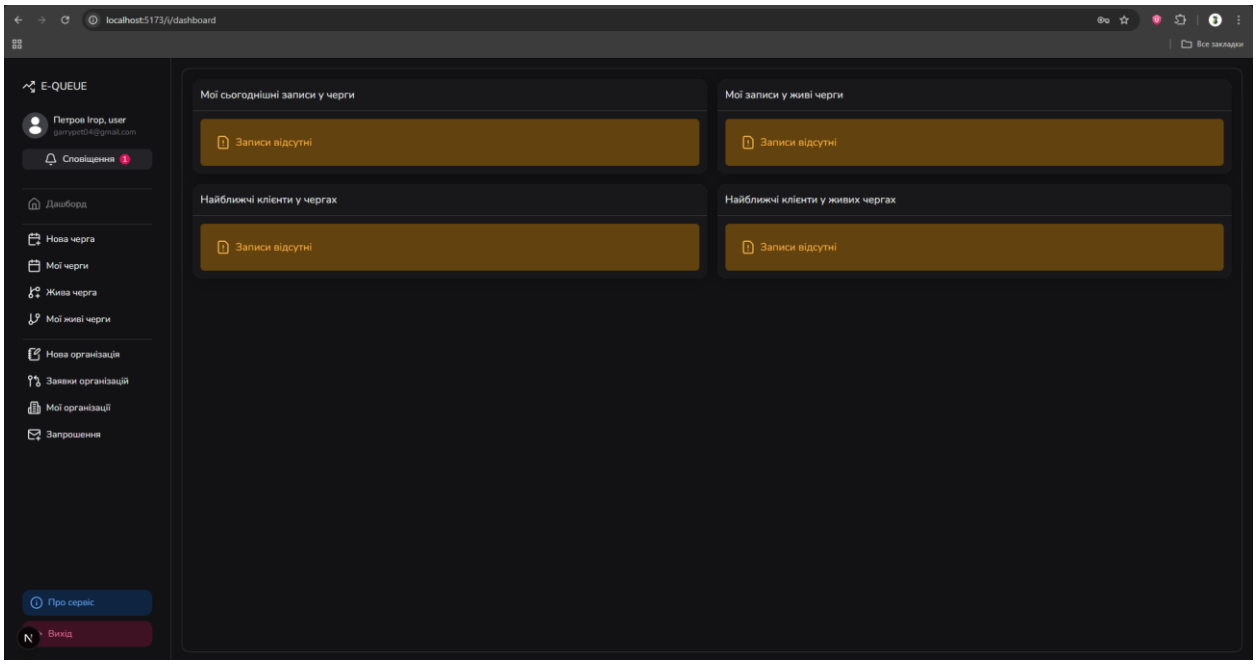


Рисунок 3.4 – Дашборд

Після входу на платформу можна створити організацію. Для необхідно зайти на сторінку створення організації. На самій сторінці знаходиться велика форма, де вказуються дані заявки на організацію. Частина цих даних далі буде використана для створення самої організації, зокрема назва, опис, логотип (рисунок 3.5).

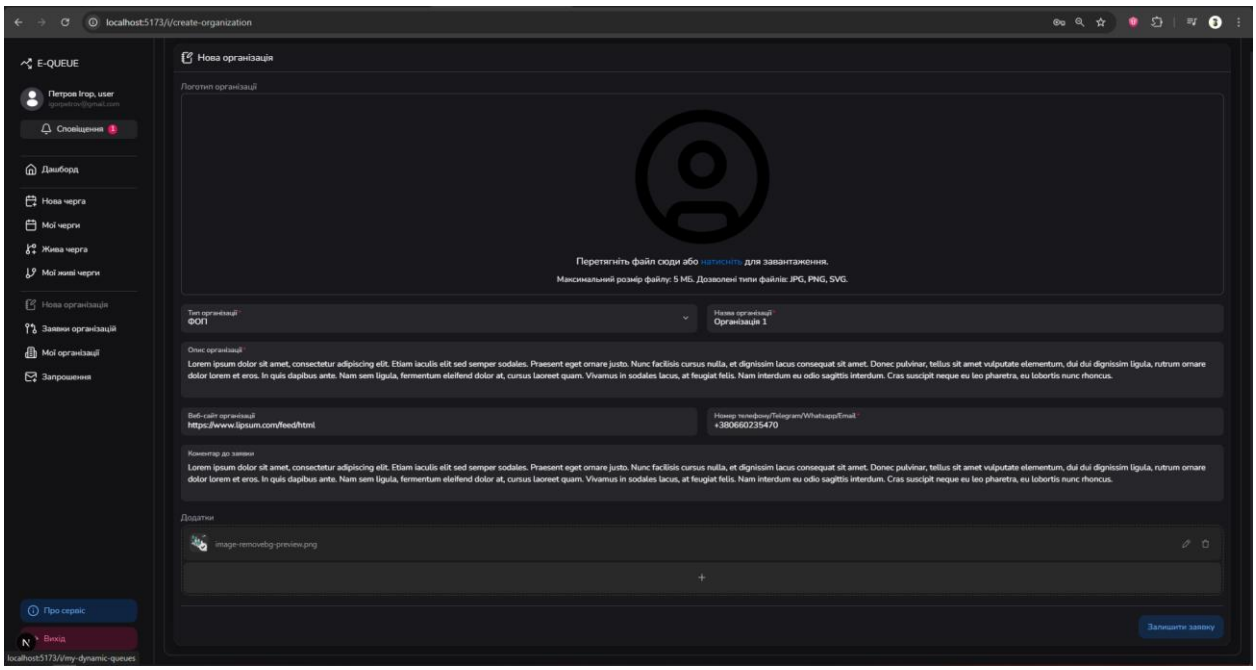


Рисунок 3.5 – Форма створення заявки на організацію

Після підтвердження форми, користувача перенаправляє на сторінку його поточних заявок на організацію. Тут він може побачити її поточний статус, дату створення (рисунок 3.6).

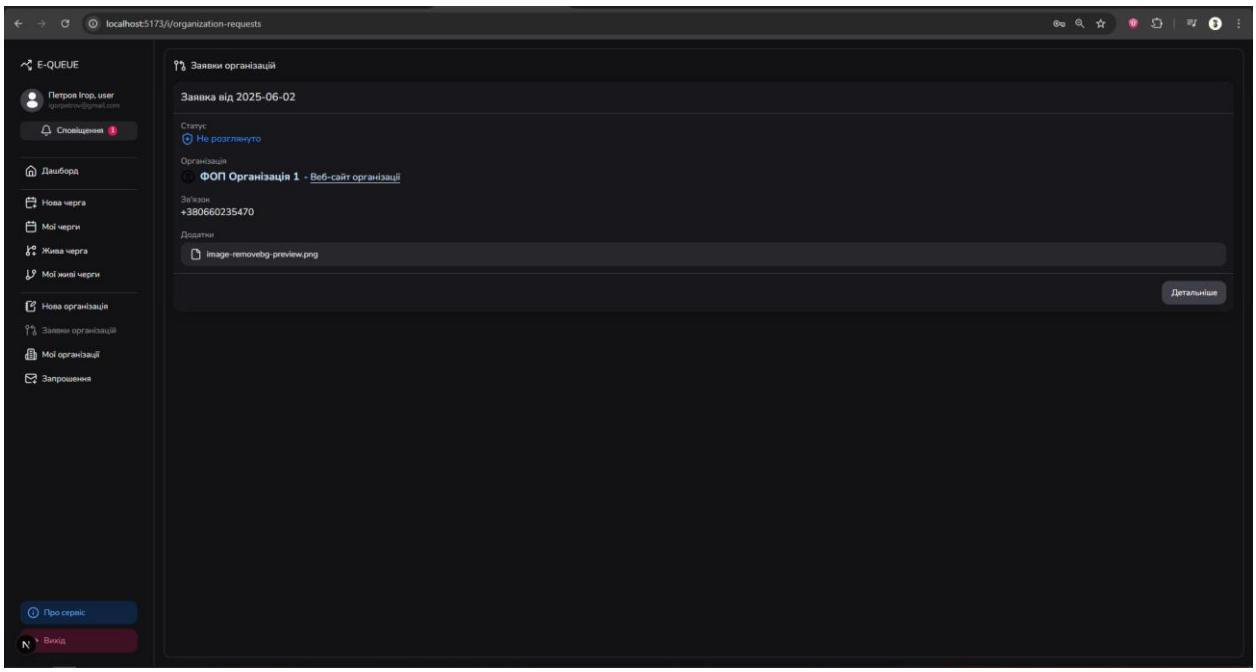


Рисунок 3.6 – Сторінка заявок користувача

При натисканні кнопки «Детальніше», можна відкрити деталі заявки, де ще раз переглянути заповнені при створенні дані (рисунок 3.7).

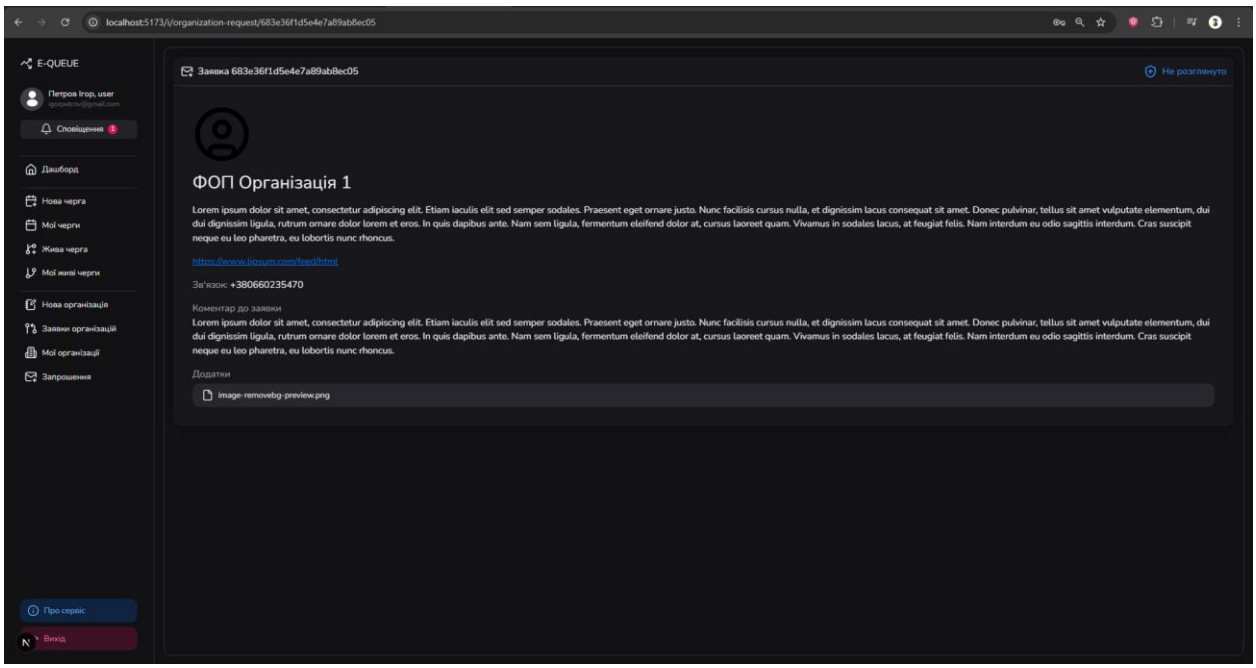


Рисунок 3.7 – Детальна сторінка заявки користувача

Коли адміністратор отримав всі необхідні деталі та поспілкувався з користувачем, він може схвалити заявку. Після схвалення заявки система створює нову організацію за даними, які користувач ввів при створенні заявки. Статус заявки змінюється, чат з адміністрацією блокується, користувач може побачити коментар до схвалення (рисунок 3.8).

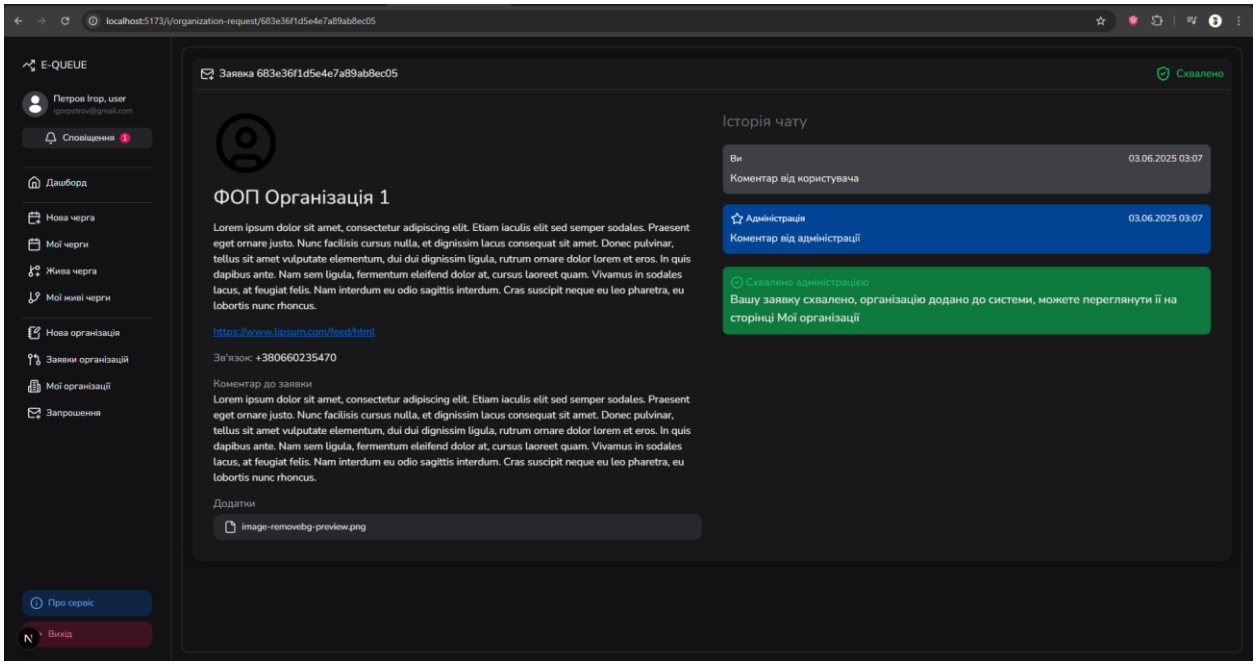


Рисунок 3.8 – Схвалена адміністратором заявка

Тепер користувач може побачити свою новостворену організацію на сторінці організацій з короткою інформацією про організацію, поточну кількість створених черг і співробітників (рисунок 3.9).

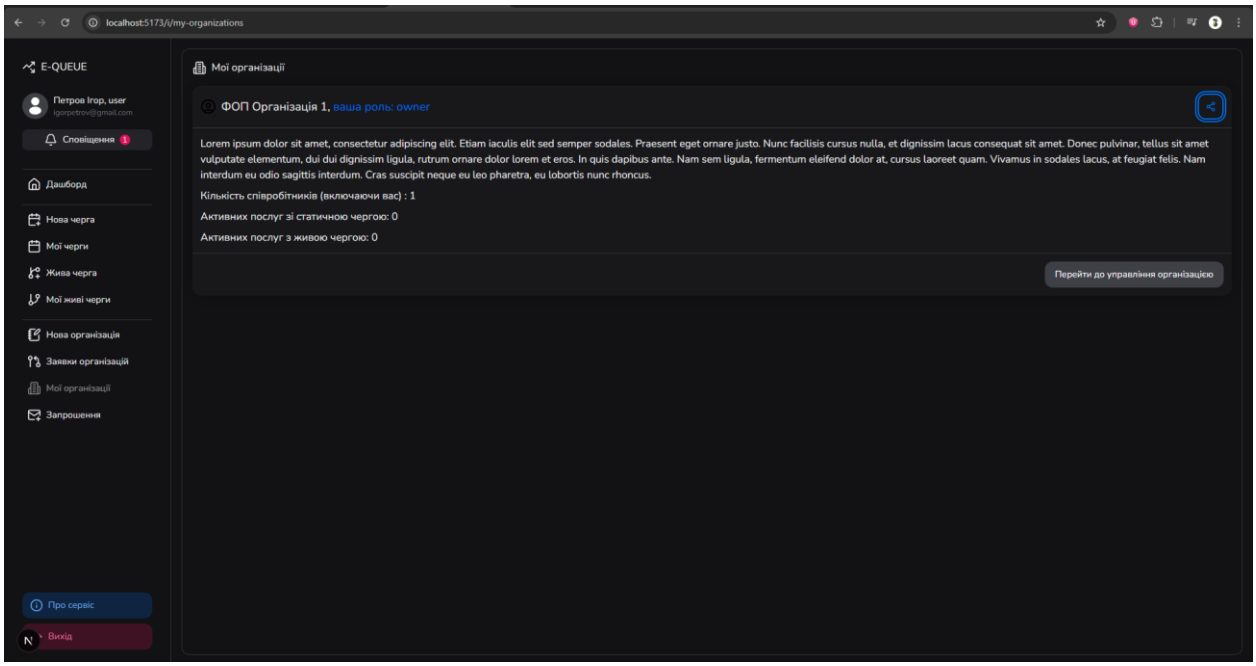


Рисунок 3.9 – Організації користувача

Тепер користувач має можливість створити першу статичну чергу для обслуговування клієнтів. Для цього користувач має зайти на сторінку створення черг. Перш за все користувач обирає організацію, у якій буде створено чергу, також користувач має вибрати дні, у які буде проходити послуга та заповнити усі поля, що відмічені червоною зірочкою. За потреби можна додати файли в додатки, зразки необхідних до заповнення перед прийомом форм та зразки вже заповнених форм для більшого ознайомлення кінцевого клієнта з послугою (рисунок 3.10).

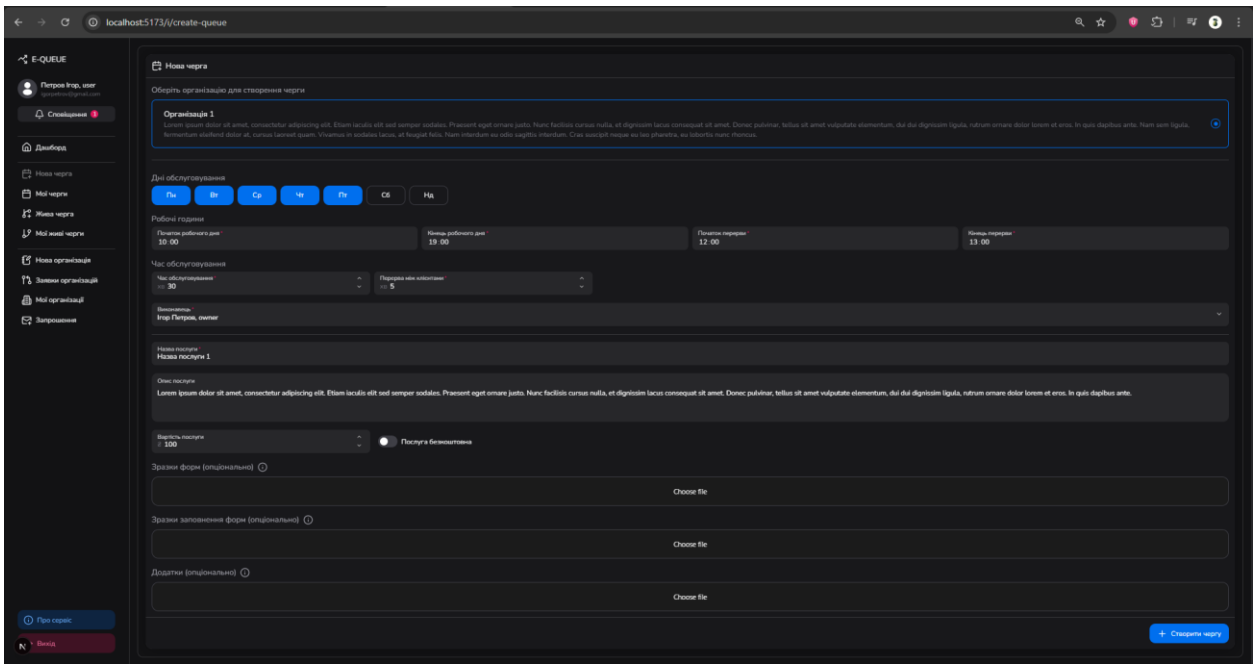


Рисунок 3.10 – Сторінка створення нової статичної черги

Якщо користувач вказав себе у якості виконавця, то новостворена черга збуде відображатись на сторінці черг користувача, де він є виконавцем, у вигляді картки з коротким описом (рисунок 3.11).

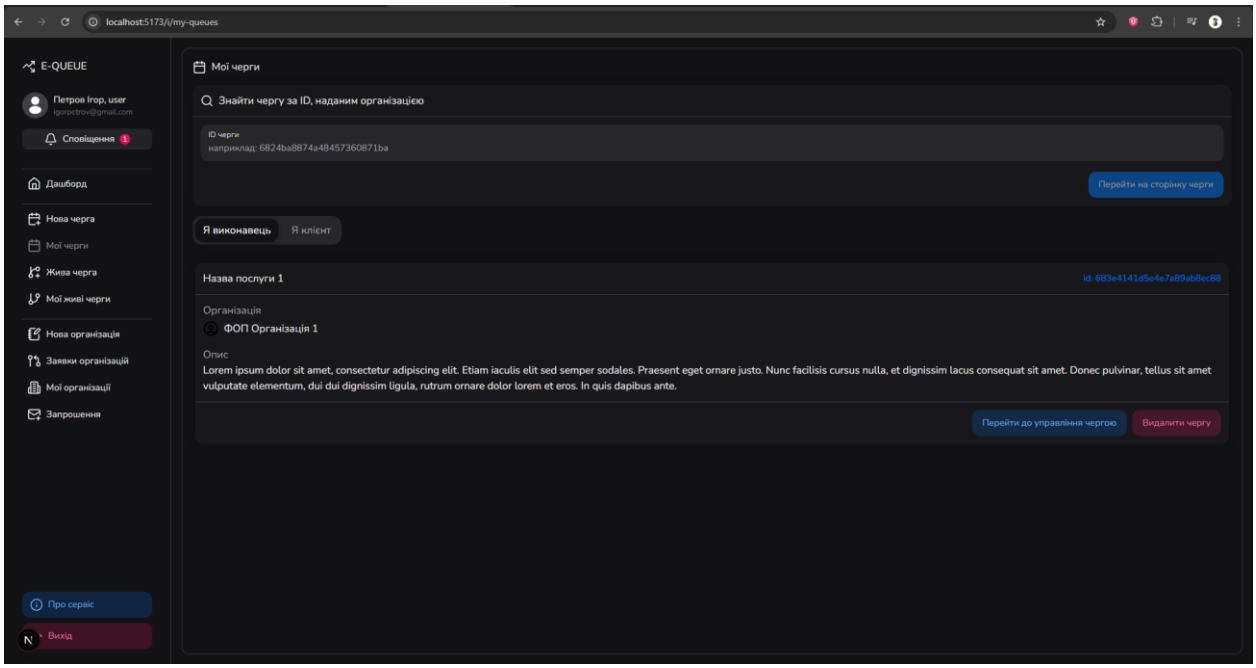


Рисунок 3.11 – Сторінка статичних черг користувача

Для залучення клієнтів організація може поділитись ідентифікатором черги зручним для них способом поза платформою. Тепер інший користувач, клієнт організації може знайти чергу за ідентифікатором, наданим

організацією. Для цього він має ввести ідентифікатор у пошук за ідентифікатором та натиснути кнопку підтвердження (рисунок 3.12).

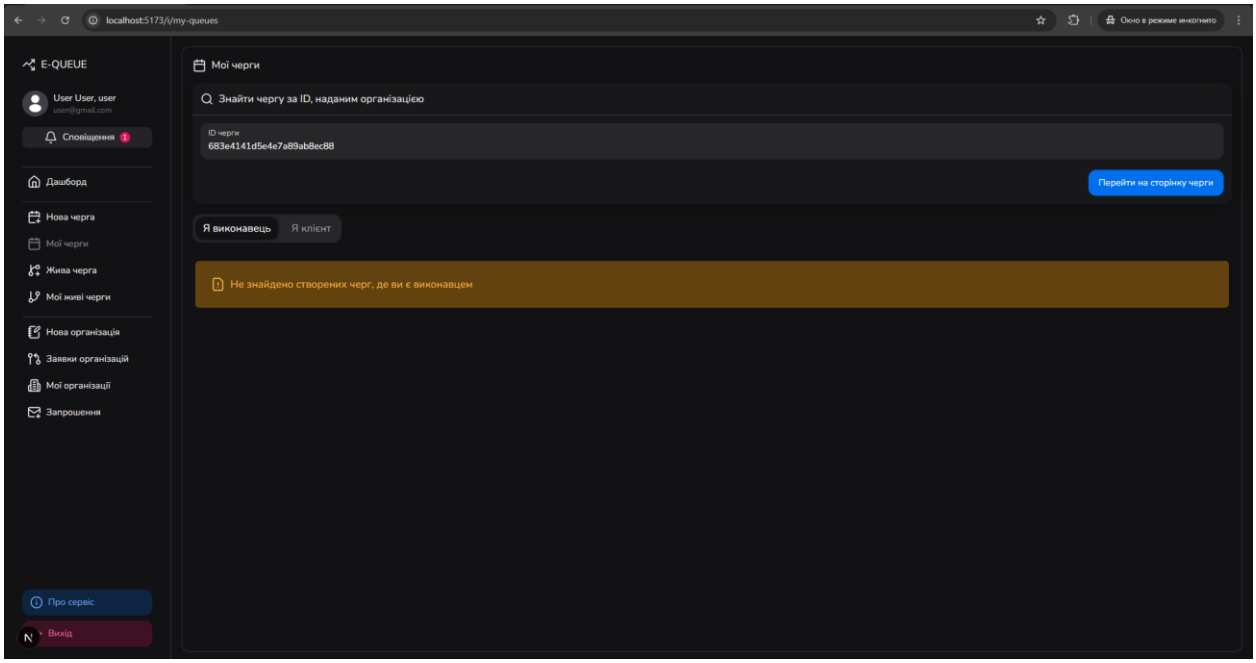


Рисунок 3.12 – Пошук черги за ідентифікатором

Якщо ідентифікатор коректний, то користувача перекине на сторінку черги, де він може обрати підходящі для нього дату для нього тайм-слот. Тайм-слоти формуються на основі часу початку і кінця роботи, а також перерв, вказаних організацією при створенні черги (рисунок 3.13).

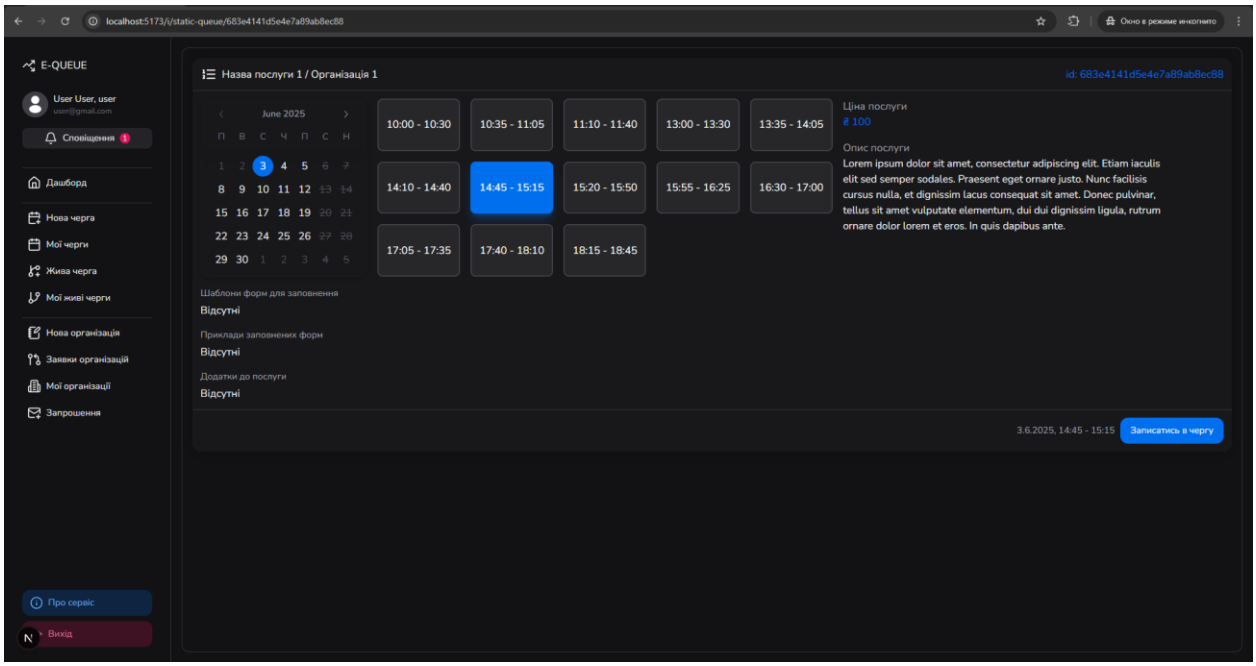


Рисунок 3.13 – Запис клієнта в чергу

Після реєстрації в черзі відповідний слот блокується на запис для інших клієнтів, а користувач бачить запис на сторінці черг, де він є клієнтом (рисунок 3.14).

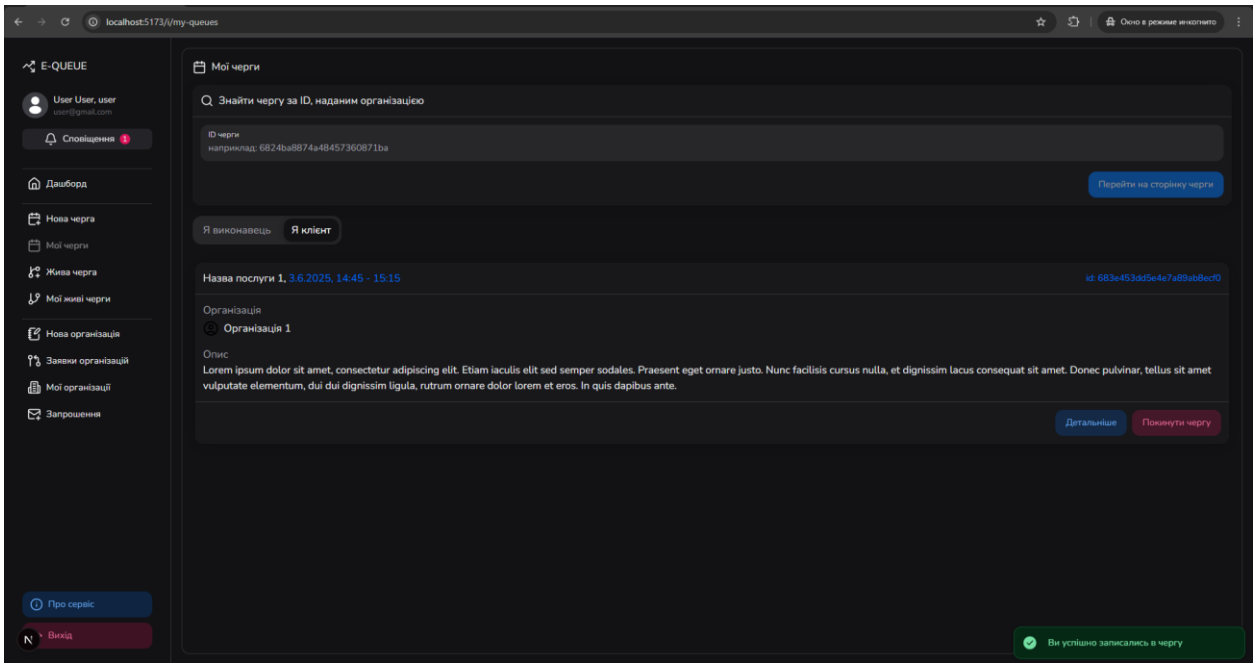


Рисунок 3.14 – Новий запис користувача в чергу

Виконавець також бачить новий запис в чергу на сторінці управління чергою (рисунок 3.15).

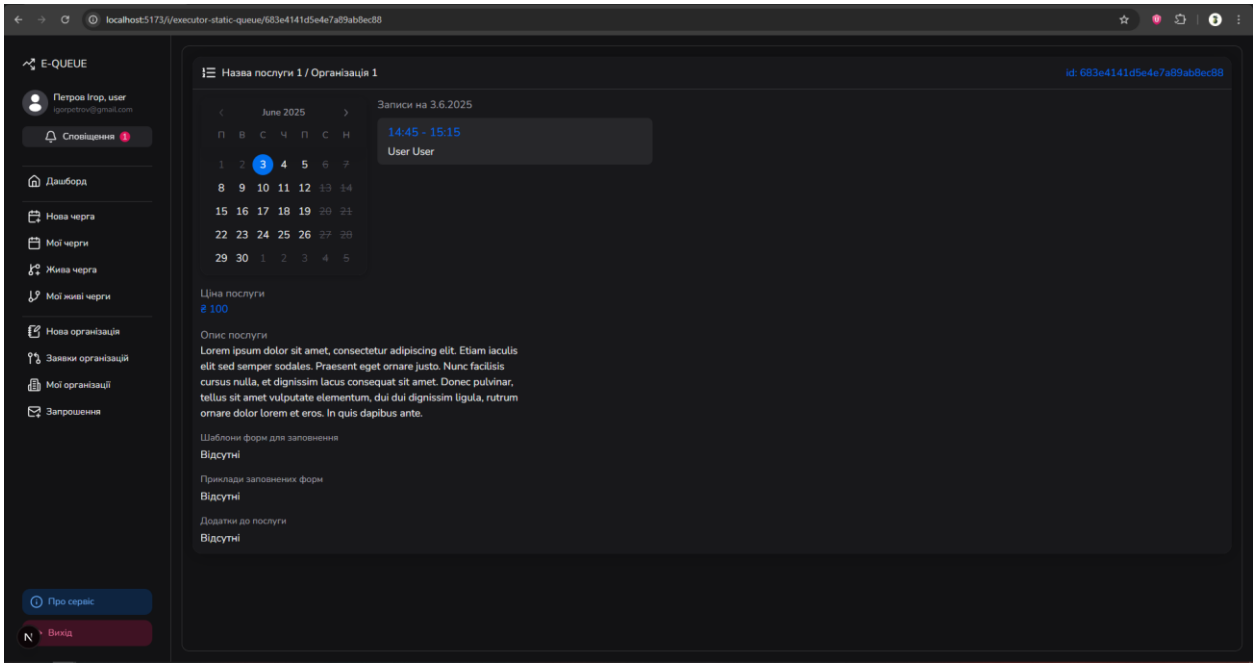


Рисунок 3.15 – Новий запис користувача в чергу на сторінці управління чергою

Окрім статичних черг з тайм-слотами користувачі в організаціях можуть також створювати менш залежні від часу живі черги, призначені для швидкого запису без прямої взаємодії клієнтів з платформою. Для цього потрібно зайти на сторінку створення живих черг. Сама форма схожа на створення статичної черги, окрім файлових додатків і днів обслуговування. Жива черга працює в рамках одного дня і клієнти не готуються до прийому заздалегіть (рисунок 3.16).

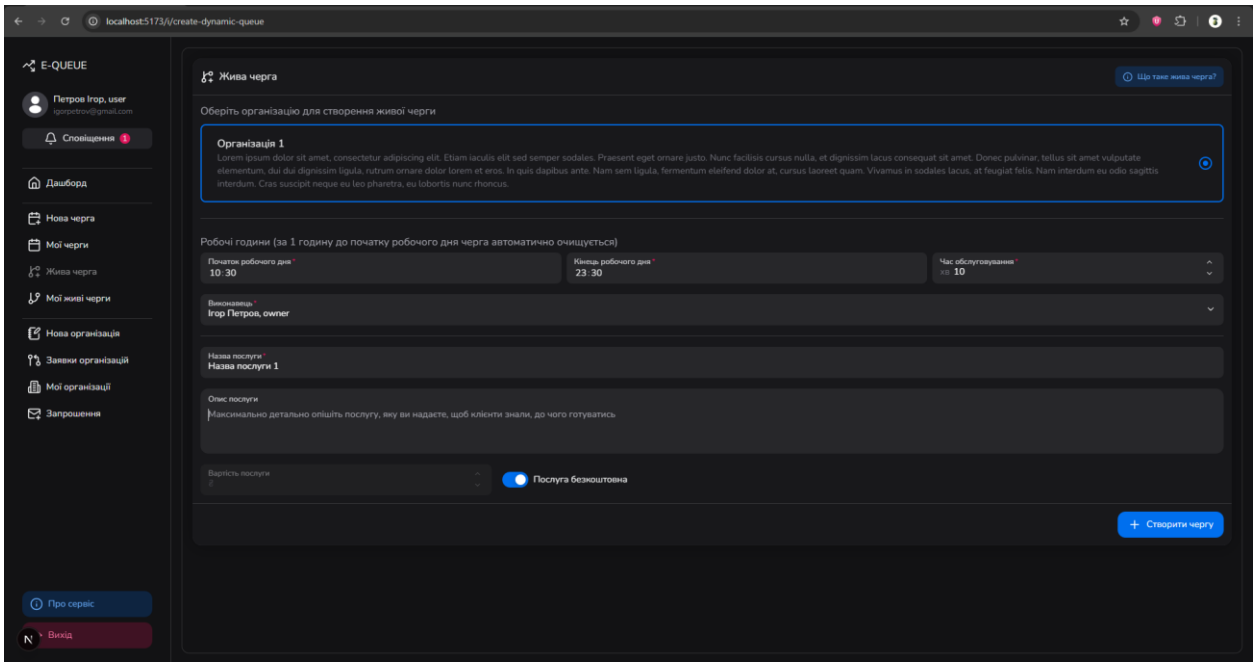


Рисунок 3.16 – Створення живої черги

Після створення живої черги, якщо користувач вказав себе у якості виконавця, він може побачити картку нової черги на сторінці живих черг, де він вказаний, як виконавець (рисунок 3.17).

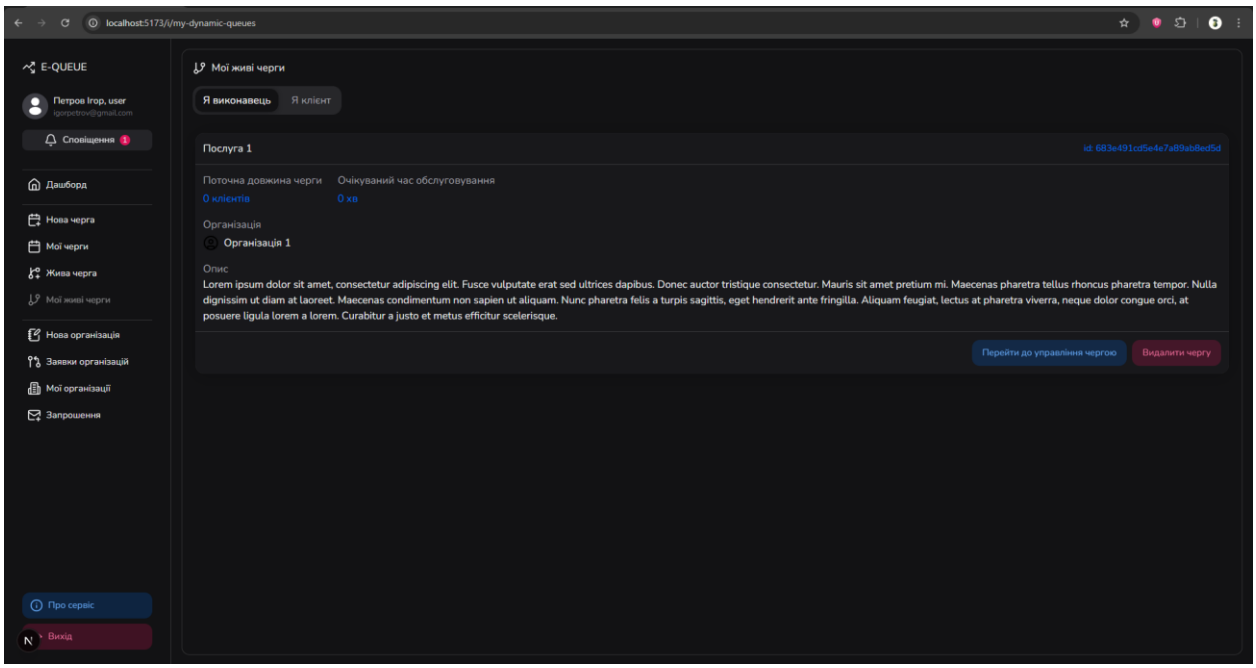


Рисунок 3.17 – Сторінка живих черг, де користувач вказаний, як виконавець

Для додавання клієнтів, користувач має перейти до управління чергою, натиснути кнопку «Додати клієнта» і ввести пошту клієнта у відповідному полі вводу у попапі та підтвердити реєстрацію клієнта. Клієнта буде додано на останню позицію в черзі і він буде відображений (рисунок 3.18).

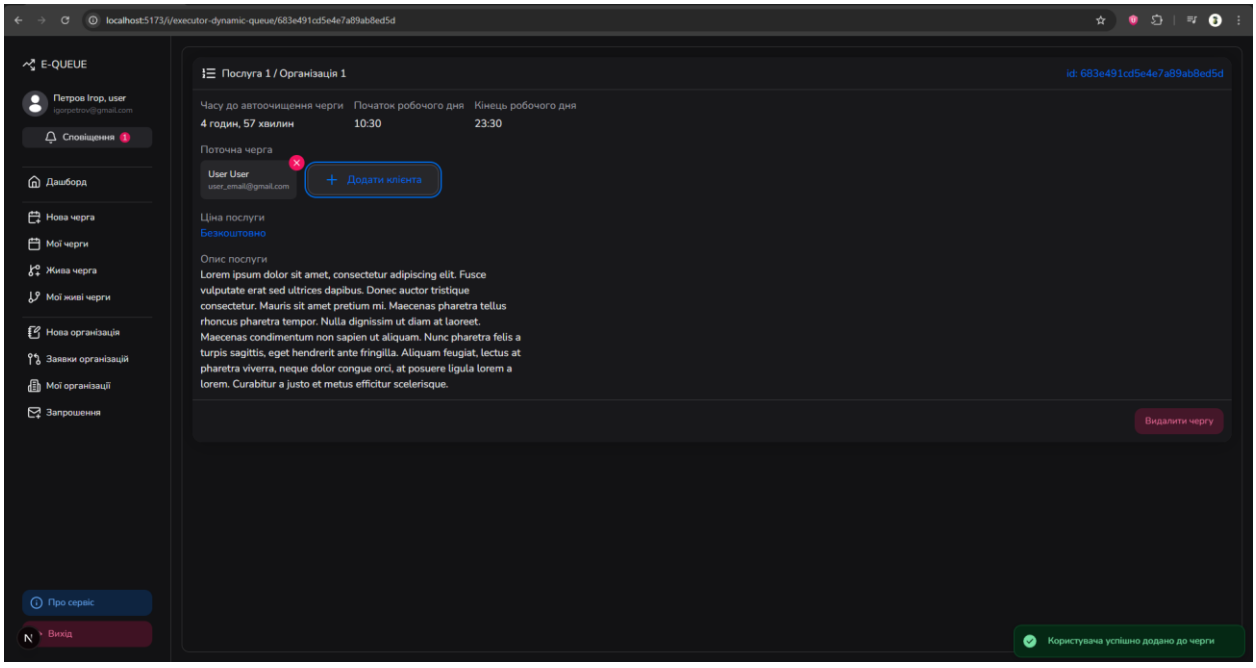


Рисунок 3.18 – Сторінка управління живою чергою

Сам клієнт також бачить цей запис у себе, у вигляді простої картки на сторінці живих черг, де він вказаний, як клієнт. На самій картці клієнт також бачить свою поточну позицію в черзі і короткий опис послуги (рисунок 3.19).

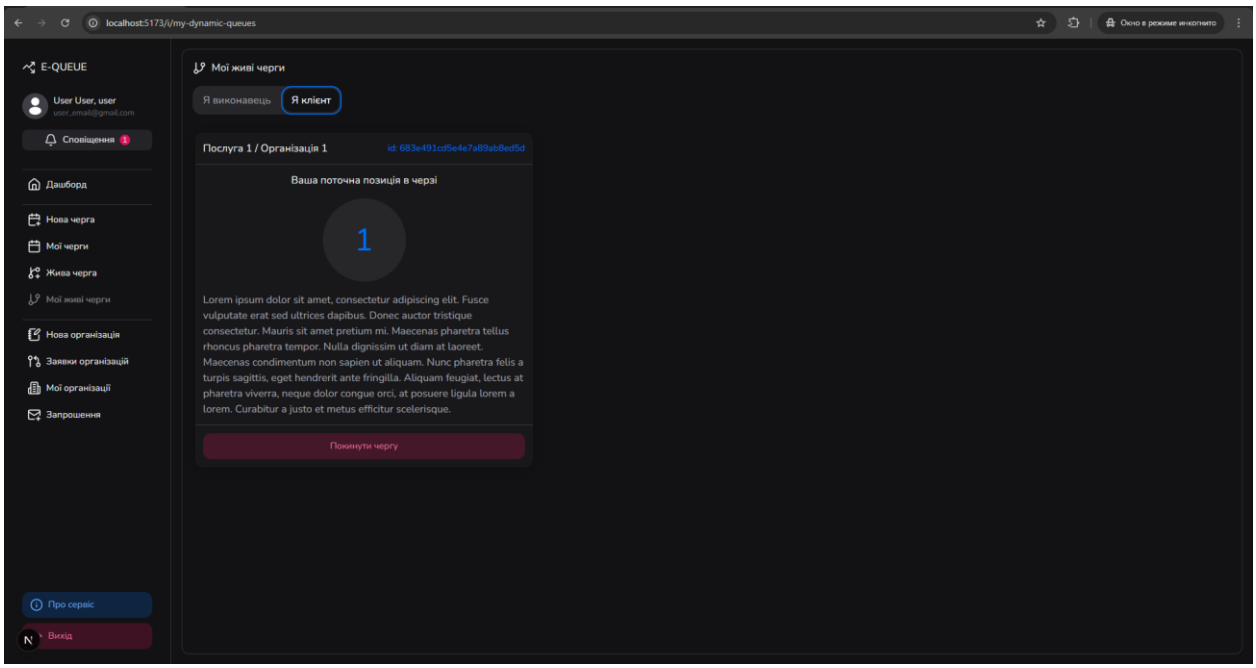


Рисунок 3.19 – Сторінка живих черг, де користувач є клієнтом

За потреби можна відрегувати власний профіль, натиснувши на кнопку профіля у панелі управління (рисунок 3.20).

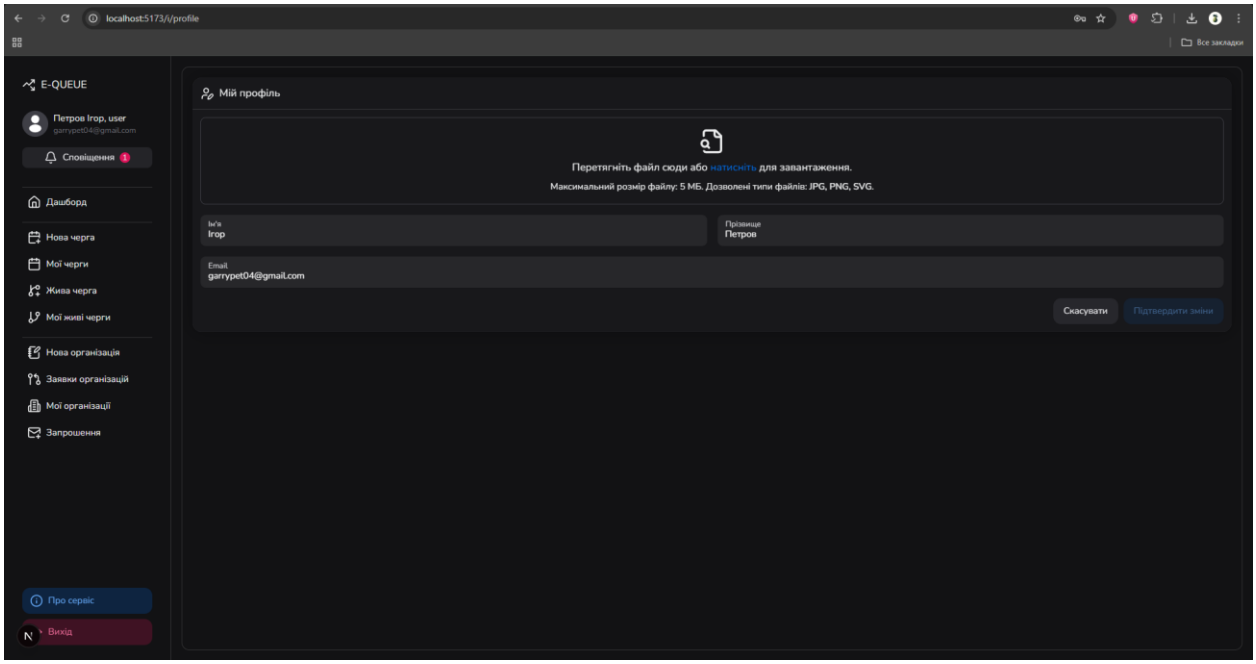


Рисунок 3.20 – Сторінка редагування профіля

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”
Завідувач кафедри
_____ Едуард ЖАРІКОВ
“___” _____ 2025 р.

ВЕБЗАСТОСУНОК ДЛЯ ОПТИМІЗАЦІЇ ЕЛЕКТРОННИХ ЧЕРГ ТА ПРИЙОМУ

Графічний матеріал

КПІ.ІП-1328.045440.06.99

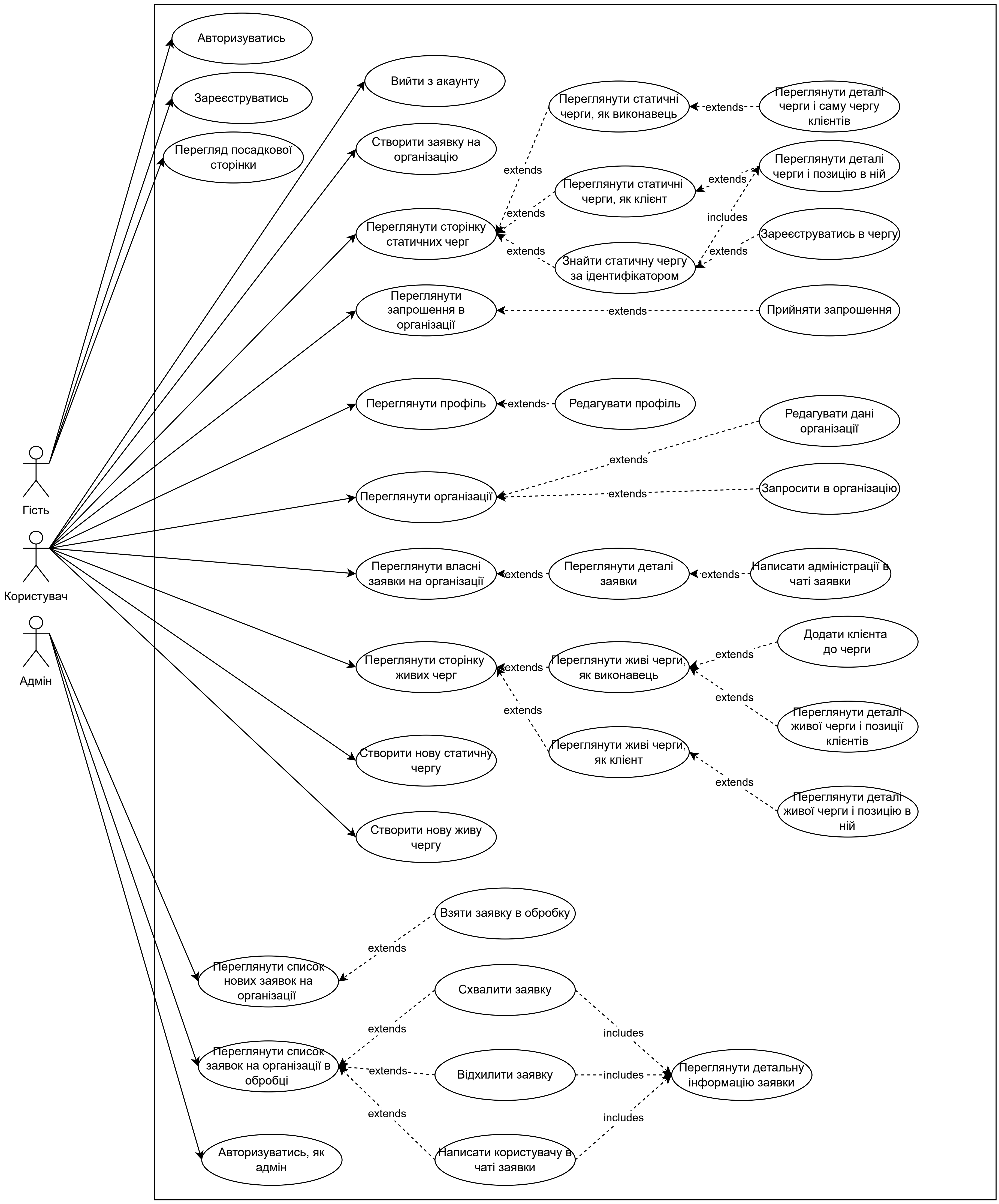
“ПОГОДЖЕНО”
Керівник проєкту:
_____ Олександр СТЕЛЬМАХ

Нормоконтроль:
_____ Ірина ВІТКОВСЬКА

Виконавець:
_____ Ігор ПЕТРОВ

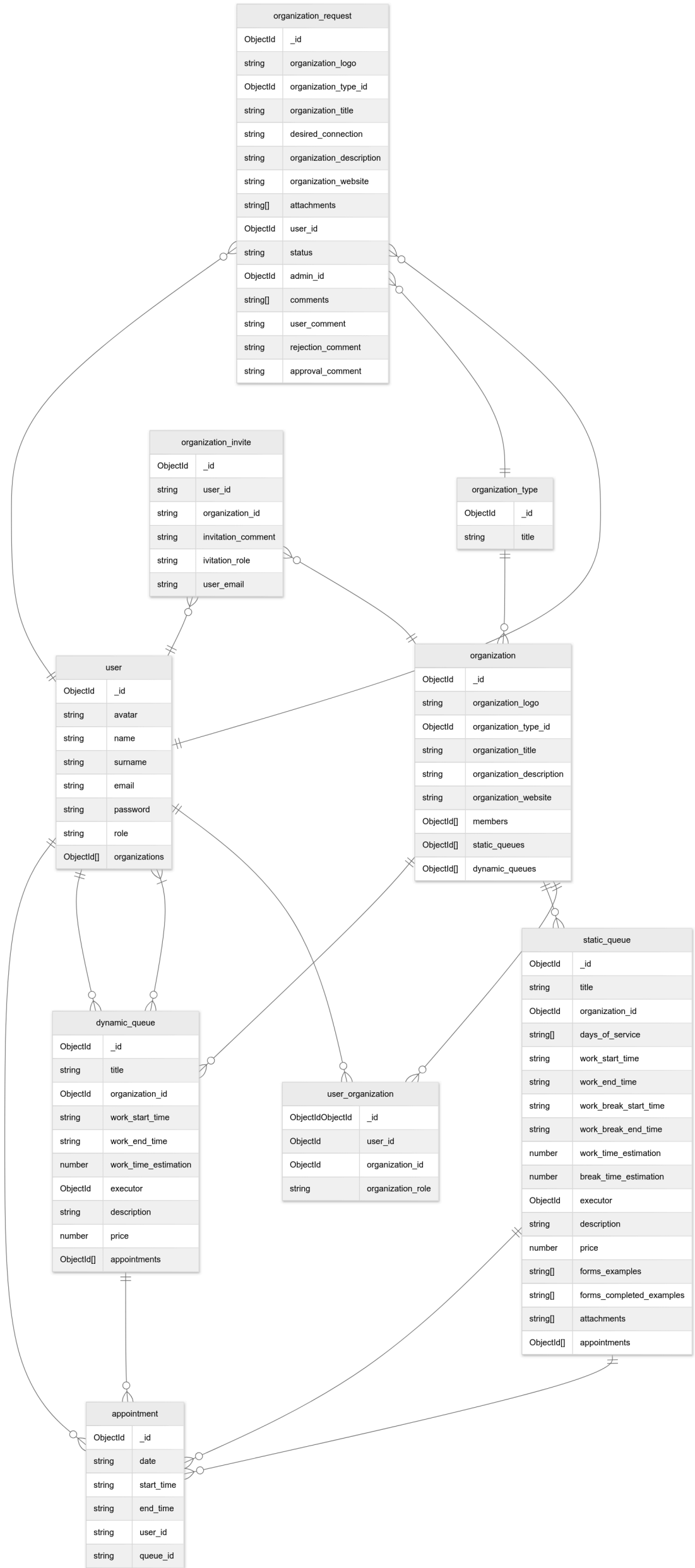
Київ – 2025

КПІ.ІП-1328.045440.06.99.ССВ



					КПІ.ІП-1328.045440.06.99.ССВ			
Зм.	Арк.	№ докум.	Підп.	Дата		Літ.	Маса	Масштаб
Розробив		Петров І.Я.			Схема структурна варіантів використань			
Перевірив		Стельмах О.П.						
Т. контр.						Аркуш 1		Аркушів 1
					Вебзастосунок для оптимізації електронних черг та прийому	КПІ ім.Ігоря Сікорського Кафедра ІПІ гр. ІП-13		
Н. контр.		Вітковська І.І.						
Затвердив		Жаріков Є.В.						

КПІ.ІП-1328.045440.06.99.СБД



Зм.	Арк.	№ докум.	Підп.	Дата
Розробив		Петров І.Я.		
Перевірив		Стельмах О.П.		
Т. контр.				
Н. контр.		Вітковська І.І.		
Затвердив		Жаріков Е.В.		

КПІ.ІП-1328.045440.06.99.СБД

Схема бази даних

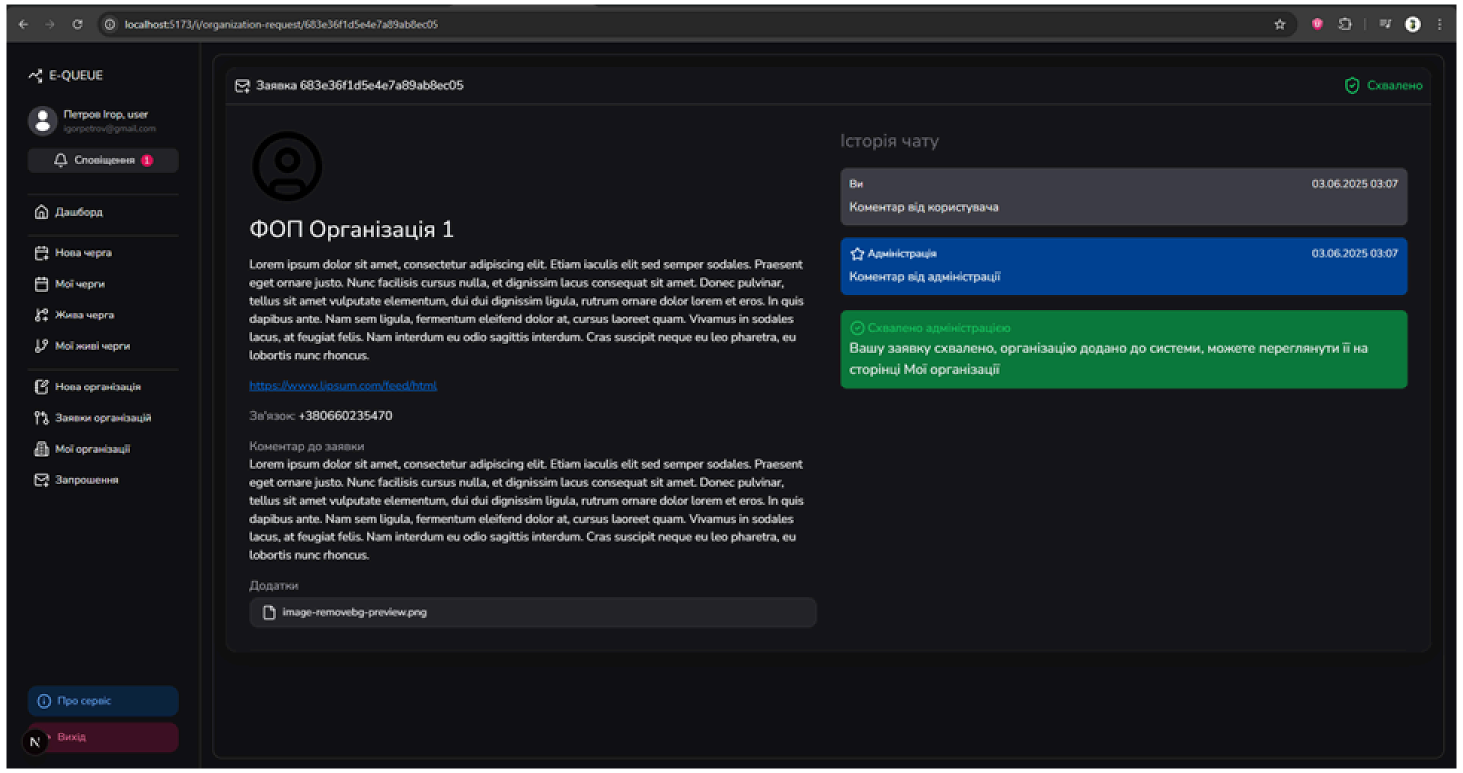
Лит.	Маса	Масштаб

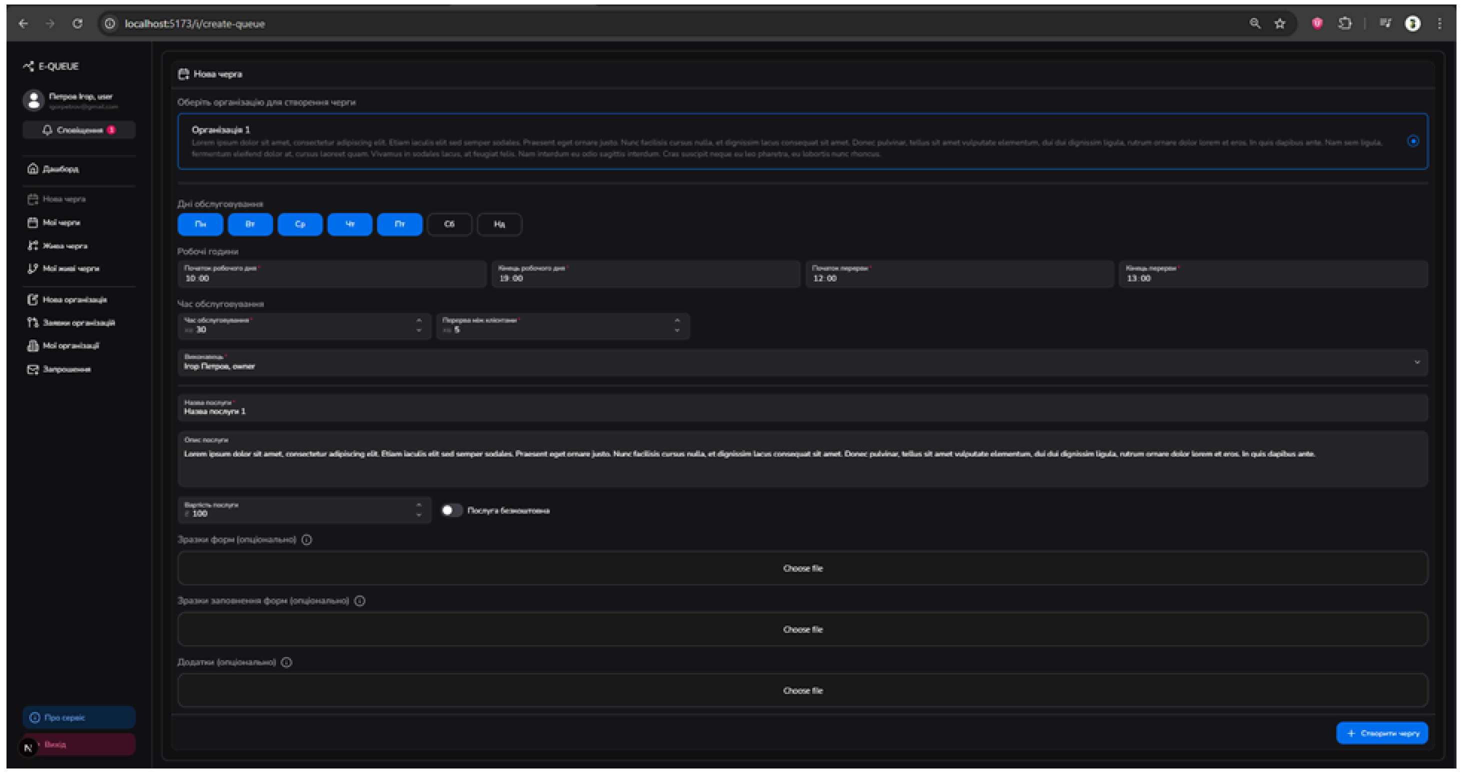
Аркуш 1	Аркушів 1

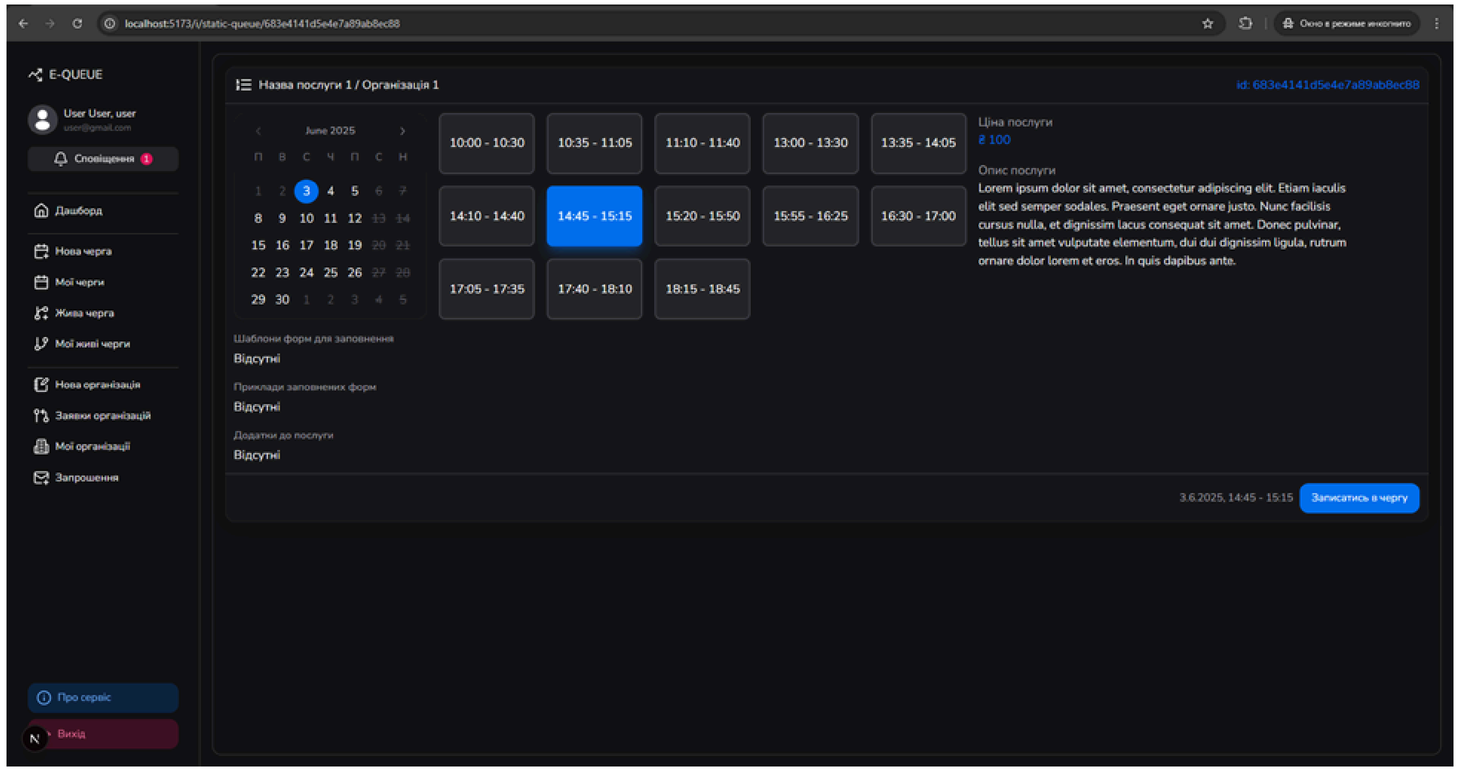
Вебзастосунок для оптимізації електронних черг та прийому

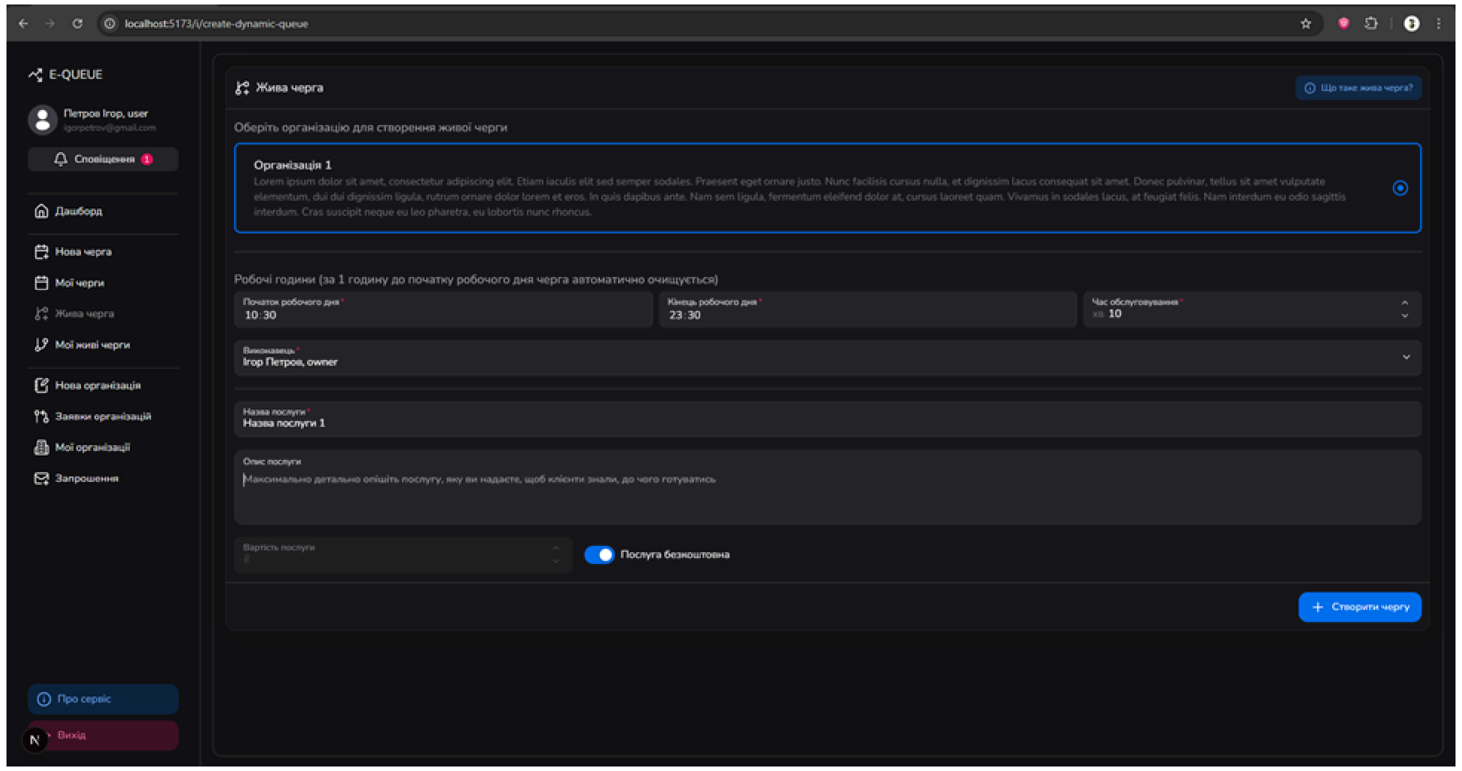
КПІ ім.Ігоря Сікорського
Кафедра ІПІ
гр. ІП-13

КПІ.ІП-1328.045440.06.99.КЕ









					КПІ.ІП-1328.045440.06.99.КЕ			
Зм.	Арк.	№ документа	Підпис	Дата	Креслення вигляду екранних форм	Літера	Маса	Масштаб
Розробив		Петров І.Я.						
Перевірив		Стельмах О.П.						
Т. контр.						Аркуш 1		Аркушів 1
					Вебзастосунок для оптимізації електронних черг та прийому	КПІ ім.Ігоря Сікорського Кафедра ІПІ гр. ІП-13		
Н. контр.		Вітковська І.І.						
Затвердив		Жаріков Е.В.						