

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Кафедра інформаційних систем та технологій

Тема E-mail клієнт

Курсова робота

З дисципліни «Технології розроблення програмного забезпечення»

Керівник

доц. Амонс О.А.

«Допущений до захисту»

(Особистий підпис керівника)

« » _____ 2024р.

Захищений з оцінкою

(оцінка)

Члени комісії:

(особистий підпис)

(особистий підпис)

Виконавець

ст. Лядський Д.С.

залікова книжка № ____ – ____

гр. ІА-23

(особистий підпис виконавця)

« » _____ 2024р.

(розшифровка підпису)

(розшифровка підпису)

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
(назва навчального закладу)

Кафедра ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ
Дисципліна «Технології розроблення програмного забезпечення»
Курс 3 Група ІА-23 Семестр 5

ЗАВДАННЯ

на курсову роботу студента

Лядський Дмитро Сергійович
(прізвище, ім'я, по батькові)

1.Тема роботи Е-mail клієнт (singleton, builder, decorator, template method, interpreter, SOA) Поштовий клієнт повинен нагадувати функціонал поштових програм Mozilla Thunderbird, The Bat і т.д. Він повинен сприймати і коректно обробляти pop3/smtp/imap протоколи, мати функції автонастройки основних поштових провайдерів для України (gmail, ukr.net, i.ua), розділяти повідомлення на папки/категорії/важливість, зберігати чернетки незавершених повідомлень, прикріплювати і обробляти прикріплені файли.

2. Строк здачі студентом закінченої роботи 01.01.2025

3. Вихідні дані до роботи:

POP3/SMTP/IMAP протоколи для обробки поштових повідомлень.

API для автоналаштування поштових сервісів (gmail, ukr.net, i.ua).

Поштові клієнти (наприклад, Mozilla Thunderbird, The Bat) для прикладу функціоналу.

4. Зміст розрахунково – пояснювальної записки (перелік питань, що підлягають розробці)

Огляд існуючих рішень для е-mail клієнтів. Аналіз протоколів POP3/SMTP/IMAP та їх реалізації.

Автоналаштування поштових сервісів. Організація чернеток та обробка прикріплених файлів.

Застосування шаблонів проєктування (singleton, builder, decorator, template method, interpreter).

Додаткова діагностика та тестування застосунку.

Додатки:

Додаток А - Вихідний код

5.Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Дата видачі завдання 17.09.2024

КАЛЕНДАРНИЙ ПЛАН

№, п/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Підписи або примітки
1.	Видача завдання	17.09.2024	
2.	Аналіз існуючих рішень та формування технічного завдання	30.09.2024	
3.	Огляд та вивчення протоколів POP3/SMTP/IMAP	05.10.2024	
4.	Проектування архітектури системи	25.10.2024	
5.	Розробка модулів автоналаштування поштових провайдерів	08.11.2024	
6.	Реалізація функціоналу роботи з папками та чернетками	30.11.2024	
7.	Тестування реалізованих компонентів	17.12.2024	
8.	Оформлення пояснювальної записки	29.12.2024	
9.	Представлення роботи до захисту	01.01.2025	
10.			
11.			
12.			
13.			
14.			
15.			
16.			
17.			
18.			

Студент _____
(підпис)

Лядський Дмитро _____
(Ім'я ПРІЗВИЩЕ)

Керівник _____
(підпис)

Олександр АМОНС _____
(Ім'я ПРІЗВИЩЕ)

«___» _____ 20__ р.

ЗМІСТ

ВСТУП	4
1 ПРОЄКТУВАННЯ СИСТЕМИ.....	6
1.1. Огляд існуючих рішень	6
1.2. Загальний опис проєкту.....	7
1.3. Вимоги до застосунку.....	7
1.3.1. Функціональні вимоги до системи.....	9
1.3.2. Нефункціональні вимоги до системи.....	9
1.4. Сценарії використання системи	10
1.5. Концептуальна модель системи	17
1.6. Вибір бази даних	19
1.7. Вибір мови програмування та середовища розробки.....	20
1.8. Проєктування фізичної структури системи.....	21
2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ	26
2.1. Структура бази даних	26
2.2. Архітектура системи.....	27
2.2.1. Специфікація системи	28
2.2.2. Вибір та обґрунтування патернів реалізації.....	29
2.2.2.1 Вирішення проблеми збереження авторизованого користувача	29
2.2.2.2 Реалізація механізму створення об'єктів електронних адрес	31
2.2.2.3 Алгоритм авторизації для вхідних та вихідних серверів.....	34
2.2.2.4 Сортування повідомлень	36
2.3. Інструкція користувача.....	39
ВИСНОВКИ.....	49
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	51

ДОДАТКИ..... 52

Додаток А..... 52

ВСТУП

У сучасному світі електронна пошта є одним із ключових засобів комунікації, як у професійній, так і в особистій сфері. З розвитком інформаційних технологій постала необхідність створення ефективних інструментів для роботи з електронною поштою. Саме поштові клієнти, такі як Mozilla Thunderbird або The Bat, надають користувачам можливість комфортно обробляти вхідні та вихідні повідомлення, працювати з прикріпленими файлами та організовувати інформацію у зручному вигляді. Враховуючи динамічний розвиток ринку інформаційних технологій та зростання обсягів електронних комунікацій, розробка сучасного поштового клієнта є актуальним завданням.

З огляду на розмаїття поштових провайдерів і протоколів (POP3, SMTP, IMAP), користувачі часто стикаються з труднощами налаштування поштових клієнтів. Особливо це актуально для українських користувачів, які використовують популярні сервіси Gmail, ukr.net та i.ua. Автоматизація налаштування та забезпечення інтуїтивно зрозумілого інтерфейсу є необхідними умовами для підвищення ефективності роботи. Крім того, функціонал поділу повідомлень на папки є важливим для організації великих обсягів інформації. Зважаючи на це, розробка поштового клієнта є не лише актуальною, але й перспективною задачею для галузі програмного забезпечення.

Метою роботи є створення функціонального поштового клієнта, який забезпечуватиме комфортну взаємодію користувача з електронною поштою через підтримку протоколів POP3, SMTP, IMAP, автоматичне налаштування популярних українських поштових сервісів, а також можливості організації повідомлень за папками.

Для досягнення цієї мети необхідно вирішити такі завдання:

- розробити модуль автоматичної конфігурації для поштових провайдерів;
- реалізувати підтримку роботи з протоколами POP3, SMTP та IMAP;
- забезпечити функціонал для управління папками;

- реалізувати можливість збереження чернеток і прикріплення файлів;

Об'єктом дослідження є процес взаємодії користувачів із електронною поштою через програмні засоби. Предметом дослідження є методи та інструменти для розробки поштового клієнта з використанням сучасних програмних підходів і технологій.

Розроблений поштовий клієнт може бути використаний як основа для створення комерційних або корпоративних програмних продуктів. Додатково, автоматизація налаштувань для українських поштових провайдерів зробить продукт особливо корисним для локального ринку, забезпечуючи конкурентоспроможність серед аналогічних рішень.

1 ПРОЄКТУВАННЯ СИСТЕМИ

1.1. Огляд існуючих рішень

На сьогоднішній день існує чимало поштових клієнтів, які забезпечують широкий спектр функціональних можливостей. Найпопулярнішими серед них є Mozilla Thunderbird, The Bat, Microsoft Outlook, а також веб-інтерфейси таких сервісів, як Gmail, Yahoo Mail та інші. Кожен із цих клієнтів має свої особливості:

Mozilla Thunderbird — безкоштовний поштовий клієнт з відкритим вихідним кодом, який підтримує розширення, інтеграцію з календарями та інші функції. Його основними перевагами є підтримка кількох облікових записів і висока гнучкість налаштувань.

The Bat — платний поштовий клієнт, орієнтований на підвищену безпеку даних і багатофункціональність. Він пропонує можливості шифрування, резервного копіювання та широкі можливості для організації пошти.

Microsoft Outlook — компонент пакету Microsoft Office, який окрім роботи з електронною поштою, забезпечує функціонал для управління календарем, контактами та завданнями. Відрізняється інтеграцією з корпоративними рішеннями та сервісами Microsoft.

Веб-інтерфейси поштових сервісів — такі як Gmail, пропонують зручний інтерфейс та автоматичну синхронізацію, але залежать від постійного доступу до Інтернету.

Попри широке різноманіття існуючих рішень, більшість із них має недоліки, зокрема складність налаштування для локальних поштових провайдерів, недостатню підтримку специфічних функцій організації пошти або обмежену інтеграцію з іншими системами. Це створює передумови для розробки нового продукту, який враховуватиме потреби українського ринку та сучасні тенденції у програмній інженерії.

1.2. Загальний опис проєкту

Запропонований поштовий клієнт розробляється з метою забезпечення користувачів інтуїтивно зрозумілим, функціональним і надійним інструментом для роботи з електронною поштою. Основні особливості проєкту:

Функціонал:

- Підтримка основних протоколів для роботи з поштою (POP3, SMTP, IMAP).
- Автоматична конфігурація популярних поштових провайдерів України (Gmail, ukr.net, i.ua).
- Організація повідомлень за папками.
- Можливість збереження чернеток і прикріплення файлів.

Технології:

- Використання бази даних для збереження даних, зокрема чернеток, контактів та організованих повідомлень.
- Інтеграція з API поштових сервісів для автоматичної конфігурації та взаємодії.

Інтерфейс: Інтуїтивно зрозумілий користувацький інтерфейс.

Цей проєкт спрямований на вирішення існуючих проблем користувачів, пов'язаних із складністю налаштування поштових клієнтів, недостатньою організацією пошти та обмеженістю функціоналу для роботи з локальними провайдерами. Завдяки сучасним підходам до розробки програмного забезпечення, проєкт забезпечить надійну та зручну роботу з електронною поштою для широкого кола користувачів.

1.3. Вимоги до застосунку

Визначення вимог до застосунку є ключовим етапом у процесі розробки програмного забезпечення. Вимоги дозволяють чітко сформулювати, які функції та властивості повинна мати система, а також встановлюють критерії її відповідності очікуванням користувачів і технічним умовам.

Для поштового клієнта вимоги формуються на основі аналізу аналогічних програмних продуктів і потреб користувачів. Основні вимоги поділяються на функціональні та нефункціональні.

На рисунку 1.3.1 представлено діаграму варіантів використання (Use Case Diagram), яка ілюструє ключові сценарії взаємодії користувача із системою та основні функціональні можливості поштового клієнта.

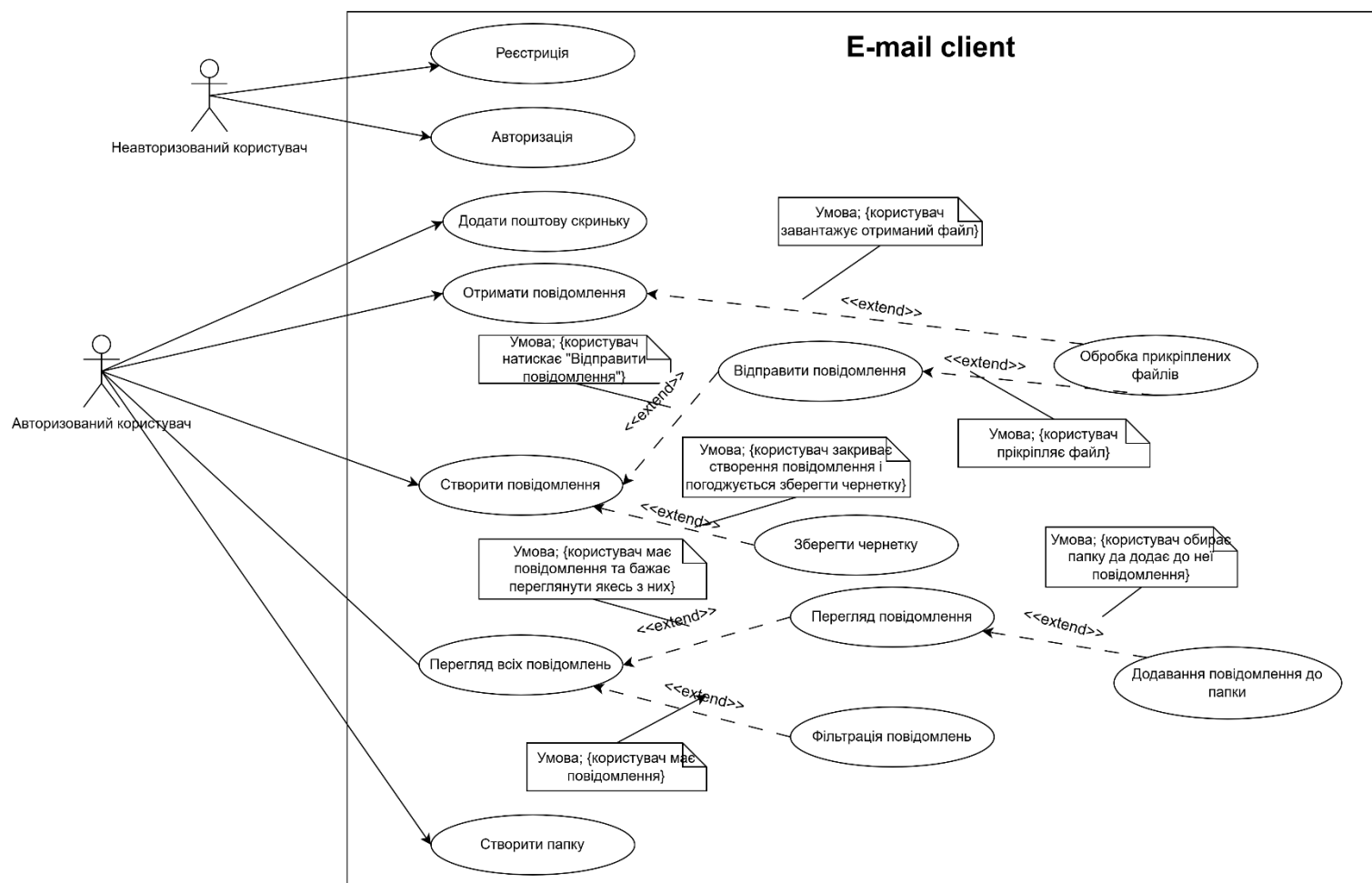


Рисунок 1.3.1. UseCase діаграма

UseCase діаграма ілюструє функціональні можливості e-mail клієнта. Основний актор — користувач, який може виконувати такі дії:

- Налаштувати поштову скриньку для роботи з POP3/SMTP/IMAP.
- Отримувати та переглядати повідомлення, застосовуючи фільтри.
- Створювати повідомлення з можливістю їх відправлення, збереження чернеток і прикріплення файлів.

- Організовувати повідомлення через додавання їх до папок або створення нових папок.

- Обробляти вкладення, зберігаючи чи відкриваючи файли.

Діаграма показує зв'язки між функціями, а також розширення для додаткових сценаріїв, як-от фільтрація, робота з файлами чи збереження чернеток.

1.3.1. Функціональні вимоги до системи

Обробка поштових протоколів: Застосунок повинен підтримувати роботу з POP3, SMTP та IMAP протоколами для прийому, відправлення та синхронізації електронної пошти.

Автонастройка поштових провайдерів: Система повинна автоматично визначати та налаштовувати параметри для популярних поштових провайдерів України (Gmail, Ukr.net, i.ua).

Розподіл повідомлень: Повідомлення повинні бути автоматично або вручну розподілені по папках.

Збереження чернеток: Невідправлені повідомлення повинні автоматично зберігатися як чернетки для подальшого редагування та надсилання.

Робота з вкладеннями: Користувач повинен мати можливість прикріплювати файли до листів.

Шаблони листів: Система повинна підтримувати використання шаблонів для пришвидшення створення листів.

Пошук: Застосунок повинен забезпечувати функцію пошуку повідомлень за ключовими словами, відправником, датою або іншими параметрами.

Багатокористувацький режим: Можливість додавання та використання кількох поштових облікових записів у межах одного клієнта.

1.3.2. Нефункціональні вимоги до системи

Зручність інтерфейсу: Інтерфейс застосунку повинен бути інтуїтивно зрозумілим, зручним для використання.

Сумісність: Застосунок повинен коректно працювати на основних операційних системах (Windows, macOS, Linux).

Масштабованість: Архітектура системи має дозволяти додавання нових функціональних можливостей (наприклад, інтеграція з календарями або сервісами для відеодзвінків).

Ресурсоефективність: Система повинна оптимально використовувати ресурси пристрою, забезпечуючи стабільну роботу навіть на пристроях з обмеженими характеристиками.

1.4. Сценарії використання системи

Сценарій 1: Реєстрація

1. Передумови: Користувач повинен мати доступ до інтерфейсу клієнта.
2. Постумови: Обліковий запис користувача створений, його дані збережені в системі.

3. Взаємодіючі сторони: Неавторизований користувач, E-mail клієнт.

4. Короткий опис: Користувач вводить необхідні дані для реєстрації, які зберігаються в клієнті.

5. Основний хід подій:

Користувач запускає програму.

Обирає опцію "Реєстрація".

Вводить дані в поля: "username", Номер телефону, пароль.

Натискає кнопку "Зареєструватися".

Система підтверджує успішну реєстрацію.

6. Виключення:

Введені некоректні дані.

Користувач вже зареєстрований.

Сценарій 2: Авторизація

1. Передумови: Користувач має існуючий обліковий запис.

2. Постумови: Користувач успішно увійшов у систему.

3. Взаємодіючі сторони: Неавторизований користувач, E-mail клієнт.

4. Короткий опис: Користувач вводить email і пароль для входу.

5. Основний хід подій:

Користувач запускає програму.

Обирає опцію "Вхід".

Вводить email і пароль.

Натискає кнопку "Увійти".

Система перевіряє дані.

Користувач отримує доступ до свого акаунту.

6. Виключення:

Невірні email або пароль.

Сервер авторизації недоступний.

Сценарій 3: Додавання поштової скриньки

1. Передумови: Користувач успішно авторизувався.

2. Постумови: Обліковий запис поштової скриньки доданий до системи.

3. Взаємодіючі сторони: Авторизований користувач, E-mail клієнт, Поштовий сервер (Gmail, Ukr.net, i.ua тощо).

4. Короткий опис: Користувач додає обліковий запис пошти, налаштування якого автоматично завантажуються.

5. Основний хід подій:

Користувач обирає опцію "Додати поштову скриньку".

Вводить email і пароль.

Програма автоматично визначає провайдера (наприклад, Gmail).

Виконує підключення через pop3/smtp/imap протокол.

Підключення успішне, поштовий обліковий запис доданий.

6. Виключення:

Невірні дані (email або пароль).

Сервер пошти недоступний.

Помилки протоколу (наприклад, невірні налаштування для IMAP).

Сценарій 4: Отримання повідомлень

1. Передумови: Користувач успішно авторизувався та додав поштову скриньку.

2. Постумови: Нові повідомлення отримані та збережені локально.

3. Взаємодіючі сторони: Авторизований користувач, E-mail клієнт, Поштовий сервер.

4. Короткий опис: Програма синхронізує повідомлення з поштового сервера та відображає їх у клієнті.

5. Основний хід подій:

Користувач обирає опцію "Оновити" або автоматичне оновлення запускається згідно налаштувань.

Клієнт підключається до поштового сервера через IMAP або POP3.

Отримує нові повідомлення.

Зберігає повідомлення локально.

Відображає їх.

6. Виключення:

Сервер недоступний.

Проблеми з мережею.

Сценарій 5: Створення повідомлення

1. Передумови: Користувач успішно авторизувався.

2. Постумови: Створено нове повідомлення, готове до відправки або збереження як чернетки.

3. Взаємодіючі сторони: Авторизований користувач, E-mail клієнт.

4. Короткий опис: Користувач створює новий email, додає текст, одержувача та за необхідності файли.

5. Основний хід подій реалізовано на діаграмі послідовностей на рисунку 1.4.1

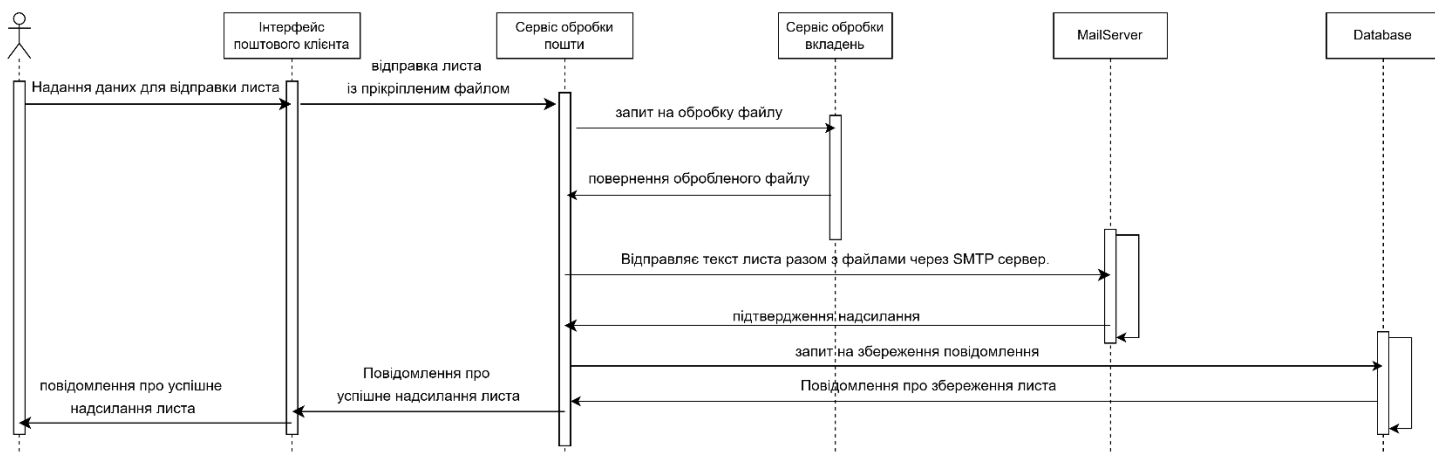


Рисунок 2.4.1. Діаграма послідовностей

Ця діаграма послідовностей демонструє процес взаємодії компонентів поштового клієнта під час відправлення електронного листа. Опис ключових етапів та ролей учасників:

Інтерфейс поштового клієнта

- Користувач ініціює створення листа через інтерфейс. Після завершення редагування натискає кнопку "Відправити".
- Інтерфейс передає запит на Сервіс обробки пошти для подальшої обробки.

Сервіс обробки пошти

- Приймає запит від інтерфейсу, аналізує структуру повідомлення.
- Передає запит до Сервісу обробки вкладень для перевірки та обробки файлів, прикріплених до повідомлення.

Сервіс обробки вкладень

- Перевіряє всі прикріплені файли та повідомляє Сервіс обробки пошти про результати.
- Після успішної перевірки повертає управління.

MailServer

- Сервіс обробки пошти передає сформоване повідомлення на MailServer.
- Сервер здійснює авторизацію, перевіряє можливість доставки листа та відправляє його адресату.

Database

– Після успішного відправлення, Сервіс обробки пошти зберігає інформацію про повідомлення в базу даних (наприклад, для відображення у папці "Надіслані").

Сценарій 6: Відправлення повідомлення

1. Передумови: Користувач створив нове повідомлення або обрав чернетку для відправки.

2. Постумови: Повідомлення успішно відправлене.

3. Взаємодіючі сторони: Авторизований користувач, E-mail клієнт, Поштовий сервер.

4. Короткий опис: Користувач відправляє повідомлення через SMTP-протокол.

5. Основний хід подій:

Користувач натискає кнопку "Відправити".

Клієнт перевіряє наявність усіх даних (одержувач, тема, текст).

Підключається до поштового сервера через SMTP.

Відправляє повідомлення.

Відображає статус успішної відправки.

6. Виключення:

Сервер недоступний.

Помилки підключення до SMTP.

Некоректні дані (наприклад, невірний email одержувача).

Сценарій 7: Обробка прикріплених файлів

1. Передумови:

Користувач авторизувався в поштовому клієнті.

Повідомлення містить прикріплений файл.

2. Постумови: Користувач завантажує прикріплений.

3. Взаємодіючі сторони: Авторизований користувач, E-mail клієнт.

4. Короткий опис: Функція дозволяє користувачу завантажувати прикріплені файли з повідомлення.

5. Основний хід подій:

Користувач відкриває отримане повідомлення.

Поштовий клієнт перевіряє наявність прикріплених файлів.

Користувач натискає на файл.

Файл завантажується або відкривається у вбудованому переглядачі.

6. Виключення:

Файл пошкоджений або недоступний на сервері.

Формат файлу не підтримується вбудованим переглядачем.

Сценарій 8: Збереження чернетки

1. Предумови: Користувач створив повідомлення, але не завершив його редагування.

2. Постумови: Повідомлення зберігається у папці "Чернетки".

3. Взаємодіючі сторони: Авторизований користувач, E-mail клієнт.

4. Короткий опис: Збереження незавершеного повідомлення для подальшого редагування.

5. Основний хід подій:

Користувач натискає кнопку "Зберегти чернетку" або закриває вікно редагування.

Клієнт зберігає текст повідомлення разом із вкладеннями та адресатами в локальній базі даних.

6. Виключення:

Недостатньо місця на диску для збереження чернетки.

Сценарій 9: Перегляд усіх повідомлень

1. Предумови: Користувач додав поштову скриньку.

2. Постумови: Усі повідомлення відображаються в інтерфейсі клієнта.

3. Взаємодіючі сторони: Авторизований користувач, E-mail клієнт, Поштовий сервер.

4. Короткий опис: Відображення списку повідомлень з можливістю сортування, фільтрації та пошуку.

5. Основний хід подій:

Користувач проходить авторизацію або натискає кнопку оновлення пошти.

Клієнт синхронізується із сервером пошти.

Відображається список повідомлень.

6. Виключення:

Немає доступу до сервера.

Пошта не синхронізована.

Сценарій 10: Фільтрація повідомлень

1. Предумови: Є список повідомлень.

2. Постумови: Відображаються лише повідомлення, що відповідають заданому критерію.

3. Взаємодіючі сторони: Авторизований користувач, E-mail клієнт.

4. Короткий опис: Фільтрація повідомлень за датою, адресатом, темою або іншими параметрами.

5. Основний хід подій:

Користувач вводить критерій фільтрації або вибирає з випадуючого списку.

Клієнт оновлює список повідомлень відповідно до вибраних критеріїв.

6. Виключення:

Користувач задає некоректний критерій (порожній або неіснуючий).

Сценарій 11: Додавання повідомлення до папки

1. Предумови: Користувач створив папку.

2. Постумови: Повідомлення додається до вибраної папки.

3. Взаємодіючі сторони: Авторизований користувач, E-mail клієнт.

4. Короткий опис: Переміщення повідомлення до певної папки для організації пошти.

5. Основний хід подій:

Користувач вибирає повідомлення зі списку.

Натискає кнопку "Перемістити до папки".

Вибирає папку зі списку.

Повідомлення додається у відповідну папку.

6. Виключення: Відсутні.

Сценарій 12: Створення папки

1. Предумови: Користувач авторизувався в клієнті.
2. Постумови: Нова папка створюється і відображається в списку папок.
3. Взаємодіючі сторони: Авторизований користувач, E-mail клієнт.
4. Короткий опис: Додавання нової папки для організації пошти.
5. Основний хід подій:

Користувач натискає кнопку "Створити папку".

Вводить назву нової папки.

Папка створюється та зберігається.

6. Виключення:

Назва папки вже використовується.

1.5. Концептуальна модель системи

Концептуальна модель системи є основою для розробки програмного забезпечення, оскільки вона забезпечує абстрактне уявлення про ключові елементи системи, їх властивості та взаємозв'язки. Ця модель допомагає розробникам зрозуміти функціональні вимоги, визначити основні об'єкти, які необхідно реалізувати, і закласти основу для подальшого проєктування.

Для поштового клієнта концептуальна модель описує логічну структуру системи, включаючи основні сутності, такі як облікові записи користувачів, повідомлення, папки та налаштування. Вона визначає, як ці об'єкти взаємодіють один з одним та з користувачем, а також відображає залежності між ними.

Розробка концептуальної моделі базується на аналізі функціональних вимог до системи, таких як підтримка протоколів POP3, SMTP, IMAP, організація повідомлень за категоріями, збереження чернеток і управління прикріпленими файлами. Використання сучасних підходів до моделювання дозволяє зробити систему гнучкою, масштабованою та готовою до розширення в майбутньому.

На рисунку 1.5.1 представлено діаграму концептуальної моделі, яка ілюструє основні об'єкти та їх взаємозв'язки, що лягають в основу реалізації функціоналу поштового клієнта.

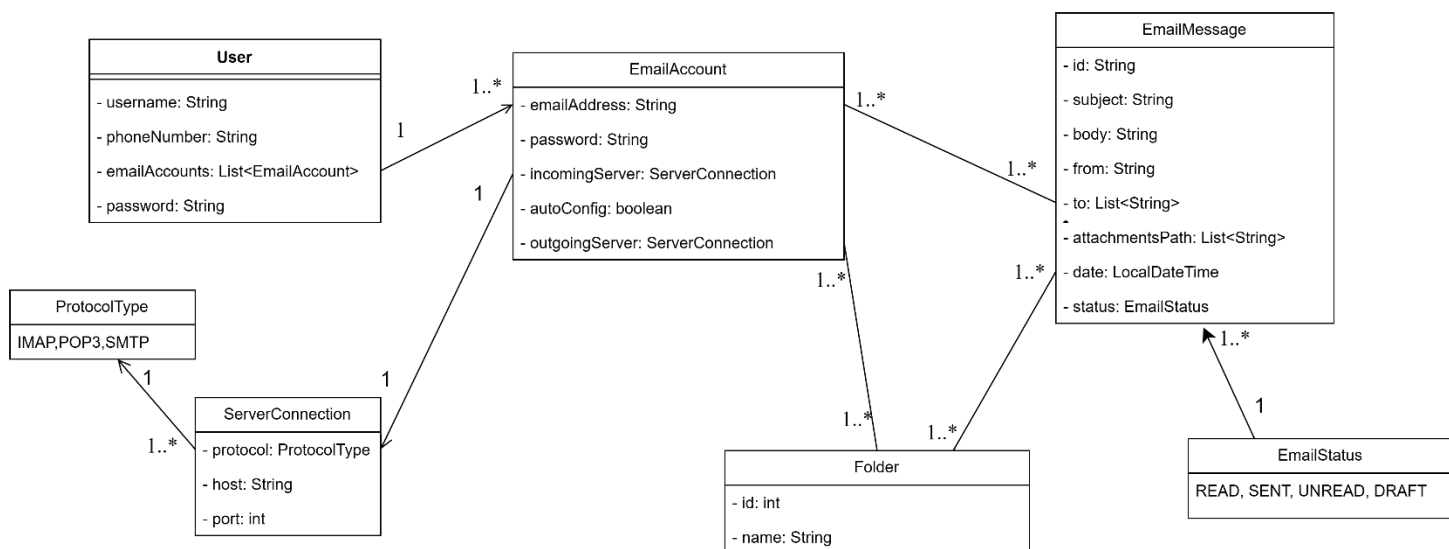


Рисунок 3.5.1. Спрощена діаграма класів

Ця діаграма класів представляє структуру поштового клієнта. Вона містить такі класи:

- **User** — зберігає інформацію про користувача, включаючи його ім'я, номер телефону, пароль та список облікових записів електронної пошти.
- **EmailAccount** — представляє обліковий запис електронної пошти, включаючи адреси серверів вхідної та вихідної пошти, протоколи (IMAP, POP3, SMTP), адресу електронної пошти, пароль та налаштування автоконфігурації.
- **ServerConnection** — описує серверне підключення із зазначенням протоколу, хоста та порту.
- **EmailMessage** — представляє електронне повідомлення з атрибутами, такими як тема, текст, адреси відправника та отримувачів, статус повідомлення (прочитане, чернетка тощо), дата та список вкладень.
- **Folder** — використовується для організації повідомлень у папках, має ідентифікатор і назву.
- **ProtocolType** — перераховує підтримувані протоколи (IMAP, POP3, SMTP).
- **EmailStatus** — перераховує статуси повідомлень (прочитане, надіслане, чернетка тощо).

Опис зв'язків між класами на діаграмі:

- Між User та EmailAccount: Користувач (User) може мати кілька облікових записів електронної пошти (EmailAccount). Зв'язок реалізований через список emailAccounts у класі User.
- Між EmailAccount та ServerConnection: Обліковий запис (EmailAccount) пов'язаний із серверними підключеннями для вхідного (incomingServer) і вихідного (outgoingServer) серверів. Сервери представлені класом ServerConnection.
- Між EmailAccount та ProtocolType: Обліковий запис використовує певний протокол (ProtocolType) для зв'язку (IMAP, POP3, SMTP).
- Між EmailAccount та EmailMessage: Обліковий запис може містити кілька електронних повідомлень (EmailMessage) і повідомлення може належати до кількох облікових записів.
- Між EmailMessage та EmailStatus: Повідомлення (EmailMessage) має статус (EmailStatus), наприклад, прочитане, чернетка, надіслане тощо.
- Між EmailMessage та Folder: Повідомлення може бути розміщене у певних папках (Folder), які організовують листи за категоріями.
- Між Folder та User: Користувач може мати доступ до кількох папок для організації своїх повідомлень.

Зв'язки показують ієрархічну структуру взаємодії між класами. Основна увага приділена взаємодії користувача з обліковими записами, організації повідомлень у папках та забезпеченню зв'язку через сервери із підтримкою різних протоколів.

1.6. Вибір бази даних

Для реалізації поштового клієнта було обрано базу даних PostgreSQL[1]. Це потужна реляційна база даних із відкритим вихідним кодом, яка забезпечує високу продуктивність, масштабованість та підтримує складні операції з даними. Вибір саме PostgreSQL обґрунтовано наступними причинами:

- PostgreSQL має багатий набір функцій для обробки складних запитів, що важливо для роботи з великими обсягами даних, такими як електронні повідомлення.

- PostgreSQL дозволяє ефективно зберігати та обробляти структуровані дані, такі як інформація про користувачів, облікові записи, папки, повідомлення та їх статуси.
- Система підтримує масштабування та оптимізована для роботи як із невеликими, так і великими обсягами даних, що робить її ідеальним вибором для поштового клієнта.
- PostgreSQL підтримує зберігання даних у форматі JSON, що дозволяє легко обробляти конфігурації облікових записів або зберігати метадані повідомлень у зручному форматі.
- PostgreSQL забезпечує високий рівень безпеки даних завдяки підтримці SSL-з'єднань, а також функціям автентифікації та контролю доступу.
- PostgreSQL легко інтегрується з сервісно-орієнтованою архітектурою (SOA), яка використовується в проєкті. Це спрощує обмін даними між модулями системи.
- PostgreSQL має широку спільноту розробників і багату документацію, що забезпечує швидке вирішення технічних проблем.

Таким чином, PostgreSQL є оптимальним вибором для розробки поштового клієнта, оскільки поєднує високу продуктивність, безпеку та гнучкість у роботі з даними.

1.7. Вибір мови програмування та середовища розробки

Для реалізації проєкту поштового клієнта було обрано мову програмування Java[2] та середовище розробки IntelliJ IDEA[3]. Цей вибір обґрунтовано наступними причинами:

Вибір мови програмування: Java

- Java є мовою програмування з підтримкою принципу "Write Once, Run Anywhere" (WORA), що дозволяє запускати додаток на будь-якій платформі, яка підтримує JVM (Java Virtual Machine).
- Java має розвинену екосистему, яка включає бібліотеки для роботи з мережею, обробки файлів, а також для інтеграції з протоколами POP3/SMTP/IMAP.

- Java забезпечує високий рівень безпеки, що є важливим для поштового клієнта, який працює з конфіденційними даними користувачів (паролі, повідомлення тощо).
- Завдяки вбудованій підтримці багатопотоковості Java дозволяє ефективно обробляти одночасні запити, наприклад, отримання пошти з різних серверів або надсилення повідомлень.
- Java є однією з найпопулярніших мов програмування, що забезпечує доступ до великої кількості ресурсів, документації та підтримки з боку спільноти.

Вибір середовища розробки: IntelliJ IDEA

- IntelliJ IDEA має інтуїтивно зрозумілий інтерфейс, що спрощує процес розробки та налагодження коду.
- IntelliJ IDEA пропонує потужні інструменти для написання, аналізу та оптимізації коду на Java, включаючи підтримку шаблонів проєктування, автоматичне завершення коду та інтеграцію з системами контролю версій (Git).
- IntelliJ IDEA дозволяє зручно працювати з базою даних PostgreSQL безпосередньо з середовища розробки, що пришвидшує тестування та налагодження.
- Середовище підтримує широкий набір плагінів, які можна використовувати для інтеграції з іншими інструментами та фреймворками, такими як Maven, Gradle, Spring тощо.
- IntelliJ IDEA має потужний відладчик, що дозволяє швидко знаходити та виправляти помилки в коді.

Отже, вибір Java як мови програмування та IntelliJ IDEA як середовища розробки забезпечує ефективну реалізацію функціоналу поштового клієнта, високу продуктивність, безпеку та масштабованість.

1.8. Проєктування фізичної структури системи

Проєктування фізичної структури системи є важливим етапом розробки програмного забезпечення, який визначає компоненти системи, їх фізичне

розташування та способи взаємодії між ними. Цей етап дозволяє забезпечити ефективну роботу системи, її масштабованість, надійність та безпеку.

Фізична структура системи включає сервери, клієнтські пристрої, бази даних, мережеве обладнання, а також інші елементи інфраструктури, необхідні для реалізації функціональних можливостей поштового клієнта. Основними аспектами проектування є розміщення компонентів системи, визначення взаємодії між ними через мережеві протоколи, а також забезпечення резервування даних і безперебійної роботи системи.

Для розробленого поштового клієнта було передбачено архітектуру, яка базується на принципах сервіс-орієнтованої архітектури (SOA)[4]. Це дозволяє окремим сервісам, таким як модуль обробки електронної пошти, модуль фільтрації, модуль керування папками, функціонувати незалежно, зберігаючи при цьому злагоджену взаємодію. У межах цієї структури розгортання включатиме серверну частину для обробки даних, клієнтську частину для взаємодії з користувачем, а також базу даних для збереження необхідної інформації.

На рисунку 1.8.1 представлено діаграму розгортання, яка ілюструє фізичну структуру системи, відображаючи основні компоненти та їх взаємозв'язки.

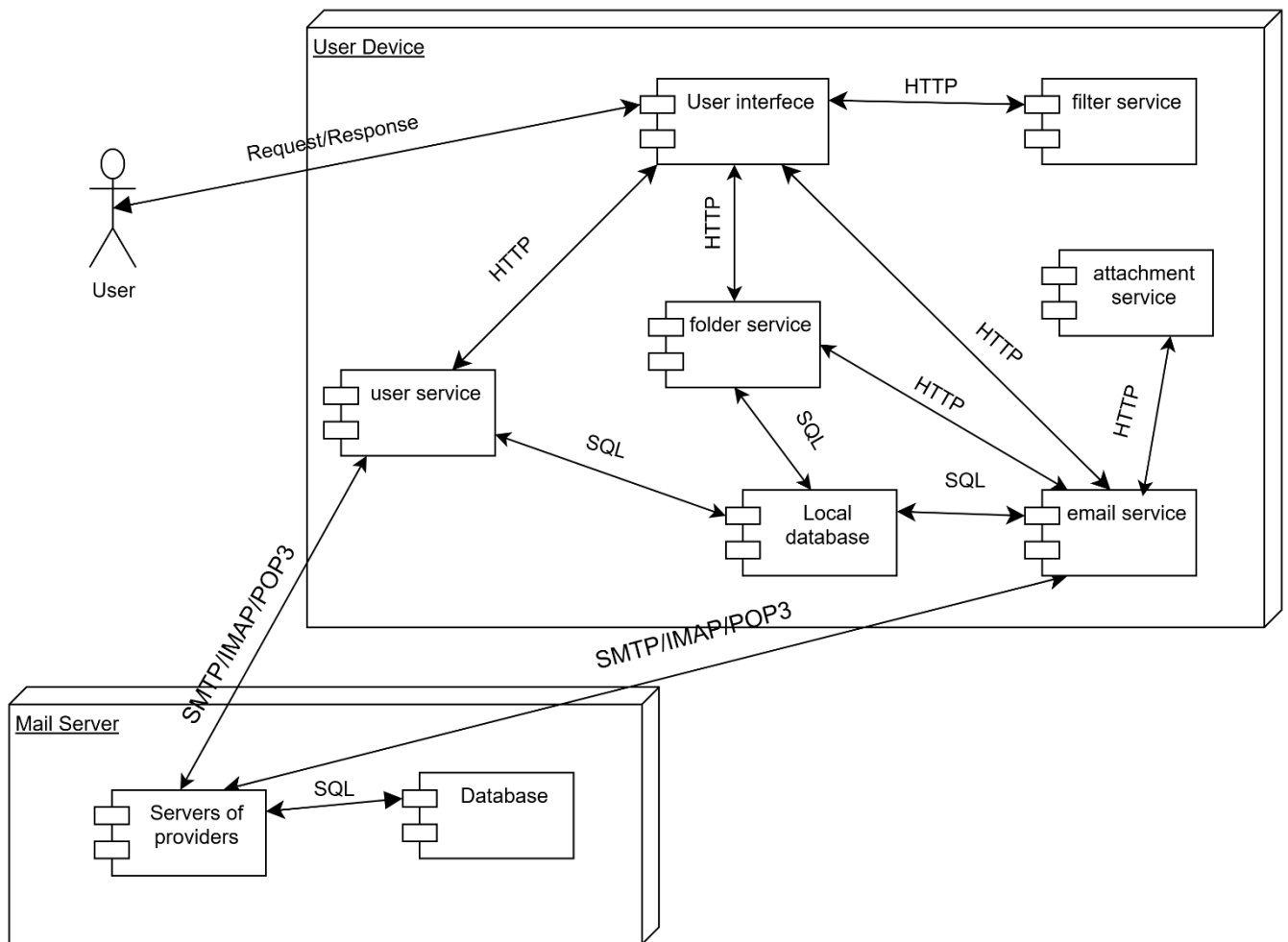


Рисунок 4.8.1. Діаграма розгортання

Діаграма розгортання зображає архітектуру поштового клієнта, що працює за моделлю Service-Oriented Architecture (SOA). Вона поділена на дві основні частини: User Device (Клієнтська частина) та Mail Server (Серверна частина).

Компоненти на стороні User Device:

1) User Interface (Інтерфейс користувача)

- Відповідає за взаємодію з користувачем.
- Спілкується з сервісами через HTTP.
- Виконує запити до сервісів: filter service, folder service, email service, user service.

2) User Service (Сервіс користувача)

- Обробляє дані користувача (наприклад, автентифікація, налаштування).

- Підключається до Local Database через SQL.

3) Folder Service (Сервіс папок)

- Відповідає за управління папками (вхідні, вихідні, чернетки, спам).
- Працює через HTTP з іншими сервісами та локальною базою даних.

4) Filter Service (Сервіс фільтрації)

- Застосовує фільтри до вхідних листів.
- Взаємодіє з User Interface через HTTP.

5) Attachment Service (Сервіс вкладень)

- Відповідає за обробку та збереження вкладень у листах.
- Підключений до Email Service через HTTP.

6) Email Service (Сервіс пошти)

- Керує відправкою, отриманням та обробкою електронних листів.
- Виконує запити до Local Database через SQL.
- Спілкується з поштовими серверами за допомогою

SMTP/POP3/IMAP.

7) Local Database (Локальна база даних)

- Містить дані про користувача, листи, налаштування та статуси.
- Обслуговує User Service, Folder Service, Email Service.

3. Компоненти на стороні Mail Server:

1) Servers of Providers (Сервери провайдерів)

- Відповідають за обробку SMTP/POP3/IMAP-запитів.
- Підключені до Database через SQL.

2) Database (База даних на сервері)

- Містить інформацію про користувачів та їхні листи.

Протоколи:

- SMTP – для відправки листів.
- POP3 – для отримання листів.

- IMAP – для отримання листів.
- SQL – для роботи з базами даних.

4. Взаємодія між компонентами:

Користувач взаємодіє з User Interface для надсилання чи отримання листів.

- User Interface надсилає запити до сервісів (Folder, Email, Filter, User).
- Дані зберігаються у Local Database.
- Email Service обмінюється даними з поштовим сервером через

SMTP/POP3/IMAP.

- Сервери провайдерів отримують або надсилають листи, використовуючи свою Database.

Ця діаграма чітко демонструє, як різні компоненти взаємодіють між собою для забезпечення функціональності поштового клієнта.

2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

Етап реалізації компонентів системи охоплює процес програмної реалізації основних функціональних модулів, передбачених концептуальною моделлю та вимогами до застосунку. Він передбачає детальний опис способів і підходів, використаних для створення окремих частин системи, їх інтеграцію та тестування.

Для поштового клієнта реалізація здійснювалася відповідно до принципів модульності, що забезпечує гнучкість і масштабованість системи. Кожен компонент реалізовано як окремий модуль із чітко визначеними функціями, що дозволяє легко додавати новий функціонал або змінювати існуючий без порушення роботи інших частин системи.

2.1. Структура бази даних

У процесі розробки поштового клієнта було реалізовано дві взаємопов'язані бази даних на основі системи управління базами даних PostgreSQL. Такий підхід забезпечує розподіл даних за логічними категоріями, що підвищує продуктивність і масштабованість системи. Структура першої бази даних зображена на рисунку 2.1.1.

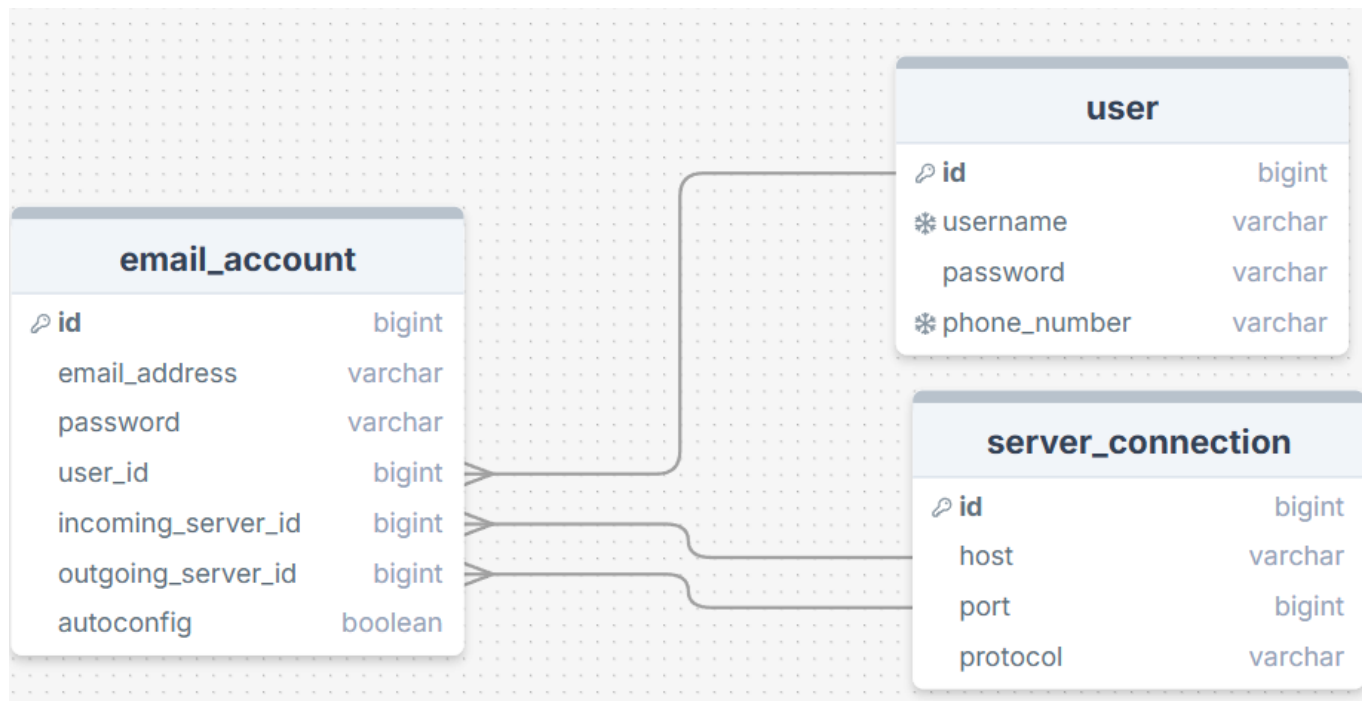


Рисунок 2.1.1. Структура першої бази даних

Перша база даних призначена для зберігання інформації про користувачів та їх поштові акаунти. Вона містить таблиці, які відображають дані про реєстрацію

користувачів, а також параметри налаштування облікових записів для доступу до поштових серверів (POP3/IMAP/SMTP).

Структура другої бази даних зображена на рисунку 2.1.2.

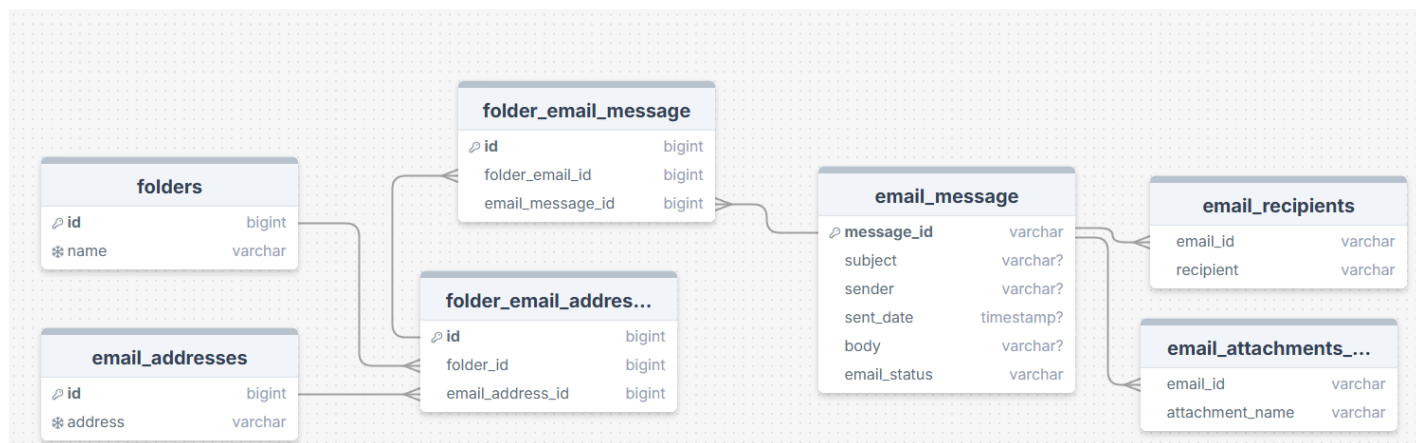


Рисунок 2.1.2. Структура другої бази даних

Друга база даних використовується для зберігання повідомлень, папок і поштових адрес. Її структура забезпечує можливість організації пошти за категоріями (папками), зберігання чернеток, а також підтримки функціоналу роботи з адресатами.

Такий поділ дозволяє не лише логічно впорядкувати дані, але й оптимізувати виконання запитів, розмежувати доступ до різних частин системи та полегшити процеси резервного копіювання й обслуговування.

2.2. Архітектура системи

Архітектура системи є фундаментальною частиною розробки програмного забезпечення, яка визначає її структуру, функціональність та взаємодію між компонентами. У даній курсовій роботі використовується модульний підхід, що дозволяє розділити систему на окремі частини, кожна з яких відповідає за конкретну функцію, наприклад, обробку поштових протоколів, роботу з інтерфейсом користувача або керування файлами. Це забезпечує гнучкість, масштабованість і можливість подальшого розширення функціоналу. У розробці архітектури також враховано принципи Service-Oriented Architecture (SOA), що дозволяє використовувати незалежні сервіси для обробки різних задач.

2.2.1. Специфікація системи

Передумови створення системи:

- Основна мета – створити поштовий клієнт із сучасним функціоналом, який відповідатиме потребам українських користувачів та забезпечуватиме зручний і безпечний обмін електронною поштою.
- Використання принципів об'єктно-орієнтованого програмування, патернів проєктування та архітектурних рішень на основі SOA (Service-Oriented Architecture) для забезпечення модульності та масштабованості.

Основні функції системи:

- Підтримка протоколів POP3, SMTP, IMAP для забезпечення прийому, надсилання та синхронізації електронних листів.
- Автоматична конфігурація популярних поштових сервісів (Gmail, Ukr.net, i.ua) для полегшення налаштувань.
- Управління повідомленнями: розподіл за категоріями, збереження чернеток.
- Робота з вкладеннями: прикріплення, зберігання.

Ключові вимоги до системи:

- Продуктивність: Забезпечення швидкої обробки повідомлень і мінімального часу затримки.
- Сумісність: Коректна робота на різних операційних системах (Windows, macOS, Linux).
- Масштабованість: Легка інтеграція нових функцій (наприклад, інтеграція з хмарними сховищами або календарями).
- Архітектурний підхід: Архітектура побудована за модульним принципом із використанням SOA для розділення компонентів.

Основні модулі:

- Модуль інтерфейсу користувача.
- Модуль обробки пошти.

- Модуль обробки поштових акаунтів.
- Модуль управління файлами.
- Модуль фільтрації повідомлень.
- Модуль управління повідомленнями.

Технології та інструменти:

- Мова програмування: Java.
- Бібліотеки: Для роботи з IMAP/SMTP/POP3 – jakarta.mail[5]; для GUI – JavaFX[6].
- Система управління базами даних: PostgreSQL для зберігання локальної інформації.

Очікувані результати:

- Система, що дозволяє користувачам легко налаштовувати поштові акаунти та ефективно управляти електронною поштою.
- Забезпечення стабільної роботи навіть у разі великої кількості облікових записів та повідомлень.
- Розширюваність для інтеграції з новими сервісами або функціями.

2.2.2. Вибір та обґрунтування патернів реалізації

Для розробки функціоналу поштового клієнта було обрано кілька шаблонів проектування, кожен із яких вирішує конкретну задачу в межах системи[7]. В наступних пунктах наведено обґрунтування вибору та опис використання кожного з них.

2.2.2.1 Вирішення проблеми збереження авторизованого користувача

Проблема:

У системі поштового клієнта необхідно забезпечити єдиний доступ до даних авторизованого користувача. Це включає інформацію про поточного користувача (наприклад, облікові дані) та пов'язані з ним електронні листи. Використання кількох об'єктів для зберігання цих даних може призвести до несинхронізованості стану або

дублювання ресурсів. Необхідно гарантувати, що в будь-який момент часу існує лише один екземпляр класу, який відповідає за збереження цих даних.

Для вирішення цієї проблеми було прийнято рішення скористатися шаблоном “Одинак” (“Singleton”).

“Одинак (англ. Singleton) — шаблон проєктування, належить до класу твірних шаблонів. Гарантує, що клас матиме тільки один екземпляр, і забезпечує глобальну точку доступу до цього екземпляра. Рішення полягає в тому, щоб сам клас контролював свою «унікальність», забороняючи створення нових екземплярів, та сам забезпечував єдину точку доступу. Це є призначенням шаблону Одинак.”[8]

На рисунку 2.2.2.1.1. зображено код класу `AuthenticatedUser`, який реалізовано за допомогою шаблону Одинак.



```
9      @Getter
10     @Setter
11     public class AuthenticatedUser {
12         3 usages
13         private static AuthenticatedUser instance;
14         private UserDto user;
15         private List<EmailMessageContextDto> emails;
16
17         1 usage
18         private AuthenticatedUser() {
19             }
20
21         5 usages
22         public static AuthenticatedUser getInstance() {
23             if (instance == null) {
24                 instance = new AuthenticatedUser();
25             }
26             return instance;
27         }
28     }
```

Рисунок 2.2.2.1.1. Код класу `AuthenticatedUser`

Патерн "Одинак" реалізовано через статичну змінну `instance`, яка зберігає єдиний екземпляр класу. Конструктор класу `AuthenticatedUser` зроблено приватним, щоб уникнути створення об'єктів за межами класу. Метод `getInstance()` гарантує створення екземпляра лише за необхідності (лінива ініціалізація).

Клас `AuthenticatedUser` містить інформацію про авторизованого користувача у вигляді об'єкта `UserDto` та список його електронних листів `emails`.

Цей підхід забезпечує централізований доступ до даних користувача, запобігаючи дублюванню стану та полегшуючи підтримку цілісності інформації в системі.

2.2.2.2 Реалізація механізму створення об'єктів електронних адрес

Проблема:

У поштовому клієнті необхідно реалізувати механізм створення об'єктів для додавання нових електронних адрес користувачів. Оскільки електронна пошта включає кілька параметрів (адресу, пароль, налаштування вхідного та вихідного серверів, автоконфігурацію), процес створення об'єкта може бути складним і потребує гнучкості. Також важливо забезпечити можливість автоматичної конфігурації серверів для популярних доменів (наприклад, Gmail, Ukr.net, i.ua).

Рішення:

Для вирішення цієї проблеми було прийнято рішення скористатися патерном "Будівельник" ("Builder"), який дозволяє поступово створювати об'єкт і забезпечує гнучкість у налаштуванні його параметрів.

"Будівельник (англ. Builder) — шаблон проєктування, належить до класу твірних шаблонів. Замість того, щоб використовувати набір конструкторів, використовують шаблон будівельник, згідно з яким використовують інший об'єкт (будівельник), що отримує на вхід вхідні параметри по одному, крок за кроком, і повертає за один прохід результуючий створений об'єкт." [9]

На рисунках 2.2.2.2.1-2.2.2.2.3 зображено основні частини коду класа `EmailAccount`, який реалізовано за допомогою шаблону Будівельник.

```

4
5 @Getter
6 public class EmailAccount {
7     1 usage
8     private String emailAddress;
9     1 usage
10    private String password;
11    1 usage
12    private ServerConnection incomingServer;
13    1 usage
14    private ServerConnection outgoingServer;
15    1 usage
16    private Boolean autoconfig;
17    1 usage
18    private EmailAccount(EmailAccountBuilder emailAccountBuilder) {
19        this.emailAddress = emailAccountBuilder.emailAddress;
20        this.password = emailAccountBuilder.password;
21        this.incomingServer = emailAccountBuilder.incomingServer;
22        this.outgoingServer = emailAccountBuilder.outgoingServer;
23        this.autoconfig = emailAccountBuilder.autoconfig;
24    }
25    7 usages
26    public static class EmailAccountBuilder {
27        3 usages
28        private String emailAddress;
29        2 usages
30        private String password;
31        5 usages
32        private ServerConnection incomingServer;
33        5 usages
34        private ServerConnection outgoingServer;

```

Рисунок 2.2.2.2.1 Частина коду класа EmailAccount

```

28 public EmailAccountBuilder(String emailAddress, String password) {
29     this.emailAddress = emailAddress;
30     this.password = password;
31 }
32 @
33 3 usages
34 public EmailAccountBuilder setAutoconfig(Boolean autoconfig){
35     if (autoconfig){
36         autoConfigure();
37     }
38     this.autoconfig = autoconfig;
39     return this;
40 }
41 1 usage
42 public EmailAccountBuilder setIncomingServer(ServerConnection serverConnection){
43     this.incomingServer = serverConnection;
44     return this;
45 }
46 1 usage
47 public EmailAccountBuilder setOutgoingServer(ServerConnection serverConnection){
48     this.outgoingServer = serverConnection;
49     return this;
50 }
51 1 usage
52 private void autoConfigure() {
53     String domain = this.emailAddress.split( regex: "@" )[1];

```

Рисунок 2.2.2.2.2. Частина коду класа EmailAccount

```

}
}

3 usages
public EmailAccount build() { return new EmailAccount( emailAccountBuilder: this); }
}

```

Рисунок 2.2.2.2.3. Частина коду класа EmailAccount

Клас EmailAccount використовується для зберігання інформації про електронну пошту, включаючи адресу, пароль, параметри серверів та статус автоконфігурації. Внутрішній клас EmailAccountBuilder забезпечує покрокове створення об'єкта EmailAccount.

- Конструктор `EmailAccountBuilder` приймає обов'язкові параметри (адресу електронної пошти та пароль).
 - Метод `setAutoconfig()` дозволяє автоматично налаштовувати сервери в залежності від домену електронної адреси.
 - Методи `setIncomingServer()` та `setOutgoingServer()` забезпечують ручне налаштування серверів.
 - Метод `build()` завершує створення об'єкта `EmailAccount` і повертає його.
- Цей підхід спрощує створення об'єктів, забезпечуючи гнучкість у налаштуванні параметрів та підтримку автоконфігурації для популярних доменів.

2.2.2.3 Алгоритм авторизації для вхідних та вихідних серверів

Проблема:

У поштовому клієнті потрібно реалізувати авторизацію для роботи з вхідними та вихідними серверами. Хоча загальний алгоритм авторизації однаковий, окремі кроки, такі як налаштування властивостей з'єднання та тестування підключення, залежать від типу сервера (вхідний чи вихідний). Це створює необхідність забезпечити повторне використання спільної логіки при збереженні можливості адаптації окремих етапів.

Рішення:

Для вирішення цієї проблеми було використано патерн “Шаблонний метод” (“Template Method”), який дозволяє визначити загальний алгоритм у базовому класі та делегувати реалізацію окремих етапів підкласам.

“Шаблонний метод (англ. Template Method) — шаблон проєктування, належить до класу шаблонів поведінки. Визначає кістяк алгоритму та дозволяє підкласам перевизначити деякі кроки алгоритму, не змінюючи структуру в цілому.”[10]

На рисунку 2.2.2.3.1 зображено діаграму класів, які реалізують Шаблонний метод для реалізації авторизації.

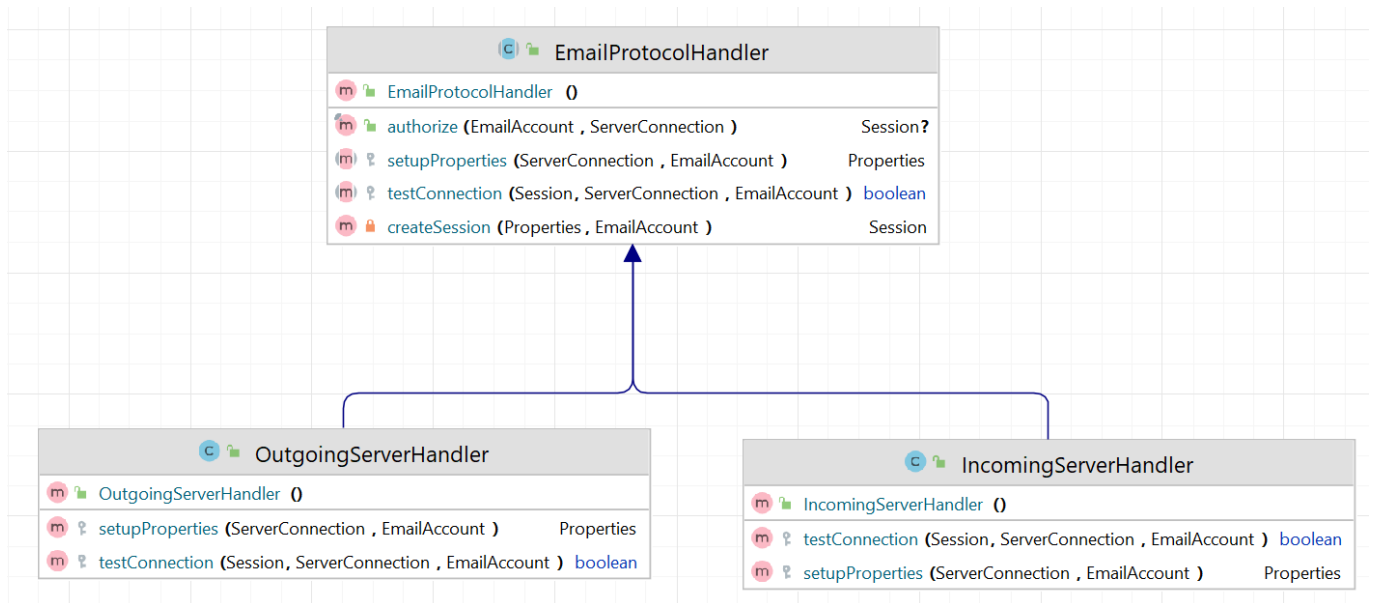


Рисунок 2.2.2.3.1. Діаграму класів шаблону Шаблонний метод

Базовий клас EmailProtocolHandler:

- Визначає загальний алгоритм авторизації через метод `authorize()`.
- Реалізація загальних етапів, таких як створення сесії (`createSession()`), зберігається в базовому класі.
- Абстрактні методи `setupProperties()` і `testConnection()` делегують реалізацію підкласам.

Підкласи IncomingServerHandler та OutgoingServerHandler:

- Реалізують специфічні кроки для вхідних (IMAP/POP3) та вихідних (SMTP) серверів.

Переваги використання:

- Спільна логіка зберігається в базовому класі, що зменшує дублювання коду.
- Підкласи забезпечують гнучкість у реалізації специфічних деталей.
- Алгоритм залишається зрозумілим і легко розширюваним.

2.2.2.4 Сортування повідомлень

Проблема:

У поштовому клієнті необхідно забезпечити можливість послідовного застосування різних фільтрів для сортування електронних листів (наприклад, за статусом, датою, відправником тощо). Створення статичних фільтрів для кожної можливої комбінації було б неефективним і призвело б до надлишкового коду.

Рішення:

Для вирішення цієї проблеми було використано патерн “Декоратор” (“Decorator”), який дозволяє динамічно додавати нову поведінку до об'єкта, зберігаючи інтерфейс базового класу.

“Декоратор (фр. *décorateur*) — структурний шаблон проєктування, призначений для динамічного підключення додаткових можливостей до об'єкта. Шаблон Decorator надає гнучку альтернативу методу визначення підкласів з метою розширення функціональності. Об'єкт, який передбачається використовувати, виконує основні функції. Проте може виникнути потреба додати до нього деяку додаткову функціональність, яка виконуватиметься до і/або після основної функціональності об'єкта.”[11]

На рисунку 2.2.2.4.1 зображено діаграму класів, які реалізують Декоратор для реалізації фільтрування.

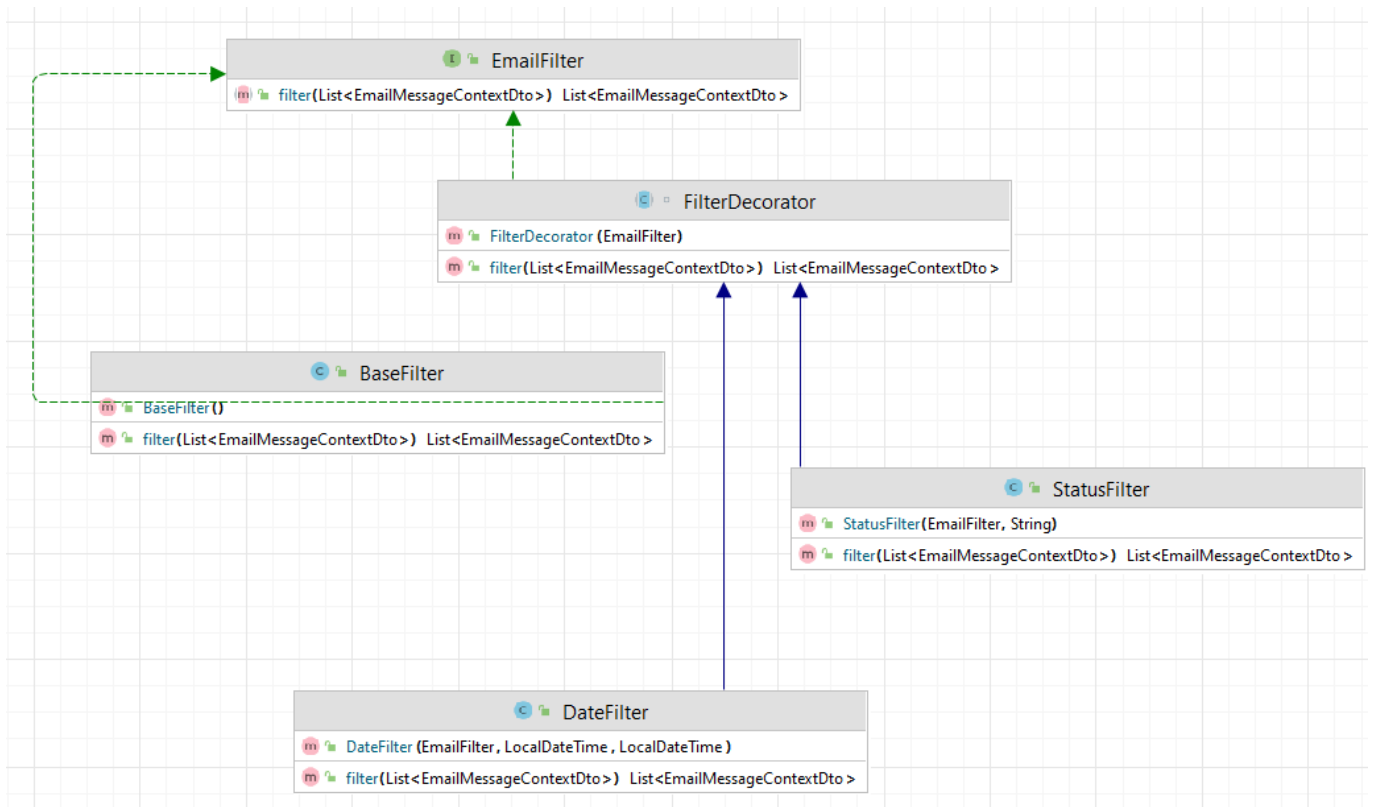


Рисунок 2.2.2.4.1. Діаграму класів шаблону Декоратор

Опис реалізації:

- Інтерфейс `EmailFilter` визначає метод `filter()`, який використовується для застосування фільтрів.
- Клас `FilterDecorator` є базовим класом-декоратором, який делегує виклик основного фільтру.
- Конкретні декоратори (`StatusFilter`, `DateFilter`) додають специфічну логіку для фільтрації.

Проблема:

Користувачі поштового клієнта можуть задавати складні критерії пошуку листів (наприклад, знайти всі листи з певним статусом і ключовим словом у темі). Необхідно реалізувати можливість комбінування умов пошуку.

Рішення:

Для вирішення цієї проблеми було використано патерн “Інтерпретатор” (“Interpreter”), який дозволяє описувати складні умови у вигляді деревоподібних структур, де кожен вузол відповідає за інтерпретацію певної частини умови.

“Інтерпретатор (англ. Interpreter) — шаблон проєктування, належить до класу шаблонів поведінки. Шаблон Інтерпретатор слід використовувати, коли є мова для інтерпретації, речення котрої можна подати у вигляді абстрактних синтаксичних дерев. Найкраще шаблон працює коли:

- граматика проста. Для складних граматик ієрархія класів стає занадто громіздкою та некерованою. У таких випадках краще застосовувати генератори синтаксичних аналізаторів, оскільки вони можуть інтерпретувати вирази, не будуючи абстрактних синтаксичних дерев, що заощаджує пам'ять, а можливо і час;

- ефективність не є головним критерієм. Найефективніші інтерпретатори зазвичай не працюють безпосередньо із деревами, а спочатку транслюють їх в іншу форму. Так, регулярний вираз часто перетворюють на скінченний автомат. Але навіть у цьому разі сам транслятор можна реалізувати за допомогою шаблону інтерпретатор.”[12]

На рисунку 2.2.2.4.2 зображено діаграму класів, які реалізують Інтерпретатор для реалізації фільтрування.

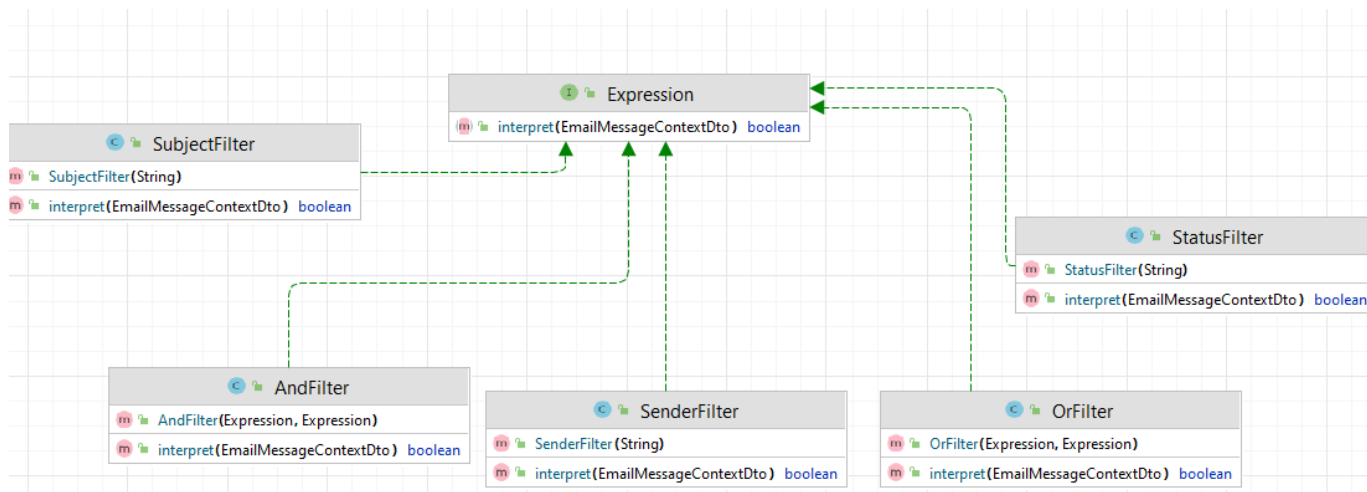


Рисунок 2.2.2.4.2. Діаграму класів шаблону Інтерпретатор

Опис реалізації:

- Інтерфейс Expression визначає метод interpret() для перевірки відповідності електронного листа заданій умові.
- Конкретні вирази (SenderFilter, SubjectFilter) реалізують базові умови.
- Комбінатори (OrFilter, AndFilter) дозволяють об'єднувати вирази у складні логічні структури.

Переваги використання:

- Патерн "Декоратор" забезпечує динамічне комбінування фільтрів без зміни існуючого коду.
- Патерн "Інтерпретатор" дозволяє гнучко комбінувати критерії пошуку, що підвищує зручність роботи користувачів із системою.

2.3. Інструкція користувача

Ця інструкція призначена для користувачів поштового клієнта, розробленого в межах даного проєкту. Вона містить покрокові вказівки для встановлення, налаштування та використання програми. Завдяки детальному опису основних функцій програми ви зможете ефективно працювати з електронною поштою, налаштовувати облікові записи та організовувати повідомлення.

Інструкція:

Для початку роботи потрібно завантажити папку з кодом поштового.

Щоб запустити застосунок потрібно створити 2 бази даних в Postgres з назвами email_service та user_service або іншими назвами, але назви потрібно вказати в файлах E-mail client\email-service\src\main\resources\application.yaml та E-mail client\user-service\src\main\resources\application.yaml також там потрібно вказати username та password та авторизації в Postgres.

Далі потрібно запустити 6 сервісів, які запускаються за допомогою наступних файлів:

user-service:	E-mail	client\user-service\src\main\java\org\example\
UserServiceMain.java		

folder-service:	E-mail	client\folder-service\src\main\java\org\example\
FolderServiceMain.java		
filter-service:	E-mail	client\filter-service\src\main\java\org\example\
FilterServiceMain.java		
email-service:	E-mail	client\email-service\src\main\java\org\example\
UserServiceMain.java		
attachment-service:	E-mail	client\attachment-service\src\main\java\org\example\
EmailServiceMain.java		
app-module:	E-mail	client\app-module\src\main\java\org\example\ AppModule.java

Початкова сторінка зображена на рисунку 2.3.1.

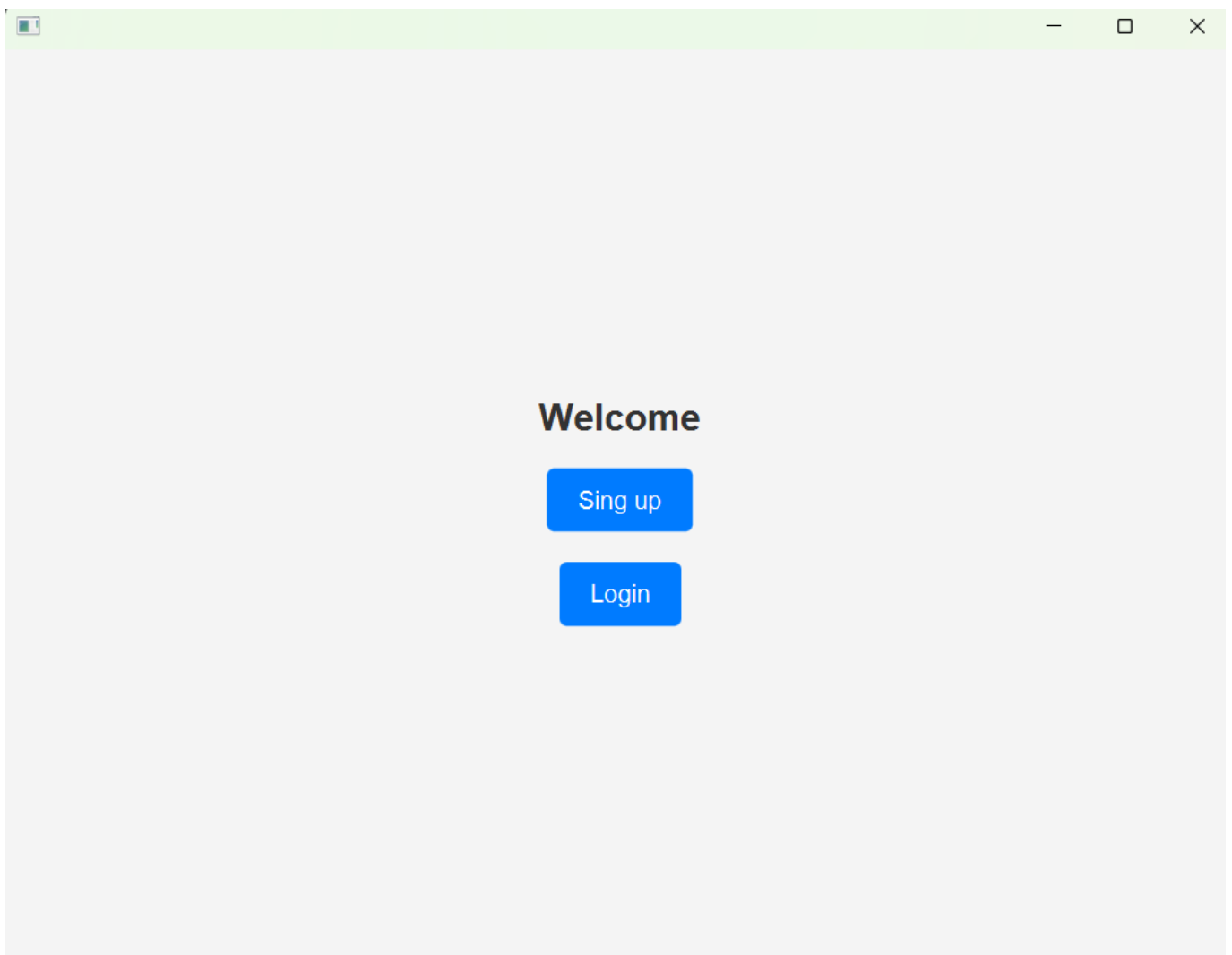
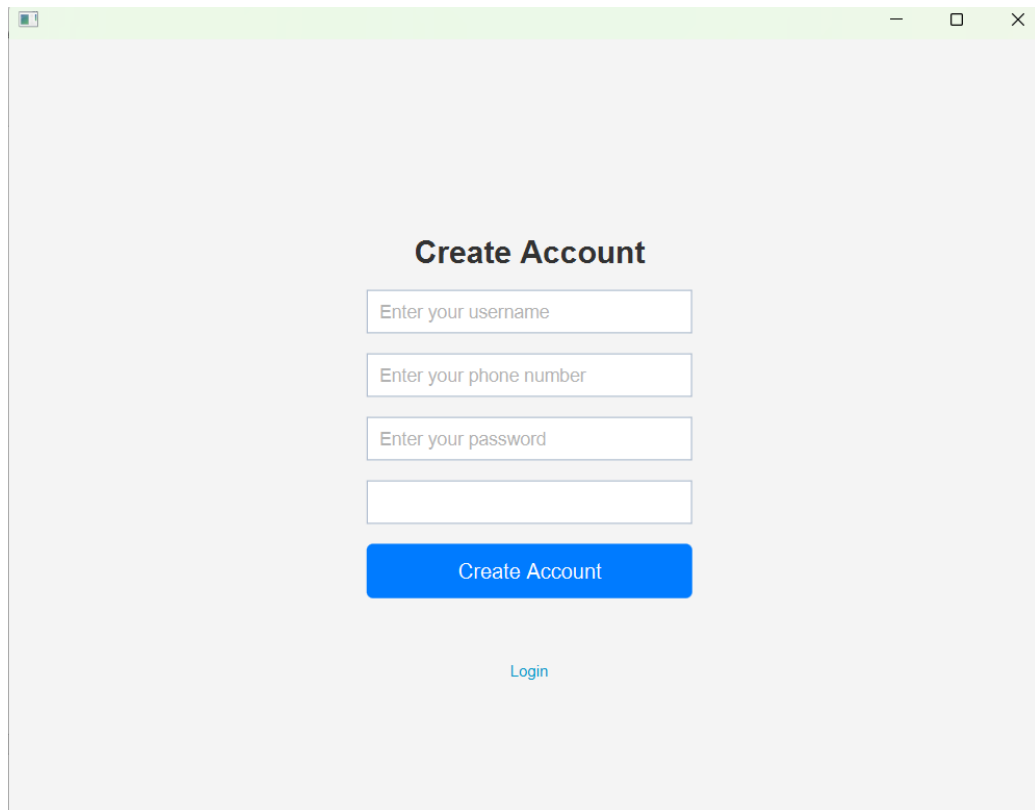


Рисунок 2.3.1. Початкова сторінка

Після запуску AppModule відкриється початкова сторінка додатку з якої можна перейти до реєстрації(рисунок 2.3.2) або авторизації(рисунок 2.3.3).



Create Account

Enter your username

Enter your phone number

Enter your password

Create Account

[Login](#)

Рисунок 2.3.2. Сторінка реєстрації

Щоб зареєструватися користувач повинен заповнити всі поля та натиснути кнопку “Create Account”.

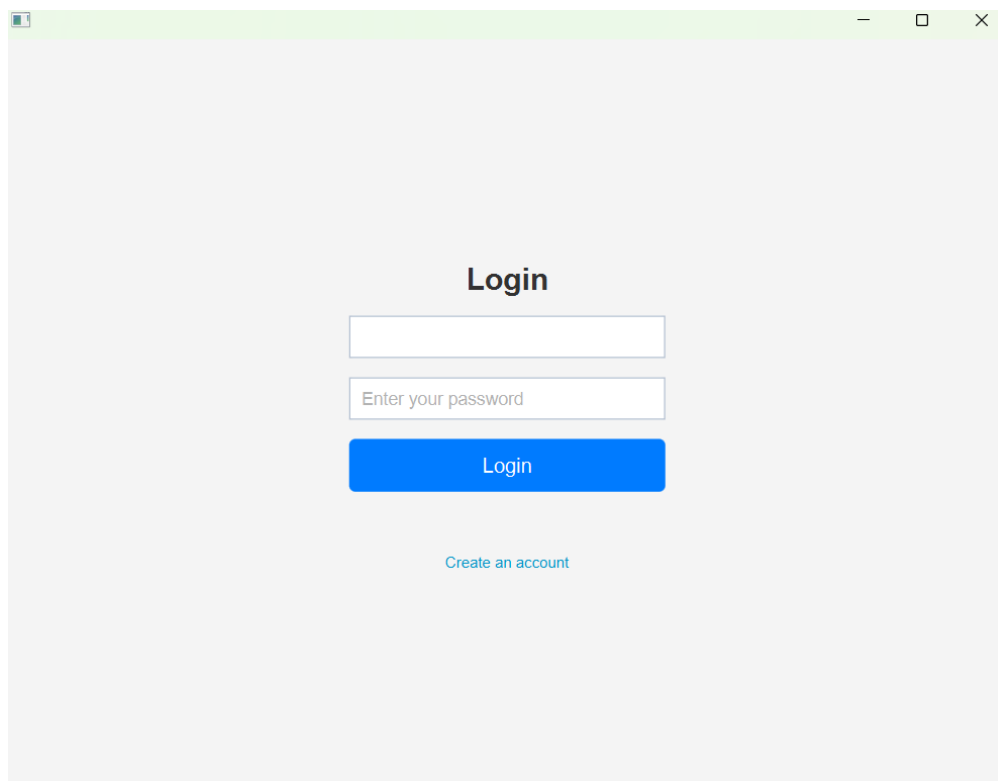
A screenshot of a web browser window displaying a login form. The window has a light green title bar with standard minimize, maximize, and close buttons. The background of the page is a light gray. In the center, the word "Login" is displayed in a bold, black font. Below it are two white input fields with thin gray borders. The first field is empty, and the second field contains the placeholder text "Enter your password". Below the input fields is a solid blue button with the word "Login" in white text. At the bottom of the form, there is a link that says "Create an account" in a small, blue font.

Рисунок 2.3.3. Сторінка авторизації

Щоб авторизуватися користувач повинен заповнити всі поля та натиснути кнопку “Login”.

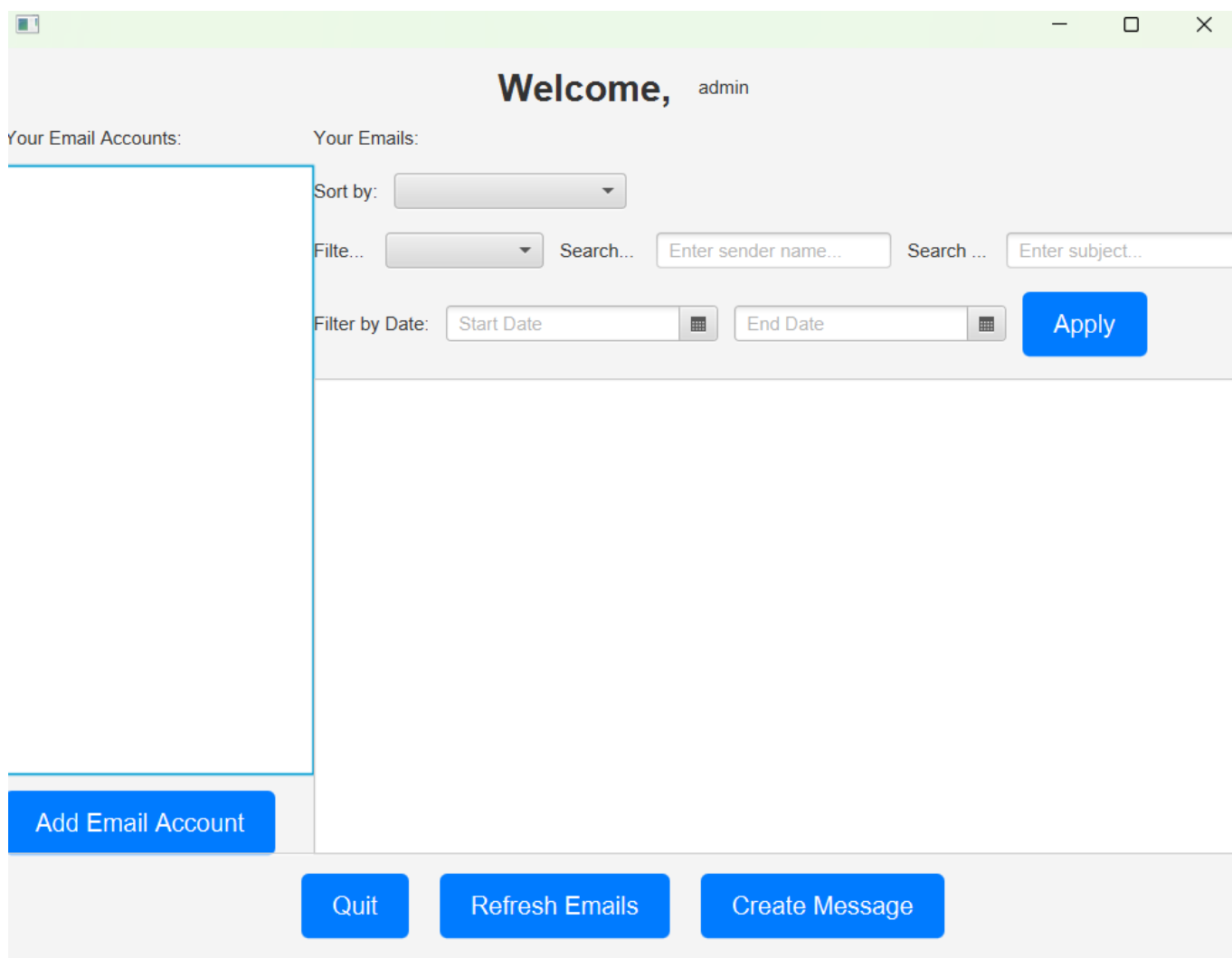


Рисунок 2.3.4. Головна сторінка

Після успішної реєстрації або авторизації користувач опиняється на головній сторінці застосунку. Щоб побачити якісь повідомлення потрібно додати обліковий запис електронної пошти за допомогою кнопки “Add Email Account”.

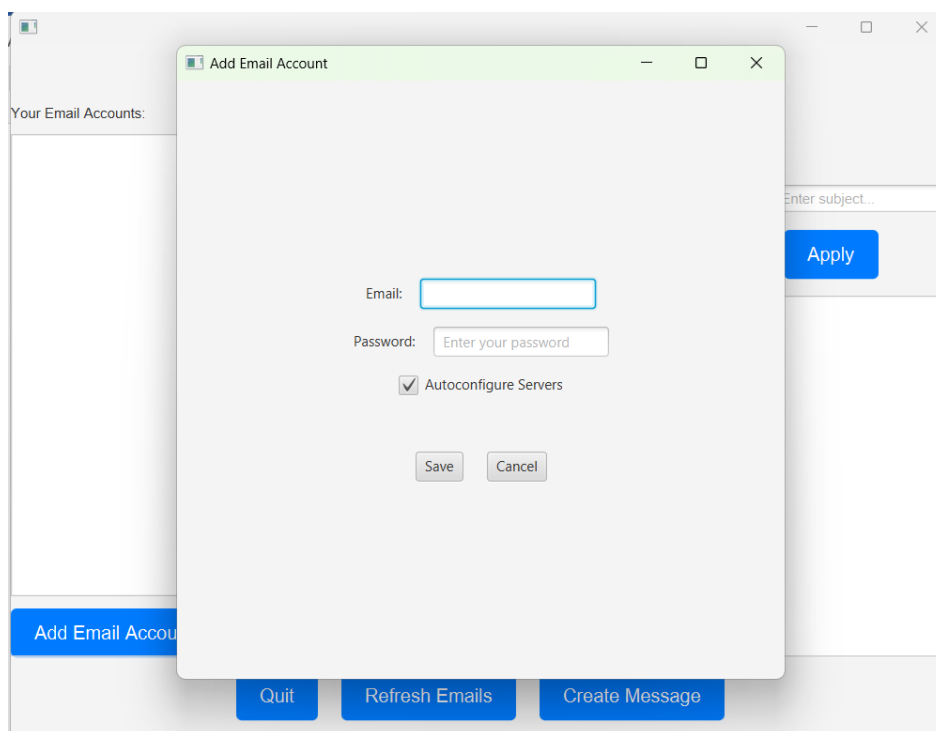


Рисунок 2.3.5. Вікно додавання облікового запису

На даному етапі користувач повинен заповнити поля, а також якщо він бажає сам налаштувати з'єднання з поштовим сервером, то повинен натиснути на галочку щоб відкрити поля для ручного налаштування.

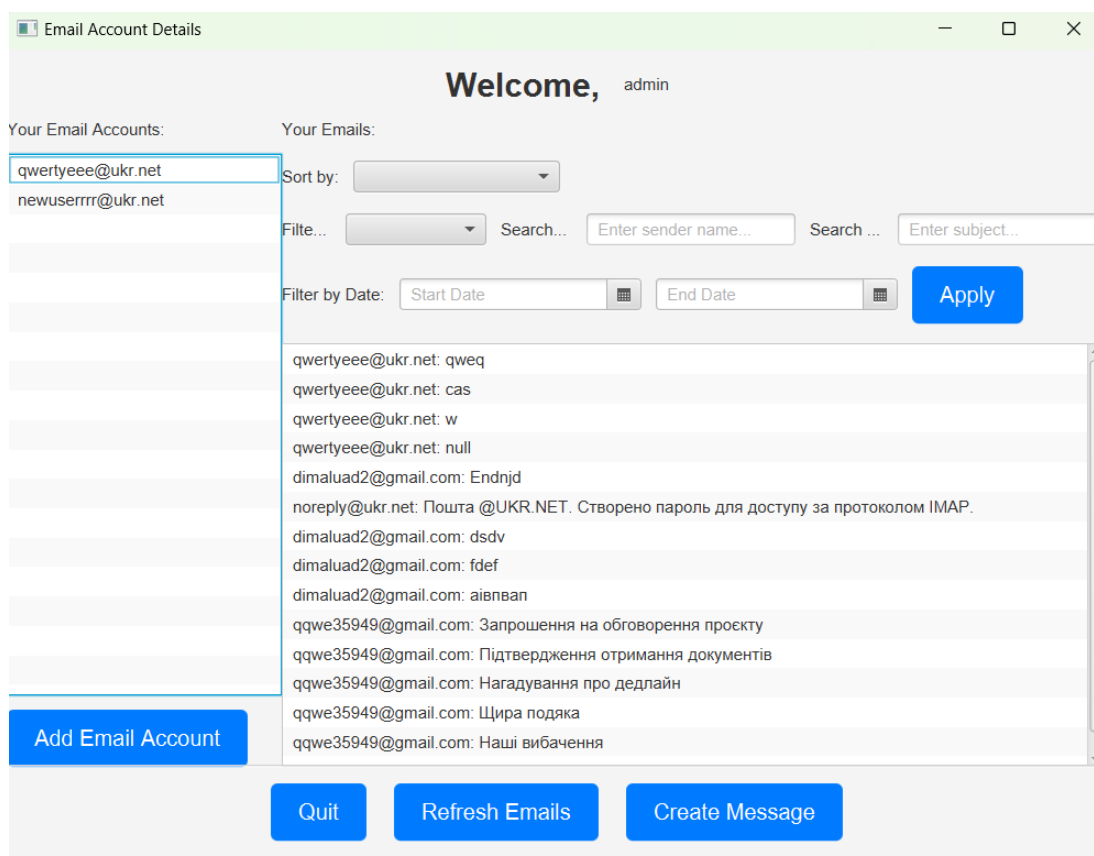


Рисунок 2.3.6. Головна сторінка

Коли користувач додав аккаунт електронної пошти, він може відсортувати всю пошту, увівши данні для сортування у відповідні поля вгорі і натиснувши кнопку “Apply”.

Кнопка “Refresh Emails” використовується для оновлення пошти.

Щоб подивитись повідомлення користувач повинен двічі натиснути на нього(рисунок 2.3.7).

Кнопка “Create Message” використовується для створення повідомлення(рисунок 2.3.8).

При натисканні двічі на адресу електронної пошти, відкривається сторінка, яка містить папки та повідомлення цієї адреси(рисунок 2.3.9).

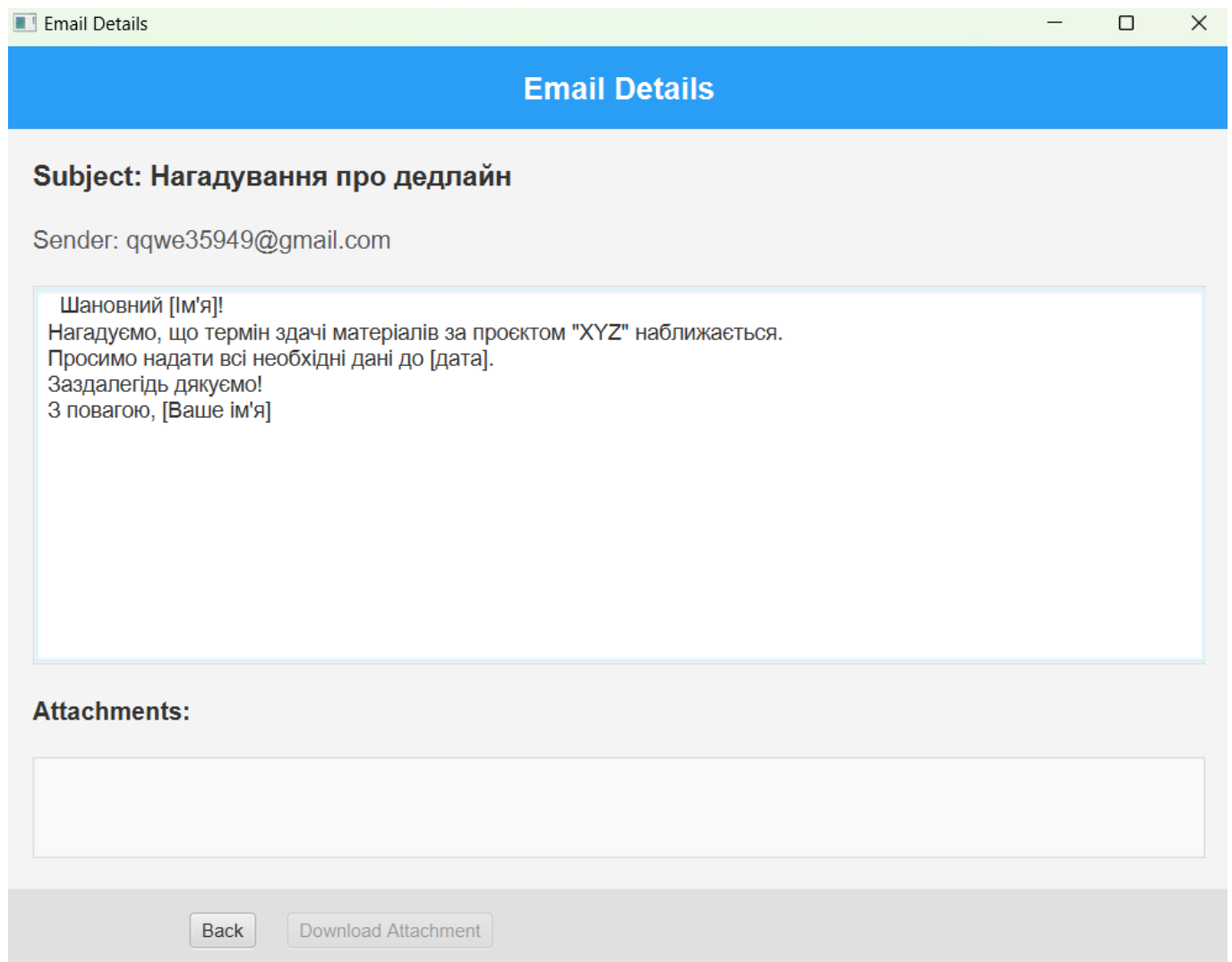


Рисунок 2.3.7. Сторінка повідомлення

Ця сторінка дозволяє переглядати деталі вибраного електронного листа.

Ключові функції сторінки

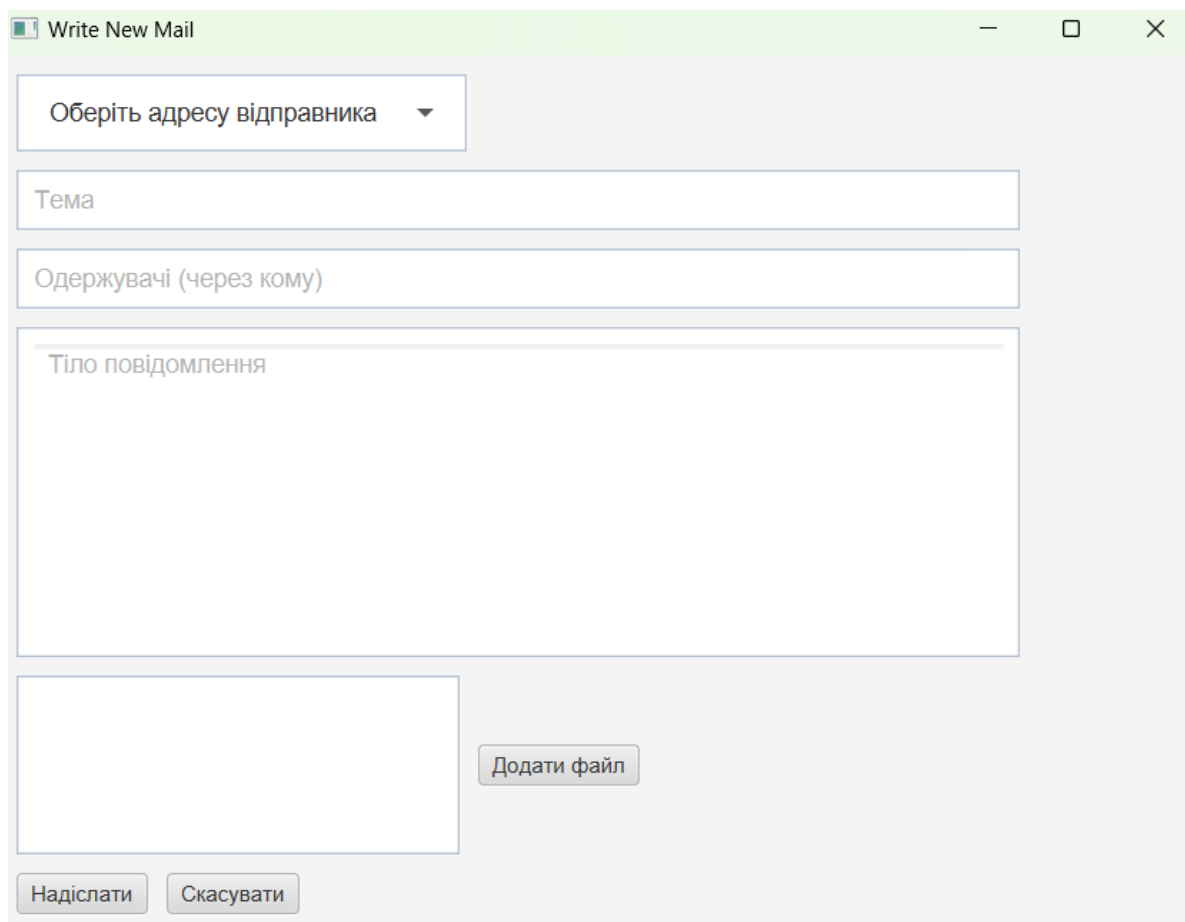
Перегляд деталей листа: Ви можете переглянути тему, відправника та текст листа.

Робота з вкладеннями: Список вкладень дає змогу переглядати, вибирати вкладені файли.

Організація листів: Якщо користувач відкриває повідомлення із сторінки аккаунта(рисунок 2.3.9), то він може додавати листи до папок або видаляти з них.

Видалення листів: Лист можна повністю видалити з поштової скриньки, якщо користувач відкриває повідомлення із сторінки аккаунта(рисунок 2.3.9) та папки “inbox”.

Навігація назад: Швидкий перехід до списку листів за допомогою кнопки "Back".



Write New Mail

Оберіть адресу відправника ▼

Тема

Одержувачі (через кому)

Тіло повідомлення

Додати файл

Надіслати Скасувати

Рисунок 2.3.8. Сторінка створення повідомлення

Ця сторінка призначена для створення та надсилання електронних листів. Ви також можете додавати вкладення та зберігати чернетки для подальшого редагування.

Як користуватися сторінкою

- Виберіть адресу відправника зі списку в полі зверху.
- Заповніть тему листа у відповідному полі.
- Додайте адреси одержувачів (через кому, якщо їх кілька).
- Напишіть текст листа у полі для тіла повідомлення.
- За потреби додайте файли через кнопку "Додати файл".

Виконайте одну з дій:

- Натисніть "Надіслати", щоб відправити лист.
- Натисніть "Зберегти як чернетку", щоб зберегти лист.
- Натисніть "Скасувати", щоб вийти без збереження.

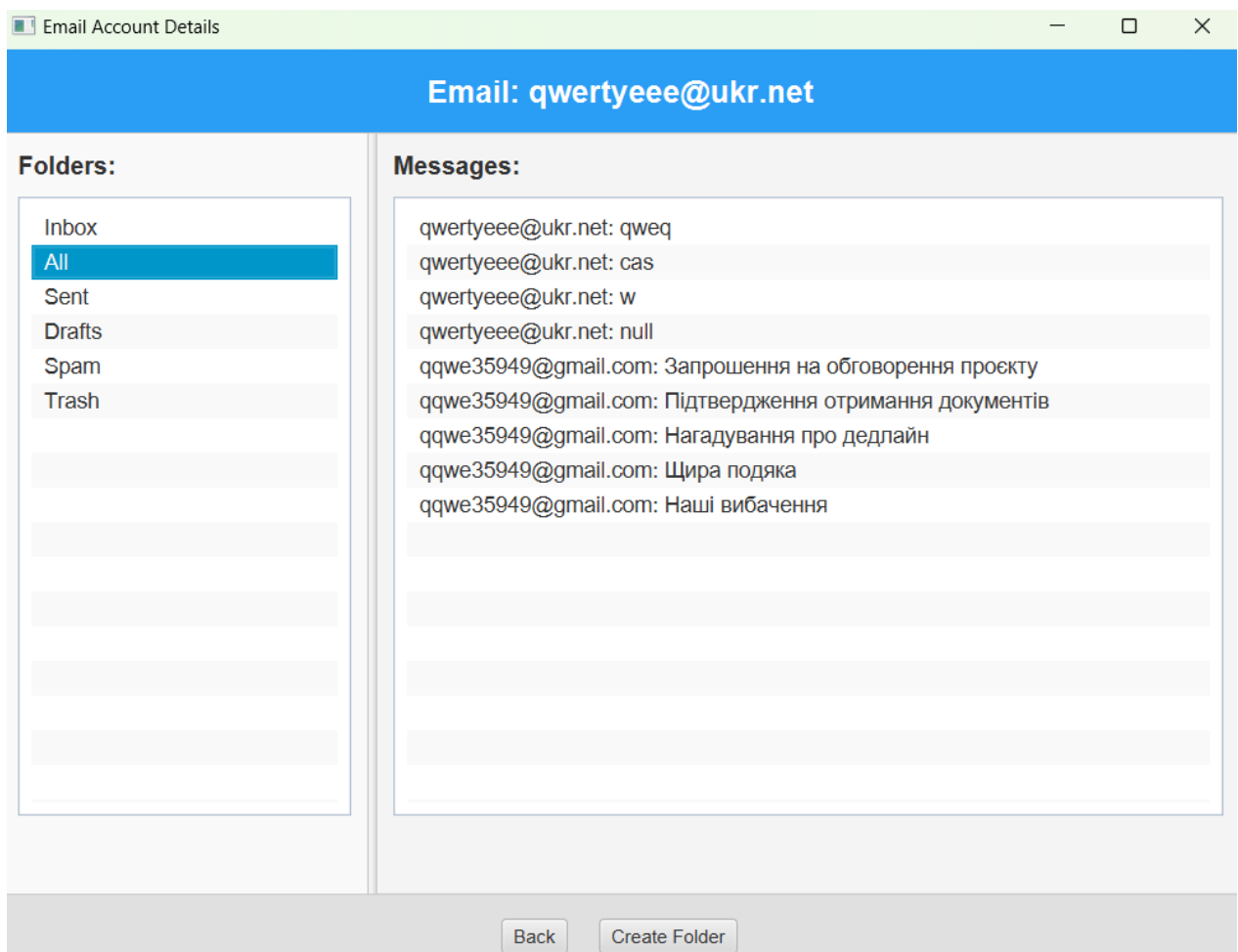


Рисунок 2.3.9. Сторінка аккаунта

Ця сторінка призначена для перегляду деталей акаунта електронної пошти, доступу до папок та повідомлень, а також управління папками.

Як користуватися сторінкою

1) Перегляд папок:

- Виберіть папку зі списку в лівій частині сторінки.
- Її вміст з'явиться у правій частині у вигляді списку повідомлень.

2) Перегляд повідомлень:

- У правій частині двічі натисніть повідомлення для детального перегляду(рисунок 2.3.10).

3) Створення папки:

- Натисніть кнопку "Create Folder", щоб додати нову папку.

4) Повернення до головного меню:

- Натисніть кнопку "Back", щоб перейти до головної сторінки програми.

ВИСНОВКИ

У процесі виконання курсової роботи на тему "Розробка e-mail клієнта" було досягнуто поставлених цілей і виконано всі завдання. Отримані результати підтверджують актуальність і практичну значущість розробки поштового клієнта, що базується на сучасних принципах проектування програмного забезпечення.

У ході роботи:

Проведено огляд існуючих поштових клієнтів, таких як Mozilla Thunderbird, The Bat, Microsoft Outlook, що дозволило виявити основні тенденції та найкращі практики у цій галузі. На основі цього було визначено ключові функціональні особливості, які повинні бути реалізовані в продукті.

Сформовано детальні функціональні та нефункціональні вимоги до системи. Зокрема, передбачено підтримку роботи з поштовими протоколами (POP3, SMTP, IMAP), автоматичне налаштування для українських поштових провайдерів (Gmail, Ukr.net, i.ua), організацію повідомлень за папками, збереження чернеток і роботу з прикріпленими файлами.

Розроблено концептуальну та фізичну моделі системи, що забезпечують її масштабованість, модульність і зручність подальшого розширення. Використання сервіс-орієнтованої архітектури (SOA) стало основою для створення незалежних модулів із чітко визначеним функціоналом.

Використано ефективні шаблони проектування:

Singleton — для управління єдиним екземпляром авторизованого користувача.

Builder — для спрощення створення та налаштування облікових записів електронної пошти.

Template Method — для реалізації алгоритму авторизації для різних поштових серверів.

Decorator — для гнучкої фільтрації повідомлень без дублювання коду.

Interpreter — для реалізації складних критеріїв пошуку та сортування листів.

Реалізовано програмний продукт, що включає:

автоматичну конфігурацію поштових провайдерів;

підтримку роботи з протоколами POP3, SMTP, IMAP;

функціонал створення та управління папками, збереження чернеток, роботи з вкладеннями;

інтуїтивно зрозумілий графічний інтерфейс для зручності користувачів.

Виконано інтеграцію із системою управління базами даних PostgreSQL, що забезпечує зберігання даних користувачів, повідомлень та конфігурацій. Архітектура бази даних розроблена таким чином, щоб оптимізувати доступ до даних і підтримувати їхню безпеку.

Проведено тестування, яке підтвердило працездатність системи у різних сценаріях використання, включаючи роботу з багатьма обліковими записами, обробку великих обсягів повідомлень і складні критерії пошуку.

Розроблений поштовий клієнт відповідає вимогам сучасних користувачів і може бути застосований для створення комерційних продуктів або впровадження в корпоративні системи.

Досвід, отриманий під час виконання курсової роботи, сприятиме подальшому професійному розвитку в галузі розробки програмного забезпечення та використанню передових технологій у реальних проєктах.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

- [1] "PostgreSQL Documentation." PostgreSQL.org [Електронний ресурс].
Доступно: <https://www.postgresql.org/docs/>
- [2] Bloch J., *Effective Java*, 3rd Edition, Addison-Wesley, 2018.
- [3] "Effective Java Development with IntelliJ IDEA." JetBrains Blog, 2024
[Електронний ресурс]. Доступно: <https://blog.jetbrains.com/>
- [4] What Is SOA (Service-Oriented Architecture)? [Електронний ресурс].
Доступно: <https://www.oracle.com/ua/service-oriented-architecture-soa>.
- [5] Jakarta Mail [Електронний ресурс]. Доступно:
<https://jakarta.ee/specifications/mail/2.0/jakarta-mail-spec-2.0>
- [6] JavaFX [Електронний ресурс]. Доступно:
<https://en.wikipedia.org/wiki/JavaFX>
- [7] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- [8] Одинак (шаблон проєктування) [Електронний ресурс]. Доступно:
[https://uk.wikipedia.org/wiki/Одинак_\(шаблон_проєктування\)](https://uk.wikipedia.org/wiki/Одинак_(шаблон_проєктування))
- [9] Будівельник (шаблон проєктування) [Електронний ресурс]. Доступно:
[https://uk.wikipedia.org/wiki/Будівельник_\(шаблон_проєктування\)](https://uk.wikipedia.org/wiki/Будівельник_(шаблон_проєктування)).
- [10] Шаблонний метод (шаблон проєктування) [Електронний ресурс].
Доступно: [https://uk.wikipedia.org/wiki/Шаблонний_метод_\(шаблон_проєктування\)](https://uk.wikipedia.org/wiki/Шаблонний_метод_(шаблон_проєктування))
- [11] Декоратор (шаблон проєктування) [Електронний ресурс]. Доступно:
[https://uk.wikipedia.org/wiki/Декоратор_\(шаблон_проєктування\)](https://uk.wikipedia.org/wiki/Декоратор_(шаблон_проєктування))
- [12] Інтерпретатор (шаблон проєктування) [Електронний ресурс]. Доступно:
[https://uk.wikipedia.org/w/Інтерпретатор_\(шаблон_проєктування\)](https://uk.wikipedia.org/w/Інтерпретатор_(шаблон_проєктування))

ДОДАТКИ

Додаток А

Вихідний код

Функції, які реалізують отримання повідомлень:

```
public List<EmailMessageContextDto>
getEmailMessages(List<EmailAccountDto> emailAccounts) throws
MessagingException, IOException, SQLException {
    Objects.requireNonNull(emailAccounts, "EmailAccounts cannot be
null");
    // Завантаження існуючих повідомлень з бази даних
    Set<String> existingMessageIds = new
HashSet<> (emailMessageRepository.findAllMessageIds());
    for (EmailAccountDto emailAccount : emailAccounts) {
        if (emailAccount == null) {
            log.warn("Null EmailAccountDto encountered, skipping.");
            continue;
        }
        // Підключення до поштового серверу
        Session = getSession(emailAccount, true);
        Store =
session.getStore(emailAccount.getIncomingServer().getProtocol().toLowerCase());
        store.connect(emailAccount.getIncomingServer().getHost(),
emailAccount.getEmailAddress(), emailAccount.getPassword());
        Folder rootFolder = store.getDefaultFolder();
        Folder[] folders = rootFolder.list();
        for (Folder : folders) {
            saveMessagesFromFolder(store, existingMessageIds,
emailAccount, folder.getName());
        }
        store.close();
    }
    // Повертаємо всі повідомлення (з бази + нові)
    return emailMessageRepository.findAll().stream()
        .map(emailMessageMapper::mapToEmailMessageContextDto)
        .toList();
}

private void saveMessagesFromFolder(Store store, Set<String>
existingMessageIds, EmailAccountDto emailAccount, String folderName)
throws MessagingException, IOException {
    Folder = store.getFolder(folderName);
    List<String> messagesToSaveInFolder = new ArrayList<>();
    try (folder) {
        folder.open(Folder.READ_ONLY);
        Message[] messages = folder.getMessages();
        List<EmailMessageEntity> newEntities = new ArrayList<>();
        for (Message : messages) {
            String[] messageIdHeader = message.getHeader("Message-
ID");
            String messageId = (messageIdHeader != null &&
```

```

messageIdHeader.length > 0) ? messageIdHeader[0] : null;
messagesToSaveInFolder.add(messageId);
// Пропустити, якщо Message-ID вже існує
if (messageId != null &&
existingMessageIds.contains(messageId)) {
    continue;
}
if (messageId == null) {
    log.warn("Message without Message-ID encountered,
skipping.");
    continue;
}
EmailMessage = new EmailMessage();
emailMessage.setMessageId(messageId);
emailMessage.setFrom(((InternetAddress)
message.getFrom()[0]).getAddress());
Address[] recipients =
message.getRecipients(Message.RecipientType.TO);
if (recipients != null) {
    List<String> recipientList = new ArrayList<>();
    for (Address recipient : recipients) {
        recipientList.add(((InternetAddress)
recipient).getAddress());
    }
    emailMessage.setTo(recipientList);
}
emailMessage.setSubject(message.getSubject());
emailMessage.setSentDate(message.getSentDate().toInstant()
    .atZone(ZoneId.systemDefault())
    .toLocalDateTime());
emailMessage.setBody(getTextFromMessage(message));

emailMessage.setAttachmentNames(getAttachmentNamesFromMessage(message)
);

Flags flags = message.getFlags();
if (flags.contains(Flags.Flag.SEEN)) {
    emailMessage.setEmailStatus(EmailStatus.READ);
} else if (flags.contains(Flags.Flag.DELETED)) {
    emailMessage.setEmailStatus(EmailStatus.DELETED);
} else if (flags.contains(Flags.Flag.DRAFT)) {
    emailMessage.setEmailStatus(EmailStatus.DRAFT);
} else {
    emailMessage.setEmailStatus(EmailStatus.UNREAD);
}
EmailMessageEntity entity =
emailMessageMapper.mapToEntity(emailMessage);
newEntities.add(entity);
emailMessageRepository.save(entity);
}
emailMessageRepository.saveAll(newEntities);
}
MessagesToFolderDto messages = new

```

```

MessagesToFolderDto(messagesToSaveInFolder,
emailAccount.getEmailAddress(), folderName);
    ResponseEntity<Void> response =
restTemplate.postForEntity(folderControllerUrl + "/save/messages",
messages, Void.class);
    if (!response.getStatusCode().is2xxSuccessful()) {
        throw new IllegalArgumentException("Failed to save emails.
Status code: " + response.getStatusCode());
    }

}

private Session getSession(EmailAccountDto emailAccount, boolean
incoming) {
    if (incoming) {
        EmailProtocolHandler incomingHandler = new
IncomingServerHandler();
        Session incomingSession =
incomingHandler.authorize(emailAccount,
emailAccount.getIncomingServer());
        if (incomingSession == null) {
            log.warn("Authorization failed on incoming server for: " +
emailAccount.getEmailAddress());
            return null;
        }
        return incomingSession;
    } else {
        EmailProtocolHandler outgoingHandler = new
OutgoingServerHandler();
        Session outgoingSession =
outgoingHandler.authorize(emailAccount,
emailAccount.getOutgoingServer());
        if (outgoingSession == null) {
            log.warn("Authorization failed on outgoing server for: " +
emailAccount.getEmailAddress());
            return null;
        }
        return outgoingSession;
    }
}

private String getTextFromMessage(Message message) throws
MessagingException, IOException {
    if (message.isMimeType("text/plain")) {
        return (String) message.getContent();
    } else if (message.isMimeType("text/html")) {
        return (String) message.getContent();
    } else if (message.isMimeType("multipart/*")) {
        Multipart = (Multipart) message.getContent();
        for (int i = 0; i < multipart.getCount(); i++) {
            BodyPart = multipart.getBodyPart(i);
            if (bodyPart.isMimeType("text/plain")) {
                return (String) bodyPart.getContent();
            }
        }
    }
}

```



```

        } else if (bodyPart.isMimeType("text/html")) {
            return (String) bodyPart.getContent();
        }
    }
}
return null;
}

```

Функція, яка реалізовує збереження чернетки:

```

public void saveDraft(EmailMessageSendDto emailMessageSendDto) throws
MessagingException {
    EmailAccountDto emailAccountDto = emailMessageSendDto.getFrom();

    Session session = getSession(emailAccountDto, true);
    MimeMessage message = new MimeMessage(session);
    message.setFrom(new
InternetAddress(emailAccountDto.getEmailAddress()));
    message.setRecipients(Message.RecipientType.TO,
        InternetAddress.parse(String.join(", ",
emailMessageSendDto.getTo())));
    message.setSubject(emailMessageSendDto.getSubject());
    MimeBodyPart mailBody = new MimeBodyPart();
    mailBody.setText(emailMessageSendDto.getBody());
    Multipart multipart = new MimeMultipart();
    multipart.addBodyPart(mailBody);
    if (!emailMessageSendDto.getAttachmentPaths().isEmpty()) {
        // Завантаження файлів з attachment-service
        List<String> attachmentPaths =
emailMessageSendDto.getAttachmentPaths();
        List<FileDto> attachments =
attachmentService.downloadAttachments(attachmentPaths); // Повертаємо
List<byte[]>
        for (FileDto fileDto : attachments) {
            MimeBodyPart attachmentPart = new MimeBodyPart();
            // Використовуємо ByteArrayDataSource замість File
            ByteArrayDataSource source = new
ByteArrayDataSource(fileDto.getFileBytes(), fileDto.getMimeType());
            attachmentPart.setDataHandler(new DataHandler(source));
            attachmentPart.setFileName(fileDto.getName()); //
Унікальна назва для кожного вкладення

            multipart.addBodyPart(attachmentPart);
        }
    }
    message.setContent(multipart);

    Store store = session.getStore("imap");
    store.connect(emailAccountDto.getIncomingServer().getHost(),
emailAccountDto.getEmailAddress(), emailAccountDto.getPassword());
    Folder draftsFolder = store.getFolder("Drafts");
    if (!draftsFolder.exists()) {
        draftsFolder.create(Folder.HOLDS_MESSAGES);
    }
}

```

```

    }
    draftsFolder.open(Folder.READ_WRITE);
    message.saveChanges();
    draftsFolder.appendMessages(new Message[]{message});
    draftsFolder.close(false);
    store.close();
    String messageId = message.getHeader("Message-ID", null);
    // Збереження у базу даних
    EmailMessageEntity entity =
emailMapper.mapToEntity(emailMessageSendDto, messageId);
    emailRepository.save(entity);

    MessagesToFolderDto messages = new
MessagesToFolderDto(List.of(messageId),
emailAccountDto.getEmailAddress(), "Draft");
    ResponseEntity<Void> response =
restTemplate.postForEntity(folderControllerUrl + "/save/messages",
messages, Void.class);
    if (!response.getStatusCode().is2xxSuccessful()) {
        throw new IllegalArgumentException("Failed to save emails.
Status code: " + response.getStatusCode());
    }
}

```

Функція, яка реалізовує відправлення повідомлення:

```

public void sendEmail(EmailMessageSendDto emailMessageSendDto) throws
MessagingException, IOException {
    EmailAccountDto emailAccountDto = emailMessageSendDto.getFrom();
    Session session = getSession(emailAccountDto, false);
    MimeMessage message = new MimeMessage(session);
    message.setFrom(new
InternetAddress(emailAccountDto.getEmailAddress()));
    message.setRecipients(Message.RecipientType.TO,
        InternetAddress.parse(String.join(",",
emailMessageSendDto.getTo())));
    message.setSubject(emailMessageSendDto.getSubject());

    MimeBodyPart mailBody = new MimeBodyPart();
    mailBody.setText(emailMessageSendDto.getBody());
    Multipart multipart = new MimeMultipart();
    multipart.addBodyPart(mailBody);
    if (!emailMessageSendDto.getAttachmentPaths().isEmpty()) {
        // Завантаження файлів з attachment-service
        List<String> attachmentPaths =
emailMessageSendDto.getAttachmentPaths();
        List<FileDto> attachments =
attachmentService.downloadAttachments(attachmentPaths); // Повертаємо
List<byte[]>
        for (FileDto fileDto : attachments) {
            MimeBodyPart attachmentPart = new MimeBodyPart();

            // Використовуємо ByteArrayDataSource замість File

```

```

        ByteArrayDataSource source = new
ByteArrayDataSource(fileDto.getFileBytes(), fileDto.getMimeType());
        attachmentPart.setDataHandler(new DataHandler(source));
        attachmentPart.setFileName(fileDto.getName()); //
        Унікальна назва для кожного вкладення
        multipart.addBodyPart(attachmentPart);
    }
}
message.setContent(multipart);
try {
    Transport.send(message);
} catch (MessagingException e) {
    log.error("Failed to send email: {}", e.getMessage(), e);
    throw new IllegalStateException("Email sending failed", e);
}
String messageId = message.getHeader("Message-ID", null);
// Збереження у базу даних
EmailMessageEntity entity =
emailMapper.mapToEntity(emailMessageSendDto, messageId);
entity.setEmailStatus(EmailStatus.SENT);
emailRepository.save(entity);
}

```

Функція, яка реалізовує видалення повідомлення:

```

public void deleteEmailMessage(EmailMessageDeleteDto
emailMessageDeleteDto) throws MessagingException {
    EmailMessageEntity emailMessage =
emailRepository.findById(emailMessageDeleteDto.getMessageId())
        .orElseThrow(() -> new IllegalArgumentException("Email
message with the given ID does not exist"));
    EmailAccountDto emailAccountDto = emailMessageDeleteDto.getFrom();
    Session session = getSession(emailAccountDto, true);
    emailMessage.setEmailStatus(EmailStatus.DELETED);
    emailRepository.save(emailMessage);
    MessagesToFolderDto messagesToTrash = new
MessagesToFolderDto(List.of(emailMessageDeleteDto.getMessageId()),
emailAccountDto.getEmailAddress(), "Trash");
    ResponseEntity<Void> response =
restTemplate.postForEntity(folderControllerUrl + "/save/messages",
messagesToTrash, Void.class);
    if (!response.getStatusCode().is2xxSuccessful()) {
        throw new IllegalArgumentException("Failed to save emails.
Status code: " + response.getStatusCode());
    }
    MessagesToFolderDto messagesDeleteFromInbox = new
MessagesToFolderDto(List.of(emailMessageDeleteDto.getMessageId()),
emailAccountDto.getEmailAddress(), "Inbox");
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);
    HttpEntity<MessagesToFolderDto> requestEntity = new
HttpEntity<>(messagesDeleteFromInbox, headers);
    restTemplate.exchange(
        folderControllerUrl + "/delete/from/folder",

```

```

        HttpMethod.DELETE,
        requestEntity,
        Void.class
    );
    MessagesToFolderDto messagesDeleteFromAll = new
MessagesToFolderDto(List.of(emailMessageDeleteDto.getMessageId()),
emailAccountDto.getEmailAddress(), "All");
    HttpEntity<MessagesToFolderDto> request = new
HttpEntity<>(messagesDeleteFromAll, headers);
    restTemplate.exchange(
        folderControllerUrl + "/delete/from/folder",
        HttpMethod.DELETE,
        request,
        Void.class
    );
    try (Store store = session.getStore("imap")) {
        store.connect(emailAccountDto.getIncomingServer().getHost(),
emailAccountDto.getEmailAddress(), emailAccountDto.getPassword());

        Folder sourceFolder = store.getFolder("INBOX");
        sourceFolder.open(Folder.READ_WRITE);
        // Знаходимо повідомлення за унікальним ID
        Message[] messages = sourceFolder.search(new
MessageIDTerm(emailMessage.getMessageId()));
        if (messages.length == 0) {
            throw new IllegalArgumentException("Email message not
found in the INBOX.");
        }
        Message messageToDelete = messages[0];
        // Перевірка, чи повідомлення не видалене
        if (messageToDelete.isExpunged()) {
            throw new IllegalArgumentException("Email message is
already removed.");
        }
        // Переміщення до папки Trash
        Folder trashFolder = store.getFolder("Trash");
        if (!trashFolder.exists()) {
            trashFolder.create(Folder.HOLDS_MESSAGES);
        }
        trashFolder.open(Folder.READ_WRITE);
        sourceFolder.copyMessages(new Message[]{messageToDelete},
trashFolder);
        messageToDelete.setFlag(Flags.Flag.DELETED, true); //
Позначаємо для видалення

        sourceFolder.close(false);
        trashFolder.close(false);
    } catch (MessageRemovedException e) {
        log.warn("Email message already removed: {}", e.getMessage());
    } catch (MessagingException e) {
        log.error("Failed to delete email message: {}",
e.getMessage());
    }
}

```

```

        throw e;
    }

    log.info("Email message with ID {} has been deleted.",
emailMessage.getMessageId());
}

```

Функції, які реалізують додавання поштового акаунта:

```

public void addEmailAccount(String username, EmailAccountDto
emailAccountDto) {
    Objects.requireNonNull(username, "username cannot be null");
    Objects.requireNonNull(emailAccountDto.getEmailAddress(), "Email
cannot be null");
    Objects.requireNonNull(emailAccountDto.getPassword(), "Password
cannot be null");
    EmailAccount emailAccount;
    if (emailAccountDto.getAutoconfig()) {
        emailAccount = new
EmailAccount.EmailAccountBuilder(emailAccountDto.getEmailAddress(),
emailAccountDto.getPassword())
            .setAutoconfig(emailAccountDto.getAutoconfig())
            .build();
    } else {
        emailAccount = emailAccountMapper.mapToModel(emailAccountDto);
    }
    //authentication
    EmailAccount authorizedEmailAccount =
authorizeEmail(emailAccount);
    if (authorizedEmailAccount != null) {
        UserEntity userEntity =
userRepository.findByUsername(username).orElseThrow(() -> new
IllegalArgumentException("User not found"));
        ServerConnectionEntity in =
emailAccountMapper.mapToServerConnectionEntity(authorizedEmailAccount.
getIncomingServer());
        ServerConnectionEntity out =
emailAccountMapper.mapToServerConnectionEntity(authorizedEmailAccount.
getOutgoingServer());
        ServerConnectionEntity incomingServer;
        ServerConnectionEntity outgoingServer;
        if
(!serverConnectionRepository.existsByHostAndPortAndProtocol(in.getHost
(), in.getPort(), in.getProtocol())) {
            incomingServer = serverConnectionRepository.save(in);
        } else {
            incomingServer =
serverConnectionRepository.findByHostAndPortAndProtocol(in.getHost(),
in.getPort(), in.getProtocol());
        }
        if
(!serverConnectionRepository.existsByHostAndPortAndProtocol(out.getHos
t(), out.getPort(), out.getProtocol())) {

```

```

        outgoingServer = serverConnectionRepository.save(out);
    } else {
        outgoingServer =
serverConnectionRepository.findByHostAndPortAndProtocol(out.getHost(),
out.getPort(), out.getProtocol());
    }
    EmailAccountEntity emailAccountEntity =
emailAccountMapper.mapToEntity(authorizedEmailAccount, userEntity);
    emailAccountEntity.setIncomingServer(incomingServer);
    emailAccountEntity.setOutgoingServer(outgoingServer);
    log.info("Email account added to user: " + username + " (" +
emailAccount.getEmailAddress() + ")");
    emailAccountMapper.mapToEmailAccountDto(emailAccountRepository.save(em
ailAccountEntity));
    return;
}
throw new IllegalArgumentException("Authorization failed for
email: " + emailAccountDto.getEmailAddress());
}

private EmailAccount authorizeEmail(EmailAccount emailAccount) {
    EmailProtocolHandler incomingHandler = new
IncomingServerHandler();
    Session incomingSession = incomingHandler.authorize(emailAccount,
emailAccount.getIncomingServer());
    if (incomingSession == null) {
        log.warn("Authorization failed on incoming server for: " +
emailAccount.getEmailAddress());
        return null;
    }
    EmailProtocolHandler outgoingHandler = new
OutgoingServerHandler();
    Session outgoingSession = outgoingHandler.authorize(emailAccount,
emailAccount.getOutgoingServer());

    if (outgoingSession == null) {
        log.warn("Authorization failed on outgoing server for: " +
emailAccount.getEmailAddress());
        return null;
    }
    EmailAccount authorizedEmailAccount = new
EmailAccount.EmailAccountBuilder(emailAccount.getEmailAddress(),
emailAccount.getPassword())
        .setAutoconfig(true)
        .build();

    log.info("Authorization successful for: " +
emailAccount.getEmailAddress());
    return authorizedEmailAccount;
}

```

Контролер в email-service:

```

@RestController
@RequestMapping("/emails")
@RequiredArgsConstructor
@Slf4j
public class EmailController {

    private final EmailMessageService emailMessageService;
    @PostMapping("/send")
    public ResponseEntity<String> sendEmail(@RequestBody
EmailMessageSendDto emailMessageSendDto) {
        try {
            emailMessageService.sendEmail(emailMessageSendDto);
            return ResponseEntity.ok("Email sent successfully");
        } catch (MessagingException | IOException e) {
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body("Failed to send email: " + e.getMessage());
        }
    }
    @PostMapping("/save/draft")
    public ResponseEntity<String> saveDraft(@RequestBody
EmailMessageSendDto emailMessageSendDto) {
        try {
            emailMessageService.saveDraft(emailMessageSendDto);
            return ResponseEntity.ok("Draft saved successfully");
        } catch (MessagingException e) {
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body("Failed to save draft: " + e.getMessage());
        }
    }
    @PostMapping
    public ResponseEntity<List<EmailMessageContextDto>>
getEmails(@RequestBody List<EmailAccountDto> emailAccounts) throws
SQLException {
        try {
            List<EmailMessageContextDto> emailMessages =
emailMessageService.getEmailMessages(emailAccounts);
            return ResponseEntity.ok(emailMessages);
        } catch (MessagingException | IOException e) {
            log.warn(e.getMessage());
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(null);
        }
    }
    @PostMapping("/get/message")
    public ResponseEntity<EmailMessageDto> getEmail(@RequestBody
EmailMessageContextDto contextDto) {
        EmailMessageDto messageDto =
emailMessageService.getEmailMessage(contextDto);
        return ResponseEntity.ok(messageDto);
    }
}

```



```

    }
    @DeleteMapping
    public ResponseEntity<Void> deleteEmail(@RequestBody
EmailMessageDeleteDto emailMessageDeleteDto) throws MessagingException
{
    emailMessageService.deleteEmailMessage(emailMessageDeleteDto);
    return ResponseEntity.ok().build();
}
}

```

Використання контролера з модуля app-module:

```

@Component
@RequiredArgsConstructor
public class EmailService {
    private final RestTemplate restTemplate;
    private final String emailControllerUrl =
"http://localhost:8082/emails";

    public EmailService() {
        this.restTemplate = new RestTemplate();
    }

    public List<EmailMessageContextDto>
getAllMessages(List<EmailAccountDto> emailAccounts) {
        if (emailAccounts.isEmpty()) {
            return new ArrayList<>();
        }
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);

        HttpEntity<List<EmailAccountDto>> requestEntity = new
HttpEntity<>(emailAccounts, headers);
        ResponseEntity<List<EmailMessageContextDto>> response =
restTemplate.exchange(
            emailControllerUrl,
            HttpMethod.POST,
            requestEntity,
            new ParameterizedTypeReference<>() {
            }
        );
        if (response.getStatusCode().is2xxSuccessful()) {
            return response.getBody();
        } else throw new IllegalArgumentException();
    }

    public void deleteMessage( EmailAccountDto emailAccountDto, String
messageId) {
        EmailMessageDeleteDto emailMessageSendDto = new
EmailMessageDeleteDto(messageId, emailAccountDto);
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        HttpEntity<EmailMessageDeleteDto> requestEntity = new
HttpEntity<>(emailMessageSendDto, headers);
    }
}

```



```

        restTemplate.exchange(
            emailControllerUrl,
            HttpMethod.DELETE,
            requestEntity,
            Void.class
        );
    }

    public EmailMessageDto getMessage(EmailMessageContextDto email) {
        try {
            ResponseEntity<EmailMessageDto> response =
                restTemplate.postForEntity(emailControllerUrl + "/get/message", email,
                    EmailMessageDto.class);
            return response.getBody();
        } catch (HttpClientErrorException e) {
            System.out.println("Error get message: " +
                e.getResponseBodyAsString());
            throw new IllegalArgumentException("Getting message
                failed: " + e.getMessage());
        }
    }

    public String sendMessage(String subject, EmailAccountDto from,
        List<String> to, String body, List<String> attachmentNamePaths) {
        try {
            EmailMessageSendDto messageSendDto = new
                EmailMessageSendDto(subject, from, to, body, attachmentNamePaths);
            ResponseEntity<String> response =
                restTemplate.postForEntity(emailControllerUrl + "/send",
                    messageSendDto, String.class);
            return response.getBody();
        } catch (HttpClientErrorException e) {
            System.out.println("Error send message: " +
                e.getResponseBodyAsString());
            throw new IllegalArgumentException("Sending message
                failed: " + e.getMessage(), e);
        }
    }

    public String saveDraft(String subject, EmailAccountDto from,
        List<String> to, String body, List<String> attachmentNamePaths) {
        try {
            EmailMessageSendDto messageSendDto = new
                EmailMessageSendDto(subject, from, to, body, attachmentNamePaths);
            ResponseEntity<String> response =
                restTemplate.postForEntity(emailControllerUrl + "/save/draft",
                    messageSendDto, String.class);
            return response.getBody();
        } catch (HttpClientErrorException e) {

```

```

        System.out.println("Error saving draft: " +
e.getResponseBodyAsString());
        throw new IllegalArgumentException("Saving draft failed: "
+ e.getMessage(), e);
    }
}
}

```

Функція збереження повідомлень в папку:

```

public void saveMessagesToFolder(MessagesToFolderDto dto) throws
ChangeSetPersister.NotFoundException {
    FolderEntity folder =
folderRepository.findByName(dto.getFolderName())
        .orElseGet(() -> {
            FolderEntity newFolder = new FolderEntity();
            newFolder.setName(dto.getFolderName());
            return folderRepository.save(newFolder);
        });
    EmailAddressEntity emailAddress =
emailAddressRepository.findByAddress(dto.getEmailAddress())
        .orElseGet(() -> {
            EmailAddressEntity entity = new EmailAddressEntity();
            entity.setAddress(dto.getEmailAddress());
            return emailAddressRepository.save(entity);
        });

    FolderEmailAddressEntity folderEmailAddress =
folderEmailAddressRepository.findByFolderAndEmailAddress(folder,
emailAddress)
        .orElseGet(() -> {
            FolderEmailAddressEntity newFolderEmail = new
FolderEmailAddressEntity();
            newFolderEmail.setFolder(folder);
            newFolderEmail.setEmailAddress(emailAddress);
            return
folderEmailAddressRepository.save(newFolderEmail);
        });
    for (String messageId: dto.getMessageIds()) {
        EmailMessageEntity entity =
emailMessageRepository.findById(messageId).orElseThrow(ChangeSetPersis
ter.NotFoundException::new);
        if
(!folderEmailMessageRepository.existsByFolderEmailAddressAndMessage(fo
lderEmailAddress, entity)) {
            FolderEmailMessageEntity folderEmailMessage = new
FolderEmailMessageEntity();

            folderEmailMessage.setFolderEmailAddress(folderEmailAddress);
            folderEmailMessage.setMessage(entity);
            folderEmailMessageRepository.save(folderEmailMessage);
        }
    }
}
}

```