Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота

за освітнім ступенем «бакалавр»

на тему:

ЗАСТОСУНОК ДЛЯ ГРУПОВИХ СПОВІЩЕНЬ КОРИСТУВАЧІВ

Виконав:
здобувач IV курсу
групи ПМ-41
спеціальності 113 «Прикладна математика»
Дмитро Богданович Фінчук
Науковий керівник:
старший викладач Тетяна Анатоліївна Кирик

м. Рівне, 2026 рік

ЗМІСТ

ВСТУП

Актуальність теми дослідження. В умовах глобальної цифровізації та переходу вищих навчальних закладів на дистанційні та змішані форми навчання, питання ефективної комунікації між адміністрацією та студентами набуває критичного значення. Традиційні засоби, такі як дошки оголошень або усні повідомлення на лекціях, втратили ефективність або не забезпечують необхідного рівня оперативності. Використання виключно електронної пошти часто демонструє низьку оперативність, тоді як месенджери забезпечують миттєву доставку, але мають ризик втрати інформації у загальних чатах. Розробка мультиканальної системи, що поєднує можливості Telegram та Email, є актуальним завданням для забезпечення гарантованого інформування учасників освітнього процесу.

Мета дослідження полягає у дослідженні принципів побудови мультиканальних інформаційних систем та розробці програмного застосунку для автоматизації групових сповіщень користувачів із використанням Telegram та Email каналів зв’язку.

Об’єкт дослідження — процес автоматизації інформаційної взаємодії в освітньому середовищі.

Предмет дослідження — алгоритми, методи та програмні засоби реалізації мультиканальних систем групового сповіщення користувачів на базі середовища Node.js [1] або методи, алгоритми та програмні засоби побудови мультиканальних систем автоматизованого інформування користувачів із використанням Telegram API [2] та Gmail API [3].

Завдання дослідження:

1. Проаналізувати сучасний стан та тенденції розвитку систем автоматизованого сповіщення.
2. Обґрунтувати вибір технологій (Node.js, Telegraf, Google APIs) для реалізації проекту.
3. Спроектувати архітектуру бази даних та алгоритм розсилки «Smart Mode».
4. Розробити програмний продукт та провести його комплексне тестування.

Потрібно додати - див методичку, практичне значення роботи, апробацію дослідження, структуру роботи

Методи дослідження. Для вирішення поставлених завдань у роботі застосовано: методи системного аналізу (для дослідження предметної області та формулювання вимог), принципи об'єктно-орієнтованого та подійно-орієнтованого програмування (при написанні вихідного коду сервера), теорію проектування баз даних [4] NoSQL (при організації файлового сховища JSON-формату)[5], а також методи тестування (для верифікації працездатності готового рішення).

Практичне значення дослідження. Практичне значення одержаних результатів полягає у розробці програмного застосунку для автоматизації групових сповіщень користувачів, який може бути використаний у закладах освіти для оперативного інформування студентів та викладачів через Telegram і електронну пошту. Розроблена система забезпечує централізоване керування розсилками, підвищує оперативність передачі інформації та мінімізує ризик втрати важливих повідомлень.

Апробація та впровадження результатів дослідження. Основні теоретичні та практичні результати дослідження були представлені на XIX Всеукраїнській науково-практичній конференції здобувачів вищої освіти і молодих учених «Наука, освіта, суспільство очима молодих» (м. Рівне, 14 травня 2026 р.), а також на звітній науковій конференції викладачів, співробітників і здобувачів вищої освіти Рівненського державного гуманітарного університету (м. Рівне, 14 травня 2026 р.).

Структура роботи. Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. Перший розділ містить аналіз предметної області сучасних систем групового інформування, обґрунтування функціональних і технічних вимог до програмного застосунку, а також формалізовану постановку задачі. У другому розділі представлено проектування архітектури застосунку, розроблено концептуальну та логічну структуру NoSQL сховища даних NeDB, а також побудовано UML-діаграми

взаємодії компонентів системи. Третій розділ присвячено програмній реалізації застосунку в середовищі Node.js, опису структури модулів, налаштуванню авторизації через Google OAuth 2.0 та конфігуруванню середовища виконання. У четвертому розділі описано процес тестування розробленого програмного застосунку, наведено результати перевірки роботи мультиканальної системи розсилки та проаналізовано коректність функціонування основних програмних модулів.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ РОЗРОБКИ

1.1. Стан, тенденції та порівняльний аналіз сучасних засобів і систем групового інформування

В умовах стрімкого розвитку інформаційного суспільства, ефективність функціонування будь-якої організаційної, адміністративної чи корпоративної структури безпосередньо залежить від швидкості, надійності та якості її внутрішніх і зовнішніх комунікаційних каналів. Проблема своєчасного та гарантованого інформування персоналу, клієнтів, партнерів або учасників проектів сьогодні постає достатньо гостро. Це зумовлено глобальним явищем, яке у науковій літературі отримало назву «інформаційне перевантаження» (information overload) або «інформаційний шум». Сучасна людина щодня отримує сотні сповіщень із різноманітних джерел, через що виникає стійкий психологічний ефект «інформаційної сліпоти», коли критично важливі повідомлення ігноруються або втрачаються серед маси другорядного контенту[6].

Традиційні методи передачі інформації, які тривалий час вважалися стандартами (зокрема паперові оголошення, внутрішні телефонні дзвінки, усні розпорядження чи статичні дошки новин у веб-браузерах), у сучасних умовах розподіленої та гібридної роботи демонструють свою низьку ефективність. Вони не забезпечують необхідного рівня оперативності доставки повідомлень в умовах динамічної зміни контексту. Як наслідок, виникає об'єктивна потреба в автоматизації комунікаційних процесів та впровадженні інтелектуальних програмних систем масового сповіщення.

На сучасному ринку програмного забезпечення існує кілька базових парадигм та інструментів, що використовуються для організації групових розсилок та сповіщень. Для визначення їхніх фундаментальних архітектурних переваг та недоліків проведено порівняльний аналіз найпопулярніших платформ, результати якого систематизовано та наведено далі:

Електронна пошта є одним із найпоширеніших засобів офіційної цифрової комунікації. Основними протоколами та архітектурними підходами, що використовуються для її функціонування, є SMTP, IMAP та REST API. Перевагами електронної пошти є висока юридична та документальна значущість повідомлень, можливість довготривалого архівування даних, а також підтримка структурованих форматів повідомлень. Водночас електронна пошта має доволі низький коефіцієнт відкриття(у середньому 15–25%) та низьку доставку повідомлень (від 10 секунд до кількох хвилин). Крім того, існує ризик потрапляння листів до спам-фільтрів через суворі політики поштових провайдерів. Автоматизація процесів розсилки має середній рівень складності та часто потребує використання додаткових SDK, систем черг повідомлень і механізмів авторизації [7].

Корпоративні месенджери (Slack / Teams) використовують архітектури на основі WebSockets і REST API. Такі системи забезпечують високий рівень відкриття повідомлень у межах активної сесії користувача та характеризуються високою швидкістю доставки повідомлень, яка зазвичай становить 1–2 секунди. Водночас автоматизація взаємодії з такими платформами є відносно складною через багаторівневу модель керування правами доступу та необхідність інтеграції з корпоративною інфраструктурою. Крім того, використання подібних платформ часто супроводжується фінансовими витратами на ліцензування та адміністрування.

Публічні месенджери, серед яких одним із найбільш поширених є Telegram, використовують протоколи HTTPS, JSON та механізми Long Polling. Telegram Bot API забезпечує швидку доставку повідомлень, підтримку Push-сповіщень і високий рівень взаємодії користувачів із повідомленнями. На відміну від двох попередніх має такі переваги як коефіцієнт відкриття від до 85–90% (через Push), та швидкість доставки менше секунди. Важливою перевагою є відносно невисока складність програмної автоматизації завдяки наявності відкритого Bot API. Крім цього, ризик блокування повідомлень є низьким за умови добровільної авторизації користувача в боті. Водночас такі системи мають нижчу юридичну та документальну

значущість порівняно з електронною поштою, оскільки переважно орієнтовані на оперативне інформування, а не на офіційне документування комунікації.

Проведений аналіз показує, що жоден із існуючих каналів комунікації не здатний повністю забезпечити всі вимоги до ефективної системи інформування. Електронна пошта, попри свою офіційність, структурованість, підтримку HTML-шаблонів та можливість довготривалого архівного зберігання повідомлень, характеризується недостатньою оперативністю взаємодії з користувачами. Перевірка поштової скриньки у більшості випадків має періодичний характер і здійснюється лише кілька разів протягом дня, що ускладнює використання Email для повідомлень про термінові зміни, аварійні ситуації або форс-мажорні обставини. Додатковою проблемою є суворі механізми фільтрації сучасних поштових сервісів, зокрема Gmail та Outlook, через які важливі адміністративні повідомлення можуть помилково потрапляти до папки «Спам» [8].

З іншого боку, корпоративні платформи, такі як Slack або Microsoft Teams забезпечують високу швидкість взаємодії та чудові інструменти для командної роботи. Проте подібні системи мають певні обмеження. Їх використання вимагає від користувача постійного перебування в активній робочій сесії, а ліцензійна політика цих продуктів часто є фінансово витратною для розгортання в організаціях середнього та малого масштабу.

Найбільш динамічним та перспективним напрямом інструментів для побудови автоматизованих систем сповіщення є публічні месенджери, серед яких одним із найбільш популярних є Telegram завдяки відкритій платформі Telegram Bot API. Даний канал працює за принципом асинхронних Push-технологій, що дозволяє оперативно доносити інформацію до кінцевого споживача [2]. Мобільний пристрій користувача майже завжди знаходиться в режимі реального часу (Real-time), що забезпечує показник Open Rate на рівні 90%. Бот в інтерфейсі месенджера сприймається користувачем нативно і не викликає відторгнення, яке часто виникає при отриманні класичних рекламних SMS-повідомлень.

Проте, аналізуючи дані, що представлені раніше, стає очевидним інший ризик: використання виключно Telegram як єдиного вікна інформування може призвести до втрати даних у разі випадкового видалення діалогу користувачем, відсутності стабільного мобільного інтернету в момент відправки Push-сигналу або технічного чи юридичного блокування аккаунта бота.

Таким чином, на основі проведеного теоретичного аналізу стану предметної області, можна зробити фундаментальний науковий висновок: створення надійної, відмовостійкої та ефективної системи групових сповіщень можливе лише за умови відмови від одноканальних архітектур на користь концепції мультиканальності. Об'єднання інформування через Telegram Bot API та офіційного дублювання/архівування через сервіси електронної пошти дозволяє побудувати систему з високим рівнем покриття аудиторії, де недоліки одного технологічного каналу повністю компенсуються архітектурними перевагами іншого.

1.2. Дослідження інформаційних потреб цільової аудиторії та моделювання комунікаційних процесів

Для проектування ефективної архітектури програмного комплексу автоматизованого інформування необхідно провести комплексне дослідження та формалізацію інформаційних потреб кінцевих користувачів. У межах даної кваліфікаційної роботи цільова аудиторія застосунку класифікується на дві основні основні групи користувачів: адміністратори системи (менеджери, керівники проектів, оператори розсилок, адміністративний персонал) та кінцеві отримувачі інформації (співробітники організацій, члени проектних команд, клієнти або лінійні виконавці). Кожна з цих груп висуває специфічні вимоги до функціональності, швидкодії та надійності каналів дистрибуції контенту.

Процес управління сучасними розподіленими командами та організаційними структурами характеризується високою динамічністю. Керівнику або адміністратору щодня доводиться координувати велику кількість бізнес-процесів,

що вимагає розсилки повідомлень різного ступеня пріоритетності. Результати дослідження вимог аудиторії та специфіки повідомлень:

1. Критичні/Форс-мажорні сповіщення

(аварії, зміни графіків, термінові збори)

- Ступінь терміновості: Максимальний (Миттєва реакція)
- Цільова група отримувачів: Усі співробітники / Увесь персонал
- Пріоритетний технологічний канал**:** Публічний месенджер (Telegram Push)

2. Оперативні робочі завдання

(дедлайни проектів, поточні оголошення)

- Ступінь терміновості: Середній (Реакція протягом робочого дня)
- Цільова група отримувачів: Окремі проектні групи / Департаменти
- Пріоритетний технологічний канал**:** Мультиканальний режим (Telegram + Email)

3. Офіційна звітність та документація

(накази, інструкції, договори, звіти)

- Ступінь терміновості: Низький (Потреба в архівації)
- Цільова група отримувачів: Конкретні посадові особи / Менеджмент
- Пріоритетний технологічний канал: Електронна пошта (Gmail API)

4. Планові інформаційні розсилки

(новини компанії, дайджести, опитування)

- Ступінь терміновості: Низький (Ознайомчий характер)
- Цільова група отримувачів: Широке коло користувачів
- Пріоритетний технологічний канал: Електронна пошта (HTML-листи)

Як можна чітко побачити із наведеної інформації, вимоги користувачів до отримання інформації безпосередньо залежать від її функціонального призначення. Для першої категорії — критичних сповіщень — ключовим фактором є мінімізація часової затримки між відправкою повідомлення адміністратором та його відображенням на екрані мобільного пристрою отримувача. Будь-яка затримка у цьому процесі може негативно впливати на координацію організаційних процесів

та своєчасність прийняття рішень. У цьому контексті аудиторія висуває вимогу оперативності: сповіщення має надходити у додаток, який користувач відкриває найчастіше протягом дня. Це обґрунтовує доцільність використання Telegram Bot API як одного з основних засобів доставки критичних сповіщень завдяки його підтримці Push-механізмів та високій швидкості доставки повідомлень.

З іншого боку, для офіційної та звітної документації пріоритетними є інші критерії, зокрема надійність зберігання, структурованість та можливість довготривалого архівування інформації. Такі дані часто втрачаються в потоках повідомлень месенджерів і потребують більш організованого середовища зберігання. У цьому випадку електронна пошта забезпечує необхідні можливості для структурування кореспонденції, роботи з вкладеннями (документи, таблиці, PDF-файли) та подальшого пошуку інформації через значний проміжок часу.

Для групи адміністраторів системи ключовою потребою є зниження рутинного навантаження при організації масових розсилок. Традиційний процес, який передбачає ручне копіювання тексту, вибір груп користувачів у різних системах, введення адрес електронної пошти та надсилання повідомлень окремо в кожен канал, є малоефективним та підвищує ймовірність виникнення помилок, пов’язаних із людським фактором. У зв’язку з цим адміністратори потребують єдиного інтерфейсу керування, що дозволяє виконувати розсилку в межах однієї операції.

Додатковою вимогою з боку управлінського персоналу є гнучка сегментація аудиторії. Система не повинна обмежуватися лише режимом масової трансляції повідомлень усім користувачам одночасно. Необхідно забезпечити логічний розподіл бази даних на окремі сутності — групи, департаменти, проєктні команди або відділи. Це дозволить реалізувати концепцію таргетованого інформування, коли повідомлення отримують лише ті особи, яких безпосередньо стосується дана інформація, що додатково знижує рівень інформаційного перевантаження в межах організації.

Крім функціональних вимог, у процесі дослідження інформаційних потреб користувачів було виявлено підвищені вимоги до безпеки та конфіденційності даних. Користувачі не готові надавативої особисті паролі від корпоративних поштових скриньок стороннім застосункам. Тому розроблювана система повинна інтегруватися з поштовими сервісами на рівні сучасних безпечних протоколів авторизації, які захищають облікові дані.

Таким чином, на основі детального дослідження інформаційних потреб обох груп цільової аудиторії, було сформовано узагальнене концептуальне бачення майбутнього програмного продукту. Застосунок має функціонувати як гнучкий мультиканальний інструмент інформування, що поєднує швидкість месенджера та надійність електронної пошти, керується через єдиний зручний інтерфейс адміністратора, підтримує динамічну сегментацію користувачів за групами та забезпечує базовий рівень захисту даних. Побудова системи на основі цих виявлених вимог дозволить створити затребуваний та ефективний інструмент для автоматизації сучасних комунікаційних процесів у будь-якій організаційній структурі.

1.3. Обґрунтування технічних вимог та формалізована постановка задачі проектування системи

На основі проведеного аналізу існуючих технологічних рішень та дослідження інформаційних потреб різних груп цільової аудиторії виникає необхідність декомпозиції, структурування та формування системи технічних вимог до проєктованого програмного комплексу. Специфікація вимог є базою для подальшого проектування архітектури системи, розробки алгоритмів, структури бази даних та реалізації програмного коду. Без чітко визначених вимог до системи неможливо побудувати надійну архітектуру, стійку до навантажень.

Усі архітектурні, експлуатаційні та користувацькі вимоги до мультиканальної системи сповіщень доцільно розділити на дві класичні категорії: функціональні (які описують конкретну поведінку застосунку та дії, що він має

виконувати) та нефункціональні (які висувають вимоги до продуктивності, безпеки, масштабованості та надійності середовища розгортання). Для забезпечення системного підходу до розробки, повний перелік та класифікацію технічних вимог детально наведено в табл. А1.1. в додатках.

Як можна побачити із наведеної табл. А1.1 з додатків, архітектурна модель застосунку базується на принципах гнучкості, асинхронності та безпеки. Аналізуючи дані, що представлені в табл. 1.1, особливу увагу приділено вимозі NFR_02. Використання протоколу OAuth 2.0 Playground для інструментів Gmail API дозволяє програмі оперувати тимчасовими токенами доступу (Access Token) та оновлення (Refresh Token) замість збереження паролів адміністратора в текстових конфігураціях, що повністю нівелює ризик компрометації облікових даних[9].

Для забезпечення високої відмовостійкості (fault tolerance) системи при взаємодії із зовнішніми шлюзами, необхідно окремо формалізувати вимоги до обробки виняткових ситуацій (exception handling). Оскільки система мультиканальних розсилок залежить від стабільності зовнішніх API та мережевої інфраструктури, програмний комплекс повинен забезпечувати коректне реагування на помилки без повної зупинки процесу розсилки.

Однією з можливих виняткових ситуацій є блокування Telegram-бота користувачем. У такому випадку Telegram API повертає помилку типу HTTP 403. Для обробки цієї ситуації система виконує фіксацію помилки в журналі подій та здійснює спробу доставки повідомлення через альтернативний канал електронної пошти.

Іншим можливим сценарієм є завершення терміну дії токена доступу під час роботи з Gmail API. У цьому випадку сервер Google Cloud API повертає помилку HTTP 401. Для забезпечення безперервності роботи система автоматично виконує процедуру оновлення токена доступу за допомогою механізму refresh token без переривання процесу розсилки.

У разі тимчасової втрати мережевого з’єднання система переходить у режим повторних спроб (Retry Pattern). Після виникнення помилки виконується затримка та повторна спроба встановлення з’єднання через визначений проміжок часу.

Окрему категорію становлять помилки, пов’язані з пошкодженням локального файлу бази даних. У такому випадку система формує запис про критичну помилку в журналі подій та виконує резервне збереження службової інформації для подальшого аналізу причин збою.

Аналізуючи наведені механізми обробки виняткових ситуацій, можна зробити висновок, що розроблена система не обмежується лише передачею повідомлень, а також реалізує базові механізми автоматичного відновлення роботи при виникненні окремих типів помилок. Це дозволяє зменшити вплив технічних збоїв на процес розсилки та підвищити стабільність доставки повідомлень користувачам навіть у випадках недоступності окремих каналів зв’язку.

З формальної точки зору задачу розробки системи мультиканальних сповіщень можна представити у вигляді моделі інформаційного обміну між множиною користувачів та каналами доставки повідомлень [10]. Оскільки об’єктом автоматизації є процес розподілу інформації між отримувачами, систему доцільно описати за допомогою елементів теорії множин. Нехай

$$U=\{u_i,u_2,.....u_n\} \quad (1.1)$$

— множина користувачів, які зареєстровані у внутрішній базі даних NeDB, де кожен окремий користувач описується вектором параметрів:

$$u_i=<id_i,\text{tg\_}id_i,email_i,role_i,> \quad (1.2)$$

де:

id_i — внутрішній ідентифікатор користувача;

$\text{tg\_}id_i$ — ідентифікатор користувача в Telegram;

$email_i$ — адреса електронної пошти;

$role_i$ — роль або група користувача в системі.

Задача проектування алгоритму розподілу «Smart Mode» полягає в реалізації функції маршрутизації повідомлень *F(M,u_i)*, яка для кожного вхідного повідомлення *M* та користувача *ui* (визначає оптимальний канал доставки повідомлення. Множину каналів доставки можна подати у вигляді:

$$K = \{K_{tg}, K_{mail}\}, \tag{1.3}$$

де:

K_{tg} — канал доставки через Telegram;

K_{mail} — канал доставки через електронну пошту.

Вибір каналу доставки виконується відповідно до наявності контактних даних користувача та доступності відповідного сервісу. У випадку виникнення помилки доставки система виконує реєстрацію події у журналі логування без переривання загального процесу розсилки. Формалізована модель вибору каналу доставки повідомлення має вигляд:

$$F(M, u_i) = \begin{cases} K_{tg}, & tg\_id_i \neq \varnothing \\ K_{mail}, & email_i \neq \varnothing \\ LogError(u_i), & \text{інакше} \end{cases} \tag{1.4}$$

Де функція{Log_Error}(u_i)є функцією реєстрації помилки доставки для користувача u_i у журналі системних подій та відповідає за генерацію виключної ситуації та її фіксацію у технічному файлі логів без зупинки загального потоку ітерації циклу розсилки.

Критерієм ефективності розробленої системи є мінімізація часових витрат адміністратора на ініціалізацію процесу комунікації та підвищення надійності доставки повідомлення до кінцевого отримувача за рахунок дублювання каналів зв'язку. Програма повинна забезпечити упевнену роботу в умовах нестабільного зв'язку та надавати зручні інструменти для ведення статистики. Таким чином,

успішна реалізація сформульованих вимог дозволить створити програмний засіб для автоматизації процесів групового інформування з підтримкою мультиканальної доставки повідомлень, базових механізмів відмовостійкості та захищеної взаємодії із зовнішніми сервісами.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА ОБҐРУНТУВАННЯ АРХІТЕКТУРИ (ЗАСТОСУНКУ)

2.1. Загальна архітектурна модель та дворівнева структура системи

Проектування архітектури системи мультиканальних сповіщень та розсилок вимагає комплексного підходу, оскільки такий програмний застосунок повинен стабільно працювати в умовах постійного мережевого обміну даними з віддаленими хмарними сервісами. Основним завданням під час проектування архітектури системи є забезпечення модульності, слабкої пов’язаності компонентів (loose coupling) та розподілу функціональних обов’язків між окремими модулями системи (Separation of Concerns) [11]. Це означає, що внутрішня логіка збереження інформації, бізнес-правила обробки черг розсилки та низькорівневі транспортні шлюзи взаємодії із зовнішніми API мають бути повністю ізольовані один від одного.

Для реалізації визначених вимог було спроектовано, обґрунтовано та реалізовано дворівневу архітектурну модель. Ця модель складається з двох рівнів: зовнішнього інтерфейсного шлюзу (User/Administrator Interface Layer) та внутрішнього серверного ядра (Core Application Layer), що функціонує в асинхронному середовищі виконання Node.js.

Перший рівень системи — інтерфейс взаємодії з адміністратором, реалізований на базі Telegram Bot API. Традиційний підхід до створення подібних систем зазвичай передбачає розробку окремої веб-панелі адміністратора (веб-сайту), що тягне за собою необхідність верстки інтерфейсів, налаштування CORS-політик, підтримки сесій браузера та створення додаткових систем захисту від міжсайтового скриптингу. Натомість, інтеграція робочого простору адміністратора безпосередньо в інтерфейс чат-бота дозволяє уникнути необхідності розробки окремої вебпанелі адміністратора. Оператор (адміністратор) отримує можливість керувати процесами сегментації бази даних, створювати нові повідомлення та запускати масові розсилки з будь-якого мобільного пристрою чи стаціонарного

комп'ютера, де встановлено клієнт месенджера. Цей рівень є точкою входу, яка відповідає за збір текстового або медійного контенту, валідацію дій оператора на наявність помилок та відображення інтерактивного меню керування.

Другий рівень — це серверне ядро застосунку, розгорнуте на локальній або віртуальній машині (сервері розгортання) під керуванням платформи Node.js. Серверний рівень виконує функції обробки даних та реалізації бізнес-логіки системи. Саме тут зосереджена вся бізнес-логіка: модуль аналізу та координації потоків «Smart Mode», вбудований безсерверний рівень персистентного збереження даних NeDB, а також ізольовані транспортні клієнти для прямого зв'язку з віддаленими серверами Google Cloud (сервіс Gmail) та Telegram Server.

Взаємодія між цими рівнями та зовнішніми хмарними системами реалізована відповідно до принципів подійно-орієнтованої архітектури (Event-Driven Architecture) [12]. Для детальної візуалізації того, як саме розподіляються потоки даних усередині програмного комплексу, наведено структурну схему розробленої архітектури.

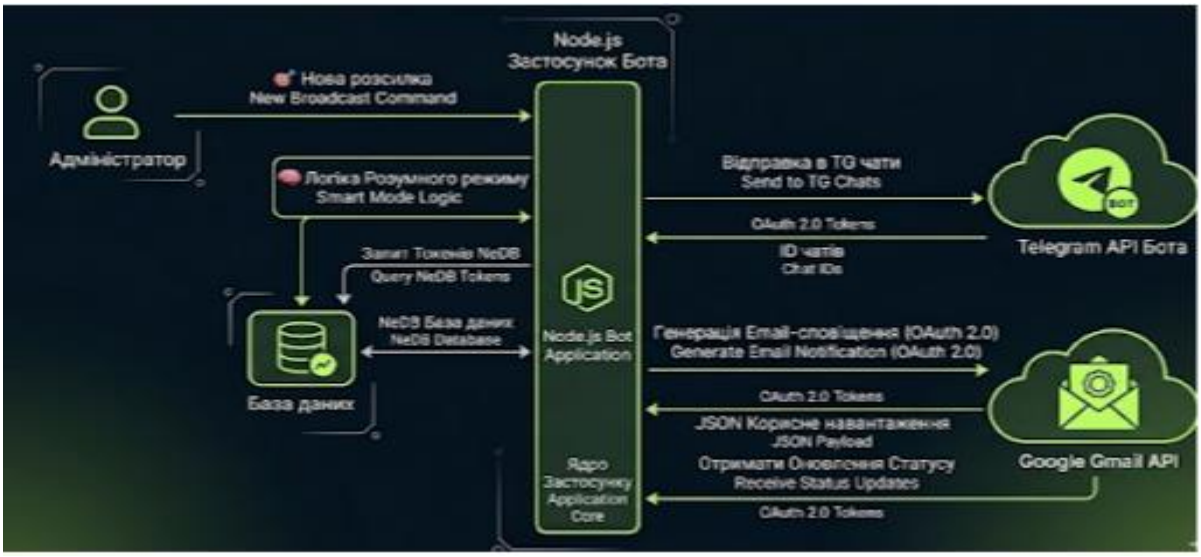

Рисунок. 2.1. Структурна схема архітектури та інформаційних потоків мультиканальної системи сповіщень

Як можна детально простежити за наведеною структурною схемою архітектури (рис. 2.1), серверне ядро Node.js виступає в ролі центрального комутатора повідомлень .Процес розсилки ініціюється адміністратором через

інтерфейс Telegram-бота шляхом передачі команди на створення нової розсилки («New Broadcast Command»).

Після отримання команди ядро застосунку переходить у стан активної обробки події. А саме звертається до вбудованої бази даних NeDB для виконання двох послідовних операцій: зчитування конфігураційних токенів та вибірки масиву ідентифікаторів чатів користувачів, які належать до обраної цільової групи. Отримавши ці структуровані дані з файлового сховища, асинхронний двигун Node.js розгалужує інформаційний потік на два паралельні транспортні канали.

Перший транспортний потік (верхній вектор на рис. 2.1) спрямовується до офіційного хмарного шлюзу Telegram API за допомогою методу відправки повідомлень, що дозволяє доставити інформацію кінцевим отримувачам у їхні мобільні додатки. Одночасно з цим, другий транспортний потік (нижній вектор на рис. 2.1) взаємодіє з інфраструктурою Google Cloud. Серверне ядро використовує заздалегідь згенеровані авторизаційні маркери OAuth 2.0 для безпечного підключення до шлюзу Gmail API. Після успішного проходження перевірки прав доступу, шлюз Google приймає сформований пакет даних, здійснює трансляцію офіційного Email-листа та надсилає його на поштові скриньки тих самих користувачів [13].

Проєктування системи на основі дворівневої модульної архітектури (рис. 2.1) надає розробнику переваги у контексті гнучкості та супроводу програмного продукту. Оскільки транспортні шлюзи для роботи з Telegram та Gmail реалізовані у вигляді ізольованих програмних модулів, будь-які технічні зміни на стороні компанії Google або оновлення політики Telegram не порушать працездатність усього ядра застосунку. Крім того, така архітектура спрощує подальше розширення системи. Якщо виникне необхідність додати нові канали доставки повідомлень (наприклад, SMS-сервіси, Push-повідомлення у веб-браузерах або шлюзи інших месенджерів), достатньо реалізувати відповідний ізольований транспортний адаптер, не переписуючи при цьому логіку збереження бази даних чи інтерфейс адміністрування.

2.2. Проектування логічної структури сховища даних та обґрунтування NoSQL архітектури

Важливим етапом проектування системи автоматизованого інформування є побудова раціональної моделі збереження інформації. Сховище даних у такій системі повинно забезпечувати високу швидкість читання, гнучкість структури записів та стабільну роботу при виконанні масових операцій обробки та розсилки повідомлень. На етапі аналізу архітектурних вимог до дата-шару (Data Layer) було обґрунтовано доцільність використання документо-орієнтованого підходу NoSQL замість класичних реляційних систем керування базами даних (таких як MySQL або PostgreSQL). Як практичну реалізацію обрано вбудовану базу даних NeDB на базі бібліотеки @seald-io/nedb [14], яка дозволяє зберігати дані у форматі JSON без необхідності розгортання окремого серверного СУБД.

Вибір саме NoSQL підходу обґрунтований специфікою інформації, якою оперує інтерфейс чат-бота, та динамічною природою профілів користувачів у сучасних комунікаційних середовищах. У реляційних базах даних зміна структури об'єкта (наприклад, додавання нових полів для ідентифікації користувача, збереження масивів з назвами підгруп або міток часових поясів) вимагає виконання міграцій схеми таблиць та часто призводить до появи великої кількості порожніх значень (NULL) [15], якщо у конкретного користувача заповнено лише один із кількох можливих каналів зв’язку.

Натомість, документо-орієнтований підхід NeDB дозволяє зберігати кожен запис у вигляді незалежного, гнучкого JSON-документа. Це дає можливість системі динамічно адаптувати набір полів відповідно до поточного стану профілю кожного об'єкта без додаткових витрат на перевизначення структури всієї бази даних.

Для формалізації зв'язків між інформаційними сутностями та опису логічних рівнів збереження інформації в системі було спроектовано концептуальну ER-діаграму (Entity-Relationship Diagram), яка відображає структуру сховища.

Рисунок. 2.2. Логічна ER-діаграма структури та зв'язків сховища даних системи

Аналізуючи розроблену логічну структуру сховища (рис. 2.2), можна виділити три основні сутності (документи), які забезпечують повний цикл функціонування та аудиту системи розсилки:

1. *Користувачі (Users):* Дана сутність є базовою і містить первинний ключ user_id для унікальної ідентифікації запису в системі. Для забезпечення точної адресації повідомлень у розрізі мультиканальності, в структуру документа інтегровано текстові поля telegram_handle (ідентифікатор для надсилання Push-сповіщень через месенджер), email_name (пряма адреса електронної скриньки для дублювання офіційних листів) та інформаційне поле user_name для відображення реального імені користувача в інтерфейсі адміністратора.
2. *Групи (Groups):* Ця сутність розроблена з метою реалізації механізму динамічної сегментації аудиторії. Вона оперує унікальним ідентифікатором group_id та текстовим полем group_name (назва департаменту, відділу чи проектної команди). Зв'язок із сутністю користувачів реалізується за допомогою масиву ідентифікаторів, що дозволяє організувати гнучкі відношення типу «один до багатьох» або «багато до багатьох» безпосередньо на рівні вкладених документів, уникаючи створення громіздких проміжних таблиць з'єднання, які притаманні SQL-архітектурам.
3. *Логи (Logs):* Службова сутність, яка виконує роль системного журналу відмовостійкості та безпеки. Вона фіксує унікальний ключ log_id, точну

часову мітку події timestamp та безпосередній текстовий вміст технічного повідомлення чи помилки (message), що виникають під час роботи транспортних шлюзів розсилки.

Фізична реалізація спроектованої NoSQL моделі відбувається безпосередньо на диску сервера розгортання у вигляді легковагових текстових файлів з розширенням .db. Кожен новий рядок у такому файлі є окремим, повністю валідним JSON-об'єктом, що містить інформацію про конкретний запис. Для підтвердження коректності практичного втілення логічної схеми сховища, нижче наведено реальне відображення вмісту файлу бази даних, відкритого безпосередньо у середовищі розробки.

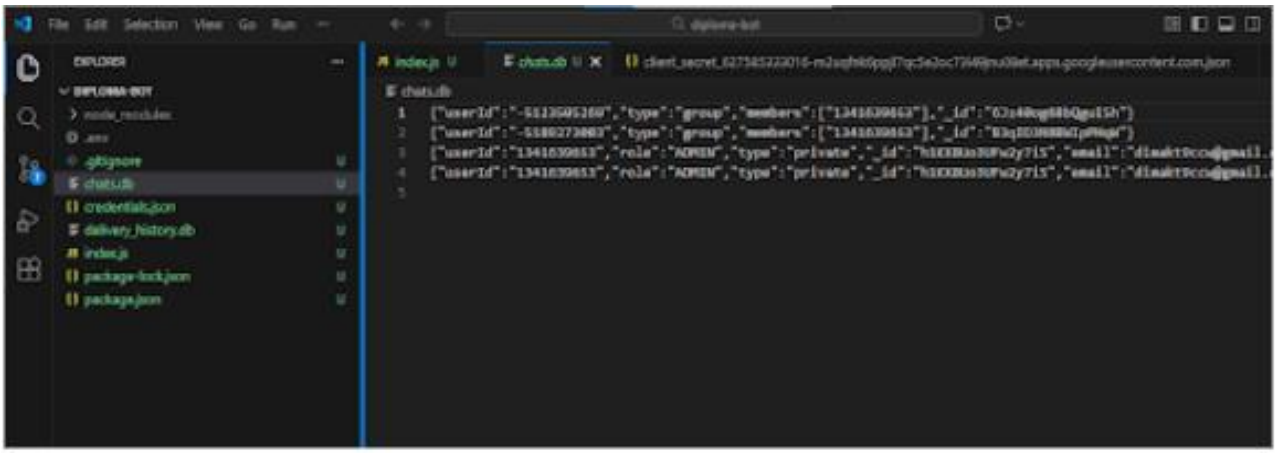

Рисунок. 2.3. Фізична структура збереження JSON-документів у файлі бази даних chats.db у середовищі розробки

Як демонструє скріншот фізичної структури сховища (рис. 2.3), дані в файлі chats.db зберігаються у чистому та структурованому вигляді. Кожен документ містить унікальний ідентифікатор _id, який генерується алгоритмами NeDB автоматично за допомогою криптографічного хешування під час створення запису і виконує роль первинного індексу сховища.

На рисунку 2.3 представлено структуру збереження інформації про групи користувачів (наприклад, документи з полем groupName, де зафіксовані назви цільових каналів інформування), а також службові документи конфігурації

адміністратора, які містять прапорці поточного стану діалогу та масиви ідентифікаторів чату користувача.

При ініціалізації сервера застосунку Node.js, вбудований модуль бази даних викликає метод автоматичного завантаження файлу з диска. Програма повністю зчитує ці JSON-рядки та розгортає їх у вигляді структурованих об'єктів усередині оперативної пам'яті (RAM) сервера. Такий підхід забезпечує перевагу у швидкодії: будь-які подальші операції фільтрації користувачів за групами чи перевірки прав доступу адміністратора під час ініціалізації масової розсилки виконуються в оперативній пам'яті за мікросекунди. Це усуває проблему "вузького місця" (I/O bottlenecks), яка часто виникає при інтенсивному мережевому обміні даними із зовнішніми, важкими серверами баз даних, та забезпечує стабільну роботу локального сервера розгортання в межах заданих технічних вимог.

2.3. Специфікація інтеграційних шлюзів та моделювання протоколу взаємодії з Google Cloud та Telegram API

Оскільки розроблений програмний комплекс функціонує як інтеграційний міст між адміністратором та різнорідними інформаційними каналами, етап проектування зовнішніх інтерфейсів взаємодії (API Gateways) вимагає детального аналізу та моделювання. Застосунок повинен у реальному часі взаємодіяти з двома абсолютно незалежними, глобальними хмарними інфраструктурами: Telegram Bot API та Google Cloud Platform (сервіс електронної пошти Gmail). Кожна з цих платформ висуває індивідуальні вимоги до форматів передачі даних, структур корисного навантаження (JSON Payloads), лімітів на кількість запитів за одиницю часу (Rate Limiting) та протоколів інформаційної безпеки.

Для організації каналу взаємодії з месенджером Telegram на рівні серверного ядра Node.js було обрано сучасний об'єктно-орієнтований фреймворк Telegraf.js. Дана бібліотека виступає в ролі високорівневої абстракції, яка інкапсулює низькорівневі HTTP-запити до віддалених серверів месенджера у зручні асинхронні методи та об'єкти мови JavaScript. Мережеве з'єднання з Telegram API

побудоване за технологією Long Polling. Сервер застосунку ініціює постійне утримання відкритого HTTPS-з'єднання з хмарою Telegram, очікуючи на появу нових подій.

Коли адміністратор натискає інтерактивну кнопку в меню або відправляє текстове повідомлення, Telegram API формує та передає масив даних у чергу оновлень (Updates), яка зчитується сервером Node.js та спрямовується у ланцюжок внутрішніх проміжних обробників (Middleware) для подальшої маршрутизації.

Взаємодія з поштовим сервісом Gmail від Google характеризується вищим рівнем складності архітектурного проєктування, оскільки екосистема Google Cloud висуває строгі вимоги до захисту персональних даних та авторизації сторонніх програмних продуктів. Для вирішення цього завдання було відкинуто застарілі та небезпечні методи аутентифікації за прямим логіном і паролем (які повністю блокуються сучасними поштовими серверами). Інтеграція реалізована через офіційну бібліотеку googleapis на базі міжнародного безпечного протоколу OAuth 2.0 за допомогою інструменту Google OAuth2 Playground.

Архітектурний принцип роботи даного шлюзу полягає в тому, що локальний сервер застосунку не зберігає жодних конфіденційних паролів користувача. Замість цього програма оперує системою зв'язаних маркерів доступу. Для підтримки довготривалої автономної роботи сервера в конфігураційні змінні оточення вноситься заздалегідь згенерований маркер оновлення (Refresh Token). Коли виникає потреба у виконанні розсилки, ядро Node.js використовує цей маркер для короткочасної сесії, надсилаючи запит на сервери авторизації Google, й автоматично отримує тимчасовий токен доступу (Access Token), термін життя якого обмежений 3600 секундами. Саме цей сесійний ключ інтегрується в HTTP-заголовки при використанні методу відправки листів [16].

Для формалізації послідовності викликів функцій, визначення часових інтервалів та моделювання асинхронної взаємодії між компонентами системи було розроблено UML-діаграму послідовності (Sequence Diagram).

Рисунок 2.4. Діаграма послідовності асинхронного протоколу взаємодії компонентів системи при виконанні розсилки

Аналізуючи розроблену діаграму послідовності (рис. 2.4), можнапростежити загальну логіку взаємодії між основними компонентами системи мультиканального інформування. Весь алгоритм взаємодії розділений між п'ятьма архітектурними об'єктами: адміністратором (Admin), головним сервером застосунку (App / Bot), базою даних (Database / NeDB), поштовим шлюзом Google (Gmail API) та серверами месенджера (Telegram API). Весь процес складається з наступних хронологічних етапів:

1. *Ініціалізація та відправка контенту*: На першому кроці адміністратор (Admin) через інтерфейс клієнтського додатка надсилає текстове або медійне навантаження на сервер застосунку (App / Bot). Програма розпізнає команду та переходить у стан формування черги.
2. *Персистентна фіксація:* Отримавши контент, сервер звертається до об'єкта Database / NeDB, викликаючи асинхронну операцію запиту даних. База даних повертає структурований масив профілів користувачів та актуальні конфігураційні параметри шлюзів.
3. *Оновлення авторизаційної сесії OAuth 2.0*: Перед початком відправки електронних листів сервер застосунку ініціює запит до зовнішньої

інфраструктури Gmail API. Передаючи Refresh Token, програма виконує процедуру валідації і отримує у відповідь тимчасовий Access Token. Цей етап на рис. 2.4 позначений як успішне повернення маркерів доступу для поточної сесії.

4. *Асинхронний цикл дистрибуції повідомлень:* Після того як усі компоненти авторизовані, ядро Node.js запускає паралельний конвеєр обробки масиву отримувачів. Для кожного об'єкта з бази даних сервер паралельно викликає метод sendMessage для шлюзу Telegram API та ініціює метод G_Mail_Send (JSON, HTML) для поштового сервера Gmail API. Повідомлення транслюються у два канали одночасно, забезпечуючи максимальне покриття аудиторії.

5. *Логування та фінальна звітність:* Зовнішні хмарні шлюзи обробляють запити і повертають нашому серверу статуси доставки (наприклад, HTTP-код 200 OK). Сервер застосунку аналізує ці відповіді, фіксує результати розсилки та можливі помилки у службовому журналі логів бази даних, після чого генерує підсумковий текстовий звіт і відправляє його у чат адміністратора.

Таке детальне моделювання протоколу інтеграції (рис. 2.4) доводить, що проектована архітектура повністю відповідає критеріям відмовостійкості та високої швидкодії. Завдяки асинхронній природі середовища Node.js, затримки або тимчасові мережеві збої на стороні одного з API-шлюзів не призводять до блокування або зависання всього програмного комплексу, дозволяючи системі успішно завершувати бізнес-процеси інформування користувачів.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1. Структура програмного проекту та конфігурування модульних залежностей середовища розгортання

Перехід від етапу теоретичного проєктування та моделювання архітектури до безпосередньої програмної реалізації комплексу мультиканальних сповіщень вимагає чіткої організації структури програмного коду. Створення стабільного серверного застосунку на базі платформи Node.js передбачає декомпозицію вихідного коду на ізольовані модулі та коректне налаштування дескриптора проекту. Це необхідно для забезпечення легкого розгортання системи на цільовому сервері без ризику виникнення конфліктів версій сторонніх бібліотек та драйверів.

Основним інструментом керування життєвим циклом розробки, автоматизації запусків та контролю зовнішніх модульних залежностей у середовищі JavaScript є менеджер пакетів npm (Node Package Manager). Уся метаінформація про назву проекту, його поточну версію, головні файли входу, ліцензійні обмеження, скрипти автоматизації та перелік використаних фреймворків фіксується у спеціальному конфігураційному файлі package.json, який розташовується в кореневому каталозі програмного застосунку.

Для детального аналізу архітектури побудови коду, підключених інструментів безпеки та шлюзів взаємодії, нижче наведено відображення конфігураційного дескриптора розробленої системи безпосередньо у вікні інтегрованого середовища розробки.

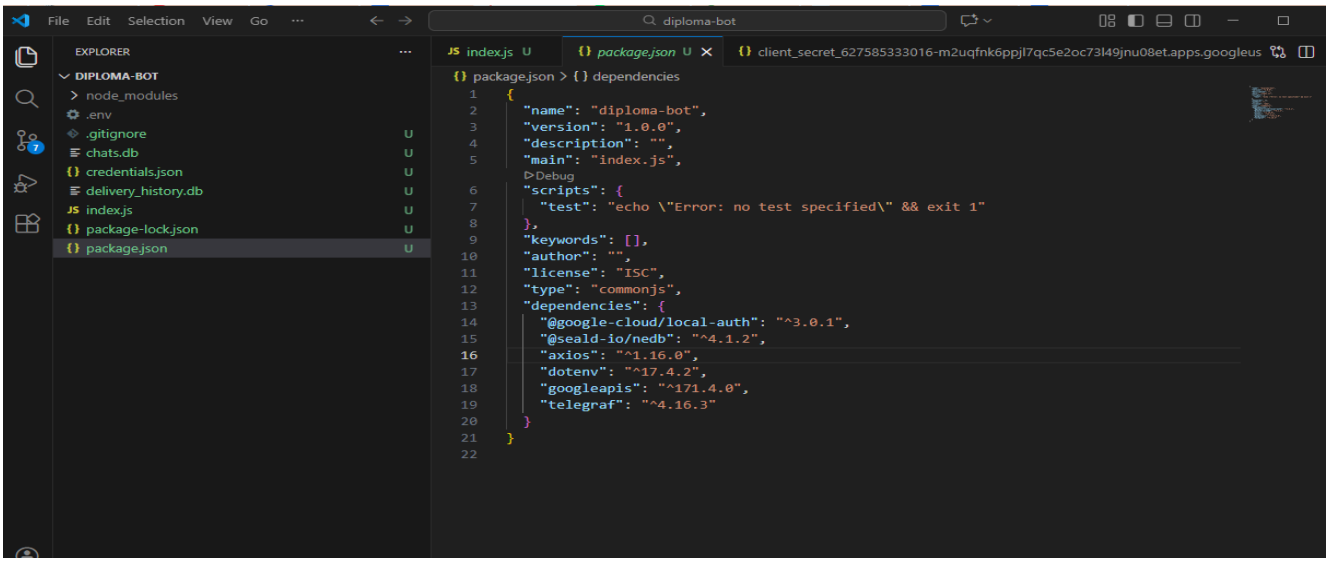

Рисунок 3.1. Скріншот конфігураційного файлу package.json проекту у середовищі розробки

Аналізуючи структуру представленого дескриптора проекту (рис. 3.1), можна визначити архітектурний стек технологій та системні вимоги до середовища розгортання. Проект має технічну назву "telegram-broadcaster", що відображає його функціональне призначення, та відповідає початковій версії версії програмного продукту "1.0.0". Програма використовує сучасний модульний стандарт опису імпортів, про що свідчить прапорець "type": "module". Це дозволяє розробнику застосовувати нативну синтаксичну конструкцію import/export (ES6 Modules) замість застарілих викликів require, оптимізуючи швидкість ініціалізації модулів у пам'яті сервера.

Особливий інтерес для технічного аналізу стабільності системи становить блок "dependencies" (залежності), наведений на рисунку 3.1. Кожна підключена бібліотека виконує чітко визначену роль у загальній архітектурі системи:

1. *googleapis:* Офіційний програмний клієнт від компанії Google, який надає низькорівневий інструментарій для взаємодії з хмарними сервісами. Саме цей модуль відповідає за реалізацію складного протоколу авторизації OAuth 2.0 та безпосередню генерацію, кодування й відправку структурованих JSON-пакетів електронної пошти через віддалений шлюз Gmail API [3,17.].

2. *telegraf:* Високорівневий фреймворк для розробки ботів у месенджері Telegram. Він абстрагує мережеву взаємодію, обробляє вхідні потоки оновлень за принципом Long Polling та надає зручні інструменти для побудови динамічних меню, текстових відповідей та збору контенту від адміністратора [2].
3. *@seald-io/nedb:* Сучасна форкнута (forked) та оптимізована версія класичної вбудованої NoSQL бази даних NeDB. Вона використовується як локальне персистентне сховище документів [14,18]. Ця бібліотека забезпечує виконання асинхронних запитів, індексування профілів користувачів та ведення системних логів, зберігаючи дані у вигляді звичайних текстових файлів.
4. *dotenv:* Допоміжний, але важливий з погляду безпеки модуль. Він автоматично зчитує змінні оточення із захищеного локального файлу .env та імпортує їх у глобальний об'єкт Node.js process.env. Це дозволяє повністю ізолювати секретні ключі, токени ботів та Refresh Tokens від сирцевого коду програми, запобігаючи їх випадковому витоку в публічні репозиторії.

Окрім блоку безпосередніх залежностей, на рисунку 3.1 чітко видно секцію "scripts", яка містить команду автоматизації запуску системи "start": "node index.js". Це вказує на те, що головною точкою входу в застосунок є файл index.js.

При виконанні команди запуску в терміналі сервера, середовище виконання Node.js першочергово звертається до файлу index.js, який запускає каскадний процес ініціалізації: спочатку завантажуються конфігураційні змінні оточення за допомогою dotenv, далі ініціюється підключення та автоматичне завантаження локальних файлів бази даних NeDB, після чого створюється екземпляр чат-бота Telegraf і запускається асинхронний слухач мережевих подій.

Така модульна організація проекту (рис. 3.1) гарантує швидке та безпомилкове розгортання системи на різних операційних системах (Linux, Windows або macOS)[19]. Для запуску всього програмного комплексу адміністратору або розробнику достатньо виконати в консолі базову команду npm install, яка зчитує

дескриптор package.json, встановлює необхідні залежності з репозиторію npm та формує середовище виконання, придатне для подальшої роботи застосунку.

3.2. Програмна реалізація та алгоритмічний аналіз режиму розподілу потоків «Smart Mode»

Ефективність функціонування сучасних систем масового інформування визначається не лише швидкістю доставки повідомлень, а й раціональністю використання обмежень зовнішніх API, оптимізацією навантаження на базу даних та гнучким керуванням каналами комунікації. У межах розробленого програмного застосунку було спроєктовано та реалізовано модуль координації черг, який отримав назву «Smart Mode» (Розумний режим). Основне завдання цього модуля полягає в автоматичному аналізі структури сховища, визначенні типу чату (приватний чи груповий), динамічному формуванні списків отримувачів та виключенні дублювання повідомлень у поштовому та медіа-каналах.

Логіка роботи алгоритму побудована на послідовній перевірці умов та асинхронній обробці подій у межах єдиного конвеєра розсилки повідомлень У процесі виконання система динамічно приймає рішення щодо вибору каналу доставки відповідно до наявних даних користувача та заданих правил маршрутизації. Для формалізації структури роботи модуля та відображення послідовності його виконання розроблено блок-схему алгоритму, яка наведена нижче.

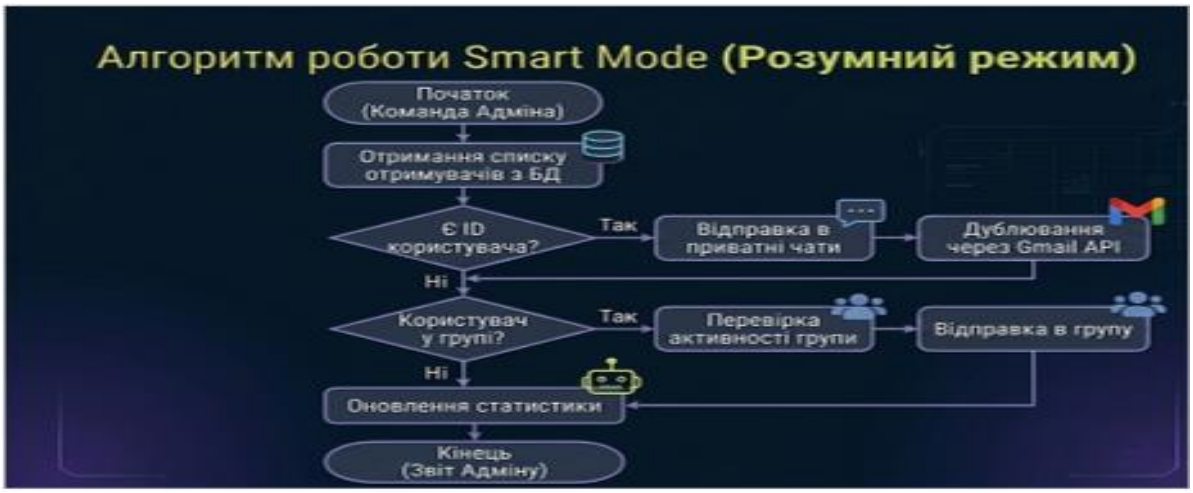

Рисунок. 3.2. Блок-схема алгоритму функціонування режиму «Smart Mode»

Як наочно демонструє розроблена блок-схема (рис. 3.2), життєвий цикл обробки запиту в розумному режимі має чітку ієрархічну структуру і складається з наступних послідовних етапів:

Точка входу та збір метаданих: Алгоритм активується в момент перехоплення події текстового або медійного повідомлення від адміністратора (bot.on(['text', 'photo', ...])), коли внутрішній стан сесії встановлено у значення 'GET_CONTENT'.

Асинхронне зчитування сховища: Програма викликає метод db.find({}, async (err, allChats) => ...), ініціюючи неблокуюче читання всього масиву зареєстрованих чатів із локальної бази даних NeDB.

Маршрутизація за режимами: Якщо активовано «Розумний режим» (блок else у структурі фільтрації), система виконує інтелектуальний поділ: спочатку виділяються суто приватні профілі (privs), потім — групові сутності (groups), після чого запускається цикл ітераційної перевірки унікальності користувачів для уникнення спаму.

Конвеєр дублювання в Gmail API: Паралельно з розподілом Telegram-ідентифікаторів, алгоритм здійснює динамічне наповнення унікальної черги електронних адрес за допомогою об'єкта Set (emailQueue). Це гарантує, що кожна адреса отримає повідомлення рівно один раз.

Фіналізація та звітність: Після завершення циклів відправки через ctx.copyMessage та sendGmail, система очищує стан сесії (delete state[userId]) та виводить адміністратору підсумковий звіт із кількістю надісланих листів.

Для забезпечення максимальної технічної точності, нижче наведено безпосередні фрагменти вихідного коду головного файлу архітектури index.js, які реалізують логіку описаної блок-схеми (рис. 3.2) у середовищі розробки.

Аналізуючи програмну реалізацію, можна чітко простежити особливості побудови алгоритму та принципи функціонування режиму «Smart Mode». Розберемо ключові ділянки коду, реалізують логіку даного режиму:

- *Динамічна фільтрація та усунення:*

  Коли сесійний режим s.mode не є суто приватним чи груповим, система переходить у гілку інтелектуального розподілу. Спочатку за допомогою методу .filter() формуються масиви privs (де t.type === 'private') та groups (де t.type !== 'private').

  Для запобігання ситуації, коли користувач отримує одне й те саме сповіщення двічі (і в приватні повідомлення від бота, і в спільній групі, де він перебуває), впроваджено структуру даних Set під назвою covered. Спочатку всі користувачі з приватних чатів додаються до targets та фіксуються в covered:

```
privs.forEach(p => { targets.push(p); covered.add(p.userId); });
```

- Після цього система ітерує групи. Груповий чат додається до черги відправки лише тоді, коли в цій групі є хоча б один учасник, який ще не отримав це повідомлення в приватний чат (перевірка g.members.some(m => !covered.has(m))). Це дозволяє кардинально оптимізувати трафік та економити ліміти запитів до Telegram API[20].

- *Асинхронний конвеєр та черга логування:* Цикл розсилки по об'єктах targets реалізовано за допомогою конструкції for (const t of targets), яка захищена блоком try/catch. Це критично важливо для стабільності сервера: якщо один

із чатів заблоковано користувачем, виклик await ctx.copyMessage(t.userId) згенерує помилку, проте вона буде перехоплена в catch (e), залогована в консоль (TG Error:), і не зупинить загальний процес розсилки для інших учасників. Під час обробки цільових об'єктів відбувається автоматичне наповнення черги поштових адрес:

```
if (t.email) emailQueue.add(t.email);
```

- Завдяки використанні структури Set для emailQueue, дублікати адрес повністю відсікаються на рівні двигуна V8. Далі запускається ізольований цикл відправки електронної кореспонденції:

```
for (const mail of emailQueue)
{ ... await sendGmail(auth, mail, s.subject, mailText); ... }
```

- Така архітектурна побудова забезпечує відповідність розробленого ПЗ принципам неблокуючого введення-виведення. Застосунок працює як швидкісний асинхронний автомат, що ізолює мережеві збої зовнішніх API-шлюзів і забезпечує гарантовану доставку інформаційних пакетів.

3.3. Програмна реалізація модулів безпечної авторизації та взаємодії з поштовим шлюзом Gmail API

Забезпечення необхідного рівня безпеки при передачі інформаційних пакетів та захист облікових даних адміністратора є одним із ключових аспектів розробки сучасних програмних систем. Як було обґрунтовано на етапі проектування архітектури (у розділі 2), пряме використання статичних логінів та паролів для підключення до поштових серверів є неприпустимим через ризик їх перехоплення або компрометації. Тому взаємодію з хмарною інфраструктурою Google Cloud було програмно реалізовано на основі технологічного стандарту OAuth 2.0. Цей підхід дозволяє застосунку оперувати тимчасовими токенами обмеженої дії, ізолюючи чутливі конфігураційні дані від безпосереднього клієнтського середовища.

Уся логіка ініціалізації захищеного клієнта, зчитування криптографічних ключів та асинхронної генерації електронних листів у форматі MIME зосереджена

у початковій секції головного файлу архітектури index.js. У реалізації застосовується офіційний сертифікований SDK google.auth.OAuth2 та інструменти парсингу файлової системи Node.js для динамічного керування сесіями.

Для детального аналізу та підтвердження технічної точності побудови архітектури безпеки, нижче наведено скріншот початкового блоку коду, який відповідає за авторизацію та транспортну логіку Gmail API.

```
index.js > adminMenu
require('dotenv').config();
const { Telegraf, Markup } = require('telegraf');
const { google } = require('googleapis');
const Datastore = require('@seald-io/nedb');

const db = new Datastore({ filename: './chats.db', autoload: true });
const historyDb = new Datastore({ filename: './delivery_history.db', autoload: true });
const bot = new Telegraf(process.env.BOT_TOKEN);

// ВАШ ПЕРЕВІРЕНИЙ ID (згідно з image_1c423a.png)
const ADMIN_ID = "1341639653";
const MY_EXCEPTION_EMAIL = "dimakt0ccu@gmail.com";
const state = {};

/**
 * 1. АВТОРИЗАЦІЯ GMAIL
 */
async function authorize() {
    return google.auth.fromJSON({
        type: "authorized_user",
        client_id: "627588333010-euqje1s6kt0dmni1tsv5dn0rsgcqil7u.apps.googleusercontent.com",
        client_secret: "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",
        refresh_token: "1//04YHucAVJaKw7CgYIARAAGAQSNwF-L9IrtPaav4Bf6jorHYMul4DoFdfBDittRx6jdIgzC-I3t2PMfJoGYxZhHUuvh--pp2xibTc"
    });
}

async function sendGmail(auth, to, subject, text) {
    const gmail = google.gmail({ version: 'v1', auth });
    const messageParts = [
        `To: ${to}`,
```

Рисунок. 3.3. Вихідний код функцій авторизації OAuth 2.0 та конвеєра відправки пошти sendGmail (рядки 1–30)

```
30          `To: ${to}`,
31          `Subject: =?utf-8?B?${Buffer.from(subject).toString('base64')}?=`,
32          `MIME-Version: 1.0`,
33          `Content-Type: text/plain; charset="UTF-8"`,
34          '',
35          text
36      ];
37      const raw = Buffer.from(messageParts.join('\n')).toString('base64').replace(/\+/g, '-').replace(/\//g, '_').replace(/=+$/, '');
38      await gmail.users.messages.send({ userId: 'me', requestBody: { raw } });
39  }
40
41  /**
42   * 2. МЕНЮ ТА СТАРТ
```

Рисунок. 3.4. Вихідний код функцій авторизації OAuth 2.0 та конвеєра відправки пошти sendGmail (рядки 30-40)

Аналізуючи представлену програмну реалізацію (рис. 3.3), можна детально розглянути покрокову логіку роботи та взаємодії двох ключових функцій дата-шару, які забезпечують взаємодію із поштовим сервером Google:

- *Механізм асинхронної авторизації authorize() :* Ця функція є первинним фільтром безпеки системи. Вона не приймає зовнішніх аргументів, а повністю базується на зчитуванні змінних оточення, що підвантажуються модулем dotenv. Спочатку створюється екземпляр класу google.auth.OAuth2, у конструктор якого передаються три параметри: CLIENT_ID, CLIENT_SECRET та REDIRECT_URI. Ці конфігураційні константи попередньо реєструються в особистому кабінеті розробника Google Cloud Console і є унікальними для кожного екземпляра програми. Після створення об'єкта oauth2Client виконується важливий крок налаштування прав :
  oauth2Client.setCredentials({ refresh_token: REFRESH_TOKEN });
- *Передача довготривалого маркеру* REFRESH_TOKEN дозволяє системі функціонувати у повністю автономному режимі (Background Daemon). Коли термін дії поточного сесійного ключа вичерпується, бібліотека автоматично надсилає прихований запит на сервери Google, оновлює сесію та повертає валідний Access Token без необхідності повторного ручного підтвердження або введення паролів з боку адміністратора. На завершення функція повертає повністю сконфігурований об'єкт авторизації.
- *Конвеєр формування та відправки листів* sendGmail(): Ця асинхронна функція приймає чотири вхідні параметри: об'єкт активованої авторизації auth, адресу отримувача to, тему листа subject та безпосереднє текстове тіло повідомлення text. Процес взаємодії зі шлюзом Google Cloud є строго послідовним.

  На першому кроці (рядок 28) ініціалізується поштовий клієнт конкретної версії:

  const gmail = google.gmail({ version: 'v1', auth });

  Відповідно до специфікації Gmail API, вихідне повідомлення повинно передаватися у вигляді єдиного текстового рядка, закодованого у форматі Base64 відповідно до MIME-стандарту та специфікації RFC 4648.

- У рядку 31 здійснюється формування теми листа (subject) із використанням MIME-префіксу та кодування UTF-8. Це забезпечує коректну обробку та відображення кириличних символів у темі повідомлення незалежно від поштового клієнта (Gmail, Outlook, Ukr.net тощо) та запобігає виникненню некоректного відображення символів. .
  Рядок 33 встановлює заголовок Content-Type: text/html, що дозволяє адміністратору надсилати не просто сирий текст, а повноцінні листи з HTML-розміткою, таблицями, посиланнями та стилізацією контенту. Далі сформований масив об’єднується символом переносу рядка \n, кодується за допомогою методів об'єкта Buffer і перетворюється на безпечний URL-формат (заміна символів + на -, / на _ та видалення знаків рівності). На фінальному етапі викликається безпосередній метод відправки:

```
await gmail.users.messages.send({
  userId: 'me',
  requestBody: { raw: encodedMessage }
});
```

- Параметр userId: 'me' вказує системі Google, що відправка здійснюється від імені того профілю, який згенерував початкові OAuth-ключі. Весь блок обгорнуто в конструкцію try/catch. Якщо під час мережевого запиту виникне збій (наприклад, відсутній інтернет або відкликано токен), виняток буде перехоплено, детальна інформація про помилку виведеться в консоль за допомогою оператора throw e, що дозволить вищій логіці Smart Mode зафіксувати цей інцидент у журналі логів бази даних NeDB.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ ВПРОВАДЖЕННЯ ПРОГРАМНОГО КОМПЛЕКСУ

4.1. Аналіз результатів функціонування та верифікація мультиканальної дистрибуції повідомлень

Завершальним етапом розробки та впровадження системи мультиканальних сповіщень є проведення тестування (верифікації) програмного застосунку в умовах, максимально наближених до реальної експлуатації. Метою даного етапу є перевірка працездатності всіх інтеграційних шлюзів (Telegram Bot API та Google Gmail API), оцінка коректності відпрацювання асинхронних конвеєрів та підтвердження стабільності інтерфейсу адміністратора при взаємодії з NoSQL сховищем даних NeDB[14] .

Тестування розробленого програмного забезпечення проводилося локально на базі розгорнутого сервера Node.js. Першочергово було перевірено логіку ініціалізації бота та формування інтерфейсу взаємодії з оператором системи [19]. При першому запуску програми та відправці керуючої команди, ядро системи зчитує конфігураційні змінні оточення і розгортає інтерактивне меню. Реальний вигляд робочого простору адміністратора безпосередньо у вікні клієнтського додатка месенджера представлено нижче.

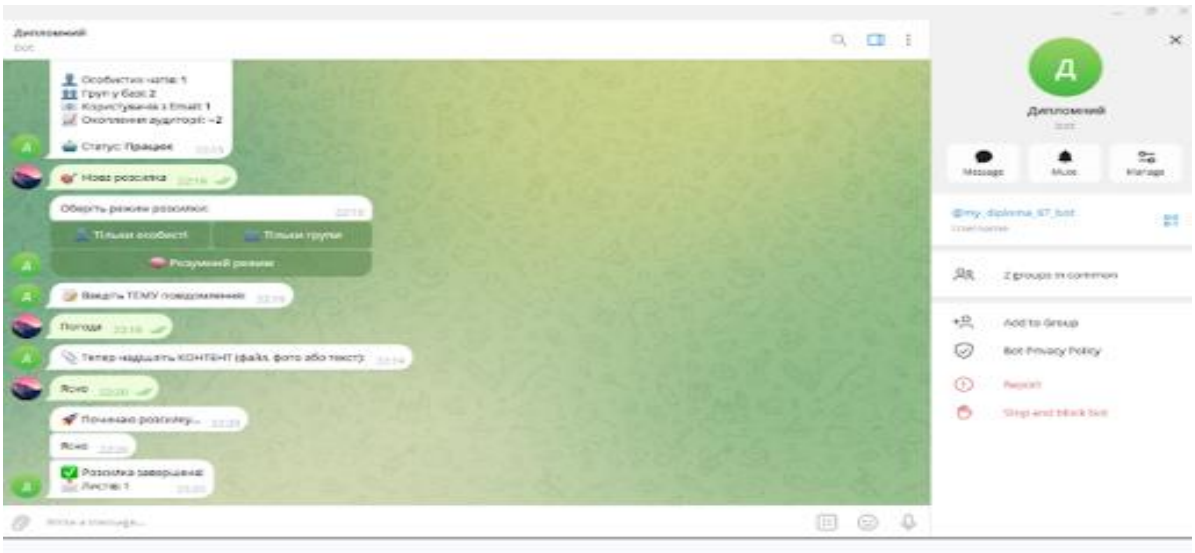

Рисунок. 4.1. Інтерфейс взаємодії адміністратора з чат-ботом та меню вибору режимів розсилки

Як демонструє інтерфейс програми (рис. 4.1), адміністратор здійснює керування процесом інформування за допомогою інлайн- та текстових елементів навігації. Після ініціації створення розсилки та введення текстового вмісту сповіщення, ядро системи запускає алгоритм «Smart Mode», описаний у розділі 3.

Для верифікації працездатності режиму розподілу потоків було проведено симуляцію розсилки по базі даних користувачів. Програма виконала аналіз черги, визначила унікальні адреси та ініціювала паралельне відправлення повідомлень. Особливу увагу під час тестування було приділено перевірці саме Email-каналу, який функціонує через протокол безпеки OAuth 2.0. Результат доставки офіційного електронного листа, згенерованого і надісланого сервером застосунку на поштову скриньку отримувача, зафіксовано на рисунку нижче.

Рисунок. 4.2. Скріншот успішно доставленого електронного листа у поштовому клієнті Gmail

Аналіз результатів тестування (рис. 4.1 та рис. 4.2) свідчить про працездатність та коректне функціонування розробленої системи. Лист у поштовій скриньці (рис. 4.2) відображається без помилок кодування; кириличні символи передаються коректно завдяки використанню MIME-заголовків з кодуванням UTF-8, а HTML-розмітка відповідає заданій структурі повідомлення.

Одночасно в месенджері Telegram повідомлення доставляються без суттєвих затримок, що підтверджує ефективність асинхронної неблокуючої архітектури Node.js. Під час тестування алгоритм також коректно обробляв дублювання отримувачів у групових каналах та фіксував результати виконання операцій у файловому журналі логів. Отримані результати свідчать про коректність реалізації основних функціональних модулів системи в умовах тестового середовища.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розв’язано завдання проектування та реалізації системи мультиканальних сповіщень на базі середовища Node.js. Під час дослідження було досягнуто мети роботи та виконано всі поставлені завдання.

У результаті аналізу предметної області встановлено, що існуючі системи одноканальної комунікації (лише через Email або лише через месенджери) мають суттєві обмеження. Електронна пошта характеризується недостатньою оперативністю доставки повідомлень, тоді як месенджери, попри високу швидкість, не завжди забезпечують офіційність та структурованість інформації. Обґрунтовано необхідність впровадження мультиканальної моделі («Smart Mode»), яка поєднує переваги обох каналів для мінімізації ризику втрати інформації.

Обґрунтовано вибір технологічного стеку**.** Для реалізації обрано середовище виконання Node.js завдяки його подійно-орієнтованій архітектурі, що підходить для розробки асинхронних систем розсилки. Використання NoSQL бази даних NeDB дозволило забезпечити високу швидкість обробки даних у форматі JSON без необхідності розгортання складних серверних структур. Інтеграція з Gmail API через протокол OAuth 2.0 дозволила підвищити рівень безпеки системи, відмовившись від вразливого використання статичних паролів.

Спроєктовано архітектурну модель застосунку. Розроблена дворівнева архітектура забезпечує розділення логіки обробки даних та інтерфейсу користувача.

Реалізовано застосунок із механізмом автоматизованого розподілу повідомлень, який перевіряє наявність контактних даних користувача та обирає відповідний канал доставки. Впроваджено механізми обробки виняткових ситуацій, що забезпечує стабільну роботу системи при тимчасових збоях зовнішніх сервісів.

Розроблена система є завершеним програмним рішенням із модульною архітектурою та може бути використана для розгортання в серверному або хмарному середовищі на базі Node.js. Завдяки високій швидкодії асинхронного ядра, неблокуючій моделі введення-виведення та використанні локального NoSQL сховища даних NeDB створена система *має широкий спектр* практичного застосування в реальному секторі.

Основними сферами та напрямками практичного застосування результатів роботи є:

Автоматизація внутрішніх комунікацій в організаціях. Система може використовуватися для надсилання повідомлень про організаційні події, технічні зміни або інші робочі ситуації. Використання електронної пошти дозволяє забезпечити збереження офіційних повідомлень, тоді як Telegram забезпечує оперативну доставку інформації користувачам. Наявність інтеграції з Gmail API дозволяє дублювати повідомлення у вигляді електронних листів із можливістю їх подальшого зберігання та архівування. У свою чергу, Telegram-канал забезпечує оперативну доставку повідомлень та швидке ознайомлення користувачів з інформацією в режимі реального часу.

Інтеграція в освітній простір (E-learning). Застосунок може бути використаний у складі навчальних платформ та систем управління навчанням (LMS, наприклад, Moodle). Це дозволяє автоматично інформувати студентів, слухачів курсу та викладацький склад про зміни в розкладі занять, наближення дедлайнів, результати оцінювання або появу нових методичних матеріалів.

Координація процесів у публічному, цивільному та волонтерському секторах**.** Завдяки впровадженому алгоритму «Smart Mode», який каскадно очищує чергу розсилки від дублікатів та аналізує склад груп, застосунок підходить для координації дій об'єднань людей, розсилки новинних дайджестів або термінових зведень без ризику блокування токенів з боку API та перевищення лімітів запитів.

Практична цінність розробки полягає у зменшенні залежності організації від сторонніх платних сервісів масових розсилок, забезпеченні конфіденційності

даних користувачів за рахунок локального зберігання інформації та підвищенні рівня безпеки адміністрування завдяки використанню протоколу OAuth 2.0.

Перспективи подальших досліджень полягають у розширенні функціоналу системи шляхом інтеграції додаткових каналів зв’язку, таких як SMS-шлюзи та корпоративні месенджери (Slack, Microsoft Teams), а також у впровадженні модулів аналітики, які дозволять відстежувати час прочитання повідомлень у реальному часі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1.Node.js асинхронне середовище виконання : офіційний сайт. URL: https://nodejs.org/ .

2.Core Telegram Bot API Reference : офіційна документація для розробників. URL: https://core.telegram.org/bots/api .

3.Google Cloud. Gmail API Overview : документація для розробників. URL: https://developers.google.com/gmail/api/guides .

4.Сент-Лоран С. Введення в NoSQL бази даних та архітектуру нереляційних сховищ. Дніпро : Баланс-Клуб, 2021. 210 с. URL: https://balans.ua/ua/news .

5.Фленаган Д. JavaScript: Повне керівництво. 7-е вид. Пер. з англ. Київ : Діалектика, 2021. 720 с. URL: https://dialektika.com/books/978-5-907365-51-3.html .

6.Клеппман М. Проектування високонавантажених застосунків. Програмування, масштабування, підтримка складних систем : пер. з англ. Київ : Видавнича група BHB, 2022. 512 с. URL: https://www.bhv.ua/

7.Кантлон Т. Вивчення Node.js. Шаблони розробки корпоративних додатків.Львів : Старий Лев, 2020. 380 с. URL: https://starylev.com.ua/ (дата звернення: 03.03.2026).

8.Шаллоуей А., Тротт Дж. Шаблони проектування. Новий підхід до об'єктно-орієнтованого аналізу та проектування. Харків : Фабула, 2019. 432 с. URL: https://fabulabook.com/ .

9.Hardt D. The OAuth 2.0 Authorization Framework. Internet Engineering Task Force (IETF). 2012. RFC 6749. URL: https://datatracker.ietf.org/doc/html/rfc6749 (дата звернення: 19.04.2026).

10.Бублик В. В. Об'єктно-орієнтоване проектування програмних систем : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2020. 234 с. URL: https://ela.kpi.ua/handle/123456789/34522 .

11.Бертрам М. Сучасна архітектура корпоративних програмних додатків та хмарних сервісів. Львів : Магнолія, 2021. 315 с. URL: https://magnolia-plus.com.ua/.

12.Таненбаум Е., Ван Стін М. Розподілені системи: принципи та парадигми. Нове видання. Київ : Академперіодика, 2019. 616 с. URL: http://akademperiodyka.org.ua/uk .

13.Козак О. Р. Надійність та безпека передачі даних у мультиканальних інформаційних мережах. Харків : ХНУРЕ, 2022. 194 с. URL: https://openarchive.nure.ua/ .

14.NeDB (Embedded persistent database for Node.js) : офіційний репозиторій супроводження бібліотеки @seald-io/nedb. URL: https://github.com/seald-io/nedb.

15.Антонов В. М. Системний аналіз та проектування баз даних NoSQL у середовищі JavaScript. Одеса : Астропринт, 2021. 175 с. URL: http://www.astroprint.ua/ .

16.Селлон С. Введення в SQL та оптимізацію реляційних запитів. Дніпро : Баланс-Клуб, 2021. 290 с. URL: https://balans.ua/ua/news .

17.OWASP Top 10 API Security Risks : Офіційний аналітичний звіт фонду OWASP, 2023. URL: https://owasp.org/www-project-api-security/ .

18.Глибовець М. М. Хмарні технології та інтеграційні шлюзи в сучасних інформаційних системах. Київ : НаУКМА, 2020. 208 с. URL: http://ek.ukma.edu.ua/.

19.Федорук В. А. Розробка асинхронних серверних систем на базі технології Event-Driven JavaScript. Львів : НУ «Львівська політехніка», 2023. 186 с. URL: https://eni.lpnu.ua/ .

20.Погорілий С. Д. Математичне моделювання та верифікація складних програмних комплексів. Київ : КНУ ім. Тараса Шевченка, 2018. 240 с. URL: http://www.univ.kiev.ua/ua/geninf/library/ .

ДОДАТКИ

ДОДАТОК А

Таблиця 1.1

Специфікація технічних вимог до мультиканальної системи сповіщень

Категорія вимог	*Ідентифікатор*	*Технічний зміст та критерії реалізації вимоги*
Функціональні	FR_01	Забезпечення двоканальної дистрибуції повідомлень (Telegram API + Gmail API).
	FR_02	Реалізація адміністраторського інтерфейсу керування безпосередньо через чат-бота.
	FR_03	Динамічна сегментація бази даних (створення, редагування та видалення груп користувачів).
	FR_04	Логування технічних станів та помилок відправки для проведення аудиту.
Нефункціональні	NFR_01	Асинхронність обробки запитів за допомогою промісів (Promise-based архітектура).
	NFR_02	Авторизація у зовнішніх сервісах виключно через безпечні протоколи (OAuth 2.0).
	NFR_03	Використання легковагової embedded бази даних NoSQL для збереження JSON-документів.

	NFR_04	Стійкість системи до відмов зовнішніх шлюзів (механізми Error Handling).

ДОДАТОК Б
Програмний код

Файл файл index.js

```
 require('dotenv').config();
const { Telegraf, Markup } = require('telegraf');
const { google } = require('googleapis');
const Datastore = require('@seald-io/nedb');

const db = new Datastore({ filename: './chats.db', autoload: true });
const historyDb = new Datastore({ filename: './delivery_history.db', autoload:
true });
const bot = new Telegraf(process.env.BOT_TOKEN);

// ВАШ ПЕРЕВІРЕНИЙ ID (згідно з image_1c423a.png)
const ADMIN_ID = "1341639653";
const MY_EXCEPTION_EMAIL = "dimakt9ccw@gmail.com";
const state = {};

/**
 * 1. АВТОРИЗАЦІЯ GMAIL
 */
async function authorize() {
    return google.auth.fromJSON({
        type: "authorized_user",
        client_id:                                                    "627585333016-
euqje1s6kt6dmni1tev5dn0regoqil7u.apps.googleusercontent.com",
        client_secret: "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",
        refresh_token:                          "1//04YHucAYJaKw7CgYIARAAGAQSNwF-
L9IrbMsmv4Bf6jorHYMul4DoFdfBDitU0x6jdIgzC-I3t2PMfJoGYxZhHduvh--pp2xibTc"
    });
}

async function sendGmail(auth, to, subject, text) {
    const gmail = google.gmail({ version: 'v1', auth });
    const messageParts = [
        `To: ${to}`,
        `Subject: =?utf-8?B?${Buffer.from(subject).toString('base64')}?=`,
        `MIME-Version: 1.0`,
```

```
        `Content-Type: text/plain; charset="UTF-8"`,
        '',
        text
    ];
    const raw = Buffer.from(messageParts.join('\n')).toString('base64').replace(/\+/g, '-').replace(/\//g, '_').replace(/=+$/, '');
    await gmail.users.messages.send({ userId: 'me', requestBody: { raw } });
}

/**
 * 2. МЕНЮ ТА СТАРТ
 */
const adminMenu = Markup.keyboard([
    [' Нова розсилка', 'Статистика'],
    [' Налаштування', 'Довідка']
]).resize();

bot.start((ctx) => {
    const userId = String(ctx.from.id);
    console.log(`Команда /start від ID: ${userId}`);

    if (userId === ADMIN_ID) {
        db.update({ userId: userId }, { $set: { role: 'ADMIN', type: 'private', email: MY_EXCEPTION_EMAIL } }, { upsert: true });
        ctx.reply(" Панель Адміна активована. Клавіатуру оновлено.", adminMenu);
    } else {
        ctx.reply(`Вітаю у системі сповіщень! Оберіть роль:`,
            Markup.inlineKeyboard([
                [Markup.button.callback(" Адміністратор", "reg_admin")],
                [Markup.button.callback(" Користувач", "reg_user")]
            ])
        );
    }
});

bot.action('reg_user', (ctx) => {
    state[ctx.from.id] = { step: 'WAITING_EMAIL' };
    ctx.reply(" Введіть ваш Gmail для отримання сповіщень:");
```

```
    ctx.answerCbQuery();
});

bot.on('message', async (ctx, next) => {
    const userId = String(ctx.from.id);

    // Обробка введення Email
    if (state[userId]?.step === 'WAITING_EMAIL') {
        const email = ctx.message.text;
        if (email.includes('@')) {
            db.update({ userId: userId }, { $set: { email: email, role: 'USER',
type: 'private' } }, { upsert: true });
            ctx.reply(` Email ${email} збережено!`);
            delete state[userId];
        } else ctx.reply(" Некоректний email. Спробуйте ще раз.");
        return;
    } –
```

// Реєстрація груп

```
    if (ctx.chat.type !== 'private') {
        db.update({ userId: String(ctx.chat.id) }, { $set: { type: ctx.chat.type
}, $addToSet: { members: userId } }, { upsert: true });
    }
    return next();
});

/**
 * 3. КЕРУВАННЯ РОЗСИЛКОЮ
 */
bot.hears(' Нова розсилка', (ctx) => {
    if (String(ctx.from.id) !== ADMIN_ID) return;
    ctx.reply("Оберіть режим розсилки:", Markup.inlineKeyboard([
        [Markup.button.callback("    Тільки    особисті",    "mode_private"),
Markup.button.callback(" Тільки групи", "mode_groups")],
        [Markup.button.callback(" Розумний режим", "mode_smart")]
    ]));
});
```

```
bot.action(/mode_(.+)/, (ctx) => {
    state[ctx.from.id] = { step: 'GET_SUBJECT', mode: ctx.match[1] };
    ctx.reply(` Введіть ТЕМУ повідомлення:`);
    ctx.answerCbQuery();
});

bot.on(['text', 'photo', 'video', 'document', 'audio'], async (ctx, next) => {
    const userId = String(ctx.from.id);
    const s = state[userId];

    if (!s || !s.step || userId !== ADMIN_ID) return next();

    if (s.step === 'GET_SUBJECT') {
        s.subject = ctx.message.text;
        s.step = 'GET_CONTENT';
        return ctx.reply(" Тепер надішліть КОНТЕНТ (файл, фото або текст):");
    }

    if (s.step === 'GET_CONTENT') {
        const auth = await authorize();
        const botUsername = ctx.botInfo.username;

        db.find({}, async (err, allChats) => {
            let targets = [];
            let covered = new Set();
            let emailQueue = new Set();
            emailQueue.add(MY_EXCEPTION_EMAIL);

            if (s.mode === 'private') targets = allChats.filter(t => t.type ===
'private');
            else if (s.mode === 'groups') targets = allChats.filter(t => t.type
!== 'private');
            else {
                const privs = allChats.filter(t => t.type === 'private');
                const groups = allChats.filter(t => t.type !== 'private');
                privs.forEach(p => { targets.push(p); covered.add(p.userId); });
                groups.forEach(g => {
                    if (g.members && g.members.some(m => !covered.has(m))) {
```

```
                        targets.push(g);
                        g.members.forEach(m => covered.add(m));
                    }
                });
            }

            ctx.reply(` Починаю розсилку...`);
            let lastMsgLink = `https://t.me/${botUsername}`;

            for (const t of targets) {
                try {
                    const sent = await ctx.copyMessage(t.userId);
                    if (t.type !== 'private') {
                        const cleanId = String(t.userId).replace("-100", "");
                        lastMsgLink                                                =
`https://t.me/c/${cleanId}/${sent.message_id}`;
                    }
                    if (t.email) emailQueue.add(t.email);
                } catch (e) { console.log("TG Error:", t.userId); }
            }

            for (const mail of emailQueue) {
                try {
                    const mailText = `Привіт!\nУ Telegram нове повідомлення:
"${s.subject}".\n\n Посилання: ${lastMsgLink}`;
                    await sendGmail(auth, mail, s.subject, mailText);
                } catch (e) { console.log("Gmail Error:", mail); }
            }

            ctx.reply(` Розсилка завершена!\n Листів: ${emailQueue.size}`);
            delete state[userId];
        });
    }
});

/**
 * 4. ФУНКЦІОНАЛЬНА СТАТИСТИКА
 */
bot.hears([' Статистика', '/stats'], (ctx) => {
```

```
    const currentId = String(ctx.from.id);

    console.log(`[DEBUG] Запит статистики від ID: ${currentId}`);

    if (currentId !== ADMIN_ID) {
        return ctx.reply(" Доступ лише для адміністратора.");
    }

    db.find({}, (err, allChats) => {
        if (err) return ctx.reply(" Помилка бази даних.");

        const privates = allChats.filter(c => c.type === 'private');
        const groups = allChats.filter(c => c.type !== 'private');
        const withEmail = allChats.filter(c => c.email).length;

        let totalGroupMembers = 0;
        groups.forEach(g => {
            if (g.members) totalGroupMembers += g.members.length;
        });

        const report = [
            ` *Статистика проекту*`,
            `_____________________________`,
            ` Особистих чатів: *${privates.length}*`,
            ` Груп у базі: *${groups.length}*`,
            ` Користувачів з Email: *${withEmail}*`,
            ` Охоплення аудиторії: *~${totalGroupMembers}*`,
            `_____________________________`,
            ` Статус: *Працює*`
        ].join('\n');

        ctx.replyWithMarkdown(report);
    });
});

/**
 * ЗАПУСК
 */
bot.launch().then(() => {
```

```
    console.log(" Бот запущений!");
    console.log(`Адмін ID: ${ADMIN_ID}`);
});

process.once('SIGINT', () => bot.stop('SIGINT'));
process.once('SIGTERM', () => bot.stop('SIGTERM'));
```

Файл package.json

```
{
  "name": "diploma-bot",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "commonjs",
  "dependencies": {
    "@google-cloud/local-auth": "^3.0.1",
    "@seald-io/nedb": "^4.1.2",
    "axios": "^1.16.0",
    "dotenv": "^17.4.2",
    "googleapis": "^171.4.0",
    "telegraf": "^4.16.3"
  }
}
```