Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Комп'ютерних наук і технологій
(повне найменування факультету)

Комп'ютерні системи та мережі
(повне найменування кафедри)

Пояснювальна записка

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

на тему: РОЗРОБКА СЕРВІСУ РЕЗЕРВУВАННЯ ЕЛЕКТРОННИХ ЛИСТІВ

Виконав: студент 4 курсу,
групи КНТ-521

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

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

ФІЛІППЕНКОВ К.Ю.
(ПРІЗВИЩЕ та ініціали)

Керівник ТЯГУНОВА М.Ю.
(ПРІЗВИЩЕ та ініціали)
Рецензент СТЕПАНЕНКО О.О.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

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

ЗАТВЕРДЖУЮ
Завідувач кафедри КУДЕРМЕТОВ Р.К.
_____
«14» квітня 2025 року

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

ФІЛІППЕНКОВА Кирилла Юрійовича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка сервісу резервування електронних листів

керівник проєкту (роботи) к.т.н., доцент, ТЯГУНОВА Марія Юріївна ,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від « 08 » квітня 2025 року № 151
2. Строк подання студентом проєкту (роботи) 01.06.2025 р.
3. Вихідні дані до проєкту (роботи) опис предметної області, технології розробки клієнтської та серверної частини, середовища розробки Rider, Webstorm, DataGrip, Docker Desktop, персональний комп'ютер із процесором AMD Ryzen 5 3600 6-Core на базі операційної системи Windows 10 Pro

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1) Опис предметної області;
2) Проєктування та моделювання ПЗ;
3) Реалізація сервісу;
4) Результати роботи.

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

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	ТЯГУНОВА М.Ю.		
Нормоконтроль	ПОЛЬСЬКА О.В.		

7. Дата видачі завдання «14» квітня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз предметної області	до 18.04.2025	
2	Проєктування архітектури	до 28.04.2025	
2	Розробка серверної частини	до 16.05.2025	
3	Розробка клієнтської частини	до 25.05.2025	
4	Оформлення пояснювальної записки	до 31.05.2025	
5	Проходження нормоконтролю	до 01.06.2025	
6	Перевірка на наявність академічного плагіату	до 03.06.2025	
7	Проходження рецензування	до 10.06.2025	

Студент(ка) _____ Кирилл ФІЛІППЕНКОВ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Марія ТЯГУНОВА
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 106 с., 4 табл., 36 рис., 2 дод., 54 джерела.

ВЕБІНТЕРФЕЙС, ЕЛЕКТРОННА ПОШТА, МУЛЬТИТЕНАНТНА АРХІТЕКТУРА, РЕЗЕРВНЕ КОПІЮВАННЯ, СИСТЕМА, ШИФРУВАННЯ, AMAZON S3, IMAP, JWT-АУТЕНТИФІКАЦІЯ, MYSQL.

Об’єкт розробки – вебзастосунок EmailTamer для резервного копіювання електронних листів із можливістю їхнього зберігання, пошуку та перегляду.

Мета роботи – створення безпечного, зручного та масштабованого сервісу для автоматизованого резервного копіювання електронних листів, їхнього зберігання в реляційній базі даних і хмарному сховищі, а також забезпечення зручного доступу через вебінтерфейс.

У першому розділі проаналізовано предметну область, підкреслено важливість захисту даних електронної пошти від збоїв, кібератак і помилок. Порівняно аналоги, виділено переваги EmailTamer, сформульовано вимоги.

У другому розділі спроєктовано архітектуру системи з мультитенантною організацією БД, описано моделі даних та клієнт-серверну взаємодію.

У третьому розділі описано реалізацію: frontend на React, backend на .NET 8. Забезпечено шифрування, JWT-аутентифікацію, адаптивність та багатомовність.

У четвертому розділі представлено результати: EmailTamer підключається до скриньок, зберігає метадані, вкладення, підтримує пошук, фільтрацію, багатомовність та адаптивність, реалізуючи мультитенантність та шифрування.

EmailTamer – конкурентне, безпечне, зручне і масштабоване рішення, що має перспективи підтримки POP3, SMTP та аналітичних функцій. Робота має практичне значення для захисту даних електронної пошти та може бути основою для подальших досліджень у галузях зберігання та безпеки даних.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

БД	–	База даних
AES	–	Advanced Encryption Standard, Симетричний алгоритм блочного шифрування
API	–	Application Programming Interface, Програмний інтерфейс додатку
Backend	–	Серверна частина
CORS	–	Cross-Origin Resource Sharing, Спільне використання ресурсів з різних джерел
CQRS	–	Command Query Responsibility Segregation, Розділення відповідальності команд і запитів
DOM	–	Document Object Model, Модель об’єктів документа
DTO	–	Data Transfer Objects, Об’єкт передачі даних
EF Core	–	Entity Framework Core, Ядро сутнісної структури
Frontend	–	Клієнтська частина
GDPR	–	General Data Protection Regulation, Нормативно-правовий акт Європейського Союзу
HIPAA	–	Health Insurance Portability and Accountability Act, Акт про мобільність та підзвітність медичного страхування
HTML	–	HyperText Markup Language, Мова розмітки гіпертексту
HTTP	–	Hypertext Transfer Protocol, Протокол передачі гіпертексту
HTTPS	–	HyperText Transfer Protocol Secure, Безпечний протокол передачі гіпертексту
IDE	–	integrated development environment, Інтегроване середовище розробки
IMAP	–	Internet Message Access Protocol, Протокол доступу до повідомлень
IoC	–	Inversion of Control, Інверсія керування

JWT – JSON Web Token, Стандарт токена доступу на основі JSON

LINQ – Language Integrated Query, Запити, інтегровані в мову

MUI – Material-UI

PDF – Portable Document Format, Формат портативного документу

POP3 – Post Office Protocol Version 3, Протокол поштового відділення, версії 3

S3 – Simple Storage Service, хмарне сховище Amazon

SQL – Structured query language, Мова структурованих запитів

UI – User Interface, Інтерфейс користувача

URL – Uniform Resource Locator, Уніфікований локатор ресурсів

UTC – Coordinated Universal Time, Всесвітній координований час

ВСТУП

Електронна пошта залишається одним із найпоширеніших засобів комунікації у сучасному світі, використовуючись для обміну інформацією в бізнесі, державному управлінні, освіті та особистих цілях. Щоденно користувачі надсилають і отримують мільйони листів, які містять критично важливі дані: контракти, фінансові документи, персональну інформацію, вкладення чи листування, що має юридичну вагу. Втрата таких даних через апаратні збої, кібератаки, вичерпання обсягу сховища поштових сервісів або людські помилки може призвести до серйозних наслідків, включаючи фінансові збитки, порушення ділових процесів, втрату довіри клієнтів чи навіть юридичні санкції. Крім того, регуляторні стандарти, такі як GDPR у Європі чи HIPAA у США, вимагають від організацій зберігати електронні листи протягом визначеного періоду, що підкреслює необхідність надійних систем для їхнього резервного копіювання.

Розвиток інформаційних технологій, зокрема хмарних обчислень, протоколів синхронізації, таких як IMAP, і хмарних сховищ, таких як Amazon S3, створює нові можливості для автоматизації бекапу електронних листів. Водночас зростання кіберзагроз, таких як фішинг, програми-вимагачі чи атаки "людина посередині", підвищує вимоги до безпеки та ізоляції даних. Мультитенантна архітектура, яка передбачає окремі бази даних для кожного користувача, дозволяє забезпечити ізоляцію даних і підвищити безпеку, що є особливо важливим для сервісів, які обробляють конфіденційну інформацію. Зручний вебінтерфейс, який підтримує пошук, фільтрацію та швидкий доступ до архівів, є ключовим для забезпечення позитивного користувацького досвіду.

Розробка вебзастосунку для створення бекапів електронних листів є актуальною задачею, оскільки вона відповідає сучасним потребам у захисті даних, автоматизації процесів і зручному управлінні архівами. Метою дипломної роботи є створення безпечного, зручного та масштабованого сервісу для автоматизованого резервного копіювання електронних листів, їхнього зберігання в реляційній базі

даних і хмарному сховищі, а також забезпечення зручного доступу через вебінтерфейс.

Для досягнення поставленої мети у роботі необхідно виконати такі завдання:

- проаналізувати аналогічні сервіси для визначення їхніх переваг і недоліків;
- сформулювати вимоги до розроблюваного сервісу;
- обґрунтувати вибір технологій і програмних засобів;
- спроєктувати архітектуру системи;
- розробити frontend, backend та інфраструктуру сервісу;
- провести демонстрацію результатів роботи через опис функціональності та інтерфейсу.

Очікується, що EmailTamer забезпечить надійне зберігання даних, ізоляцію між користувачами завдяки мультитенантній архітектурі, високу продуктивність і зручний інтерфейс, що зробить його конкурентоспроможним рішенням для індивідуальних і корпоративних користувачів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Основні поняття бекапу електронних листів

Бекап електронних листів — це процес створення резервних копій електронної пошти, включаючи тіло листів, метадані (відправник, одержувач, дата, тема) та вкладення, з метою збереження та забезпечення доступу. Основна мета бекапу полягає у захисті даних від втрати через апаратні збої, кібератаки (наприклад, програми-вимагачі), випадкове видалення, вичерпання обсягу сховища поштового сервісу або помилки користувача. Бекап також забезпечує відповідність регуляторним вимогам, наприклад, у фінансовій чи медичній сферах, де збереження листів є обов’язковим.

Ключові аспекти бекапу електронних листів:

- надійність: резервні копії повинні зберігатися у безпечному вигляді, із захистом від несанкціонованих змін;

- доступність: користувачі повинні мати швидкий доступ до архівів через зручний інтерфейс;

- автоматизація: періодична синхронізація з поштовими скриньками зменшує потребу у ручному управлінні;

- масштабованість: система має підтримувати велику кількість користувачів і обсягів даних;

- безпека: дані повинні бути ізольованими між користувачами, із захистом за допомогою шифрування та автентифікації;

Загрози для електронних листів включають:

- технічні збої: пошкодження серверів або носіїв даних;

- кібератаки: фішинг, програми-вимагачі, атаки "людина посередині";

- людський фактор: випадкове видалення листів або неправильне налаштування поштового клієнта;

- регуляторні ризики: втрата даних може порушити вимоги GDPR, HIPAA чи інших стандартів.

Методи бекапу включають використання протоколу IMAP для синхронізації з поштовими серверами, збереження даних у реляційних базах даних (наприклад, MySQL) та хмарних сховищах (Amazon S3), а також застосування мультитенантної архітектури для ізоляції даних користувачів [1-7].

1.2 Аналіз аналогічних сервісів

Для визначення позиціонування EmailTamer було проаналізовано кілька популярних сервісів для бекапу та управління електронними листами: MailStore, Nylas, CloudHQ і Dropsuite. Нижче наведено їхній огляд, схожості та відмінності.

1.2.1 Сервіс MailStore

MailStore — це програмне забезпечення для архівування електронної пошти, яке підтримує IMAP, POP3 та Exchange. Воно дозволяє створювати резервні копії листів і забезпечує пошук та відновлення.

Функціонал:

- синхронізація через IMAP/POP3;
- локальне або хмарне зберігання;
- пошук за ключовими словами, фільтрація за папками;
- експорт листів у різні формати (PDF, EML).

Переваги:

- висока швидкість архівування;
- підтримка багатьох поштових клієнтів (Gmail, Outlook);
- інтеграція з корпоративними системами.

Недоліки:

- вимагає встановлення на сервер або ПК;
- обмежена підтримка мультитенантності;
- складне налаштування для непрофесійних користувачів.

Схожості з EmailTamer:

- використання IMAP для синхронізації;

- можливість пошуку та фільтрації листів.

Відмінності:

- EmailTamer є вебзастосунком, не потребує локальної установки;

- EmailTamer використовує мультитенантну архітектуру для ізоляції даних користувачів [1, 8-9].

1.2.2 Сервіс Nylas

Nylas — це хмарна платформа для інтеграції з поштовими клієнтами, яка забезпечує синхронізацію, пошук і управління листами через API.

Функціонал:

- синхронізація через IMAP і API;

- хмарне зберігання даних;

- розширений пошук і аналітика листів;

- інтеграція з додатками через API.

Переваги:

- гнучкість для розробників завдяки API;

- підтримка великих обсягів даних;

- хмарна масштабованість.

Недоліки:

- висока вартість для малих компаній;

- складність інтеграції для некваліфікованих користувачів;

- менший акцент на бекап, більше на інтеграцію.

Схожості з EmailTamer:

- хмарне зберігання та синхронізація через IMAP;

- вебінтерфейс для доступу до листів.

Відмінності:

- EmailTamer фокусується на бекапі, а не на API-інтеграції;

- EmailTamer пропонує простіший інтерфейс взаємодії для кінцевих користувачів [1, 10-11].

1.2.3 Сервіс CloudHQ

CloudHQ — це хмарний сервіс для синхронізації та бекапу даних, включаючи електронні листи, між різними платформами (Gmail, Dropbox, OneDrive).

Функціонал:

- автоматична синхронізація листів із Gmail та Outlook;
- збереження листів у хмарних сховищах;
- пошук і фільтрація архівів.

Переваги:

- простота використання;
- інтеграція з популярними хмарними сервісами;
- автоматизовані бекапи.

Недоліки:

- обмежена підтримка IMAP для непопулярних клієнтів;
- відсутність мультитенантної архітектури;
- залежність від сторонніх хмарних сервісів.

Схожості з EmailTamer:

- автоматизовані бекапи та хмарне зберігання;
- можливість пошуку листів.

Відмінності:

- EmailTamer підтримує мультитенантність;
- EmailTamer інтегрується з Amazon S3 для вкладень [1, 12-15].

1.2.4 Сервіс Dropsuite

Dropsuite — це хмарний сервіс для бекапу електронної пошти та інших даних, орієнтований на корпоративних користувачів.

Функціонал:

- синхронізація через IMAP і Exchange;
- хмарне зберігання з шифруванням;
- розширений пошук і відновлення листів;
- відповідність регуляціям (GDPR, HIPAA).

Переваги:

- високий рівень безпеки та шифрування;

- підтримка регуляторних вимог;

- зручний інтерфейс для адміністрування.

Недоліки:

- висока вартість для індивідуальних користувачів.

Схожості з EmailTamer:

- підтримка IMAP і хмарного зберігання;

- фокус на безпеці даних.

Відмінності:

- EmailTamer пропонує мультитенантну архітектуру [1, 16-17].

1.2.5 Порівняльна таблиця

У таблиці 1.1 наведено підсумовуюче порівняння сервісів.

Таблиця 1.1 - Порівняльна таблиця аналогічних сервісів

Сервіс	IMAP	Мультитенантність	Хмарне зберігання	Вебінтерфейс	Вартість	Простота використання
MailStore	Так	Ні	Так	Частково	Середня	Складно
Nylas	Так	Так, з налаштуванням	Так	Так	Висока	Складно
CloudHQ	Так	Ні	Так	Так	Середня	Просто
Dropsuite	Так	Так	Так	Так	Висока	Середньо
EmailTamer	Так	Так	Так	Так	-	Просто

1.3 Вимоги до розроблюваного сервісу

1.3.1 Функціональні вимоги

Функціональні вимоги визначають, які саме дії має виконувати система та які задачі повинні бути реалізовані для досягнення поставлених цілей. Вони

охоплюють основні функції, що забезпечують роботу сервісу згідно з технічним завданням.

До функціональних вимог належать:

а) автентифікація користувачів: підтримка JWT-автентифікації для безпечного доступу;

б) синхронізація листів: періодична або ручна синхронізація з поштовими скриньками через IMAP [1];

в) збереження даних:

1) метадані листів (відправник, одержувач, дата, тема і т. д.) у реляційній базі даних (MySQL [4]);

2) вкладення у хмарному сховищі (Amazon S3 [3]);

г) перегляд листів: відображення листів і ланцюжків листів у вебінтерфейсі [18-19];

д) пошук і фільтрація:

1) пошук за ключовими словами у тілі листів і метаданих [18-19];

2) фільтрація за папками, поштовими скриньками, датами;

е) завантаження вкладень: можливість завантажувати вкладення з S3 [3];

є) мультитенантність: ізоляція даних кожного користувача у окремій базі даних [5-7];

ж) шифрування: шифрування чутливих даних (поштова скринька, пароль і т. д.) [20].

1.3.2 Нефункціональні вимоги

Нефункціональні вимоги описують обмеження та характеристики, які не стосуються конкретних функцій, але впливають на загальну якість системи, її зручність, надійність, масштабованість і продуктивність.

До нефункціональних вимог належать:

- масштабованість: підтримка великої кількості користувачів і обсягів даних;
- продуктивність: швидкий доступ до листів і вкладень;
- зручність використання: інтуїтивний вебінтерфейс із адаптивним дизайном та підтримкою локалізації.

1.4 Вибір та обґрунтування технологій

Для розробки EmailTamer було обрано сучасний стек технологій, який відповідає вимогам безпеки, масштабованості, продуктивності та зручності розробки. Нижче наведено детальний опис обраних технологій для frontend, backend, інфраструктури та інструментів розробки, із поясненням їхнього вибору.

1.4.1 Frontend

React: бібліотека для створення реактивних вебінтерфейсів, яка забезпечує швидке оновлення UI завдяки віртуальному DOM і компонентному підходу. React дозволяє створювати модульні компоненти, такі як списки листів чи форми автентифікації, що спрощує розробку та підтримку [21].

TypeScript: мова програмування, що додає статичну типізацію до JavaScript. TypeScript обрано замість JavaScript, оскільки він зменшує кількість помилок під час розробки завдяки перевірці типів на етапі компіляції. Наприклад, TypeScript дозволяє визначити типи для API-відповідей, що запобігає помилкам при роботі з даними. Крім того, TypeScript покращує підтримку великих проєктів, забезпечуючи кращу автодоповнення та рефакторинг у редакторах коду [22-23].

React Router: бібліотека для управління навігацією у вебзастосунку. Вона дозволяє створювати маршрути для різних сторінок, наприклад, для перегляду листів чи налаштувань користувача [24].

React Hook Form: бібліотека для обробки форм, яка оптимізує валідацію та управління станом форм, таких як форма введення даних поштової скриньки [25].

Material-UI: бібліотека компонентів для створення сучасного та адаптивного дизайну. MUI надає готові компоненти, такі як таблиці, кнопки чи діалогові вікна, що прискорюють розробку та забезпечують єдиний стиль [26].

Tanstack React Query: інструмент для асинхронного управління даними з backend. React Query оптимізує запити до API, кешує відповіді та синхронізує дані, що зменшує навантаження на сервер і покращує продуктивність [27].

Zustand: легка бібліотека для управління глобальним станом, яка використовується для зберігання даних користувача чи налаштувань інтерфейсу. Zustand є простішим і швидшим порівняно з Redux, що робить його ідеальним для проєктів середнього розміру [28].

i18next: бібліотека для локалізації інтерфейсу, яка дозволяє підтримувати різні мови (наприклад, українську, англійську). Це важливо для залучення міжнародних користувачів [29].

Vite: швидкий інструмент для збирання frontend, який забезпечує миттєвий запуск серверу розробки та оптимізовану продакшен-збірку [30].

Обґрунтування: React і TypeScript створюють надійну основу для модульного та типізованого frontend, що зменшує помилки та прискорює розробку. MUI та React Hook Form спрощують створення UI, Tanstack React Query оптимізує роботу з API, а Zustand і i18next забезпечують гнучкість і локалізацію. Vite прискорює процес розробки порівняно з традиційними інструментами, такими як Webpack [31].

1.4.2 Backend

.NET 8: сучасна платформа для створення продуктивних і безпечних RESTful API. ASP.NET Core, який є частиною .NET 8, підтримує асинхронну обробку запитів, що ідеально для роботи з великою кількістю користувачів [32-34].

Entity Framework Core: ORM (Object-Relational Mapping) для роботи з MySQL. EF Core спрощує управління мультитенантними базами даних, дозволяючи створювати запити через LINQ і автоматично мапувати об’єкти на таблиці [35-36].

AutoMapper: бібліотека для мапінгу об’єктів, яка використовується для перетворення моделей бази даних у DTO і навпаки. AutoMapper зменшує обсяг boilerplate-коду та забезпечує чистоту архітектури [37].

MediatR: бібліотека для реалізації CQRS-патерну, яка розділяє запити (queries) і команди (commands). Це підвищує модульність і спрощує організацію логіки [38-40].

FluentValidation: бібліотека для валідації вхідних даних, яка захищає API від некоректних запитів, наприклад, неправильних форматів email чи паролів [41].

JWT-автентифікація: механізм для безпечної автентифікації та авторизації користувачів. JWT-токени містять зашифровану інформацію про користувача, що дозволяє перевіряти його права доступу [42].

MailKit: бібліотека для роботи з протоколом IMAP, яка забезпечує синхронізацію з поштовими скриньками. MailKit підтримує популярні поштові сервери та забезпечує стабільну роботу з великими обсягами листів [1, 43].

AWSSDK.S3: бібліотека для інтеграції з сервісом Amazon S3, яка використовується для зберігання вкладень. S3 забезпечує високу доступність і масштабованість [3, 44].

Serilog: бібліотека для структурованого логування, яка дозволяє зберігати логи у зрозумілому форматі (JSON) і інтегрувати їх із зовнішніми системами, такими як Seq чи Elasticsearch. Serilog використовується для відстеження дій користувачів і діагностики помилок [45].

Swashbuckle.AspNetCore: бібліотека для генерації OpenAPI-специфікації та створення інтерактивної документації API через Swagger UI. Swagger дозволяє розробникам і тестувальникам швидко ознайомитися з ендпоінтами API та протестувати їх [46-48].

Обґрунтування: .NET 8 забезпечує високу продуктивність і безпеку, Entity Framework Core і AutoMapper спрощують роботу з даними, MediatR і FluentValidation підвищують модульність і безпеку, а MailKit і AWSSDK.S3 надають надійні інструменти для інтеграції. Serilog забезпечує детальне логування, а Swashbuckle/Swagger спрощує документування та тестування API.

1.4.3 Інфраструктура та бази даних

MySQL 8.3: реляційна база даних для зберігання метаданих листів, користувачів і поштових скриньок.

MySQL підтримує мультитенантність через окремі схеми та має високу продуктивність для реляційних даних [4, 49].

Amazon S3 (LocalStack для тестування): хмарне сховище для вкладень, яке забезпечує масштабованість, надійність і низьку вартість зберігання. LocalStack дозволяє тестувати S3 локально, зменшуючи витрати на розробку [3, 50].

Docker: інструмент для контейнеризації застосунку, бази даних і LocalStack. Docker спрощує розгортання та забезпечує однакове середовище на різних платформах [51].

LocalStack: інструмент для локального тестування хмарних сервісів, таких як S3, що дозволяє розробляти та тестувати застосунок без доступу до реальних хмарних ресурсів [50].

Обґрунтування: MySQL є перевіреною базою даних із широкою підтримкою, Amazon S3 забезпечує масштабоване зберігання, а Docker і LocalStack спрощують розгортання та тестування, зменшуючи залежність від платних сервісів.

1.4.4 Інструменти розробки

TypeScript: використовується для типізації коду на frontend та backend. Переваги TypeScript над JavaScript включають статичну перевірку типів, що знижує ймовірність помилок, кращу підтримку IDE та можливість роботи з великими кодовими базами.

ESLint і Vite Plugin ESLint: інструменти для забезпечення якості коду та відповідності стандартам. ESLint виявляє потенційні помилки та забезпечує єдиний стиль коду [52].

OpenAPI Codegen: інструмент для генерації клієнтського коду на основі OpenAPI-специфікації, що спрощує інтеграцію frontend з backend.

Vite: швидкий інструмент для збирання frontend, який підтримує гаряче перезавантаження та оптимізацію.

Обґрунтування: TypeScript і ESLint підвищують якість і підтримуваність коду, OpenAPI Codegen прискорює інтеграцію, а Vite забезпечує швидкий цикл розробки.

1.5 Висновки до розділу

Аналіз предметної області показав, що бекап електронних листів є критично важливим для захисту даних від втрат, викликаних технічними збоями, кібератаками чи людськими помилками. Ключові аспекти бекапу включають надійність, доступність, автоматизацію, масштабованість і безпеку, що формують основу для розробки EmailTamer. Аналіз аналогічних сервісів виявив їхні сильні та слабкі сторони: MailStore підходить для корпоративного використання, але складний у налаштуванні; Nylas гнучкий для розробників, але дорогий; CloudHQ простий, але не підтримує мультитенантність; Dropsuite безпечний, але коштовний. EmailTamer поєднує мультитенантну архітектуру, хмарне зберігання, простий інтерфейс і низьку вартість, що робить його конкурентоспроможним.

Вимоги до сервісу включають підтримку JWT-автентифікації, IMAP-синхронізації, збереження даних у MySQL і S3, а також пошук і фільтрацію листів. Для реалізації обрано стек технологій: React і TypeScript для frontend, .NET 8 з Entity Framework Core, AutoMapper, MediatR, Serilog і Swashbuckle для backend, MySQL і Docker для інфраструктури. TypeScript обрано замість JavaScript через статичну типізацію, що зменшує помилки та полегшує підтримку коду. Ці технології забезпечують баланс між продуктивністю, безпекою та зручністю розробки, закладаючи основу для подальшого проєктування та реалізації.

2 ПРОЄКТУВАННЯ ТА МОДЕЛЮВАННЯ СЕРВІСУ

2.1 Загальна структура сервісу

2.1.1 Архітектурний огляд

Вебзастосунок EmailTamer побудовано за клієнт-серверною архітектурою, яка складається з наступних компонентів: frontend, backend, бази даних та сервісу-

сховища файлів, загальну схему наведено на рисунку 2.1). Кожен компонент спроєктовано для забезпечення модульності, масштабованості та ізоляції даних між користувачами (тенантами). Архітектура відповідає принципам чистої архітектури (Clean Architecture) та CQRS, що сприяє чіткому розділенню обов'язків і спрощує підтримку системи [38, 40].

Frontned відповідає за взаємодію з користувачем через вебінтерфейс. Реалізовано з використанням бібліотеки React для створення реактивних компонентів, TypeScript для статичної типізації, MUI для адаптивного дизайну, Tanstack React Query для асинхронного управління даними та Zustand для глобального стану [21, 22, 26, 27, 28]. Frontend комунікує з backend через RESTful API, використовуючи згенеровані TypeScript-хуків (OpenAPI Codegen) для типобезпечних запитів [53].

Backend обробляє бізнес-логіку, запити користувачів і інтеграцію з зовнішніми сервісами. Побудовано на платформі .NET 8 з ASP.NET Core, використовуючи MediatR для CQRS, Entity Framework Core для роботи з MySQL, AutoMapper для мапінгу об'єктів, FluentValidation для валідації даних, MailKit для синхронізації з IMAP і AWSSDK.S3 для роботи з хмарним сховищем [32, 35, 37, 38, 41, 43, 44]. Backend підтримує JWT-автентифікацію та логування через Serilog [42, 45]. API документується через Swashbuckle (Swagger) [46].

Інфраструктура включає реляційну базу даних MySQL 8.3 для зберігання метаданих (користувачі, листи, поштові скриньки), Amazon S3 для зберігання вкладень і Docker для контейнеризації застосунку [4, 44, 51]. Для локального тестування S3 використано LocalStack, який емулює хмарні сервіси [50]. Мультитенантність реалізовано через окремі БД MySQL і бакети (сховища) S3 для кожного тенанта [5-7].

Архітектура побудована з урахуванням нефункціональних вимог, таких як масштабованість, продуктивність і безпека [14]. Використання Docker забезпечує однакове середовище для розробки, тестування та розгортання, а LocalStack зменшує витрати на тестування хмарних сервісів [50, 51].

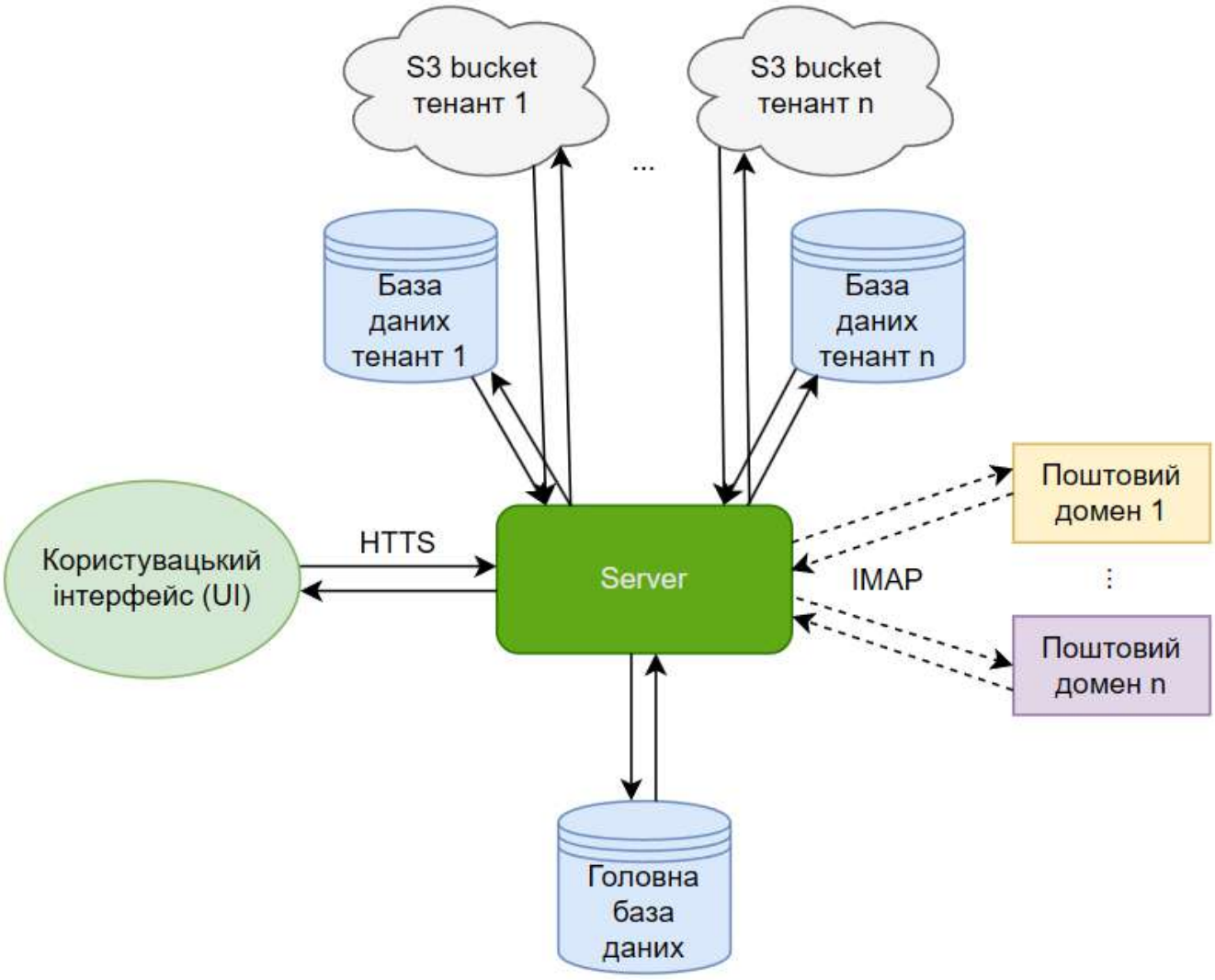


Рисунок 2.1 – Загальна схема застосунку

2.1.2 Мультитенантна архітектура

Мультитенантна архітектура є ключовою особливістю EmailTamer, оскільки вона забезпечує логічну та фізичну ізоляцію даних для кожного окремого користувача (“тенанту”), що критично важливо для обробки конфіденційної інформації, такої як електронні листи.

З архітектурної точки зору, система реалізує окрему базу даних для кожного тенанта (Database-per-Tenant) та окреме сховище об'єктів для кожного тенанта (Bucket-per-Tenant). Це означає, що кожен тенант має власну, незалежну базу даних для зберігання метаданих електронної пошти та власний, окремий бакет у сховищі S3 для зберігання тіл листів та вкладень. Ця модель забезпечує найвищий рівень ізоляції даних, унеможливлюючи випадковий або зловмисний доступ до даних

одного тенанта з боку іншого на рівні інфраструктури зберігання.

Кожен користувач має унікальний ідентифікатор (TenantId), який визначає його контекст у системі. Контекст включає з’єднання до окремої бази даних у MySQL і окремий бакет (сховище) у Amazon S3, що гарантує, що дані одного користувача недоступні іншому [5-7]. Реалізація мультитенантності:

- ідентифікація тенанта: кожен зареєстрований користувач (EmailTamerUser) асоційований з унікальним ідентифікатором тенанта (TenantId). Поточний тенант для активного HTTP-запиту визначається на основі ідентифікатора автентифікованого користувача;

- аксесор контексту тенанта: сервіс, відповідальний за визначення та надання ідентифікатора поточного тенанта (TenantId), а також за генерацію імен ресурсів, специфічних для цього тенанта (наприклад, назви бази даних та бакета S3);

- фабрика контексту бази даних тенанта: відповідає за динамічне створення екземпляра контексту БД, налаштованого на підключення до конкретної бази даних поточного тенанта, використовуючи ім'я бази даних, отримане з аксесору тенанту;

- фабрика репозиторію тенанта: відповідає за створення екземпляра репозиторію, налаштованого на роботу з конкретним бакетом S3 поточного тенанта, використовуючи ім'я бакета, отримане з аксесору тенанту;

- сервіс шифрування тенанта: надає функціональність для шифрування та дешифрування даних (наприклад, облікових даних для підключення до поштової скриньки), використовуючи симетричний ключ (AES-256), що генерується на основі ідентифікатора поточного тенанта (TenantId) та секретної солі, що зберігається в конфігурації. Це забезпечує шифрування даних у спокої з унікальним ключем для кожного тенанта.

Переваги мультитенантності:

- максимальна безпека та приватність: дані тенантів фізично розділені, що мінімізує ризик витоку даних між ними;

- гнучкість схеми (для БД): теоретично, схема бази даних може еволюціонувати по-різному для різних тенантів (хоча в поточній реалізації це не використовується);

- масштабованість тенанта: легше масштабувати ресурси (БД, сховище) для конкретного "важкого" тенанта;

- спрощення резервного копіювання/відновлення: можна легко зробити бекап або відновити дані одного тенанта.

Недоліки:

- вища вартість інфраструктури: утримання окремої БД та бакета для кожного тенанта дорожче, ніж використання спільної інфраструктури;

- складність управління: збільшення кількості баз даних та бакетів ускладнює управління, моніторинг та розгортання міграцій (потрібні інструменти для застосування міграцій до всіх баз даних тенантів);

- складність міжтенантних операцій: виконання запитів або аналітики, що охоплюють дані кількох тенантів, стає дуже складним.

Виклики та рішення:

- складність управління: автоматизація створення контекстів БД (з’єднань до окремих БД) MySQL реалізовано за допомогою TenantDbContextFactory, що наведено в лістингу 3.2, а актуальна конфігурація таблиць в БД через міграції EF Core, які застосовуються при з’єднанні чи реєстрації нового тенанта;

- тестування: LocalStack дозволяє створювати тестові бакети S3 для кожного тенанта без витрат на реальні хмарні ресурси [50].

2.2 Моделювання сутностей БД

Моделювання даних є важливою частиною проєктування EmailTamer, оскільки воно визначає, як зберігатимуться метадані листів, налаштування поштових скриньок і вкладення. Модель даних розроблено з урахуванням мультитенантності, безпеки та продуктивності.

Головна БД містить лише EmailTamerUser та схеми Identity Framework. Схему наведено на рисунку 2.2.

EmailTamerUser (Користувач EmailTamer):

- Id (string): первинний ключ, згенерований Identity;
- Email (string): електронна адреса користувача;
- PasswordHash (string): хеш паролю користувача;
- TenantId (Guid): ідентифікатор тенанта користувача;
- інші (різні, не задіяні): стандартні властивості користувача ASP.NET Identity.

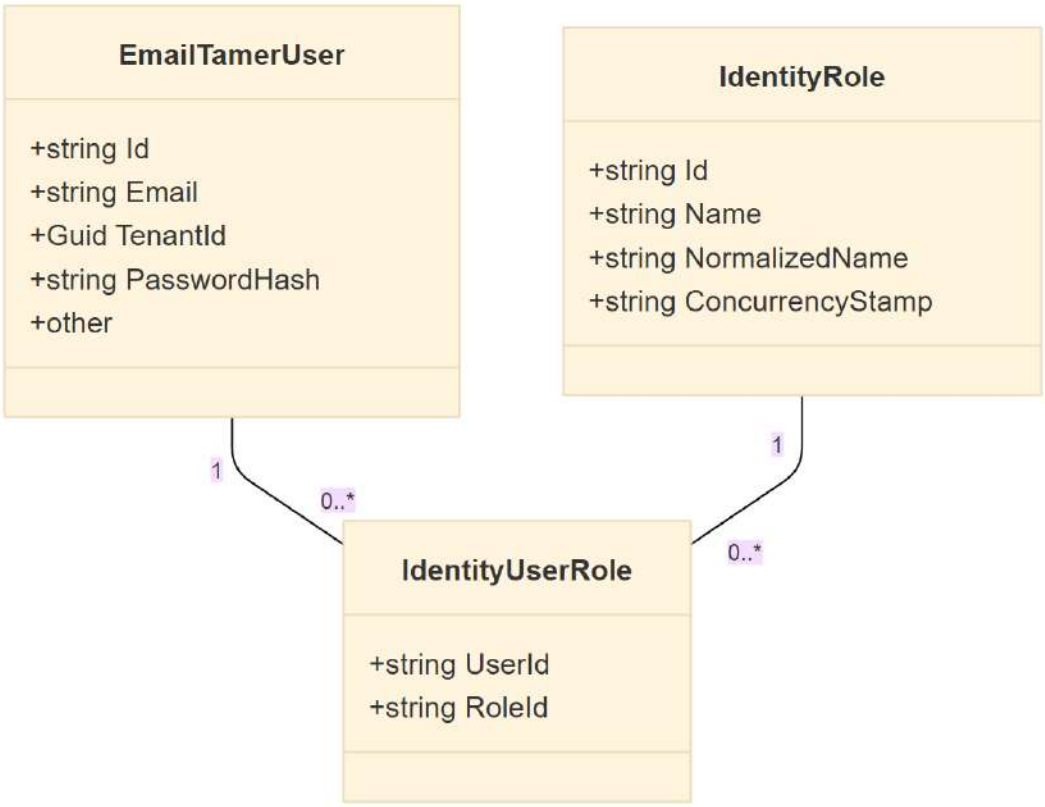


Рисунок 2.2 - Схема головної БД

БД тенанту містить такі сутності:

а) EmailBox (Поштова скринька):

1) Id (Guid): первинний ключ;

2) BoxName (string): відображуване ім'я поштової скриньки;

3) UserName (string): ім'я користувача для автентифікації (зашифроване);

4) Email (string): електронна адреса (зашифрована);

5) Password (string): пароль для автентифікації (зашифрований);

6) EmailDomainConnectionHost (string): ім'я хоста IMAP-сервера (зашифроване);

7) EmailDomainConnectionPort (int): порт IMAP-сервера (за замовчуванням: 993);

8) UseSSl (bool): чи використовувати SSL для з'єднання (за замовчуванням: true);

9) AuthenticateByEmail (bool): чи автентифікуватися за допомогою електронної пошти замість імені користувача;

10) LastSyncAt (DateTime): часова мітка останньої успішної синхронізації;

11) ConnectionFault (ConnectionFault?): зберігає інформацію про помилки з'єднання;

12) BackupStatus (BackupStatus): поточний статус операцій резервного копіювання;

13) CreatedAt (DateTime): коли була створена поштова скринька;

14) ModifiedAt (DateTime): коли поштова скринька була востаннє змінена;

б) Message (Повідомлення):

1) Id (string): первинний ключ (ідентифікатор повідомлення електронної пошти);

2) ThreadId (string): ідентифікатор для групування повідомлень у ланцюжки розмов;

3) Subject (string): тема електронного листа;

4) TextBody (string): текстовий вміст електронного листа;

5) HasHtmlBody (bool): вказує, чи існує HTML-версія (зберігається в S3);

6) Date (DateTime): дата відправлення повідомлення;

7) ResentDate (DateTime?): дата пересилання повідомлення;

8) InReplyTo (string): посилання на повідомлення, на яке відповідають;

9) References (List<string>): ідентифікатори повідомлень, на які посилаються в цьому повідомленні;

10) From, To (List< EmailAddress>): JSON-серіалізовані адреси відправників та отримувачів (зберігаються як JSON, що наведено в лістингу 2.1);

11) Attachments (List<Attachment>): колекція вкладених файлів;

12) Folders (List<Folder>): папки, що містять це повідомлення;

13) EmailBoxes (List<EmailBox>): поштові скриньки, що містять це повідомлення;

в) Folder (Папка):

1) Id (Guid): первинний ключ;

2) Name (string): назва папки;

3) Messages (List<Message>): повідомлення, що містяться в цій папці;

г) Attachment (Вкладення):

1) Id (Guid): первинний ключ;

2) FileName (string): назва прикріпленого файлу;

3) MessageId (string): зовнішній ключ до батьківського повідомлення;

4) Message (Message): навігаційна властивість до батьківського повідомлення;

д) EmailAddress (Адреса електронної пошти):

1) Name (string?): ім'я відправника/отримувача;

2) Address (string): адреса електронної пошти;

3) Domain (string): домен електронної пошти.

Лістинг 2.1 - Приклад серіалізації From/To

```
[{"Name": "John Doe", "Address": "john@example.com", "Domain": "example.com"},
  {"Name": null, "Address": "jane@example.com", "Domain": "example.com"}]
```

БД тенанту також містить такі перелічення (enum):

а) ConnectionFault:

1) Other = 1;

2) ConnectionRefused = 2;

3) PortOutOfRange = 3;

4) WrongAuthenticationCredentials = 4;

б) BackupStatus:

1) Idle = 0;

2) Queued = 1;

3) InProgress = 2;

4) Failed = 3.

Схему БД тенанту наведено на рисунку 2.3.

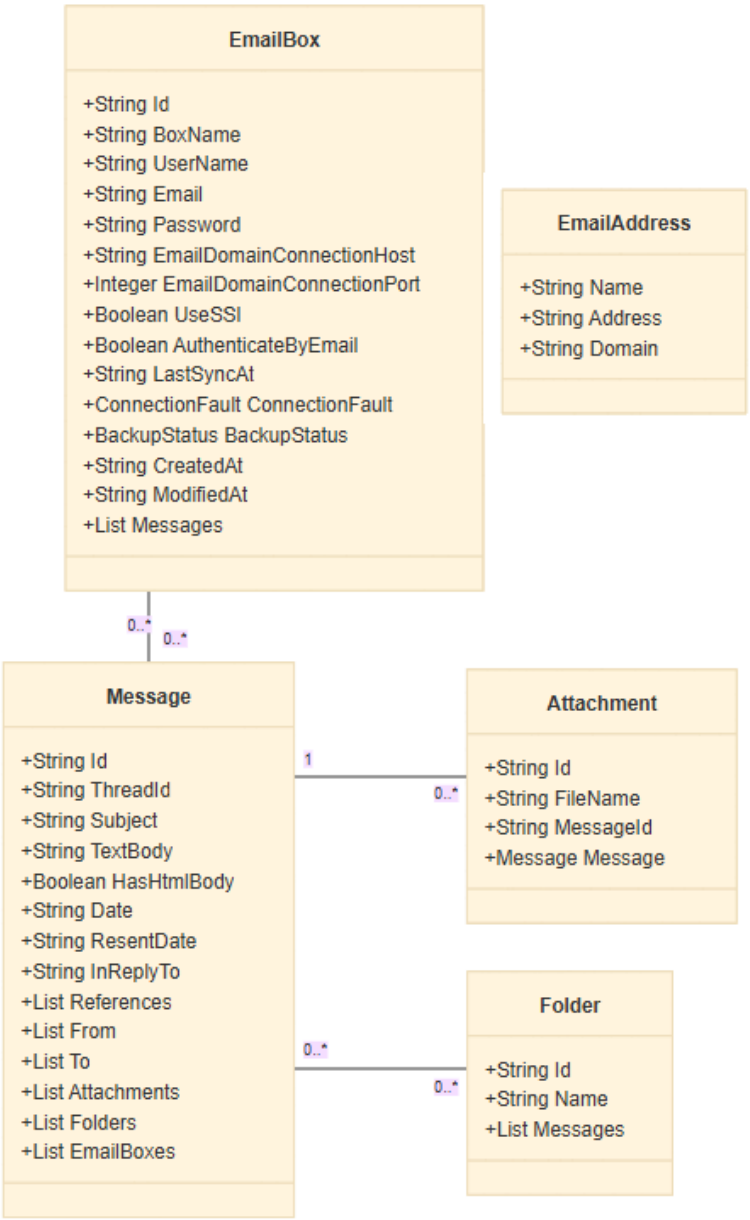


Рисунок 2.3 – Схема бази даних тенанту

2.3 Структура клієнтської частини

Клієнтська частина вебзастосунку EmailTamer відповідає за взаємодію з користувачем через вебінтерфейс, забезпечуючи зручне керування поштовими скриньками, перегляд листів, пошук і завантаження вкладень. Frontend побудовано на основі бібліотеки React із використанням TypeScript для статичної типізації, що зменшує помилки та полегшує підтримку коду [21, 22]. Для створення адаптивного дизайну використано MUI, а для асинхронного управління API-запитами — Tanstack React Query [26, 27]. Глобальний стан керується через Zustand, локалізація підтримується i18next, а маршрутизація реалізована через React Router [24, 28, 29]. Інструмент Vite забезпечує швидке збирання та гаряче перезавантаження під час розробки [30]. Клієнтська частина комунікує з backend через RESTful API, використовуючи згенеровані TypeScript-хуків на основі OpenAPI-специфікації [53]. Архітектуру frontend наведено на рисунку 2.4.

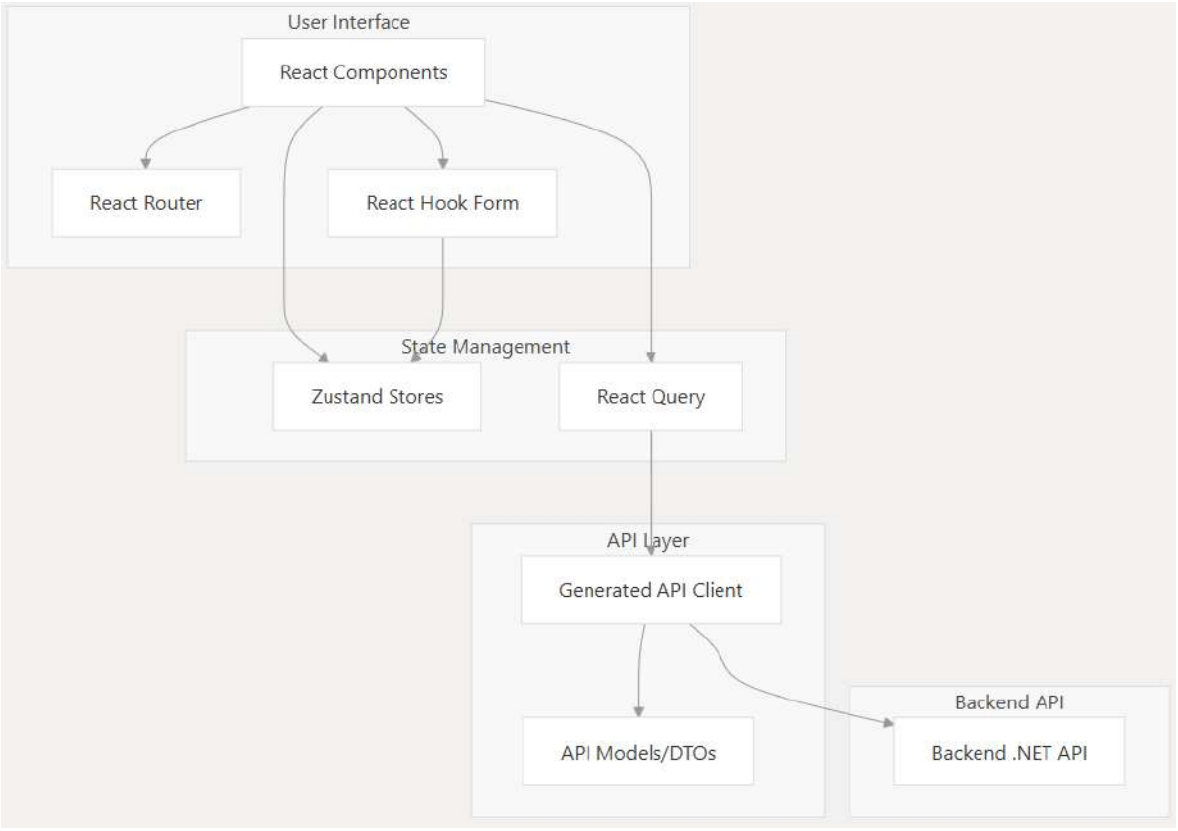


Рисунок 2.4 – Загальна архітектура клієнтської частини

2.3.1 Компонентна архітектура та маршрутизація

Основні компоненти системи:

а) компоненти сторінок:

1) Layout - базовий компонент, який містить спільні елементи для всіх сторінок, включаючи Header;

2) HomePage - головна сторінка для керування електронною поштою, містить три основні секції;

3) ThreadPage - сторінка для перегляду окремого ланцюжку повідомлень;

4) LoginPage - сторінка входу в систему;

5) RegisterPage - сторінка реєстрації нового користувача;

6) DemoPage - демонстраційна сторінка для неавторизованих користувачів;

7) NotFoundPage - сторінка 404 для неіснуючих маршрутів;

б) компоненти головної сторінки:

1) FoldersSection - секція для відображення та вибору папок електронної пошти;

2) MessagesSection - центральна секція для відображення списку ланцюжків повідомлень;

3) EmailBoxesSection - секція для управління поштовими скриньками та ініціювання резервного копіювання;

в) компоненти повідомлень:

1) ThreadPreview - компонент для відображення прев’ю ланцюжка повідомлень у списку;

2) Message - компонент для відображення окремого повідомлення з можливістю розгортання;

3) Attachment - компонент для відображення та завантаження вкладень;

4) PagesHandler - компонент для пагінації списку повідомлень;

г) компоненти поштових скриньок:

1) AddEmailBoxDialogForm - форма для додавання нової поштової скриньки;

2) EditEmailBoxDialogForm - форма для редагування існуючої поштової скриньки;

3) EmailBoxMoreMenu - контекстне меню для додаткових дій з поштовою скринькою;

д) спільні компоненти:

1) Header - шапка додатку з логотипом, пошуком та елементами авторизації;

2) SearchBar - компонент для пошуку повідомлень;

3) LanguageSelector - компонент для вибору мови інтерфейсу;

4) AuthControls - компоненти для входу/виходу та реєстрації;

5) GenericEmailTamerList - універсальний компонент списку для відображення різних типів даних;

6) ContentLoading - компонент для відображення стану завантаження.

е) компоненти маршрутизації:

1) Router - головний компонент маршрутизації;

2) GuardedRoute - компонент для захисту маршрутів від неавторизованого доступу;

3) TranslationScopeProvider - компонент для забезпечення перекладів у відповідному контексті;

є) хуки та утиліти:

1) useUrlParam - хук для роботи з URL-параметрами;

2) useScopedContextTranslator - хук для отримання перекладів у поточному контексті;

3) useAuthStore - хук для роботи з авторизаційними даними;

4) formatDateTime - утиліта для форматування дати та часу.

Ця архітектура забезпечує гнучкість, масштабованість та зручність підтримки додатку, дозволяючи ефективно керувати електронною поштою з різних джерел через єдиний інтерфейс. Потік даних наведено на рисунку 2.5.

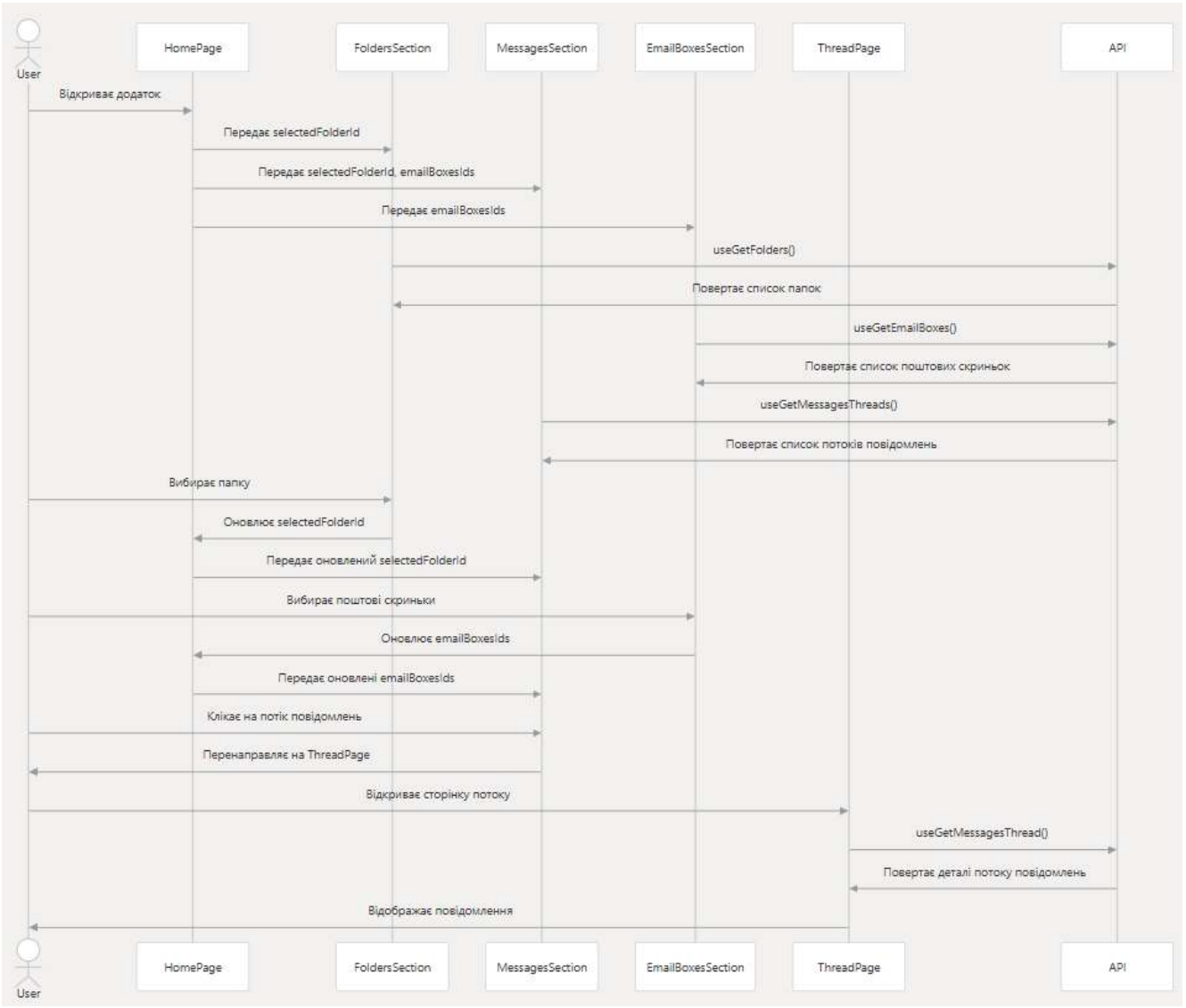


Рисунок 2.5 – Потік даних між компонентами

Маршрутизація в додатку реалізована за допомогою React Router. Структуру маршрутів наведено на рисунку 2.6.

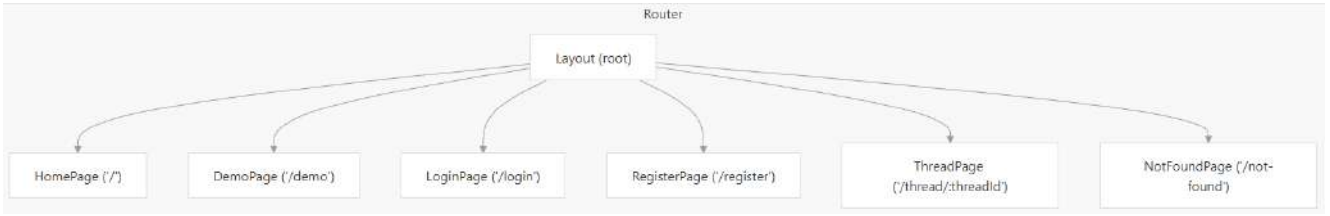


Рисунок 2.6 – Структура маршрутів клієнтської частини

2.3.2 Управління станом

Frontend використовує комбінацію локального та глобального управління станом:

- локальний стан: керується через useState, useReducer та інші хуки, також через url-параметри. Зв'язок наведено на рисунку 2.7;

- глобальний стан: Zustand зберігає дані автентифікації (схему наведено на рисунку 2.8) і налаштування інтерфейсу (мова, інше);

- API-стан: Tanstack React Query кешує відповіді API, синхронізує дані та обробляє стани завантаження/помилок.

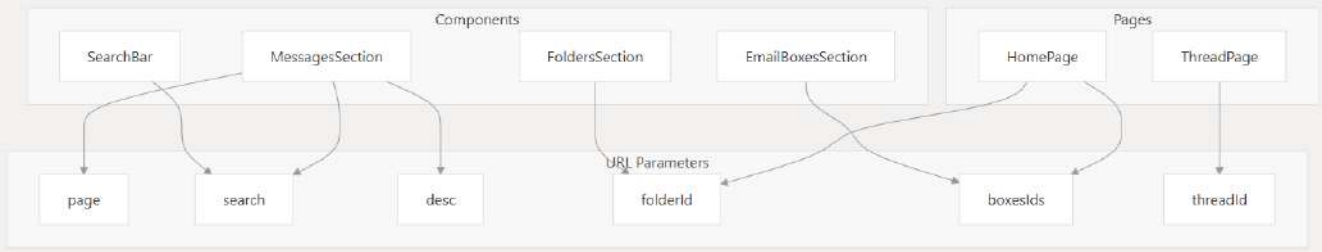


Рисунок 2.7 – Стан додатку та URL-параметри

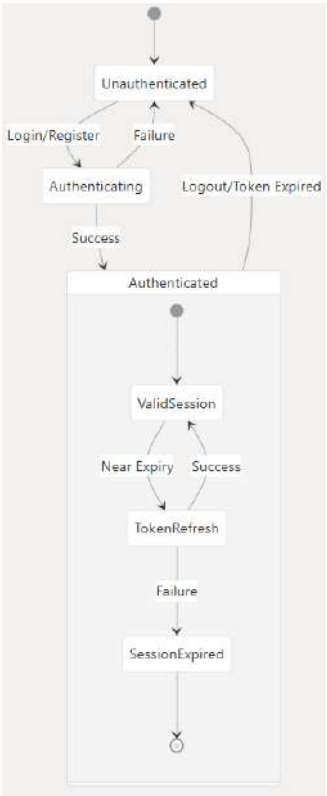


Рисунок 2.8 – Керування станом AuthStore

2.3.3 API-взаємодія

Frontend взаємодіє з backend через RESTful API, використовуючи згенеровані TypeScript-хуки (OpenAPI Codegen). Такі хуки забезпечують типобезпеку та автодоповнення, що є зручним для розробки та відлагодження. Приклад згенерованого хуку наведено в лістингу 2.2.

Лістинг 2.2 – Хук useRegister

```
export type RegisterError = Fetcher.ErrorWrapper<undefined>;


export type RegisterVariables = {
  body?: Schemas.CreateUserDto;
} & EmailTamerApiContext['fetcherOptions'];


export const fetchRegister = (
    variables: RegisterVariables,
    signal?: AbortSignal,
) =>
    emailTamerApiFetch<
    undefined,
    RegisterError,
    Schemas.CreateUserDto,
    {},
    {},
    {}
  >({ url: '/api/auth/register', method: 'post', ...variables,
signal });

export const useRegister = (
    options?: Omit<
    reactQuery.UseMutationOptions<undefined, RegisterError,
RegisterVariables>,
    'mutationFn'
  >,
) => {
    const { fetcherOptions } = useEmailTamerApiContext();
    return reactQuery.useMutation<undefined, RegisterError,
RegisterVariables>({
        mutationFn: (variables: RegisterVariables) =>
            fetchRegister(deepMerge(fetcherOptions, variables)),
        ...options,
    });
};
```

2.4 Структура серверної частини

Серверна частина відповідає за обробку бізнес-логіки, взаємодію з інфраструктурою (MySQL, S3, IMAP) і надання API для frontend. Backend побудовано на .NET 8 з ASP.NET Core, що забезпечує високу продуктивність і асинхронну обробку запитів [32, 34]. Для реалізації використано Entity Framework Core для роботи з MySQL, AutoMapper для мапінгу, MediatR для CQRS, FluentValidation для валідації, MailKit для IMAP, AWSSDK.S3 для S3, Serilog для логування та Swashbuckle для документації API [35, 37, 38, 41, 43-46].

2.4.1 Модульна організація

Серверна частина Email Tamer побудована на принципі модульності, що забезпечує чіткий розподіл функціональності та відповідальності між компонентами системи. Кожен модуль на рівні ASP.NET Core є окремим проєктом, що об’єднані в Solution ("рішення" або "проєкт").Схему ієрархії модулів наведено на рисунку 2.9. Кожен модуль має чітко визначену відповідальність:

- Application - є точкою входу в серверну частину застосунку, виконує конфігурацію та інтеграцію модулів, запускає вебсервер;

- Core - містить базові інтерфейси, моделі та утиліти, що використовуються в усіх інших модулях;

- Infrastructure - реалізує інфраструктурні сервіси (логування, конфігурація, тощо);

- Database - забезпечує доступ до основної бази даних та містить основну логіку взаємодії з БД;

- Database.Tenant - реалізує мультитенантний доступ до даних;

- Dependencies.Amazon - інтеграція з сервісами Amazon (S3);

- Parts - містить логіку інтеграції інших Parts модулів;

- Parts.Auth - відповідає за аутентифікацію та авторизацію;

- Parts.EmailBox - управління поштовими скриньками;

- Parts.Backup - резервне копіювання та доступ до повідомлень.

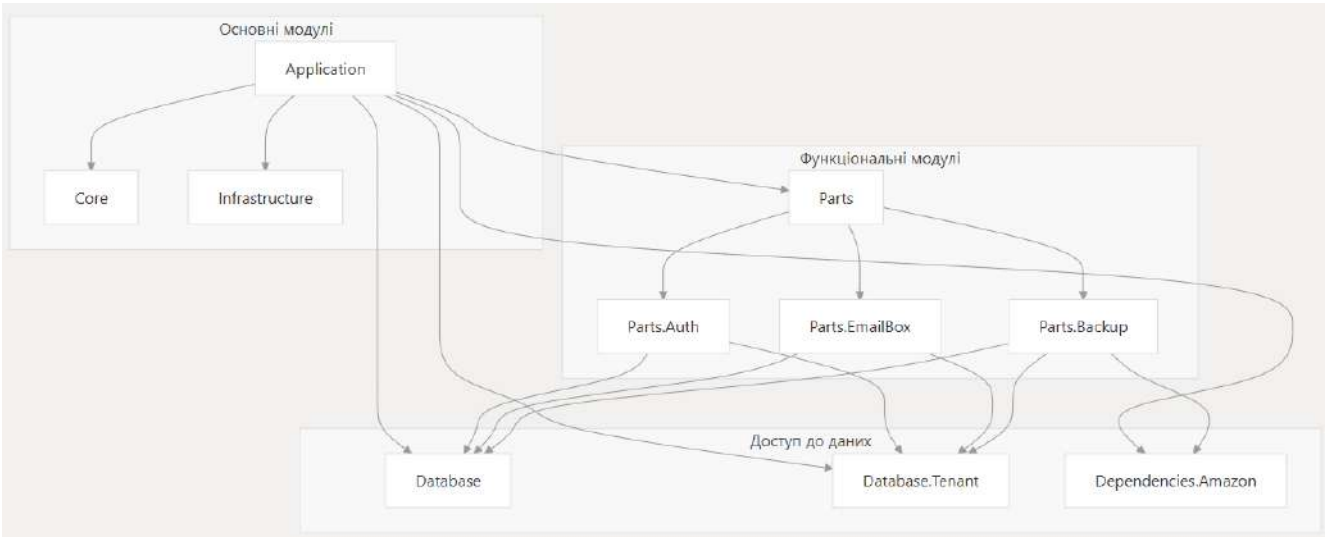


Рисунок 2.9 - Ієрархія модулів

Модуль Application є центральним компонентом, який інтегрує всі інші модулі. Він налаштовує та запускає веб-сервер, реєструє сервіси та middleware, а також конфігурує безпеку та логування.

Функціональні модулі (Auth, EmailBox, Backup) реалізують бізнес-логіку застосунку та надають API для взаємодії з клієнтською частиною. Вони використовують модулі доступу до даних (Database, Database.Tenant) для збереження та отримання даних.

Модуль Core містить базові інтерфейси та моделі, які використовуються всіма іншими модулями. Модуль Infrastructure реалізує інфраструктурні сервіси, такі як логування, конфігурація та інші.

Модуль Dependencies.Amazon забезпечує інтеграцію з сервісами Amazon, зокрема S3, для зберігання вмісту листів та вкладень.

2.4.2 Архітектурні патерни

Email Tamer використовує ряд сучасних архітектурних патернів, які забезпечують гнучкість, масштабованість та підтримуваність коду.

Command and Query Responsibility Segregation - це патерн, який розділяє операції читання (запити) та запису/дії (команди) в окремі моделі.

Суть патерну CQRS наочно продемонстровано діаграмою послідовності на рисунку 2.10.

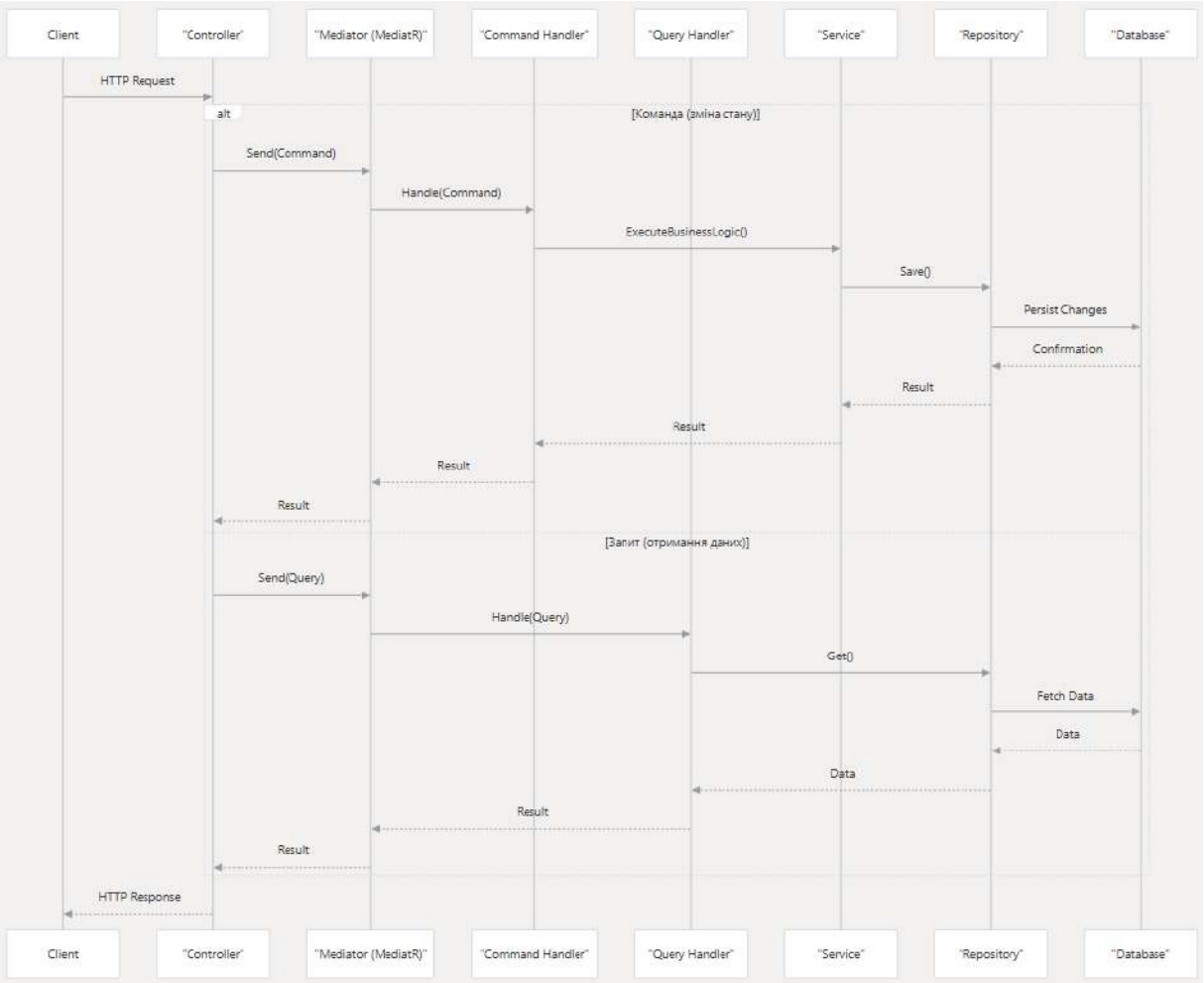


Рисунок 2.10 – Суть патерну CQRS

В Email Tamer цей патерн реалізований через:

а) команди (Commands) - класи, що представляють наміри змінити стан системи:

1) BackUpEmailBoxMessages - команда для резервного копіювання повідомлень;

2) CreateEmailBox - команда для створення поштової скриньки;

3) EditEmailBox - команда для редагування поштової скриньки;

4) DeleteEmailBox - команда для видалення поштової скриньки;

б) запити (Queries) - класи, що представляють наміри отримати дані без зміни стану:

1) GetEmailBoxesStatuses - запит для отримання статусів скриньок;

2) GetMessageDetails - запит для отримання деталей повідомлення;

3) GetMessageAttachment - запит для отримання вкладення;

4) GetMessagesThread - запит для отримання потоку повідомлень.

Переваги використання CQRS:

- розділення відповідальності між операціями читання та запису;

- можливість оптимізації кожного типу операцій окремо;

- спрощення масштабування та тестування.

Mediator Pattern використовується для зменшення залежностей між компонентами системи шляхом введення посередника.

У затосунку EmailTamer цей патерн реалізовано за допомогою бібліотеки MediatR.

Контролери не містять логіку самі по собі, а відправляють команди та запити через медіатор класам, що реалізують IRequestHandler.

Переваги використання Mediator Pattern:

- зменшення зв'язаності між компонентами;

- централізована обробка запитів;

- можливість додавання крос-функціональної логіки (логування, валідація, кешування).

Dependency Injection (DI) - патерн проєктування програмного забезпечення, що передбачає надання зовнішньої залежності програмному компоненту, використовуючи «інверсію керування» для розв'язання (отримання) залежностей.

Переваги використання DI:

- зменшення зв'язаності між компонентами;

- спрощення тестування через можливість заміни залежностей;

- централізоване управління життєвим циклом об'єктів.

Factory Pattern у цьому застосунку використовується в комбінації із DI для створення залежностей «на льоту», себто без прямої передачі необхідних параметрів напряму.

Переваги використання Factory Pattern:

- інкапсуляція логіки створення об'єктів;

- можливість створення об'єктів з різними параметрами;

- спрощення управління життєвим циклом об'єктів.

2.4.3 Сервіси та бізнес-логіка

Бізнес-логіка Email Tamer інкапсульована в спеціалізованих сервісах, які відповідають за конкретні функціональні аспекти. Діаграму взаємодії з зовнішніми сервісами наведено на рисунку 2.11.

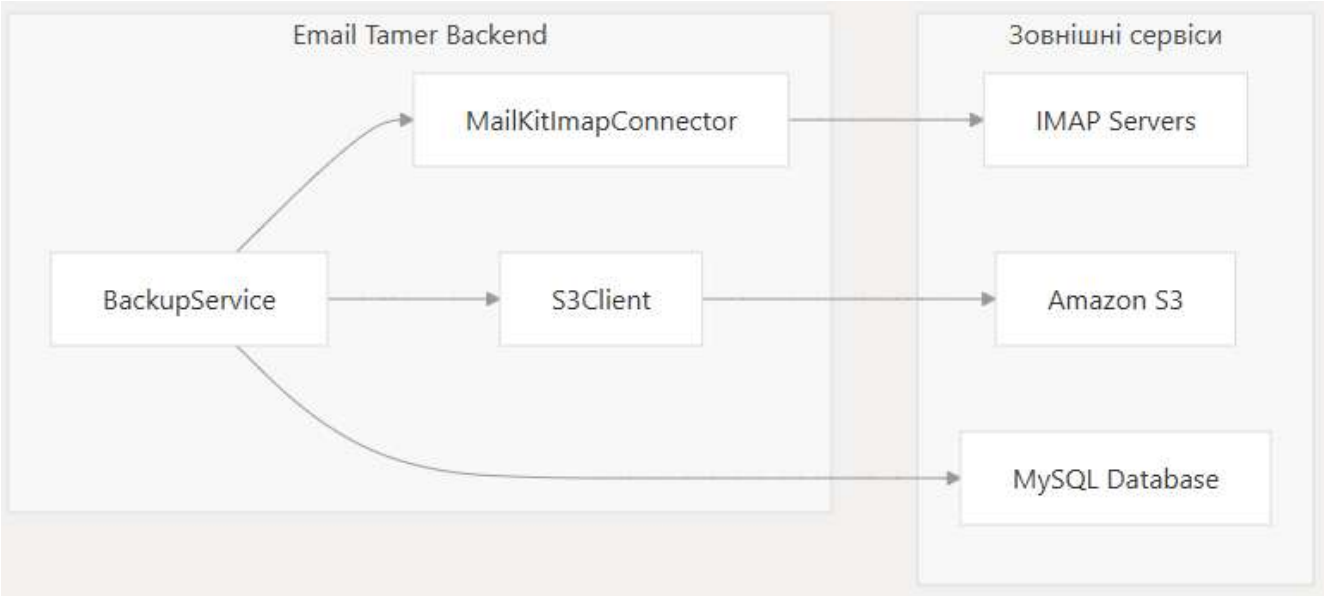


Рисунок 2.11 - Діаграма взаємодії з зовнішніми сервісами

MailKitImapConnector - це сервіс, що виконує підключення до поштових IMAP-серверів та авторизації за даними поштової скриньки за допомогою бібліотеки MailKit.

Основні функції MailKitImapConnector:

- підключення до IMAP-сервера з використанням налаштувань поштової скриньки;

- аутентифікація на сервері з використанням облікових даних;

- обробка помилок підключення та аутентифікації.

Обробка помилок включає:

- ConnectionRefused - сервер недоступний;

- PortOutOfRange - неправильний порт;

- WrongAuthenticationCredentials - неправильні облікові дані;

- Other - необроблена помилка.

BackupService - це ключовий сервіс, який містить основну бізнес логіку резервного копіювання повідомлень, їх HTML-вмісту (якщо він є) та вкладень (якщо вони є) з поштових скриньок.

Основні функції BackupService:

- підключення до IMAP-серверів через MailKitImapConnector;

- отримання повідомлень з поштових скриньок;

- збереження повідомлень в базі даних тенанта;

- збереження вкладень в S3 бакеті тенанта;

- організація повідомлень у потоки на основі заголовків References та In-Reply-To.

Процес резервного копіювання:

- завантаження початкових даних (поштова скринька, існуючі повідомлення, папки);

- оновлення статусу поштової скриньки;

- підключення до IMAP-сервера;

- отримання списку папок.

Для кожної папки:

- отримання повідомлень;

- обробка кожного повідомлення;

- збереження нових повідомлень та вкладень;

- оновлення статусу поштової скриньки;

- оновлення часу останньої синхронізації.

PeriodicBackupService - це фоновий сервіс, який автоматично запускає резервне копіювання поштових скриньок за допомогою BackupService для всіх тенантів за розкладом

Основні функції PeriodicBackupService:

- запуск за розкладом (наприклад, щодня о 1:00 UTC);

- отримання списку всіх тенантів з основної бази даних;

- запуск резервного копіювання для кожного тенанта;

- обробка помилок на рівні тенанта (помилка для одного тенанта не впливає на інших).

EncryptionService - це сервіс призначений для шифрування чутливих даних, таких як паролі, юзернейми та інші дані, що необхідні для входу до поштової скриньки.

Основні функції EncryptionService:

- генерація ключа шифрування на основі ідентифікатора тенанта;
- шифрування чутливих даних перед збереженням в базі даних;
- розшифрування даних при читанні з бази даних.

Процес шифрування:

- отримання ідентифікатора тенанта;
- генерація ключа шифрування з використанням PBKDF2;
- шифрування/розшифрування даних з використанням AES-256;
- збереження/читання зашифрованих даних в базі даних.

2.4.4 API та контролери

Email Tamer надає RESTful API через контролери, які забезпечують взаємодію з клієнтською частиною.

BackupsController - це контролер для операцій з резервним копіюванням, отримання статусу бекапу та доступу до вже збережених повідомлень, папок та вкладень.

Ендпоїнти контролеру BackupsController наведено в таблиці 2.1.

EmailBoxesController - це контролер для операцій (створення/читання/редагування/видалення) над поштовими скриньками користувача та для перевірки з’єднання із поштовим сервером і коректності реквізитів.

Ендпоїнти контролеру EmailBoxesController наведено в таблиці 2.2.

AuthController - це контролер, який надає API інтерфейс для реєстрації та входу користувача, а також отримання поглибленої інформації про поточного користувача системи.

Ендпоїнти контролеру AuthController наведено в таблиці 2.3.

Таблиця 2.1 – Ендпоїнти BackupsController

Ендпоінт	Метод	Опис	Доступ
/api/backup/{id}	POST	Запуск резервного копіювання поштової скриньки	Авторизований користувач
/api/backup	POST	Запуск резервного копіювання для кількох або усіх поштових скриньок	Авторизований користувач
/api/backup/emailBoxesStatuses	GET	Отримання статусів бекапу поштових скриньок	Авторизований користувач
/api/backup/message	GET	Отримання деталей повідомлення	Авторизований користувач
/api/backup/attachment	GET	Отримання вкладення	Авторизований користувач
/api/backup/thread	GET	Отримання ланцюжку повідомлень	Авторизований користувач
/api/backup	GET	Отримання списку ланцюжків повідомлень з пагінацією	Авторизований користувач

Таблиця 2.2 – Ендпоїнти EmailBoxesController

Ендпоінт	Метод	Опис	Доступ
/api/emailBoxes	POST	Створення нової поштової скриньки	Авторизований користувач
/api/emailBoxes	GET	Отримання списку поштових скриньок	Авторизований користувач
/api/emailBoxes/{id}	GET	Отримання деталей поштової	Авторизований користувач
/api/emailBoxes	PATCH	Редагування поштової скриньки	Авторизований користувач
/api/emailBoxes/{id}	DELETE	Видалення поштової скриньки	Авторизований користувач
/api/emailBoxes/testConnection	POST	Тестування підключення до поштового домену та входу до поштової скриньки	Авторизований користувач

Таблиця 2.3 – Ендпоїнти AuthController

Ендпоінт	Метод	Опис	Доступ
/api/auth/register	POST	Реєстрація нового користувача	Неавторизований користувач
/api/auth/login	POST	Вхід користувача в систему	Неавторизований користувач
/api/auth/user	GET	Отримання інформації про поточного користувача	Авторизований користувач

2.4.4 Процес аутентифікації

Сервер використовує JWT для аутентифікації користувачів. Діаграму послідовності аутентифікації наведено на рисунку 2.12.

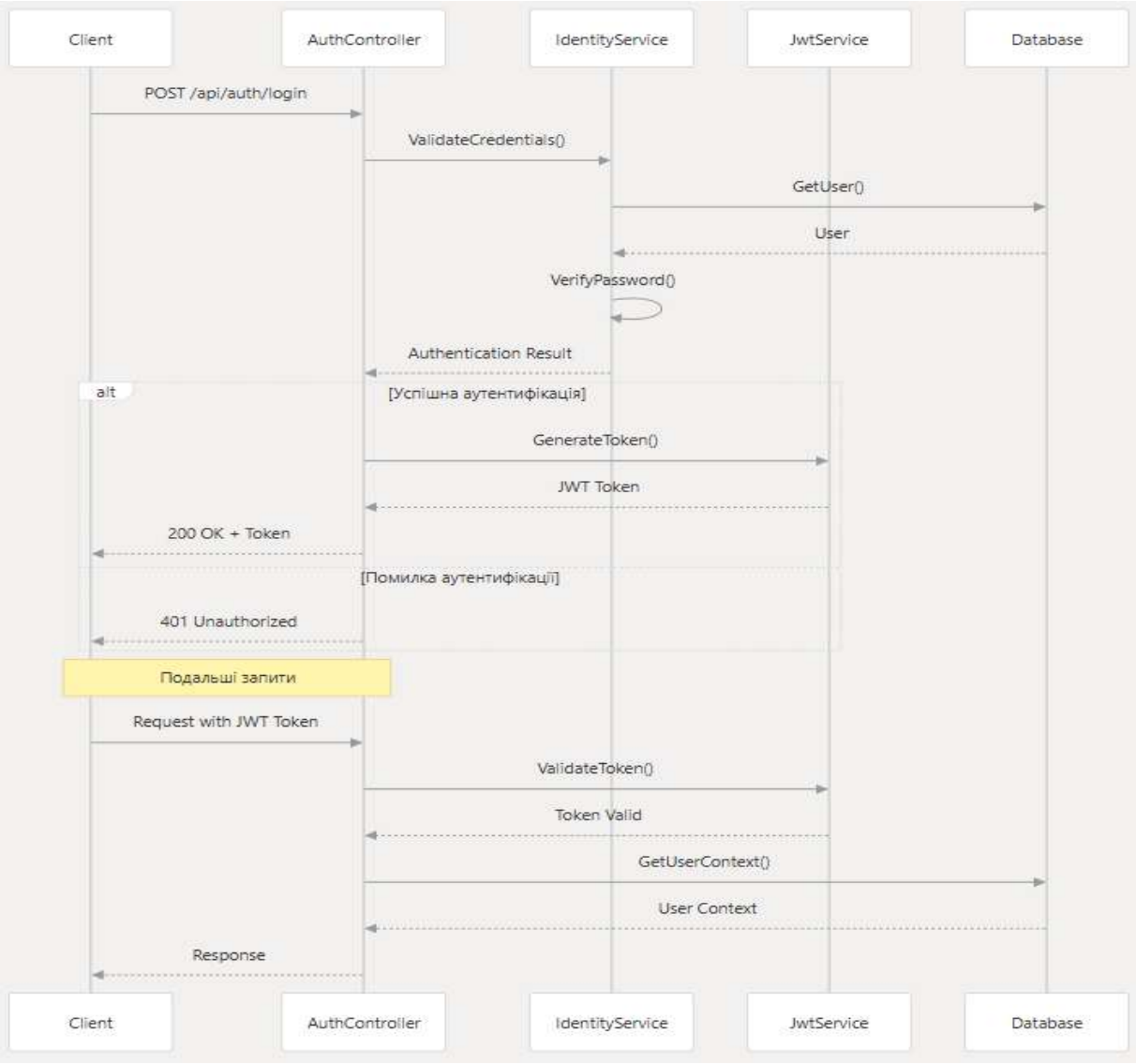


Рисунок 2.12 – Діаграма послідовності процесу аутентифікації

Процес аутентифікації включає:

- генерація токена: при успішному вході сервер генерує JWT-токен, який містить ідентифікатор користувача, його роль та інші необхідні дані;

- зберігання токена: клієнт зберігає отриманий токен (зазвичай у локальному сховищі або в пам'яті);

- використання токена: клієнт додає токен до заголовка Authorization при кожному запиті до API.

На серверній частині використовуються стандартні middleware ASP.NET Core для обробки аутентифікації та авторизації.

Послідовність middleware важлива:

- UseRouting() - визначає маршрутизацію запитів;

- UseAuthentication() - перевіряє аутентифікацію користувача;

- UseAuthorization() - перевіряє авторизацію користувача;

- UseCors() - застосовує політику CORS;

- UseEmailTamerAuth() - користувацький middleware для роботи з контекстом користувача.

Для роботи з контекстом користувача реалізовано спеціальний middleware AuthMiddleware, цей middleware:

- перевіряє, чи шлях запиту дозволений для анонімного доступу;

- отримує ідентифікатор користувача з JWT-токена;

- налаштовує контекст користувача для подальшого використання в застосунку.

2.5 Структура інфраструктури

Інфраструктура EmailTamer забезпечує зберігання даних, контейнеризацію, тестування та розгортання. Вона включає реляційну базу даних MySQL 8.3, хмарне сховище Amazon S3, інструмент контейнеризації Docker і локальний емулятор

хмарних сервісів LocalStack [4, 44, 50, 51]. Інфраструктура спроєктована для підтримки мультитенантності, масштабованості та безпеки.

Компоненти інфраструктури

а) MySQL 8.3:

1) зберігає метадані користувачів (EmailTamerUser) і поштових скриньок/листів (EmailBox, Message та ін.) у мультитенантних БД;

2) використовує пул підключень для оптимізації ресурсів;

3) міграції застосовуються через Entity Framework Core;

б) Amazon S3:

1) зберігає вкладення та HTML-вміст листів у бакетах;

2) забезпечує високу доступність і низьку затримку;

3) для тестування використовується LocalStack;

в) Docker: контейнеризує MySQL і LocalStack;

г) LocalStack: емулює Amazon S3 для локального тестування, зменшуючи витрати.

2.6 Висновки до розділу

У процесі проєктування змодельовано застосунок із клієнт-серверною архітектурою, яка включає frontend, backend та інфраструктуру, з урахуванням принципів чистої архітектури та CQRS для забезпечення модульності, масштабованості та легкості підтримки. Ключовою особливістю є мультитенантна архітектура, реалізована через окремі бази даних MySQL і бакети Amazon S3 для кожного користувача, що гарантує ізоляцію даних і підвищує безпеку.

Frontend, побудований на React і TypeScript, забезпечує реактивний і адаптивний вебінтерфейс із підтримкою локалізації (i18next), управління станом (Zustand) і асинхронних запитів (Tanstack React Query). Компонентна архітектура та маршрутизація через React Router дозволяють ефективно керувати поштовими

скриньками, переглядати листи, завантажувати вкладення та виконувати пошук. TypeScript забезпечує типобезпеку, що зменшує помилки, полегшує відладку коду та спрощує розробку.

Backend, реалізований на .NET 8 з ASP.NET Core, використовує Entity Framework Core для роботи з MySQL, AutoMapper для мапінгу даних між моделями, MediatR для CQRS і FluentValidation для валідації даних. Інтеграція з MailKit забезпечує синхронізацію через IMAP, а AWSSDK.S3 — збереження вкладень у хмарному сховищі Amazon S3. JWT-автентифікація та шифрування гарантують безпеку даних, а Serilog і Swashbuckle забезпечують структуроване логування та документацію API. Модульна організація backend (Application, Core, Infrastructure, Database, Parts та інші) сприяє чіткому розподілу відповідальностей між компонентами.

Інфраструктура включає MySQL 8.3 для метаданих, Amazon S3 для вкладень, Docker для контейнеризації та LocalStack для локального тестування, що спрощує розгортання та знижує витрати. Моделювання бази даних враховує мультитенантність, із головною базою для користувачів (EmailTamerUser) і окремими базами для кожного тенанта (EmailBox, Message, Folder, Attachment), що оптимізує безпеку та продуктивність.

Проєктування врахувало функціональні (автентифікація, синхронізація, пошук, фільтрація) і нефункціональні вимоги (масштабованість, продуктивність, зручність). Використання сучасних патернів (CQRS, Mediator, Factory, Dependency Injection) і технологій забезпечує гнучкість і надійність системи.

Розроблена архітектура та моделі даних створюють міцну основу для реалізації сервісу, що підтверджується детальними схемами, діаграмами та лістингами.

Результати цього етапу дозволяють перейти до реалізації програмного забезпечення, зберігаючи відповідність поставленим вимогам і забезпечуючи конкурентоспроможність EmailTamer серед аналогічних сервісів.

3 РЕАЛІЗАЦІЯ СЕРВІСУ

3.1 Реалізація мультитенантної архітектури

Мультитенантна архітектура є ключовою особливістю EmailTamer, що забезпечує ізоляцію даних між користувачами. Реалізація цієї архітектури базується на патерні "окрема база даних для кожного тенанта" (Database-per-Tenant) та "окреме сховище об'єктів для кожного тенанта" (Bucket-per-Tenant).

3.1.1 Реалізація контексту тенанта

Для ідентифікації поточного тенанта реалізовано клас TenantContextAccessor, який відповідає за визначення ідентифікатора тенанта та генерацію імен ресурсів. Для реєстрації сервісу в IoC-контейнері (Inversion of Control) також визначено інтерфейс ITenantContextAccessor, що наведено в лістингу 3.1.

Лістинг 3.1 – Інтерфейс ITenantContextAccessor та реалізація TenantContextAccessor

```
public interface ITenantContextAccessor{
    public Task<string> GetTenantId();
    public string GetDatabaseName();
    public string GetS3BucketName();
}

public sealed class TenantContextAccessor : ITenantContextAccessor{
    private IUserContextAccessor? _userContextAccessor;
    private UserManager<EmailTamerUser>? _userManager;
    private string? _id;
    public TenantContextAccessor(
        IUserContextAccessor userContextAccessor,
        UserManager<EmailTamerUser> userManager){
        _userContextAccessor = userContextAccessor;
        _userManager = userManager;
    }
    public TenantContextAccessor(Guid tenantId){
        _id = tenantId.ToString();
    }
    public async Task<string> GetTenantId(){
        if (string.IsNullOrEmpty(_id)){
            var userId = _userContextAccessor.Id;
            var user = await _userManager.FindByIdAsync(userId);
            _id = user!.TenantId.ToString();
        }
```

```
        return _id;
    }
    public string GetDatabaseName(){
        var tenantId = GetTenantId().GetAwaiter().GetResult();
        var dbName = $"tenant_{tenantId.Replace("-", "_")}";
        return dbName;
    }
    public string GetS3BucketName(){
        var tenantId = GetTenantId().GetAwaiter().GetResult();
        var bucketName = $"tenant-{tenantId}";
        return bucketName;
    }
}
```

3.1.2 Фабрика контексту бази даних тенанта

Для створення контексту бази даних для конкретного тенанта реалізовано фабрику TenantDbContextFactory, яка динамічно створює з'єднання до відповідної бази даних, що наведено в лістингу 3.2.

Лістинг 3.2 – TenantDbContextFactory

```
public class TenantDbContextFactory(
    DbContextOptionsBuilder<TenantDbContext> options,
    IServiceProvider serviceProvider)
    : IDbContextFactory<TenantDbContext>{
    public TenantDbContext CreateDbContext(){
        var connectionString = GetConnectionStringForTenant();
        return CreateDbContextInternal(connectionString);
    }
    public TenantDbContext CreateDbContext(ITenantContextAccessor
tenant, IEncryptionService encryptionService){
        var connectionString =
GetConnectionString(tenant.GetDatabaseName());
        return CreateDbContextInternal(connectionString,
encryptionService);
    }
    private TenantDbContext CreateDbContextInternal(string
connectionString, IEncryptionService? encryptionService = null){
        var dbConfig =
serviceProvider.GetRequiredService<IOptionsMonitor<TenantsDatabaseCo
nfig>>().CurrentValue;
        options.UseMySQL(connectionString,
            builder =>
            {
                builder
                    .EnableRetryOnFailure(dbConfig.Retries)
                    .CommandTimeout(dbConfig.Timeout);
            });
```

```
        var configurator =
serviceProvider.GetRequiredService<IDatabaseConfigurator>();
        var encryptService = encryptionService ??
serviceProvider.GetRequiredService<IEncryptionService>();

        var dbContext = new TenantDbContext(options.Options,
configurator, encryptService);
        dbContext.Database.Migrate();
        return dbContext;
    }
    private string GetConnectionString(string dbName){
        var dbConfig =
serviceProvider.GetRequiredService<IOptionsMonitor<TenantsDatabaseCo
nfig>>().CurrentValue;
        return string.Format(dbConfig.DefaultConnectionString,
dbName);
    }

    private string GetConnectionStringForTenant(){
        var accessor =
serviceProvider.GetRequiredService<ITenantContextAccessor>();
        var dbName = accessor.GetDatabaseName();
        return GetConnectionString(dbName);
    }
}
```

3.1.3 Фабрика репозиторію тенанта

Для роботи з файлами в Amazon S3 реалізовано фабрику TenantRepositoryFactory, яка створює репозиторій для роботи з конкретним бакетом S3. Ця фабрика перевіряє існування бакета S3 для тенанта і створює його за потреби. Для реєстрації сервісу в IoC-контейнері також визначено інтерфейс ITenantRepositoryFactory, що наведено в лістингу 3.3.

Лістинг 3.3 – Інтерфейс ITenantRepositoryFactory та реалізація TenantRepositoryFactory

```
internal interface ITenantRepositoryFactory{
    Task<ITenantRepository> Create(CancellationToken
cancellationToken);
    Task<ITenantRepository> Create(ITenantContextAccessor
tenantAccessor, CancellationToken cancellationToken);
}
internal class TenantRepositoryFactory(
    IBlobStorage blobStorage,
    ILogger<ITenantRepository> logger,
    ITenantContextAccessor accessor,
```

```
    IAmazonS3 client
    ) : ITenantRepositoryFactory{
    public async Task<ITenantRepository> Create(CancellationToken
cancellationToken){
        var bucketName = accessor.GetS3BucketName();
        if (!await AmazonS3Util.DoesS3BucketExistV2Async(client,
bucketName)){
            await client.PutBucketAsync(bucketName,
cancellationToken);
            logger.LogInformation("Initialized {BucketName} bucket",
bucketName);
        }
        else{
            logger.LogInformation("Bucket {BucketName} already
exists", bucketName);
        }

        return new TenantRepository(blobStorage, bucketName);
    }
    public async Task<ITenantRepository>
Create(ITenantContextAccessor tenantAccessor, CancellationToken
cancellationToken){
        var bucketName = tenantAccessor.GetS3BucketName();
        if (!await AmazonS3Util.DoesS3BucketExistV2Async(client,
bucketName)){
            await client.PutBucketAsync(bucketName,
cancellationToken);
            logger.LogInformation("Initialized {BucketName} bucket",
bucketName);
        }
        else{
            logger.LogInformation("Bucket {BucketName} already
exists", bucketName);
        }
        return new TenantRepository(blobStorage, bucketName);
    }
}
```

3.1.4 Шифрування даних тенанта

Для забезпечення безпеки чутливих даних реалізовано сервіс шифрування EncryptionService, який використовує унікальний ключ для кожного тенанта, даний сервіс використовується у EncryptionConvertor, щоб виконувати шифрування/розшифрування на рівні Entity Framework Core (без явного використання). Для реєстрації сервісу в IoC-контейнері також визначено інтерфейс IEncryptionService, що наведено в лістингу 3.4.

Лістинг 3.4 – EncryptionConvertor, IEncryptionService та EncryptionService (стисло)

```
public class EncryptionConvertor(IEncryptionService
encryptionService, ConverterMappingHints mappingHints = null)
    : ValueConverter<string, string>(
        x => encryptionService.Encrypt(x),
        x => encryptionService.Decrypt(x),
        mappingHints);

public interface IEncryptionService{
    string Encrypt(string value);
    string Decrypt(string value);
}

public sealed class EncryptionService(
    ITenantContextAccessor tenantContextAccessor,
    IConfiguration configuration,
    ILogger<EncryptionService> logger)
    : IEncryptionService
{
    private const int AesKeySizeBytes = 32;
    private const int AesIvSizeBytes = 16;

    private byte[] DeriveKeyFromTenantId(){
        try{
            var tenantId =
tenantContextAccessor.GetTenantId().GetAwaiter().GetResult();

            var salt =
configuration.GetValue<string>("Encryption:Tenant:Salt")
                ?? throw new InvalidOperationException("Tenant
encryption salt not configured.");

            if (string.IsNullOrEmpty(salt)){
                logger.LogError("Encryption salt is empty or not
configured.");
                throw new InvalidOperationException("Encryption salt
cannot be empty.");
            }
            using var pbkdf2 = new Rfc2898DeriveBytes(
                Encoding.UTF8.GetBytes(tenantId),
                Encoding.UTF8.GetBytes(salt),
                100000,
                HashAlgorithmName.SHA256);
            var key = pbkdf2.GetBytes(AesKeySizeBytes);
            return key;
        }
        catch (Exception ex){
            logger.LogError(ex, "Failed to derive encryption key for
tenant.");
            throw;
        }
```

```
    }
    public string Encrypt(string dataToEncrypt){
        \\ логіка перевірки валідності даних для зашифровки
        try{
            byte[] encrypted;
            byte[] iv;
            using (var aesAlg = Aes.Create())
            {
                aesAlg.Key = DeriveKeyFromTenantId();

                \\ логіка шифрування
            }
            var result = Convert.ToBase64String(encrypted);
            return result;
        }
        catch (Exception ex){
            \\ обробка помилки
        }
    }

    public string Decrypt(string dataToDecrypt){
        try{
            \\ логіка перевірки валідності шифротексту
            using var aes = Aes.Create();
            \\ логіка шифрування
            return result;
        }
        catch (Exception ex){
            \\ обробка помилки
        }
    }
}
```

Для позначення полів сутностей, які мають бути зашифровані, використовується атрибут EncryptProperty.

3.2 Реалізація синхронізації з поштовими серверами

3.2.1 MailKitImapConnector

Для підключення до IMAP-серверів реалізовано статичний клас MailKitImapConnector, який використовує бібліотеку MailKit, що наведено в лістингу 3.5.

Лістинг 3.5 – MailKitImapConnector

```
public static class MailKitImapConnector{
    public static async Task<IImapClient>
ConnectToImapClient(EmailBox emailBox, CancellationToken
cancellationToken)
    {
        var client = new ImapClient();
        try
        {
            await client.ConnectAsync(
                emailBox.EmailDomainConnectionHost,
                emailBox.EmailDomainConnectionPort,
                emailBox.UseSSl,
                cancellationToken);
        }
        catch (Exception e)
        {
            if (e is System.Net.Sockets.SocketException se)
                throw new
MailKitImapConnectorException(ConnectionFault.ConnectionRefused,
se);
            if (e is System.ArgumentOutOfRangeException auore)
                throw new
MailKitImapConnectorException(ConnectionFault.PortOutOfRange,
auore);
            throw new
MailKitImapConnectorException(ConnectionFault.Other, e);
        }
        try
        {
            await client.AuthenticateAsync(
                emailBox.AuthenticateByEmail ? emailBox.Email :
emailBox.UserName,
                emailBox.Password,
                cancellationToken);
        }
        catch (Exception e)
        {
            if (e is MailKit.Security.AuthenticationException ae)
                throw new
MailKitImapConnectorException(ConnectionFault.WrongAuthenticationCre
dentials, ae);
            throw new
MailKitImapConnectorException(ConnectionFault.Other, e);
        }
        return client;
    }
}
```

3.2.2 BackupService

Основна логіка резервного копіювання реалізована в класі BackupService,

який відповідає за отримання повідомлень з поштових скриньок та їх збереження. Для реєстрації сервісу в IoC-контейнері також визначено інтерфейс IBackupService – лістинги А.1 та А.2 додатку А, діаграму послідовності резервного копіювання однієї скриньки наведено на рисунку Б.2 додатку Б.

Процес резервного копіювання включає:

- завантаження початкових даних (поштова скринька, існуючі повідомлення та папки);

- оновлення статусу поштової скриньки;

- підключення до IMAP-сервера;

- отримання списку папок;

- обробка кожної папки та повідомлень у ній: для кожного повідомлення виконується перевірка на існування в базі даних. Якщо повідомлення нове, воно зберігається разом із HTML-вмістом та вкладеннями за наявності;

- після завершення синхронізації оновлюється статус поштової скриньки та час останньої синхронізації.

3.2.3 PeriodicBackupService

Для автоматичного резервного копіювання реалізовано фоновий сервіс PeriodicBackupService, який запускається за розкладом. Сервіс отримує усі тенанти з основної БД та запускає резервне копіювання для кожного, що наведено в лістингу А.3 додатку А, діаграму послідовності наведено на рисунку Б.1 додатку Б.

3.3 Реалізація API та контролерів

Контролери в EmailTamer є мінімалістичними і відповідають лише за маршрутизацію запитів до відповідних обробників команд та запитів через MediatR. Також контролери мають набір атрибутів для формування OpenAPI специфікації, приклад сформованої специфікації наведено на рисунку 3.1. Приклад наведено в лістингу 3.6.

Лістинг 3.6 – Фрагмент з EmailBoxesController

```
[Route("api/emailBoxes")]
public class EmailBoxesController(IMediator mediator) : Controller {
    [HttpPost(Name = nameof(CreateEmailBox))]
    [Authorize(Policy = AuthPolicy.User)]
    [ProducesResponseType(200)]
    [ProducesResponseType(400)]
    public Task<IActionResult> CreateEmailBox([FromBody] EmailBoxDto emailBox, CancellationToken ct = default) =>
        mediator.Send(new CreateEmailBox(emailBox), ct);

    // Інші методи...
}
```

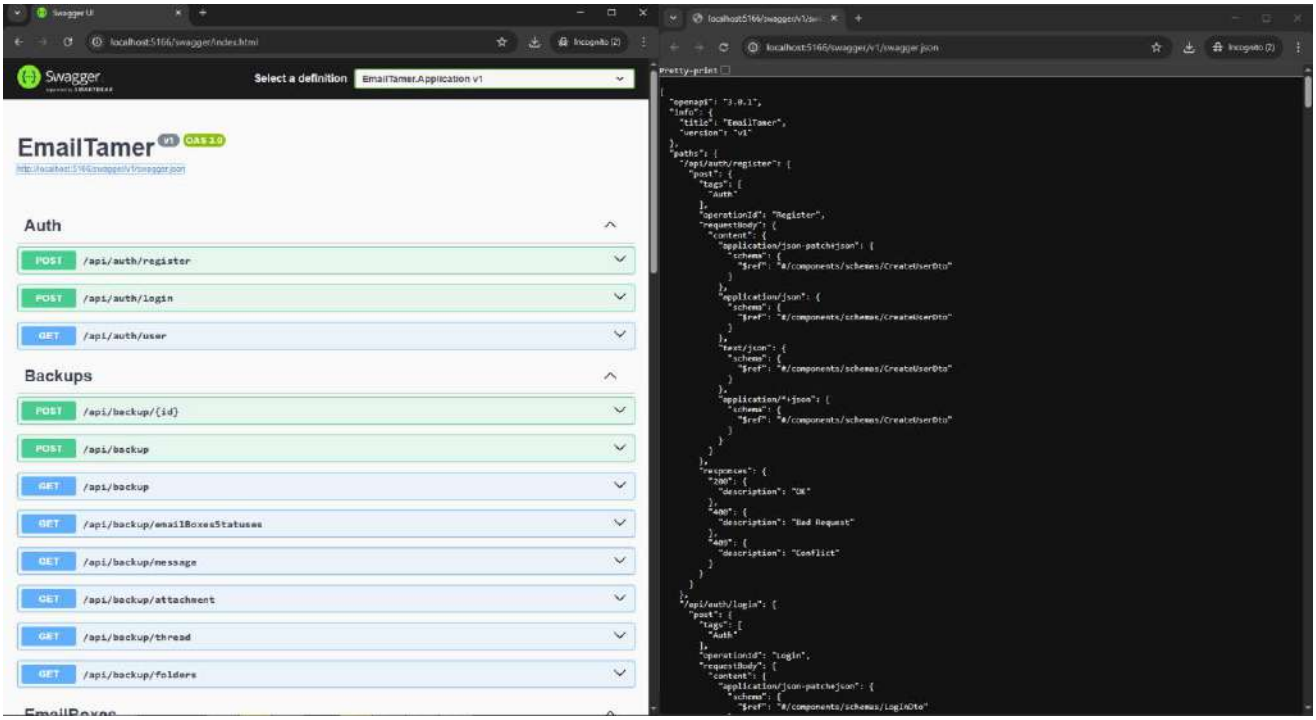


Рисунок 3.1 – Сформована OpenAPI специфікація

3.4 JWT-аутентифікація

3.4.1 Конфігурування та випуск токену

Конфігурування JWT-аутентифікації на серверній частині відбувається в модулі Parts.Auth через розширення сервісів ASP.NET Core. Основні налаштування

JWT визначаються в файлі конфігурації appsettings.json, де зберігаються параметри для видачі та валідації токенів – приклад конфігурації в лістингу 3.7.

Лістинг 3.7 – Приклад конфігурації JWT

```
"JWT": {
    "Issuer": "emailtamer-backend",
    "Authority": "https://localhost/",
    "Audience": "emailtamer-frontend",
    "Key": "mysupersecret_secretsecretsecretkey!123"
  }
```

Ці параметри завантажуються при старті застосунку та використовуються для налаштування JWT-аутентифікації. Реєстрацію JWT-аутентифікації наведено в лістингу 3.8.

Лістинг 3.8 – Реєстрація JWT

```
services.AddAuthentication(options =>{
            options.DefaultAuthenticateScheme =
                options.DefaultChallengeScheme =
                    options.DefaultForbidScheme =
                        options.DefaultScheme =
                            options.DefaultSignInScheme =
                                options.DefaultSignOutScheme =
JwtBearerDefaults.AuthenticationScheme;
        }).AddJwtBearer(options =>{
            options.TokenValidationParameters = new
TokenValidationParameters
            {
                ValidateIssuer = true,
                ValidIssuer = jwtConfig.Issuer,
                ValidateAudience = true,
                ValidAudience = jwtConfig.Audience,
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = new SymmetricSecurityKey(

System.Text.Encoding.UTF8.GetBytes(jwtConfig.Key)
                )
            };
        });
```

Випуск JWT-токену відбувається при успішній автентифікації користувача через ендпоінт /api/auth/login. Токен підписується за допомогою симетричного ключа, визначеного в конфігурації, і має обмежений термін дії. Метод випуску

токену наведено в лістингу 3.10.

Лістинг 3.10 – Метод IssueToken

```
private async Task<string> IssueToken(EmailTamerUser user)
{
    var jwtTokenHandler = new JwtSecurityTokenHandler();
    var config = jwtConfig.CurrentValue;

    var key = Encoding.UTF8.GetBytes(config.Key);
    var secretKey = new SymmetricSecurityKey(key);

    var claims = new List<Claim>
    {
        new ("id", user.Id ),
        new (ClaimTypes.Email, user.Email!)
    };
    var roles = await userManager.GetRolesAsync(user);

    claims.AddRange(roles.Select(role => new Claim(ClaimTypes.Role,
role)));

    var tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new(claims),
        Issuer = config.Issuer,
        Audience = config.Audience,
        Expires = systemClock.UtcNow.AddHours(24).DateTime,
        SigningCredentials = new(secretKey,
SecurityAlgorithms.HmacSha256)
    };

    var token = jwtTokenHandler.CreateToken(tokenDescriptor);
    var jwtToken = jwtTokenHandler.WriteToken(token);
    return jwtToken;
}
```

3.4.2 Автентифікація на клієнтській частині

На клієнтській частині JWT-аутентифікація реалізована з використанням бібліотеки Zustand для управління станом. Основний стан аутентифікації зберігається в сховищі AuthStore, що наведено в лістингу 3.11.

Лістинг 3.11 – AuthStore

```
const userSchema = z.object({
    id: z.string(),
    email: z.string(),
```

```
    role: userRolesSchema,
});
type User = z.infer<typeof userSchema>;
type AuthState = {
    user: User | null;
    token: string | null;
    isAuthenticated: boolean;
    setUser: (user: UserDto | null) => void;
    setToken: (token: string | null) => void;
    logout: () => void;
};

const initialToken = localStorage.getItem('token') || null;
const isInitialTokenValid = initialToken &&
!isTokenExpired(initialToken);

const useAuthStore = create<AuthState>()(
    devtools(
        (set) => ({
            user: null,
            token: isInitialTokenValid ? initialToken : null,
            isAuthenticated: !!isInitialTokenValid,
            setUser: (userDto: UserDto | null) => {
                const user = userDto ? userSchema.parse(userDto) :
null;
                set((state) => ({
                    user,
                    isAuthenticated: !!user && !!state.token &&
!isTokenExpired(state.token),
                }));
            },
            setToken: (token: string | null) => {
                if (token) {
                    localStorage.setItem('token', token);
                } else {
                    localStorage.removeItem('token');
                }
                set((state) => ({
                    token,
                    isAuthenticated: !!state.user && !!token &&
!isTokenExpired(token),
                }));
            },
            logout: () => {
                localStorage.removeItem('token');
                set({user: null, token: null, isAuthenticated:
false});
            },
        }),
        {name: 'auth-store'}
    )
);
```

Це сховище ініціалізується з перевіркою наявності збереженого токену в localStorage та його валідності. Сховище також надає методи для управління аутентифікацією, такі як setToken та logout.

Для захисту маршрутів від неавторизованого доступу використовується компонент GuardedRoute, який перевіряє стан аутентифікації та роль користувача, що наведено в лістингу 3.12.

Лістинг 3.12 – GuardedRoute

```
interface GuardedRouteProps {
    page: ReactElement;
    roles?: UserRole[];
}

export const GuardedRoute: FC<GuardedRouteProps> = (props) => {
    const {page, roles} = props;
    const location = useLocation();

    const {isAuthenticated, user} = useAuthStore((state) => ({
        isAuthenticated: state.isAuthenticated,
        user: state.user,
    }));

    const isUserAuthorized = !_.isEmpty(roles) && user ?
roles?.includes(user.role) : true;

    const isAccessAllowed = isAuthenticated && isUserAuthorized;

    if (isAccessAllowed) return <>{page}</>;
    if (!isAuthenticated) return <Navigate
        to={`${LOGIN_ROUTE}?${REDIRECT_TO_PARAM}=${location.pathname
+ location.search}`}
        />;
    if (isAuthenticated && !isUserAuthorized)
        return <Navigate to={DEMO_ROUTE} replace/>;
};
```

3.5 Конфігурація маршрутів

Маршрути визначені в файлі router.tsx з використанням декларативного підходу, що наведено в лістингу 3.13.

Лістинг 3.13 – Оголошення маршрутів

```
export const router = createBrowserRouter(
    createRoutesFromElements(
        <Route element={<Layout/>}>
            <Route path={HOME_ROUTE} element={
                <GuardedRoute page={
                    <TranslationScopeProvider scope='homePage'>
                        <HomePage/>
                    </TranslationScopeProvider>}
roles={allowedRoles}/>}/>
            <Route path={DEMO_ROUTE} element={
                <TranslationScopeProvider scope='demoPage'>
                    <DemoPage/>
                </TranslationScopeProvider>}/>
            <Route path={LOGIN_ROUTE} element={
                <TranslationScopeProvider scope='loginPage'>
                    <LoginPage/>
                </TranslationScopeProvider>}/>
            <Route path={REGISTER_ROUTE} element={
                <TranslationScopeProvider scope='registerPage'>
                    <RegisterPage/>
                </TranslationScopeProvider>}/>
            <Route path={threadRoute.template} element={
                <GuardedRoute page={
                    <TranslationScopeProvider scope='threadPage'>
                        <ThreadPage/>
                    </TranslationScopeProvider>}
roles={allowedRoles}/>}/>
            <Route path={NOT_FOUND_ROUTE} element={
                <TranslationScopeProvider scope='notFoundPage'>
                    <NotFoundPage/>
                </TranslationScopeProvider>}/>
            <Route path='*' element={<Navigate to={DEMO_ROUTE}
replace/>}/>
        </Route>));
```

Для навігації між сторінками використовується хук useNavigate та компонент Link, що наведено в лістингу 3.14.

Лістинг 3.14 – Приклади використання хуку useNavigate та компоненту Link

```
\\ використання компоненту Link
const Header = () => {
    const {t} = useScopedContextTranslator();
    const {isAuthenticated} = useAuthStore((state) => ({
        isAuthenticated: state.isAuthenticated,
    }));

    return (
        <AppBar position='sticky'>
            <Toolbar
                sx={{
                    height: HEADER_HEIGHT,
                    display: 'flex',
                    alignItems: 'center',
                }}
            >
                <Stack
                    direction='row'
                    alignItems='center'
                    component={Link}
                    to={isAuthenticated ? HOME_ROUTE : DEMO_ROUTE}
                    \\ решта властивостей
                >
                    <Logo sx={{width: {xs: 30, sm: 35}, height: {xs:
30, sm: 35}, mr: 1}}/>
                    \\ решта коду
                </Stack>
                \\ решта коду
            </Toolbar>
        </AppBar>
    );
};


\\ використання хуку useNavigate:
const DemoPage = () => {
    const {t} = useScopedContextTranslator();
    const navigate = useNavigate();

    return (
        <Container maxWidth='lg' sx={{py: 6}}>
            \\ решта коду
            <Box textAlign='center'>
                <Button
                    variant='contained'
                    color='primary'
```

```
                    size='large'
                    onClick={() => navigate(REGISTER_ROUTE)}
                >
                    {t('ctaButton')}
                </Button>
            </Box>
        </Container>
    );
};
```

3.6 Інтернаціоналізація

EmailTamer підтримує багатомовність за допомогою бібліотеки i18next. Конфігурація i18next наведена в лістингу 3.15.

Лістинг 3.15 – i18n.ts

```
export const defaultNS = 'translation';
export const resources = {
    en: {translation: en},
    ua: {translation: ua},
} as const;

const savedPreferences = localStorage.getItem('preferences');
const initialLanguage = savedPreferences
    ? JSON.parse(savedPreferences).language
    : 'en';

i18n.use(initReactI18next).init({
    debug: true,
    fallbackLng: initialLanguage,
    resources: resources,
    defaultNS,
    interpolation: {escapeValue: false},
});
```

Переклади зберігаються в JSON-файлах для кожної мови -лістинг 3.16.

Лістинг 3.16 – Уривок файлу з перекладами українською мовою

```
{
  "header": {
    "title": "EmailTamer",
```

```
    "login": "Увійти",
    "register": "Зареєструватися",
    "logout": "Вийти",
    "searchBarPlaceholder": "Пошук..."
  },
  // Інші переклади...
}
```

У EmailTamer реалізовано систему локалізації, яка використовує контекстний підхід для організації перекладів. Ця система складається з двох основних компонентів (лістинг 3.17):

- TranslationScopeProvider - компонент-провайдер, який створює контекст для перекладів. Цей React-компонент використовує React Context API для передачі інформації про поточну область перекладів (scope) вниз по дереву компонентів;

- useScopedContextTranslator - хук, який використовується для отримання функції перекладу в поточному контексті.

Лістинг 3.17 – TranslationScopeProvider та useScopedContextTranslator

```
interface TranslationScopeContextProps {
    scope: string;
}

const TranslationScopeContext =
createContext<TranslationScopeContextProps>({
    scope: '',
});

export function TranslationScopeProvider({
    scope,
    children,
    rewriteScope = false
}: {
    scope: string;
    children: ReactNode;
    rewriteScope?: boolean;
}) {
    const parentScope = useContext(TranslationScopeContext).scope;
    const fullScope = parentScope ? joinTranslatorKey(parentScope,
scope) : scope;

    return (
        <TranslationScopeContext.Provider value={{ scope:
rewriteScope ? scope : fullScope }}>
            {children}
        </TranslationScopeContext.Provider>
```

```
    );
}

export function useTranslationScopeContext() {
    return useContext(TranslationScopeContext);
}
```

TranslationScopeProvider використовується для обгортання інших компонентів і встановлення області перекладів, що наведено в лістингу 3.18.

Лістинг 3.18 – Приклад використання TranslationScopeProvider

```
<TranslationScopeProvider scope='foldersSection'>
    <FoldersSection
        selectedFolderId={selectedFolderId}
        setSelectedFolderId={handleSetSelectedFolderId}
    />
</TranslationScopeProvider>
```

Хук useScopedContextTranslator використовується в компонентах для отримання функції перекладу, що наведено в лістингу 3.19.

Лістинг 3.19 – Приклад використання TranslationScopeProvider

```
const FoldersSection = ({selectedFolderId, setSelectedFolderId}:
FoldersSectionProps) => {
    const {data: folders, isLoading} = useGetFolders({});
    const {t} = useScopedContextTranslator();

    const items = [
        {
            id: 'allFoldersIndex',
            label: t('all'),
            onClick: () => setSelectedFolderId(null),
            selected: selectedFolderId === null,
        },
        // решта коду
    ];
    // решта коду
}
```

Важливою особливістю цієї системи є можливість вкладення областей перекладів.

Для зміни мови інтерфейсу застосунку реалізовано компонент

LanguageSelector, що наведено в лістингу 3.20.

Лістинг 3.20 – Приклад використання LanguageSelector

```
const LanguageSelector = ({onLanguageChange}: LanguageSelectorProps)
=> {
    const [langMenuAnchor, setLangMenuAnchor] = useState<null |
HTMLElement>(null);
    const open = Boolean(langMenuAnchor);

    const {setLanguage} = usePreferencesStore();

    // Обробник зміни мови
    const handleLanguageChange = (lang: string) => {
        i18n.changeLanguage(lang);
        setLanguage(lang);
        handleLangMenuClose();
        if (onLanguageChange) onLanguageChange();
    };
    // Решта коду...
}
```

3.7 Адаптивний дизайн

EmailTamer використовує MUI для створення адаптивного інтерфейсу, який коректно відображається на різних пристроях. Для адаптації інтерфейсу до різних розмірів екрану використовуються медіа-запити через систему точок зупинки MUI, реалізацію наведено в лістингу 3.21.

Лістинг 3.21 – Приклад організації адаптивності

```
<Button
    variant='contained'
    size='small'
    startIcon={<CloudDownload/>}
    onClick={handleBackupButtonClick}
    disabled={isBackuping}
    sx={{
        mr: 1,
        display: {xs: 'none', sm: 'flex'}  // Приховати на малих
екранах
    }}
```

```
>
    {t('backup')}
</Button>

<Fab
    size='small'
    color='primary'
    onClick={handleBackupButtonClick}
    disabled={isBackuping}
    sx={{
        mr: 1,
        display: {xs: 'flex', sm: 'none'}  // Показати тільки на
малих екранах
    }}
>
    <CloudDownload/>
</Fab>
```

На мобільних пристроях використовуються висувні панелі (drawers) для доступу до функціональності, яка на десктопі відображається постійно, що наведено в лістингу 3.22.

Лістинг 3.22 – Приклад використання висувної панелі

```
<SwipeableDrawer
    anchor='left'
    open={foldersDrawerOpen}
    onClose={toggleFoldersDrawer}
    onOpen={toggleFoldersDrawer}
    sx={{
        '& .MuiDrawer-paper': {
width: {xs: 250, sm: 350}, boxSizing: 'border-box', top:
HEADER_HEIGHT, height: `calc(100% - ${HEADER_HEIGHT}px)`},display:
{sm: 'block', md: 'none'}  // Показати тільки на малих екранах
    }}
    variant='temporary'
>
    <Box sx={{p: 2}}>
        <TranslationScopeProvider scope='foldersSection'>
            <FoldersSection
                selectedFolderId={selectedFolderId}
                setSelectedFolderId={(id) => {
                    handleSetSelectedFolderId(id);
                    toggleFoldersDrawer();
                }}
            />
        </TranslationScopeProvider>
    </Box>
</SwipeableDrawer>
```

3.8 Висновки до розділу

У процесі реалізації EmailTamer було створено повноцінний вебзастосунок для резервного копіювання електронних листів, який відповідає всім поставленим вимогам. Серверна частина, побудована з використанням CQRS та Mediator, забезпечує надійну обробку запитів, взаємодію з IMAP-серверами та зберігання даних у MySQL і Amazon S3. Мультитенантна архітектура з окремими БД та бакетами для кожного користувача гарантує ізоляцію даних та підвищує безпеку.

Клієнтська частина, реалізована на React і TypeScript з використанням MUI, забезпечує зручний інтерфейс для керування поштовими скриньками, перегляду листів та виконання резервного копіювання. Використання TypeScript замість JavaScript дозволило забезпечити типобезпеку та зменшити кількість помилок під час розробки. Компонентна архітектура React сприяє модульності та повторному використанню коду, а React Query оптимізує взаємодію з API.

Реалізація JWT-аутентифікації забезпечує безпечний доступ до системи та захист даних користувачів. Шифрування чутливих даних, таких як паролі від поштових скриньок, підвищує загальний рівень безпеки системи. Використання патерну CQRS дозволило розділити операції читання та запису, що спрощує масштабування та оптимізацію кожного типу операцій окремо.

Інтернаціоналізація інтерфейсу з використанням i18next та контекстного підходу до перекладів забезпечує підтримку різних мов, що розширює аудиторію потенційних користувачів. Адаптивний дизайн, реалізований за допомогою можливостей MUI, забезпечує коректне відображення інтерфейсу на різних пристроях, від мобільних телефонів до десктопних комп'ютерів.

Отже, реалізація EmailTamer демонструє ефективне застосування сучасних технологій та архітектурних підходів для створення надійного, безпечного та зручного вебзастосунку для резервного копіювання електронних листів. Система відповідає всім функціональним та нефункціональним вимогам, забезпечуючи надійний захист даних користувачів та зручний доступ до архівованих листів.

4 РЕЗУЛЬТАТИ РОБОТИ

4.1 Огляд реалізованого сервісу

У цьому розділі представлено результати розробки сервісу EmailTamer - системи для резервного копіювання електронних листів. Розглянемо основні функціональні можливості системи через інтерфейс користувача та продемонструємо, як реалізовані ключові вимоги.

4.1.1 Демо-сторінка

Демо-сторінка має на меті окреслити цільову аудиторію та продемонструвати функціонал, що надає застосунок. Вигляд сторінки наведено на рисунку 4.1

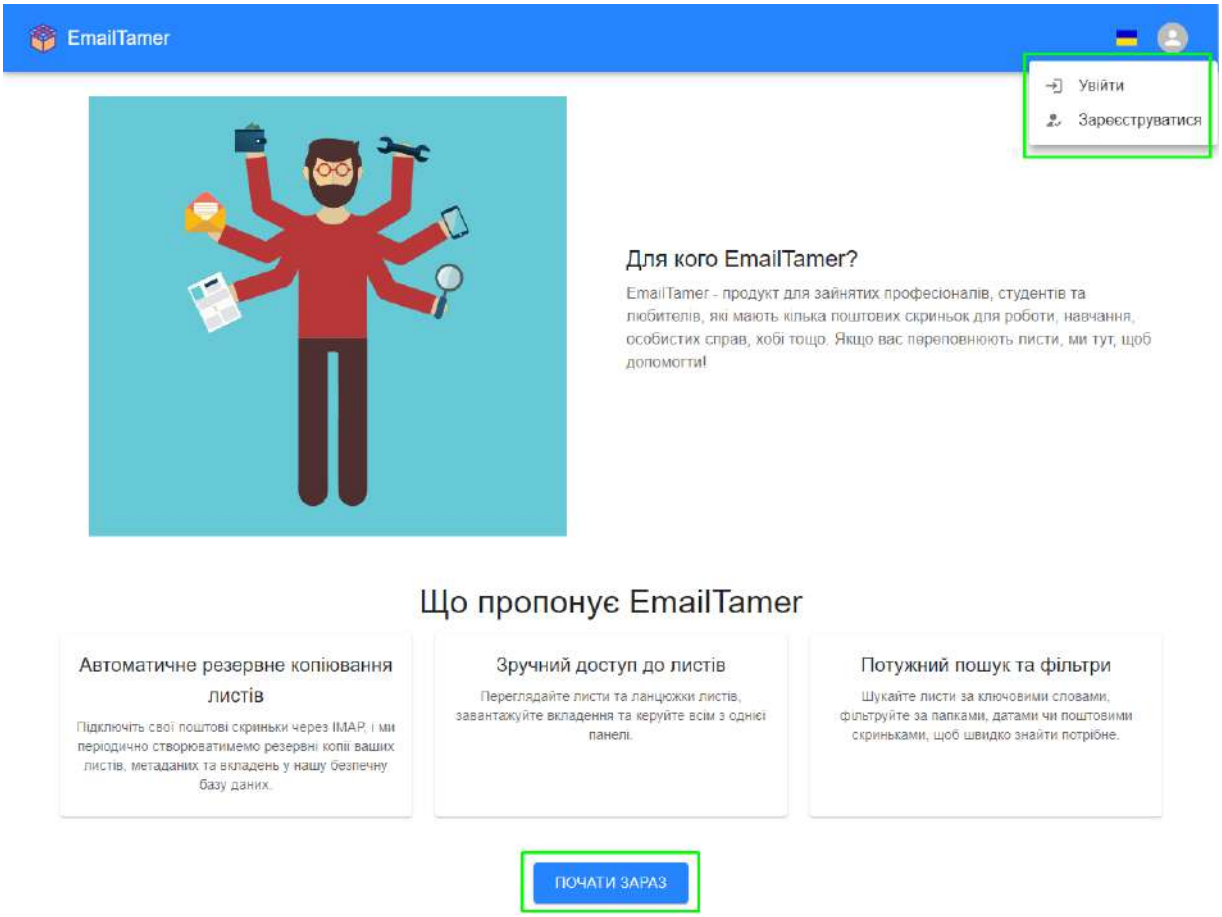


Рисунок 4.1 – Демо-сторінка

Зі сторінки користувач має змогу перейти до реєстрації, скориставшись кнопкою «Почати» або пунктом контекстного меню «Зареєструватися», чи увійти

до вже існуючого облікового запису за допомогою пункту «Увійти» в контекстному меню.

4.1.2 Сторінки реєстрації та входу

Сторінки реєстрації та входу надають зручний і інтуїтивний інтерфейс для створення та входу в обліковий запис. Вигляд сторінок наведено на рисунках 4.2 та 4.3 відповідно.

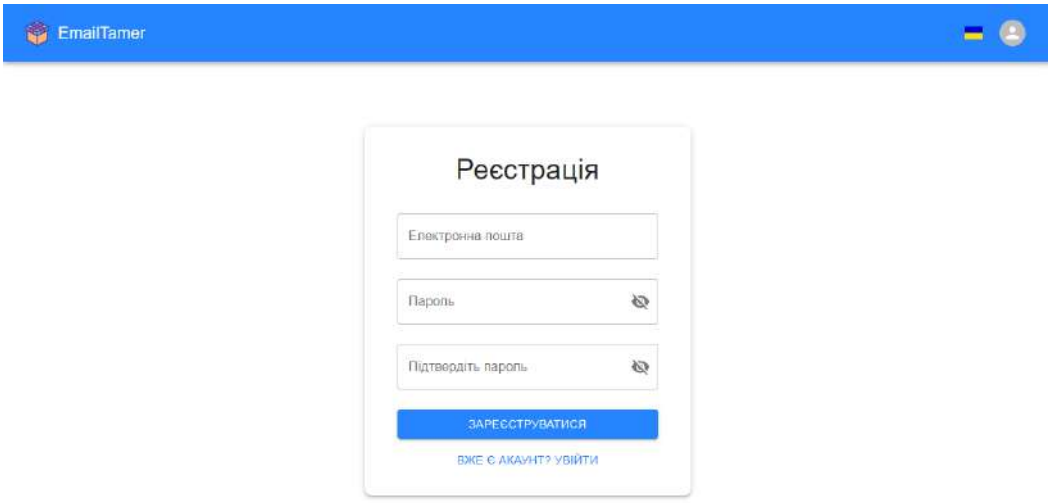


Рисунок 4.2 – Сторінка реєстрації

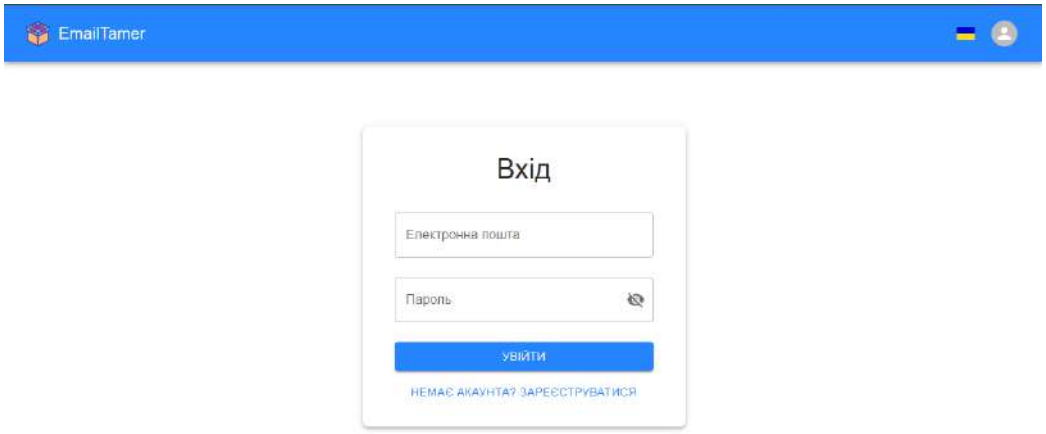


Рисунок 4.3 – Сторінка входу в обліковий запис

Форми мають валідацію, що перевіряє формат електронної пошти, довжину і склад паролю, форма реєстрації також перевіряє співпадіння в обох полях, на рисунку 4.4 наведено підбірку невдалих спроб відправки форми.

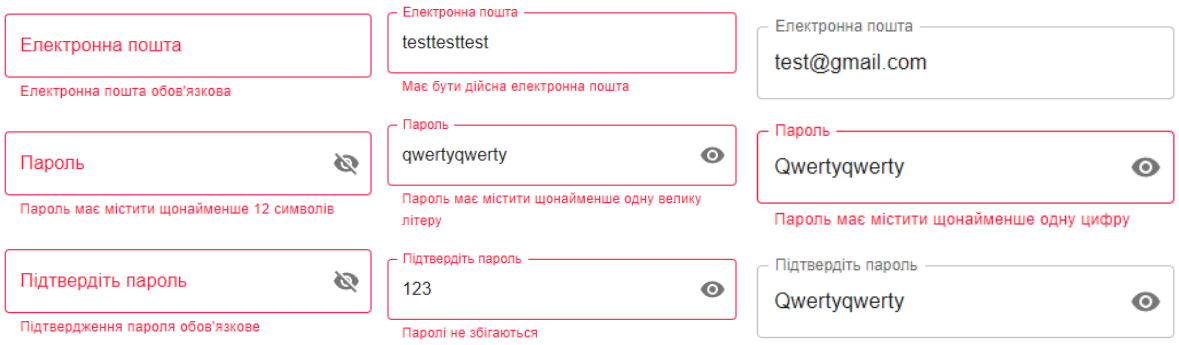


Рисунок 4.4 – Підбірка невдалих спроб відправки форми

За успішної відправки форми чи помилці не пов’язаній із валідацією даних виводиться відповідне повідомлення. Приклади повідомлень для користувача наведено на рисунках 4.5

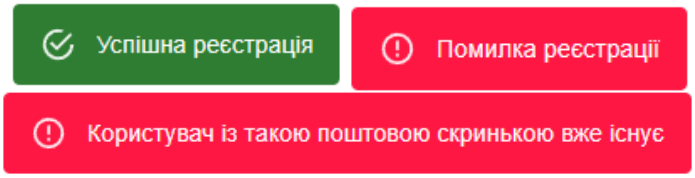


Рисунок 4.5 – Підбірка повідомлень

4.1.3 Хедер

Хедер відповідає за відображення верхньої панелі навігації, яка присутня на всіх сторінках застосунку – вигляд на різних екранах наведено на рисунку 4.6.

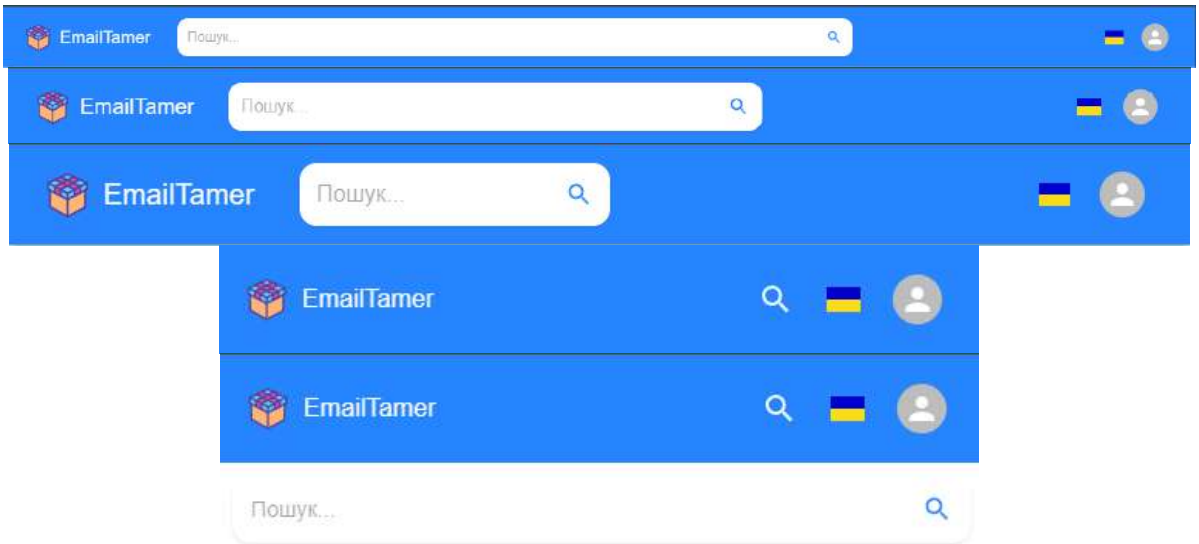


Рисунок 4.6 – Адаптивність хедеру

Хедер складається з наступних елементів:

- логотип та назва застосунку - розташовані в лівій частині хедеру. Логотип та назва застосунку є посиланням, яке веде на головну сторінку для авторизованих користувачів або на демо-сторінку для неавторизованих;

- пошукова стрічка - відображається в центральній частині хедеру для авторизованих користувачів. Пошукова стрічка реалізована як окремий компонент SearchBar, який дозволяє користувачам шукати повідомлення;

- елементи керування - розташовані в правій частині хедеру і включають в себе кнопку пошуку для мобільних пристроїв, селектор мови, елементи автентифікації.

4.1.4 Головна сторінка та інтерфейс користувача

Головна сторінка надає користувачу доступ до всіх основних функцій системи. Інтерфейс реалізовано з використанням компонентного підходу React та бібліотеки MUI, що забезпечує інтуїтивно зрозумілий дизайн. Інтерфейс розділений на кілька функціональних блоків (зліва направо на рисунку 4.7):

- секція папок, де відображаються папки згідно поштових клієнтів синхронізованих скриньок;

- секція повідомлень, де відображаються листи згідно фільтрів;

- секція поштових скриньок, де користувач може додавати, редагувати та видаляти підключення до своїх поштових акаунтів.

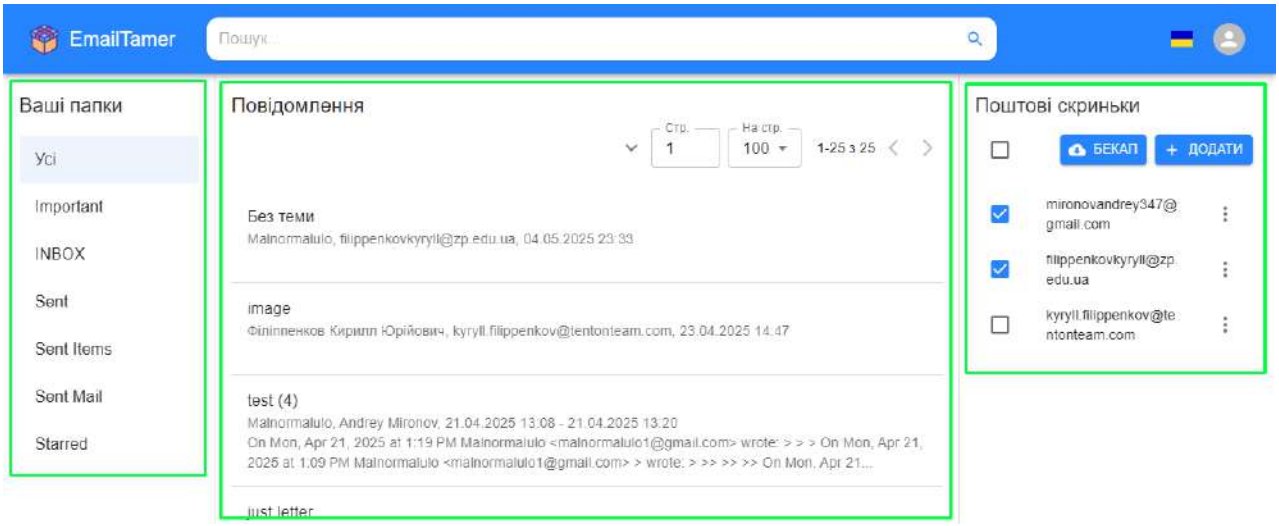


Рисунок 4.7 - Головна сторінка EmailTamer

Вигляд головної сторінки на малому екрані наведено на рисунку 4.8.

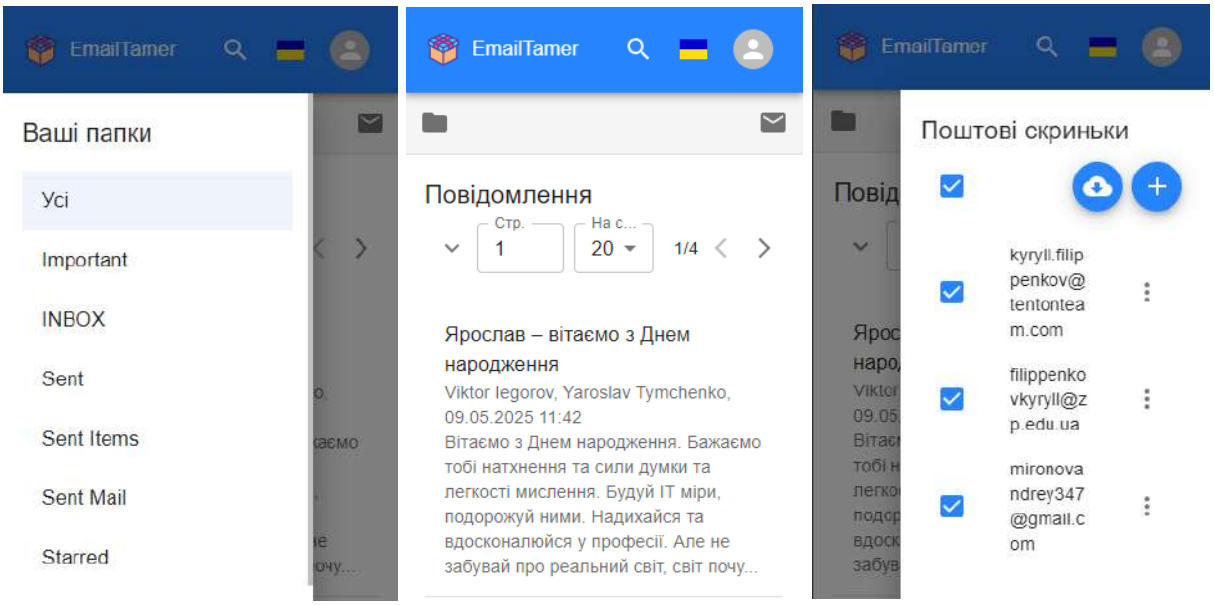


Рисунок 4.8 - Головна сторінка EmailTamer на екрані 320x480

Секція папок є одним із трьох основних компонентів на головній сторінці застосунку EmailTamer. Вона відображає список папок електронної пошти та дозволяє користувачам фільтрувати повідомлення за вибраною папкою. Ідентифікатор обраної папки зберігається в URL для можливості збереження стану між сесіями.

На головній сторінці компонент розміщується в лівій частині інтерфейсу на десктопних пристроях або в бічній панелі на мобільних – рисунки 4.7, 4.8.

Секція повідомлень є центральним компонентом головної сторінки застосунку EmailTamer, який відображає список повідомлень (ланцюжків листів) з вибраних поштових скриньок та папок.

Після успішного підключення та резервного копіювання поштових скриньок, користувач може переглядати всі свої повідомлення в єдиному інтерфейсі. Вигляд секції наведено на рисунку 4.9.

Система надає можливості:

- фільтрації повідомлень за папками та поштовими скриньками;
- пошуку повідомлень за ключовими словами;
- пагінації для зручного перегляду великої кількості повідомлень.

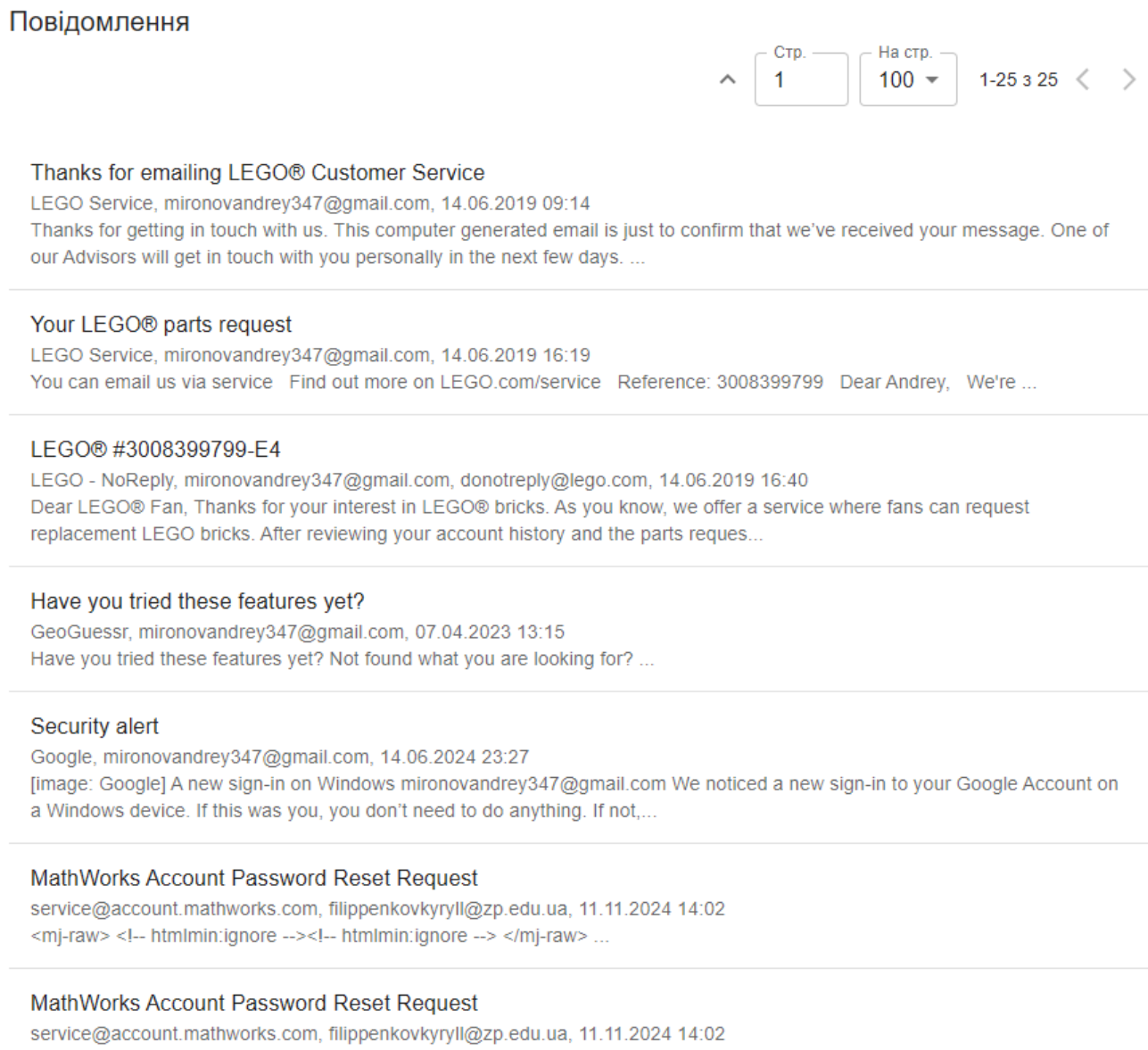


Рисунок 4.9 - Список повідомлень

Запит повідомлень включає наступні параметри:

- folderId - ID вибраної папки (якщо вибрана), поточне значення зберігається в url параметрі;

- emailBoxesIds - ID вибраних поштових скриньок), поточне значення зберігається в url параметрі;

- page - номер сторінки), поточне значення зберігається в url параметрі;

- size - кількість повідомлень на сторінці,), поточне значення зберігається в localStore;

- isByDescending - порядок сортування), поточне значення зберігається в url

параметрі;

- searchTerm - пошуковий запит (якщо є), поточне значення зберігається в url параметрі.

Управління поштовими скриньками є однією з ключових функцій застосунку EmailTamer, яка дозволяє користувачам додавати, редагувати, видаляти та ініціювати резервне копіювання обраних поштових скриньок.

Основним компонентом для управління поштовими скриньками є EmailBoxesSection. На головній сторінці компонент розміщується в правій частині інтерфейсу на десктопних пристроях, в бічній панелі - мобільних, див. рис. 4.7, 4.8.

Список скриньок включає в себе елементи, що містять:

- назву поштової скриньки;
- час останньої синхронізації;
- чекбокс для вибору скриньки;
- статус синхронізації;
- меню для додаткових дій (редагування, видалення).

Кейси використання наочно продемонстровано на рисунку 4.10.

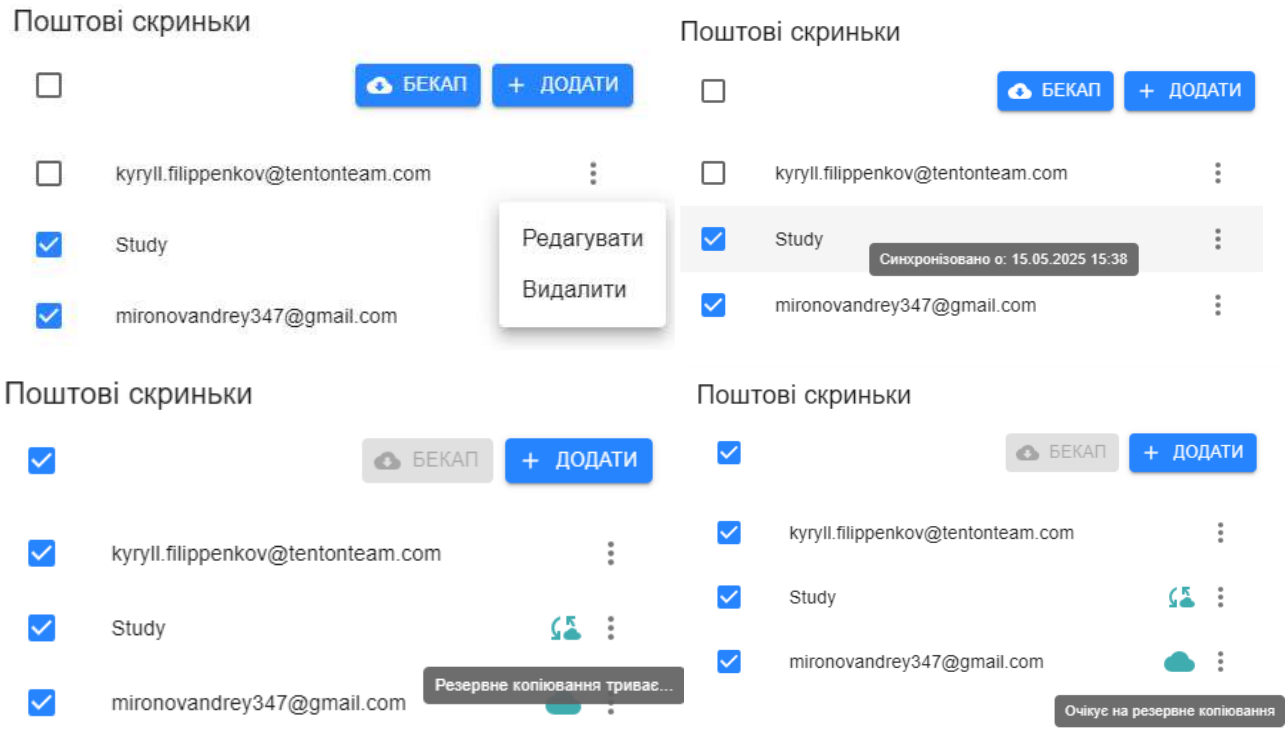


Рисунок 4.10 – Кейси використання секції управління поштовими скриньками

Користувач може вибрати одну або кілька скриньок для резервного копіювання, використовуючи чекбокси, та запустити процес натисканням кнопки "Бекап".

4.1.5 Додавання/редагування/видалення поштової скриньки

Додавання нової поштової скриньки реалізоване через діалогове вікно, яке викликається натисканням кнопки "Додати" в секції поштових скриньок головної сторінки.

Форму наведено на рисунку 4.11.

Інтерфейс додавання поштової скриньки являє собою форму з наступними полями:

- назва скриньки (BoxName) - опціональне поле для зручної ідентифікації скриньки;

- ім'я користувача (UserName) - використовується для автентифікації, якщо не вибрано автентифікацію за email;

- електронна пошта (Email) - обов'язкове поле з валідацією формату email;

- пароль (Password) - обов'язкове поле для автентифікації на поштовому сервері;

- хост підключення (EmailDomainConnectionHost) - адреса IMAP-сервера;

- порт підключення (EmailDomainConnectionPort) - порт IMAP-сервера (за замовчуванням 993 для SSL).

Додаткові налаштування включають:

- перемикач типу автентифікації - вибір між автентифікацією за email або за ім'ям користувача;

- використання SSL/TLS - прапорець для безпечного з'єднання;

- використання стандартних портів IMAP - автоматично встановлює порт 993 для SSL або 143 для незахищеного з'єднання.

Процес додавання:

- користувач заповнює форму з даними поштової скриньки, за потреби вручну ініціює перевірку з'єднання;

- перед збереженням виконується перевірка з'єднання з поштовим сервером;

- у разі успішної перевірки, дані відправляються на сервер для створення нової поштової скриньки;

- користувач отримує повідомлення про успішне додавання або помилку.

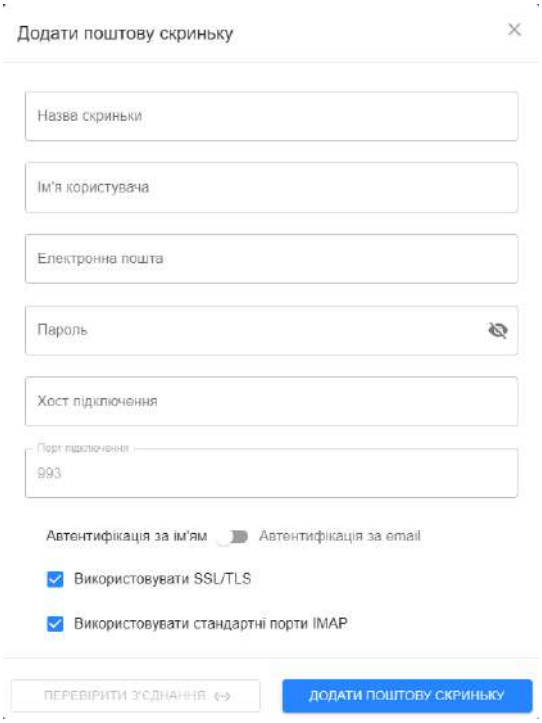


Рисунок 4.11 - Форма додавання нової поштової скриньки

Функціональність редагування існуючої поштової скриньки доступна через контекстне меню, яке відкривається при натисканні на кнопку з трьома крапками біля кожної скриньки.

Діалогове вікно редагування схоже на вікно додавання, але поля заповнюються існуючими даними. Вигляд форми наведено на рисунку 4.12.

Форма редагування містить ті ж поля, що й форма додавання, але з деякими відмінностями:

- поле пароля не заповнюється автоматично з міркувань безпеки;

- користувач може змінити будь-які параметри підключення.

Процес редагування:

- при відкритті форми завантажуються дані існуючої поштової скриньки;

- користувач вносить необхідні зміни;

- перед збереженням також виконується перевірка з'єднання;

- у разі успішної перевірки, оновлені дані відправляються на сервер;

- користувач отримує повідомлення про успішне оновлення або помилку;

- на серверній стороні запит обробляється через метод EditEmailBox контролера EmailBoxesController, який перевіряє, чи були внесені зміни, і оновлює тільки змінені поля.

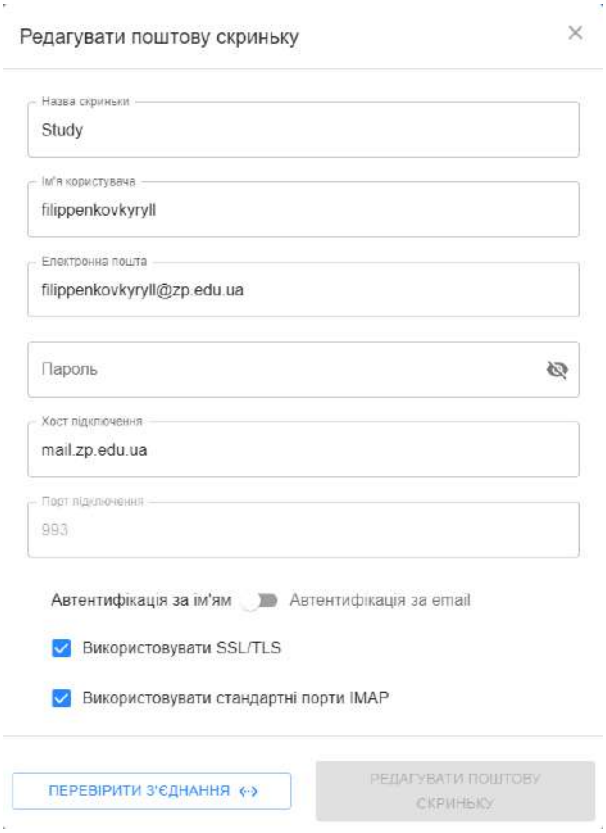


Рисунок 4.12 - Форма редагування поштової скриньки

Функціональність видалення поштової скриньки також доступна через контекстне меню. При виборі опції "Видалити" відображається діалогове вікно підтвердження, що наведене на рисунку 4.13.

Діалогове вікно містить:

- повідомлення з підтвердженням видалення, що включає назву цільової скриньки;

- кнопки "Скасувати" та "Видалити".

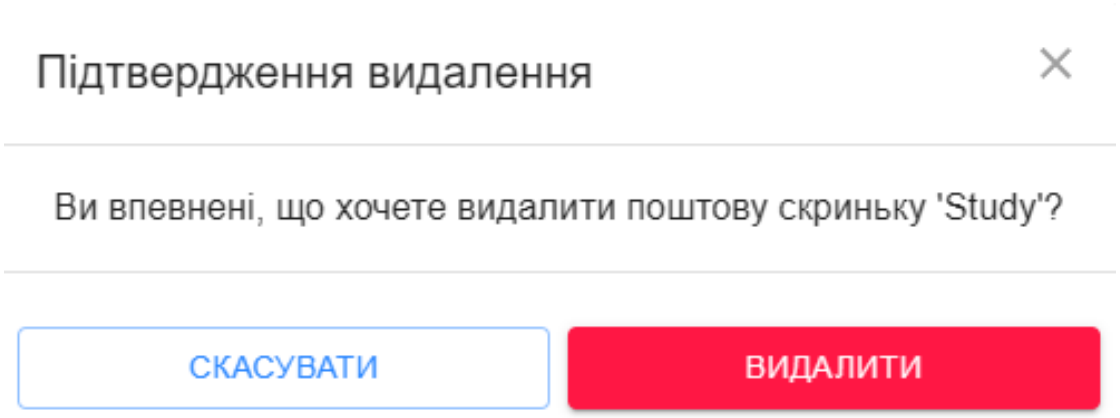


Рисунок 4.13 – Діалогове вікно підтвердження видалення поштової скриньки

4.1.5 Перегляд ланцюжків повідомлень

Система організовує повідомлення в ланцюжки. Користувач може перейти до перегляду ланцюжка повідомлень зі списку повідомлень на головній сторінці. Кожен елемент у списку повідомлень є посиланням на відповідний ланцюжок, приклад якого наведено на рисунку 4.14.

При натисканні на повідомлення користувач переходить на сторінку ланцюжка, де може переглянути всю історію листування.

Функціональність перегляду ланцюжків повідомлень реалізована через компонент ThreadPage, який відображає повну історію листування з певної теми. Сторінка ланцюжка повідомлень має наступну структуру:

- заголовок сторінки - містить кнопку повернення назад, тему ланцюжка, інформацію про кількість повідомлень та діапазон дат;

- список повідомлень - відображає всі повідомлення ланцюжка в хронологічному порядку.

Кожне повідомлення в ланцюжку відображається за допомогою компонента Message, який має два режими відображення:

- згорнутий режим - показує тільки тему, дату та короткий текст повідомлення;

- розгорнутий режим - показує повну інформацію про повідомлення, включаючи відправника, отримувача, повний текст, HTML-вміст (за наявності, приклад наведено на рисунку 4.15) та вкладення (за наявності – рисунок 4.16).

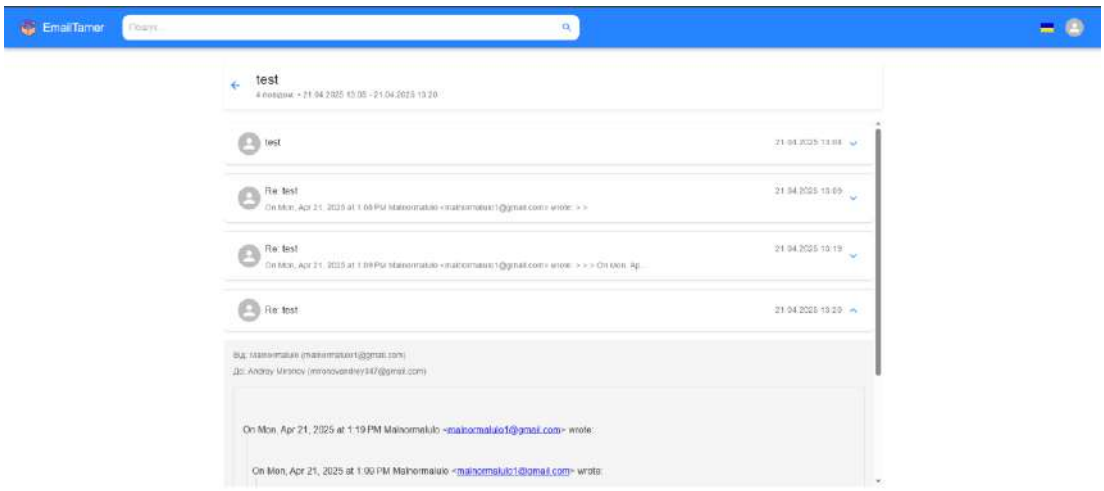


Рисунок 4.14 - Перегляд ланцюжка повідомлень

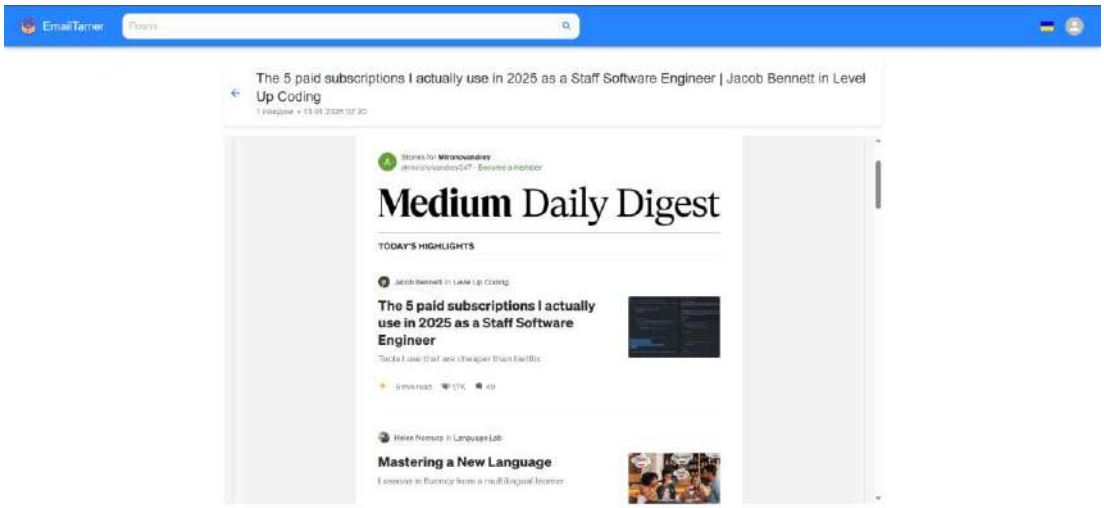


Рисунок 4.15 - Перегляд повідомлення із HTML-вмістом

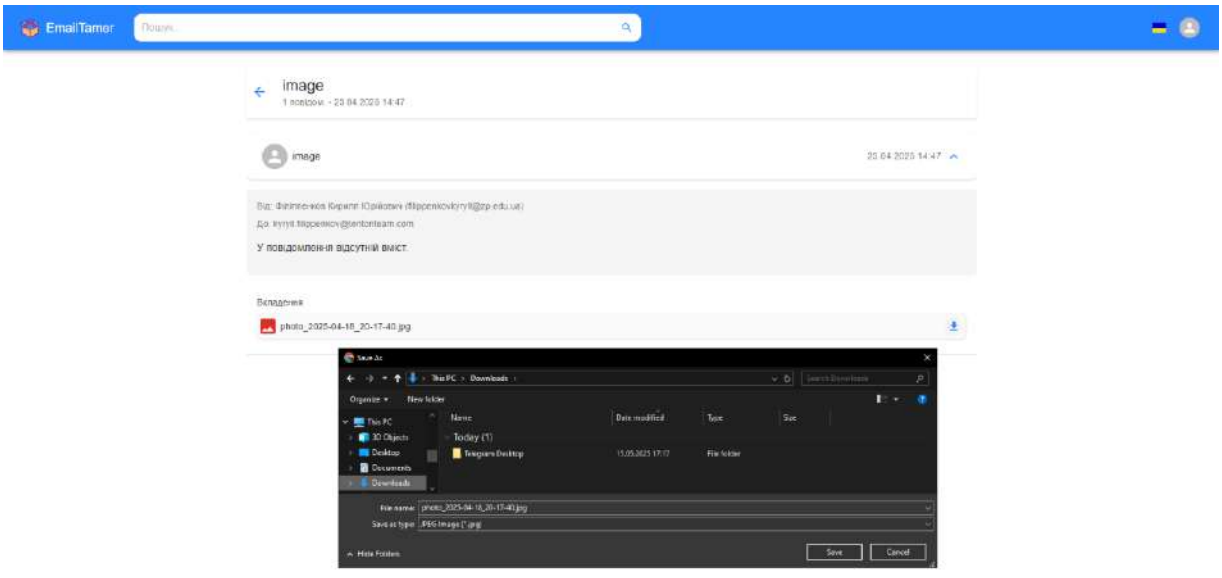


Рисунок 4.16 – Завантаження вкладення

4.2 Демонстрація ключових функціональних можливостей

4.2.1 Процес резервного копіювання

Процес резервного копіювання є ключовою функцією системи. Процес ініціюється автоматично за розкладом або користувач може запустити його вручну для вибраних скриньок – логи успішного виконання операції резервного копіювання наведено на рисунку 4.17. Під час процесу система:

- підключається до поштового сервера через IMAP;
- отримує нові повідомлення та їх вкладення;
- зберігає дані в базі даних користувача;
- відображає статус процесу в інтерфейсі.

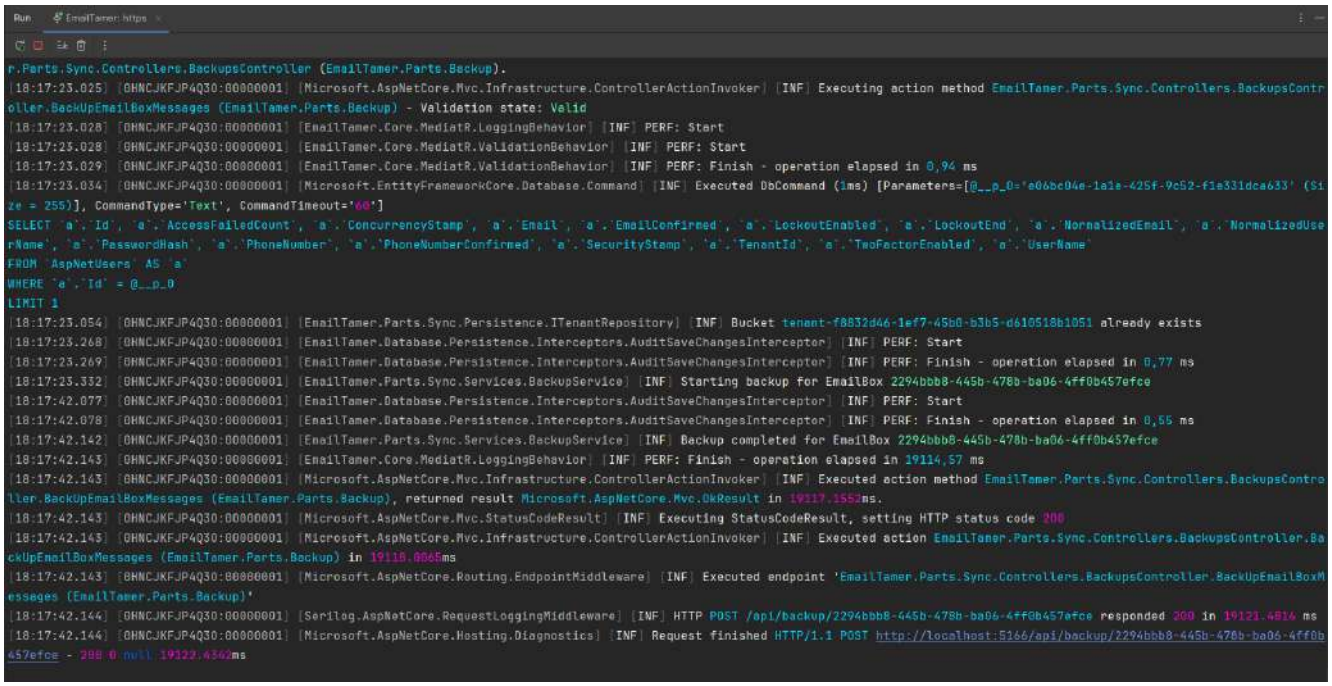


Рисунок 4.17 – Логи успішного виконання операції резервного копіювання

4.2.2 Автоматичне резервне копіювання

Система підтримує автоматичне резервне копіювання, яке виконується за розкладом без участі користувача. Сервіс PeriodicBackupService запускається щодня о 1:00 UTC та виконує резервне копіювання для всіх користувачів системи.

Логи успішного виконання роботи PeriodicBackupService наведено на рисунку 4.18.

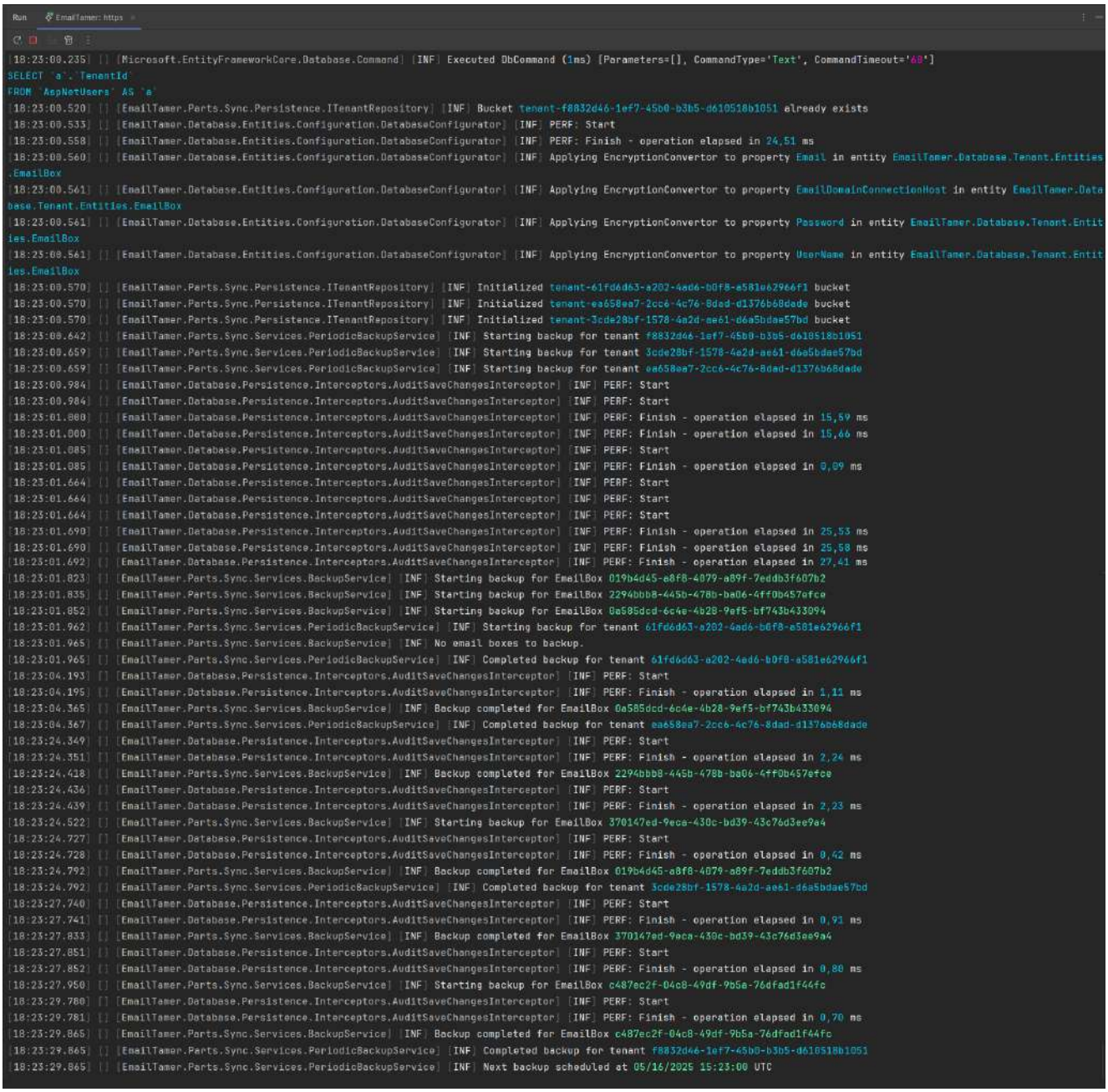


Рисунок 4.18 - Логи автоматичного резервного копіювання

4.2.3 Багатомовний інтерфейс

Система підтримує повну інтернаціоналізацію інтерфейсу. Реалізовано підтримку української та англійської мов. Всі елементи інтерфейсу, повідомлення та підказки доступні обома мовами. Демонстрацію української локалізації наведено на рисунку 4.19, англійської – на рисунку 4.20.

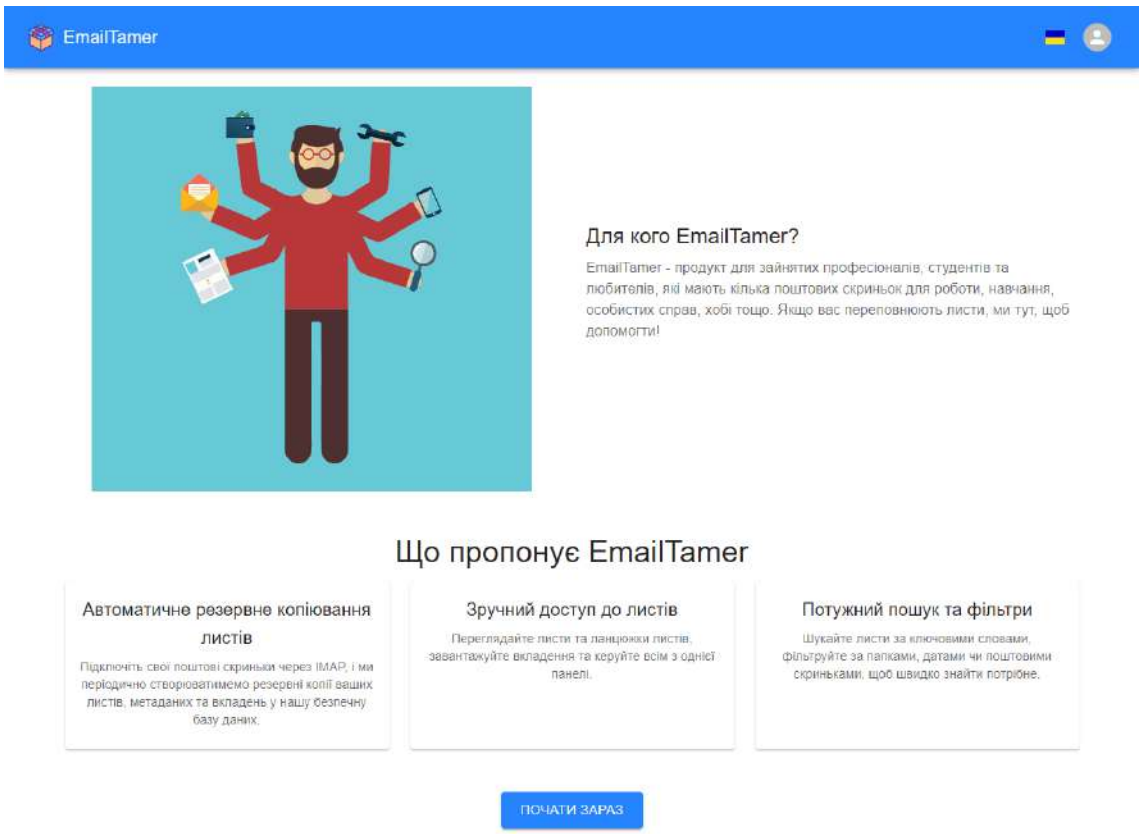


Рисунок 4.19 - Інтерфейс українською мовою

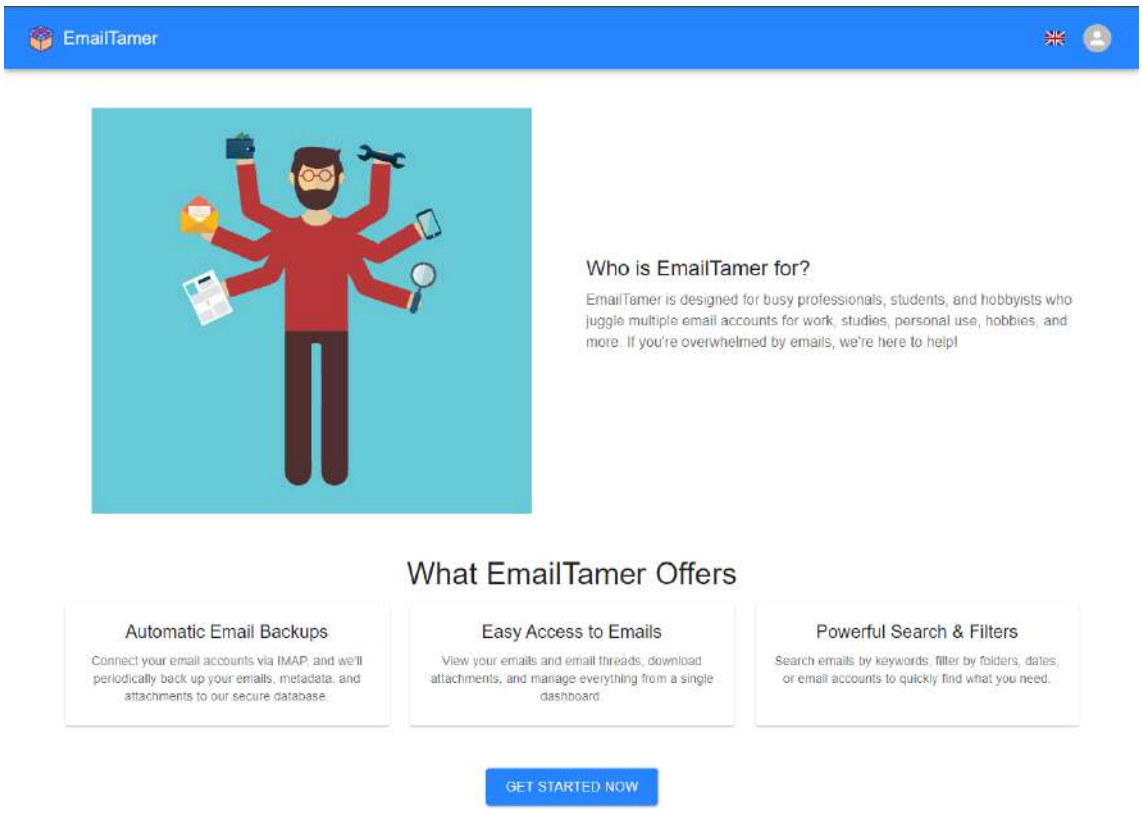


Рисунок 4.20 - Інтерфейс англійською мовою

4.3 Безпека та захист даних

4.3.1 Аутентифікація користувачів

Система забезпечує безпечну аутентифікацію користувачів за допомогою JWT-токенів. Всі API-запити до сервера, окрім реєстрації та входу, вимагають аутентифікації.

4.3.2 Хешування паролів облікових записів

Система не зберігає паролі від облікових записів користувачів - тільки хеш, паролі користувачів (EmailTamerUser) хешуються за допомогою стандартного механізму ASP.NET Identity, оскільки клас EmailTamerUser розширює базовий клас IdentityUser. Наочний приклад хешу паролів наведено на рисунку 4.21.

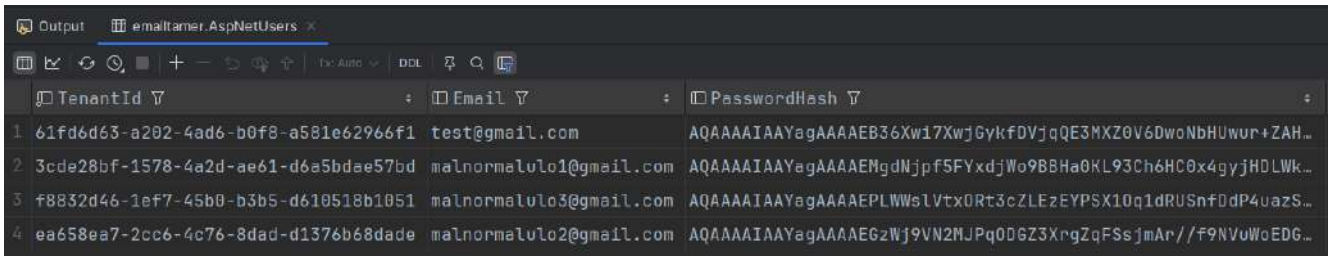


Рисунок 4.21 – Хеш паролів в головній БД застосунку

4.3.3 Шифрування чутливих даних

Чутливі дані, такі як поштові скриньки, паролі від них, хости поштових серверів, зберігаються в зашифрованому вигляді, що забезпечує додатковий рівень захисту. Наочний приклад зашифрованих даних наведено на рисунку 4.22.

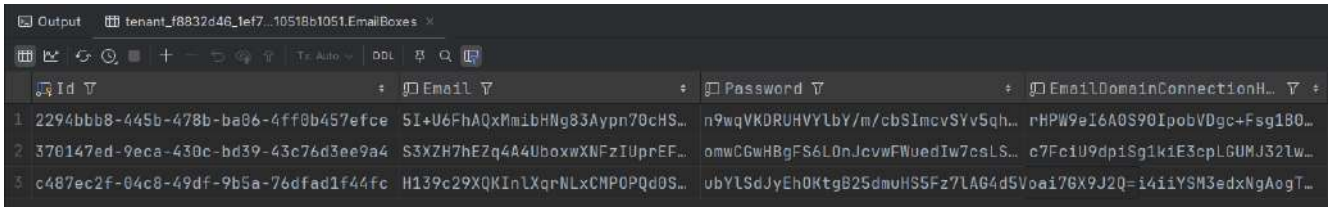


Рисунок 4.22 – Зашифровані чутливі дані

4.3.4 Ізоляція даних користувачів

Система використовує мультитенантну архітектуру, яка забезпечує повну ізоляцію даних різних користувачів. Кожен користувач має власну базу даних та сховище для зберігання повідомлень та вкладень. Окремі БД тенантів наочно продемонстровано на рисунку 4.23.

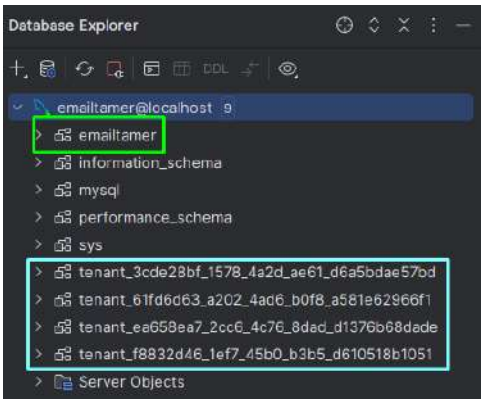


Рисунок 4.23 – Головна БД та БД тенантів

4.4 Висновки до розділу

У цьому розділі представлено результати розробки сервісу EmailTamer [54]. Система успішно реалізує всі заявлені функціональні вимоги:

- підключення до поштових скриньок через IMAP;
- резервне копіювання повідомлень та вкладень;
- зручний перегляд та управління повідомленнями;
- автоматичне резервне копіювання;
- багатомовний інтерфейс;
- адаптивний дизайн;
- безпека та захист даних.

Розроблений сервіс надає користувачам зручний та функціональний інструмент для централізованого резервного копіювання та перегляду електронних листів зі скриньок різних поштових клієнтів.

ВИСНОВКИ

Розроблений вебзастосунок EmailTamer дозволяє користувачам створювати резервні копії електронних листів через протокол IMAP, зберігати їх у реляційній базі даних (MySQL) і хмарному сховищі (Amazon S3), а також переглядати, шукати та фільтрувати листи через вебінтерфейс.

Аналіз предметної області підкреслив актуальність проблеми резервного копіювання електронних листів у сучасному світі, де пошта залишається одним з ключових інструментів комунікації. Порівняння з аналогічними сервісами (MailStore, Nylas, CloudHQ, Dropsuite) виявило їхні обмеження, такі як складність налаштування, висока вартість або відсутність мультитенантності, що дозволило позиціонувати EmailTamer як конкурентоспроможне рішення з мультитенантною архітектурою, інтеграцією з Amazon S3 і простим інтерфейсом. Вимоги до системи охопили функціональні аспекти (JWT-автентифікація, IMAP-синхронізація, збереження даних у MySQL і S3, пошук і фільтрація) та нефункціональні (безпека, масштабованість, продуктивність), що забезпечило комплексний підхід до розробки.

Для реалізації застосунку використано сучасний стек технологій: React і TypeScript для frontend, .NET 8 з Entity Framework Core, AutoMapper, MediatR, FluentValidation, Serilog і Swashbuckle для backend, а також MySQL і Docker для інфраструктури. TypeScript обрано замість JavaScript через статичну типізацію, яка зменшує помилки та полегшує підтримку коду. Інтеграція з MailKit для IMAP-синхронізації та AWSSDK.S3 для зберігання вкладень забезпечила надійність і масштабованість. Serilog реалізував структуроване логування, а Swashbuckle створив інтерактивну документацію API через Swagger.

Розроблений вебзастосунок дозволяє користувачам створювати бекапи листів, переглядати їх у зручному вебінтерфейсі, виконувати пошук і фільтрацію за різними критеріями, а також завантажувати вкладення. Мультитенантна архітектура гарантує ізоляцію даних, шифрування та JWT-автентифікація

забезпечують безпеку, а розгортання через Docker спрощує інсталяцію. EmailTamer перевершує аналоги за поєднанням функціональності, безпеки та доступності, що робить його цінним інструментом для індивідуальних і корпоративних користувачів.

Робота має практичне значення, оскільки пропонує ефективне рішення для захисту даних електронної пошти, яке може бути адаптоване для різних сценаріїв використання. Перспективи розвитку включають підтримку додаткових протоколів (POP3, SMTP), інтеграцію з іншими хмарними сховищами, а також додавання аналітичних функцій, таких як статистика листування чи прогнозування активності. Отримані результати та навички розробки вебзастосунків із хмарними технологіями стануть основою для подальших досліджень у галузі інформаційних технологій і кібербезпеки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. IMAP. URL: https://uk.wikipedia.org/wiki/IMAP (дата звернення: 08.05.2025).

2. How we chose the right database solution for Freshsales email content. URL: https://www.freshworks.com/explore-crm/how-we-chose-the-right-database-solution-for-freshsales-email-content/ (дата звернення: 08.05.2025).

3. Amazon S3 examples using SDK for .NET. URL: https://docs.aws.amazon.com/sdk-for-net/v3/developer-guide/csharp_s3_code_examples.html (дата звернення: 08.05.2025).

4. MySQL. URL: https://www.mysql.com/ (дата звернення: 08.05.2025).

5. Microsoft article: Multi-tenancy. URL: https://learn.microsoft.com/en-us/ef/core/miscellaneous/multitenancy (дата звернення: 03.05.2025).

6. Multitenancy in ASP.NET Core - Simplest Way to achieve Multitenancy. URL: https://codewithmukesh.com/blog/multitenancy-in-aspnet-core/ (дата звернення: 03.05.2025).

7. Multi-Tenancy with Separate Databases Approach in .NET Core — A blog creation example. URL: https://dotnetfullstackdev.medium.com/multi-tenancy-with-separate-databases-approach-in-net-core-a-blog-creation-example-20a2ebca7581 (дата звернення: 03.05.2025).

8. Mailstore. URL: https://www.mailstore.com/en/ (дата звернення: 08.05.2025).

9. POP3. URL: https://uk.wikipedia.org/wiki/Post_Office_Protocol (дата звернення: 08.05.2025).

10. Nylas. URL: https://www.nylas.com/ (дата звернення: 08.05.2025).

11. API. URL: https://en.wikipedia.org/wiki/API (дата звернення: 08.05.2025).

12. Cloudhq. URL: https://www.cloudhq.net/g_suite (дата звернення: 08.05.2025).

13. Gmail. URL: https://uk.wikipedia.org/wiki/Gmail (дата звернення: 08.05.2025).

14. Dropbox. URL: https://www.dropbox.com (дата звернення: 08.05.2025).

15. OneDrive. URL: https://uk.wikipedia.org/wiki/OneDrive (дата звернення: 08.05.2025).

16. Dropsuite. URL: https://dropsuite.com/ (дата звернення: 08.05.2025).

17. Wikipedia: Microsoft Exchange Server. URL: https://en.wikipedia.org/wiki/Microsoft_Exchange_Server (дата звернення: 08.05.2025).

18. Retrieve the chain mail of thread. URL: https://github.com/jstedfast/MailKit/issues/1254 (дата звернення: 03.05.2025).

19. Message threading. URL: https://www.jwz.org/doc/threading.html (дата звернення: 03.05.2025).

20. Encryption and Data Security in Clean Architecture using EF Core Value Converters: A Guide to Database Encryption Implementation. URL: https://gor-grigoryan.medium.com/encryption-and-data-security-in-clean-architecture-using-ef-core-value-converters-a-guide-to-911711a1ec52 (дата звернення: 03.05.2025).

21. React. URL: https://react.dev/ (дата звернення: 08.05.2025).

22. TypeScript is JavaScript with syntax for types. URL: https://www.typescriptlang.org/ (дата звернення: 03.05.2025).

23. Сучасний підручник з JavaScript. URL: https://uk.javascript.info/ (дата звернення: 03.05.2025).

24. React Router. URL: https://reactrouter.com/ (дата звернення: 08.05.2025).

25. React Hook Form. URL: https://react-hook-form.com/ (дата звернення: 08.05.2025).

26. Material UI: https://mui.com/material-ui/ (дата звернення: 08.05.2025).

27. Tanstack React Query. URL: https://tanstack.com/query/latest (дата звернення: 08.05.2025).

28. Zustand. URL: https://zustand-demo.pmnd.rs/ (дата звернення: 08.05.2025).

29. I18next. URL: https://www.i18next.com/ (дата звернення: 08.05.2025).

30. Vite. URL: https://vite.dev/ (дата звернення: 08.05.2025).

31. Webpack. URL: https://webpack.js.org/ (дата звернення: 08.05.2025).

32. .NET. URL: https://uk.wikipedia.org/wiki/.NET (дата звернення: 08.05.2025).

33. What is REST API? URL: https://www.ibm.com/think/topics/rest-apis (дата звернення: 08.05.2025).

34. ASP.NET Core. URL: https://dotnet.microsoft.com/en-us/apps/aspnet (дата звернення: 08.05.2025).

35. Learn Entity Framework Core: What is Entity Framework Core? URL: https://www.learnentityframeworkcore.com/ (дата звернення: 03.05.2025).

36. LINQ. URL: https://uk.wikipedia.org/wiki/LINQ (дата звернення: 03.05.2025).

37. AutoMapper. URL: https://automapper.org/ (дата звернення: 08.05.2025).

38. Чому вам варто спробувати бібліотеку MediatR для ваших .NET проектів. URL: https://devzone.org.ua/post/chomu-vam-varto-sprobuvaty-biblioteku-mediatr-dlia-vashykh-net-proektiv (дата звернення: 03.05.2025).

39. What is MediatR in .net core? Why we use it. URL: https://medium.com/@neer.s/what-is-mediatr-in-net-core-why-we-use-it-50e38bfe6f5b (дата звернення: 03.05.2025).

40. CQRS. URL: https://uk.wikipedia.org/wiki/CQRS (дата звернення: 08.05.2025).

41. FluentValidation. URL: https://docs.fluentvalidation.net/en/latest/ (дата звернення: 08.05.2025).

42. JSON Web Token. URL: https://en.wikipedia.org/wiki/JSON_Web_Token (дата звернення: 08.05.2025).

43. MailKit Documentation. URL: https://mimekit.net/docs/html/Introduction.htm (дата звернення: 03.05.2025).

44. AWSSDK.S3. URL: https://www.nuget.org/packages/AWSSDK.S3 (дата звернення: 03.05.2025).

45. Serilog. URL: https://serilog.net/ (дата звернення: 03.05.2025).

46. Nuget package: Swashbuckle.AspNetCore. URL: https://www.nuget.org/packages/swashbuckle.aspnetcore/ (дата звернення: 03.05.2025).

47. API Development forEveryone. URL: https://swagger.io/ (дата звернення: 03.05.2025).

48. OpenAPI Specification. URL: https://swagger.io/specification/ (дата звернення: 03.05.2025).

49. Nuget package: MySql.EntityFrameworkCore. URL: https://www.nuget.org/packages/MySql.EntityFrameworkCore (дата звернення: 08.05.2025).

50. Run Locally, Deploy Globally. URL: https://www.localstack.cloud/ (дата звернення: 08.05.2025).

51. Develop faster. Run anywhere. URL: https://www.docker.com/ (дата звернення: 08.05.2025).

52. ESLint. URL: https://eslint.org/ (дата звернення: 08.05.2025).

53. OpenAPI Codegen. https://swagger.io/tools/swagger-codegen/ (дата звернення: 08.05.2025).

54. EmailTamer repository. https://github.com/Malnormalulos-git/email-tamer (дата звернення: 21.05.2025).

ДОДАТОК А
ЛІСТИНГИ ПРОГРАМ

Лістинг А.1 – IBackupService

```
public interface IBackupService
{
    Task<IActionResult> BackupEmailBoxAsync(
        Guid emailBoxId,
        IEmailTamerRepository repository,
        ITenantRepository tenantRepository,
        CancellationToken cancellationToken);

    Task<IActionResult> BackupTenantEmailBoxesAsync(
        Guid[]? emailBoxesIds,
        IEmailTamerRepository repository,
        ITenantRepository tenantRepository,
        CancellationToken cancellationToken);
}
```

Лістинг А.2 - BackupService

```
internal class BackupService(
    IMapper mapper,
    ISystemClock clock,
    ILogger<BackupService> logger)
    : IBackupService
{
    private const int BatchSize = 100;

    public async Task<IActionResult> BackupEmailBoxAsync(
        Guid emailBoxId,
        IEmailTamerRepository repository,
        ITenantRepository tenantRepository,
        CancellationToken cancellationToken)
    {
        var (emailBox, backedUpMessages, existingFolders) =
            await LoadInitialData(emailBoxId, repository,
cancellationToken);

        if (emailBox is null)
            return new NotFoundResult();

        emailBox.BackupStatus = BackupStatus.InProgress;
        repository.Update(emailBox);
        await repository.PersistAsync(cancellationToken);

        logger.LogInformation("Starting backup for EmailBox
{EmailBoxId}", emailBoxId);
```

```
        try
        {
            using var client = await
MailKitImapConnector.ConnectToImapClient(emailBox,
cancellationToken);

            var folders = await GetRelevantFolders(client,
cancellationToken);
            var synchronizationStartedAt = clock.UtcNow.DateTime;
            var newMessagesDictionary = new Dictionary<string,
Message>();

            foreach (var folder in folders)
            {
                await ProcessFolder(
                    folder,
                    emailBox,
                    backedUpMessages,
                    existingFolders,
                    newMessagesDictionary,
                    repository,
                    tenantRepository,
                    cancellationToken);
            }

            await client.DisconnectAsync(true, cancellationToken);

            AssignThreadIds(backedUpMessages,
newMessagesDictionary);

            await
repository.WriteInTransactionAsync(IsolationLevel.ReadCommitted,
async (repo, ct) =>
            {
                if (newMessagesDictionary.Count > 0)
                {
                    var messages =
newMessagesDictionary.Values.ToList();

                    for (var i = 0; i < messages.Count; i +=
BatchSize)
                    {
                        var batch =
messages.Skip(i).Take(BatchSize).ToList();
                        repo.AddRange(batch);
                    }
                }

                emailBox.ConnectionFault = null;
                emailBox.LastSyncAt = synchronizationStartedAt;
                emailBox.BackupStatus = BackupStatus.Idle;
                repo.Update(emailBox);
```

```
                await repo.PersistAsync(ct);
            }, cancellationToken);

            logger.LogInformation("Backup completed for EmailBox
{EmailBoxId}", emailBoxId);
            return new OkResult();
        }
        catch (Exception e)
        {
            logger.LogError(e, "Error while backing up EmailBox
{EmailBoxId}", emailBoxId);

            emailBox.BackupStatus = BackupStatus.Failed;
            repository.Update(emailBox);
            await repository.PersistAsync(cancellationToken);

            if (e is MailKitImapConnectorException
mailKitImapConnectorException)
            {
                emailBox.ConnectionFault =
mailKitImapConnectorException.Fault;
                repository.Update(emailBox);
                await repository.PersistAsync(cancellationToken);
                return new
BadRequestObjectResult(mailKitImapConnectorException.Fault);
            }

            return new BadRequestResult();
        }
    }

    public async Task<IActionResult> BackupTenantEmailBoxesAsync(
        Guid[]? emailBoxesIds,
        IEmailTamerRepository repository,
        ITenantRepository tenantRepository,
        CancellationToken cancellationToken)
    {
        var filterByEmailBoxes = emailBoxesIds != null &&
emailBoxesIds.Length > 0;

        var emailBoxes = await repository.ReadAsync((r, ct) =>
                r.Set<EmailBox>()
                    .AsNoTracking()
                    .WhereIf(filterByEmailBoxes, x =>
emailBoxesIds!.Contains(x.Id))
                    .Where(x => x.BackupStatus !=
BackupStatus.Queued && x.BackupStatus != BackupStatus.InProgress)
                    .ToListAsync(ct),
            cancellationToken);

        if (!emailBoxes.Any())
        {
            logger.LogInformation("No email boxes to backup.");
```

```
            return new OkResult();
        }

        foreach (var emailBox in emailBoxes)
        {
            emailBox.BackupStatus = BackupStatus.Queued;
        }

        repository.UpdateRange(emailBoxes);
        await repository.PersistAsync(cancellationToken);

        foreach (var emailBox in emailBoxes)
        {
            try
            {
                var result = await BackupEmailBoxAsync(emailBox.Id,
repository, tenantRepository, cancellationToken);
                if (result is not OkResult)
                {
                    logger.LogWarning("Backup failed for EmailBox
{EmailBoxId}", emailBox.Id);
                }
            }
            catch (Exception ex)
            {
                logger.LogError(ex, "Error during backup of EmailBox
{EmailBoxId}", emailBox.Id);
            }
        }

        return new OkResult();
    }

    private async Task<(EmailBox? EmailBox, List<Message>
BackedUpMessages, List<Folder> ExistingFolders)>
        LoadInitialData(Guid emailBoxId, IEmailTamerRepository
repository, CancellationToken cancellationToken)
    {
        return await repository.ReadAsync(async (r, ct) =>
        {
            var box = await r.Set<EmailBox>()
                .FirstOrDefaultAsync(x => x.Id == emailBoxId, ct);

            if (box == null)
                return (null, [], []);

            var messages = await r.Set<Message>()
                .Include(x => x.EmailBoxes)
                .Include(x => x.Folders)
                .ToListAsync(ct);

            var folders = await r.Set<Folder>()
                .ToListAsync(ct);
```

```
            return (box, messages, folders);
        }, cancellationToken);
    }

    private async Task<List<IMailFolder>>
GetRelevantFolders(IImapClient client, CancellationToken
cancellationToken)
    {
        var rootNamespace = client.PersonalNamespaces.Count > 0
            ? client.PersonalNamespaces[0]
            : new FolderNamespace('.', "");

        var folders = (await client.GetFoldersAsync(rootNamespace,
cancellationToken: cancellationToken))
            .Where(f =>
                !f.Attributes.HasFlag(FolderAttributes.NonExistent)
&&
                !f.Attributes.HasFlag(FolderAttributes.Drafts) &&
                !f.Attributes.HasFlag(FolderAttributes.Trash) &&
                !f.Attributes.HasFlag(FolderAttributes.Junk) &&
                !f.Attributes.HasFlag(FolderAttributes.All))
            .ToList();

        if (!folders.Contains(client.Inbox))
        {
            folders.Add(client.Inbox);
        }

        return folders;
    }

    private async Task ProcessFolder(
        IMailFolder folder,
        EmailBox emailBox,
        List<Message> backedUpMessages,
        List<Folder> existingFolders,
        Dictionary<string, Message> newMessagesDictionary,
        IEmailTamerRepository repository,
        ITenantRepository tenantRepository,
        CancellationToken cancellationToken)
    {
        await folder.OpenAsync(FolderAccess.ReadOnly,
cancellationToken);
        var messages = await folder.SearchAsync(SearchQuery.All,
cancellationToken);

        foreach (var messageIndex in messages)
        {
            var mimeMessage = await
folder.GetMessageAsync(messageIndex, cancellationToken);
            await ProcessMessage(
                mimeMessage,
```

```
                folder,
                emailBox,
                backedUpMessages,
                existingFolders,
                newMessagesDictionary,
                repository,
                tenantRepository,
                cancellationToken);
        }

        await folder.CloseAsync(cancellationToken:
cancellationToken);
    }

    private async Task ProcessMessage(
        MimeMessage mimeMessage,
        IMailFolder folder,
        EmailBox emailBox,
        List<Message> backedUpMessages,
        List<Folder> existingFolders,
        Dictionary<string, Message> newMessagesDictionary,
        IEmailTamerRepository repository,
        ITenantRepository tenantRepository,
        CancellationToken cancellationToken)
    {
        var backedUpMessage = backedUpMessages
            .FirstOrDefault(x => x.Id == mimeMessage.MessageId);
        if (backedUpMessage != null)
        {
            await ProcessExistingMessage(backedUpMessage, folder,
emailBox, existingFolders, repository);
            return;
        }

        if
(!newMessagesDictionary.TryGetValue(mimeMessage.MessageId, out var
newMessage))
        {
            newMessage = await CreateNewMessage(mimeMessage,
emailBox, tenantRepository, cancellationToken);
            newMessagesDictionary[mimeMessage.MessageId] =
newMessage;
        }

        await ProcessMessageFolder(newMessage, folder,
existingFolders, repository);
    }

    private void AssignThreadIds(List<Message> backedUpMessages,
Dictionary<string, Message> newMessagesDictionary)
    {
        var allMessages =
backedUpMessages.Concat(newMessagesDictionary.Values).ToList();
```

```
        var messageLookup = allMessages.ToDictionary(m => m.Id, m =>
m);

        foreach (var message in newMessagesDictionary.Values)
        {
            if (string.IsNullOrEmpty(message.InReplyTo) &&
                (message.References.Count == 0))
            {
                message.ThreadId = message.Id;
                continue;
            }

            string threadId = null;

            if (!string.IsNullOrEmpty(message.InReplyTo) &&
                messageLookup.TryGetValue(message.InReplyTo, out var
repliedTo))
            {
                threadId = repliedTo.ThreadId ?? repliedTo.Id;
            }

            if (threadId == null && message.References.Count > 0)
            {
                foreach (var refId in message.References)
                {
                    if (messageLookup.TryGetValue(refId, out var
refMessage))
                    {
                        threadId = refMessage.ThreadId ?? refId;
                        break;
                    }
                }
            }

            message.ThreadId = threadId ?? message.Id;
        }
    }

    private Task ProcessExistingMessage(
        Message message,
        IMailFolder folder,
        EmailBox emailBox,
        List<Folder> existingFolders,
        IEmailTamerRepository repository)
    {
        if (!string.IsNullOrEmpty(folder.Name) &&
            message.Folders.All(x => x.Name != folder.Name))
        {
            var existingFolder = existingFolders.FirstOrDefault(x =>
x.Name == folder.Name)
                                 ?? CreateNewFolder(folder.Name,
existingFolders, repository);
```

```
            message.Folders.Add(existingFolder);
            repository.Update(message);
        }

        if (!message.EmailBoxes.Contains(emailBox))
        {
            message.EmailBoxes.Add(emailBox);
            repository.Update(message);
        }

        return Task.CompletedTask;
    }

    private Folder CreateNewFolder(string folderName, List<Folder>
existingFolders, IEmailTamerRepository repository)
    {
        var newFolder = new Folder
        {
            Id = Guid.NewGuid(),
            Name = folderName
        };

        existingFolders.Add(newFolder);
        repository.Add(newFolder);
        return newFolder;
    }

    private async Task<Message> CreateNewMessage(
        MimeMessage mimeMessage,
        EmailBox emailBox,
        ITenantRepository tenantRepository,
        CancellationToken cancellationToken)
    {
        var newMessage = mapper.Map<Message>(mimeMessage);
        newMessage.EmailBoxes.Add(emailBox);

        var saveToRepoTasks = new List<Task>();

        if (mimeMessage.HtmlBody != null)
        {
            saveToRepoTasks.Add(SaveMessageBody(mimeMessage,
tenantRepository, cancellationToken));
        }

        await ProcessAttachments(mimeMessage, newMessage,
saveToRepoTasks, tenantRepository, cancellationToken);
        await Task.WhenAll(saveToRepoTasks);

        return newMessage;
    }
```

```
    private Task SaveMessageBody(MimeMessage mimeMessage,
ITenantRepository tenantRepository, CancellationToken
cancellationToken)
    {
        return tenantRepository.SaveBodyAsync(
            new MessageBodyKey { MessageId = mimeMessage.MessageId
},
            new MessageBody(new
MemoryStream(System.Text.Encoding.UTF8.GetBytes(mimeMessage.HtmlBody
!))),
            cancellationToken);
    }

    private async Task ProcessAttachments(MimeMessage mimeMessage,
Message newMessage, List<Task> saveToRepoTasks,
        ITenantRepository tenantRepository, CancellationToken
cancellationToken)
    {
        foreach (var attachment in mimeMessage.Attachments)
        {
            if (!attachment.IsAttachment) continue;

            var fileName = attachment.ContentDisposition?.FileName
?? attachment.ContentType.Name;
            var content = ((MimePart)attachment).Content.Open();
            var contentStream = new MemoryStream();
            await content.CopyToAsync(contentStream,
cancellationToken);

            var newAttachment = new Attachment
            {
                Id = Guid.NewGuid(),
                FileName = fileName,
                MessageId = newMessage.Id
            };

            saveToRepoTasks.Add(
                tenantRepository.SaveAttachmentAsync(
                    MessageAttachmentKey.Create(newAttachment,
mimeMessage),
                    new MessageAttachment(
                        contentStream,
                        fileName,
                        attachment.ContentType.MimeType),
                    cancellationToken));

            newMessage.Attachments.Add(newAttachment);
        }
    }

    private Task ProcessMessageFolder(Message message, IMailFolder
folder, List<Folder> existingFolders, IEmailTamerRepository
repository)
```

```
    {
        if (string.IsNullOrEmpty(folder.Name))
            return Task.CompletedTask;

        var existingFolder = existingFolders.FirstOrDefault(x =>
x.Name == folder.Name)
                              ?? CreateNewFolder(folder.Name,
existingFolders, repository);

        message.Folders.Add(existingFolder);
        return Task.CompletedTask;
    }
}
```

Лістинг А.3 – PeriodicBackupService

```
internal class PeriodicBackupService(
    ILogger<PeriodicBackupService> logger,
    IServiceScopeFactory factory)
    : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            var now = DateTimeOffset.UtcNow.DateTime;
            var today = now.Date;
            var nextRun = today.AddHours(1);

            if (now > nextRun)
            {
                nextRun = nextRun.AddDays(1);
            }

            var delay = nextRun - now;

            logger.LogInformation("Next backup scheduled at
{NextRun} UTC", nextRun);
            await Task.Delay(delay, stoppingToken);

            try
            {
                await using var mainScope =
factory.CreateAsyncScope();
                var mainRepository = mainScope.ServiceProvider

.GetRequiredKeyedService<IEmailTamerRepository>(nameof(EmailTamerDbC
ontext));

                var tenants = await mainRepository.ReadAsync((r, ct)
=>
                        r.Set<EmailTamerUser>()
```

```
                            .AsNoTracking()
                            .Select(u => new
TenantContextAccessor(u.TenantId))
                            .ToListAsync(ct),
                    stoppingToken);

                var backupTasks = tenants.Select(async tenant =>
                {
                    try
                    {
                        await using AsyncServiceScope scope =
factory.CreateAsyncScope();
                        var backupService =
scope.ServiceProvider.GetRequiredService<IBackupService>();

                        var tenantRepositoryFactory =
scope.ServiceProvider.GetRequiredService<ITenantRepositoryFactory>()
;
                        var tenantRepository = await
tenantRepositoryFactory.Create(tenant, stoppingToken);

                        var dbContextFactory =
scope.ServiceProvider.GetRequiredService<IDbContextFactory<TenantDbC
ontext>>();
                        var databasePolicySet =
scope.ServiceProvider.GetRequiredService<IDatabasePolicySet>();
                        var configuration =
scope.ServiceProvider.GetRequiredService<IConfiguration>();
                        var encServLogger =
scope.ServiceProvider.GetRequiredService<ILogger<EncryptionService>>
();
                        var encryptionService = new
EncryptionService(tenant, configuration, encServLogger);
                        var dbContext = (dbContextFactory as
TenantDbContextFactory)?.CreateDbContext(tenant, encryptionService);
                        var repository = new
EmailTamerRepository<TenantDbContext>(dbContext, databasePolicySet);

                        var tenantId = await tenant.GetTenantId();
                        logger.LogInformation("Starting backup for
tenant {TenantId}", tenantId);
                        var result = await
backupService.BackupTenantEmailBoxesAsync(
                            null,
                            repository,
                            tenantRepository,
                            stoppingToken);
                        logger.LogInformation("Completed backup for
tenant {TenantId}", tenantId);
                    }
                    catch (Exception ex)
                    {
```

```
                        logger.LogError(ex, "Failed to backup tenant
{TenantId}", await tenant.GetTenantId());
                    }
                });

                await Task.WhenAll(backupTasks);
            }
            catch (Exception ex)
            {
                logger.LogError(ex, "Failed to execute
PeriodicHostedService");
            }
        }
    }
}
```

ДОДАТОК Б

ГРАФІЧНІ МАТЕРІАЛИ

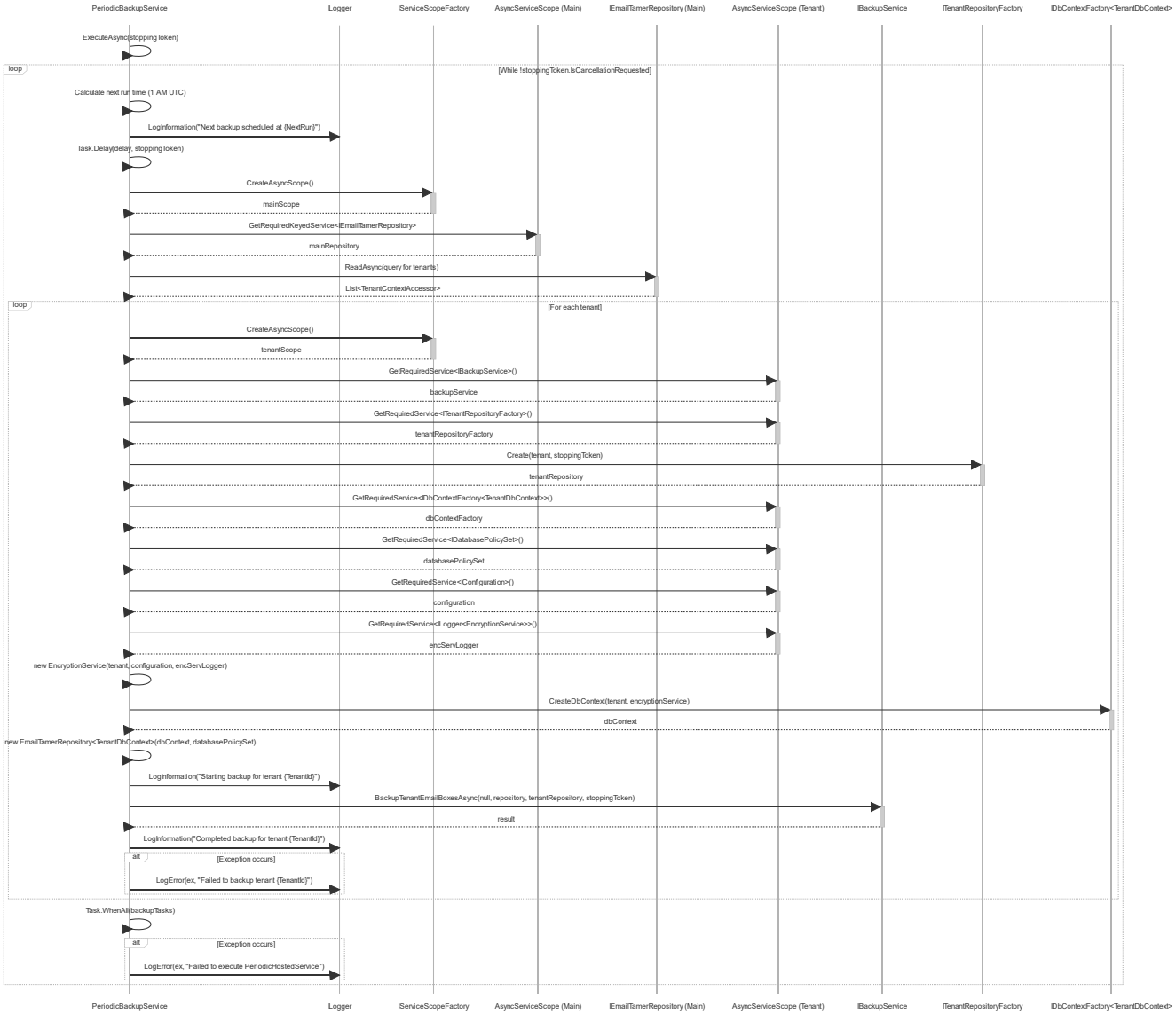


Рисунок Б.1 – Діаграма послідовності сервісу резервного копіювання

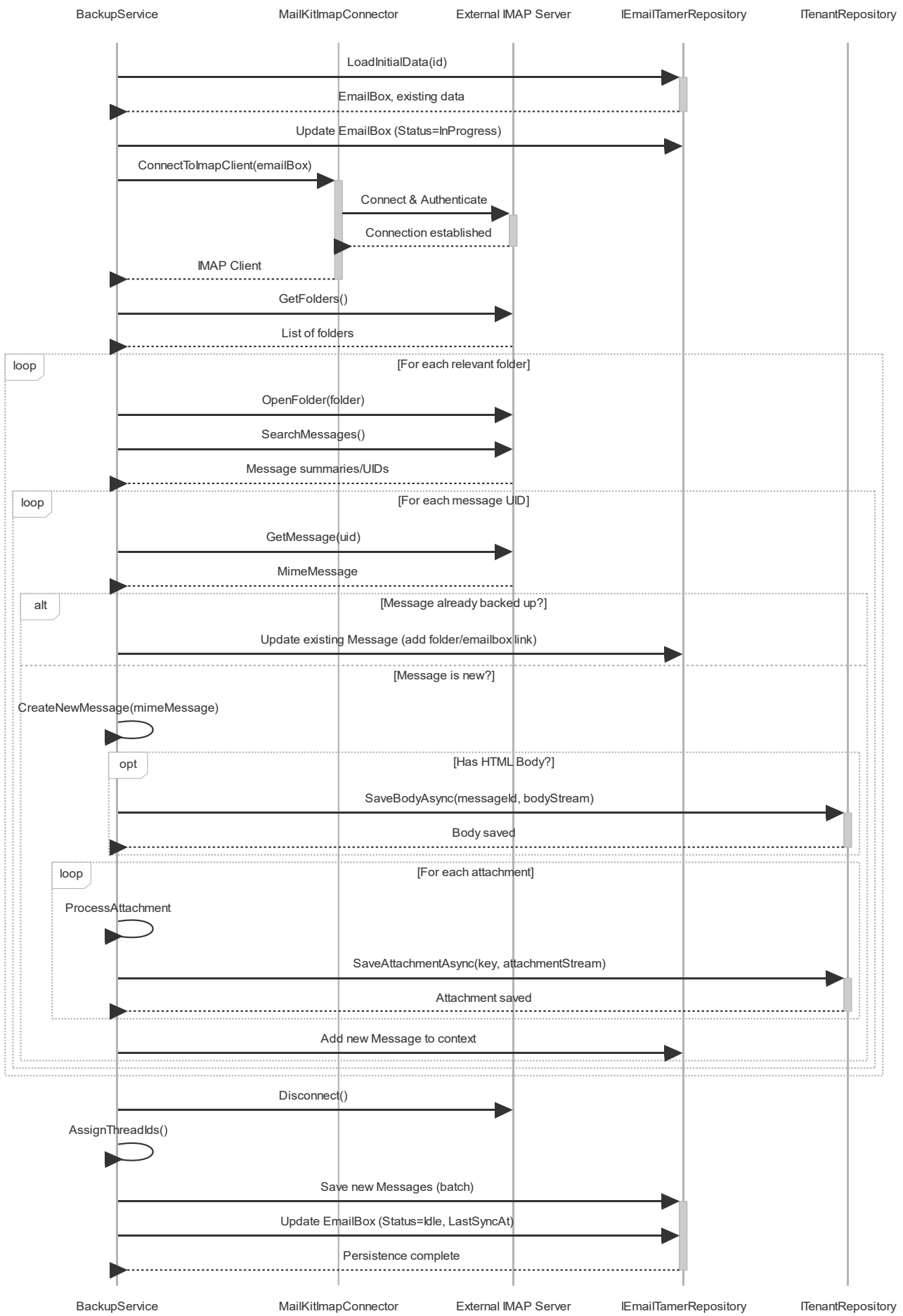


Рисунок Б.2 – Діаграма послідовності резервного копіювання поштової скриньки

ДИПЛОМНА РОБОТА

РОЗРОБКА СЕРВІСУ РЕЗЕРВУВАННЯ ЕЛЕКТРОННИХ ЛИСТІВ

Розробив: ст. гр. КНТ-521 К.Ю. ФІЛІППЕНКОВ
Керівник: доцент, к.т.н. М.Ю. ТЯГУНОВА

МЕТА РОБОТИ

Створення безпечного, зручного та масштабованого сервісу для автоматизованого резервного копіювання електронних листів, їхнього зберігання в реляційній базі даних і хмарному сховищі, а також забезпечення зручного доступу через вебінтерфейс.

Об'єкт розробки – вебзастосунок EmailTamer для резервного копіювання електронних листів із можливістю їхнього зберігання, пошуку та перегляду.

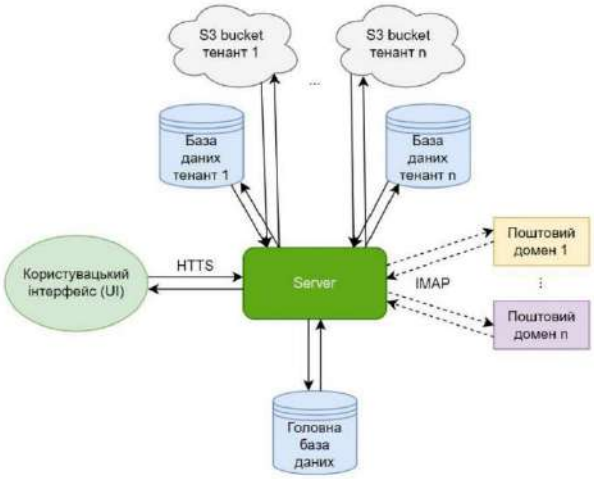
ТИПОВІ ПРОБЛЕМИ ВЗАЄМОДІЇ ІЗ ЕЛЕКТРОННОЮ ПОШТОЮ

- Наявність кількох поштових скриньок;
- Великий обсяг електронних листів;
- Втрата електронних листів чи доступу до поштових скриньок через апаратні збої, кібератаки, вичерпання обсягу сховища поштового сервісу або помилки користувача.

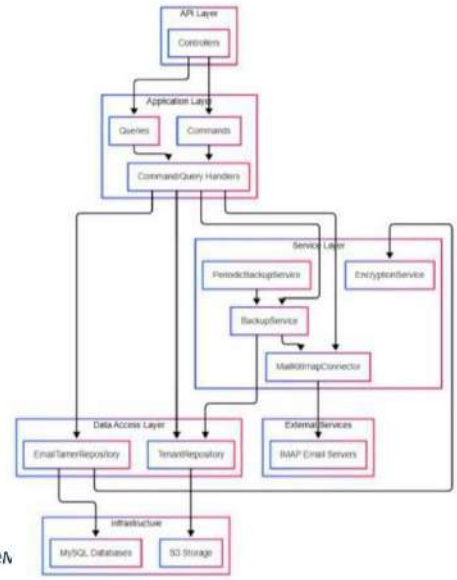
ОСНОВНІ ВИМОГИ ДО ПРОЄКТНОГО РІШЕННЯ

- Аутентифікація та реєстрація для користувачів системи;
- Мультитенантність
- Синхронізація з поштовими скриньками;
- Автоматизація резервного копіювання;
- Збереження даних;
- Перегляд листів;
- Пошук і фільтрація;
- Завантаження вкладень;
- Шифрування даних;
- Зручний інтерактивний інтерфейс.

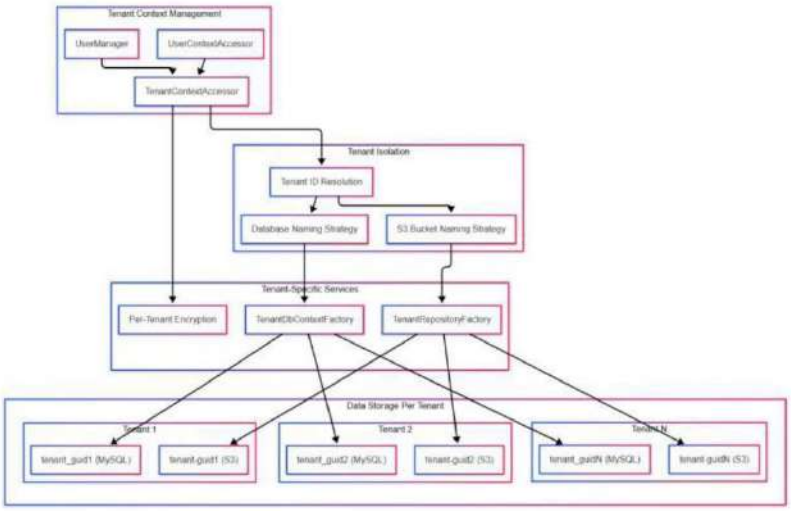
СТРУКТУРА СИСТЕМИ



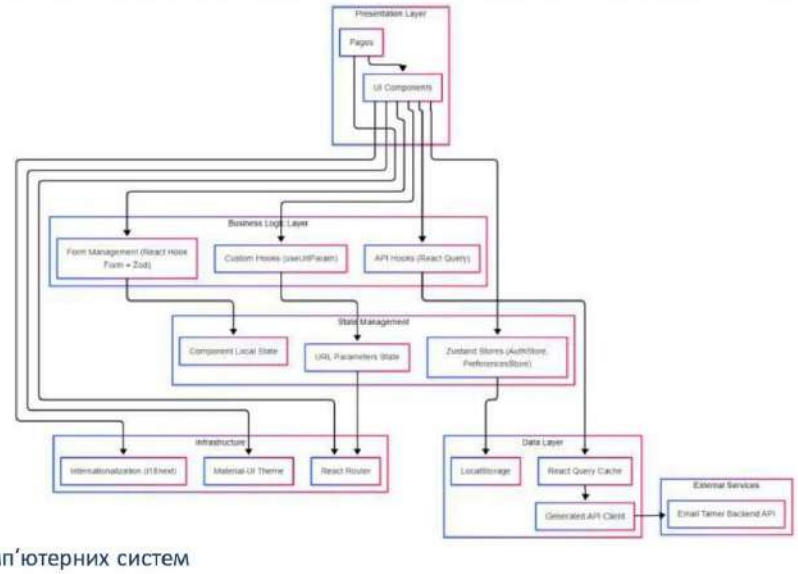
СТРУКТУРА СЕРВЕРНОЇ ЧАСТИНИ



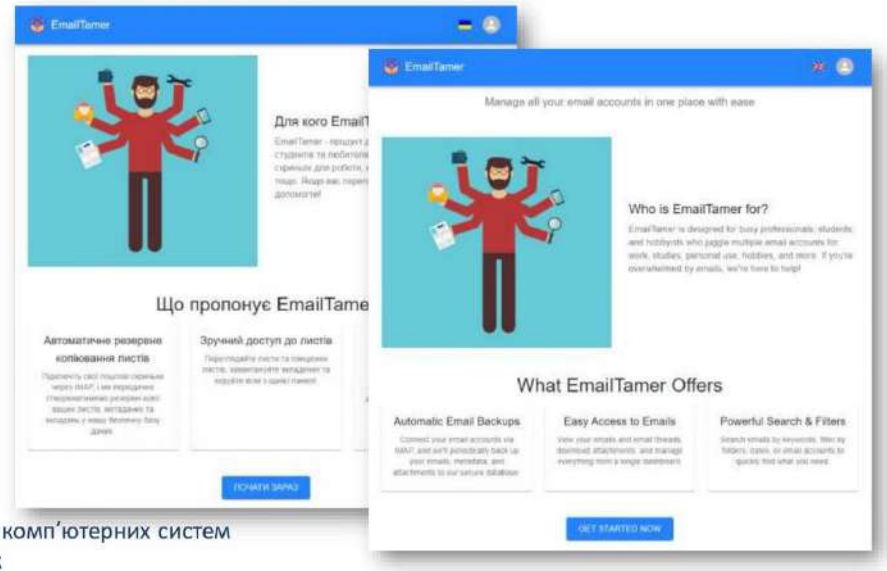
МУЛЬТИТЕНАНТНА АРХІТЕКТУРА



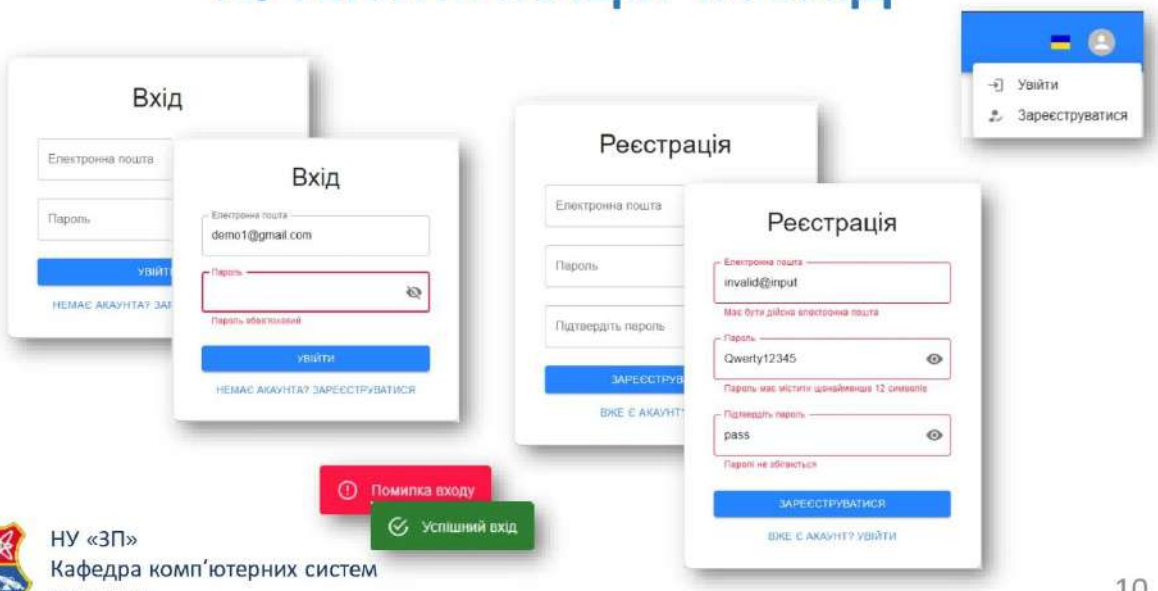
СТРУКТУРА КЛІЄНТСЬКОЇ ЧАСТИНИ



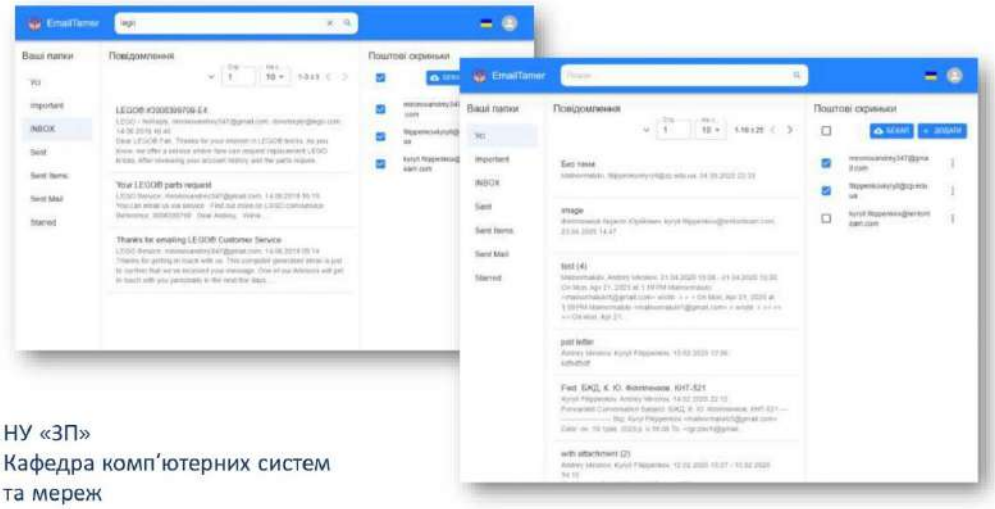
ЛОКАЛІЗАЦІЯ



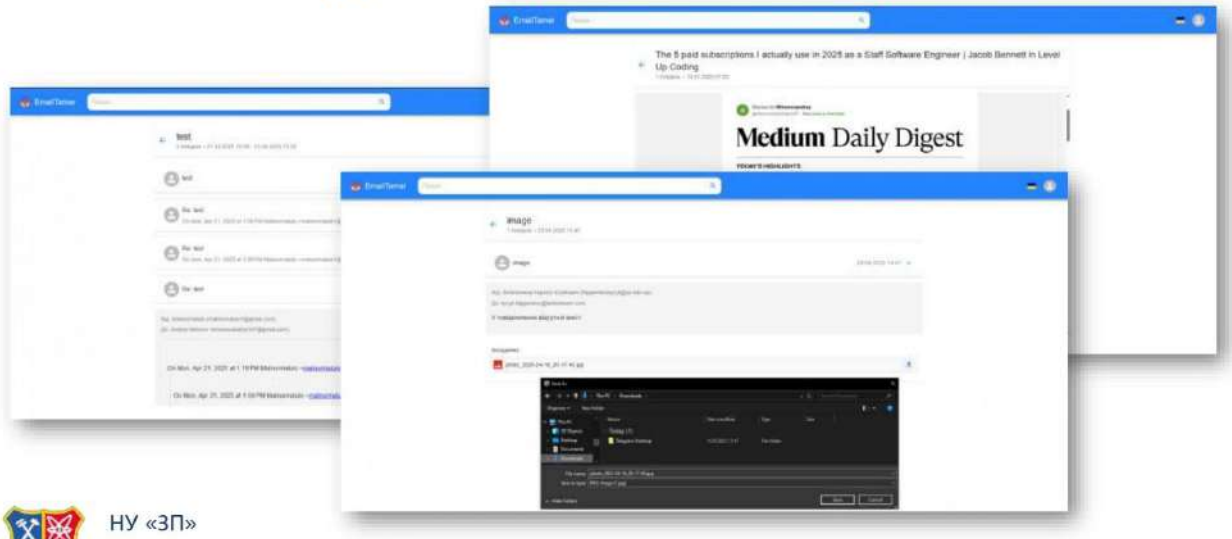
АУТЕНТИФІКАЦІЯ ТА ВХІД



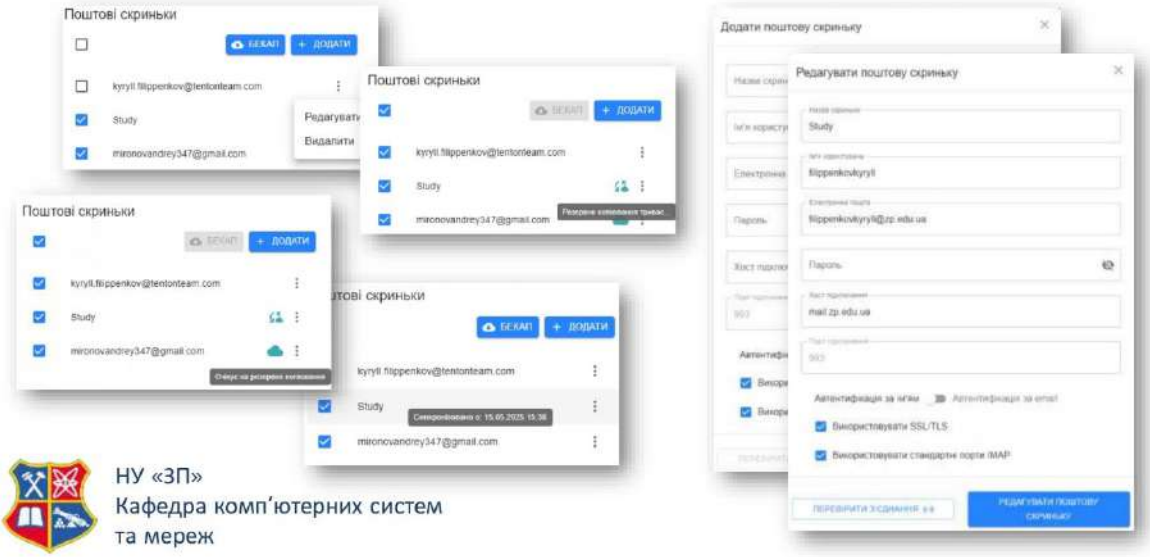
ОТРИМАННЯ ЛАНЦЮЖКІВ ПОВІДОМЛЕНЬ, ФІЛЬТРАЦІЯ ТА ПОШУК



ПЕРЕГЛЯД ЛАНЦЮЖКУ ПОВІДОМЛЕНЬ



ОПЕРАЦІЇ НА ПОШТОВИМИ СКРИНЬКАМИ



ВИСНОВКИ

- Проведено аналіз предметної області;
- Спроєктовано систему згідно висунутих вимог;
- Успішно реалізовано сервіс резервного копіювання електронної пошти;
- Проведено тестування системи.

ДОКЛАД ЗАКІНЧЕНО

ДЯКУЮ ЗА УВАГУ!