Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережевих технологій факультету інформатики

РОЗРОБКА БАГАТОКАНАЛЬНОЇ СИСТЕМИ СПОВІЩЕНЬ ЯК АЛЬТЕРНАТИВИ TELEGRAM ДЛЯ ІНФОРМУВАННЯ СТУДЕНТІВ УНІВЕРСИТЕТУ

Текстова частина до кваліфікаційної роботи
за спеціальністю 121 “Інженерія програмного забезпечення”

Керівник кваліфікаційної роботи
ст. викл. Кобзар О.О.

_____
(підпис)
“___” _____ 2025 р.

Виконав студент Пелович Д.В.

_____
(підпис)
“___” _____ 2025 р.

Київ 2025

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережевих технологій факультету інформатики

ЗАТВЕРДЖУЮ

проф., док. фіз.-мат. наук Малашонок Г. І.

_____

(підпис)

20 вересня 2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

Студенту 4 року БП ІПЗ факультету інформатики

Пелович Дмитру Владиславовичу

Розробити Багатоканальну систему сповіщень як альтернативу Telegram для інформування студентів університету

Зміст ТЧ до кваліфікаційної роботи:

1. Зміст
2. Анотація
3. Вступ
4. Аналіз потреб і вимог користувачів системи
5. Аналіз і порівняння провайдерів
6. Проєктування архітектури системи
7. Реалізація функціональних частин системи
8. Проведення перевірки та випробування системи
9. Висновки
10. Список використаних джерел
11. Додатки

Дата видачі 19 вересня 2024 р. Керівник _____ *(підпис)*

Завдання отримав _____ *(підпис)*

Тема: Багатоканальна система сповіщень як альтернатива Telegram для інформування студентів університету

Календарний план виконання роботи:

№ п/п	Назва етапу роботи	Початок виконання етапу	Кінець виконання етапу	Примітка
1	Отримання завдання на роботу	19.09.2024		
2	Попередній аналіз предмета роботи, формування контексту	21.11.2024	21.11.2024	
3	Огляд домену провайдерів зв’язку	22.11.2024	22.11.2024	
4	Аналіз вимог і потреб користувачів системи	01.12.2024	07.12.2024	
5	Аналіз, порівняння і обрання провайдерів зв’язку для інтеграції	17.01.2025	21.01.2025	
6	Проєктування архітектури системи, обрання технологій	03.02.2025	15.02.2025	
7	Формування плану розробки	16.02.2025	16.02.2025	
8	Реалізація компонентів системи	01.03.2025	12.05.2025	
9	Випробування системи	17.05.2025	17.05.2025	
10	Підготовка доповіді та презентації	19.05.2025	24.05.2025	
11	Захист роботи	02.06.2025		

Студент _____ *(підпис)*

Керівник _____ *(підпис)*

“___” _____ 2025 р.

ЗМІСТ

АНОТАЦІЯ

У роботі описано створення багатоканальної системи сповіщень, розробленої як відповідь на обмежувальні заходи адміністрації університету щодо використання месенджера Telegram у службових та освітніх процесах. Ця система призначена для заміщення попередніх механізмів інформування та забезпечення централізованої доставки повідомлень учасникам освітнього процесу через різні канали зв'язку. Розроблення системи ґрунтується на принципах модульного моноліту, що дозволяє керувати змістом повідомлень, їхніми версіями, налаштуваннями каналів розсилки, а також доступом до системи з боку зовнішніх системи.

Результати роботи можуть бути використані для автоматизації інформування в університетському середовищі, забезпечуючи своєчасне доведення академічних, адміністративних та інших сповіщень учасниками освітнього процесу за допомогою дозволених платформ.

Ключові слова: система сповіщень, багатоканальність, модульність, автоматизоване інформування, версіонування та управління контентом, комунікаційні платформи.

ВСТУП

Обґрунтування вибору теми роботи

Організація освітнього процесу в університеті передбачає постійний обмін інформацією між адміністрацією, викладачами та здобувачами вищої освіти. Студенти потребують синхронізації з великою кількістю даних щодо академічних процесів, подій, дедлайнів, адміністративних розпоряджень та іншої інформації. Оперативне інформування студентів та цільової авдиторії загалом про актуальні події та процеси — критично важливий чинник для ефективного функціонування університету та підтримки високої якості освітнього процесу. Традиційні засоби комунікації, як-от електронна пошта, можуть бути недостатніми або незручними для забезпечення студентів повноцінною і вчасною інформацією.

У відповідь на цю потребу в НаУКМА використовували автоматизовані засоби на базі платформ обміну повідомленнями. Зокрема, для інформування студентів про різні аспекти навчання, події, анонси тощо застосовувалися можливості месенджера Telegram. Telegram — це багатоплатформний хмарний засіб зв'язку (далі — провайдер) [1], який також надає можливість використовувати спеціальних програмних агентів — Telegram-ботів [2], які в свою чергу дозволяють надсилати сповіщення користувачам та взаємодіяти з ними. Власне, цей блок механізмів забезпечував канал комунікації. Прикладом такого використання було інформування студентів про етапи роботи системи автоматизованого запису (САЗ) — інструменту для формування індивідуальних навчальних планів [3].

В умовах змін у політиці використання комунікаційних платформ у НаУКМА виникла потреба в перегляді існуючих механізмів сповіщення

студентів. Зокрема, з 1 січня 2025 року, згідно з Наказом №596 НаУКМА від 15 листопада 2024 року [4], вводиться заборона на використання месенджера Telegram для всіх службових та освітніх комунікацій, а також вимагається припинення функціонування офіційних облікових записів університету в цьому месенджері. Така ситуація створює виклик для забезпечення безперервної комунікації з учасниками освітнього процесу через різні канали зв'язку.

Зазначені виклики у сфері комунікацій в освітньому процесі університету зумовлюють потребу в розробленні нового механізму багатоканального інформування. Такий механізм, що використовує різні провайдери, дозволить розширити авдиторію та нівелювати ризики, пов'язані з відмовою одного з каналів зв'язку, забезпечуючи таким чином диверсифікацію системи. Така система має стати відповіддю на поточні обмеження, зокрема, як альтернатива до рішень, які до цього функціонували на базі можливостей месенджера Telegram. Ключовим завданням системи є надання масштабованого і, що важливо, централізованого інструменту для інформування студентів та інших учасників освітнього процесу за допомогою дозволених платформ: про академічні дедлайни, адміністративні розпорядження, анонси подій тощо.

Спроба задовольнити цю потребу через інтеграцію функціоналу сповіщень безпосередньо в кожну окрему внутрішньоуніверситетську систему (наприклад, САЗ чи інші адміністративні платформи) може значно ускладнити ці системи. Це ускладнює і масштабування, і підтримку, і інтеграцію цих окремих систем з новими провайдерами. Крім того, такий підхід обмежує можливість єдиного управління всіма університетськими сповіщеннями, що і суперечить принципам розмежування повноважень (Separation of Concerns), і створює надлишкові залежності. Тобто такий підхід

створює багато точок відмови і додає складності, якої можна уникнути шляхом розподілення.

Важливо зазначити, що така система не має на меті повне заміщення точкових комунікацій, які здійснюються безпосередньо викладачами, адміністрацією або іншими учасниками освітнього процесу. Система радше є механізмом інформування на загальному рівні, що забезпечить доставку повідомлень до широкого кола користувачів. Функціонал системи зосереджений саме на централізованому доведенні структурованої інформації про події та процеси, які стосуються значної кількості потенційно зацікавлених у інформації осіб.

Мета і завдання роботи

Метою роботи є створення та впровадження багатоканальної системи сповіщень для університету, призначеної для заміщення попередніх механізмів інформування та доведення інформації до студентів через різні канали зв'язку.

Об'єктом роботи є процеси автоматизованого інформування учасників освітнього процесу в НаУКМА, зокрема в контексті адаптації до змін в політиці використання комунікаційних платформ.

Предметом роботи є проєктування та реалізація багатоканальної системи сповіщень, що забезпечує управління змістом повідомлень, способами їхньої доставки та доступом користувачів.

Виконано такі відокремлені *завдання* для досягнення поставленої мети в роботі:

- аналіз ролей користувачів системи та їхніх потреб для визначення ключових функціональних компонентів і вимог до інтерфейсів взаємодії;
- аналіз наявних провайдерів з метою оцінки їхньої відповідності вимогам системи та розроблення принципів масштабованості та розширюваності інтеграційних механізмів;
- проєктування архітектури системи, що включає проєктування схеми бази даних, серверної частини та адміністративної панелі;
- формування плану розробки;
- реалізація ключових функціональних частин системи, зокрема управління користувачами, повідомленнями, провайдерами та каналами сповіщень;
- проведення перевірки та випробування створеної системи на відповідність вимогам;
- формування перспективних напрямків розвитку системи, включаючи можливі розширення функціональності, інтеграцію з новими провайдерами та покращення існуючих компонентів;

Застосовані методології

Методологічну основу роботи становлять методи системного та порівняльного аналізу для обґрунтування обраних рішень. Розроблення системи базується на принципах системного та об'єктно-орієнтованого проєктування, що охоплює створення її архітектури та програмних компонентів. Для візуалізації високорівневої взаємодії компонентів системи використано C4-діаграми, для візуалізація взаємодії сутностей — ER-діаграми. Для проєктування інтерфейсів застосовано принципи RESTful

API. Для механізмів автентифікації використано методи криптографічного захисту даних.

Структура та обсяг роботи

Загальний обсяг кваліфікаційної роботи становить 63 сторінки. Робота містить 5 розділів основної частини. Список використаних джерел налічує 26 одиниць. Кожен розділ розкриває суть відокремлених задач, описаних вище.

РОЗДІЛ 1
АНАЛІЗ ВИМОГ І ПОТРЕБ КОРИСТУВАЧІВ СИСТЕМИ

1.1. Вступ до розділу

Аналіз вимог — перший і ключовий етап розроблення системи, що задає траєкторію подальшому проєктуванню та реалізації. У цьому розділі ідентифіковано ключових акторів системи та проаналізовано їхні потреби й очікування. На основі цього аналізу сформульовано функціональні та нефункціональні вимоги, а також обмеження, що надалі визначатимуть архітектурні рішення та функціональні можливості системи. Застосування такого підходу гарантує, що розроблена система максимально відповідатиме своєму призначенню та задовольнятиме потреби користувачів.

1.2. Ідентифікація та аналіз акторів системи

У процесі проєктування системи виділено декілька категорій користувачів (або акторів), взаємодія яких із системою визначають її основні функціональні блоки.

1.2.1. Актор: Студент — отримання інформації

Роль та потреби: Студент є кінцевим бенефіціаром системи. Основна *потреба* студента полягає в отриманні актуальної інформації про події та процеси, що стосуються його навчання та університетського життя, через канал зв'язку, який є для нього зручним та доступним. Ця модель спілкування відповідає принципу Publisher-Subscriber [5], де система надає інформацію, а студент підписується на відповідні канали розсилки. Приклад такої моделі був реалізований за допомогою Telegram-каналів, що дозволяло користувачам вільно підписатися на розсилку. Переваги моделі Publisher-Subscriber,

порівняно з надсиланням повідомлень кожному користувачеві окремо, полягають у:

- *ефективності та масштабованості*: система надсилає повідомлення одноразово до каналу, незалежно від кількості підписників, що значно скорочує ресурси, необхідні для масових розсилок;
- *гнучкості для отримувача*: студенти можуть самостійно керувати своєю підпискою, приєднуючись або залишаючи канал залежно від власних потреб, що забезпечує контроль над отримуваною інформацією;
- *централізації інформації*: уся важлива інформація збирається в одному місці, доступному для перегляду підписникам у будь-який час;

Також важливо зауважити, що надсилання сповіщень має відбуватися паралельно у кожен з обраних каналів задля забезпеченням отримувачів інформацією одномоментно, адже це є важливим принципом роботи комунікаційних систем.

Висновок щодо вимог до системи: Система повинна забезпечувати доставку повідомлень до широкого кола отримувачів через різні платформи, не вимагаючи індивідуальної конфігурації для кожного, зокрема мінімізуючи відхилення часу доставки за рахунок паралельності запитів до провайдерів. Також отримувач не повинен бути об'єктом прямої взаємодії з системою — отримувач має управляти лише своєю підпискою на канал розсилки, через який бажає отримувати сповіщення.

1.2.2. Актор: Менеджер — управління сповіщеннями та каналами

Роль та потреби: Цей актор відповідає за формування змісту сповіщень, їхню конфігурацію, вибір каналів доставки та ініціювання

відправлення. Для нього важливо, щоб процес керування був простим і зрозумілим, не вимагаючи знань про особливості роботи інтегрованих провайдерів.

Потреби менеджера контенту включають:

- створення та редагування повідомлень з різним контентом;
- активація і деактивація повідомлень;
- перегляд історії змін контенту повідомлень та керування версіями;
- додавання, конфігурація та управління каналами, пов'язаними з різними провайдерами;
- активація і деактивація каналів;
- вибір конкретних каналів для відправлення певного повідомлення;
- функціонал для ручного ініціювання надсилання повідомлень;

Висновок щодо вимог до системи: Система має надати централізовану адміністративну панель, що забезпечує інтерфейс для управління всіма аспектами роботи зі сповіщеннями і каналами. Система має абстрагувати технічні особливості взаємодії з провайдерами від менеджера.

1.2.3. Актор: Інтегратор — програмна взаємодія

Роль та потреби: Інтегратор — це розробник або адміністратор іншої внутрішньоуніверситетської системи (наприклад, системи автоматизованого запису — САЗ, системи керування гуртожитками — DMS), яка потребує програмної інтеграції з системою сповіщень. Його *потреба* полягає у простому способі авторизованої ініціації відправлення повідомлень із зовнішньої системи.

Висновок щодо вимог до системи: Система повинна надавати чіткий та мінімалістичний програмний інтерфейс для інтеграції зі сторони зовнішніх

клієнтів. Цей інтерфейс має дозволяти інтегратору ініційовувати надсилання визначеного повідомлення за його унікальним ідентифікатором без знання внутрішньої будови та налаштувань системи. Перевірка авторизованості запитів має бути забезпечена зі сторони системи сповіщень.

1.3. Функціональні вимоги до системи

Проаналізувавши вимоги і потреби користувачів, можна визначити такі функціональні вимоги до системи:

- управління повідомленнями:
    - створення, перегляд і редагування повідомлень;
    - ведення історії змін повідомлень через механізм ревізій;
    - активація і деактивація повідомлень;
    - можливість ручного ініціювання надсилання повідомлень з адміністративної панелі;
- управління каналами:
    - додавання, перегляд, редагування каналів, прив'язаних до конкретних провайдерів;
    - обрання провайдера каналу, що налаштовується;
    - активація і деактивація каналів;
    - конфігурація специфічних параметрів для кожного каналу за типом інтеграції їхнього провайдера;
- управління користувачами та доступом:
    - механізм автентифікації користувачів системи;
    - генерація та відкликання токенів для зовнішніх інтеграцій;
- багатоканальна доставка сповіщень:
    - надсилання повідомлень за їхнім унікальним ідентифікатором через програмний інтерфейс;

- адаптація змісту повідомлення під вимоги обраних провайдерів;
- паралельне надсилання одного повідомлення до кількох каналів для мінімізації затримки;

1.4. Нефункціональні вимоги до системи

Окрім функціональних можливостей, до системи також сформовано і нефункціональні вимоги, які визначають її якісні характеристики:

- *надійність*: система повинна забезпечувати високу ймовірність доставки сповіщень, тому це вимагає обробки помилок і механізмів повторних спроб надсилання у разі тимчасових збоїв;
- *розширюваність*: необхідність у належному проєктуванні системи з метою легкої інтеграції нових провайдерів у майбутньому;
- *безпека*: забезпечення автентифікації і авторизації користувачів, захист токенів та паролів.
- *зручність використання*: адміністративна панель системи повинна бути зрозумілою та зручною для менеджерів та інтеграторів;
- *продуктивність*: забезпечення мінімальної затримки від моменту ініціації надсилання сповіщення до його доставки в канали;

1.5. Обмеження системи

Під час розробки системи враховуються такі обмеження:

- система не має надавати механізм інтеграції нових провайдерів через користувацький інтерфейс, адже така інтеграція потребує прямого втручання на рівні програми;
- система обмежена доступними методами інтеграції, які пропонують провайдери;

1.6. Висновок до розділу

У цьому розділі проведено ідентифікацію ключових акторів системи та аналіз їхніх потреб. На основі цього аналізу сформовано функціональні та нефункціональні вимоги до системи, а також визначено її обмеження. Сформульовані вимоги — це фундамент для подальшого проєктування системи та реалізації системи, що представлено у наступних розділах роботи.

РОЗДІЛ 2
АНАЛІЗ І ПОРІВНЯННЯ ПРОВАЙДЕРІВ

2.1. Вступ до розділу

Проєктування багатоканальної системи сповіщень вимагає ретельного вибору провайдерів зв'язку, через які буде здійснюватися доставка інформації. Поточний ринок комунікаційних платформ пропонує різноманітні рішення, що відрізняються функціональністю, моделлю взаємодії та умовами інтеграції. Метою цього розділу є визначення та обґрунтування вибору оптимальних провайдерів для інтеграції в систему, що розробляється. Для об’єктивної оцінки сформовано набір критеріїв, проаналізовано низку поширених в Україні комунікаційних платформ на відповідність цим критеріям та зроблено висновок щодо найбільш доцільних кандидатів для первинної реалізації.

2.2. Критерії вибору провайдерів зв'язку

Для об'єктивної оцінки та вибору провайдерів визначено такі критерії:

- *простота програмної інтеграції*: наявність чітко задокументованих API та простих механізмів для надсилання повідомлень, можливість інтеграції провайдера з мінімальними технічними ускладненнями;
- *підтримка моделі Publisher-Subscriber*: наявність механізмів, що дозволяють реалізувати модель, де користувачі можуть вільно підписуватися на канали розсилки;
- *безкоштовність*: відсутність фінансових витрат на використання API провайдера для некомерційних освітніх цілей;

- *відсутність жорстких обмежень за кількістю отримувачів*: канали провайдера не повинні мати суттєвих обмежень за кількістю підписників;
- *популярність*: активне використання платформи, зокрема серед студентів;
- *наявність у корпоративному просторі університету*: інтеграція з платформами, що вже використовуються в університеті для офіційних комунікацій;

2.3. Огляд популярних комунікаційних платформ

Для аналізу обрано низку поширених в Україні комунікаційних платформ. Згідно з даними аналітичних сервісів SimilarWeb [6] та Appfigures [7], до провідних месенджерів належать Viber і WhatsApp. Окремо розглянуто Discord через його поширеність серед студентів та Microsoft Teams як платформу, вже інтегровану в корпоративний простір НаУКМА в межах корпоративного пакету Microsoft 365.

2.4. Аналіз провайдерів на відповідність критеріям

2.4.1. Discord

Простота інтеграції: Надає механізм вхідних вебхуків [8], що дозволяє надсилати повідомлення через HTTP POST—запити. Також існують Discord Apps [9] для розширених завдань.

Підтримка моделі Publisher-Subscriber: Текстові канали на серверах Discord реалізують цю модель.

Безкоштовність: Використання вебхуків та базова функціональність серверів є безкоштовними.

Обмеження за кількістю отримувачів: Ліміти на кількість учасників сервера та каналу є високими — 25 000 учасників [10].

Популярність: Виходячи з емпіричних спостережень, платформа є поширеною серед студентів.

Наявність у корпоративному просторі: Не інтегрована.

Висновок щодо Discord: Платформа відповідає ключовим критеріям, зокрема щодо простоти інтеграції, підтримки канальної моделі, безкоштовності та поширеності серед цільової авдиторії, що робить її доцільним кандидатом для інтеграції.

2.4.2. Microsoft Teams

Простота інтеграції: Підтримує вхідні вебхуки для надсилання повідомлень у канали [11]. Для складніших сценаріїв інтеграції існує Microsoft Bot Framework. Водночас варто врахувати, що підтримка класичних вебхуків має оголошену дату припинення, що потенційно може вимагати адаптації системи в майбутньому.

Підтримка моделі Publisher-Subscriber: Канали в командах MS Teams дозволяють реалізувати цю модель.

Безкоштовність: Входить до складу корпоративного пакету Microsoft 365, використовуваного в НаУКМА.

Обмеження за кількістю отримувачів: Ліміти на кількість учасників команди є достатніми для університетських потреб.

Популярність: Використання є значним в контексті офіційного навчального процесу.

Наявність у корпоративному просторі: Є основною корпоративною платформою для комунікацій в НаУКМА.

Висновок щодо Microsoft Teams: Платформа відповідає важливим критеріям, особливо щодо інтеграції в корпоративний простір університету. Простота інтеграції через вебхуки (на поточний момент) та відсутність додаткових витрат роблять її придатною для включення в систему.

2.4.3. WhatsApp

Простота інтеграції: WhatsApp пропонує Cloud API [12], хостований Meta. Інтеграція передбачає створення спеціального бізнес-акаунту, верифікацію шаблонів повідомлень, наявність номерів користувачів [13].

Підтримка моделі Publisher-Subscriber: Комунікаційна модель WhatsApp не передбачає публічних, відкритих для підписки каналів. Розсилки через Cloud API зазвичай орієнтовані на користувачів, які надали попередню згоду (opt-in) на отримання повідомлень від конкретного бізнес-акаунту [14].

Безкоштовність: Ціна використання залежить від кількості ініційованих бізнесом розмов. Meta надає певну кількість безкоштовних одиниць, але подальше використання є платним і тарифікується залежно від категорії розмови та регіону [15].

Обмеження за кількістю отримувачів: Існують ліміти на кількість унікальних користувачів, яким бізнес може надсилати повідомлення протягом доби. Ці ліміти багаторівневі та можуть збільшуватися залежно від обсягу використання та репутації бізнесу-акаунту, а для підвищення лімітів бізнес-акаунт має отримати згоду від Meta [16]. Є можливість додати бот-акаунт у групу з учасниками, але кількість учасників обмежена — 1 024 учасника [17].

Популярність: Висока в Україні для особистого спілкування.

Наявність у корпоративному просторі: Не інтегрована.

Висновок щодо WhatsApp: Характеристики WhatsApp як платформи, враховуючи потенційні витрати на розсилку та складність дотримання політик для масштабних розсилок, роблять його менш пріоритетним для інтеграції в систему.

2.4.4. Viber

Простота інтеграції: Надає Broadcast API для масових розсилок [18].

Підтримка моделі Publisher-Subscriber: Користувачі можуть підписуватися на створений бот-акаунт, надіславши йому повідомлення.

Безкоштовність: Створення бот-акаунту, за допомогою якого реалізується використання Broadcast API, є комерційною послугою.

Обмеження за кількістю отримувачів: Broadcast API дозволяє надсилати не більше 300 повідомлень, ліміти становлять 500 запитів на 10 секунд.

Популярність: Висока в Україні для особистого спілкування.

Наявність у корпоративному просторі: Не інтегрована.

Висновок щодо Viber: Потенційні витрати на масові розсилки та складніша інтеграція порівняно з вебхуками роблять Viber менш придатним для первинної інтеграції.

2.5. Висновки та обґрунтування вибору

На основі проведеного аналізу, найбільш доцільними для первинної інтеграції в систему сповіщень є Discord та Microsoft Teams.

Ці платформи:

- підтримують модель каналів, що дозволяє реалізувати механізм Publisher-Subscriber;

- пропонують прості механізми програмної інтеграції для надсилання повідомлень через вебхуки, що значно спрощує їхню реалізацію для задачі одностороннього надсилання сповіщень;
- є безкоштовними або доступними в рамках існуючих університетських ліцензій;
- мають достатньо високі ліміти на кількість учасників у каналах;
- охоплюють значну частину цільової авдиторії університету;

Інші розглянуті популярні провайдери — WhatsApp, Viber — мають суттєві обмеження, пов'язані зі складністю API для масових розсилок, відсутністю повноцінних відкритих каналів, політиками використання та потенційними фінансовими витратами для некомерційного використання в необхідних масштабах.

Архітектура системи сповіщень проєктується з урахуванням принципу розширюваності. Це передбачає можливість додавання підтримки інших каналів доставки в майбутньому, за умови наявності потреби та відповідних технічних можливостей з боку провайдерів.

РОЗДІЛ 3

ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ

3.1. Вступ до розділу

Цей розділ представляє проєктування архітектури системи. Визначені архітектурні рішення — це основа для подальшої реалізації, що визначають принципи будови системи, її компонентів та взаємодії між ними. Розглянуто загальний архітектурний підхід, обґрунтовано вибір ключових технологій, деталізовано архітектуру серверної та клієнтської частин, принципи взаємодії їхніх складових, а також механізм багатоканальної доставки повідомлень.

3.2. Загальний архітектурний підхід

Загальна структура кодової бази — монорепозиторій. Такий підхід передбачає зберігання коду серверної частини та адміністративної панелі в одному проєкті. Такий підхід значно спрощує процес додавання нового функціоналу. Також тут закладається потенційне додавання процесів розгортання (далі — CI/CD) системи або у хмарному середовищі, або на внутрішній фізичних серверах університету, адже монорепозиторій забезпечує уніфікацію процесів CI/CD.

Проєкт розділений на клієнтський та серверний підпроєкти. Серверна частина реалізує основну бізнес-логіку системи, керує даними, взаємодіє з БД та інтегрує зовнішні провайдери. Клієнтська частина, в свою чергу, є адміністративною панеллю, середовищем використання якої є веббраузер.

Структура серверної частини — модульний моноліт. Цей архітектурний патерн передбачає організацію великого застосунку у вигляді сукупності логічно відокремлених модулів, кожен з яких відповідає за певні домен і

функціональність. Перевага такого підходу полягає у можливості збереження переваг єдиного процесу розгортання та відносної простоти управління кодом, що характерно для монолітних систем. При цьому забезпечуються і чітка структуризація коду, і розділення модулів за їхніми зонами відповідальності.

Структура клієнтської частини реалізована як односторінковий застосунок (далі — SPA). Це забезпечує динамічну взаємодію з користувачем без перезавантаження сторінок.

3.3. Високорівнева структура системи

Високорівнева структура системи та її взаємодія з основними зовнішніми сутностями візуалізована за методологією C4.

3.3.1. Діаграма контексту

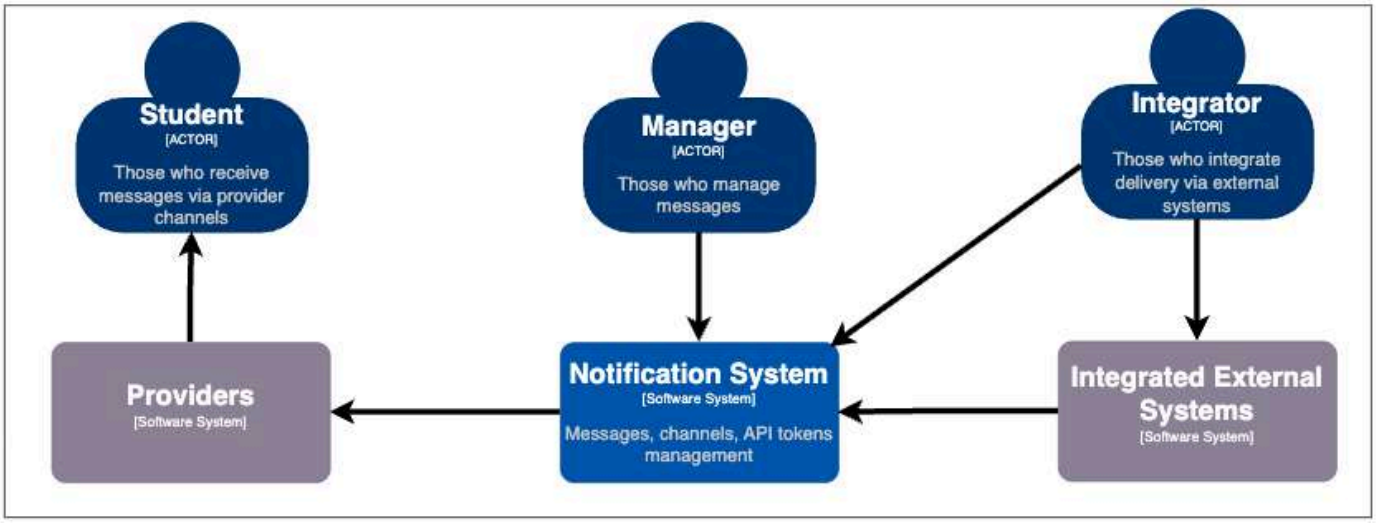


Рисунок 3.1. — Діаграма контексту взаємодії основних сутностей

Діаграма контексту (рисунок 3.1.) відображає систему сповіщень як єдине ціле, її основних акторів і взаємодію із зовнішніми компонентами.

Основними акторами є Студент, Менеджер та Інтегратор. Система взаємодіє з з провайдерами для доставки сповіщень у необхідні канали. У систему можуть надходити запити із інтегрованих зовнішніх систем.

3.3.2. Діаграма контейнерів

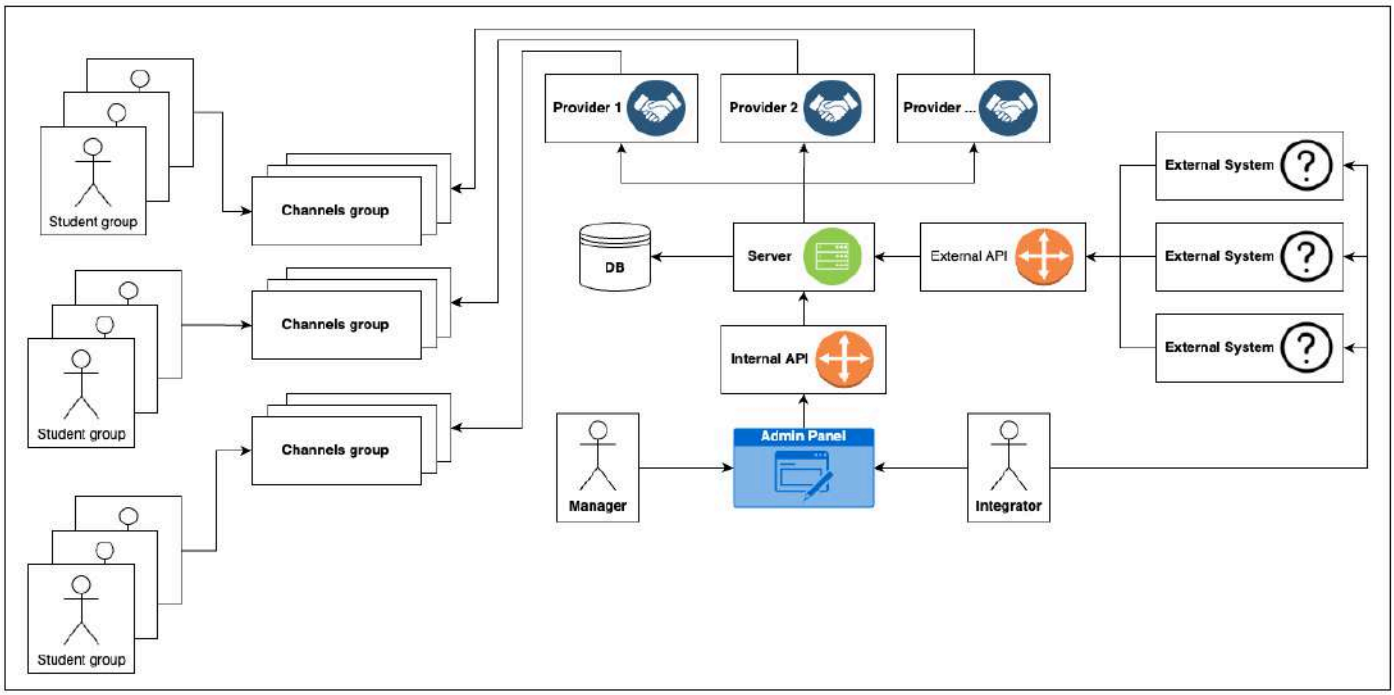


Рисунок 3.2. — Діаграма контейнерів взаємодії внутрішніх і зовнішніх сутностей

Діаграма контейнерів (рисунок 3.2.) деталізує систему, показуючи її основні компоненти. Це включає клієнтську частину (адміністративну панель як React-застосунок), серверну частину (NestJS-сервер) та PostgreSQL-БД. Адміністративна панель взаємодіє з серверною частиною через внутрішнього API. Також на сервер можуть надходити запити із зовнішнього API. Серверна частина зберігає дані у БД та надсилає сповіщення через провайдерів у обрані канали.

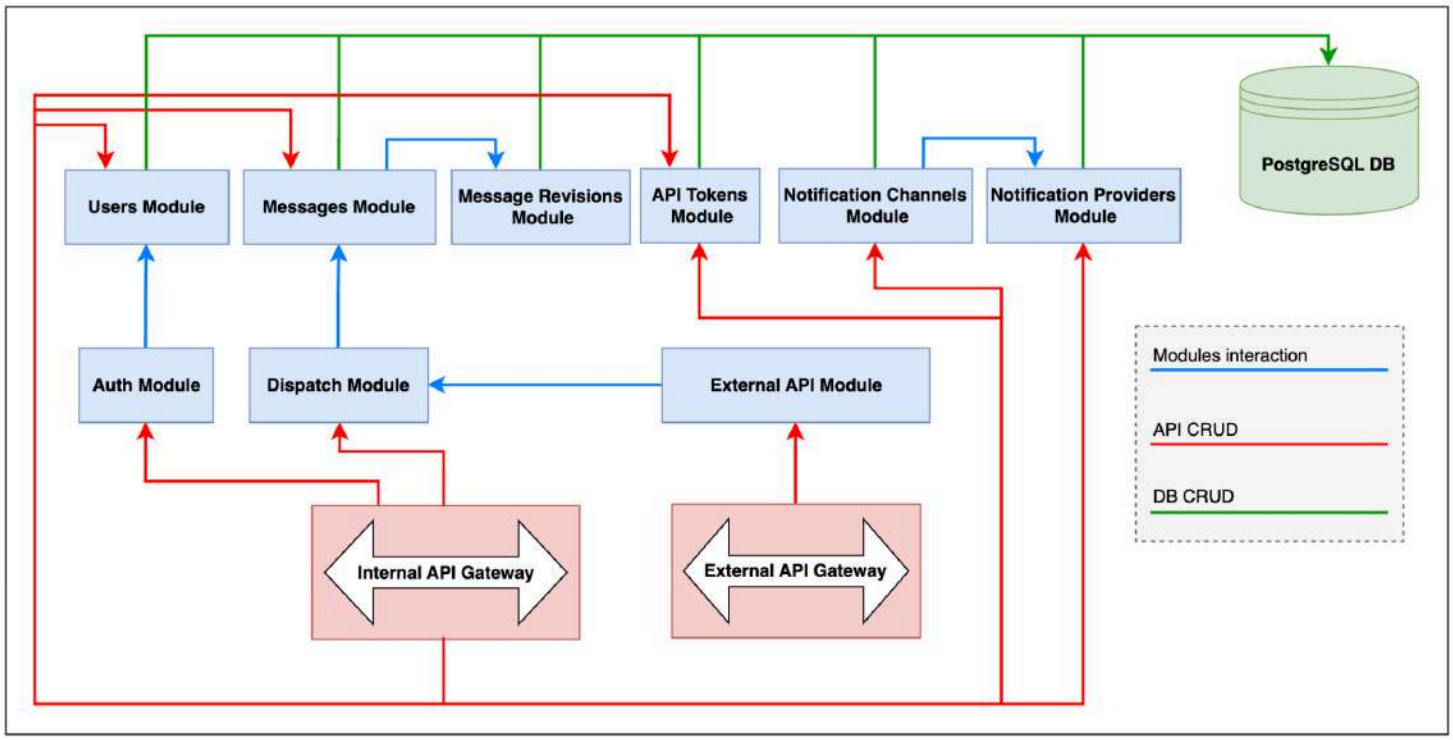


Рисунок 3.3. — Діаграма компонентів взаємодії внутрішніх сутностей

Діаграма компонентів (рисунок 3.3.) розкриває внутрішню будову серверної частини, деталізуючи її логічні компоненти, або ж модулі, їхню взаємодію, взаємодію з інтерфейсами внутрішнього і зовнішнього API, а також взаємодії з БД. Кожен модуль виконує функції відповідно до свого домену. Опис роботи кожного модулю представлено в наступних секціях.

3.4. Обґрунтування вибору технологій

Вибір технологій для реалізації системи базується на їхніх можливостях щодо управління даними, організації коду та побудови користувацького інтерфейсу, відповідно до вимог, сформованих у попередніх розділах. Враховуються прийняті рішення щодо патернів архітектури, сформовані в попередніх розділах.

3.4.1. Мова програмування

Обрана мова — TypeScript. Вибір ґрунтується на власних побажаннях автора роботи та наявності в нього значного комерційного досвіду роботи з цією мовою.

3.4.2. База даних

Обрана технологія — PostgreSQL. Вибір PostgreSQL обумовлений його статусом потужної реляційної системи керування БД. PostgreSQL підходить для реалізації складних зв'язків між сутностями, що присутні в моделі даних системи. Ця система керування БД забезпечує цілісність даних та підтримує тип даних JSONB, який використовується для гнучкої конфігурації каналів.

3.4.3. Серверне середовище

Обрана технологія — Node.js. Як середовище виконання для серверної частини, Node.js обрано завдяки його простоті, асинхронній, подієво-орієнтованій архітектурі. Це дозволяє ефективно обробляти велику кількість запитів на надсилання сповіщень без блокування операцій. Також надано перевагу Node.js через доступність для нього фреймворків для якісної структруризації коду.

3.4.4 Фреймворк для сервера

Обрана технологія — NestJS [19]. NestJS — популярний фреймворк для розробки серверної частини в середовищі Node.js для TypeScript. Його вибір зумовлений архітектурними перевагами, такими як чітке розділення коду на модулі, дотримання принципів об'єктно-орієнтованого проєктування та

використання механізму Dependency Injection (далі — DI). Це сприяє написанню коду, який легко підтримується.

3.4.5. Об’єктно-реляційне відображення (ORM)

Обрана технологія — TypeORM [20]. TypeORM використовується як ORM для взаємодії БД, зокрема з PostgreSQL. TypeORM дозволяє працювати з базою даних через об'єкти TypeScript, що спрощує розроблення, надаючи можливість як налаштовувати схему БД, так і використовувати об'єкти сутностей безпосередньо в коді. Також зручним є те, що TypeORM підтримує створення складних запитів (містить query builder), що додає гнучкості при формуванні запитів. Це важливий аспект для роботи зі складними моделями даних, що присутні в роботі. Що не менш важливо, NestJS має підтримку для TypeORM [21], що спрощує розробку.

3.4.6. Клієнтський застосунок

Обрана технологія — чистий React [22]. Для розробки адміністративної панелі обрано цю бібліотеку. Її компонентний підхід до побудови інтерфейсів забезпечує можливість створення багаторазово використовуваних елементів інтерфейсу. React підходить для створення SPA. Враховуючи, що адміністративна панель — доволі простий застосунок і не потребує складних надбудов, у застосуванні фреймворків, таких як Next.js [23], немає потреби. Для навігації між сторінками використовується допоміжна бібліотека React Router [24]. Для керування глобальним станом обрано бібліотеку Zustand [25] — він простотий, має мінімалістичний API та ефективно управляє станом у схожого роду додатках без надлишкового коду, тому немає потреби в інтеграції складніших бібліотек для керування глобальним станом, як-от Redux [26].

3.5. Архітектура серверної частини та взаємодія компонентів

Серверна частина системи, реалізована на NestJS, побудована за модульним принципом. Це сприяє розділенню зон відповідальності між логічними блоками, що спрощує розробку і дає можливість закласти масштабованість на майбутні ітерації покращень системи.

Основними модулями серверної частини, відповідно до їхньої структури:

- *AuthModule*: Відповідає за процеси автентифікації користувачів системи. Взаємодіє з *UsersModule* для перевірки облікових даних та використовує *JwtModule* для генерації та перевірки JWT-токенів доступу.
- *UsersModule*: Керує даними внутрішніх користувачів. Надає функціонал для створення, пошуку та перевірки користувачів — реєстрації та автентифікації.
- *ApiTokensModule*: Забезпечує управління токенами, що використовуються зовнішніми системами для авторизованих запитів. Також містить функціонал створення, перегляду та відкликання токенів.
- *NotificationProvidersModule*: Керує даними про провайдерів.
- *NotificationChannelsModule*: Керує даними про канали сповіщень. Взаємодіє з *NotificationProvidersModule* для отримання інформації про провайдерів, до яких прив'язані самі канали.
- *MessagesModule*: Керує життєвим циклом повідомлень, зокрема їхнім створенням, редагуванням та керуванням версіями. Цей модуль взаємодіє *MessageRevisionsModule* для роботи з ревізіями повідомлень та з *NotificationChannelsModule* для прив'язки повідомлень до каналів.

- *MessageRevisionsModule*: Відповідає за зберігання та управління версіями (ревізіями) повідомлень, включаючи їхній зміст. Винесення сутності контенту повідомлення в окремих модуль станом на зараз вважається надлишковим кроком.
- *DispatchModule*: Оркеструє процесом надсилання повідомлень до провайдерів. Отримує дані про повідомлення з *MessagesModule* та використовує стратегії диспетчеризації для взаємодії з конкретними провайдерами в залежності від того, з якими налаштуваннями зберігається повідомлення та його компоненти, зокрема дані про канали розсилки в окремій таблиці.
- *ExternalApiModule*: Реалізує інтерфейс для зовнішніх систем. Він забезпечує перевірку авторизованості запитів за токеном, що реалізується за рахунок взаємодії з *ApiTokensModule,* та ініціює процес диспетчеризації сповіщень, що реалізується за рахунок взаємодії з *DispatchModule*.

Взаємодія модулів відбувається за принципом інверсії управління (Inversion of Control), де модулі взаємодіють між собою через ін'єкцію сервісів, що реалізується за рахунок механізму DI. Наприклад, *AuthModule* покладається на *UsersService* для операцій з користувачами, а *ExternalApiModule* використовує *ApiTokensService* для перевірки токенів та *DispatchService* для ініціації надсилання. *DispatchModule* в свою чергу звертається до *MessagesService* для отримання даних повідомлення.

3.5. Архітектура даних та взаємодія сутностей

Архітектура даних системи побудована навколо низки взаємопов'язаних сутностей, що зберігаються у базі даних PostgreSQL. Кожна сутність

відповідає за зберігання певного типу інформації, а їхні зв'язки відображають логічні відносини між даними.

Основні сутності системи та їхня взаємодія:

- Користувач: Представляє внутрішніх акторів системи. Кожен користувач має унікальну електронну пошту, ім'я та хеш пароля.
- API-токен: Використовується для перевірки авторизованості запитів від зовнішніх систем. Містить хеш токена, префікс та опис. Може бути відкликаний.
- Провайдер: Описує загальні характеристики провайдера, включаючи його тип та тип інтеграції.
- Канал: Представляє конкретний канал сповіщення. Кожен канал належить до певного Провайдера та містить конфігураційні дані у форматі JSONB, що дозволяє зберігати параметри, специфічні для кожного провайдера.
- Повідомлення: Центральна сутність системи. Містить унікальний текстовий ідентифікатор (далі — слаг), опис та статус. Кожне повідомлення має посилання на свою актуальну Ревізію. Повідомлення може бути призначене для надсилання в один або декілька Каналів.
- Ревізія повідомлення: Зберігає історію змін змісту повідомлення. Кожна ревізія належить до певного Повідомлення і має унікальний порядковий номер для цього повідомлення, що генерується автоматично. Ревізія може бути активною або заблокованою.
- Контент повідомлення: Зберігає безпосередній зміст конкретної ревізії повідомлення — заголовок та тіло. Кожен запис змісту прив'язаний до однієї Ревізії повідомлення.

Детальна структура бази даних та взаємозв'язки між сутностями системи представлені на ER-діаграмі (рисунок. 3.4.).

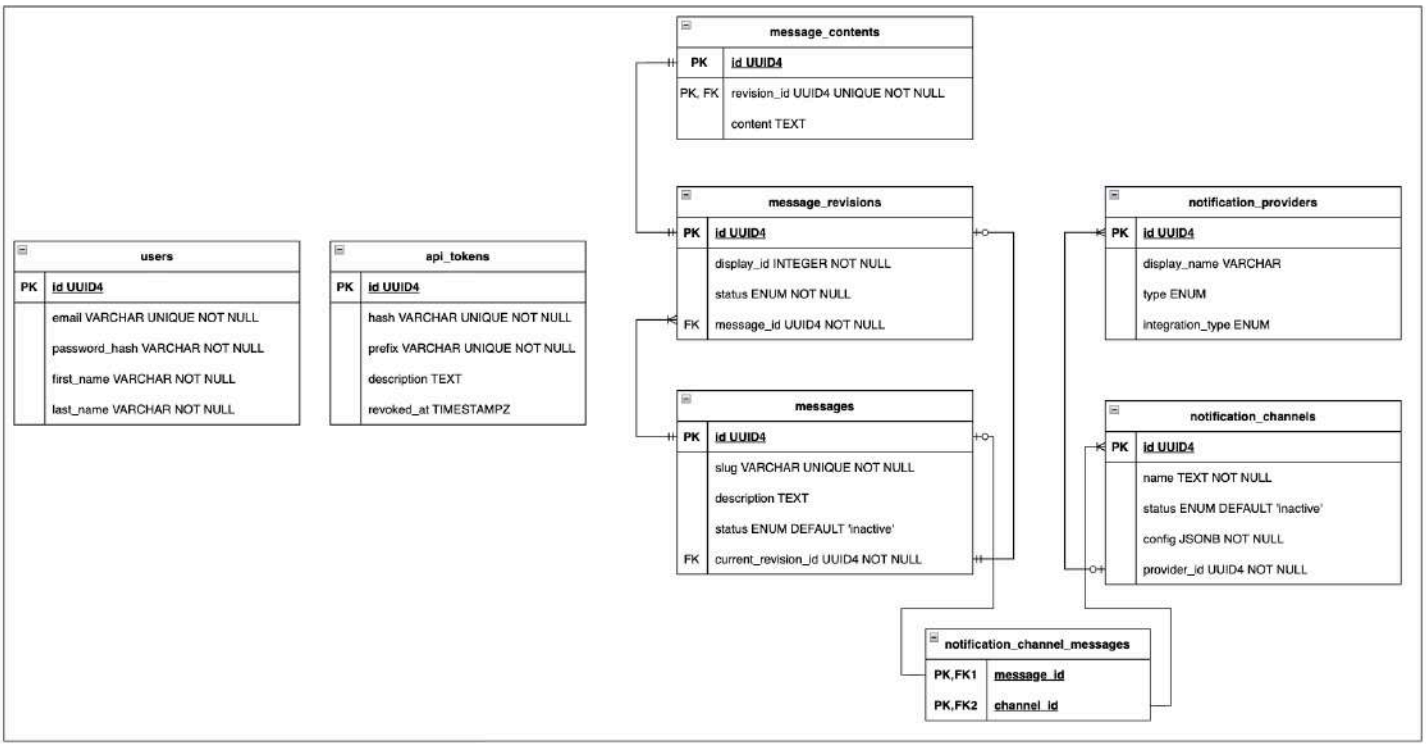


Рисунок 3.4. — ER-діаграма системи

3.6. Архітектура клієнтської частини

Клієнтська частина системи реалізована як SPA на базі бібліотеки React. Такий підхід забезпечує динамічну взаємодію з користувачем без необхідності у перезавантаженні сторінок, а також надає гнучкість у побудові інтерфейсу. Навігація між сторінками реалізована за допомогою бібліотеки React Router. Для керування глобальним станом використовується бібліотека Zustand.

Структура панелі поділяється на декілька сторінок:

- *сторінка реєстрації*: надає можливість новим користувачам створити обліковий запис у системі, вказавши необхідні для цього дані;

- *сторінка автентифікації*: є вхідною точкою для зареєстрованих користувачів, що забезпечує їм доступ до функціоналу панелі;
- *сторінка з таблицею API-токенів*: дозволяє авторизованим менеджерам та інтеграторам переглядати, генерувати та відкликати API-токени, які використовуються для авторизації запитів від зовнішніх систем;
- *сторінка з таблицею каналів*: надає інтерфейс для управління каналами сповіщень, включаючи створення, редагування параметрів, активацію/деактивацію та перегляд наявних каналів;
- *сторінка з таблицею повідомлень*: центральний компонент для роботи зі сповіщеннями, де користувачі можуть створювати нові повідомлення, редагувати їхній контент, переглядати ревізії, активовувати/деактивовувати та ініційовувати їхню відправку через обрані канали;

3.7. Висновок до розділу

У цьому розділі представлено архітектурні рішення для системи, обґрунтовано вибір монорепозиторію та модульного моноліту, деталізовано будову серверної частини та її взаємодію з клієнтською частиною і базою даних. Також обґрунтовано вибір ключових технологій та описано основні принципи роботи системи. Ці архітектурні рішення створюють основу для подальшої реалізації системи.

РОЗДІЛ 4

РЕАЛІЗАЦІЯ ФУНКЦІОНАЛЬНИХ ЧАСТИН СИСТЕМИ

4.1. Вступ до розділу

Цей розділ присвячений огляду практичної реалізації функціональних компонентів системи, спираючись на архітектурні рішення, викладені у Розділі 3. Розкрито особливості програмної імплементації серверної та клієнтської частин системи. Основна увага зосереджена на специфічних деталях розробки.

4.2. Реалізація серверної частини

Серверна частина системи розроблена на платформі Node.js з використанням фреймворку NestJS. Це і є ядро, що відповідає за обробку запитів, управління даними, зберігання у БД та взаємодію із зовнішніми провайдерами.

4.2.1. Застосування модульного підходу NestJS

Відповідно до спроєктованої архітектури модульного моноліту, серверна частина структурована за допомогою системи модулів NestJS. Кожен функціональний блок, як-от управління повідомленнями, каналами, користувачами чи API-токенами тощо, інкапсульований в окремому модулі. Таким чином забезпечується чітке розмежування відповідальності та слабка зв'язаність між компонентами. Взаємодія між сервісами різних модулів реалізована через механізм DI.

На рисунку 4.1. можна побачити готову структуру серверної частини.

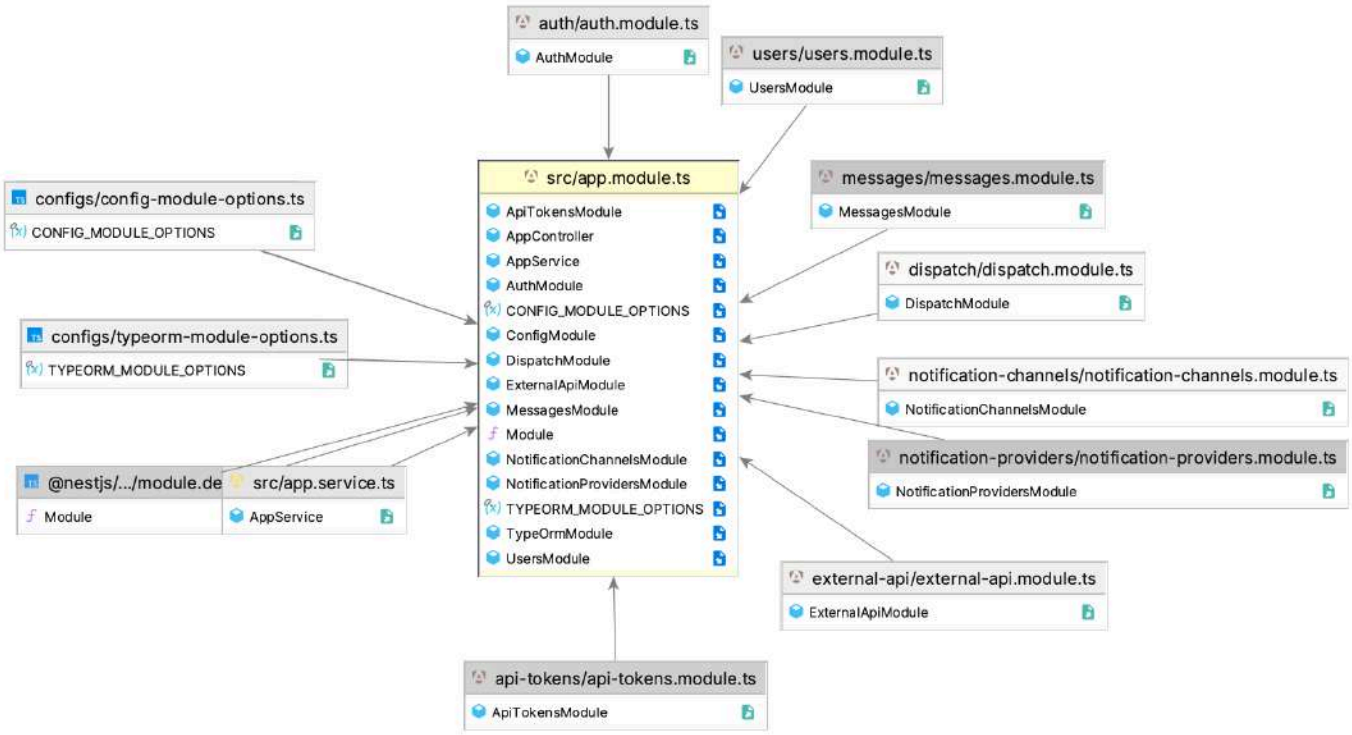


Рисунок 4.1 — Схематичне представлення взаємодії модулів серверної частини

4.2.2. Забезпечення цілісності даних при операціях з повідомленнями

Критичним аспектом під час роботи з повідомленнями та їхніми ревізіями є забезпечення цілісності даних. Так, при створенні нового повідомлення система гарантує, що саме повідомлення та його первинна ревізія будуть збережені в межах єдиною, атомарної операції. Для цього застосовані механізми транзакцій. Логіка створення цих пов'язаних сутностей інкапсульована у відповідному сервісі та виконується в межах однієї транзакції. Такий підхід унеможливлює ситуації, коли, наприклад, повідомлення створено, а його обов'язкова перша ревізія — ні. Це запобігає виникненню ситуацій, де система працює некоректно [Додаток А].

4.2.3. Механізми автентифікації та авторизації

Система включає два основних контури безпеки: автентифікацію менеджерів та інтеграторів для доступу до адміністративної панелі та авторизацію запитів від зовнішніх систем.

Автентифікація користувачів реалізована за допомогою JWT. Після перевірки облікових даних, а саме електронної пошти та хешованого пароля, сервер генерує JWT, який передається клієнту у вигляді HTTP-куки. Для захисту відповідних ендпоінтів використовується *JwtAuthGuard* [Додаток Б].

Для авторизації зовнішніх систем передбачено механізм API-токенів. При генерації токена система створює унікальний префікс та сам токен, після чого зберігає в базі даних лише хеш повного токена, згенерований за допомогою криптографічної функції SHA256. Користувачу ж повертається комбінація префікса та другої частини токена. Перевірка таких запитів здійснюється спеціальним *ApiTokenAuthGuard*, який валідує токен з HTTP-заголовка *Authorization*, порівнюючи хеші та перевіряючи статус токена [Додаток В].

4.2.4. Реалізація диспетчеризації повідомлень

Центральним компонентом, що відповідає за доставку сповіщень, є модуль диспетчеризації — *DispatchModule*. Для забезпечення гнучкості та розширюваності при роботі з різними провайдерами застосовано патерн проєктування “Стратегія”. Спеціальна фабрика *DispatchStrategyFactory* на основі типу провайдера, пов'язаного з каналом, динамічно обирає відповідну реалізацію стратегії [Додаток Г]. Кожна така стратегія інкапсулює логіку формування запиту, включно з адаптацією контенту під формат провайдера, та надсилання повідомлення через специфічний для цього провайдера

механізм [Додаток Ґ]. У поточній реалізації основним механізмом взаємодії є вебхуки. Але закладення згаданого патерну дозволить відносно легко додавати підтримку нових провайдерів у майбутньому, мінімально впливаючи на існуючу кодову базу системи. Надсилання одного сповіщення до кількох обраних каналів відбувається паралельно, використовуючи *Promise.all*, для оптимізації часу доставки.

4.2.5. Валідація конфігураційних даних каналів на рівні сутності

Для уникнення ситуацій некоректного збереження конфігураційних даних для каналів сповіщень реалізовано механізм валідації даних на рівні сутності *NotificationChannel* в TypeORM [Додаток Д]. За допомогою декоратора *@BeforeInsert*, перед безпосереднім збереженням сутності в БД, викликається метод, який перевіряє наявність та коректність необхідних полів у об'єкті *config* сутності. Наприклад, у випадку з каналом, що використовує інтеграцію за рахунок вебхуків, система має перевірити наявність поля *webhookUrl* [Додаток Е]. Якщо валідація не проходить, операція збереження переривається, система повертає помилку. Це забезпечує узгодженість даних ще на етапі їхнього запису в БД.

4.3. Реалізація клієнтської частини

Адміністративна панель, розроблена як SPA на React, слугує інтерфейсом для менеджерів та інтеграторів системи.

4.3.1. Архітектура SPA та управління станом

Клієнтський застосунок використовує компонентний підхід React для побудови користувацького інтерфейсу. Навігація між основними розділами —

API-токенами, каналами, повідомленнями — реалізована за допомогою бібліотеки *react-router-dom*. Для управління глобальним станом, таким як дані автентифікованого користувача та черга системних нотифікацій для користувача, застосовано бібліотеку Zustand. Її використання дозволило створити реактивні сховища стану (*useAuthStore*, *useNotificationStore*) без надлишкової складності, притаманної деяким іншим рішенням для управління станом. Взаємодія з API серверної частини організована через сервісний шар, що абстрагує логіку HTTP-запитів від компонентів [Додаток Є].

4.3.2. Захист сторінок та ініціалізація сесії користувача

Доступ до сторінок панелі обмежується для неавторизованих користувачів за допомогою компонента вищого порядку *ProtectedRoute* [Додаток Ж]. При першому завантаженні застосунку або при переході на захищений маршрут, цей компонент перевіряє стан автентифікації користувача в *useAuthStore*. Якщо користувач не автентифікований, його автоматично перенаправляють на сторінку автентифікації. Для ініціалізації та підтвердження стану автентифікації при завантаженні застосунку, клієнт робить запит до ендпоінту */auth/me* на сервері. Успішна відповідь від цього ендпоінту підтверджує валідність сесії користувача і завантажує дані про нього у сховище Zustand.

4.4. Висновок до розділу

У цьому розділі детально особливості реалізації системи на серверному та клієнтському рівнях. Застосування технологій та підходів, таких як модульна архітектура NestJS, компонентна модель React, механізми транзакцій для забезпечення цілісності даних, патерн “Стратегія” для

масштабованості диспетчеризації, а також надійні методи автентифікації та авторизації, дозволили створити функціонально насичену, розширювану та безпечну систему. Практичне втілення спроєктованих рішень демонструє готовність системи до виконання поставлених завдань з інформування користувачів.

РОЗДІЛ 5

ПРОВЕДЕННЯ ПЕРЕВІРКИ ТА ВИПРОБОВУВАННЯ СИСТЕМИ

5.1. Вступ до розділу

Після завершення етапу реалізації ключових функціональних частин системи, наступним логічним кроком є перевірка результатів. Цей розділ присвячений представленню сценаріїв тестування основних функцій та аналізу отриманих результатів. Метою цього етапу є підтвердження відповідності системи заявленим функціональним та нефункціональним вимогам, виявлення потенційних дефектів та оцінка загальної якості розробки.

5.2. Демонстрація роботи ключового функціоналу

5.2.1. Управління API-токенами для зовнішніх інтеграцій

Для забезпечення авторизованого доступу зовнішніх систем до API сповіщень, адміністративна панель надає функціонал управління API-токенами. Процес створення нового токена ініціюється на сторінці "API Tokens" (рисунок 5.1.). Менеджер або інтегратор вказує опис токена у модальному вікні "Create API Token".

Після успішного створення, згенерований токен відображається у модальному вікні (рисунок 5.2.). Система інформує користувача, що повний токен буде показано лише один раз, та надає можливість скопіювати його в буфер обміну. Спливаюче повідомлення підтверджує успішне копіювання.

Створений токен додається до загального списку на сторінці "API Tokens". У таблиці відображається префікс токена, його опис, дата створення

та поточний статус. Передбачено також функціонал для відкликання токена, що дозволяє припинити його дію за потреби.

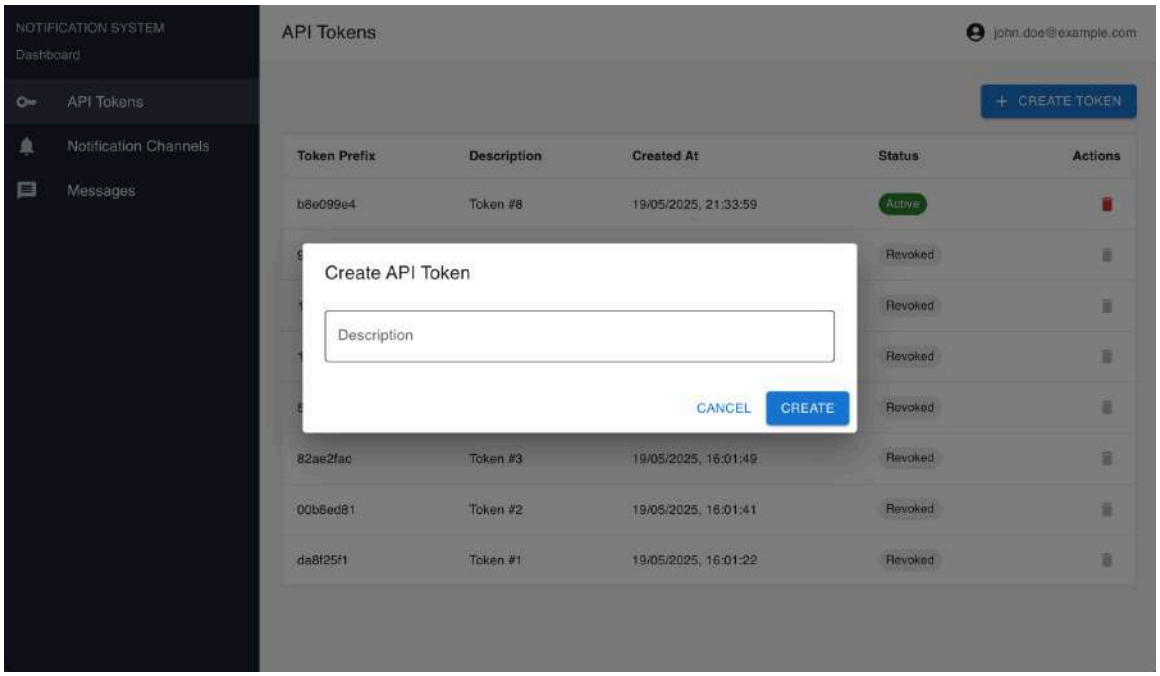


Рисунок 5.1. — Модальне вікно для створення токена

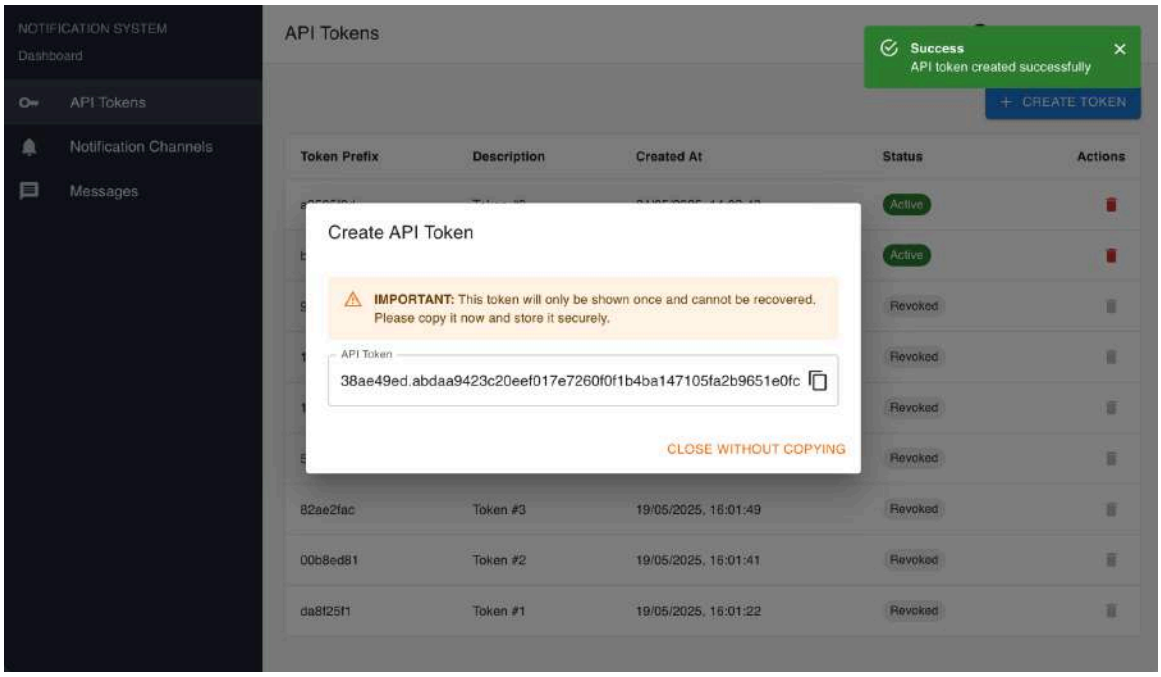


Рисунок 5.2. — Модальне вікно зі створених токеном

5.2.2. Налаштування та управління каналами сповіщень

Система дозволяє налаштовувати канали, через які будуть надсилатися сповіщення. На сторінці "Notification Channels" (рисунок 5.3.) менеджер може переглядати наявні канали та створювати нові. При створенні каналу обирається відповідний провайдер та вказуються необхідні конфігураційні параметри.

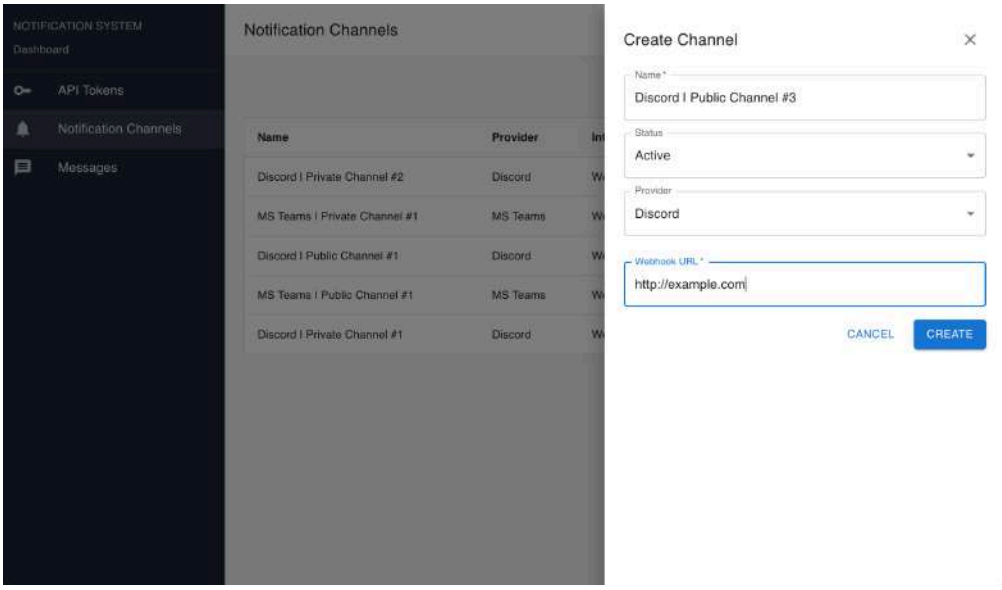


Рисунок 5.3. — Створення нового каналу

Редагування існуючих каналів також доступне. Менеджер може змінювати назву, статус каналу та його конфігурацію.

5.2.3. Створення та управління контентом повідомлень

Ключовою функцією системи є створення та управління повідомленнями. Сторінка "Messages" містить таблицю з усіма повідомленнями, де відображаються їхні слаги, описи, статуси, номери активних ревізій та дати створення. Доступні операції включають надсилання повідомлення та перегляд історії його ревізій (рисунок 5.4.).

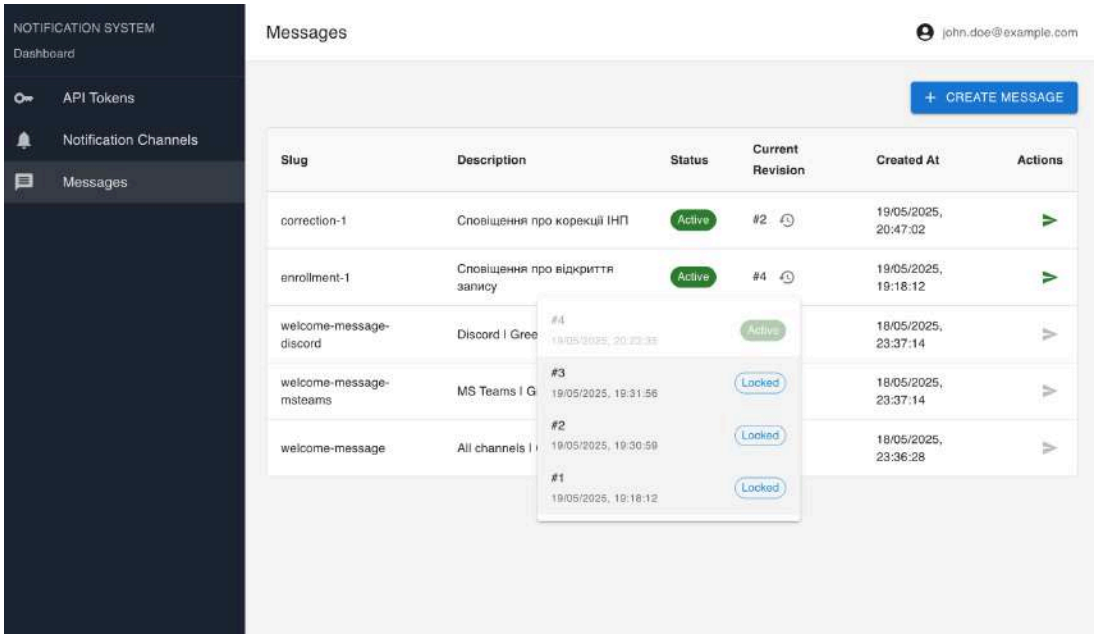


Рисунок 5.4. — Обрання ревізії

Для створення нового повідомлення використовується бічна панель (рисунок 5.5). Менеджер заповнює унікальний слаг, опис, заголовок та тіло повідомлення, обирає канали доставки (рисунок 5.5.).

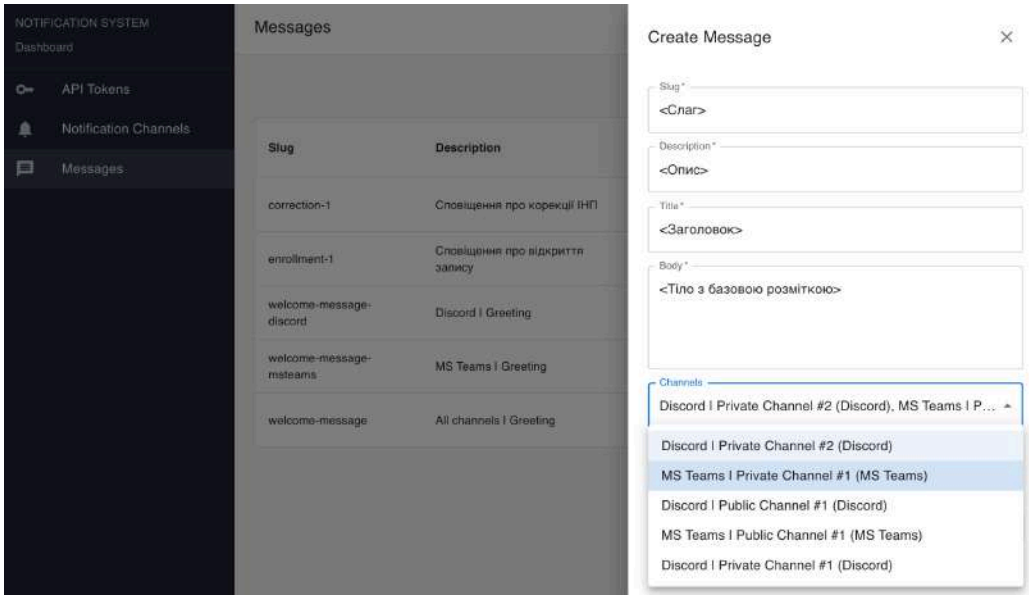


Рисунок 5.5. — Створення повідомлення

Редагування контенту наявного повідомлення здійснюється у схожому інтерфейсі. Менеджер може оновити опис, заголовок, тіло, змінити список асоційованих каналів та статус повідомлення.

Також повідомлення може бути надіслане з самої панелі, використовуючи відповідну кнопку відправки справа.

5.2.4. Процес надсилання повідомлення ззовні

Ініціювання надсилання сповіщення зокрема відбувається через зовнішній API-запит. На рисунку 5.6. показано приклад HTTP POST-запиту до ендпоінту */external-api/dispatch*, який містить слаг повідомлення та API-токен для авторизації.

Рисунок 5.6. — Приклад запиту ззовні

У разі успішної обробки запиту, система повертає деталізований JSON-об'єкт (рисунок 5.7), що містить інформацію про ідентифікатори повідомлення, загальний статус операції, статистику надсилань та масив каналів доставки. Останній надає інформацію про статус доставки для кожного конкретного каналу.

```
{
  "messageId": "a517fce5-b7b5-44d4-b78b-474410542832",
  "messageSlug": "correction-1",
  "status": "success",
  "stats": {
    "successfullySent": 2,
    "failedToSend": 0
  },
  "channelResults": [
    {
      "channelName": "Discord | Public Channel #1",
      "providerType": "discord",
      "status": "success"
    },
    {
      "channelName": "MS Teams | Public Channel #1",
      "providerType": "ms_teams",
      "status": "success"
    }
  ]
}
```


Рисунок 5.7. — Результат запиту ззовні

Результатом успішної диспетчеризації є доставка сповіщень до кінцевих користувачів у відповідних провайдерах (рисунки 5.8.-5.9.).

Рисунок 5.8. — Отримане повідомлення в Discord

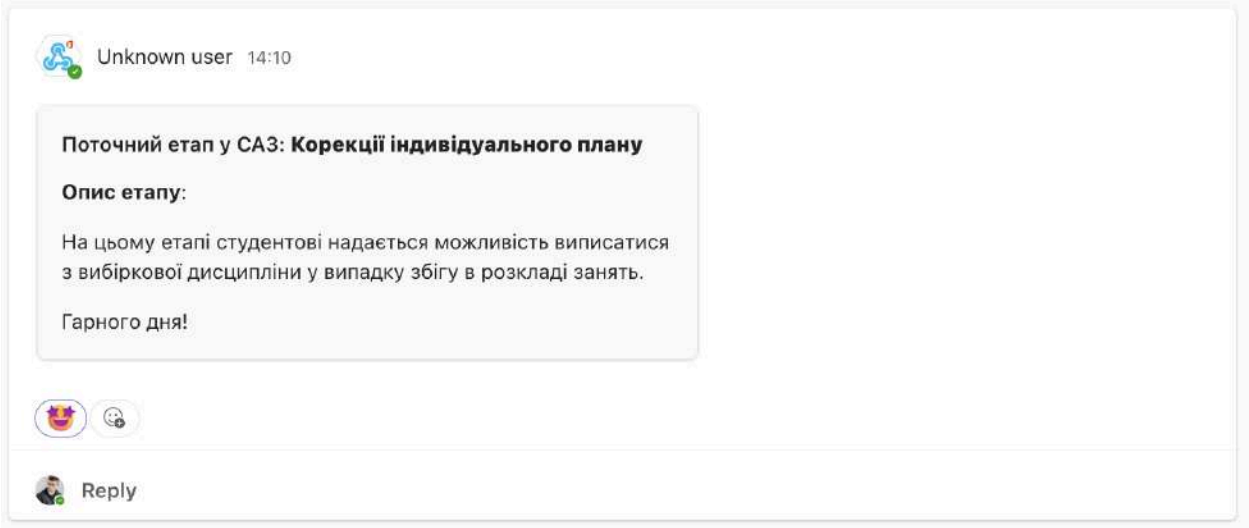


Рисунок 5.9. — Отримане повідомлення в Microsoft Teams

5.3 Висновок до розділу

Представлено основні сценарії для перевірки функціональності серверної та клієнтської частин, а також проаналізовано отримані результати. Проведені випробування підтвердили працездатність системи та її відповідність ключовим вимогам, що дозволяє перейти до обговорення перспективних напрямків її подальшого розвитку.

ВИСНОВКИ

У межах цієї роботи розроблено багатоканальну системи сповіщень. Проєкт мав на меті створення централізованого інструменту для інформування студентів, що міг би слугувати альтернативою забороненим забороненим каналам комунікації, зокрема враховуючи інформаційну політику НаУКМА.

На основі аналізу потреб користувачів та огляду доступних технологій спроєктовано та реалізовано систему, що включає серверну частину для обробки логіки та клієнтську адміністративну панель для управління. Серверна частина забезпечує автентифікацію менеджерів, авторизацію зовнішніх систем через API-токени, управління контентом повідомлень та каналами доставки, а також диспетчеризацію сповіщень через інтегровані провайдери — Discord та Microsoft Teams. Клієнтська частина надає інтерфейс для виконання всіх зазначених операцій.

Проведене тестування підтвердило працездатність ключових функцій системи та її відповідність основним вимогам. Система продемонструвала здатність доставляти сповіщення через різні канали та надавати інструменти для управління цим процесом.

Подальший розвиток системи може охоплювати декілька напрямків.

Функціональні покращення:

- інтеграція з додатковими провайдерами сповіщень (наприклад, електронна пошта за допомогою SMTP);
- розроблення механізмів планування (або відкладення) розсилок;
- впровадження механізму шаблонізації повідомлень;
- розширена аналітика доставки сповіщень;

- додавання пагінації в таблицях адміністративної панелі;
- реалізація пошуку та фільтрації для повідомлень, каналів та API-токенів;

Технічні вдосконалення:

- покращення механізмів повторних спроб надсилання повідомлень у разі збоїв;
- підготовка та реалізація розгортання системи в хмарному середовищі або на університетських серверах;
- розширення можливостей моніторингу та деталізованого логування;

Таким чином, розроблена система є робочим прототипом, що вирішує поставлені завдання та має чітко окреслені перспективи для подальшого вдосконалення та впровадження. Реалізація запропонованих покращень дозволить підвищити надійність, ефективність та зручність використання системи, перетворивши її на повноцінний інструмент для внутрішніх комунікацій у межах університету.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Telegram Messenger Inc. (n.d.-a). *Telegram Press Info*. Telegram. https://telegram.org/press
2. Telegram Messenger Inc. (n.d.-b). *Bots: An introduction for developers*. Telegram Core. https://core.telegram.org/bots/api
3. Персональний журнал Сергія Квіта. (2024). *Система автоматизованого запису (САЗ)*. Kvit. https://kvit.ukma.edu.ua/2024/06/система-автоматизованого-запису-саз
4. NaUKMA. (n.d.). *Про мінімізацію кіберзагроз пов'язаних з месенджером Telegram*. https://www.ukma.edu.ua/index.php/science/gradschool/plans/doc_download/3884-pro-minimizatsiiu-kiberzahroz-pov-iazanykh-z-mesendzherom-telegram
5. Microsoft. (n.d.-a). *Publisher-Subscriber pattern*. Microsoft Learn. https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber
6. Similarweb. (n.d.). *Most Popular Communication Apps in Ukraine*. Similarweb. https://www.similarweb.com/top-apps/google/ukraine/communication
7. Appfigures. (n.d.). *Top Communication Apps - Google Play - Ukraine*. Appfigures. https://appfigures.com/top-apps/google-play/ukraine/communication
8. Discord Inc. (n.d.-a). *Webhook resource*. Discord Developer Portal. https://discord.com/developers/docs/resources/webhook
9. Discord Inc. (n.d.-b). *Developing a user-installable app*. Discord Developer Portal.

https://discord.com/developers/docs/tutorials/developing-a-user-installable-app

10. Support Discord. (n.d.). *Group chat member limit*. Discord Support. https://support.discord.com/hc/en-us/community/posts/18963637551127-Group-chat-member-limit
11. Microsoft. (n.d.-b). *Add an incoming webhook to a Teams channel*. Microsoft Learn. https://learn.microsoft.com/en-us/microsoftteams/platform/webhooks-and-connectors/how-to/add-incoming-webhook
12. Facebook, Inc. (n.d.-a). *Get started - WhatsApp Cloud API*. Meta for Developers. https://developers.facebook.com/docs/whatsapp/cloud-api/get-started
13. Facebook, Inc. (n.d.-b). *WhatsApp Business policy*. Meta Business Help Center. https://business.whatsapp.com/policy#overview
14. Facebook, Inc. (n.d.-c). *WhatsApp Business policy (Quality Experience)*. Meta Business Help Center. https://business.whatsapp.com/policy#overview
15. Facebook, Inc. (n.d.-d). *Pricing - WhatsApp Cloud API*. Meta for Developers. https://developers.facebook.com/docs/whatsapp/pricing
16. Facebook, Inc. (n.d.-e). *Messaging limits - WhatsApp Cloud API*. Meta for Developers. https://developers.facebook.com/docs/whatsapp/messaging-limits
17. WhatsApp LLC. (n.d.). *About group participant limits*. WhatsApp Help Center. https://faq.whatsapp.com/3242937609289432
18. Viber Media S.à r.l. (n.d.). *Broadcast message*. Viber Developer Portal. https://developers.viber.com/docs/api/rest-bot-api/#broadcast-message
19. NestJS. (n.d.-a). *First steps*. NestJS Documentation. https://docs.nestjs.com/first-steps
20. TypeORM. (n.d.). *TypeORM*. TypeORM. https://typeorm.io

21. NestJS. (n.d.-b). *SQL (TypeORM)*. NestJS Documentation. https://docs.nestjs.com/recipes/sql-typeorm
22. React. (n.d.). *React*. React Documentation. https://react.dev/reference/react
23. Next.js. (n.d.). *Pages*. Next.js Documentation. https://nextjs.org/docs/pages
24. React Router. (n.d.). *React Router*. React Router Documentation. https://reactrouter.com/home
25. GitHub. (n.d.). *pmndrs/zustand: Bear necessities for state management in React*. GitHub. https://github.com/pmndrs/zustand/blob/main/README.md
26. Redux. (n.d.). *Getting started with Redux*. Redux Documentation. https://redux.js.org/introduction/getting-started

ДОДАТКИ

Додаток А.

```
async create(dto: CreateMessageRequestDto): Promise<Message> {  Show usages  Dmytro Pelovych *
    const { slug, description, content, channelIds } = dto;

    const channels = await this.notificationChannelsRepository.findBy({
        id: In(channelIds),
    });

    if (channels.length !== channelIds.length) {
        throw new Error(MESSAGE_ERROR_MESSAGES.CHANNELS_REQUIRED);
    }

    return this.dataSource.transaction(async manager : EntityManager => {
        const message = manager.create(Message, {
            slug,
            channels,
            description,
            status: MessageStatus.INACTIVE,
        });

        await manager.save(message);

        const revision = await this.messageRevisionsService.create({
            messageId: message.id,
            content,
        }, manager);

        message.currentRevisionId = revision.id;
        message.currentRevision = revision;

        await manager.save(message);

        const createdMessage = await manager.findOne(Message, {
            where: { id: message.id },
            relations: { currentRevision: { content: true }, channels: true },
        });

        if (!createdMessage) {
            throw new Error(MESSAGE_ERROR_MESSAGES.MESSAGE_NOT_FOUND_IN_TRANSACTION);
        }

        return createdMessage;
    });
}
```

Додаток Б.

```
@Controller( prefix: 'notification-providers') Show usages  Dmytro Pelovych
@UseGuards(JwtAuthGuard)
export class NotificationProvidersController {
    constructor(private readonly notificationProvidersService: NotificationProvidersService) {}

    @Get() no usages  Dmytro Pelovych
    @HttpCode(HttpStatus.OK)
    async findAll(): Promise<FindAllProvidersResponseDto> {
        const providers = await this.notificationProvidersService.findAll();

        return { providers };
    }
}
```

Додаток В.

```
@Controller( prefix: 'external-api')  Show usages  Dmytro Pelovych
@UseGuards(ApiTokenAuthGuard)
export class ExternalApiController {
    constructor(private readonly externalApiService: ExternalApiService) {}  no usages  Dmytro Pelovych

    @Post( path: 'dispatch')  no usages  Dmytro Pelovych
    @HttpCode(HttpStatus.OK)
    async dispatchMessageBySlug(
        @Body() dto: DispatchMessageBySlugRequestDto
    ): Promise<DispatchMessageBySlugResponseDto> {
        const result = await this.externalApiService.dispatchMessageBySlug(dto.slug);

        return { result };
    }
}
```

Додаток Г.

```
@Injectable()  Show usages  Dmytro Pelovych
export class DispatchStrategyFactory {
    constructor(  no usages  Dmytro Pelovych
        private readonly discordStrategy: DiscordDispatchStrategy,
        private readonly msTeamsStrategy: MSTeamsDispatchStrategy
    ) {}

    getStrategy(type: NotificationProviderType): DispatchStrategy {  Show usages  Dmytro Pelovych
        switch (type) {
            case NotificationProviderType.DISCORD:
                return this.discordStrategy;
            case NotificationProviderType.MS_TEAMS:
                return this.msTeamsStrategy;
            default:
                throw new Error(DISPATCH_ERROR_MESSAGES.UNSUPPORTED_PROVIDER);
        }
    }
}
```

Додаток Ґ.

```
@Injectable()  Show usages  Dmytro Pelovych
export class DiscordDispatchStrategy implements DispatchStrategy {
    async dispatch(webhookUrl: string, content: MessageContent): Promise<void> {  no usages  Dmytro Pelovych
        const webhook = new WebhookClient({ url: webhookUrl });

        await webhook.send({
            username: 'Notification System',
            embeds: [
                {
                    title: content.title,
                    description: content.body,
                },
            ],
        });
    }
}
```

Додаток Д.

```
@Entity( name: 'notification_channels') Show usages  Dmytro Pelovych *
export class NotificationChannel extends BaseEntity {
    @Column()
    name: string;

    @Column({
        type: 'enum',
        enum: NotificationChannelStatus,
        default: NotificationChannelStatus.INACTIVE,
    })
    status: NotificationChannelStatus;

    @Column({ type: 'jsonb' })
    config: Record<string, unknown>;

    @Column({ type: 'uuid', name: 'provider_id' })
    providerId: string;

    @ManyToOne(() => NotificationProvider, provider : NotificationProvider  => provider.channels)
    @JoinColumn({ name: 'provider_id' })
    provider: NotificationProvider;

    @ManyToMany(() => Message, message : Message  => message.channels)
    messages: Message[];

    @BeforeInsert()  no usages  Dmytro Pelovych *
    validateConfig() {
        switch (this.provider.integrationType) {
            case NotificationProviderIntegrationType.WEBHOOK:
                if (!WEBHOOK_PROVIDER_INTEGRATION_TYPE_CONFIG_VALIDATION_RULE.validate(this.config)) {
                    throw new Error(WEBHOOK_PROVIDER_INTEGRATION_TYPE_CONFIG_VALIDATION_RULE.errorMessage);
                }
            // case NotificationProviderIntegrationType.SOMETHING_ELSE:
            //     if (!SOMETHING_ELSE_VALIDATION_RULE.validate(this.config)) {
            //         throw new Error(SOMETHING_ELSE_VALIDATION_RULE.errorMessage)
            //     }
        }
    }
}
```

Додаток Е.

```
interface NotificationChannelConfigValidationRule {  Show usages  Dmytro Pelovych
    validate: (config: Record<string, unknown>) => boolean;
    errorMessage: string;
}

export const WEBHOOK_PROVIDER_INTEGRATION_TYPE_CONFIG_VALIDATION_RULE: NotificationChannelConfigValidationRule = {
    validate: config : Record<string, unknown>  => (
        'webhookUrl' in config
        && typeof config.webhookUrl === 'string'
    ),
    errorMessage: 'Config must be defined with "webhookUrl"',
};
```

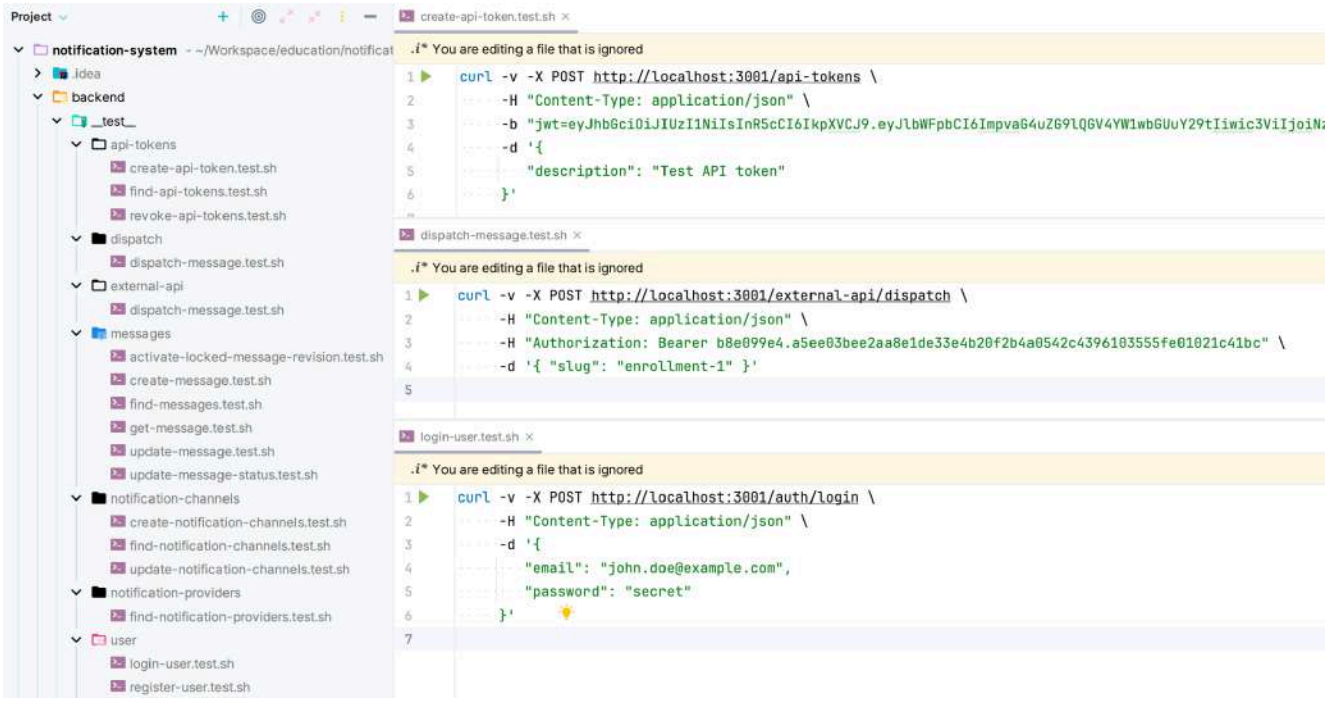
Додаток Є.

```
export class ApiTokensService extends BaseApiService {  Show usages  Dmytro Pelovych
    async create(data: CreateApiTokenRequest) {  Show usages  Dmytro Pelovych
        return this.api.post<CreateApiTokenResponse>(API_CONFIG.ENDPOINTS.API_TOKENS.CREATE, data);
    }

    async list() {  Show usages  Dmytro Pelovych
        return this.api.get<{ tokens: ApiToken[] }>(API_CONFIG.ENDPOINTS.API_TOKENS.LIST);
    }

    async revoke(id: string) {  Show usages  Dmytro Pelovych
        return this.api.delete(API_CONFIG.ENDPOINTS.API_TOKENS.REVOKE(id));
    }
}

export const apiTokensService = new ApiTokensService();  Show usages  Dmytro Pelovych
```

Додаток Ж.

```tsx
export const ProtectedRoute: FC<Props> = ({ children } : { children?: ReactNode } ) => {
    const location = useLocation();

    const user = useAuthStore(s : AuthState  => s.user);
    const isAuthenticated = useAuthStore(s : AuthState  => s.isAuthenticated);

    if (!isAuthenticated || !user) {
        if (location.pathname === '/login') {
            return <>{children}</>;
        }

        return <Navigate to="/login" state={{ from: location }} replace />;
    }

    return <>{children}</>;
}
```

Додаток З.


Project
notification-system - ~/Workspace/education/notificat
.idea
backend
__test__
api-tokens
create-api-token.test.sh
find-api-tokens.test.sh
revoke-api-tokens.test.sh
dispatch
dispatch-message.test.sh
external-api
dispatch-message.test.sh
messages
activate-locked-message-revision.test.sh
create-message.test.sh
find-messages.test.sh
get-message.test.sh
update-message.test.sh
update-message-status.test.sh
notification-channels
create-notification-channels.test.sh
find-notification-channels.test.sh
update-notification-channels.test.sh
notification-providers
find-notification-providers.test.sh
user
login-user.test.sh
register-user.test.sh
create-api-token.test.sh
.i* You are editing a file that is ignored
curl -v -X POST http://localhost:3001/api-tokens \
-H "Content-Type: application/json" \
-b "jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImpvaG4uZG9lQGV4YW1wbGUuY29tIiwic3ViIjoiN
-d '{
"description": "Test API token"
}'
dispatch-message.test.sh
.i* You are editing a file that is ignored
curl -v -X POST http://localhost:3001/external-api/dispatch \
-H "Content-Type: application/json" \
-H "Authorization: Bearer b8e099e4.a5ee03bee2aa8e1de33e4b20f2b4a0542c4396103555fe01021c41bc" \
-d '{ "slug": "enrollment-1" }'
login-user.test.sh
.i* You are editing a file that is ignored
curl -v -X POST http://localhost:3001/auth/login \
-H "Content-Type: application/json" \
-d '{
"email": "john.doe@example.com",
"password": "secret"
}'