МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп’ютерних наук та управління проєктами

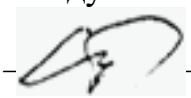
Кафедра програмного забезпечення автоматизованих систем

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма «Інженерія програмного забезпечення»

«Допущений до захисту»

Завідувач кафедри

_  __ проф. Приходько С.Б.

«___» _____ 2026 р.

КВАЛІФІКАЦІЙНАРОБОТА

на здобуття ступеня вищої освіти «бакалавр»

на тему: Розробка програмного забезпечення для автоматичної обробки повідомлень електронної пошти

Виконав: студент групи 4151

 Євдошин О. А.

(підпис, ПІБ)

Керівник роботи:

асистент кафедри ПЗАС, доктор філософії

(посада, науковий ступень вчене звання)

 Трухов А. С.

(підпис, ПІБ)

Миколаїв – 2026 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп'ютерних наук та управління проєктами
Кафедра програмного забезпечення автоматизованих систем
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма «Інженерія програмного забезпечення»

«ЗАТВЕРДЖУЮ»
Гарант освітньої програми
_____ доц. Макарова Л.М.
(підпис)
«_07_» ___04___ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
на здобуття ступеня вищої освіти «бакалавр»

Студенту Євдошину Олегу Андрійовичу
(Прізвище, ім'я, по батькові)

1. Тема роботи: Розробка програмного забезпечення для автоматичної обробки повідомлень електронної пошти
Керівник роботи асистент кафедри ПЗАС, доктор філософії Трухов А. С.
Затверджені наказом ректора №275-уч від 07.04.2026 р.
2. Термін подання роботи: 01.06.2026 р.
3. Вихідні дані по роботі: _____
_____
4. Перелік питань, що належать до розробки (найменування розділів)_____
Титульний аркуш; Завдання на кваліфікаційну роботу; Анотація (українською та англійською мовами); Зміст; Вступ; Аналіз практичної задачі інженерії програмного забезпечення; Проєкт програмного забезпечення; Результати розробки програмного забезпечення; Розділ з охорони праці; Висновки; Список використаних джерел; Додатки (технічне завдання, текст програми, опис програми, інструкція користувача, програма та методика випробувань програмного забезпечення).
5. Перелік презентаційних матеріалів Титульний слайд; Мета та завдання роботи; Актуальність теми; Постановка задачі; Огляд існуючих рішень; Технології та інструменти розробки; Діаграма варіантів використання; Архітектура програмного забезпечення; Діаграма класів сервісного шару та шару доступу до даних; Проєктування бази даних; Демонстрація застосунку: налаштування системи; Демонстрація застосунку: результати обробки; Висновки.
_____
_____
_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
4	Гурець Н.В., ст. викладач		

7. Дата видачі завдання 07.04.2026 р.

К А Л Е Н Д А Р Н И Й П Л А Н

Номер	Назва етапів роботи	Терміни виконання	Примітка
1.	Підготовка розділу ВСТУП	30.03.2026	ПП*
2.	Аналіз практичної задачі інженерії ПЗ	06.04.2026	ПП*
3.	Аналіз вимог до ПЗ	13.04.2026	ПП*
4.	Розробка архітектури ПЗ	20.04.2026	
5.	Розробка проекту ПЗ	04.05.2026	
6.	Реалізація та тестування ПЗ	11.05.2026	
7.	Підготовка розділу Результати розробки ПЗ	15.05.2026	
8.	Підготовка розділу з охорони праці	18.05.2026	ПП*
9.	Підготовка розділу ВИСНОВКИ	22.05.2026	
10.	Оформлення списку використаних джерел та додатків	25.05.2026	
11.	Подання на кафедру ПЗАС тексту остаточного варіанту роботи, підписаного її керівником, у роздрукованому та електронному форматі разом із заявами щодо самостійності виконання роботи та ідентичності друкованої та електронної версій роботи (Додатки 1 і 2 «Порядку здійснення заходів з перевірки робіт на наявність текстових збігів/ідентичності/схожості із використанням програмно-технічних засобів», який введений в дію наказом ректора НУК за №20 від 20.01.2020 р.)	01.06.2026	не пізніше, ніж за 14 діб до захисту (згідно п.4.1 зазначеного Порядку
12.	Підготовка презентації та доповіді	05.06.2026	
13.	Попередній захист роботи на засіданні кафедри ПЗАС	08.06.2026	
14.	Подання на кафедру ПЗАС електронних версій наступних документів у форматі pdf: кваліфікаційної роботи; файлу-опису кваліфікаційної роботи (згідно Додатку до наказу ректора НУК за №287-уч від 19.05.2020 р.); презентації доповіді, а також Авторського договору з додатком	15.06.2026	

* - за результатами переддипломної практики (ПП), яка була з 02.02.2026 до 22.03.2026 р.

Студент _____ Євдошин О. А.
(підпис) (ПІБ)

Керівник роботи _____ Трухов А. С.
(підпис) (ПІБ)

АНОТАЦІЯ

Кваліфікаційна робота присвячена розробці програмного забезпечення для автоматичної обробки повідомлень електронної пошти. Актуальність роботи зумовлена тим, що в багатьох організаціях електронна пошта залишається одним із основних каналів отримання документів, звітів, рахунків, заявок та інших вкладених файлів. Ручна обробка таких повідомлень потребує значних витрат часу, залежить від уважності працівника та може супроводжуватися помилками, зокрема пропуском потрібного листа, неправильним збереженням вкладення або повторною обробкою одного й того самого повідомлення.

У роботі проаналізовано практичну задачу автоматизації обробки електронної пошти, розглянуто існуючі програмні засоби та обґрунтовано доцільність створення власного програмного рішення. Сформовано функціональні й нефункціональні вимоги до програмного забезпечення, розроблено архітектуру WPF-застосунку з використанням шаблону MVVM, спроєктовано структуру класів, базу даних і сценарії взаємодії користувача із системою.

Розроблене програмне забезпечення реалізовано мовою C# на платформі .NET. Для побудови інтерфейсу користувача використано технологію WPF, для доступу до бази даних — Entity Framework Core і Microsoft SQL Server, для роботи з електронною поштою — бібліотеку MailKit, для передавання даних до зовнішніх систем — HTTP API.

Кваліфікаційна робота викладена на 180 сторінках друкованого тексту, та містить 13 рисунків, 7 таблиць, список використаних джерел з 14 найменувань та 5 додатків.

ABSTRACT

The qualification work is devoted to the development of software for automatic processing of email messages. The relevance of the work is determined by the fact that in many organizations email remains one of the main channels for receiving documents, reports, invoices, requests, and other attached files. Manual processing of such messages requires considerable time, depends on the attentiveness of employees, and may lead to errors, including missing an important email, saving an attachment incorrectly, or processing the same message more than once.

The work analyzes the practical task of automating email processing, reviews existing software solutions, and substantiates the need to develop a custom software solution. Functional and non-functional requirements for the software are defined. The architecture of the WPF application based on the MVVM pattern is designed, along with the class structure, database structure, and user interaction scenarios.

The developed software is implemented in C# on the .NET platform. WPF is used to build the user interface, Entity Framework Core and Microsoft SQL Server are used for database access, MailKit is used for working with email, and HTTP API is used for transferring data to external systems.

The qualification work is presented on 180 pages of printed text and contains 13 figures, 7 tables, a list of used sources from 14 titles and 5 appendices.

ЗМІСТ

ВСТУП

Електронна пошта залишається одним із найпоширеніших засобів обміну службовою інформацією між організаціями, працівниками, клієнтами, партнерами та зовнішніми інформаційними системами. Через електронні повідомлення регулярно передаються рахунки, акти, звіти, заявки, договори, замовлення, таблиці, архіви та інші файли, які потребують подальшого збереження, перевірки, реєстрації або передавання до внутрішніх програмних систем. Незважаючи на розвиток спеціалізованих платформ електронного документообігу та корпоративних інформаційних систем, електронна пошта продовжує використовуватися як універсальний канал отримання документів і повідомлень, оскільки вона є доступною, простою у використанні та підтримується більшістю організацій [1].

Практична задача інженерії програмного забезпечення, що розглядається в цій кваліфікаційній роботі, полягає в автоматизації процесу отримання, фільтрації та обробки повідомлень електронної пошти й вкладених файлів. У багатьох організаціях такий процес виконується вручну: відповідальний працівник відкриває поштову скриньку, переглядає нові повідомлення, визначає, чи є лист важливим, перевіряє наявність вкладень, завантажує файли, перейменовує їх, зберігає у визначений каталог або передає до іншої інформаційної системи. У разі великої кількості повідомлень такі операції стають повторюваними, трудомісткими та схильними до помилок.

Комплексність цієї задачі зумовлена тим, що програмне забезпечення повинно одночасно взаємодіяти з декількома різнорідними компонентами: поштовими серверами, базою даних, файловою системою, зовнішніми API та графічним інтерфейсом користувача. Крім того, необхідно забезпечити коректну обробку вкладень, захист облікових даних, запобігання повторній обробці одних і тих самих листів, збереження журналу подій і обробку помилок, які можуть виникати під час мережевої взаємодії або роботи із зовнішніми ресурсами.

Невизначеність умов полягає в тому, що електронні повідомлення можуть мати різних відправників, різну структуру теми й тексту, різні формати вкладень, непередбачувані назви файлів, а також можуть надходити нерівномірно в часі. Крім того, зовнішні сервіси можуть бути тимчасово недоступними, поштові сервери можуть повертати помилки підключення, а файли можуть мати некоректні або потенційно небезпечні назви й розширення.

Існують програмні засоби, які частково розв’язують подібні задачі. До них можна віднести правила Microsoft Outlook [2], фільтри Gmail [3], сервіси Microsoft Power Automate [4] і Zapier [5], а також спеціалізовані засоби Mailparser [6] та Parseur [7]. Microsoft Outlook Rules і Gmail Filters дозволяють виконувати базову автоматизацію роботи з електронною поштою: сортувати листи, переміщувати їх до папок, позначати повідомлення або застосовувати прості умови фільтрації. Microsoft Power Automate і Zapier надають ширші можливості автоматизації, оскільки дозволяють створювати сценарії взаємодії між електронною поштою, хмарними сховищами, таблицями та зовнішніми сервісами. Mailparser і Parseur орієнтовані на вилучення структурованих даних із повідомлень електронної пошти та вкладень, що може бути корисним для обробки рахунків, заявок або типових формалізованих листів. Однак зазначені аналоги не повністю відповідають потребам.

Тому доцільною є розробка власного програмного забезпечення для автоматичної обробки повідомлень електронної пошти. Такий підхід дозволяє реалізувати функції, безпосередньо пов’язані з поставленою практичною задачею: налаштування декількох поштових скриньок, підключення за протоколом IMAP [8], фільтрацію листів за правилами, обробку вкладених файлів, збереження результатів у базі даних, передавання файлів або метаданих до зовнішнього API та журналювання всіх значущих подій. Крім того, власне програмне забезпечення дає змогу краще контролювати структуру даних, порядок обробки повідомлень, правила безпеки та подальший розвиток системи.

Метою кваліфікаційної роботи є розробка програмного забезпечення для автоматичної обробки повідомлень електронної пошти, яке забезпечує

отримання листів, перевірку їх за правилами, обробку вкладень, виконання налаштованих дій і контроль результатів через журнал подій.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– проаналізувати практичну задачу автоматизації процесу обробки повідомлень електронної пошти та вкладених файлів;

– розглянути існуючі програмні засоби й сервіси, які можуть використовуватися для роботи з електронною поштою та автоматизації обробки повідомлень;

– обґрунтувати доцільність розробки власного програмного забезпечення з урахуванням переваг і обмежень наявних аналогів;

– визначити функціональні та нефункціональні вимоги до програмного забезпечення;

– сформувати основні сценарії використання системи та побудувати модель варіантів використання;

– розробити архітектуру програмного забезпечення з урахуванням шаблону MVVM і багатошарового підходу;

– розробити проєкт програмного забезпечення, зокрема структуру класів, основні сервіси, модель бази даних і алгоритм обробки повідомлень;

– реалізувати програмне забезпечення з використанням C#, WPF, Entity Framework Core, Microsoft SQL Server, MailKit та засобів взаємодії із зовнішніми API;

– забезпечити механізми захисту облікових даних, безпечної обробки вкладень і запобігання повторній обробці повідомлень;

– підготувати програму та методику випробувань програмного забезпечення;

– виконати перевірку працездатності розробленого застосунку та проаналізувати отримані результати.

Об’єктом кваліфікаційної роботи є процес розробки програмного забезпечення для автоматичної обробки повідомлень електронної пошти.

1 АНАЛІЗ ПРАКТИЧНОЇ ЗАДАЧІ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ АВТОМАТИЧНОЇ ОБРОБКИ ПОВІДОМЛЕНЬ ЕЛЕКТРОННОЇ ПОШТИ

1.1 Аналіз практичної задачі інженерії програмного забезпечення

Електронна пошта є одним із найбільш поширених засобів обміну інформацією між окремими користувачами, організаціями, підприємствами та програмними системами. Незважаючи на розвиток месенджерів, корпоративних порталів, CRM-систем і спеціалізованих сервісів електронного документообігу, електронна пошта продовжує активно використовуватися для передавання службових повідомлень, документів, звітів, заявок, рахунків, договорів, актів виконаних робіт, повідомлень від автоматизованих сервісів та інших даних

У багатьох організаціях електронна пошта фактично виконує роль одного з основних каналів вхідної інформації. Через поштові скриньки надходять повідомлення від клієнтів, постачальників, партнерів, внутрішніх підрозділів, державних сервісів, банківських установ та інших джерел. Значна частина таких повідомлень містить вкладені файли, які необхідно зберегти, передати до іншої системи, зареєструвати, перевірити або використати у подальших бізнес-процесах.

Практична задача, що розглядається у межах даної роботи, пов’язана з автоматизацією процесу обробки вхідних повідомлень електронної пошти. Її сутність полягає в тому, що користувач або відповідальний працівник не повинен вручну переглядати всі нові листи, шукати серед них потрібні повідомлення, завантажувати вкладення, перевіряти їх назви або розширення, зберігати файли у відповідні каталоги та передавати їх до зовнішніх інформаційних систем. Замість цього такі операції можуть виконуватися автоматично на основі заздалегідь визначених правил.

Ручна обробка електронної пошти є прийнятною лише за умови невеликої

кількості повідомлень. Якщо ж організація щоденно отримує десятки або сотні листів, серед яких лише частина потребує подальшої обробки, цей процес стає трудомістким і ненадійним. Працівник повинен постійно контролювати поштову скриньку, відокремлювати важливі повідомлення від другорядних, відкривати кожен лист, перевіряти відправника, тему, зміст повідомлення та наявність вкладень. Після цього необхідно вручну завантажити потрібні файли, зберегти їх у правильне місце, іноді перейменувати відповідно до прийнятого формату, а також передати інформацію про отриманий документ до іншої системи або відповідального підрозділу.

Такий підхід має низку суттєвих недоліків. По-перше, він потребує значних витрат часу, особливо у випадках, коли повідомлення надходять регулярно протягом робочого дня. По-друге, результат обробки залежить від уважності конкретного працівника, що створює ризик людських помилок. Працівник може пропустити важливий лист, завантажити не те вкладення, зберегти файл у неправильний каталог, випадково обробити один і той самий лист декілька разів або не зафіксувати факт виконання операції. По-третє, ручна обробка ускладнює контроль за виконаними діями, оскільки не завжди зрозуміло, які саме листи вже були опрацьовані, які вкладення збережено, коли це було зроблено та чи виникали помилки під час обробки.

Окремою проблемою є відсутність єдиного журналу обробки повідомлень. Якщо робота з поштою виконується вручну, інформація про виконані дії може зберігатися фрагментарно або взагалі не фіксуватися. Наприклад, працівник може завантажити файл на локальний комп’ютер або в мережеву папку, але не залишити запису про те, з якого листа було отримано цей файл, хто був відправником, коли повідомлення надійшло та яке правило обробки мало бути застосоване. У разі виникнення помилки, втрати документа або необхідності перевірки історії обробки відновити повну картину подій стає складно.

З огляду на це виникає потреба у створенні програмного забезпечення, яке дозволить автоматизувати роботу з вхідними повідомленнями електронної пошти. Таке програмне забезпечення повинно виконувати не функції звичайного

поштового клієнта, а саме функції автоматичного відбору, аналізу та обробки повідомлень. Його основне призначення полягає в тому, щоб підключатися до поштової скриньки, отримувати нові листи, перевіряти їх відповідність заданим умовам, обробляти вкладення та виконувати визначені користувачем дії.

Предметною галуззю для розробки такого програмного забезпечення можна вважати автоматизацію обробки вхідної електронної кореспонденції та електронних документів в організації. У межах цієї галузі програмна система має забезпечувати підтримку типових сценаріїв, пов’язаних з отриманням листів, відбором повідомлень за ознаками, завантаженням вкладених файлів, їх збереженням, передаванням до зовнішніх сервісів і веденням історії обробки. Такий підхід дозволяє розглядати електронну пошту не лише як засіб комунікації, а як джерело даних для подальшої автоматизованої обробки.

Одним із ключових елементів такої системи є механізм правил. Саме правила визначають, які повідомлення повинні бути оброблені. Умовами відбору можуть бути адреса відправника, тема листа, наявність певних слів у тексті повідомлення, дата отримання, наявність вкладень, назва вкладеного файлу, його розширення або інші параметри.

Другим важливим елементом є модуль обробки вкладень. Більшість практичних сценаріїв автоматизації електронної пошти пов’язана саме з файлами, що надходять разом із листами. Система повинна вміти визначати, які вкладення відповідають заданим критеріям, завантажувати їх, формувати безпечні імена файлів, уникати дублювання, зберігати файли у визначеному каталозі або передавати їх до зовнішнього API. При цьому важливо зберігати інформацію про кожне вкладення: його початкову назву, розширення, розмір, джерело отримання, дату обробки та результат виконаної дії.

Не менш важливою є задача збереження службової інформації в базі даних. Для коректної роботи програмного забезпечення необхідно зберігати налаштування поштових скриньок, правила обробки, параметри дій, відомості про отримані повідомлення, інформацію про вкладення та журнал обробки. Використання бази даних дозволяє уникнути повторної обробки одних і тих

самих листів, забезпечити пошук і фільтрацію записів, аналізувати результати роботи системи та відновлювати історію виконаних операцій.

Важливою складовою практичної задачі є забезпечення надійності обробки. Під час роботи з електронною поштою можуть виникати різні помилки: недоступність поштового сервера, неправильні параметри підключення, відсутність мережі, пошкоджені вкладення, неможливість збереження файлу, помилка під час звернення до зовнішнього API або збій підключення до бази даних. Програмне забезпечення повинно не лише виконувати основні операції, а й коректно реагувати на такі ситуації. Зокрема, необхідно фіксувати помилки в журналі, зберігати статуси обробки, надавати користувачу інформацію про проблеми та запобігати втраті даних.

У контексті інженерії програмного забезпечення дана задача є комплексною, оскільки поєднує декілька напрямів розробки. Необхідно реалізувати взаємодію з поштовим сервером, механізм фільтрації повідомлень, роботу з файловою системою, інтеграцію із зовнішніми API, збереження даних у базі, журналювання подій і користувацький інтерфейс для налаштування системи.

Таким чином, практична задача інженерії програмного забезпечення полягає у створенні спеціалізованого застосунку для автоматизованого отримання, аналізу та обробки електронних повідомлень. На відміну від звичайних поштових клієнтів, таке програмне забезпечення має бути орієнтоване не на ручне листування, а на виконання повторюваних операцій за правилами. Його впровадження є доцільним для організацій, які регулярно працюють із великою кількістю електронних документів і потребують надійного механізму їх автоматичного приймання, збереження, передавання та обліку.

1.2 Огляд програмних засобів автоматизації роботи з електронною поштою та обґрунтування розробки власного рішення

Одним із важливих етапів підготовки до розробки програмного забезпечення є вивчення наявних рішень, які вже використовуються для

розв’язання подібних задач. Такий огляд дозволяє визначити, які функції вже реалізовані в існуючих програмних продуктах, які підходи є найбільш поширеними, а також які обмеження залишаються актуальними для користувачів. У межах даної роботи розглядаються програмні засоби, що пов’язані з автоматизацією роботи з електронною поштою, фільтрацією повідомлень, обробкою вкладень та передаванням даних до інших сервісів.

Задача автоматичної обробки електронної пошти може частково вирішуватися різними типами програмного забезпечення. До них належать поштові клієнти з вбудованими правилами, вебпоштові сервіси з фільтрами, хмарні платформи автоматизації бізнес-процесів, сервіси інтеграції застосунків та спеціалізовані інструменти для аналізу вмісту електронних листів. Однак кожен із цих варіантів має власне призначення, переваги та обмеження. Тому їх необхідно розглядати не лише з погляду наявності окремих функцій, а й з позиції відповідності поставленій задачі — створенню програмного забезпечення, яке може отримувати листи, відбирати їх за правилами, завантажувати вкладення, зберігати результати обробки у базі даних та виконувати подальші дії з файлами.

Одним із найвідоміших програмних засобів для роботи з електронною поштою є Microsoft Outlook. Це поштовий клієнт, який широко використовується як окремими користувачами, так і організаціями. Однією з корисних можливостей Outlook є система правил, за допомогою якої користувач може автоматизувати окремі дії з вхідними повідомленнями. Наприклад, можна створити правило для переміщення листів від певного відправника до конкретної папки, позначення повідомлень, зміни їх важливості або виконання інших стандартних дій.

Перевагою Microsoft Outlook є його поширеність, зручний інтерфейс, інтеграція з екосистемою Microsoft 365 та наявність базових засобів автоматизації поштової кореспонденції. Користувач може створювати правила без написання програмного коду, що робить цей інструмент доступним для широкого кола користувачів. Крім того, Outlook добре підходить для щоденної роботи з листами, календарем, контактами та завданнями.

Разом із тим Outlook не є спеціалізованим засобом для комплексної автоматичної обробки вкладень і передавання їх до зовнішніх інформаційних систем. Його правила переважно орієнтовані на організацію поштової скриньки: переміщення, позначення, пересилання або сортування повідомлень. Якщо необхідно реалізувати складну логіку обробки вкладень, зберігати детальний журнал виконаних дій у власній базі даних, контролювати повторну обробку листів або передавати файли до зовнішнього API, стандартних можливостей Outlook може бути недостатньо. Також використання Outlook як основи для автоматизації залежить від встановленого клієнта, облікового запису користувача та обмежень конкретного поштового середовища.

Іншим поширеним рішенням є Gmail Filters — система фільтрів у поштовому сервісі Gmail. Вона дозволяє створювати правила для автоматичної обробки вхідних повідомлень на основі певних критеріїв. Користувач може налаштувати фільтр за відправником, одержувачем, темою, ключовими словами, наявністю вкладення та іншими ознаками. Після спрацювання фільтра Gmail може застосувати до повідомлення певну дію, наприклад архівувати його, позначити зірочкою, застосувати мітку, видалити або переслати на іншу адресу.

Перевагами Gmail Filters є простота налаштування, доступність через вебінтерфейс, відсутність необхідності встановлювати додаткове програмне забезпечення та можливість швидко організувати поштову скриньку. Для індивідуального користувача або невеликої кількості простих правил цього функціоналу часто достатньо. Фільтри Gmail зручно використовувати для сортування повідомлень, розподілу листів за мітками, автоматичного пересилання або очищення вхідної пошти.

Недоліком Gmail Filters у контексті даної роботи є обмеженість дій, які можна виконати з повідомленнями та вкладеннями. Сервіс не орієнтований на збереження вкладених файлів у локальну або мережеву папку, не забезпечує ведення власного журналу обробки в SQL Server, не надає гнучкого механізму передавання файлів до довільного API та не дозволяє повністю контролювати логіку обробки з боку користувацького застосунку. Крім того, Gmail є хмарним

сервісом, тому його використання залежить від правил і можливостей конкретної платформи, а не від потреб локальної інформаційної системи організації.

Для автоматизації процесів, пов’язаних із поштою, також може використовуватися Microsoft Power Automate. Це хмарна платформа для створення автоматизованих потоків, які запускаються за певними подіями та виконують визначені дії. У контексті електронної пошти Power Automate може реагувати на отримання повідомлення, наявність вкладення, певні властивості листа або інші події. Після цього система може виконувати дії з файлами, надсилати повідомлення, зберігати дані в хмарних сховищах або взаємодіяти з іншими сервісами Microsoft.

Перевагою Power Automate є широка інтеграція з Microsoft 365, велика кількість готових конекторів і можливість будувати сценарії автоматизації без традиційного програмування. Такий підхід є зручним для організацій, які вже використовують хмарну інфраструктуру Microsoft і мають потребу в інтеграції пошти з OneDrive, SharePoint, Teams або іншими сервісами.

Проте для задачі, що розглядається у даній роботі, Power Automate має низку обмежень. По-перше, це хмарне рішення, використання якого залежить від облікового запису, тарифного плану та доступності відповідних конекторів. По-друге, налаштування складних сценаріїв може бути неочевидним для користувачів, які не мають досвіду роботи з подібними платформами. По-третє, у випадку потреби в локальному зберіганні службової інформації в Microsoft SQL Server, гнучкому контролі обробки вкладень і створенні спеціалізованого інтерфейсу під конкретний процес власний застосунок може бути більш доцільним. Power Automate добре підходить для типової хмарної автоматизації, але не завжди є оптимальним рішенням для локальної системи з власною логікою, базою даних і журналюванням.

Ще одним класом рішень є сервіси інтеграції застосунків, наприклад Zapier. Такі платформи дозволяють поєднувати різні вебсервіси між собою та будувати автоматизовані сценарії без розробки повноцінного програмного

забезпечення. У межах таких сценаріїв електронна пошта може використовуватися як джерело події, після чого дані передаються до інших систем: таблиць, CRM, месенджерів, сховищ файлів або API.

Перевагою Zapier та подібних сервісів є велика кількість інтеграцій, швидке створення автоматизацій і відсутність потреби розгортати власну інфраструктуру. Такі платформи зручні тоді, коли потрібно швидко зв’язати декілька популярних вебсервісів і виконувати прості повторювані дії.

Недоліки таких рішень пов’язані передусім із залежністю від зовнішньої платформи. Користувач обмежений доступними можливостями сервісу, тарифним планом, правилами обробки даних і підтримуваними інтеграціями. Крім того, передавання службових документів або вкладень через сторонню хмарну платформу може бути небажаним для організацій, які працюють із конфіденційною інформацією. Також подібні сервіси не завжди забезпечують потрібний рівень деталізації журналу, локальне зберігання інформації в SQL Server і повну адаптацію інтерфейсу під конкретну предметну галузь.

Окремо варто розглянути спеціалізовані сервіси для аналізу електронних листів, такі як Mailparser або Parseur. Їх основне призначення полягає у витягуванні структурованих даних із вхідних повідомлень. Такі інструменти можуть аналізувати текст листа, виділяти потрібні поля, працювати з шаблонами та передавати отримані дані до інших систем. Вони можуть бути корисними для обробки замовлень, заявок, повідомлень із вебформ або однотипних листів, які мають передбачувану структуру.

Перевагою таких систем є спеціалізація саме на поштових повідомленнях. Вони дозволяють автоматизувати розбір листів, виділення потрібної інформації та інтеграцію з іншими сервісами. Для деяких задач це може бути ефективним рішенням, особливо якщо основна мета полягає не у збереженні вкладень, а у витягуванні текстових даних із повідомлень.

Водночас такі сервіси не повністю відповідають задачі, яка розглядається в даній роботі. Розроблюване програмне забезпечення повинно бути орієнтоване не лише на аналіз тексту листів, а й на отримання повідомлень із поштової

скриньки, фільтрацію за правилами, завантаження вкладень, збереження файлів, передавання їх через API та ведення локального журналу обробки. Крім того, використання сторонніх сервісів може потребувати регулярної оплати, підключення до хмарної інфраструктури та передавання даних за межі локального середовища організації.

Порівняння розглянутих програмних засобів показує, що кожне з них вирішує лише частину поставленої задачі. Microsoft Outlook і Gmail добре підходять для базового сортування пошти, однак їх можливості щодо обробки вкладень, інтеграції з власною базою даних і реалізації спеціалізованих сценаріїв є обмеженими. Power Automate та Zapier надають ширші засоби автоматизації, але залежать від хмарних сервісів, тарифних планів і готових конекторів. Mailparser і Parseur орієнтовані на витягування даних із листів, проте не є повноцінними локальними застосунками для керування процесом обробки повідомлень і вкладень.

Для задачі автоматичної обробки повідомлень електронної пошти важливими є такі вимоги: можливість підключення до поштової скриньки, отримання нових повідомлень, перевірка листів за правилами, завантаження вкладень, збереження інформації в базі даних, передавання файлів до зовнішнього API, контроль статусів обробки та ведення журналу. Також важливо, щоб користувач міг самостійно налаштовувати правила відбору повідомлень і дії, які необхідно виконувати після їх спрацювання.

Наявні програмні рішення не забезпечують повного поєднання зазначених можливостей у вигляді окремого настільного застосунку, орієнтованого на роботу з Microsoft SQL Server, бібліотекою MailKit [9] і локальною обробкою вкладень. Саме тому доцільною є розробка власного програмного забезпечення, яке буде враховувати специфіку поставленої задачі та дозволить реалізувати потрібний набір функцій без надлишкової залежності від сторонніх хмарних платформ.

Запропоноване програмне забезпечення має поєднувати функції поштового модуля, механізму правил, обробника вкладень, модуля інтеграції з

API та журналу обробки. На відміну від стандартних поштових клієнтів, воно буде спрямоване не на ручне листування, а на автоматизоване виконання повторюваних операцій. На відміну від хмарних сервісів автоматизації, розроблюваний застосунок може забезпечити локальне зберігання налаштувань і результатів у SQL Server, гнучку адаптацію логіки під вимоги користувача та контроль за всіма етапами обробки.

Отже, результати огляду існуючих програмних засобів підтверджують доцільність створення власного програмного забезпечення для автоматичної обробки повідомлень електронної пошти. Розробка такого застосунку дозволить реалізувати спеціалізоване рішення, яке забезпечує отримання листів, фільтрацію повідомлень, обробку вкладень, інтеграцію із зовнішніми сервісами та збереження повної історії виконаних дій у базі даних.

1.3 Постановка задачі розробки програмного забезпечення для автоматичної обробки електронних повідомлень

На основі аналізу практичної задачі інженерії програмного забезпечення та огляду наявних програмних засобів можна зробити висновок, що автоматизація обробки електронної пошти є актуальною задачею для організацій, які регулярно отримують значну кількість однотипних повідомлень і вкладених файлів. Існуючі рішення дозволяють частково автоматизувати роботу з листами, однак не завжди забезпечують поєднання таких можливостей, як локальне зберігання службової інформації, гнучке налаштування правил, обробка вкладень, передавання файлів до зовнішніх API та ведення повного журналу результатів.

У межах кваліфікаційної роботи необхідно розробити програмне забезпечення для автоматичної обробки повідомлень електронної пошти. Розроблюваний застосунок має забезпечувати підключення до поштової скриньки, отримання нових повідомлень, перевірку листів за встановленими користувачем правилами, завантаження потрібних вкладень, збереження файлів у визначене місце або їх передавання до зовнішньої інформаційної системи через

API. Крім того, система повинна зберігати інформацію про оброблені листи, вкладення, правила та результати виконаних дій у базі даних Microsoft SQL Server.

Застосунок орієнтується на роботу з поштовими скриньками, що підтримують протокол IMAP. Обробка повідомлень виконується лише за тими правилами, які були попередньо налаштовані користувачем. Система не призначена для повної заміни поштового клієнта, оскільки її основна функція полягає не в листуванні, а в автоматичному відборі й обробці повідомлень. Програмне забезпечення передбачається реалізувати як застосунок для операційної системи Windows із використанням мови програмування C#, бібліотеки MailKit для роботи з електронною поштою та Microsoft SQL Server для зберігання службових даних.

У межах розробки не передбачається реалізація складного аналізу змісту документів, розпізнавання тексту у вкладеннях або автоматичної класифікації листів за допомогою методів машинного навчання. Основна увага приділяється правиловій обробці повідомлень, тобто перевірці листів за заданими умовами та виконанню відповідних дій у разі спрацювання правила. Такий підхід є достатнім для більшості типових сценаріїв, коли повідомлення мають відомі ознаки: конкретного відправника, характерну тему, певний формат вкладень або повторювану структуру.

Програмне забезпечення повинно забезпечувати виконання таких основних функцій:

– додавання, редагування, перегляд і видалення поштових облікових записів;

– налаштування параметрів підключення до поштового сервера, зокрема адреси сервера, порту, логіна, пароля, типу захищеного з’єднання та папки для перевірки;

– ручний запуск перевірки поштової скриньки користувачем;

– автоматичну перевірку поштової скриньки з визначеним інтервалом;

– отримання нових повідомлень із поштової скриньки;

– збереження основної інформації про повідомлення в базі даних;

– створення, редагування, перегляд і видалення правил обробки повідомлень;

– фільтрацію повідомлень за відправником, темою, датою отримання, наявністю вкладень та параметрами вкладених файлів;

– завантаження вкладень із повідомлень, які відповідають заданим правилам;

– перевірку вкладень за назвою, розширенням і розміром;

– збереження вкладень у визначену користувачем папку;

– передавання вкладень або даних повідомлення до зовнішнього API;

– фіксацію результатів обробки в журналі;

– перегляд списку отриманих та оброблених повідомлень;

– перегляд списку завантажених вкладень;

– пошук і фільтрацію записів журналу за датою, відправником, темою, правилом або статусом обробки;

– відображення повідомлень про помилки, що виникають під час підключення до пошти, збереження файлів, звернення до API або роботи з базою даних.

Для роботи програмного забезпечення необхідно передбачити такі вимоги до складу й параметрів технічних засобів:

– операційна система Windows 10 або новіша;

– процесор із тактовою частотою не менше 2,0 ГГц;

– оперативна пам’ять не менше 4 ГБ;

– не менше 500 МБ вільного місця на диску для встановлення застосунку та збереження службових файлів;

– доступ до мережі Інтернет або локальної мережі для підключення до поштового сервера та зовнішніх API;

– доступ до Microsoft SQL Server для зберігання даних застосунку;

– наявність поштової скриньки з підтримкою протоколу IMAP.

Вимоги до інформаційної та програмної сумісності визначаються

технологіями, які планується використовувати під час реалізації застосунку. Програмне забезпечення повинно бути сумісним з операційними системами сімейства Windows, платформою .NET, базою даних Microsoft SQL Server, поштовими серверами, що підтримують протокол IMAP, а також зовнішніми сервісами, які приймають HTTP-запити через API. Для взаємодії з поштовими серверами доцільно використовувати бібліотеку MailKit, яка забезпечує програмний доступ до електронної пошти та підтримує отримання повідомлень і вкладень.

Інформаційна модель програмного забезпечення повинна включати основні сутності предметної області. До них належать користувач, поштова скринька, електронне повідомлення, вкладення, правило обробки, дія та журнал обробки. Поштова скринька є джерелом електронних повідомлень. Одне повідомлення може містити одне або декілька вкладень. Правило обробки визначає умови, за яких повідомлення або його вкладення мають бути опрацьовані. Дія визначає, що саме необхідно виконати після спрацювання правила: зберегти файл, передати його через API або зафіксувати результат у журналі. Журнал обробки містить інформацію про виконані операції, їх статус і можливі помилки.

Бізнес-процес роботи системи можна описати такою послідовністю дій. Спочатку користувач налаштовує поштову скриньку, вводить параметри підключення та визначає папку, з якої необхідно отримувати повідомлення. Далі користувач створює правила обробки, у яких зазначає умови відбору листів і параметри роботи з вкладеннями. Після запуску перевірки система підключається до поштового сервера, отримує нові повідомлення та перевіряє їх відповідність заданим правилам. Якщо повідомлення відповідає умовам, система аналізує його вкладення, завантажує потрібні файли та виконує налаштовані дії. Після завершення обробки результат фіксується в журналі, а користувач може переглянути інформацію про оброблені повідомлення, вкладення та можливі помилки.

Основними варіантами використання програмного забезпечення є: налаштування поштової скриньки, створення правила обробки, редагування правила, ручний запуск перевірки пошти, автоматична перевірка поштової скриньки, перегляд отриманих повідомлень, перегляд завантажених вкладень, налаштування дії збереження файлів, налаштування передавання даних до API, перегляд журналу обробки та аналіз помилок. Головним користувачем системи є відповідальна особа або технічний спеціаліст, який налаштовує параметри роботи застосунку та контролює результати автоматичної обробки.

Отже, постановка задачі полягає в необхідності розробити програмне забезпечення для автоматичної обробки повідомлень електронної пошти, яке забезпечує отримання листів із поштової скриньки, їх фільтрацію за правилами, обробку вкладень, виконання подальших дій із файлами та збереження результатів роботи в базі даних. Детальний опис вимог до програмного забезпечення, його призначення, функціональних можливостей, вхідних і вихідних даних, вимог до технічних засобів та порядку приймання доцільно оформити у вигляді документа «Технічне завдання» і розмістити в додатку до звіту з переддипломної практики.

Детальні вимоги до програмного забезпечення наведені у Додатку А – Технічне завдання.

2 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ АВТОМАТИЧНОЇ ОБРОБКИ ПОВІДОМЛЕНЬ ЕЛЕКТРОННОЇ ПОШТИ

2.1 Аналіз вимог до програмного забезпечення

Розроблюване програмне забезпечення призначене для використання в організаціях, де електронна пошта є одним із основних каналів отримання документів, звітів, рахунків, заявок та інших службових файлів. У таких умовах користувачеві необхідно не просто переглядати листи, а регулярно виконувати повторювані дії: знаходити потрібні повідомлення за відправником або темою, перевіряти наявність вкладень, завантажувати файли, зберігати їх у визначене місце, передавати до зовнішніх інформаційних систем і контролювати результат виконання цих операцій. Тому основною вимогою до системи є забезпечення автоматичного виконання таких дій на основі попередньо налаштованих правил.

Головним користувачем програмного забезпечення є відповідальна особа, адміністратор або працівник організації, який налаштовує роботу системи та контролює результати обробки. Такий користувач повинен мати можливість додавати поштові облікові записи, задавати параметри підключення до IMAP-сервера, створювати правила відбору повідомлень, визначати дії після спрацювання правил, запускати обробку вручну або використовувати автоматичний режим, переглядати журнал подій і аналізувати помилки. При цьому система повинна бути зрозумілою для користувача, який не є розробником програмного забезпечення, але має базові навички роботи з комп’ютером і електронною поштою.

Для кращого розуміння функціональних можливостей програмного забезпечення було сформовано модель варіантів використання (рис. 2.1). Вона відображає взаємодію користувача із системою, а також основні сценарії, які повинні бути підтримані під час розробки.

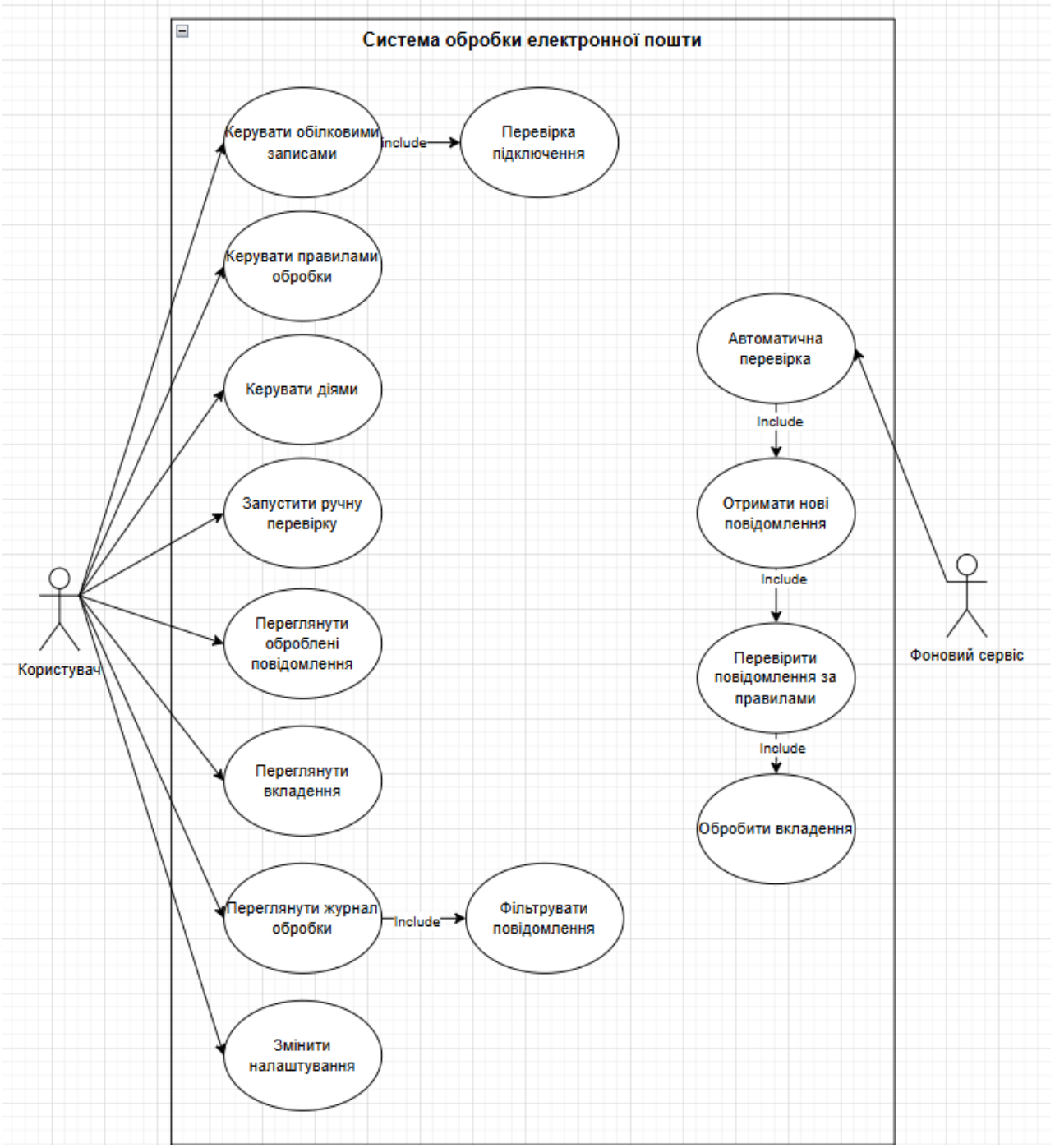


Рисунок 2.1 – Діаграма варіантів використання програмного забезпечення

Користувач має можливість налаштовувати поштові облікові записи, перевіряти підключення до поштового сервера, створювати та редагувати правила обробки, налаштовувати дії, запускати ручну перевірку пошти, переглядати отримані повідомлення, переглядати вкладення, аналізувати журнал обробки та змінювати загальні параметри системи. Окремо можна показати фоновий процес, який автоматично опитує активні поштові скриньки

відповідно до заданих інтервалів.

У межах проєкту визначено такі основні функціональні сценарії роботи системи. Перший сценарій пов’язаний із налаштуванням поштової скриньки. Користувач вводить назву облікового запису, адресу електронної пошти, IMAP-сервер, порт, логін, пароль або пароль застосунку, вибирає папку для моніторингу та задає інтервал перевірки. Після збереження параметрів система записує їх до бази даних, при цьому пароль зберігається не у відкритому вигляді, а в зашифрованому форматі.

Другий сценарій пов’язаний зі створенням правила обробки. Користувач задає назву правила, його активність, пріоритет і умови спрацювання. До таких умов належать адреса або частина адреси відправника, фрагмент теми листа, фрагмент тексту повідомлення, діапазон дат, вимога щодо наявності вкладень, список допустимих розширень файлів та фрагмент назви вкладення. Система використовує ці параметри під час перевірки отриманих електронних повідомлень.

Третій сценарій полягає в налаштуванні дій, які виконуються після спрацювання правила. Одне правило може мати одну або декілька пов’язаних дій. До підтримуваних типів дій належать збереження вкладення у визначену папку, передавання вкладення до зовнішнього API, передавання метаданих повідомлення до API або переміщення листа в іншу папку поштової скриньки. Завдяки такому розділенню правило визначає умови обробки, а дії визначають, що саме необхідно виконати після знаходження відповідного повідомлення.

Четвертий сценарій описує автоматичну обробку електронної пошти. Фоновий сервіс перевіряє, які активні поштові скриньки потребують опитування відповідно до заданих інтервалів. Для кожної такої скриньки запускається повний цикл обробки: отримання нових повідомлень, перевірка їх за правилами, завантаження вкладень, виконання дій, оновлення статусів і запис результатів у журнал. Якщо повідомлення не відповідає жодному правилу, воно може бути позначене як пропущене, а відповідна інформація фіксується у журналі.

П’ятий сценарій пов’язаний із переглядом результатів роботи системи.

Користувач може переглядати список оброблених повідомлень, вкладень і журнал подій. Для журналу передбачено фільтрацію за датою, поштовим обліковим записом, відправником, темою, правилом і статусом. Це дозволяє швидко знаходити помилки, перевіряти факт обробки конкретного повідомлення та аналізувати загальний стан роботи системи.

На основі наведених сценаріїв можна сформулювати функціональні вимоги до програмного забезпечення. Система повинна забезпечувати створення, редагування, перегляд і видалення поштових облікових записів; створення та редагування правил обробки; налаштування дій; автоматичне й ручне отримання поштових повідомлень; фільтрацію листів за заданими умовами; обробку вкладень; збереження файлів; передавання даних до зовнішнього API; ведення журналу; перегляд результатів обробки та фільтрацію записів журналу.

Крім функціональних вимог, необхідно врахувати вимоги до надійності. Програмне забезпечення повинно коректно реагувати на помилки підключення до поштового сервера, неправильні облікові дані, недоступність зовнішнього API, помилки збереження файлів, проблеми з базою даних і некоректні вкладення. У разі виникнення помилки система не повинна завершувати роботу аварійно. Помилка має бути зафіксована в журналі, а користувач повинен мати можливість переглянути її причину.

Важливою є вимога щодо захисту даних. Поштові паролі не повинні зберігатися у відкритому вигляді. Для цього передбачається використання механізму шифрування облікових даних. Також система повинна виконувати перевірку назв вкладень перед збереженням, не дозволяти автоматичний запуск завантажених файлів і блокувати потенційно небезпечні розширення. Це необхідно, оскільки вкладення надходять із зовнішнього джерела — електронної пошти — і не можуть вважатися повністю довіреними.

До вимог інформаційної сумісності належить підтримка поштових серверів, які працюють за протоколом IMAP, можливість взаємодії із зовнішніми сервісами через HTTP API, використання файлової системи Windows для

збереження вкладень і зберігання службових даних у Microsoft SQL Server. Програмне забезпечення має працювати на платформі .NET і використовувати бібліотеку MailKit для взаємодії з поштовими серверами.

До вимог технічної сумісності належить можливість роботи застосунку в середовищі Windows 10 або новішої версії, на комп'ютері з доступом до мережі Інтернет або локальної мережі, а також з доступом до сервера бази даних Microsoft SQL Server. Оскільки застосунок виконує фонову перевірку пошти, він повинен мати стабільне мережеве підключення для взаємодії з поштовими серверами та зовнішніми API.

Таким чином, аналіз вимог показав, що розроблюване програмне забезпечення повинно поєднувати функції налаштування поштових облікових записів, правилової обробки повідомлень, роботи з вкладеннями, інтеграції із зовнішніми API, журналювання та фонового виконання операцій.

2.2 Розробка архітектури програмного забезпечення

Розроблюване програмне забезпечення призначене для автоматичної обробки повідомлень електронної пошти та вкладених файлів. Застосунок має забезпечувати підключення до поштових скриньок, отримання нових повідомлень, перевірку листів за налаштованими правилами, обробку вкладень, виконання дій із файлами, передавання даних до зовнішніх API та ведення журналу результатів обробки. З огляду на це архітектура програмного забезпечення повинна забезпечувати розділення відповідальності між окремими частинами системи, можливість подальшого розширення функціональності, зручність тестування та підтримки програмного коду.

Під час розробки програмного забезпечення було обрано архітектурний підхід, що поєднує шаблон MVVM для побудови інтерфейсу користувача та багатошарову організацію програмної системи. Такий підхід є доцільним для WPF-застосунків, оскільки дозволяє відокремити графічний інтерфейс від логіки обробки даних і бізнес-операцій. У результаті зміни в інтерфейсі користувача не потребують суттєвого переписування сервісів обробки електронної пошти, а

зміни в механізмах отримання листів, збереження вкладень або журналювання не впливають безпосередньо на структуру вікон застосунку.

Шаблон MVVM передбачає поділ інтерфейсної частини програми на три складові: View, ViewModel і Model (рис. 2.2) [10]. View відповідає за відображення даних і взаємодію з користувачем. У розроблюваному застосунку до цього рівня належать вікна та сторінки для перегляду панелі керування, налаштування поштових облікових записів, створення правил і дій, перегляду вкладень, журналу обробки та параметрів системи. ViewModel містить стан відповідного екрана, команди користувача та логіку підготовки даних для відображення. Model представлена сутностями предметної області, такими як поштова скринька, правило обробки, повідомлення, вкладення та запис журналу.

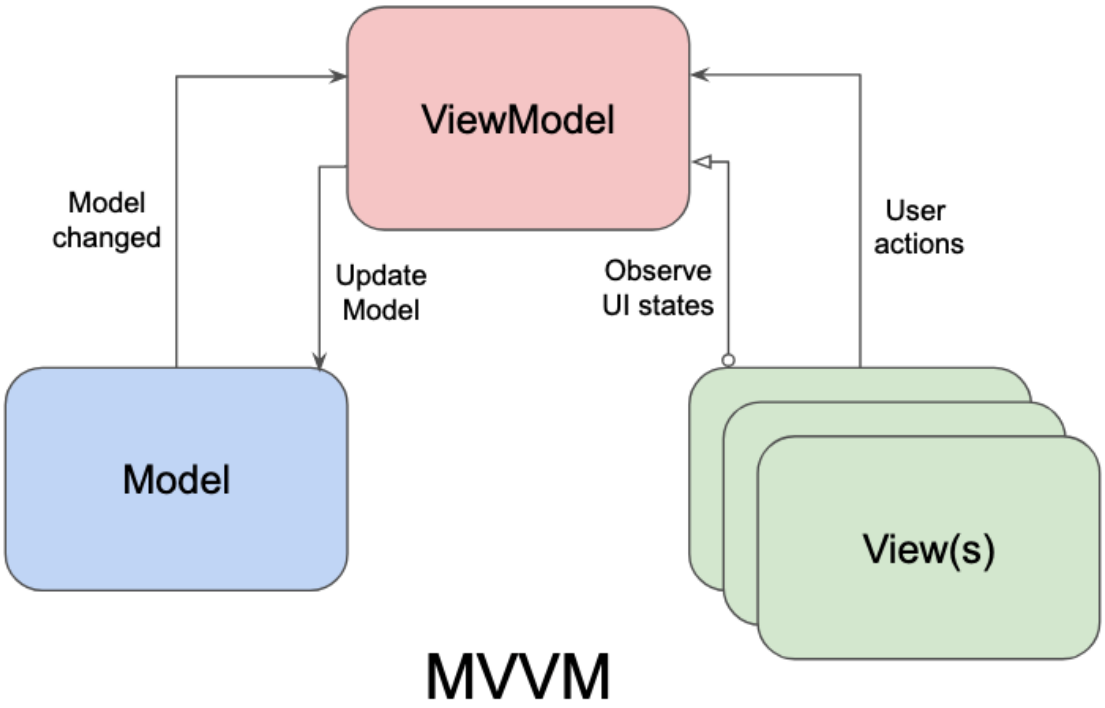


Рисунок 2.2 – Архітектура шаблону MVVM [10]

Використання MVVM дозволяє уникнути розміщення бізнес-логіки безпосередньо у графічних елементах інтерфейсу. Наприклад, натискання

кнопки ручної перевірки пошти в інтерфейсі не повинно самостійно виконувати підключення до IMAP-сервера або обробляти вкладення. Замість цього ViewModel викликає відповідний сервіс або оркестратор, який уже виконує необхідний сценарій. Завдяки цьому інтерфейс залишається відповідальним лише за взаємодію з користувачем, а основна логіка системи зосереджується в сервісному шарі. Це дозволяє побудувати загальну архітектуру програмного забезпечення, яку показано на рисунку 2.3.

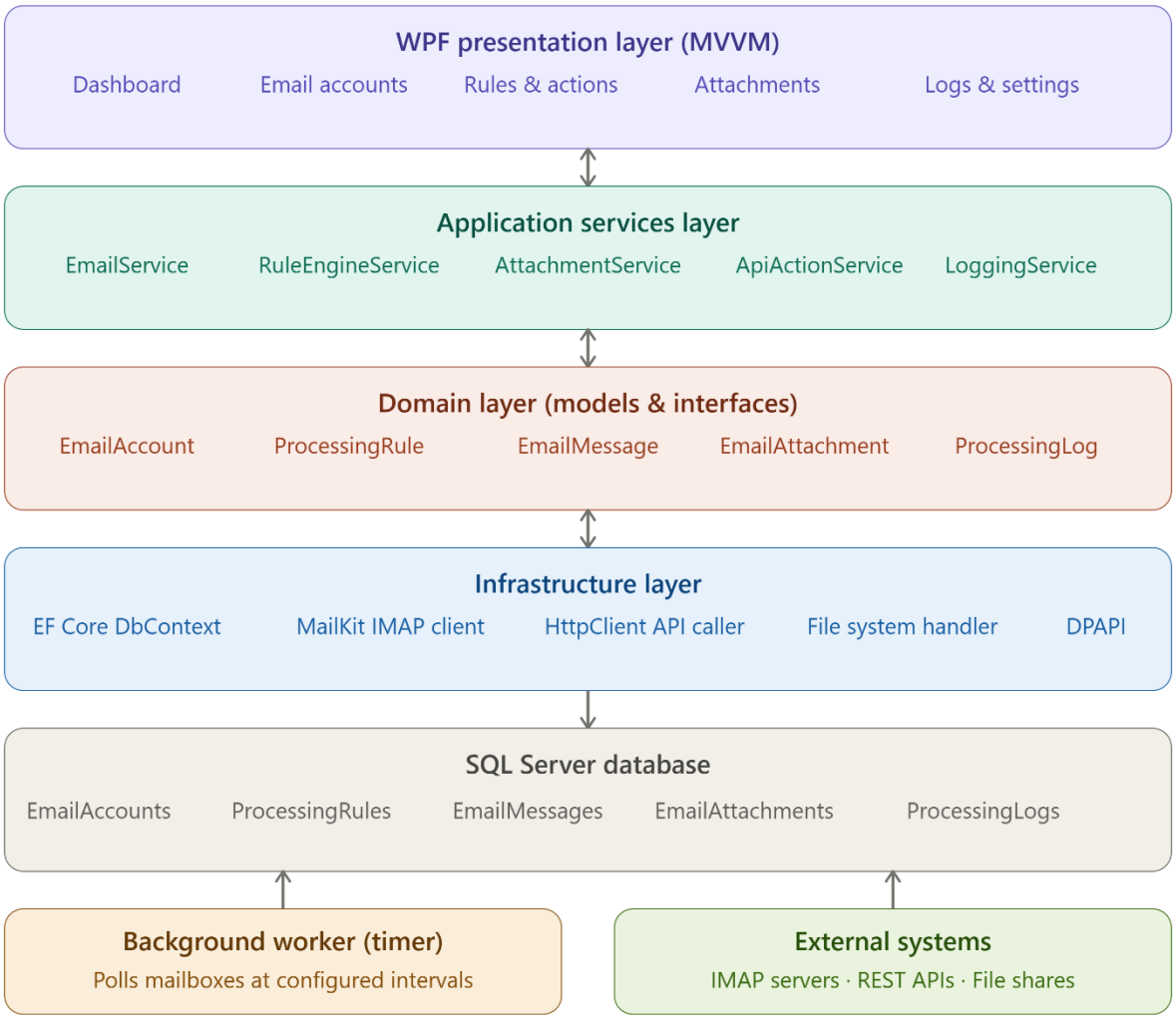


Рисунок 2.3 – Загальна архітектура WPF-застосунку для автоматичної обробки електронної пошти

На рисунку 2.3 показано, що програмне забезпечення має багатошарову

структуру. Верхній рівень становить WPF presentation layer, який реалізує інтерфейс користувача на основі шаблону MVVM. До нього належать основні екрани застосунку: Dashboard, Email accounts, Rules, Actions, Attachments, Logs та Settings. Цей рівень забезпечує введення параметрів, перегляд результатів обробки, запуск окремих операцій користувачем і відображення поточного стану системи.

Нижче розташований application services layer, який містить сервіси прикладної логіки. До нього належать EmailService, RuleEngineService, AttachmentService, ApiActionService і LoggingService. Цей шар відповідає за виконання основних дій системи: отримання повідомлень із поштової скриньки, перевірку повідомлень за правилами, обробку вкладених файлів, виконання HTTP-запитів до зовнішніх API та запис результатів роботи до журналу. Саме сервісний шар є центральним для реалізації бізнес-процесу автоматичної обробки електронної пошти.

Наступним рівнем є domain layer, який містить моделі предметної області та інтерфейси. До основних доменних сутностей належать EmailAccount, ProcessingRule, EmailMessage, EmailAttachment і ProcessingLog. Ці класи описують ключові об’єкти системи: поштову скриньку, правило обробки, отримане повідомлення, вкладення та запис журналу. Доменний шар використовується як основа для передачі даних між сервісами, збереження інформації в базі даних і відображення даних в інтерфейсі користувача.

Infrastructure layer забезпечує взаємодію з технічними засобами та зовнішніми ресурсами. До цього рівня належать EF Core DbContext для роботи з базою даних, MailKit IMAP client для підключення до поштових серверів, HttpClient API caller для звернення до зовнішніх REST API, file system handler для збереження вкладень у файловій системі та DPAPI для захищеного зберігання паролів. Завдяки виділенню інфраструктурного шару прикладна логіка не залежить безпосередньо від конкретних технічних механізмів і може бути змінена або розширена з мінімальним впливом на інші частини системи.

Окремо в архітектурі виділено SQL Server database, яка використовується

для зберігання службової інформації застосунку. У базі даних зберігаються поштові облікові записи, правила обробки, отримані повідомлення, вкладення, дії та журнали обробки. Використання бази даних дозволяє забезпечити збереження історії роботи системи, уникнути повторної обробки одних і тих самих листів, виконувати пошук і фільтрацію журналу, а також зберігати параметри конфігурації застосунку.

Фонову автоматизацію роботи системи забезпечує background worker. Він періодично перевіряє активні поштові скриньки відповідно до налаштованих інтервалів. Кожна поштова скринька має власний інтервал опитування, тому система може одночасно працювати з кількома обліковими записами, не перевіряючи їх усі з однаковою частотою. Крім автоматичного режиму, застосунок також передбачає можливість ручного запуску перевірки пошти користувачем.

Зовнішні системи представлені поштовими серверами, REST API та файловими ресурсами. Поштові сервери використовуються як джерело електронних повідомлень, REST API — як засіб передавання вкладень або метаданих листа до інших інформаційних систем, а файлові ресурси — як місце збереження вкладених файлів. Таким чином, розроблюване програмне забезпечення виступає проміжною ланкою між електронною поштою, файловою системою, базою даних і зовнішніми сервісами.

Для організації взаємодії між основними модулями застосунку використано принцип інверсії залежностей. Основні компоненти системи працюють не з конкретними реалізаціями сервісів, а з їхніми інтерфейсами. Наприклад, обробка електронної пошти виконується через інтерфейс IEmailService, перевірка правил — через IRuleEngine, обробка вкладень — через IAttachmentService, виконання API-запитів — через IApiActionService, журналювання — через ILoggingService. Такий підхід дозволяє зменшити зв’язаність між класами, спростити тестування та забезпечити можливість заміни окремих реалізацій без зміни основної логіки застосунку.

Основною реалізацією сервісу роботи з поштою є MailKitEmailService. Він

використовує бібліотеку MailKit для підключення до IMAP-сервера, отримання повідомлень, позначення листів як прочитаних і переміщення повідомлень між папками. При цьому інші частини системи не залежать від MailKit напряму, оскільки працюють через інтерфейс IEmailService. У майбутньому це дозволяє замінити механізм отримання листів, наприклад на інший поштовий протокол або сервіс, без зміни інтерфейсу користувача, механізму правил чи структури бази даних.

RuleEngineService відповідає за визначення того, чи підходить отримане повідомлення під налаштовані користувачем правила. Правила перевіряються в порядку пріоритету, а перше правило, яке відповідає умовам, визначає подальшу обробку повідомлення. До умов можуть належати адреса відправника, тема листа, вміст тіла повідомлення, діапазон дат, наявність вкладень, розширення файлів і частина назви вкладення. Такий підхід дозволяє реалізувати гнучку обробку поштових повідомлень без необхідності змінювати програмний код для кожного нового сценарію.

AttachmentService відповідає за обробку вкладених файлів. Він перевіряє вкладення, виконує захисну фільтрацію небезпечних розширень, формує безпечні імена файлів, зберігає вкладення у визначену папку або передає їх до інших сервісів для подальшої обробки. Важливою частиною цього сервісу є запобігання перезапису файлів з однаковими назвами та заборона обробки потенційно небезпечних файлів, таких як виконувані або скриптові файли. Це підвищує безпеку роботи із вкладеннями, отриманими з електронної пошти.

ApiActionService реалізує інтеграцію із зовнішніми інформаційними системами через HTTP-запити. Він формує запити на основі налаштувань дії, додає необхідні заголовки, токени авторизації, метадані повідомлення та, за потреби, вкладений файл. Залежно від конфігурації дія може передавати вкладення як multipart/form-data або надсилати службові дані повідомлення у форматі JSON. Це дозволяє використовувати застосунок не лише для локального збереження файлів, а й для автоматичної передачі інформації в інші програмні системи.

LoggingService відповідає за збереження журналу обробки. У журналі фіксуються успішні операції, попередження та помилки, що виникають під час підключення до пошти, перевірки правил, збереження вкладень, звернення до API або роботи з базою даних. Журнал містить контекст події: поштовий обліковий запис, відправника, тему листа, правило, вкладення, виконану дію, статус і текст помилки. Наявність такого журналу є важливою для контролю роботи системи, аналізу помилок і підтвердження факту обробки повідомлень.

Окремим компонентом архітектури є EmailProcessingOrchestrator. Він координує повний цикл обробки повідомлень для однієї поштової скриньки. Оркестратор отримує список уже опрацьованих IMAP UID, завантажує нові листи, зберігає інформацію про повідомлення, звертається до механізму правил, запускає обробку вкладень, змінює статуси повідомлень і формує записи журналу. Саме цей компонент поєднує окремі сервіси в єдиний бізнес-процес автоматичної обробки електронної пошти.

BackgroundPollingService забезпечує автоматичний запуск обробки без постійної участі користувача. Він працює у фоновому режимі, визначає активні поштові скриньки, перевіряє, чи настав час їх опитування, і викликає EmailProcessingOrchestrator для кожного облікового запису, який потребує перевірки. Крім автоматичного режиму, цей компонент підтримує ручний запуск перевірки, що використовується в інтерфейсі користувача для негайної обробки пошти.

Для доступу до даних використовується AppDbContext, який реалізовано на основі Entity Framework Core. У межах архітектури він виконує роль репозиторію та одиниці роботи. DbSet-властивості забезпечують доступ до колекцій сутностей, а метод SaveChangesAsync використовується для фіксації змін у базі даних. Такий підхід дозволяє не створювати додатковий шар репозиторіїв, оскільки Entity Framework Core уже надає необхідні механізми запитів, відстеження змін і збереження даних.

Захищене зберігання паролів поштових облікових записів забезпечується через IEncryptionService та його реалізацію DpapiEncryptionService. Пароль не

зберігається у базі даних у відкритому вигляді, а перед записом шифрується засобами Windows Data Protection API. Під час підключення до поштової скриньки сервіс роботи з поштою отримує вже розшифроване значення через відповідний сервіс. Це дозволяє зменшити ризики несанкціонованого доступу до облікових даних.

Обрана архітектура відповідає вимогам до розроблюваного програмного забезпечення, оскільки забезпечує модульність, розширюваність і зрозумілий поділ відповідальності. Кожен шар виконує окрему роль: рівень представлення забезпечує взаємодію з користувачем, сервісний рівень реалізує прикладну логіку, доменний рівень описує сутності предметної області, інфраструктурний рівень відповідає за технічну взаємодію із зовнішніми ресурсами, а база даних зберігає конфігурацію та результати обробки. Такий підхід дозволяє підтримувати програмне забезпечення, змінювати окремі компоненти без суттєвого впливу на інші частини системи та надалі розширювати функціональність застосунку.

2.3 Розробка проєкту програмного забезпечення

Після визначення загальної архітектури програмного забезпечення наступним етапом є розробка його проєкту. На цьому етапі необхідно деталізувати склад системи, визначити основні функціональні модулі, описати взаємодію між класами, сформувати структуру бази даних і встановити зв’язок між вимогами до програмного забезпечення та проєктними рішеннями. Проєктування є важливим етапом розробки, оскільки саме воно забезпечує перехід від загального опису задачі до конкретної структури майбутнього програмного продукту.

Розроблюване програмне забезпечення призначене для автоматичного отримання, фільтрації та обробки повідомлень електронної пошти. Основними об’єктами предметної області є поштова скринька, електронне повідомлення, вкладення, правило обробки, дія, журнал обробки та загальні налаштування застосунку. У процесі проєктування було враховано, що система повинна

працювати як у ручному, так і в автоматичному режимі, підтримувати декілька поштових облікових записів, уникати повторної обробки повідомлень, забезпечувати обробку вкладень за правилами та фіксувати всі результати в журналі.

Проєкт програмного забезпечення побудовано з використанням об’єктно-орієнтованого підходу. Це дозволяє представити предметну область у вигляді набору взаємопов’язаних класів, кожен з яких має власну відповідальність. Такий підхід добре відповідає обраній платформі розробки C#/.NET і шаблону MVVM, що використовується для побудови WPF-застосунку. У межах проєкту виділено класи сутностей, сервіси прикладної логіки, інтерфейси для взаємодії між компонентами, клас доступу до бази даних та фоновий сервіс для автоматичного опитування поштових скриньок.

Діаграма класів сервісного шару та шару доступу до даних наведена на рисунку 2.4.

Діаграма класів відображає основні інтерфейси, їх реалізації, клас доступу до бази даних і компоненти оркестрації. У проєкті використано підхід, за якого ключові сервіси визначаються через інтерфейси. Це дозволяє зменшити залежність між окремими модулями системи та забезпечити можливість подальшої заміни реалізацій без зміни коду, який використовує ці сервіси.

Інтерфейс IEmailService визначає контракт для роботи з поштовою скринькою. Він передбачає операції отримання нових повідомлень, позначення листа як прочитаного, переміщення листа до іншої папки та перевірки підключення. Реалізацією цього інтерфейсу є клас MailKitEmailService, який використовує бібліотеку MailKit для взаємодії з IMAP-сервером. Цей клас відповідає за підключення до поштової скриньки, відкриття потрібної папки, отримання MIME-повідомлень і перетворення їх у внутрішню модель ParsedEmail, зручну для подальшої обробки.

Інтерфейс IRuleEngine використовується для відокремлення логіки перевірки правил від інших частин системи. Його реалізація RuleEngineService має одну основну задачу — знайти перше правило, яке відповідає отриманому

повідомленню. Правила перевіряються в порядку пріоритету, при цьому неактивні правила не беруть участі в обробці. Якщо певна умова в правилі не задана, вона не враховується під час перевірки. Такий підхід дозволяє створювати як вузькоспеціалізовані правила, так і загальні правила для обробки або журналювання всіх повідомлень.

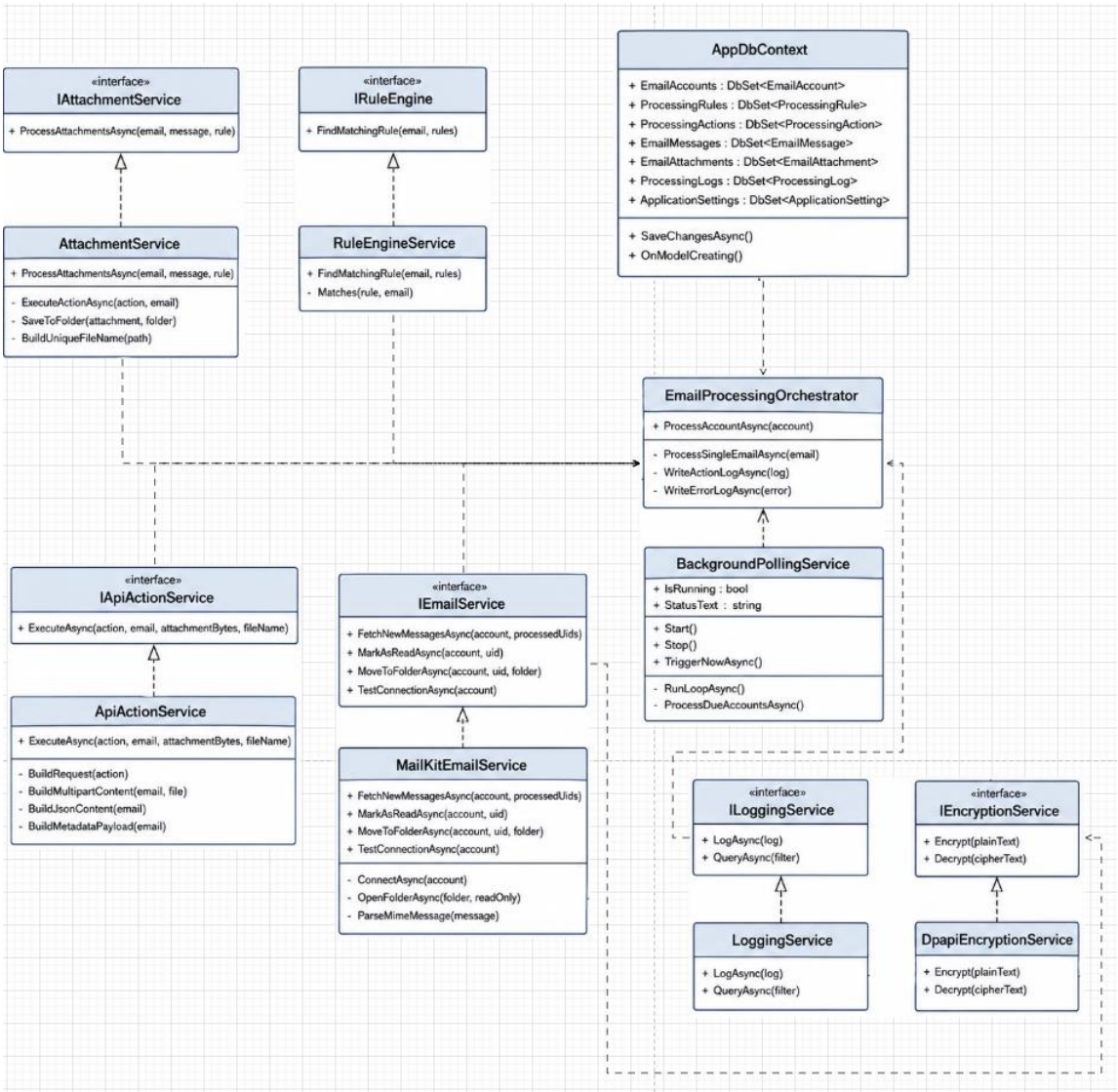


Рисунок 2.4 – Діаграма класів сервісного шару та шару доступу до даних

Інтерфейс IAttachmentService визначає операції, пов’язані з обробкою вкладень. Реалізація AttachmentService відповідає за перевірку вкладених файлів, блокування небезпечних розширень, збереження файлів у каталог, формування унікальних імен файлів і делегування API-дій окремому сервісу. У

проєкті передбачено захисну перевірку вкладень: файли з потенційно небезпечними розширеннями, наприклад виконувані або скриптові файли, не обробляються незалежно від налаштувань користувацьких правил.

Інтерфейс IApiActionService і його реалізація ApiActionService відповідають за виконання HTTP-запитів до зовнішніх інформаційних систем. На основі параметрів дії формується HTTP-запит із потрібним методом, заголовками, токеном авторизації, метаданими листа та, за потреби, вкладеним файлом. Сервіс підтримує передавання файлів у форматі multipart/form-data, а також передавання службової інформації у форматі JSON. Це дозволяє інтегрувати застосунок із зовнішніми REST API без зміни основної логіки обробки пошти.

Інтерфейс ILoggingService визначає операції запису та вибірки журналу. Реалізація LoggingService зберігає записи про результати обробки в базі даних і дозволяє отримувати записи журналу відповідно до заданих фільтрів. Для роботи з базою даних сервіс використовує IDbContextFactory, що дозволяє створювати окремі короткоживучі контексти для асинхронних операцій. Це зменшує ризик конфліктів при одночасному виконанні декількох операцій обробки.

Інтерфейс IEncryptionService використовується для захисту конфіденційних даних. Його реалізація DpapiEncryptionService виконує шифрування та розшифрування паролів поштових облікових записів із використанням Windows Data Protection API. Завдяки цьому пароль поштової скриньки не зберігається у базі даних у відкритому вигляді.

Entity Framework Core забезпечує доступ до бази даних застосунку. У межах проєкту він виконує роль репозиторію та одиниці роботи. AppDbContext містить набори сутностей для таблиць EmailAccounts, ProcessingRules, ProcessingActions, EmailMessages, EmailAttachments, ProcessingLogs і ApplicationSettings. Налаштування сутностей, обмежень, індексів і зв’язків виконується через окремі класи конфігурації Entity Framework Core, що дозволяє залишити доменні сутності простими класами без надлишкової технічної логіки.

Центральним компонентом прикладної логіки є EmailProcessingOrchestrator. Він координує повний цикл обробки повідомлень для однієї поштової скриньки. Оркестратор отримує з бази даних уже оброблені IMAP UID, завантажує нові повідомлення через IEmailService, зберігає інформацію про отримані листи, звертається до IRuleEngine для пошуку відповідного правила, передає вкладення до IAttachmentService, оновлює статуси повідомлень і формує записи журналу через ILoggingService. Завдяки тому, що оркестратор працює через інтерфейси, він не залежить від конкретної реалізації сервісів і може бути протестований окремо від інфраструктурних компонентів.

BackgroundPollingService визначає, які активні поштові облікові записи потребують перевірки, і викликає EmailProcessingOrchestrator для їх обробки.

Для збереження службової інформації застосунку спроєктовано реляційну базу даних Microsoft SQL Server. Структура бази даних побудована таким чином, щоб забезпечити зберігання поштових облікових записів, правил, дій, повідомлень, вкладень, журналів і загальних налаштувань. Структуру бази даних наведено на рисунку 2.5.

База даних містить сім основних таблиць: EmailAccounts (табл. 2.1), ProcessingRules (табл. 2.2), ProcessingActions (табл. 2.3), EmailMessages (табл. 2.4), EmailAttachments (табл. 2.5), ProcessingLogs (табл. 2.6) і ApplicationSettings (табл. 2.7). Кожна таблиця відповідає окремій сутності предметної області або службовому об’єкту конфігурації.

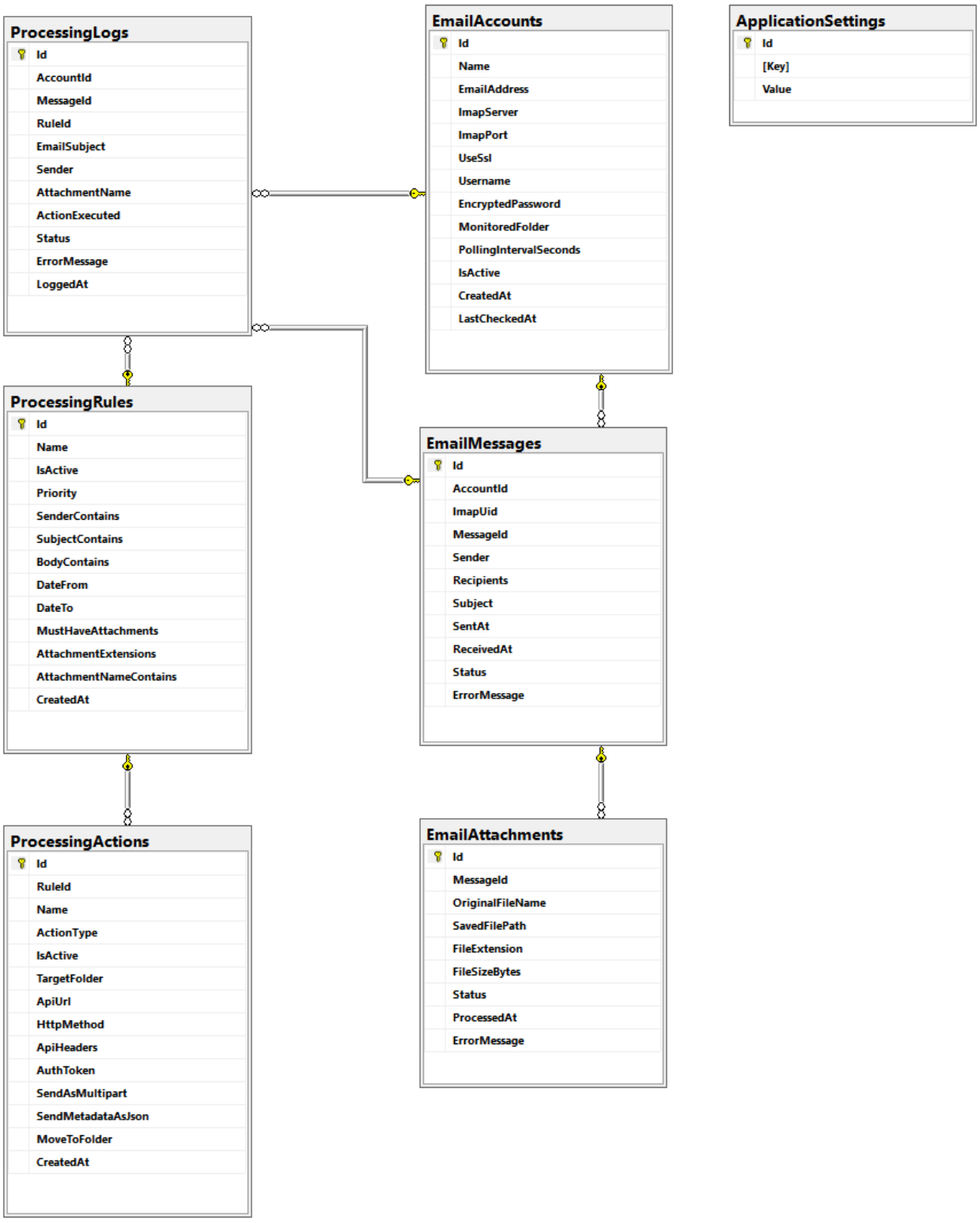


Рисунок 2.5 – Структура бази даних ПЗ

Таблиця EmailAccounts використовується для зберігання параметрів поштових скриньок, які перевіряються застосунком. Для прискорення пошуку

облікових записів за адресою електронної пошти передбачено індекс IX_EmailAccounts_EmailAddress.

Таблиця 2.1 – Структура таблиці EmailAccounts

Поле	Тип даних	Ключ	Опис
Id	INT IDENTITY	PK	Унікальний ідентифікатор поштового облікового запису
Name	NVARCHAR(200)		Назва поштової скриньки, що відображається в інтерфейсі користувача
EmailAddress	NVARCHAR(320)		Адреса електронної пошти
ImapServer	NVARCHAR(256)		Адреса IMAP-сервера
ImapPort	INT		Порт IMAP-сервера, за замовчуванням 993
UseSsl	BIT		Ознака використання захищеного SSL/TLS-з’єднання
Username	NVARCHAR(320)		Ім’я користувача для підключення до поштової скриньки
EncryptedPassword	NVARCHAR(1024)		Пароль або пароль застосунку, збережений у зашифрованому вигляді
MonitoredFolder	NVARCHAR(256)		Поштова папка, що перевіряється застосунком, за замовчуванням INBOX
PollingIntervalSeconds	INT		Інтервал перевірки поштової скриньки в секундах
IsActive	BIT		Ознака активності поштового облікового запису
CreatedAt	DATETIME2		Дата та час створення запису
LastCheckedAt	DATETIME2		Дата та час останньої успішної перевірки поштової скриньки

Таблиця ProcessingRules містить умови, за якими система визначає, чи потрібно обробляти отримане повідомлення. Якщо певне поле умови не заповнене, відповідна перевірка під час аналізу повідомлення не виконується. Це дозволяє створювати як деталізовані, так і загальні правила обробки.

Таблиця 2.2 – Структура таблиці ProcessingRules

Поле	Тип даних	Ключ	Опис
1	**2**	**3**	**4**
Id	INT IDENTITY	PK	Унікальний ідентифікатор правила
Name	NVARCHAR(200)		Назва правила, що відображається в інтерфейсі

Кінець таблиці 2.2

1	2	3	4
IsActive	BIT		Ознака активності правила
Priority	INT		Пріоритет перевірки правила; менше значення означає вищий пріоритет
SenderContains	NVARCHAR(320)		Умова перевірки відправника повідомлення
SubjectContains	NVARCHAR(998)		Умова перевірки теми повідомлення
BodyContains	NVARCHAR(MAX)		Умова перевірки тексту повідомлення
DateFrom	DATETIME2		Початкова дата діапазону, у межах якого має бути надіслано повідомлення
DateTo	DATETIME2		Кінцева дата діапазону, у межах якого має бути надіслано повідомлення
MustHaveAttachments	BIT		Ознака того, що повідомлення повинно містити вкладення
AttachmentExtensions	NVARCHAR(500)		Список допустимих розширень вкладень, розділених комами
AttachmentNameContains	NVARCHAR(500)		Умова перевірки назви вкладеного файлу
CreatedAt	DATETIME2		Дата та час створення правила

Таблиця ProcessingActions зберігає дії, які виконуються після спрацювання правила. Одне правило може мати декілька пов’язаних дій, що дозволяє одночасно виконувати кілька операцій, наприклад зберігати вкладення у файлову систему та передавати його до зовнішнього API. Зв’язок між ProcessingRules і ProcessingActions «один до багатьох» із каскадним видаленням дій при видаленні правила.

Таблиця 2.3 – Структура таблиці ProcessingActions

Поле	Тип даних	Ключ	Опис
1	2	3	4
Id	INT IDENTITY	PK	Унікальний ідентифікатор дії
RuleId	INT	FK	Посилання на правило, до якого належить дія
Name	NVARCHAR(200)		Назва дії
ActionType	NVARCHAR(50)		Тип дії: SaveToFolder, SendAttachmentToApi, SendMetadataToApi або MoveToFolder
IsActive	BIT		Ознака активності дії
TargetFolder	NVARCHAR(1024)		Каталог для збереження вкладень

Кінець таблиці 2.3

1	2	3	4
ApiUrl	NVARCHAR(2048)		Адреса зовнішнього API
HttpMethod	NVARCHAR(10)		HTTP-метод для API-запиту
ApiHeaders	NVARCHAR(4096)		Додаткові HTTP-заголовки у форматі JSON
AuthToken	NVARCHAR(2048)		Токен авторизації або API-ключ
SendAsMultipart	BIT		Ознака передавання вкладення у форматі multipart/form-data
SendMetadataAsJson	BIT		Ознака передавання метаданих повідомлення у форматі JSON
MoveToFolder	NVARCHAR(256)		Поштова папка, до якої потрібно перемістити повідомлення
CreatedAt	DATETIME2		Дата та час створення дії

Таблиця EmailMessages призначена для зберігання метаданих отриманих електронних повідомлень. Для запобігання повторній обробці листів використовується обмеження унікальності UQ_EmailMessages_AccountUid для пари AccountId та ImapUid. Завдяки цьому одне й те саме повідомлення не може бути повторно збережене й оброблене в межах однієї поштової скриньки.

Таблиця 2.4 – Структура таблиці EmailMessages

Поле	Тип даних	Ключ	Опис
Id	INT IDENTITY	PK	Унікальний ідентифікатор повідомлення
AccountId	INT	FK	Посилання на поштовий обліковий запис
ImapUid	BIGINT		Унікальний IMAP UID повідомлення в межах поштової папки
MessageId	NVARCHAR(998)		Значення заголовка Message-ID електронного листа
Sender	NVARCHAR(320)		Адреса відправника
Recipients	NVARCHAR(4000)		Список отримувачів повідомлення
Subject	NVARCHAR(998)		Тема повідомлення
SentAt	DATETIME2		Дата та час надсилання повідомлення
ReceivedAt	DATETIME2		Дата та час отримання повідомлення застосунком
Status	NVARCHAR(20)		Статус обробки повідомлення: Pending, Processed, Failed або Skipped
ErrorMessage	NVARCHAR(4000)		Текст помилки, якщо обробка завершилася невдало

Таблиця EmailAttachments містить інформацію про вкладені файли, отримані з електронних повідомлень. Один лист може містити декілька

вкладень, тому між EmailMessages і EmailAttachments встановлено зв’язок «один до багатьох». У разі видалення повідомлення відповідні записи про вкладення можуть видалятися каскадно, оскільки вкладення існують лише в контексті конкретного листа.

Таблиця 2.5 – Структура таблиці EmailAttachments

Поле	Тип даних	Ключ	Опис
Id	INT IDENTITY	PK	Унікальний ідентифікатор вкладення
MessageId	INT	FK	Посилання на електронне повідомлення
OriginalFileName	NVARCHAR(512)		Початкова назва файлу, отримана з електронного листа
SavedFilePath	NVARCHAR(1024)		Повний шлях до збереженого файлу
FileExtension	NVARCHAR(20)		Розширення файлу
FileSizeBytes	BIGINT		Розмір вкладення в байтах
Status	NVARCHAR(20)		Статус обробки вкладення: Pending, Saved, Sent, Failed або Skipped
ProcessedAt	DATETIME2		Дата та час обробки вкладення
ErrorMessage	NVARCHAR(4000)		Текст помилки, якщо обробка вкладення завершилася невдало

Таблиця ProcessingLogs використовується як журнал аудиту. Вона фіксує успішні операції, попередження та помилки, що виникають під час обробки поштових повідомлень і вкладень. Записи журналу повинні зберігатися навіть у разі видалення пов’язаного повідомлення або правила, тому для зв’язків із EmailMessages і ProcessingRules доцільно використовувати поведінку set null on delete. Для швидкого перегляду останніх подій передбачено індекс за полем LoggedAt.

Таблиця 2.6 – Структура таблиці ProcessingLogs

Поле	Тип даних	Ключ	Опис
1	**2**	**3**	**4**
Id	INT IDENTITY	PK	Унікальний ідентифікатор запису журналу
AccountId	INT	FK	Посилання на поштовий обліковий запис
MessageId	INT	FK	Посилання на електронне повідомлення, якщо воно пов’язане із записом журналу
RuleId	INT	FK	Посилання на правило, яке спричинило запис журналу

Кінець таблиці 2.6

1	2	3	4
EmailSubject	NVARCHAR(998)		Тема повідомлення на момент створення запису журналу
Sender	NVARCHAR(320)		Адреса відправника на момент створення запису журналу
AttachmentName	NVARCHAR(512)		Назва вкладення, якщо подія пов’язана з файлом
ActionExecuted	NVARCHAR(500)		Опис виконаної дії
Status	NVARCHAR(20)		Статус події: Success, Failure або Warning
ErrorMessage	NVARCHAR(4000)		Текст помилки, якщо подія завершилася невдало
LoggedAt	DATETIME2		Дата та час створення запису журналу

Таблиця ApplicationSettings призначена для зберігання загальних параметрів застосунку у форматі «ключ-значення». У ній можуть зберігатися максимальний розмір вкладень, список заборонених розширень, стандартний інтервал перевірки пошти та інші системні налаштування. Для поля Key передбачено обмеження унікальності, що гарантує наявність лише одного запису для кожного параметра.

Таблиця 2.7 – Структура таблиці ApplicationSettings

Поле	Тип даних	Ключ	Опис
Id	INT IDENTITY	PK	Унікальний ідентифікатор параметра
Key	NVARCHAR(200)		Назва параметра застосунку
Value	NVARCHAR(4000)		Значення параметра у текстовому форматі

Загалом розроблена структура бази даних забезпечує зберігання всіх даних, необхідних для роботи програмного забезпечення: параметрів поштових скриньок, правил обробки, дій, отриманих повідомлень, вкладень, журналу подій і глобальних налаштувань. Зв’язки між таблицями відповідають логіці предметної області та дозволяють забезпечити цілісність даних, уникнути повторної обробки повідомлень і зберегти історію виконаних операцій.

Для кращого розуміння логіки роботи системи доцільно описати загальний алгоритм автоматичної обробки повідомлення. Він складається з таких етапів: отримання активних поштових облікових записів, перевірка необхідності

опитування кожної скриньки, підключення до IMAP-сервера, отримання нових повідомлень, перевірка унікальності повідомлення за IMAP UID, збереження метаданих листа, пошук відповідного правила, обробка вкладень, виконання дій, оновлення статусів і запис результатів у журнал.

Спочатку фоновий сервіс отримує список активних поштових облікових записів із бази даних. Для кожного облікового запису перевіряється, чи настав час чергового опитування відповідно до його інтервалу. Якщо скринька потребує перевірки, система викликає оркестратор обробки, який отримує з бази даних список уже оброблених IMAP UID і передає його до сервісу роботи з поштою. Сервіс MailKitEmailService підключається до поштового сервера, відкриває потрібну папку та повертає тільки ті повідомлення, які ще не були опрацьовані.

Після отримання нового повідомлення система зберігає його метадані в таблиці EmailMessages. Це виконується на початку обробки для того, щоб у разі збою повідомлення не було втрачено з погляду системного обліку. Далі повідомлення передається до RuleEngineService, який перевіряє активні правила в порядку пріоритету. Якщо жодне правило не відповідає повідомленню, лист отримує статус Skipped, а в журналі створюється відповідний запис.

Якщо правило знайдено, система переходить до обробки вкладень і виконання дій, пов’язаних із правилом. AttachmentService перевіряє вкладення, відсіює заборонені або такі, що не відповідають умовам правила, формує безпечні імена файлів і виконує налаштовані дії. Якщо дія передбачає збереження вкладення, файл записується у визначену папку. Якщо дія передбачає передавання даних до API, відповідний запит формує та виконує ApiActionService. Результати обробки кожного вкладення зберігаються в таблиці EmailAttachments.

Після завершення обробки система оновлює статус повідомлення. Якщо всі необхідні дії виконано успішно, повідомлення отримує статус Processed. Якщо під час обробки виникла помилка, статус змінюється на Failed, а текст помилки зберігається у відповідному полі та в журналі. Усі важливі події фіксуються через LoggingService, що забезпечує можливість подальшого

перегляду та аналізу роботи системи користувачем.

Розроблений проєкт програмного забезпечення відповідає вимогам, сформульованим у постановці задачі. Він забезпечує підтримку декількох поштових облікових записів, правил фільтрації повідомлень, дій з вкладеннями, інтеграції із зовнішніми API, захищеного зберігання облікових даних, журналювання та збереження службової інформації в базі даних. Об’єктно-орієнтована структура класів, використання інтерфейсів і виділення окремих сервісів дозволяють зробити систему зрозумілою, підтримуваною та придатною до подальшого розширення.

2.4 Реалізація програмного забезпечення

Реалізація програмного забезпечення виконувалася відповідно до спроєктованої багатошарової архітектури. Інтерфейс користувача відокремлено від прикладної логіки, прикладна логіка — від доступу до даних і зовнішніх сервісів, а робота з поштовими серверами, файловою системою, API та шифруванням винесена в окремі інфраструктурні компоненти. Такий підхід дозволив зменшити зв’язаність між частинами програми та спростити супровід програмного коду.

Основними технологіями, використаними під час реалізації, є WPF для створення інтерфейсу користувача, Entity Framework Core для доступу до бази даних Microsoft SQL Server, бібліотека MailKit для роботи з поштовими серверами за протоколом IMAP, HttpClient для виконання HTTP-запитів до зовнішніх API та Windows DPAPI для захищеного зберігання паролів поштових облікових записів.

Реалізація рівня представлення виконана на основі шаблону MVVM. Для кожного основного екрана застосунку створено відповідну ViewModel, яка містить дані, що відображаються в інтерфейсі, та команди, які викликаються внаслідок дій користувача. До основних розділів інтерфейсу належать панель керування, розділ поштових облікових записів, розділ правил і дій, перегляд вкладень, журнал обробки та налаштування застосунку. View відповідає лише за

відображення даних, тоді як ViewModel звертається до сервісів прикладного шару для виконання необхідних операцій.

Наприклад, під час натискання кнопки ручної перевірки пошти користувацький інтерфейс не виконує безпосереднє підключення до поштової скриньки. Замість цього команда у ViewModel викликає фоновий сервіс або оркестратор обробки, який запускає повний цикл отримання та обробки повідомлень. Такий підхід відповідає принципу розділення відповідальності та дозволяє уникнути розміщення бізнес-логіки у коді графічного інтерфейсу.

```
ProcessNowCommand = new AsyncRelayCommand(async () =>
{
    IsBusy = true;
    try   { await _polling.TriggerNowAsync(); }
    finally { IsBusy = false; }
    await DashboardVm.LoadAsync();
});
```

Для зберігання даних застосунку використовується база даних Microsoft SQL Server. Доступ до неї реалізовано за допомогою Entity Framework Core. Центральним класом шару доступу до даних є AppDbContext, який містить DbSet-властивості для основних сутностей системи: EmailAccounts, ProcessingRules, ProcessingActions, EmailMessages, EmailAttachments, ProcessingLogs і ApplicationSettings. Через ці набори сутностей здійснюється читання, додавання, оновлення та видалення даних.

У межах реалізації AppDbContext виконує роль репозиторію та одиниці роботи. Окремий універсальний репозиторій не створювався, оскільки Entity Framework Core уже надає необхідні механізми запитів до даних, відстеження змін і збереження результатів через метод SaveChangesAsync. Налаштування таблиць, зв’язків, обмежень, значень за замовчуванням та індексів винесено в окремі класи конфігурації сутностей. Це дозволяє не перевантажувати доменні класи технічними деталями збереження даних.

```
public void Configure(EntityTypeBuilder<EmailAccount> b)
{
    b.ToTable("EmailAccounts");
    b.HasKey(e => e.Id);
    b.Property(e => e.Name).HasMaxLength(200).IsRequired();
```

```
        b.Property(e =>
e.EmailAddress).HasMaxLength(320).IsRequired();
        b.Property(e =>
e.ImapServer).HasMaxLength(256).IsRequired();
        b.Property(e =>
e.Username).HasMaxLength(320).IsRequired();
        b.Property(e =>
e.EncryptedPassword).HasMaxLength(1024).IsRequired();
        b.Property(e =>
e.MonitoredFolder).HasMaxLength(256).HasDefaultValue("INBOX")
;
        b.HasIndex(e => e.EmailAddress);
    }
```

Однією з важливих частин реалізації є механізм захисту від повторної обробки повідомлень. Кожен лист, отриманий із поштової скриньки, має IMAP UID, який є унікальним у межах відповідної папки поштового облікового запису. Перед отриманням нових повідомлень система завантажує з бази даних набір уже оброблених UID для конкретного облікового запису. Після цього сервіс роботи з поштою повертає лише ті повідомлення, яких ще немає в цьому наборі. Додатково на рівні бази даних передбачено унікальне обмеження для пари AccountId та ImapUid, що унеможливлює повторне збереження одного й того самого листа.

Робота з електронною поштою реалізована в сервісі MailKitEmailService, який є реалізацією інтерфейсу IEmailService. Цей сервіс відповідає за підключення до поштового сервера, відкриття заданої папки, отримання нових листів, позначення повідомлень як прочитаних, переміщення листів між папками та перевірку підключення. Для взаємодії з поштовим сервером використовується бібліотека MailKit, яка підтримує роботу з IMAP і MIME-повідомленнями.

Під час підключення до поштової скриньки сервіс використовує параметри, збережені в таблиці EmailAccounts: адресу IMAP-сервера, порт, ознаку використання SSL/TLS, ім’я користувача, зашифрований пароль і назву папки для моніторингу. Перед автентифікацією пароль розшифровується за допомогою IEncryptionService. Після встановлення з’єднання сервіс відкриває потрібну папку та зчитує повідомлення, які ще не були оброблені застосунком.

```
private async Task<ImapClient> ConnectAsync(EmailAccount
account, CancellationToken ct)
{
    var client   = new ImapClient();
    var sslMode  = account.UseSsl ?
SecureSocketOptions.SslOnConnect :
SecureSocketOptions.StartTlsWhenAvailable;
    var password =
_encryption.Decrypt(account.EncryptedPassword);

    await client.ConnectAsync(account.ImapServer,
account.ImapPort, sslMode, ct);
    await client.AuthenticateAsync(account.Username,
password, ct);
    return client;
}
```

Отримане MIME-повідомлення перетворюється у внутрішню модель ParsedEmail. Така модель містить основну інформацію про лист: відправника, отримувачів, тему, текст, дату надсилання, IMAP UID, Message-ID і список вкладень. Перетворення зовнішнього формату повідомлення у внутрішню модель спрощує подальшу обробку, оскільки інші сервіси не залежать від конкретних класів бібліотеки MailKit і працюють із власною структурою даних застосунку.

Механізм перевірки правил реалізовано в сервісі RuleEngineService, який є реалізацією інтерфейсу IRuleEngine. Його основне завдання полягає у знаходженні першого правила, яке відповідає отриманому повідомленню. Правила перевіряються в порядку пріоритету, при цьому неактивні правила не беруть участі в обробці. Якщо певна умова правила не задана, вона пропускається, що дає змогу створювати правила різного рівня деталізації.

Під час перевірки повідомлення враховуються такі параметри: адреса відправника, тема листа, текст повідомлення, дата надсилання, наявність вкладень, допустимі розширення вкладень і частина назви файлу. Якщо всі задані умови виконуються, правило вважається таким, що спрацювало. Якщо жодне правило не відповідає повідомленню, лист отримує статус Skipped, а інформація про це може бути записана в журнал.

```
public ProcessingRule? FindMatchingRule(ParsedEmail email,
IEnumerable<ProcessingRule> rules)
```

```
{
    foreach (var rule in rules.Where(r =>
r.IsActive).OrderBy(r => r.Priority))
    {
        if (Matches(email, rule))
        {
            _logger.LogDebug("Rule '{Rule}' matched message
'{Subject}'.", rule.Name, email.Subject);
            return rule;
        }
    }

    _logger.LogDebug("No rule matched message '{Subject}'.",
email.Subject);
    return null;
}
```

Обробка вкладень реалізована в сервісі AttachmentService. Цей сервіс відповідає за перевірку вкладених файлів, виконання дій, пов'язаних із правилом, збереження вкладень у файлову систему та формування метаданих для запису в базу даних. Перед обробкою вкладення система перевіряє його розширення. Для підвищення безпеки в реалізації передбачено список заблокованих розширень, до якого належать виконувані та скриптові файли. Такі файли не обробляються навіть у тому випадку, якщо користувач помилково дозволив їх у правилі.

Під час збереження вкладення у файлову систему виконується очищення назви файлу від недопустимих символів і символів, які можуть використовуватися для обходу шляху збереження. Також реалізовано формування унікальної назви файлу, щоб не перезаписати вже існуючий файл з такою самою назвою. Після завершення обробки для кожного вкладення формується запис EmailAttachment, який містить назву файлу, шлях збереження, розширення, розмір, статус обробки та текст помилки, якщо вона виникла.

```
if (BlockedExtensions.Contains(attachment.Extension))
{
    _logger.LogWarning("Blocked dangerous attachment
'{File}'.", attachment.FileName);
    record.Status       = AttachmentStatus.Skipped;
    record.ErrorMessage = $"Blocked extension:
{attachment.Extension}";
    results.Add(record);
    continue;
```

```
}
```

Передавання вкладень або метаданих повідомлення до зовнішніх систем реалізовано в сервісі ApiActionService. Цей сервіс використовує HttpClient і формує HTTP-запит на основі параметрів дії, заданих користувачем. До таких параметрів належать адреса API, HTTP-метод, додаткові заголовки, токен авторизації, спосіб передавання вкладення та ознака додавання метаданих повідомлення.

Якщо дія передбачає передавання файлу, запит може формуватися у форматі multipart/form-data. У цьому випадку вкладення додається до тіла запиту як файлова частина, а метадані повідомлення можуть додатково передаватися як JSON-частина. Якщо потрібно передати лише службову інформацію про повідомлення, формується JSON-запит, який містить відправника, отримувачів, тему, дату надсилання та список вкладень. Результат виконання API-запиту фіксується в журналі обробки.

```
case Domain.Enums.ActionType.SendAttachmentToApi:
    await _apiService.ExecuteAsync(action, email,
attachment.Content, attachment.FileName, ct);
    record.Status = AttachmentStatus.Sent;
    break;
```

Координацію повного циклу обробки повідомлень виконує EmailProcessingOrchestrator. Цей клас є центральним компонентом прикладної логіки. Він отримує з бази даних список уже оброблених IMAP UID, завантажує активні правила, викликає сервіс отримання пошти, зберігає інформацію про нові повідомлення, передає повідомлення до механізму правил, запускає обробку вкладень, оновлює статуси повідомлень і створює записи журналу. Результатом роботи оркестратора є підсумковий об’єкт ProcessingResult, який містить кількість отриманих, оброблених, пропущених і помилкових повідомлень.

Важливою особливістю реалізації є те, що EmailProcessingOrchestrator залежить не від конкретних класів, а від інтерфейсів IEmailService, IRuleEngine, IAttachmentService і ILoggingService. Це відповідає принципу інверсії

залежностей і дозволяє спростити подальше тестування та розширення системи. Наприклад, у майбутньому реалізацію роботи з поштою можна замінити на інший сервіс без зміни логіки оркестратора.

```
public EmailProcessingOrchestrator(
        IEmailService emailService,
        IRuleEngine ruleEngine,
        IAttachmentService attachmentService,
        ILoggingService loggingService,
        IDbContextFactory<AppDbContext> dbFactory,
        ILogger<EmailProcessingOrchestrator> logger)
```

Для захисту паролів поштових облікових записів реалізовано сервіс DpapiEncryptionService, який відповідає за шифрування та розшифрування конфіденційних даних. Під час збереження поштового облікового запису пароль не записується до бази даних у відкритому вигляді, а передається до сервісу шифрування. Під час підключення до поштової скриньки збережене зашифроване значення розшифровується і використовується лише для автентифікації на поштовому сервері.

Використання Windows DPAPI є доцільним для настільного застосунку Windows, оскільки механізм шифрування прив'язаний до середовища користувача операційної системи. Це зменшує ризики розкриття паролів у разі несанкціонованого перегляду бази даних. Разом із цим у програмному забезпеченні передбачено, що чутливі параметри, такі як API-токени, також повинні зберігатися з урахуванням вимог інформаційної безпеки.

```
public string Encrypt(string plainText)
{
    if (string.IsNullOrEmpty(plainText)) return string.Empty;

        var bytes     = Encoding.UTF8.GetBytes(plainText);
        var encrypted = ProtectedData.Protect(bytes, null,
DataProtectionScope.CurrentUser);
        return Convert.ToBase64String(encrypted);
    }
```

У процесі реалізації також було передбачено обробку помилок. Помилки можуть виникати під час підключення до поштового сервера, автентифікації, отримання повідомлень, збереження файлів, виконання HTTP-запитів, роботи з базою даних або обробки некоректних вкладень. Такі помилки не повинні

призводити до аварійного завершення роботи застосунку. Замість цього система змінює статус відповідного повідомлення або вкладення на Failed, зберігає текст помилки та створює запис журналу.

Окрему увагу приділено безпечній роботі з вкладеннями. Назви файлів очищуються від недопустимих символів, небезпечні розширення блокуються, а завантажені файли не запускаються автоматично. Застосунок лише зберігає вкладення або передає їх до зовнішнього API відповідно до налаштованих правил. Це зменшує ризики, пов’язані з отриманням файлів із зовнішніх джерел через електронну пошту.

Реалізоване програмне забезпечення забезпечує виконання основних функціональних вимог, визначених на етапі аналізу та проєктування. Застосунок дозволяє налаштовувати поштові облікові записи, створювати правила обробки, задавати дії, автоматично та вручну перевіряти поштові скриньки, отримувати нові повідомлення, обробляти вкладення, передавати дані до зовнішніх API, зберігати результати обробки в базі даних і переглядати журнал. Реалізація виконана відповідно до обраної архітектури, що забезпечує модульність, підтримуваність і можливість подальшого розширення програмного продукту.

3 РЕЗУЛЬТАТИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У результаті виконання кваліфікаційної роботи було розроблено програмне забезпечення для автоматичної обробки повідомлень електронної пошти. Розроблений застосунок призначений для автоматизації роботи з поштовими повідомленнями, які містять вкладені файли або службову інформацію, що потребує подальшого збереження, фільтрації або передавання до зовнішніх інформаційних систем.

Розроблений застосунок дозволяє користувачеві налаштовувати поштові облікові записи, задавати параметри підключення до IMAP-сервера, створювати правила обробки повідомлень, визначати дії після спрацювання правил, виконувати ручну перевірку пошти, використовувати автоматичний режим обробки, переглядати отримані повідомлення, вкладення та журнал подій. Завдяки цьому програмне забезпечення може використовуватися для автоматизації повторюваних операцій, пов’язаних із отриманням документів, рахунків, звітів або інших файлів електронною поштою.

Початковим екраном застосунку є панель керування (рис. 3.1), яка дає змогу користувачеві швидко оцінити загальний стан системи. На ній відображається інформація про кількість оброблених повідомлень, успішні та помилкові операції, активність фонового сервісу, а також останні записи журналу. Такий екран дозволяє контролювати роботу програмного забезпечення без необхідності одразу переходити до детального журналу.

Однією з основних функцій розробленого програмного забезпечення є керування поштовими обліковими записами (рис. 3.2). Користувач може додати нову поштову скриньку, вказавши її назву, адресу електронної пошти, IMAP-сервер, порт, логін, пароль або пароль застосунку, папку для моніторингу та інтервал перевірки. Також передбачено можливість активувати або деактивувати обліковий запис, редагувати його параметри та перевірити правильність

підключення до поштового сервера.

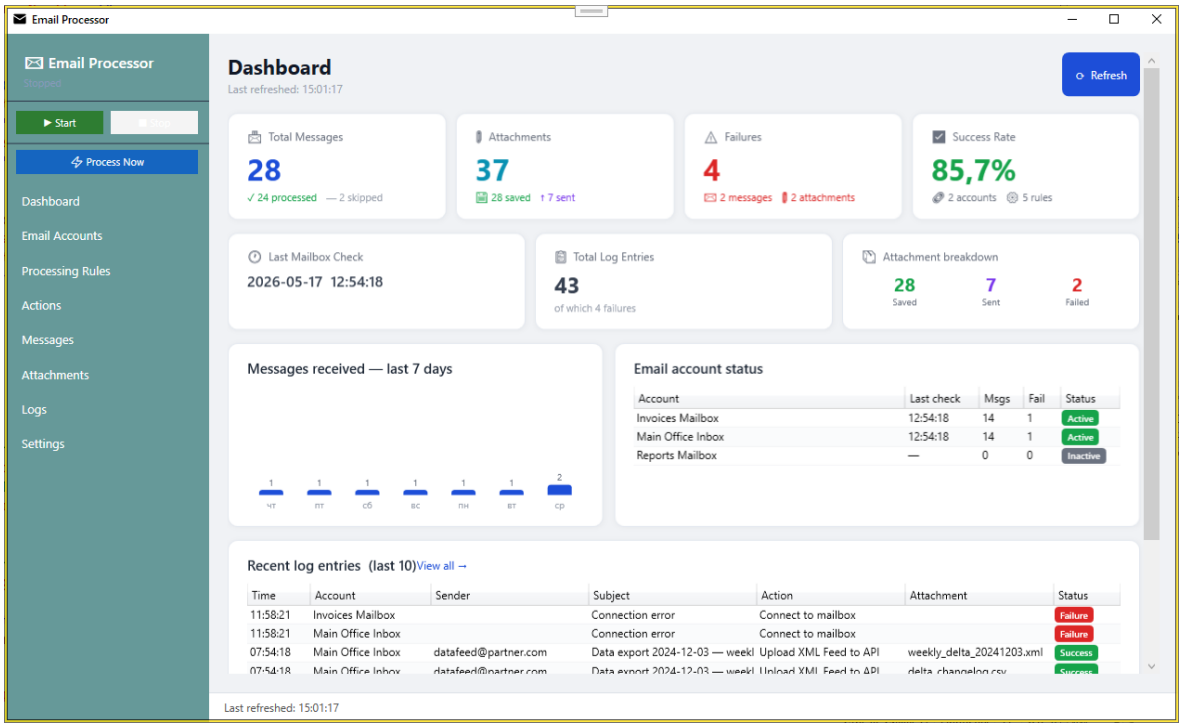


Рисунок 3.1 – Головне вікно програмного забезпечення

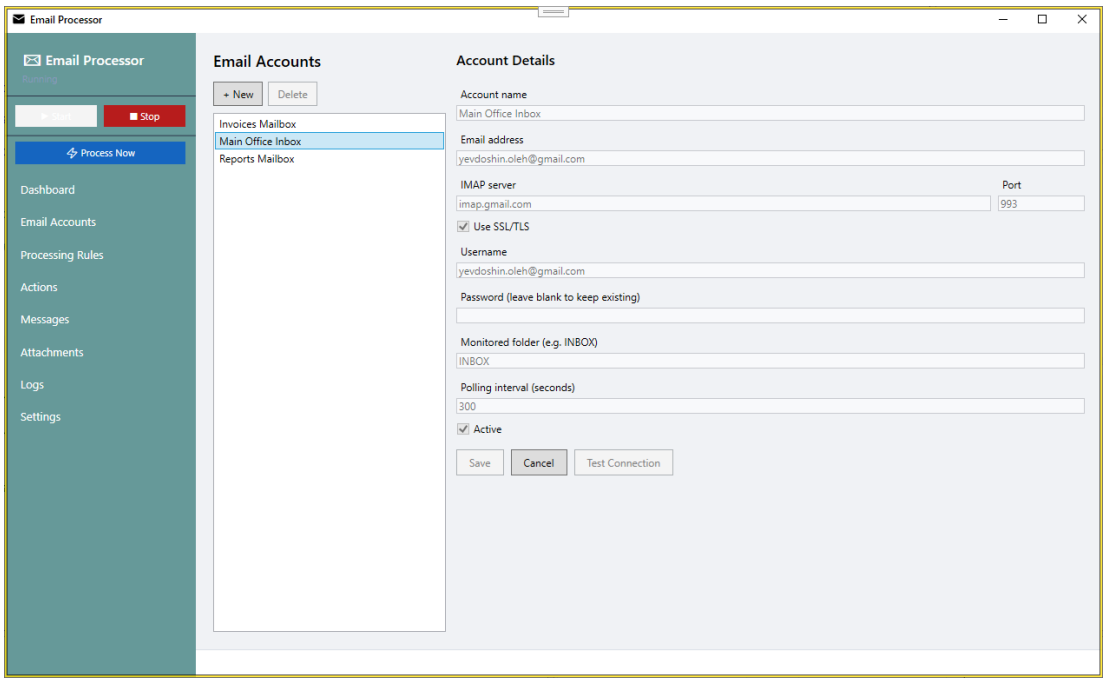


Рисунок 3.2 – Вікно керування поштовими обліковими записами

Для забезпечення гнучкої обробки повідомлень у застосунку реалізовано механізм правил (рис. 3.3). Користувач може створювати правила, які

визначають, які саме листи повинні оброблятися системою. До умов правила належать відправник, тема листа, текст повідомлення, дата надсилання, наявність вкладень, допустимі розширення файлів і частина назви вкладення. Кожне правило має ознаку активності та пріоритет, що дозволяє визначати порядок перевірки повідомлень.

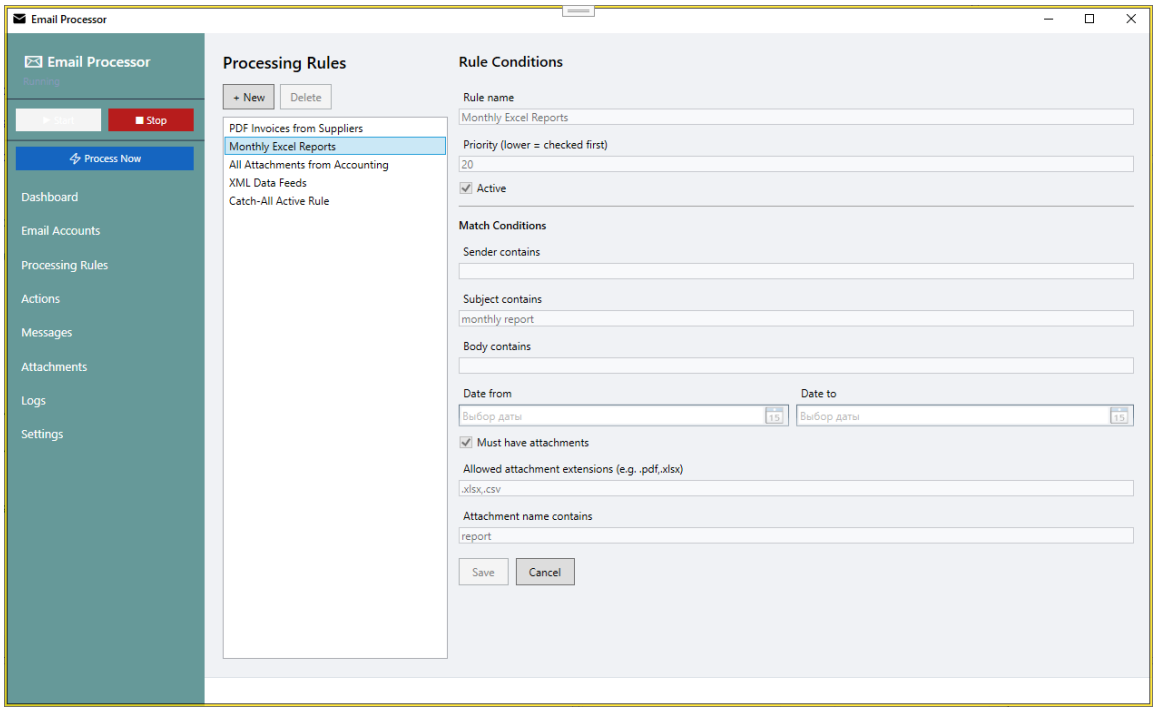


Рисунок 3.3 – Вікно керування правилами обробки повідомлень

Після спрацювання правила система виконує одну або декілька налаштованих дій (рис. 3.4). У розробленому програмному забезпеченні передбачено збереження вкладень у локальний або мережевий каталог, передавання вкладень до зовнішнього API, передавання метаданих повідомлення у форматі JSON, а також переміщення листа до іншої папки поштової скриньки. Наявність окремого механізму дій дозволяє користувачеві самостійно визначати, що саме має відбутися після знаходження відповідного повідомлення.

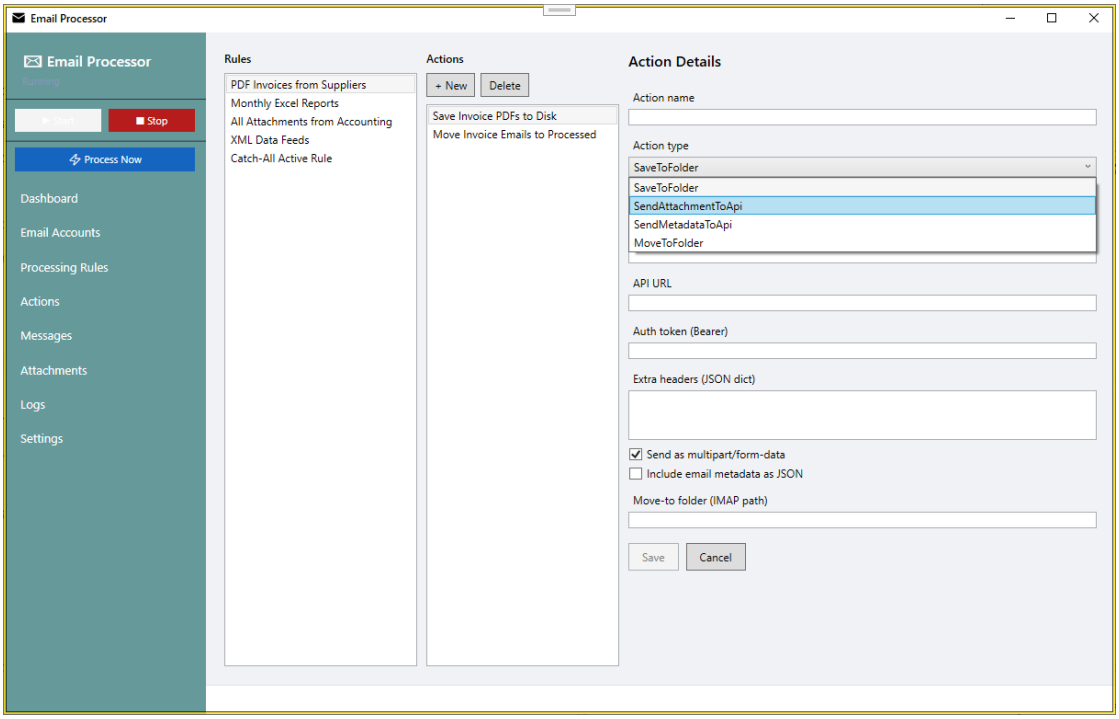


Рисунок 3.4 – Вікно налаштування дій після спрацювання правила

У програмному забезпеченні реалізовано як ручний, так і автоматичний режим обробки пошти (рис. 3.5). У ручному режимі користувач може самостійно запустити перевірку активних поштових скриньок. В автоматичному режимі фоновий сервіс періодично перевіряє поштові облікові записи відповідно до заданого інтервалу. Для кожної скриньки зберігається час останньої перевірки, що дозволяє опитувати різні поштові облікові записи з різною частотою.

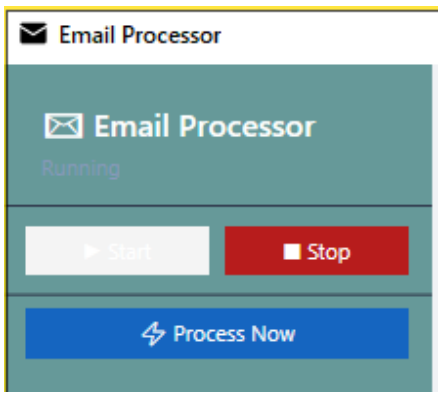


Рисунок 3.5 – Ручний та автоматичний запуск обробки поштових повідомлень

Під час обробки повідомлень система отримує нові листи з поштової скриньки (рис. 3.6), перевіряє їх за налаштованими правилами, обробляє вкладення та зберігає інформацію про результат у базі даних. Для запобігання повторній обробці використовується IMAP UID повідомлення в межах конкретного поштового облікового запису. Це дозволяє повторно запускати перевірку пошти або перезапускати застосунок без ризику дублювання обробки одного й того самого листа.

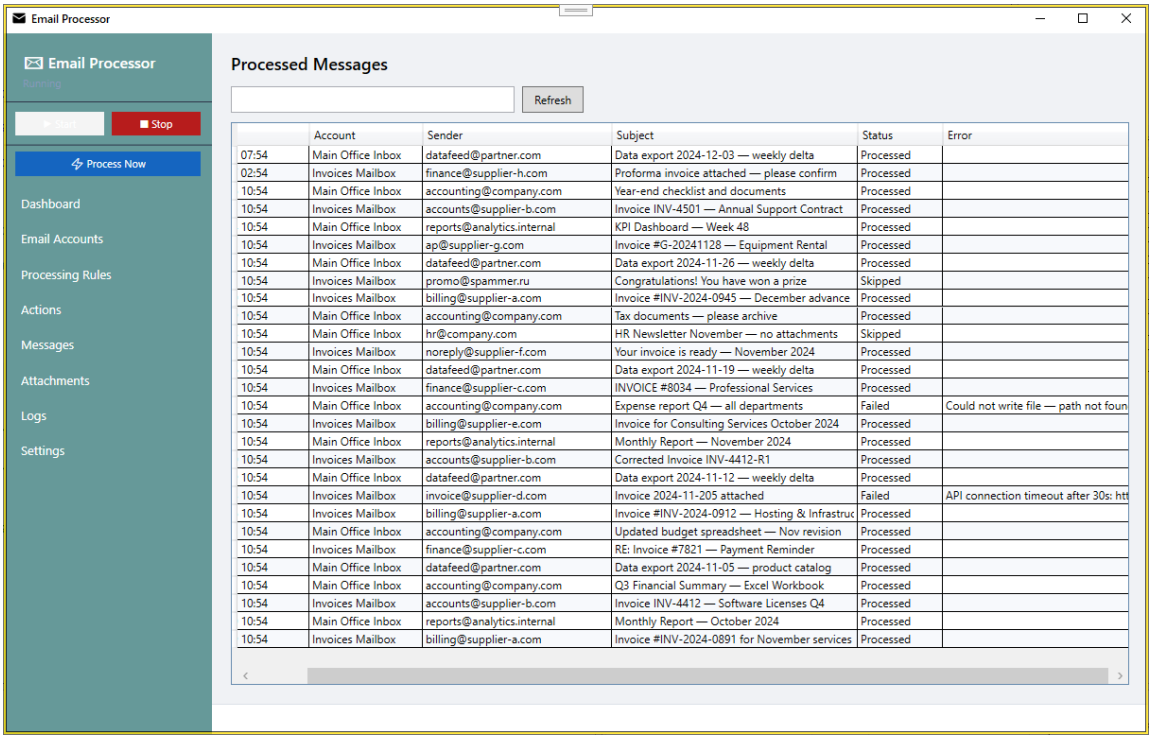


Рисунок 3.6 – Перегляд оброблених повідомлень

Окремий розділ застосунку призначений для перегляду вкладень (рис. 3.7). У ньому користувач може побачити назву вкладеного файлу, розширення, розмір, шлях збереження, статус обробки та повідомлення про помилку за наявності. Це дозволяє контролювати, які саме файли були отримані, збережені або передані до зовнішньої системи.

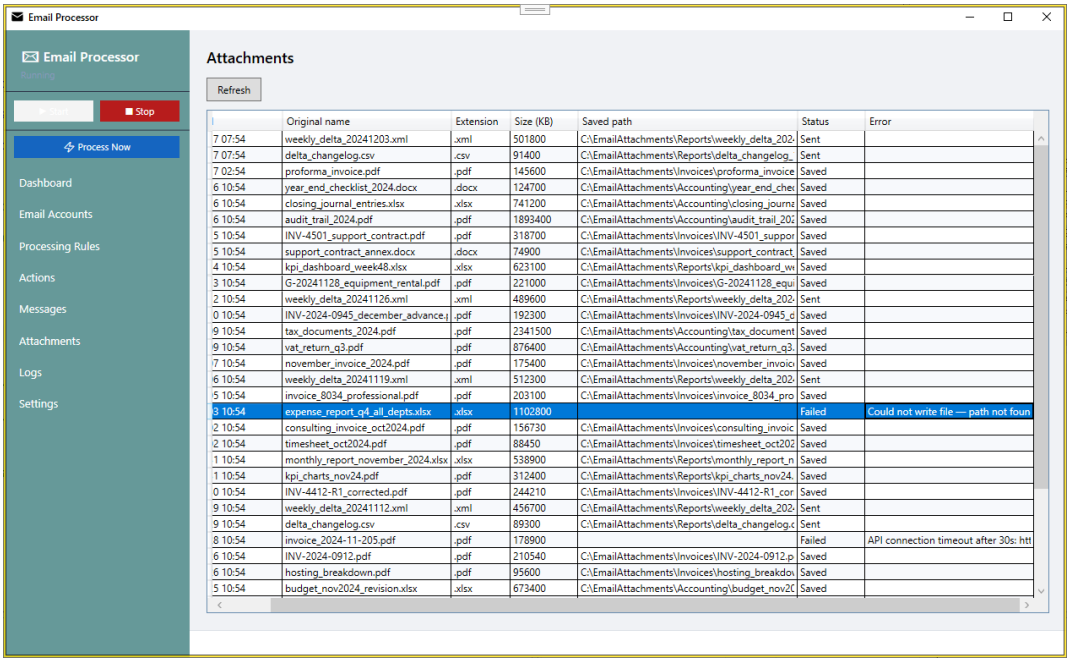


Рисунок 3.7 – Перегляд інформації про вкладення

Для контролю роботи системи реалізовано журнал обробки (рис. 3.8). У журналі фіксуються успішні операції, попередження та помилки, які виникають під час підключення до поштового сервера, отримання повідомлень, перевірки правил, збереження вкладень, передавання даних до API або роботи з базою даних. Користувач може переглядати записи журналу та фільтрувати їх за датою, поштовим обліковим записом, відправником, темою, правилом і статусом.

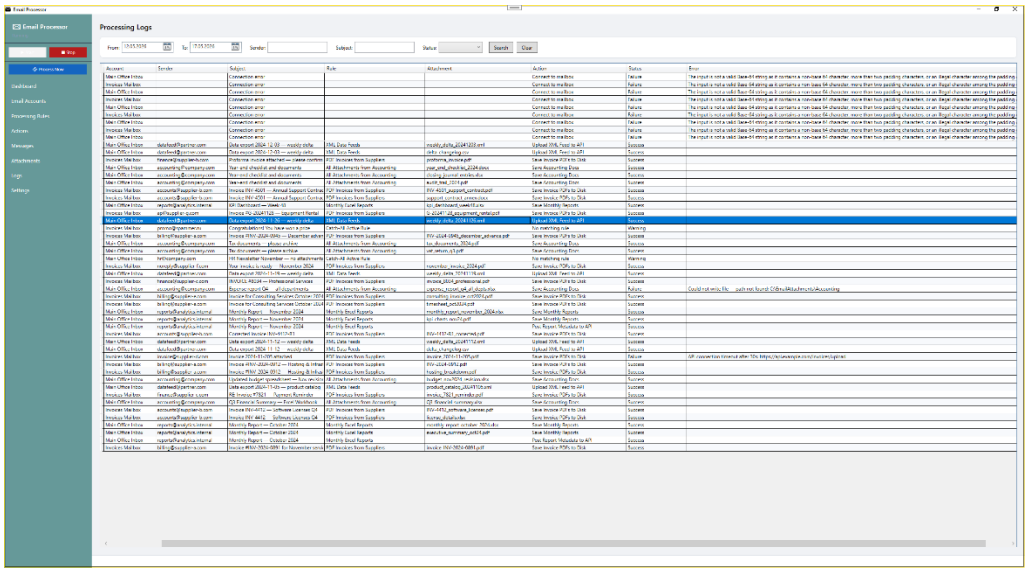


Рисунок 3.8 – Журнал обробки повідомлень електронної пошти

Реалізоване програмне забезпечення відповідає основним вимогам, визначеним у технічному завданні. Зокрема, у застосунку реалізовано керування поштовими обліковими записами, підключення до поштової скриньки за протоколом IMAP, отримання нових повідомлень, запобігання повторній обробці листів, створення правил обробки, налаштування дій, обробку вкладень, збереження файлів, передавання даних до зовнішнього API, ведення журналу та перегляд результатів роботи системи.

Працездатність програмного забезпечення перевірялася шляхом виконання основних сценаріїв роботи: додавання поштового облікового запису, перевірка підключення, створення правила, налаштування дії, ручний запуск перевірки пошти, автоматична обробка повідомлення, збереження вкладення, передавання даних до API та перегляд журналу. За результатами перевірки встановлено, що застосунок виконує основні функції, коректно зберігає дані в базі даних, фіксує результати обробки та дозволяє користувачеві контролювати роботу системи через інтерфейс.

До роботи також підготовлено комплект додаткових матеріалів. У додатку А подано технічне завдання на розробку програмного забезпечення для автоматичної обробки повідомлень електронної пошти. Додаток Б містить код основних модулів застосунку. У додатку В наведено опис програмного забезпечення, його структури та функціональних можливостей. Для користувача підготовлено інструкцію з експлуатації розробленого застосунку, яку розміщено в додатку Г. Перевірка працездатності програмного забезпечення виконувалася відповідно до програми та методики випробувань, наведених у додатку Д.

4 ОХОРОНА ПРАЦІ

4.1 Нормативно-правові вимоги до безпечної роботи розробника програмного забезпечення

Охорона праці є важливою складовою організації будь-якої професійної діяльності, у тому числі діяльності, пов’язаної з розробкою програмного забезпечення. На перший погляд робота програміста не належить до видів діяльності з підвищеною фізичною небезпекою, оскільки вона не передбачає використання важкого обладнання, виконання робіт на висоті або безпосереднього контакту з небезпечними виробничими механізмами. Проте тривале перебування за персональним комп’ютером, значне інтелектуальне навантаження, напружена робота із зоровою інформацією та необхідність постійної концентрації уваги можуть негативно впливати на стан здоров’я працівника. Саме тому під час організації робочого місця розробника необхідно враховувати вимоги охорони праці, виробничої санітарії, ергономіки, електробезпеки та пожежної безпеки.

Розробка програмного забезпечення для автоматичної обробки повідомлень електронної пошти виконується із застосуванням комп’ютерної техніки, засобів розробки програмного коду, систем керування базами даних, поштових сервісів, середовищ тестування та мережевих ресурсів. Основна частина робочого часу пов’язана з аналізом вимог, проєктуванням структури програми, написанням і налагодженням коду, роботою з базою даних Microsoft SQL Server, тестуванням взаємодії із поштовими серверами та перевіркою коректності обробки електронних повідомлень і вкладень. Усі ці види робіт потребують тривалого використання персонального комп’ютера, що зумовлює необхідність створення безпечних і комфортних умов праці.

Правове регулювання питань охорони праці в Україні здійснюється на основі системи законодавчих і нормативно-правових актів, які визначають

гарантії права працівника на безпечні та належні умови праці, обов’язки роботодавця щодо організації безпечного робочого середовища, а також вимоги до санітарно-гігієнічних умов і режиму праці. Загальні правові засади забезпечення права людини на безпечні умови праці закріплено в Конституції України [11]. Основні положення щодо організації охорони праці, прав і обов’язків роботодавців та працівників визначає Закон України «Про охорону праці» [12]. Окремі питання трудових відносин, робочого часу, часу відпочинку та гарантій працівників регулюються Кодексом законів про працю України [13]. Крім того, під час організації робочого місця необхідно враховувати вимоги законодавства про систему громадського здоров’я [14].

Відповідно до загальних вимог охорони праці роботодавець повинен забезпечити працівникам належні умови для виконання професійних обов’язків, організувати безпечне робоче середовище, забезпечити справність обладнання, провести інструктажі з питань охорони праці та контролювати дотримання встановлених правил. Працівник, зі свого боку, повинен виконувати вимоги нормативних документів, дотримуватися правил безпечної експлуатації комп’ютерної техніки, правильно користуватися обладнанням, своєчасно повідомляти про несправності та не виконувати дій, які можуть створити небезпеку для нього або інших осіб.

Під час виконання робіт із розробки програмного забезпечення особливе значення мають вимоги до організації робочого місця користувача персонального комп’ютера. Робоче місце повинно бути облаштоване таким чином, щоб зменшити навантаження на органи зору, опорно-руховий апарат і нервову систему працівника. Для цього необхідно забезпечити правильне розташування монітора, клавіатури, миші, робочого столу та крісла. Монітор має бути встановлений на зручній відстані від очей, екран не повинен створювати відблисків, а положення тіла працівника під час роботи має бути природним і не викликати надмірного напруження м’язів шиї, спини та рук.

Важливим нормативним аспектом є дотримання санітарно-гігієнічних вимог до приміщення, у якому виконується розробка програмного забезпечення.

Приміщення повинно мати достатній рівень природного або штучного освітлення, нормальні параметри мікроклімату, допустимий рівень шуму та належну вентиляцію. Недостатнє освітлення може спричиняти швидку втому очей, головний біль і зниження продуктивності праці. Надмірне освітлення або наявність відблисків на екрані монітора також негативно впливає на комфорт роботи, оскільки змушує працівника додатково напружувати зір.

Під час роботи розробника програмного забезпечення значне навантаження припадає на органи зору. Це пов’язано з необхідністю тривалого читання програмного коду, аналізу повідомлень про помилки, роботи з таблицями бази даних, вивчення технічної документації та перевірки елементів користувацького інтерфейсу. Тому нормативні вимоги до режиму праці та відпочинку мають особливе значення. Раціональна організація робочого часу передбачає чергування періодів інтенсивної роботи за комп’ютером із короткими перервами, під час яких працівник може зменшити навантаження на зір, змінити положення тіла та виконати прості фізичні вправи.

Не менш важливим є дотримання вимог електробезпеки. Робоче місце розробника обладнується персональним комп’ютером, монітором, мережевими пристроями, блоками живлення та іншими електричними засобами. Усе обладнання повинно бути справним, підключеним до електромережі з дотриманням правил безпеки та використовуватися відповідно до інструкцій виробника. Забороняється експлуатувати пристрої з пошкодженими кабелями, нестабільною роботою блоку живлення або іншими ознаками несправності. Для зменшення ризику пошкодження обладнання та втрати даних доцільно використовувати мережеві фільтри або джерела безперебійного живлення.

Пожежна безпека також є складовою нормативних вимог до організації робочого місця. У приміщенні, де використовується комп’ютерна техніка, необхідно забезпечити справність електромережі, не допускати перевантаження розеток, правильно розміщувати кабелі та уникати використання несправних електроприладів. Робоче місце не повинно перешкоджати доступу до евакуаційних виходів. Працівники мають бути ознайомлені з правилами

поведінки у разі виникнення пожежі або іншої надзвичайної ситуації.

Для розробника програмного забезпечення важливою є не лише фізична безпека, а й зниження психоемоційного навантаження. Робота над програмним продуктом потребує аналізу складних логічних зв'язків, пошуку помилок, перевірки взаємодії окремих модулів, роботи з обліковими даними, поштовими серверами та зовнішніми API. У процесі розробки можуть виникати ситуації, пов'язані з помилками підключення, некоректною обробкою вкладень, збоями бази даних або неправильним виконанням правил. Такі фактори можуть спричиняти нервове напруження, особливо за обмежених термінів виконання роботи. Тому важливо раціонально планувати розробку, розподіляти завдання на окремі етапи, документувати результати та виконувати тестування поступово.

Під час розробки програмного забезпечення для автоматичної обробки повідомлень електронної пошти необхідно також враховувати інформаційну безпеку як додатковий аспект організації роботи. У процесі тестування застосунку можуть використовуватися облікові дані поштових скриньок, тестові повідомлення, вкладені файли та параметри підключення до API. Хоча ці питання безпосередньо не належать до класичної охорони праці, правильна організація роботи з такими даними впливає на безпечність виконання професійних обов'язків. Розробник повинен уникати відкритого зберігання паролів, не використовувати реальні конфіденційні дані без потреби та дотримуватися правил безпечної роботи з інформацією.

Таким чином, нормативно-правові вимоги до безпечної роботи розробника програмного забезпечення спрямовані на створення умов, за яких працівник може ефективно виконувати професійні завдання без негативного впливу на здоров'я. Під час розробки програмного забезпечення для автоматичної обробки повідомлень електронної пошти необхідно забезпечити ергономічну організацію робочого місця, достатній рівень освітлення, належний мікроклімат, справність комп'ютерної техніки, дотримання електро- та пожежної безпеки, а також раціональний режим праці й відпочинку. Виконання цих вимог дозволяє зменшити ризики перевтоми, професійних захворювань і технічних небезпек, а

також сприяє підвищенню якості та продуктивності розробки програмного забезпечення.

4.2 Аналіз умов праці та виробничих факторів під час розробки ПЗ

Розробка програмного забезпечення для автоматичної обробки повідомлень електронної пошти виконується переважно в офісному або домашньому робочому середовищі з використанням персонального комп’ютера. Основними видами діяльності розробника є аналіз вимог до програмного продукту, проєктування архітектури застосунку, створення структури бази даних, написання програмного коду мовою C#, налаштування взаємодії з поштовими серверами, реалізація механізму обробки вкладень, тестування API-запитів, перевірка роботи користувацького інтерфейсу та документування результатів роботи.

На відміну від виробничих професій, діяльність розробника не пов’язана з інтенсивною фізичною працею або використанням небезпечного механічного обладнання. Проте вона має власні шкідливі та небезпечні фактори, які можуть негативно впливати на самопочуття, працездатність і здоров’я працівника. Такі фактори пов’язані насамперед із тривалим перебуванням у сидячому положенні, постійною роботою з екраном монітора, високим рівнем розумового навантаження, необхідністю тривалої концентрації уваги та експлуатацією електричного обладнання.

Робоче місце розробника програмного забезпечення зазвичай складається з персонального комп’ютера або ноутбука, монітора, клавіатури, маніпулятора типу «миша», робочого столу, крісла, мережевого обладнання та периферійних пристроїв. Додатково під час розробки можуть використовуватися сервер бази даних Microsoft SQL Server, локальне або віддалене середовище тестування, поштові облікові записи, засоби налагодження програмного коду, системи контролю версій та інструменти для перевірки HTTP-запитів. Комплексність робочого процесу вимагає від розробника одночасної роботи з кількома програмними середовищами та джерелами інформації.

Одним із найбільш суттєвих факторів, що впливає на умови праці розробника, є зорове навантаження. Під час створення програмного забезпечення працівник протягом тривалого часу аналізує текст програмного коду, переглядає структуру бази даних, читає технічну документацію, перевіряє журнали помилок, тестує інтерфейс користувача та контролює результати обробки електронних повідомлень. Постійна концентрація погляду на екрані монітора може спричиняти втому очей, сухість, зниження чіткості зору, головний біль та загальне зниження працездатності.

Зорове напруження посилюється у випадках, коли робоче місце має недостатнє або надмірне освітлення. Якщо освітлення приміщення є слабким, очі змушені постійно адаптуватися до контрасту між екраном і навколишнім середовищем. Якщо ж освітлення надто яскраве або на екрані з’являються відблиски, це також призводить до дискомфорту та швидкої втоми. Тому під час роботи з програмним кодом, базою даних і графічним інтерфейсом застосунку важливо забезпечити рівномірне освітлення робочої зони та правильне розташування монітора відносно джерел світла.

Іншим важливим фактором є статичне навантаження на опорно-руховий апарат. Розробник більшу частину робочого часу проводить у сидячому положенні, що може спричиняти напруження м’язів шиї, спини, плечового поясу, попереку та рук. Неправильно підібране крісло, незручна висота столу, неправильне положення монітора або клавіатури можуть призводити до порушення постави, болю в спині, оніміння кінцівок і загального фізичного дискомфорту. Особливо це проявляється під час тривалих періодів налагодження програмного коду або пошуку помилок, коли працівник може залишатися в одній позі протягом значного часу.

Під час розробки застосунку для автоматичної обробки електронної пошти додаткове навантаження створює необхідність постійного перемикання між різними завданнями. Розробник працює з кодом, SQL-запитами, налаштуваннями поштового підключення, тестовими листами, вкладеннями, API-запитами, журналами помилок і вікнами інтерфейсу. Такий характер роботи

вимагає підвищеної уваги та здатності швидко аналізувати причини некоректної роботи системи. Наприклад, помилка може виникнути через неправильні параметри IMAP-підключення, недоступність поштового сервера, неправильну фільтрацію листів, помилку збереження файлу або некоректну відповідь зовнішнього API.

Психоемоційне навантаження є характерним фактором для роботи програміста. Воно виникає через необхідність розв’язання складних логічних задач, пошуку помилок, контролю великої кількості деталей і дотримання встановлених термінів виконання роботи. Під час розробки даного застосунку розробник повинен забезпечити коректну взаємодію кількох компонентів: поштового модуля, механізму правил, модуля обробки вкладень, бази даних, API-клієнта та інтерфейсу користувача. Помилки в одному з цих компонентів можуть впливати на роботу всієї системи, що підвищує відповідальність розробника та рівень нервового напруження.

Особливість розроблюваного програмного забезпечення полягає також у роботі з електронними повідомленнями та вкладеними файлами. У процесі тестування можуть використовуватися різні типи файлів, зокрема документи, таблиці, PDF-файли, архіви або зображення. Це потребує уважності під час налаштування правил обробки, перевірки розширень, шляхів збереження та результатів передавання файлів до зовнішніх сервісів. Неправильна обробка вкладення може призвести до втрати файлу, дублювання даних або помилкового передавання інформації.

До факторів виробничого середовища також належить мікроклімат приміщення. Температура, вологість і рух повітря безпосередньо впливають на самопочуття працівника. Надмірно висока температура може спричиняти швидку втому, сонливість і зниження концентрації уваги. Занадто низька температура створює фізичний дискомфорт і також знижує продуктивність праці. Недостатня вологість повітря може посилювати сухість очей, що особливо відчутно під час тривалої роботи за монітором. Тому для комфортної роботи розробника необхідно підтримувати стабільний мікроклімат у приміщенні.

Шумове навантаження під час розробки програмного забезпечення зазвичай не є значним, однак його також потрібно враховувати. Джерелами шуму можуть бути системні блоки, вентилятори охолодження, принтери, кондиціонери, мережеве обладнання або сторонні розмови в приміщенні. Навіть помірний фоновий шум може заважати концентрації уваги, особливо під час аналізу програмного коду або пошуку складних помилок. Для роботи, що потребує високої точності мислення, бажано забезпечити спокійне середовище без постійних відволікань.

Під час експлуатації комп'ютерної техніки існують ризики, пов'язані з електробезпекою. Персональний комп'ютер, монітор, маршрутизатор, зарядний пристрій ноутбука та інше обладнання підключаються до електромережі. Несправні кабелі, перевантажені розетки, використання неякісних подовжувачів або пошкодження електроприладів можуть створювати небезпеку ураження електричним струмом або виникнення пожежі. Тому робоче місце повинно бути обладнане справними електричними пристроями, а кабелі мають бути розташовані так, щоб не створювати перешкод для пересування та не пошкоджуватися під час експлуатації.

Окремо слід звернути увагу на ризик втрати даних під час розробки. Хоча цей фактор не є безпосередньо фізично небезпечним для працівника, він може створювати додаткове психоемоційне навантаження та впливати на організацію праці. Під час роботи над програмним забезпеченням можуть зберігатися вихідний код, налаштування бази даних, тестові облікові записи, приклади електронних листів, вкладення та технічна документація. Втрата цих даних через збій електроживлення, помилку користувача або несправність обладнання може призвести до необхідності повторного виконання значного обсягу роботи. Тому важливим є використання систем контролю версій, резервного копіювання та джерел безперебійного живлення.

Під час роботи над програмним забезпеченням для автоматичної обробки електронної пошти розробник може взаємодіяти з обліковими даними поштових скриньок, токенами доступу до API та іншою службовою інформацією.

Неправильне поводження з такими даними може призвести до їх розголошення або несанкціонованого доступу до поштових повідомлень. Тому умови праці повинні передбачати не лише фізичну безпеку, а й організований підхід до захисту службової інформації: використання тестових облікових записів, обмеження доступу до конфігураційних файлів, уникнення зберігання паролів у відкритому вигляді та контроль за файлами, що використовуються під час тестування.

Отже, під час розробки програмного забезпечення для автоматичної обробки повідомлень електронної пошти на працівника можуть впливати такі основні фактори: зорове навантаження, статичне навантаження на опорно-руховий апарат, психоемоційне напруження, параметри освітлення, мікроклімат приміщення, шум, ризики електробезпеки, пожежної безпеки та організаційні ризики, пов’язані з втратою або неправильним використанням службових даних. Аналіз цих факторів є необхідним для визначення заходів, які дозволять створити безпечні, комфортні та продуктивні умови праці під час виконання робіт із проєктування, розробки й тестування програмного забезпечення.

4.3 Заходи з охорони праці під час проєктування, розробки та тестування програмного забезпечення

Забезпечення безпечних і комфортних умов праці під час розробки програмного забезпечення є необхідною умовою ефективного виконання робіт, пов’язаних із проєктуванням, програмуванням, тестуванням і документуванням програмного продукту. Оскільки розробка застосунку для автоматичної обробки повідомлень електронної пошти виконується переважно за персональним комп’ютером, основні заходи з охорони праці мають бути спрямовані на зниження навантаження на органи зору, опорно-руховий апарат, нервову систему, а також на забезпечення електричної, пожежної та інформаційної безпеки.

Першочергове значення має правильна організація робочого місця розробника. Робоче місце повинно забезпечувати зручне розташування

комп’ютерної техніки, достатній простір для роботи та можливість підтримання правильної робочої пози. Робочий стіл має бути достатньо просторим для розміщення монітора, клавіатури, миші, документів та інших необхідних засобів. Поверхня столу бажано повинна бути матовою, щоб не створювати відблисків, які можуть додатково навантажувати органи зору. Висота столу повинна дозволяти розташовувати руки у природному положенні без надмірного напруження плечей і передпліч.

Робоче крісло повинно бути ергономічним і забезпечувати підтримку спини під час тривалої роботи. Бажано використовувати крісло з можливістю регулювання висоти сидіння, нахилу спинки та положення підлокітників. Під час роботи спина повинна спиратися на спинку крісла, плечі мають бути розслабленими, а стопи повинні повністю розташовуватися на підлозі або на спеціальній підставці. Таке положення дозволяє зменшити навантаження на хребет, поперек, шию та плечовий пояс. Неправильна поза під час тривалої роботи може призводити до болю в спині, втоми м’язів і зниження працездатності, тому питання ергономіки є одним із ключових.

Монітор необхідно розташовувати таким чином, щоб верхня межа екрана знаходилася приблизно на рівні очей або трохи нижче. Відстань від очей до екрана має бути комфортною для читання тексту без необхідності нахилятися вперед або напружувати зір. Екран не повинен бути розташований навпроти яскравого джерела світла або так, щоб на ньому утворювалися відблиски. Під час роботи з програмним кодом, SQL-запитами, журналами обробки повідомлень і технічною документацією важливо забезпечити чітке зображення, достатній розмір шрифту та оптимальні параметри яскравості й контрастності.

Для зменшення навантаження на органи зору необхідно правильно організувати освітлення робочого місця. Приміщення повинно мати достатній рівень природного або штучного освітлення. Бажано, щоб світло було рівномірним і не створювало різких контрастів між екраном монітора та навколишнім середовищем. У разі використання настільної лампи її слід розташовувати так, щоб світло не потрапляло безпосередньо в очі та не

відбивалося від екрана. Для роботи у вечірній час доцільно використовувати джерела світла з м’яким рівномірним освітленням.

Під час тривалої роботи за комп’ютером доцільно дотримуватися режиму праці та відпочинку. Розробка програмного забезпечення потребує високої концентрації уваги, тому безперервна робота протягом тривалого часу може призводити до перевтоми, зниження уважності та збільшення кількості помилок. Рекомендується робити короткі перерви після кожних 45–60 хвилин роботи. Під час таких перерв бажано змінити положення тіла, пройтися, виконати прості вправи для шиї, спини, плечей і кистей рук. Це дозволяє зменшити статичне навантаження на опорно-руховий апарат і підтримувати нормальний рівень працездатності.

Для профілактики зорової втоми можна застосовувати правило «20-20-20»: приблизно кожні 20 хвилин роботи переводити погляд на об’єкт, розташований на відстані близько 6 метрів, на 20 секунд. Така вправа допомагає розслабити очні м’язи та зменшити напруження, яке виникає під час постійної концентрації погляду на екрані. Також доцільно періодично моргати частіше, уникати надмірної яскравості екрана та використовувати комфортний розмір шрифту в середовищі розробки.

Під час розробки застосунку для автоматичної обробки повідомлень електронної пошти важливо забезпечити раціональну організацію робочого процесу. Програмний продукт складається з кількох функціональних частин: модуля підключення до поштової скриньки, механізму фільтрації листів, модуля обробки вкладень, бази даних, інтеграції із зовнішнім API та користувацького інтерфейсу. Щоб зменшити психоемоційне навантаження, доцільно розподіляти роботу на окремі етапи, послідовно реалізовувати й тестувати кожен модуль, вести документацію та використовувати систему контролю версій. Такий підхід зменшує ризик накопичення помилок і полегшує пошук причин некоректної роботи програми.

Особливу увагу слід приділяти організації тестування. Тестування взаємодії з поштовими серверами, завантаження вкладень, збереження файлів і

передавання даних через API може супроводжуватися різними помилками: неправильними параметрами підключення, відсутністю доступу до мережі, недоступністю зовнішнього сервісу, некоректним форматом вкладення або помилками бази даних. Для зменшення нервового напруження та підвищення якості роботи доцільно використовувати тестові поштові скриньки, тестові API-адреси, контрольовані набори повідомлень і заздалегідь підготовлені сценарії перевірки. Це дозволяє уникнути хаотичного тестування й забезпечує прогнозований процес пошуку помилок.

Необхідно також підтримувати належний мікроклімат у робочому приміщенні. Температура, вологість і вентиляція мають забезпечувати комфортні умови для тривалої розумової праці. Надмірна спека, холод або сухе повітря можуть знижувати концентрацію, підвищувати втому та негативно впливати на загальне самопочуття. Доцільно регулярно провітрювати приміщення, підтримувати оптимальну температуру, уникати тривалого перебування в задушливому середовищі та за потреби використовувати засоби кондиціонування або зволоження повітря.

Для зменшення впливу шуму необхідно організувати робоче місце в спокійному середовищі. Сторонні розмови, шум обладнання, гучна музика або інші відволікальні фактори можуть негативно впливати на концентрацію уваги, особливо під час написання програмного коду або пошуку складних помилок. За можливості робоче приміщення має бути відокремленим від джерел постійного шуму. Якщо повністю усунути шум неможливо, доцільно використовувати організаційні заходи: планування складних завдань на час найменшого навантаження, зменшення кількості відволікань і чітке структурування робочого процесу.

Важливою складовою охорони праці є дотримання правил електробезпеки. Усе комп’ютерне обладнання, яке використовується під час розробки програмного забезпечення, повинно бути справним і підключеним до електромережі відповідно до вимог безпеки. Не допускається використання пошкоджених кабелів, несправних розеток, перевантажених подовжувачів або

обладнання з ознаками перегрівання. Кабелі живлення та мережеві кабелі повинні бути розташовані таким чином, щоб не заважати пересуванню та не створювати ризику випадкового пошкодження.

Для захисту обладнання та даних доцільно використовувати мережеві фільтри або джерела безперебійного живлення. Це особливо важливо під час роботи з базою даних, тестовими повідомленнями, вкладеннями та вихідним кодом. Раптове вимкнення електроенергії може призвести до втрати незбережених змін, пошкодження файлів або переривання процесу тестування. Використання джерела безперебійного живлення дозволяє безпечно завершити роботу системи або зберегти важливі дані.

Заходи пожежної безпеки передбачають правильну експлуатацію електрообладнання, недопущення перевантаження електромережі та підтримання порядку на робочому місці. Не слід розміщувати легкозаймисті матеріали поблизу джерел тепла або електричних пристроїв. Робоче місце повинно бути організоване таким чином, щоб не блокувати доступ до виходу з приміщення. У разі роботи в офісному середовищі працівник має знати розташування первинних засобів пожежогасіння та порядок дій у разі виникнення пожежі.

Під час розробки програмного забезпечення для автоматичної обробки повідомлень електронної пошти необхідно враховувати й заходи інформаційної безпеки. У процесі тестування можуть використовуватися облікові дані поштових скриньок, паролі, токени доступу до API, тестові листи та вкладені файли. Такі дані не слід зберігати у відкритому вигляді в програмному коді або передавати стороннім особам. Доцільно використовувати тестові облікові записи, обмежувати доступ до конфігураційних файлів, застосовувати засоби захисту паролів і не використовувати реальні конфіденційні документи без потреби.

Окремим заходом є організація резервного копіювання та контролю версій. Під час розробки програмного забезпечення необхідно регулярно зберігати вихідний код, структуру бази даних, технічну документацію та інші

важливі матеріали. Використання системи контролю версій дозволяє відстежувати зміни, повертатися до попередніх версій і зменшувати ризик втрати результатів роботи. Резервне копіювання бази даних і тестових матеріалів також сприяє зменшенню стресу та підвищенню надійності процесу розробки.

Для підтримання психологічного комфорту розробника доцільно застосовувати раціональне планування робочого часу. Завдання з розробки слід розподіляти за складністю та виконувати поступово: спочатку реалізувати базову структуру застосунку, потім модуль роботи з поштою, далі механізм правил, обробку вкладень, інтеграцію з API та журналювання. Такий порядок дозволяє краще контролювати прогрес, своєчасно виявляти проблеми й уникати надмірного перевантаження наприкінці роботи.

Таким чином, забезпечення безпечних умов праці під час проєктування, розробки та тестування програмного забезпечення потребує комплексного підходу. Він включає ергономічну організацію робочого місця, правильне розташування обладнання, дотримання режиму праці та відпочинку, зниження зорового й статичного навантаження, підтримання комфортного мікроклімату, дотримання правил електро- та пожежної безпеки, а також організацію захисту службових даних і резервного копіювання. Виконання зазначених заходів дозволяє зменшити негативний вплив виробничих факторів, підтримати працездатність розробника та забезпечити якісне виконання робіт зі створення програмного забезпечення для автоматичної обробки повідомлень електронної пошти.

ВИСНОВКИ

У кваліфікаційній роботі було розв’язано практичну задачу інженерії програмного забезпечення, пов’язану з розробкою програмного забезпечення для автоматичної обробки повідомлень електронної пошти. Актуальність цієї задачі зумовлена тим, що в багатьох організаціях електронна пошта залишається одним із основних каналів отримання документів, рахунків, звітів, заявок та іншої службової інформації. Ручна обробка таких повідомлень потребує значних витрат часу, залежить від уважності користувача та може призводити до помилок, повторної обробки листів або втрати важливої інформації. Отже, мету кваліфікаційної роботи досягнуто.

Для досягнення поставленої мети було виконано такі завдання:

– проаналізовано практичну задачу автоматизації процесу обробки повідомлень електронної пошти та вкладених файлів;

– розглянуто існуючі програмні засоби й сервіси, які можуть використовуватися для роботи з електронною поштою та автоматизації обробки повідомлень;

– обґрунтовано доцільність розробки власного програмного забезпечення з урахуванням переваг і обмежень наявних аналогів;

– визначено функціональні та нефункціональні вимоги до програмного забезпечення;

– сформовано основні сценарії використання системи та побудовано модель варіантів використання;

– розроблено архітектуру програмного забезпечення з урахуванням шаблону MVVM і багатошарового підходу;

– розроблено проєкт програмного забезпечення, зокрема структуру класів, основні сервіси, модель бази даних і алгоритм обробки повідомлень;

– реалізовано програмне забезпечення з використанням C#, WPF, Entity

Framework Core, Microsoft SQL Server, MailKit та засобів взаємодії із зовнішніми API;

– забезпечено механізми захисту облікових даних, безпечної обробки вкладень і запобігання повторній обробці повідомлень;

– підготовлено програму та методику випробувань програмного забезпечення;

– виконано перевірку працездатності розробленого застосунку та проаналізовано отримані результати.

Практичне значення роботи полягає в можливості використання розробленого програмного забезпечення для зменшення обсягу ручних операцій під час обробки електронних листів і вкладених файлів. Застосунок може бути корисним в організаціях, де електронна пошта регулярно використовується для отримання документів, рахунків, звітів або інших службових файлів. Подальший розвиток програмного забезпечення може бути пов’язаний із розширенням умов правил, додаванням ролей користувачів, підтримкою складніших сценаріїв інтеграції із зовнішніми системами та вдосконаленням аналітики журналу обробки.

Отже, мету кваліфікаційної роботи повністю досягнуто. У результаті виконання роботи було розроблено програмне забезпечення для автоматичної обробки повідомлень електронної пошти, яке забезпечує отримання листів, перевірку їх за правилами, обробку вкладень, виконання налаштованих дій, збереження результатів у базі даних і контроль роботи системи через журнал подій. Розроблений застосунок відповідає поставленим вимогам.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Gunelius, S. Ultimate Guide to Email Marketing for Business. Irvine: Entrepreneur Press, 2018, 226 p.

2. What is Outlook? URL: https://support.microsoft.com/en-us/outlook/training/what-is-outlook (дата звернення: 20.03.2026).

3. Create rules to filter your emails. URL: https://support.google.com/mail/answer/6579?hl=en (дата звернення: 21.03.2026).

4. What is Power Automate? URL: https://learn.microsoft.com/en-us/power-automate/flow-types (дата звернення: 18.03.2026).

5. What is Zapier? URL: https://help.zapier.com/hc/en-us/articles/37518970271245-What-is-Zapier (дата звернення: 21.03.2026).

6. What Mailparser can do for you. URL: https://help.mailparser.io/hc/en-us/articles/16253436537748-What-Mailparser-can-do-for-you (дата звернення: 18.03.2026).

7. Send documents to Parseur using the API. URL: https://help.parseur.com/en/articles/3566112-send-documents-to-parseur-using-the-api (дата звернення: 19.03.2026).

8. IMAP Protocol Overview. URL: https://imap.org/imap-protocol/ (дата звернення: 22.03.2026).

9. MimeKit and MailKit Introduction. URL: https://mimekit.net/docs/html/Introduction.htm (дата звернення: 21.03.2026).

10. MVVM with Clean Architecture. URL: https://sanjanahumanintech.medium.com/mvvm-with-clean-architecture-in-react-native-a-detailed-guide-e5e25b815db0 (дата звернення: 22.03.2026).

11. Конституція України : Закон України від 28.06.1996 № 254к/96-ВР. Відомості Верховної Ради України. 1996. № 30. Ст. 141.

12. Про охорону праці : Закон України від 14.10.1992 № 2694-XII.

Відомості Верховної Ради України. 1992. № 49. Ст. 668.

13. Кодекс законів про працю України : Кодекс України від 10.12.1971 № 322-VIII. Відомості Верховної Ради УРСР. 1971. Додаток до № 50. Ст. 375.

14. Про систему громадського здоров’я : Закон України від 06.09.2022 № 2573-IX. Відомості Верховної Ради України. 2022. № 45. Ст. 273.

ДОДАТОК А. Технічне завдання на розробку програмного забезпечення для автоматичної обробки повідомлень електронної пошти

Вступ

Повна назва проєкту: «Розробка програмного забезпечення для автоматичної обробки повідомлень електронної пошти». Сферою застосування програмного забезпечення є автоматизація обробки вхідної електронної кореспонденції в організаціях, де через електронну пошту регулярно надходять повідомлення з документами, звітами, заявками, рахунками, актами або іншими вкладеними файлами.

1 Підстави для розробки

Підставою для розробки програмного забезпечення є завдання на виконання кваліфікаційної роботи бакалавра за спеціальністю 121 «Інженерія програмного забезпечення». Необхідність розробки зумовлена потребою в автоматизації повторюваних операцій, пов’язаних з обробкою електронної пошти. У багатьох організаціях працівники вручну переглядають вхідні повідомлення, визначають потрібні листи за темою або відправником, завантажують вкладення, зберігають файли та передають їх до інших інформаційних систем. Такий процес потребує значних витрат часу, залежить від уважності користувача та може супроводжуватися помилками.

2 Призначення розробки

2.1 Експлуатаційне призначення

Експлуатаційне призначення програмного забезпечення полягає в автоматизації процесу обробки повідомлень електронної пошти та вкладених файлів.

Програмне забезпечення призначене для використання працівниками організацій, технічними спеціалістами, адміністраторами інформаційних систем або іншими відповідальними особами, які виконують регулярну обробку вхідної електронної кореспонденції.

Застосунок може використовуватися в організаціях, де необхідно автоматично отримувати листи від визначених відправників, завантажувати вкладення певного типу, зберігати їх у встановлених каталогах або передавати до зовнішніх інформаційних систем через API.

2.2 Функціональне призначення

Функціональне призначення програмного забезпечення полягає в забезпеченні користувача засобами для налаштування поштових скриньок, створення правил обробки повідомлень, автоматичного отримання листів, завантаження вкладень, виконання дій із файлами та перегляду журналу результатів обробки.

Програмне забезпечення повинно забезпечувати виконання таких основних функцій:

– додавання, редагування, перегляд і видалення поштових облікових записів;

– налаштування параметрів підключення до поштової скриньки;

– ручний запуск перевірки електронної пошти;

– автоматичну перевірку поштової скриньки з визначеним інтервалом;

– отримання нових повідомлень електронної пошти;

– збереження основної інформації про отримані повідомлення в базі даних;

– створення, редагування, перегляд і видалення правил обробки повідомлень;

– фільтрацію листів за відправником, темою, датою, наявністю вкладень та іншими ознаками;

– завантаження вкладень із повідомлень, які відповідають заданим правилам;

– фільтрацію вкладень за назвою, розширенням і розміром;

– збереження вкладень у визначений користувачем каталог;

– передавання вкладень або службових даних повідомлення до зовнішнього API;

– ведення журналу обробки повідомлень, вкладень і виконаних дій;

– перегляд інформації про оброблені повідомлення;

– перегляд інформації про завантажені вкладення;

– пошук і фільтрацію записів журналу;

– відображення інформації про помилки, що виникли під час обробки.

3 Вимоги до програмного забезпечення

3.1 Вимоги до функціональних характеристик

Програмним забезпеченням буде користуватися відповідальна особа або технічний спеціаліст, який налаштовує поштові скриньки, правила обробки повідомлень і контролює результати роботи системи.

Основний сценарій роботи користувача полягає в налаштуванні поштового облікового запису, створенні правил відбору повідомлень, визначенні дій із вкладеннями та перегляді журналу обробки.

Для поштового облікового запису повинні бути доступні такі операції:

– додавання поштового облікового запису;

– редагування поштового облікового запису;

– видалення поштового облікового запису;

– перегляд поштових облікових записів.

Для правил обробки повідомлень повинні бути доступні такі операції:

– додавання правила обробки;

– редагування правила обробки;

– видалення правила обробки;

– перегляд правил обробки.

Для отримання та обробки повідомлень повинні бути доступні такі функції:

– підключення до поштової скриньки за допомогою протоколу IMAP;

– отримання нових повідомлень із вибраної поштової папки;

– перевірка повідомлень за заданими правилами;

– запобігання повторній обробці вже опрацьованих листів;

– збереження інформації про повідомлення в базі даних;

– визначення наявності вкладень;

– завантаження вкладень, які відповідають умовам правила;

– збереження вкладень у локальний або мережевий каталог;

– передавання вкладень або даних листа до зовнішнього API;

– фіксація результатів обробки в журналі.

Для журналу обробки повинні бути доступні такі операції:

– перегляд журналу обробки;

– фільтрація журналу;

– перегляд деталей запису журналу.

3.2 Вимоги до надійності

Програмне забезпечення повинно забезпечувати коректну роботу в умовах можливих помилок підключення до поштового сервера, бази даних або зовнішнього API.

Система повинна:

– фіксувати помилки підключення до поштової скриньки;

– повідомляти користувача про неправильні параметри доступу;

– не допускати повторної обробки одного й того самого повідомлення;

– зберігати статуси обробки повідомлень і вкладень;

– фіксувати помилки збереження файлів;

– фіксувати помилки HTTP-запитів до зовнішніх API;

– забезпечувати збереження журналу обробки;

– не завершувати роботу аварійно у разі виникнення помилки під час обробки окремого повідомлення.

3.3 Вимоги до інформаційної безпеки

Програмне забезпечення повинно забезпечувати захист службових даних, які використовуються під час роботи із поштовими скриньками та зовнішніми сервісами.

Необхідно передбачити:

– обмеження доступу до параметрів підключення;

– недопущення зберігання паролів у відкритому вигляді, якщо це технічно можливо;

– використання захищеного з’єднання з поштовим сервером;

– перевірку імен файлів перед збереженням вкладень;

– заборону автоматичного запуску завантажених вкладень;

– можливість використання тестових облікових записів під час перевірки роботи системи.

3.4 Вимоги до складу й параметрів технічних засобів

Для роботи програмного забезпечення необхідні такі технічні засоби:

– операційна система Windows 10 або новіша;

– процесор із тактовою частотою не менше 2,0 ГГц;

– оперативна пам’ять не менше 4 ГБ;

– не менше 500 МБ вільного місця на диску для встановлення програми та збереження службових файлів;

– доступ до мережі Інтернет або локальної мережі;

– доступ до поштового сервера з підтримкою протоколу IMAP;

– доступ до Microsoft SQL Server;

– за потреби доступ до зовнішнього API для передавання файлів або службових даних.

3.5 Вимоги до інформаційної та програмної сумісності

Програмне забезпечення повинно бути сумісним із:

– операційними системами сімейства Windows;

– платформою .NET;

– Microsoft SQL Server;

– поштовими серверами, що підтримують протокол IMAP;

– бібліотекою MailKit для роботи з електронною поштою;

– зовнішніми сервісами, які приймають HTTP-запити через API;

– файловою системою Windows для збереження вкладень.

Інформація, яка зберігається в базі даних, повинна включати відомості про поштові облікові записи, правила обробки, повідомлення, вкладення, дії та журнал результатів обробки.

4 Вимоги до програмної документації

У процесі розробки програмного забезпечення повинна бути підготовлена така документація:

– технічне завдання на розробку програмного забезпечення;

– опис структури бази даних;

– опис основних функціональних модулів;

– інструкція користувача;

– опис порядку встановлення та налаштування програмного забезпечення;

– матеріали щодо тестування основних функцій застосунку.

5 Стадії та етапи роботи

Основні етапи розробки наведено у таблиці А.1.

Таблиця А.1 – Етапи розробки програмного забезпечення

№	Назва етапу	Термін
1	Дослідження практичної задачі та предметної області розробки	27.02.2026
2	Визначення та систематизація вимог до програмного забезпечення	06.03.2026
3	Проєктування загальної архітектури програмної системи	27.03.2026
4	Детальне проєктування компонентів програмного забезпечення	10.04.2026
5	Програмна реалізація, перевірка працездатності та тестування розробленого застосунку	01.05.2026

Порядок контролю і приймання

Контроль якості програмного забезпечення здійснюється шляхом перевірки відповідності реалізованих функцій вимогам технічного завдання.

Під час приймання програмного забезпечення необхідно перевірити:

– можливість додавання та редагування поштового облікового запису;

– коректність підключення до поштової скриньки;

– отримання повідомлень із поштового сервера;

– роботу правил фільтрації повідомлень;

– завантаження вкладень;

– збереження вкладень у визначений каталог;

– передавання файлів або службових даних до зовнішнього API;

– запобігання повторній обробці повідомлень;

– збереження інформації про листи та вкладення в базі даних;

– ведення журналу обробки;

– відображення помилок і статусів виконання;

– працездатність користувацького інтерфейсу.

Програмне забезпечення вважається таким, що відповідає технічному завданню, якщо воно забезпечує виконання основних функцій автоматичної обробки повідомлень електронної пошти, коректно працює з базою даних, дозволяє налаштовувати правила обробки та фіксує результати виконаних дій у журналі.

ДОДАТОК Б. Вихідні тексти програмних модулів

```
using EmailProcessor.Infrastructure.Data;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;

namespace EmailProcessor.App.Services;

/// <summary>
/// Background service that wakes up periodically and runs a
processing cycle
/// for every active email account whose polling interval has
elapsed.
///
/// The service is started and stopped from the WPF UI via
<see cref="Start"/>
/// and <see cref="Stop"/>. Processing can also be triggered
manually via
/// <see cref="TriggerNowAsync"/>.
/// </summary>
public sealed class BackgroundPollingService : IDisposable
{
    private readonly EmailProcessingOrchestrator
_orchestrator;
    private readonly IDbContextFactory<AppDbContext>
_dbFactory;
    private readonly ILogger<BackgroundPollingService>
_logger;

    private CancellationTokenSource? _cts;
    private Task? _loopTask;

    // Tracks the last time each account (keyed by Id) was
checked
    private readonly Dictionary<int, DateTime> _lastChecked =
new();

    // How often the main loop wakes up to check intervals
(every 30 s)
    private static readonly TimeSpan WakeInterval =
TimeSpan.FromSeconds(30);

    public bool   IsRunning  { get; private set; }
    public string StatusText { get; private set; } =
"Stopped";

    /// <summary>Fired on the UI thread whenever IsRunning or
StatusText changes.</summary>
    public event EventHandler? StateChanged;
```

```
    public BackgroundPollingService(
        EmailProcessingOrchestrator orchestrator,
        IDbContextFactory<AppDbContext> dbFactory,
        ILogger<BackgroundPollingService> logger)
    {
        _orchestrator = orchestrator;
        _dbFactory    = dbFactory;
        _logger       = logger;
    }

    public void Start()
    {
        if (IsRunning) return;

        _cts      = new CancellationTokenSource();
        _loopTask = RunLoopAsync(_cts.Token);
        IsRunning = true;
        StatusText = "Running";
        NotifyStateChanged();
        _logger.LogInformation("Background polling service
started.");
    }

    public void Stop()
    {
        if (!IsRunning) return;

        _cts?.Cancel();
        IsRunning  = false;
        StatusText = "Stopped";
        NotifyStateChanged();
        _logger.LogInformation("Background polling service
stopped.");
    }

    /// <summary>Immediately processes all active accounts
regardless of their polling interval.</summary>
    public async Task TriggerNowAsync(CancellationToken ct =
default)
    {
        StatusText = "Running now…";
        NotifyStateChanged();

        await using var db = await
_dbFactory.CreateDbContextAsync(ct);
        var accounts = await db.EmailAccounts
            .Where(a => a.IsActive)
            .ToListAsync(ct);

        foreach (var account in accounts)
        {
            ct.ThrowIfCancellationRequested();
```

```
            await _orchestrator.ProcessAccountAsync(account,
ct);
            _lastChecked[account.Id] = DateTime.UtcNow;
        }

        StatusText = IsRunning ? "Running" : "Stopped";
        NotifyStateChanged();
    }

    private async Task RunLoopAsync(CancellationToken ct)
    {
        while (!ct.IsCancellationRequested)
        {
            try
            {
                await ProcessDueAccountsAsync(ct);
            }
            catch (OperationCanceledException)
            {
                break;
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Unexpected error in
polling loop.");
            }

            try
            {
                await Task.Delay(WakeInterval, ct);
            }
            catch (OperationCanceledException)
            {
                break;
            }
        }
    }

    private async Task
ProcessDueAccountsAsync(CancellationToken ct)
    {
        await using var db = await
_dbFactory.CreateDbContextAsync(ct);
        var accounts = await db.EmailAccounts
            .Where(a => a.IsActive)
            .ToListAsync(ct);

        foreach (var account in accounts)
        {
            ct.ThrowIfCancellationRequested();

            var due = !_lastChecked.TryGetValue(account.Id,
out var last)
```

```
                || (DateTime.UtcNow - last).TotalSeconds >=
account.PollingIntervalSeconds;

            if (!due) continue;

            _logger.LogDebug("Polling account '{Name}'.",
account.Name);
            StatusText = $"Checking '{account.Name}'…";
            NotifyStateChanged();

            await _orchestrator.ProcessAccountAsync(account,
ct);
            _lastChecked[account.Id] = DateTime.UtcNow;
        }

        StatusText = "Running";
        NotifyStateChanged();
    }

    private void NotifyStateChanged() =>
        StateChanged?.Invoke(this, EventArgs.Empty);

    public void Dispose()
    {
        Stop();
        _cts?.Dispose();
    }
}

using EmailProcessor.Domain.Enums;
using EmailProcessor.Domain.Interfaces;
using EmailProcessor.Domain.Models;
using EmailProcessor.Infrastructure.Data;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;

namespace EmailProcessor.App.Services;

/// <summary>
/// Orchestrates one complete processing cycle for a single
email account:
///   1. Fetch new messages via IMAP
///   2. Match each message against active processing rules
///   3. Process attachments according to matched rule
actions
///   4. Persist message and attachment metadata
///   5. Write audit log entries
///   6. Optionally mark messages as read / move them
/// </summary>
public sealed class EmailProcessingOrchestrator
{
    private readonly IEmailService _emailService;
    private readonly IRuleEngine _ruleEngine;
```

```
    private readonly IAttachmentService _attachmentService;
    private readonly ILoggingService _loggingService;
    private readonly IDbContextFactory<AppDbContext>
_dbFactory;
    private readonly ILogger<EmailProcessingOrchestrator>
_logger;

    public EmailProcessingOrchestrator(
        IEmailService emailService,
        IRuleEngine ruleEngine,
        IAttachmentService attachmentService,
        ILoggingService loggingService,
        IDbContextFactory<AppDbContext> dbFactory,
        ILogger<EmailProcessingOrchestrator> logger)
    {
        _emailService      = emailService;
        _ruleEngine        = ruleEngine;
        _attachmentService = attachmentService;
        _loggingService    = loggingService;
        _dbFactory         = dbFactory;
        _logger            = logger;
    }

    //  Public entry point —

    /// <summary>
    /// Runs a complete polling cycle for the specified
account.
    /// Returns a summary of the processing results.
    /// </summary>
    public async Task<ProcessingResult> ProcessAccountAsync(
        EmailAccount account, CancellationToken ct = default)
    {
        var result = new ProcessingResult { AccountId =
account.Id };

        _logger.LogInformation("Starting processing cycle for
account '{Name}'.", account.Name);

        //  1. Load already-processed UIDs to avoid
duplicates —
        await using var db = await
_dbFactory.CreateDbContextAsync(ct);

        var processedUids = db.EmailMessages
            .Where(m => m.AccountId == account.Id)
            .Select(m => m.ImapUid)
            .ToHashSet();

        //  2. Load active processing rules
        var rules = await db.ProcessingRules
            .Include(r => r.Actions)
            .Where(r => r.IsActive)
```

```
            .OrderBy(r => r.Priority)
            .ToListAsync(ct);

        //  3. Fetch new messages from the IMAP server
        IReadOnlyList<ParsedEmail> newEmails;
        try
        {
            newEmails = await
_emailService.FetchNewMessagesAsync(account, processedUids,
ct);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to connect to
mailbox '{Name}'.", account.Name);
            await WriteErrorLogAsync(account, ex.Message,
ct);
            result.ConnectionError = ex.Message;
            return result;
        }

        result.FetchedCount = newEmails.Count;
        _logger.LogInformation("Fetched {Count} new
message(s).", newEmails.Count);

        //  4. Process each message
        foreach (var email in newEmails)
        {
            ct.ThrowIfCancellationRequested();
            await ProcessSingleEmailAsync(db, account, email,
rules, result, ct);
        }

        //  5. Update last-checked timestamp —
        await db.EmailAccounts
            .Where(a => a.Id == account.Id)
            .ExecuteUpdateAsync(s => s.SetProperty(a =>
a.LastCheckedAt, DateTime.UtcNow), ct);

        _logger.LogInformation(
            "Cycle complete — Processed: {P}, Skipped: {S},
Failed: {F}.",
            result.ProcessedCount, result.SkippedCount,
result.FailedCount);

        return result;
    }

    //  Private helpers

    private async Task ProcessSingleEmailAsync(
        AppDbContext db,
        EmailAccount account,
```

```
        ParsedEmail email,
        IReadOnlyList<ProcessingRule> rules,
        ProcessingResult result,
        CancellationToken ct)
    {
        // Persist the message record immediately so we don't
reprocess on crash
        var msgRecord = new EmailMessage
        {
            AccountId   = account.Id,
            ImapUid     = email.ImapUid,
            MessageId   = email.MessageId,
            Sender      = email.Sender,
            Recipients  = email.Recipients,
            Subject     = email.Subject,
            SentAt      = email.SentAt,
            ReceivedAt  = DateTime.UtcNow,
            Status      = MessageStatus.Pending
        };

        db.EmailMessages.Add(msgRecord);
        await db.SaveChangesAsync(ct);

        // Find matching rule
        var matchedRule = _ruleEngine.FindMatchingRule(email,
rules);

        if (matchedRule is null)
        {
            msgRecord.Status = MessageStatus.Skipped;
            await db.SaveChangesAsync(ct);
            result.SkippedCount++;

            await _loggingService.LogAsync(new ProcessingLog
            {
                AccountId      = account.Id,
                MessageId      = msgRecord.Id,
                EmailSubject   = email.Subject,
                Sender         = email.Sender,
                ActionExecuted = "No matching rule",
                Status         = LogStatus.Warning,
                LoggedAt       = DateTime.UtcNow
            }, ct);

            return;
        }

        try
        {
            // Process attachments
            var attachments = await
_attachmentService.ProcessAttachmentsAsync(email, msgRecord,
matchedRule, ct);
```

```
            foreach (var att in attachments)
                db.EmailAttachments.Add(att);

            msgRecord.Status = attachments.Any(a => a.Status
== AttachmentStatus.Failed)
                ? MessageStatus.Failed
                : MessageStatus.Processed;

            await db.SaveChangesAsync(ct);

            // Write success log per attachment (or one
record if no attachments)
            if (attachments.Count == 0)
            {
                await WriteActionLogAsync(account, msgRecord,
matchedRule,
                    null, "Processed (no attachments)",
LogStatus.Success, null, ct);
            }
            else
            {
                foreach (var att in attachments)
                {
                    await WriteActionLogAsync(account,
msgRecord, matchedRule,
                        att.OriginalFileName,
                        att.Status == AttachmentStatus.Failed
? "Failed" : "Saved/Sent",
                        att.Status == AttachmentStatus.Failed
? LogStatus.Failure : LogStatus.Success,
                        att.ErrorMessage, ct);
                }
            }

            // Optionally move to processed folder
            var moveAction = matchedRule.Actions
                .FirstOrDefault(a => a.IsActive &&
a.ActionType == Domain.Enums.ActionType.MoveToFolder);

            if (moveAction?.MoveToFolder is not null)
            {
                try
                {
                    await
_emailService.MoveToFolderAsync(account, email.ImapUid,
moveAction.MoveToFolder, ct);
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, "Could not move
message UID {Uid} to folder '{F}'.",
```

```
                        email.ImapUid,
moveAction.MoveToFolder);
                }
            }

            result.ProcessedCount++;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to process message
'{Subject}'.", email.Subject);

            msgRecord.Status       = MessageStatus.Failed;
            msgRecord.ErrorMessage = ex.Message;
            await db.SaveChangesAsync(ct);

            await WriteActionLogAsync(account, msgRecord,
matchedRule,
                null, "Processing failed", LogStatus.Failure,
ex.Message, ct);

            result.FailedCount++;
        }
    }

    private Task WriteActionLogAsync(
        EmailAccount account, EmailMessage msg,
ProcessingRule rule,
        string? attachmentName, string action, LogStatus
status, string? error,
        CancellationToken ct)
        => _loggingService.LogAsync(new ProcessingLog
        {
            AccountId      = account.Id,
            MessageId      = msg.Id,
            RuleId         = rule.Id,
            EmailSubject   = msg.Subject,
            Sender         = msg.Sender,
            AttachmentName = attachmentName,
            ActionExecuted = action,
            Status         = status,
            ErrorMessage   = error,
            LoggedAt       = DateTime.UtcNow
        }, ct);

    private Task WriteErrorLogAsync(EmailAccount account,
string error, CancellationToken ct)
        => _loggingService.LogAsync(new ProcessingLog
        {
            AccountId      = account.Id,
            EmailSubject   = "Connection error",
            Sender         = string.Empty,
            ActionExecuted = "Connect to mailbox",
```

```
            Status         = LogStatus.Failure,
            ErrorMessage   = error,
            LoggedAt       = DateTime.UtcNow
        }, ct);
}

//  Result summary DTO

public sealed class ProcessingResult
{
    public int     AccountId       { get; set; }
    public int     FetchedCount    { get; set; }
    public int     ProcessedCount  { get; set; }
    public int     SkippedCount    { get; set; }
    public int     FailedCount     { get; set; }
    public string? ConnectionError { get; set; }
    public bool    HasError        => ConnectionError is not
null;
}


namespace EmailProcessor.Domain.Enums;

/// <summary>Overall processing status of an email
message.</summary>
public enum MessageStatus
{
    Pending   = 0,
    Processed = 1,
    Failed    = 2,
    Skipped   = 3
}

/// <summary>Processing status of a single
attachment.</summary>
public enum AttachmentStatus
{
    Pending   = 0,
    Saved     = 1,
    Sent      = 2,
    Failed    = 3,
    Skipped   = 4
}

/// <summary>Result written to the processing log.</summary>
public enum LogStatus
{
    Success = 0,
    Failure = 1,
    Warning = 2
}
```

```
/// <summary>Type of action to execute when a rule
matches.</summary>
public enum ActionType
{
    SaveToFolder          = 0,
    SendAttachmentToApi   = 1,
    SendMetadataToApi     = 2,
    MoveToFolder          = 3
}

/// <summary>HTTP method used for API actions.</summary>
public enum HttpMethod
{
    POST = 0,
    PUT  = 1,
    GET  = 2
}

using EmailProcessor.Domain.Models;

namespace EmailProcessor.Domain.Interfaces;

//  Email retrieval —

/// <summary>Abstraction over the MailKit IMAP
connection.</summary>
public interface IEmailService
{
    /// <summary>
    /// Connects to the account mailbox, fetches all UIDs
that are newer than
    /// the ones already stored in the database, and returns
fully parsed messages.
    /// </summary>
    Task<IReadOnlyList<ParsedEmail>> FetchNewMessagesAsync(
        EmailAccount account, IReadOnlySet<uint>
alreadyProcessedUids,
        CancellationToken ct = default);

    /// <summary>Marks the message read in the remote
mailbox.</summary>
    Task MarkAsReadAsync(EmailAccount account, uint uid,
CancellationToken ct = default);

    /// <summary>Moves the message to the specified remote
folder.</summary>
    Task MoveToFolderAsync(EmailAccount account, uint uid,
string targetFolder,
        CancellationToken ct = default);

    /// <summary>Quick connectivity check — returns true when
credentials are valid.</summary>
```

```
    Task<bool> TestConnectionAsync(EmailAccount account,
CancellationToken ct = default);
}

//  Rule engine —

/// <summary>Evaluates incoming emails against the configured
set of processing rules.</summary>
public interface IRuleEngine
{
    /// <summary>Returns the first active rule that matches
the email, or null if none match.</summary>
    ProcessingRule? FindMatchingRule(ParsedEmail email,
IEnumerable<ProcessingRule> rules);
}

//  Attachment processing —

/// <summary>Downloads and saves/sends attachments from
matched emails.</summary>
public interface IAttachmentService
{
    /// <summary>
    /// Processes all attachments of the parsed email
according to the matched rule and its actions.
    /// Returns attachment metadata records ready to be
persisted.
    /// </summary>
    Task<IReadOnlyList<EmailAttachment>>
ProcessAttachmentsAsync(
        ParsedEmail email, EmailMessage messageRecord,
        ProcessingRule rule, CancellationToken ct = default);
}

//  External API caller —

/// <summary>Sends data to external REST API endpoints as
configured in <see cref="ProcessingAction"/>.</summary>
public interface IApiActionService
{
    Task ExecuteAsync(ProcessingAction action, ParsedEmail
email,
        byte[]? attachmentBytes, string? attachmentFileName,
        CancellationToken ct = default);
}

//  Logging —

/// <summary>Writes structured audit entries to the
ProcessingLogs table.</summary>
public interface ILoggingService
{
```

```
    Task LogAsync(ProcessingLog entry, CancellationToken ct =
default);

    Task<IReadOnlyList<ProcessingLog>> QueryAsync(LogQuery
query, CancellationToken ct = default);
}

//  Crypto

/// <summary>Encrypt/decrypt sensitive strings (passwords)
using Windows DPAPI.</summary>
public interface IEncryptionService
{
    string Encrypt(string plainText);
    string Decrypt(string cipherText);
}

// —
// Data-transfer / value objects
// —

/// <summary>Parsed email message with decoded attachment
bytes, passed between services.</summary>
public sealed class ParsedEmail
{
    public uint     ImapUid    { get; init; }
    public string   MessageId  { get; init; } = string.Empty;
    public string   Sender     { get; init; } = string.Empty;
    public string   Recipients { get; init; } = string.Empty;
    public string   Subject    { get; init; } = string.Empty;
    public string   Body       { get; init; } = string.Empty;
    public DateTime SentAt     { get; init; }
    public IReadOnlyList<ParsedAttachment> Attachments { get;
init; } = [];
}

/// <summary>In-memory representation of a single decoded
attachment.</summary>
public sealed class ParsedAttachment
{
    public string FileName  { get; init; } = string.Empty;
    public string Extension { get; init; } = string.Empty;
    public long   SizeBytes { get; init; }
    public byte[] Content   { get; init; } = [];
}

/// <summary>Filter parameters for log queries from the
UI.</summary>
public sealed class LogQuery
{
    public DateTime? DateFrom  { get; init; }
    public DateTime? DateTo    { get; init; }
    public string?   Sender    { get; init; }
```

```
    public string?   Subject   { get; init; }
    public int?      RuleId    { get; init; }
    public int?      Status    { get; init; }
    public int       PageSize  { get; init; } = 200;
    public int       PageIndex { get; init; } = 0;
}

using EmailProcessor.Domain.Enums;

namespace EmailProcessor.Domain.Models;

// —
// EmailAccount
// —

/// <summary>
/// Represents a configured email mailbox that the
application monitors via IMAP.
/// Passwords are stored encrypted using Windows DPAPI.
/// </summary>
public class EmailAccount
{
    public int    Id                { get; set; }
    public string Name              { get; set; } =
string.Empty;
    public string EmailAddress      { get; set; } =
string.Empty;
    public string ImapServer        { get; set; } =
string.Empty;
    public int    ImapPort          { get; set; } = 993;
    public bool   UseSsl            { get; set; } = true;
    public string Username          { get; set; } =
string.Empty;

    /// <summary>Password encrypted with DPAPI — never stored
as plain text.</summary>
    public string EncryptedPassword { get; set; } =
string.Empty;

    /// <summary>IMAP folder to monitor (e.g.
"INBOX").</summary>
    public string MonitoredFolder  { get; set; } = "INBOX";

    /// <summary>How often to poll the mailbox, in
seconds.</summary>
    public int    PollingIntervalSeconds { get; set; } = 300;

    public bool   IsActive          { get; set; } = true;
    public DateTime CreatedAt       { get; set; } =
DateTime.UtcNow;
    public DateTime? LastCheckedAt { get; set; }

    // Navigation
```

```
    public ICollection<EmailMessage>  Messages { get; set; }
= [];
    public ICollection<ProcessingLog> Logs     { get; set; }
= [];
}

// —
// ProcessingRule
// —

/// <summary>
/// A filter rule that decides which incoming emails are
processed.
/// All non-null conditions must match for the rule to
trigger.
/// </summary>
public class ProcessingRule
{
    public int    Id                          { get; set; }
    public string Name                        { get; set; } =
string.Empty;
    public bool   IsActive                    { get; set; } =
true;

    //  Matching conditions (null = condition not used)
    public string? SenderContains            { get; set; }
    public string? SubjectContains           { get; set; }
    public string? BodyContains              { get; set; }
    public DateTime? DateFrom               { get; set; }
    public DateTime? DateTo                 { get; set; }

    /// <summary>When true, the email must have at least one
attachment to match.</summary>
    public bool   MustHaveAttachments        { get; set; } =
false;

    /// <summary>Comma-separated list of allowed extensions,
e.g. ".pdf,.xlsx".</summary>
    public string? AttachmentExtensions      { get; set; }

    /// <summary>Attachment file name must contain this
substring (case-insensitive).</summary>
    public string? AttachmentNameContains    { get; set; }

    /// <summary>Execution order when multiple rules are
active (lower = first).</summary>
    public int    Priority                   { get; set; } =
100;

    public DateTime CreatedAt               { get; set; } =
DateTime.UtcNow;

    // Navigation
```

```
    public ICollection<ProcessingAction> Actions { get; set;
} = [];
    public ICollection<ProcessingLog>    Logs    { get; set;
} = [];
}

// —
// ProcessingAction
// —

/// <summary>
/// A concrete action executed when its parent <see
cref="ProcessingRule"/> matches.
/// Multiple actions can be attached to a single rule.
/// </summary>
public class ProcessingAction
{
    public int        Id              { get; set; }
    public int        RuleId          { get; set; }
    public string     Name            { get; set; } =
string.Empty;
    public ActionType ActionType      { get; set; }
    public bool       IsActive        { get; set; } = true;

    //  Save-to-folder settings
    /// <summary>Target directory for SaveToFolder
actions.</summary>
    public string?    TargetFolder    { get; set; }

    //  API settings
    public string?    ApiUrl          { get; set; }
    public Enums.HttpMethod HttpMethod { get; set; } =
Enums.HttpMethod.POST;

    /// <summary>JSON-serialised dictionary of additional
HTTP headers.</summary>
    public string?    ApiHeaders      { get; set; }

    /// <summary>Bearer token or API key, stored as plain
text (non-secret).</summary>
    public string?    AuthToken       { get; set; }

    /// <summary>Send the attachment file as multipart/form-
data.</summary>
    public bool       SendAsMultipart { get; set; } = true;

    /// <summary>Include email metadata (sender, subject,
date) in the API payload.</summary>
    public bool       SendMetadataAsJson { get; set; } =
false;

    //  MoveToFolder settings
    public string?    MoveToFolder    { get; set; }
```

```
    public DateTime   CreatedAt       { get; set; } =
DateTime.UtcNow;

    // Navigation
    public ProcessingRule Rule { get; set; } = null!;
}

// —
// EmailMessage
// —

/// <summary>
/// Metadata record for every email retrieved from the
mailbox.
/// Unique message UIDs prevent the same email from being
processed twice.
/// </summary>
public class EmailMessage
{
    public int    Id            { get; set; }
    public int    AccountId     { get; set; }

    /// <summary>IMAP UID — unique within the monitored
folder.</summary>
    public uint   ImapUid       { get; set; }

    /// <summary>Message-ID header value from the email
itself.</summary>
    public string MessageId     { get; set; } = string.Empty;

    public string Sender        { get; set; } = string.Empty;

    /// <summary>Semicolon-separated list of recipient
addresses.</summary>
    public string Recipients    { get; set; } = string.Empty;

    public string Subject       { get; set; } = string.Empty;
    public DateTime SentAt      { get; set; }
    public DateTime ReceivedAt  { get; set; } =
DateTime.UtcNow;
    public MessageStatus Status { get; set; } =
MessageStatus.Pending;
    public string? ErrorMessage { get; set; }

    // Navigation
    public EmailAccount              Account     { get; set;
} = null!;
    public ICollection<EmailAttachment> Attachments { get;
set; } = [];
    public ICollection<ProcessingLog>   Logs        { get;
set; } = [];
}
```

```
// —
// EmailAttachment
// —

/// <summary>
/// Metadata record for each attachment that was downloaded
and processed.
/// </summary>
public class EmailAttachment
{
    public int    Id                 { get; set; }
    public int    MessageId          { get; set; }
    public string OriginalFileName   { get; set; } =
string.Empty;
    public string? SavedFilePath      { get; set; } =
string.Empty;
    public string FileExtension      { get; set; } =
string.Empty;
    public long   FileSizeBytes      { get; set; }
    public AttachmentStatus Status   { get; set; } =
AttachmentStatus.Pending;
    public DateTime ProcessedAt      { get; set; } =
DateTime.UtcNow;
    public string? ErrorMessage      { get; set; }

    // Navigation
    public EmailMessage? Message { get; set; }
}

// —
// ProcessingLog
// —

/// <summary>
/// Audit log entry for every processing action (successful
or failed).
/// </summary>
public class ProcessingLog
{
    public int    Id               { get; set; }
    public int    AccountId        { get; set; }
    public int?   MessageId        { get; set; }
    public int?   RuleId           { get; set; }
    public string EmailSubject     { get; set; } =
string.Empty;
    public string Sender           { get; set; } =
string.Empty;
    public string? AttachmentName  { get; set; }
    public string ActionExecuted   { get; set; } =
string.Empty;
    public LogStatus Status        { get; set; }
    public string? ErrorMessage    { get; set; }
```

```
    public DateTime LoggedAt       { get; set; } =
DateTime.UtcNow;

    // Navigation
    public EmailAccount    Account { get; set; } = null!;
    public EmailMessage?   Message { get; set; }
    public ProcessingRule? Rule    { get; set; }
}

// —
// ApplicationSetting
// —

/// <summary>
/// Simple key-value store for application-wide settings
persisted in the database.
/// </summary>
public class ApplicationSetting
{
    public int    Id    { get; set; }
    public string Key   { get; set; } = string.Empty;
    public string Value { get; set; } = string.Empty;
}

using EmailProcessor.Domain.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace EmailProcessor.Infrastructure.Data;

// —
// DbContext
// —

/// <summary>
/// Entity Framework Core database context.
/// Connection string is injected via DI / appsettings.json.
/// Run <c>dotnet ef migrations add InitialCreate</c> to
generate the first migration.
/// </summary>
public sealed class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext>
options) : base(options) { }

    public DbSet<EmailAccount>      EmailAccounts      =>
Set<EmailAccount>();
    public DbSet<ProcessingRule>    ProcessingRules    =>
Set<ProcessingRule>();
    public DbSet<ProcessingAction>  ProcessingActions  =>
Set<ProcessingAction>();
    public DbSet<EmailMessage>      EmailMessages      =>
Set<EmailMessage>();
```

```
    public DbSet<EmailAttachment>   EmailAttachments   =>
Set<EmailAttachment>();
    public DbSet<ProcessingLog>     ProcessingLogs     =>
Set<ProcessingLog>();
    public DbSet<ApplicationSetting> ApplicationSettings =>
Set<ApplicationSetting>();

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        modelBuilder.ApplyConfiguration(new
EmailAccountConfiguration());
        modelBuilder.ApplyConfiguration(new
ProcessingRuleConfiguration());
        modelBuilder.ApplyConfiguration(new
ProcessingActionConfiguration());
        modelBuilder.ApplyConfiguration(new
EmailMessageConfiguration());
        modelBuilder.ApplyConfiguration(new
EmailAttachmentConfiguration());
        modelBuilder.ApplyConfiguration(new
ProcessingLogConfiguration());
        modelBuilder.ApplyConfiguration(new
ApplicationSettingConfiguration());
    }
}

// —
// Fluent API configurations
// —

file sealed class EmailAccountConfiguration :
IEntityTypeConfiguration<EmailAccount>
{
    public void Configure(EntityTypeBuilder<EmailAccount> b)
    {
        b.ToTable("EmailAccounts");
        b.HasKey(e => e.Id);
        b.Property(e =>
e.Name).HasMaxLength(200).IsRequired();
        b.Property(e =>
e.EmailAddress).HasMaxLength(320).IsRequired();
        b.Property(e =>
e.ImapServer).HasMaxLength(256).IsRequired();
        b.Property(e =>
e.Username).HasMaxLength(320).IsRequired();
        b.Property(e =>
e.EncryptedPassword).HasMaxLength(1024).IsRequired();
        b.Property(e =>
e.MonitoredFolder).HasMaxLength(256).HasDefaultValue("INBOX")
;
        b.HasIndex(e => e.EmailAddress);
    }
```

```
}

file sealed class ProcessingRuleConfiguration :
IEntityTypeConfiguration<ProcessingRule>
{
    public void Configure(EntityTypeBuilder<ProcessingRule>
b)
    {
        b.ToTable("ProcessingRules");
        b.HasKey(r => r.Id);
        b.Property(r =>
r.Name).HasMaxLength(200).IsRequired();
        b.Property(r => r.SenderContains).HasMaxLength(320);
        b.Property(r => r.SubjectContains).HasMaxLength(998);
        b.Property(r =>
r.AttachmentExtensions).HasMaxLength(500);
        b.Property(r =>
r.AttachmentNameContains).HasMaxLength(500);

        b.HasMany(r => r.Actions)
            .WithOne(a => a.Rule)
            .HasForeignKey(a => a.RuleId)
            .OnDelete(DeleteBehavior.Cascade);
    }
}

file sealed class ProcessingActionConfiguration :
IEntityTypeConfiguration<ProcessingAction>
{
    public void Configure(EntityTypeBuilder<ProcessingAction>
b)
    {
        b.ToTable("ProcessingActions");
        b.HasKey(a => a.Id);
        b.Property(a =>
a.Name).HasMaxLength(200).IsRequired();
        b.Property(a => a.TargetFolder).HasMaxLength(1024);
        b.Property(a => a.ApiUrl).HasMaxLength(2048);
        b.Property(a => a.ApiHeaders).HasMaxLength(4096);
        b.Property(a => a.AuthToken).HasMaxLength(2048);
        b.Property(a => a.MoveToFolder).HasMaxLength(256);
        b.Property(a =>
a.ActionType).HasConversion<string>();
        b.Property(a =>
a.HttpMethod).HasConversion<string>();
    }
}

file sealed class EmailMessageConfiguration :
IEntityTypeConfiguration<EmailMessage>
{
    public void Configure(EntityTypeBuilder<EmailMessage> b)
    {
```

```
        b.ToTable("EmailMessages");
        b.HasKey(m => m.Id);
        b.Property(m =>
m.MessageId).HasMaxLength(998).IsRequired();
        b.Property(m =>
m.Sender).HasMaxLength(320).IsRequired();
        b.Property(m => m.Recipients).HasMaxLength(4000);
        b.Property(m => m.Subject).HasMaxLength(998);
        b.Property(m => m.ErrorMessage).HasMaxLength(4000);
        b.Property(m => m.Status).HasConversion<string>();

        // Composite unique index prevents duplicate
processing
        b.HasIndex(m => new { m.AccountId, m.ImapUid
}).IsUnique();

        b.HasOne(m => m.Account)
            .WithMany(a => a.Messages)
            .HasForeignKey(m => m.AccountId)
            .OnDelete(DeleteBehavior.Restrict);

        b.HasMany(m => m.Attachments)
            .WithOne(a => a.Message)
            .HasForeignKey(a => a.MessageId)
            .OnDelete(DeleteBehavior.Cascade);
    }
}

file sealed class EmailAttachmentConfiguration :
IEntityTypeConfiguration<EmailAttachment>
{
    public void Configure(EntityTypeBuilder<EmailAttachment>
b)
    {
        b.ToTable("EmailAttachments");
        b.HasKey(a => a.Id);
        b.Property(a =>
a.OriginalFileName).HasMaxLength(512).IsRequired();
        b.Property(a => a.SavedFilePath).HasMaxLength(1024);
        b.Property(a => a.FileExtension).HasMaxLength(20);
        b.Property(a => a.ErrorMessage).HasMaxLength(4000);
        b.Property(a => a.Status).HasConversion<string>();
    }
}

file sealed class ProcessingLogConfiguration :
IEntityTypeConfiguration<ProcessingLog>
{
    public void Configure(EntityTypeBuilder<ProcessingLog> b)
    {
        b.ToTable("ProcessingLogs");
        b.HasKey(l => l.Id);
        b.Property(l => l.EmailSubject).HasMaxLength(998);
```

```
        b.Property(l => l.Sender).HasMaxLength(320);
        b.Property(l => l.AttachmentName).HasMaxLength(512);
        b.Property(l => l.ActionExecuted).HasMaxLength(500);
        b.Property(l => l.ErrorMessage).HasMaxLength(4000);
        b.Property(l => l.Status).HasConversion<string>();

        b.HasIndex(l => l.LoggedAt);
        b.HasIndex(l => l.Status);

        b.HasOne(l => l.Account)
            .WithMany(a => a.Logs)
            .HasForeignKey(l => l.AccountId)
            .OnDelete(DeleteBehavior.Restrict);

        b.HasOne(l => l.Message)
            .WithMany(m => m.Logs)
            .HasForeignKey(l => l.MessageId)
            .OnDelete(DeleteBehavior.SetNull);

        b.HasOne(l => l.Rule)
            .WithMany(r => r.Logs)
            .HasForeignKey(l => l.RuleId)
            .OnDelete(DeleteBehavior.SetNull);
    }
}

file sealed class ApplicationSettingConfiguration :
IEntityTypeConfiguration<ApplicationSetting>
{
    public void
Configure(EntityTypeBuilder<ApplicationSetting> b)
    {
        b.ToTable("ApplicationSettings");
        b.HasKey(s => s.Id);
        b.Property(s =>
s.Key).HasMaxLength(200).IsRequired();
        b.Property(s => s.Value).HasMaxLength(4000);
        b.HasIndex(s => s.Key).IsUnique();

        // Seed default settings
        b.HasData(
            new ApplicationSetting { Id = 1, Key =
"MaxAttachmentSizeMb",   Value = "25"    },
            new ApplicationSetting { Id = 2, Key =
"AllowedExtensions",      Value =
".pdf,.xlsx,.csv,.docx,.xml,.zip" },
            new ApplicationSetting { Id = 3, Key =
"BlockedExtensions",      Value =
".exe,.bat,.cmd,.vbs,.ps1,.sh"    },
            new ApplicationSetting { Id = 4, Key =
"DefaultPollingInterval", Value = "300"   },
            new ApplicationSetting { Id = 5, Key =
"MarkMessagesAsRead",     Value = "true"  }
```

```
        );
    }
}

using System.Security.Cryptography;
using System.Text;
using EmailProcessor.Domain.Interfaces;

namespace EmailProcessor.Infrastructure.Security;

/// <summary>
/// Encrypts and decrypts strings using the Windows Data
Protection API (DPAPI).
/// Data is protected per-machine (CurrentUser scope), so it
can only be decrypted
/// on the same machine/user account that encrypted it.
/// Falls back to Base64 on non-Windows platforms
(development convenience only —
/// remove or replace for production cross-platform
deployments).
/// </summary>
public sealed class DpapiEncryptionService :
IEncryptionService
{
    private static readonly bool IsWindows =
        OperatingSystem.IsWindows();

    public string Encrypt(string plainText)
    {
        if (string.IsNullOrEmpty(plainText)) return
string.Empty;

        if (IsWindows)
        {
            var bytes     =
Encoding.UTF8.GetBytes(plainText);
            var encrypted = ProtectedData.Protect(bytes,
null, DataProtectionScope.CurrentUser);
            return Convert.ToBase64String(encrypted);
        }

        // Non-Windows fallback — NOT for production use
        return
Convert.ToBase64String(Encoding.UTF8.GetBytes(plainText));
    }

    public string Decrypt(string cipherText)
    {
        if (string.IsNullOrEmpty(cipherText)) return
string.Empty;

        try
        {
```

```
            var bytes = Convert.FromBase64String(cipherText);

            if (IsWindows)
            {
                var decrypted =
ProtectedData.Unprotect(bytes, null,
DataProtectionScope.CurrentUser);
                return Encoding.UTF8.GetString(decrypted);
            }

            return Encoding.UTF8.GetString(bytes);
        }
        catch (CryptographicException ex)
        {
            throw new InvalidOperationException(
                "Failed to decrypt password. The data may
have been encrypted on a different machine or user account.",
ex);
        }
    }
}

using System.Net.Http.Headers;
using System.Text;
using System.Text.Json;
using EmailProcessor.Domain.Interfaces;
using EmailProcessor.Domain.Models;
using Microsoft.Extensions.Logging;

namespace EmailProcessor.Infrastructure.Services;

/// <summary>
/// Executes HTTP calls to external REST APIs as configured
in a <see cref="ProcessingAction"/>.
/// Supports:
///   • Sending an attachment as multipart/form-data (binary
upload)
///   • Sending email metadata as a JSON body
///   • Bearer token / API-key authentication headers
///   • Arbitrary extra headers stored as a JSON dictionary
in the action config
/// </summary>
public sealed class ApiActionService : IApiActionService
{
    private readonly HttpClient _http;
    private readonly ILogger<ApiActionService> _logger;

    public ApiActionService(HttpClient http,
ILogger<ApiActionService> logger)
    {
        _http   = http;
        _logger = logger;
    }
```

```
    public async Task ExecuteAsync(
        ProcessingAction action,
        ParsedEmail email,
        byte[]? attachmentBytes,
        string? attachmentFileName,
        CancellationToken ct = default)
    {
        if (string.IsNullOrWhiteSpace(action.ApiUrl))
            throw new InvalidOperationException($"Action
'{action.Name}' has no API URL configured.");

        using var request = BuildRequest(action, email,
attachmentBytes, attachmentFileName);

        _logger.LogInformation("Calling API {Method} {Url}",
request.Method, action.ApiUrl);

        var response = await _http.SendAsync(request, ct);

        if (!response.IsSuccessStatusCode)
        {
            var body = await
response.Content.ReadAsStringAsync(ct);
            throw new HttpRequestException(
                $"API returned {(int)response.StatusCode}
{response.ReasonPhrase}: {body}");
        }

        _logger.LogInformation("API call succeeded
({Status}).", response.StatusCode);
    }

    //  Request construction —

    private static HttpRequestMessage BuildRequest(
        ProcessingAction action,
        ParsedEmail email,
        byte[]? attachmentBytes,
        string? attachmentFileName)
    {
        var method  = action.HttpMethod switch
        {
            Domain.Enums.HttpMethod.POST =>
System.Net.Http.HttpMethod.Post,
            Domain.Enums.HttpMethod.PUT  =>
System.Net.Http.HttpMethod.Put,
            Domain.Enums.HttpMethod.GET  =>
System.Net.Http.HttpMethod.Get,
            _                            =>
System.Net.Http.HttpMethod.Post
        };
```

```
        var request = new HttpRequestMessage(method,
action.ApiUrl);

        // Bearer / API-key authentication
        if (!string.IsNullOrWhiteSpace(action.AuthToken))
            request.Headers.Authorization =
                new AuthenticationHeaderValue("Bearer",
action.AuthToken);

        // Extra custom headers stored as {"Header-Name":
"value"} JSON
        if (!string.IsNullOrWhiteSpace(action.ApiHeaders))
        {
            var headers =
JsonSerializer.Deserialize<Dictionary<string,
string>>(action.ApiHeaders);
            if (headers is not null)
                foreach (var (name, value) in headers)

request.Headers.TryAddWithoutValidation(name, value);
        }

        // Build content
        request.Content = action.SendAsMultipart &&
attachmentBytes is not null
            ? BuildMultipartContent(email, attachmentBytes,
attachmentFileName ?? "attachment",
action.SendMetadataAsJson)
            : BuildJsonContent(email);

        return request;
    }

    private static MultipartFormDataContent
BuildMultipartContent(
        ParsedEmail email, byte[] bytes, string fileName,
bool includeMetadata)
    {
        var content = new MultipartFormDataContent();

        var fileContent = new ByteArrayContent(bytes);
        fileContent.Headers.ContentType = new
MediaTypeHeaderValue("application/octet-stream");
        content.Add(fileContent, "file", fileName);

        if (includeMetadata)
        {
            var meta = BuildMetadataPayload(email);
            content.Add(new StringContent(meta,
Encoding.UTF8, "application/json"), "metadata");
        }

        return content;
```

```
    }

    private static StringContent BuildJsonContent(ParsedEmail
email)
    {
        var json = BuildMetadataPayload(email);
        return new StringContent(json, Encoding.UTF8,
"application/json");
    }

    private static string BuildMetadataPayload(ParsedEmail
email)
    {
        var payload = new
        {
            messageId  = email.MessageId,
            sender     = email.Sender,
            recipients = email.Recipients,
            subject    = email.Subject,
            sentAt     = email.SentAt.ToString("O"),
            attachments = email.Attachments.Select(a => new
            {
                fileName  = a.FileName,
                extension = a.Extension,
                sizeBytes = a.SizeBytes
            })
        };

        return JsonSerializer.Serialize(payload, new
JsonSerializerOptions
        {
            WriteIndented = false,
            PropertyNamingPolicy = JsonNamingPolicy.CamelCase
        });
    }
}

using EmailProcessor.Domain.Enums;
using EmailProcessor.Domain.Interfaces;
using EmailProcessor.Domain.Models;
using Microsoft.Extensions.Logging;

namespace EmailProcessor.Infrastructure.Services;

/// <summary>
/// Iterates over the parsed attachments of a matched email
and executes each
/// configured <see cref="ProcessingAction"/> (save to
folder, send to API, etc.).
/// Returns a list of <see cref="EmailAttachment"/> metadata
records for persistence.
/// </summary>
public sealed class AttachmentService : IAttachmentService
```

```
{
    private readonly IApiActionService _apiService;
    private readonly ILogger<AttachmentService> _logger;

    // Extensions that must never be saved — defence-in-depth
on top of UI validation
    private static readonly HashSet<string> BlockedExtensions
=
        [".exe", ".bat", ".cmd", ".vbs", ".ps1", ".sh",
".msi", ".scr", ".pif"];

    public AttachmentService(IApiActionService apiService,
ILogger<AttachmentService> logger)
    {
        _apiService = apiService;
        _logger     = logger;
    }

    public async Task<IReadOnlyList<EmailAttachment>>
ProcessAttachmentsAsync(
        ParsedEmail email,
        EmailMessage messageRecord,
        ProcessingRule rule,
        CancellationToken ct = default)
    {
        var results  = new List<EmailAttachment>();
        var actions  = rule.Actions.Where(a =>
a.IsActive).ToList();

        foreach (var attachment in email.Attachments)
        {
            ct.ThrowIfCancellationRequested();

            var record = new EmailAttachment
            {
                MessageId        = messageRecord.Id,
                OriginalFileName = attachment.FileName,
                FileExtension    = attachment.Extension,
                FileSizeBytes    = attachment.SizeBytes,
                ProcessedAt      = DateTime.UtcNow
            };

            // Block dangerous extensions unconditionally
            if
(BlockedExtensions.Contains(attachment.Extension))
            {
                _logger.LogWarning("Blocked dangerous
attachment '{File}'.", attachment.FileName);
                record.Status       =
AttachmentStatus.Skipped;
                record.ErrorMessage = $"Blocked extension:
{attachment.Extension}";
                results.Add(record);
```

```
                continue;
            }

            try
            {
                foreach (var action in actions)
                {
                    ct.ThrowIfCancellationRequested();
                    await ExecuteActionAsync(action, email,
attachment, record, ct);
                }

                record.Status = AttachmentStatus.Saved;
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Failed to process
attachment '{File}'.", attachment.FileName);
                record.Status       =
AttachmentStatus.Failed;
                record.ErrorMessage = ex.Message;
            }

            results.Add(record);
        }

        return results;
    }

    //  Action dispatching —

    private async Task ExecuteActionAsync(
        ProcessingAction action,
        ParsedEmail email,
        ParsedAttachment attachment,
        EmailAttachment record,
        CancellationToken ct)
    {
        switch (action.ActionType)
        {
            case Domain.Enums.ActionType.SaveToFolder:
                SaveToFolder(action, attachment, record);
                break;

            case Domain.Enums.ActionType.SendAttachmentToApi:
                await _apiService.ExecuteAsync(action, email,
attachment.Content, attachment.FileName, ct);
                record.Status = AttachmentStatus.Sent;
                break;

            case Domain.Enums.ActionType.SendMetadataToApi:
                await _apiService.ExecuteAsync(action, email,
null, null, ct);
```

```
                break;

            case Domain.Enums.ActionType.MoveToFolder:
                // Folder move is handled at the message
level by the orchestrator service.
                // Nothing to do here for individual
attachments.
                break;

            default:
                _logger.LogWarning("Unknown action type:
{Type}", action.ActionType);
                break;
        }
    }

    //  Save-to-folder implementation

    private void SaveToFolder(
        ProcessingAction action,
        ParsedAttachment attachment,
        EmailAttachment record)
    {
        if (string.IsNullOrWhiteSpace(action.TargetFolder))
            throw new InvalidOperationException($"Action
'{action.Name}' has no TargetFolder configured.");

        Directory.CreateDirectory(action.TargetFolder);

        var safeName    =
BuildUniqueFileName(action.TargetFolder,
attachment.FileName);
        var destination = Path.Combine(action.TargetFolder,
safeName);

        File.WriteAllBytes(destination, attachment.Content);

        record.SavedFilePath = destination;
        _logger.LogInformation("Saved attachment '{File}' →
'{Dest}'.", attachment.FileName, destination);
    }

    /// <summary>
    /// Returns a file name that does not already exist in
<paramref name="folder"/>.
    /// If "report.pdf" exists, tries "report_1.pdf",
"report_2.pdf", and so on.
    /// </summary>
    private static string BuildUniqueFileName(string folder,
string fileName)
    {
        var nameWithoutExt =
Path.GetFileNameWithoutExtension(fileName);
```

```
        var ext            = Path.GetExtension(fileName);
        var candidate      = fileName;
        var counter        = 1;

        while (File.Exists(Path.Combine(folder, candidate)))
        {
            candidate = $"{nameWithoutExt}_{counter++}{ext}";
        }

        return candidate;
    }
}

using EmailProcessor.Domain.Interfaces;
using EmailProcessor.Domain.Models;
using EmailProcessor.Infrastructure.Data;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;

namespace EmailProcessor.Infrastructure.Services;

/// <summary>
/// Persists audit log entries to the <c>ProcessingLogs</c>
table
/// and supports filtered queries for the UI log viewer.
/// </summary>
public sealed class LoggingService : ILoggingService
{
    private readonly IDbContextFactory<AppDbContext>
_dbFactory;
    private readonly ILogger<LoggingService> _logger;

    public LoggingService(IDbContextFactory<AppDbContext>
dbFactory, ILogger<LoggingService> logger)
    {
        _dbFactory = dbFactory;
        _logger    = logger;
    }

    public async Task LogAsync(ProcessingLog entry,
CancellationToken ct = default)
    {
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync(ct);
            db.ProcessingLogs.Add(entry);
            await db.SaveChangesAsync(ct);
        }
        catch (Exception ex)
        {
            // Logging must never crash the caller
```

```
            _logger.LogError(ex, "Failed to persist
processing log entry.");
        }
    }

    public async Task<IReadOnlyList<ProcessingLog>>
QueryAsync(LogQuery query, CancellationToken ct = default)
    {
        await using var db = await
_dbFactory.CreateDbContextAsync(ct);

        var q = db.ProcessingLogs
            .Include(l => l.Account)
            .Include(l => l.Rule)
            .AsNoTracking()
            .AsQueryable();

        if (query.DateFrom.HasValue)
            q = q.Where(l => l.LoggedAt >=
query.DateFrom.Value);

        if (query.DateTo.HasValue)
            q = q.Where(l => l.LoggedAt <=
query.DateTo.Value);

        if (!string.IsNullOrWhiteSpace(query.Sender))
            q = q.Where(l =>
l.Sender.Contains(query.Sender));

        if (!string.IsNullOrWhiteSpace(query.Subject))
            q = q.Where(l =>
l.EmailSubject.Contains(query.Subject));

        if (query.RuleId.HasValue)
            q = q.Where(l => l.RuleId == query.RuleId.Value);

        if (query.Status.HasValue)
            q = q.Where(l => (int)l.Status ==
query.Status.Value);

        return await q
            .OrderByDescending(l => l.LoggedAt)
            .Skip(query.PageIndex * query.PageSize)
            .Take(query.PageSize)
            .ToListAsync(ct);
    }
}

using EmailProcessor.Domain.Interfaces;
using EmailProcessor.Domain.Models;
using MailKit;
using MailKit.Net.Imap;
using MailKit.Search;
```

```
using MailKit.Security;
using MimeKit;
using Microsoft.Extensions.Logging;

namespace EmailProcessor.Infrastructure.Services;

/// <summary>
/// Connects to an IMAP mailbox via MailKit, retrieves new
messages, and
/// decodes their attachments into <see cref="ParsedEmail"/>
value objects.
/// </summary>
public sealed class MailKitEmailService : IEmailService
{
    private readonly IEncryptionService _encryption;
    private readonly ILogger<MailKitEmailService> _logger;

    public MailKitEmailService(IEncryptionService encryption,
ILogger<MailKitEmailService> logger)
    {
        _encryption = encryption;
        _logger     = logger;
    }

    //  Public API

    public async Task<IReadOnlyList<ParsedEmail>>
FetchNewMessagesAsync(EmailAccount account,
IReadOnlySet<uint> alreadyProcessedUids, CancellationToken ct
= default)
    {
        using var client = await ConnectAsync(account, ct);
        var inbox = await OpenFolderAsync(client,
account.MonitoredFolder, false, ct);

        var allUids = await
inbox.SearchAsync(SearchQuery.All, ct);

        var newUids = allUids
            .Where(uid =>
!alreadyProcessedUids.Contains(uid.Id))
            .ToList();

        _logger.LogInformation("Account '{Name}': {Total}
messages in folder, {New} are new.",
            account.Name, allUids.Count, newUids.Count);

        var results = new List<ParsedEmail>(newUids.Count);

        foreach (var uid in newUids)
        {
            ct.ThrowIfCancellationRequested();
            try
```

```
            {
                var mime = await inbox.GetMessageAsync(uid,
ct);
                results.Add(ParseMimeMessage(uid.Id, mime));
            }
            catch (Exception ex)
            {
                _logger.LogWarning(ex, "Failed to parse
message UID {Uid}.", uid);
            }
        }

        await client.DisconnectAsync(true, ct);
        return results;
    }

    public async Task MarkAsReadAsync(EmailAccount account,
uint uid, CancellationToken ct = default)
    {
        using var client = await ConnectAsync(account, ct);
        var folder = await OpenFolderAsync(client,
account.MonitoredFolder, writable: true, ct);
        await folder.AddFlagsAsync(new UniqueId(uid),
MessageFlags.Seen, true, ct);
        await client.DisconnectAsync(true, ct);
    }

    public async Task MoveToFolderAsync(
        EmailAccount account, uint uid, string targetFolder,
CancellationToken ct = default)
    {
        using var client = await ConnectAsync(account, ct);
        var source = await OpenFolderAsync(client,
account.MonitoredFolder, writable: true, ct);

        var dest = await client.GetFolderAsync(targetFolder,
ct);
        await source.MoveToAsync(new UniqueId(uid), dest,
ct);
        await client.DisconnectAsync(true, ct);
    }

    public async Task<bool> TestConnectionAsync(EmailAccount
account, CancellationToken ct = default)
    {
        try
        {
            using var client = await ConnectAsync(account,
ct);
            var ok = client.IsAuthenticated;
            await client.DisconnectAsync(true, ct);
            return ok;
        }
```

```
        catch (Exception ex)
        {
            _logger.LogWarning(ex, "Connection test failed
for account '{Name}'.", account.Name);
            return false;
        }
    }

    //  Helpers —

    private async Task<ImapClient> ConnectAsync(EmailAccount
account, CancellationToken ct)
    {
        var client   = new ImapClient();
        var sslMode  = account.UseSsl ?
SecureSocketOptions.SslOnConnect :
SecureSocketOptions.StartTlsWhenAvailable;
        var password =
_encryption.Decrypt(account.EncryptedPassword);

        await client.ConnectAsync(account.ImapServer,
account.ImapPort, sslMode, ct);
        await client.AuthenticateAsync(account.Username,
password, ct);
        return client;
    }

    private static async Task<IMailFolder> OpenFolderAsync(
        ImapClient client, string folderName, bool writable,
CancellationToken ct)
    {
        var folder = await client.GetFolderAsync(folderName,
ct);
        var access = writable ? FolderAccess.ReadWrite :
FolderAccess.ReadOnly;
        await folder.OpenAsync(access, ct);
        return folder;
    }

    /// <summary>Transforms a <see cref="MimeMessage"/> into
a domain value object.</summary>
    private static ParsedEmail ParseMimeMessage(uint uid,
MimeMessage mime)
    {
        var attachments = new List<ParsedAttachment>();

        foreach (var part in
mime.BodyParts.OfType<MimePart>())
        {
            // Only treat parts with a ContentDisposition of
"attachment" as downloadable
            if (part.ContentDisposition?.Disposition !=
ContentDisposition.Attachment
```

```
                && string.IsNullOrEmpty(part.FileName))
            {
                continue;
            }

            var fileName = SanitizeFileName(part.FileName ??
$"attachment_{Guid.NewGuid()}");
            var ext      =
Path.GetExtension(fileName).ToLowerInvariant();

            using var ms = new MemoryStream();
            part.Content.DecodeTo(ms);
            var bytes = ms.ToArray();

            attachments.Add(new ParsedAttachment
            {
                FileName  = fileName,
                Extension = ext,
                SizeBytes = bytes.Length,
                Content   = bytes
            });
        }

        return new ParsedEmail
        {
            ImapUid    = uid,
            MessageId  = mime.MessageId ??
Guid.NewGuid().ToString(),
            Sender     =
mime.From.Mailboxes.FirstOrDefault()?.Address ?? "unknown",
            Recipients = string.Join(";",
mime.To.Mailboxes.Select(m => m.Address)),
            Subject    = mime.Subject ?? "(no subject)",
            Body       = mime.TextBody ?? mime.HtmlBody ??
string.Empty,
            SentAt     = mime.Date.UtcDateTime,
            Attachments = attachments
        };
    }

    /// <summary>
    /// Strips directory traversal characters and other
dangerous path components
    /// from an attachment file name received over the
network.
    /// </summary>
    public static string SanitizeFileName(string rawName)
    {
        // Remove any directory component
        var name = Path.GetFileName(rawName);

        // Replace characters that are invalid on Windows
file systems
```

```
        foreach (var ch in Path.GetInvalidFileNameChars())
            name = name.Replace(ch, '_');

        // Guard against leading dots (hidden files on Linux)
and overly long names
        name = name.TrimStart('.').Trim();
        if (name.Length > 200)
            name = name[..196] + Path.GetExtension(name);

        return string.IsNullOrWhiteSpace(name) ?
$"attachment_{Guid.NewGuid()}" : name;
    }
}

using EmailProcessor.Domain.Interfaces;
using EmailProcessor.Domain.Models;
using Microsoft.Extensions.Logging;

namespace EmailProcessor.Infrastructure.Services;

/// <summary>
/// Evaluates a parsed email against an ordered list of
active processing rules.
/// Returns the first rule whose every non-null condition
matches the email.
/// </summary>
public sealed class RuleEngineService : IRuleEngine
{
    private readonly ILogger<RuleEngineService> _logger;

    public RuleEngineService(ILogger<RuleEngineService>
logger) => _logger = logger;

    public ProcessingRule? FindMatchingRule(ParsedEmail
email, IEnumerable<ProcessingRule> rules)
    {
        foreach (var rule in rules.Where(r =>
r.IsActive).OrderBy(r => r.Priority))
        {
            if (Matches(email, rule))
            {
                _logger.LogDebug("Rule '{Rule}' matched
message '{Subject}'.", rule.Name, email.Subject);
                return rule;
            }
        }

        _logger.LogDebug("No rule matched message
'{Subject}'.", email.Subject);
        return null;
    }

    //  Condition evaluation —
```

```
    private static bool Matches(ParsedEmail email,
ProcessingRule rule)
    {
        // Sender condition
        if (!string.IsNullOrEmpty(rule.SenderContains) &&
            !email.Sender.Contains(rule.SenderContains,
StringComparison.OrdinalIgnoreCase))
            return false;

        // Subject condition
        if (!string.IsNullOrEmpty(rule.SubjectContains) &&
            !email.Subject.Contains(rule.SubjectContains,
StringComparison.OrdinalIgnoreCase))
            return false;

        // Body condition
        if (!string.IsNullOrEmpty(rule.BodyContains) &&
            !email.Body.Contains(rule.BodyContains,
StringComparison.OrdinalIgnoreCase))
            return false;

        // Date range conditions
        if (rule.DateFrom.HasValue && email.SentAt <
rule.DateFrom.Value)
            return false;

        if (rule.DateTo.HasValue && email.SentAt >
rule.DateTo.Value)
            return false;

        // Attachment presence condition
        if (rule.MustHaveAttachments &&
email.Attachments.Count == 0)
            return false;

        // Attachment extension condition — at least one
attachment must match
        if (!string.IsNullOrEmpty(rule.AttachmentExtensions))
        {
            var allowed = rule.AttachmentExtensions
                .Split(',',
StringSplitOptions.RemoveEmptyEntries |
StringSplitOptions.TrimEntries)
                .Select(e => e.ToLowerInvariant())
                .ToHashSet();

            if (!email.Attachments.Any(a =>
allowed.Contains(a.Extension)))
                return false;
        }
```

```
        // Attachment name condition — at least one
attachment must match
        if
(!string.IsNullOrEmpty(rule.AttachmentNameContains))
        {
            if (!email.Attachments.Any(a =>

a.FileName.Contains(rule.AttachmentNameContains,
StringComparison.OrdinalIgnoreCase)))
                return false;
        }

        return true;
    }
}

using System.Globalization;
using System.Windows;
using System.Windows.Data;

namespace EmailProcessor.WPF.Converters;

/// <summary>Returns Visible when the value is non-null,
Collapsed when null.</summary>
[ValueConversion(typeof(object), typeof(Visibility))]
public sealed class NullToVisibilityConverter :
IValueConverter
{
    public object Convert(object? value, Type targetType,
object? parameter, CultureInfo culture)
        => value is null ? Visibility.Collapsed :
Visibility.Visible;

    public object ConvertBack(object? value, Type targetType,
object? parameter, CultureInfo culture)
        => throw new NotSupportedException();
}

/// <summary>Returns Visible when the bool is true, Collapsed
when false.</summary>
[ValueConversion(typeof(bool), typeof(Visibility))]
public sealed class BoolToVisibilityConverter :
IValueConverter
{
    public object Convert(object? value, Type targetType,
object? parameter, CultureInfo culture)
        => value is true ? Visibility.Visible :
Visibility.Collapsed;

    public object ConvertBack(object? value, Type targetType,
object? parameter, CultureInfo culture)
        => (value is Visibility v) && v ==
Visibility.Visible;
```

```
}

/// <summary>Converts a file size in bytes to a human-
readable KB/MB string.</summary>
[ValueConversion(typeof(long), typeof(string))]
public sealed class FileSizeConverter : IValueConverter
{
    public object Convert(object? value, Type targetType,
object? parameter, CultureInfo culture)
    {
        if (value is not long bytes) return string.Empty;
        if (bytes < 1024)        return $"{bytes} B";
        if (bytes < 1024 * 1024) return $"{bytes / 1024.0:F1}
KB";
        return $"{bytes / (1024.0 * 1024):F2} MB";
    }

    public object ConvertBack(object? value, Type targetType,
object? parameter, CultureInfo culture)
        => throw new NotSupportedException();
}

using System.Globalization;
using System.Windows;
using System.Windows.Data;
using System.Windows.Media;
using EmailProcessor.Domain.Enums;

namespace EmailProcessor.WPF.Converters;

/// <summary>
/// Maps a LogStatus enum value (or a bool for account
IsActive) to a
/// solid background brush used in the status badge cells.
/// </summary>
[ValueConversion(typeof(object), typeof(Brush))]
public sealed class StatusToBrushConverter : IValueConverter
{
    private static readonly SolidColorBrush Green  =
new(Color.FromRgb(22,  163, 74));
    private static readonly SolidColorBrush Red    =
new(Color.FromRgb(220, 38,  38));
    private static readonly SolidColorBrush Amber  =
new(Color.FromRgb(217, 119, 6));
    private static readonly SolidColorBrush Gray   =
new(Color.FromRgb(107, 114, 128));
    private static readonly SolidColorBrush Blue   =
new(Color.FromRgb(29,  78,  216));

    public object Convert(object? value, Type targetType,
object? parameter, CultureInfo culture)
    {
        // bool → account active/inactive badge
```

```
        if (value is bool b) return b ? Green : Gray;

        // LogStatus → log row badge
        if (value is LogStatus ls)
        {
            return ls switch
            {
                LogStatus.Success => Green,
                LogStatus.Failure => Red,
                LogStatus.Warning => Amber,
                _                 => Gray
            };
        }

        // MessageStatus
        if (value is MessageStatus ms)
        {
            return ms switch
            {
                MessageStatus.Processed => Green,
                MessageStatus.Failed    => Red,
                MessageStatus.Skipped   => Gray,
                MessageStatus.Pending   => Blue,
                _                       => Gray
            };
        }

        // AttachmentStatus
        if (value is AttachmentStatus att)
        {
            return att switch
            {
                AttachmentStatus.Saved    => Green,
                AttachmentStatus.Sent     => Blue,
                AttachmentStatus.Failed   => Red,
                AttachmentStatus.Skipped  => Gray,
                AttachmentStatus.Pending  => Amber,
                _                         => Gray
            };
        }

        return Gray;
    }

    public object ConvertBack(object? value, Type targetType,
object? parameter, CultureInfo culture)
        => throw new NotSupportedException();
}

/// <summary>
/// Scales an integer message count to a bar height in pixels
(0–110 px).
```

```
/// The max value is capped at 110; counts above the cap
still render at full height.
/// Minimum visible height is 4 px so a zero-count day shows
a tiny stub.
/// </summary>
[ValueConversion(typeof(int), typeof(double))]
public sealed class CountToHeightConverter : IValueConverter
{
    private const double MaxHeight = 110;
    private const double MinHeight = 4;
    private const int    ScaleCap  = 20;   // counts ≥ 20 →
full bar

    public object Convert(object? value, Type targetType,
object? parameter, CultureInfo culture)
    {
        if (value is not int count || count <= 0) return
MinHeight;
        var ratio  = Math.Min(count / (double)ScaleCap, 1.0);
        return Math.Max(MinHeight, ratio * MaxHeight);
    }

    public object ConvertBack(object? value, Type targetType,
object? parameter, CultureInfo culture)
        => throw new NotSupportedException();
}

using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Input;

namespace EmailProcessor.WPF.Commands
{
    // ─
    // RelayCommand  (sync)
    // ─

    /// <summary>
    /// A lightweight ICommand implementation that delegates
Execute and CanExecute
    /// to lambda callbacks. Supports automatic CanExecute
refresh via
    /// <see cref="RaiseCanExecuteChanged"/>.
    /// </summary>
    public sealed class RelayCommand : ICommand
    {
        private readonly Action<object?> _execute;
        private readonly Func<object?, bool>? _canExecute;

        public RelayCommand(Action<object?> execute,
Func<object?, bool>? canExecute = null)
        {
            _execute = execute;
```

```
            _canExecute = canExecute;
        }

        public RelayCommand(Action execute, Func<bool>?
canExecute = null)
            : this(_ => execute(), canExecute is null ? null
: _ => canExecute()) { }

        public event EventHandler? CanExecuteChanged;

        public bool CanExecute(object? parameter) =>
_canExecute?.Invoke(parameter) ?? true;
        public void Execute(object? parameter) =>
_execute(parameter);
        public void RaiseCanExecuteChanged() =>
CanExecuteChanged?.Invoke(this, EventArgs.Empty);
    }

    // —
    // AsyncRelayCommand
    // —

    /// <summary>
    /// An ICommand that wraps an async Task, disables itself
while executing,
    /// and surfaces exceptions via the <see
cref="Exception"/> property.
    /// </summary>
    public sealed class AsyncRelayCommand : ICommand
    {
        private readonly Func<object?, Task> _execute;
        private readonly Func<object?, bool>? _canExecute;
        private bool _isExecuting;

        public AsyncRelayCommand(Func<Task> execute,
Func<bool>? canExecute = null)
            : this(_ => execute(), canExecute is null ? null
: _ => canExecute()) { }

        public AsyncRelayCommand(Func<object?, Task> execute,
Func<object?, bool>? canExecute = null)
        {
            _execute = execute;
            _canExecute = canExecute;
        }

        public Exception? Exception { get; private set; }

        public event EventHandler? CanExecuteChanged;

        public bool CanExecute(object? parameter) =>
            !_isExecuting && (_canExecute?.Invoke(parameter)
?? true);
```

```
        public async void Execute(object? parameter)
        {
            _isExecuting = true;
            Exception = null;
            RaiseCanExecuteChanged();
            try
            {
                await _execute(parameter);
            }
            catch (Exception ex)
            {
                Exception = ex;
            }
            finally
            {
                _isExecuting = false;
                RaiseCanExecuteChanged();
            }
        }

        public void RaiseCanExecuteChanged() =>
            CanExecuteChanged?.Invoke(this, EventArgs.Empty);

        /// <summary>Awaitable version for use in
initialisation code paths.</summary>
        public Task ExecuteAsync(object? parameter = null) =>
_execute(parameter);
    }
}

using System.Collections.ObjectModel;
using EmailProcessor.Domain.Enums;
using EmailProcessor.Domain.Models;
using EmailProcessor.Infrastructure.Data;
using EmailProcessor.WPF.Commands;
using Microsoft.EntityFrameworkCore;

namespace EmailProcessor.WPF.ViewModels;

/// <summary>
/// ViewModel for the Dashboard page.
/// Loads summary statistics, per-account status, and the
last 10 log entries.
/// Refreshes on demand and auto-refreshes when the
background service fires StateChanged.
/// </summary>
public sealed class DashboardViewModel : ViewModelBase
{
    private readonly IDbContextFactory<AppDbContext>
_dbFactory;

    //  Summary counters
```

```
    private int _totalMessages;
    private int _processedMessages;
    private int _failedMessages;
    private int _skippedMessages;
    private int _totalAttachments;
    private int _savedAttachments;
    private int _sentAttachments;
    private int _failedAttachments;
    private int _totalLogs;
    private int _failedLogs;
    private int _activeAccounts;
    private int _totalRules;
    private string _lastChecked = "Never";

    public int    TotalMessages      { get => _totalMessages;
set => SetField(ref _totalMessages,      value); }
    public int    ProcessedMessages  { get =>
_processedMessages;  set => SetField(ref _processedMessages,
value); }
    public int    FailedMessages     { get =>
_failedMessages;     set => SetField(ref _failedMessages,
value); }
    public int    SkippedMessages    { get =>
_skippedMessages;    set => SetField(ref _skippedMessages,
value); }
    public int    TotalAttachments   { get =>
_totalAttachments;   set => SetField(ref _totalAttachments,
value); }
    public int    SavedAttachments   { get =>
_savedAttachments;   set => SetField(ref _savedAttachments,
value); }
    public int    SentAttachments    { get =>
_sentAttachments;    set => SetField(ref _sentAttachments,
value); }
    public int    FailedAttachments  { get =>
_failedAttachments;  set => SetField(ref _failedAttachments,
value); }
    public int    TotalLogs          { get => _totalLogs;
set => SetField(ref _totalLogs,          value); }
    public int    FailedLogs         { get => _failedLogs;
set => SetField(ref _failedLogs,         value); }
    public int    ActiveAccounts     { get =>
_activeAccounts;     set => SetField(ref _activeAccounts,
value); }
    public int    TotalRules         { get => _totalRules;
set => SetField(ref _totalRules,         value); }
    public string LastChecked        { get => _lastChecked;
set => SetField(ref _lastChecked,        value); }

    //  Derived success rate
    public double SuccessRate => TotalMessages == 0
        ? 0
```

```
        : Math.Round(ProcessedMessages /
(double)TotalMessages * 100, 1);

    public string SuccessRateText => $"{SuccessRate}%";

    //  Per-account status table
    public ObservableCollection<AccountStatus>
AccountStatuses { get; } = [];

    //  Recent log entries (last 10) —
    public ObservableCollection<ProcessingLog> RecentLogs {
get; } = [];

    //  Daily message chart data (last 7 days)
    public ObservableCollection<DayBar> DailyBars { get; } =
[];

    //  Commands
    public AsyncRelayCommand RefreshCommand { get; }

    public DashboardViewModel(IDbContextFactory<AppDbContext>
dbFactory)
    {
        _dbFactory     = dbFactory;
        RefreshCommand = new AsyncRelayCommand(LoadAsync);
    }

    //  Public entry point

    public async Task LoadAsync()
    {
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();

            //  Message counters
            TotalMessages     = await
db.EmailMessages.CountAsync();
            ProcessedMessages = await
db.EmailMessages.CountAsync(m => m.Status ==
MessageStatus.Processed);
            FailedMessages    = await
db.EmailMessages.CountAsync(m => m.Status ==
MessageStatus.Failed);
            SkippedMessages   = await
db.EmailMessages.CountAsync(m => m.Status ==
MessageStatus.Skipped);

            //  Attachment counters —
            TotalAttachments  = await
db.EmailAttachments.CountAsync();
```

```
            SavedAttachments  = await
db.EmailAttachments.CountAsync(a => a.Status ==
AttachmentStatus.Saved);
            SentAttachments   = await
db.EmailAttachments.CountAsync(a => a.Status ==
AttachmentStatus.Sent);
            FailedAttachments = await
db.EmailAttachments.CountAsync(a => a.Status ==
AttachmentStatus.Failed);

            //  Log counters
            TotalLogs  = await
db.ProcessingLogs.CountAsync();
            FailedLogs = await db.ProcessingLogs.CountAsync(l
=> l.Status == LogStatus.Failure);

            //  Config counters —
            ActiveAccounts = await
db.EmailAccounts.CountAsync(a => a.IsActive);
            TotalRules     = await
db.ProcessingRules.CountAsync(r => r.IsActive);

            //  Last checked
            var latest = await db.EmailAccounts
                .Where(a => a.LastCheckedAt.HasValue)
                .MaxAsync(a => (DateTime?)a.LastCheckedAt);
            LastChecked = latest.HasValue
                ? latest.Value.AddMonths(-
1).ToLocalTime().ToString("yyyy-MM-dd  HH:mm:ss")
                : "Never";

            OnPropertyChanged(nameof(SuccessRate));
            OnPropertyChanged(nameof(SuccessRateText));

            //  Per-account status
            var accounts = await db.EmailAccounts
                .AsNoTracking()
                .OrderBy(a => a.Name)
                .ToListAsync();

            AccountStatuses.Clear();
            foreach (var acc in accounts)
            {
                var msgCount  = await
db.EmailMessages.CountAsync(m => m.AccountId == acc.Id);
                var failCount = await
db.EmailMessages.CountAsync(m => m.AccountId == acc.Id &&
m.Status == MessageStatus.Failed);
                AccountStatuses.Add(new AccountStatus
                {
                    Name         = acc.Name,
                    EmailAddress = acc.EmailAddress,
                    IsActive     = acc.IsActive,
```

```
                    LastChecked  = acc.LastCheckedAt.HasValue
                                   ?
acc.LastCheckedAt.Value.AddMonths(-
1).ToLocalTime().ToString("HH:mm:ss")
                                   : "—",
                    MessageCount = msgCount,
                    FailedCount  = failCount,
                    StatusLabel  = acc.IsActive ? "Active" :
"Inactive"
                });
            }

            //  Recent logs —
            var logs = await db.ProcessingLogs
                .Include(l => l.Account)
                .Include(l => l.Rule)
                .AsNoTracking()
                .OrderByDescending(l => l.LoggedAt)
                .Take(10)
                .ToListAsync();

            RecentLogs.Clear();
            foreach (var l in logs) RecentLogs.Add(l);

            //  Daily bar chart (last 7 days) —
            var since = DateTime.UtcNow.Date.AddDays(-6);
            var daily = await db.EmailMessages
                .Where(m => m.ReceivedAt >= since)
                .GroupBy(m => m.ReceivedAt.Date)
                .Select(g => new { Date = g.Key, Count =
g.Count() })
                .ToListAsync();

            DailyBars.Clear();
            for (int i = 6; i >= 0; i--)
            {
                var day   = DateTime.UtcNow.Date.AddDays(-i);
                var count = daily.FirstOrDefault(d => d.Date
== day)?.Count ?? 0;
                DailyBars.Add(new DayBar { Label =
day.ToString("ddd"), Count = count });
            }

            SetStatus($"Last refreshed:
{DateTime.Now:HH:mm:ss}");
        }
        catch (Exception ex)
        {
            SetError(ex);
        }
        finally
        {
            IsBusy = false;
```

```
        }
    }
}

//  Supporting data models

public sealed class AccountStatus
{
    public string Name         { get; init; } = string.Empty;
    public string EmailAddress { get; init; } = string.Empty;
    public bool   IsActive     { get; init; }
    public string LastChecked  { get; init; } = "—";
    public int    MessageCount { get; init; }
    public int    FailedCount  { get; init; }
    public string StatusLabel  { get; init; } = string.Empty;
}

public sealed class DayBar
{
    public string Label { get; init; } = string.Empty;
    public int    Count { get; init; }
}

using System.Collections.ObjectModel;
using System.Windows;
using EmailProcessor.App.Services;
using EmailProcessor.Infrastructure.Data;
using EmailProcessor.WPF.Commands;
using Microsoft.EntityFrameworkCore;

namespace EmailProcessor.WPF.ViewModels;

/// <summary>
/// Root ViewModel for the application shell.
/// Manages page navigation, dashboard statistics, and the
/// background processing service start/stop cycle.
/// </summary>
public sealed class MainViewModel : ViewModelBase
{
    private readonly BackgroundPollingService _polling;
    private readonly IDbContextFactory<AppDbContext>
_dbFactory;

    //  Navigation
    private ViewModelBase _currentPage;

    public ViewModelBase CurrentPage
    {
        get => _currentPage;
        private set => SetField(ref _currentPage, value);
    }
```

```
    public ObservableCollection<NavItem> NavItems { get; } =
[];

    //  Dashboard stats —
    private int    _totalProcessed;
    private int    _totalSuccessful;
    private int    _totalFailed;
    private string _lastChecked = "Never";
    private string _serviceStatus = "Stopped";

    public int    TotalProcessed  { get => _totalProcessed;
set => SetField(ref _totalProcessed,  value); }
    public int    TotalSuccessful { get => _totalSuccessful;
set => SetField(ref _totalSuccessful, value); }
    public int    TotalFailed     { get => _totalFailed;
set => SetField(ref _totalFailed,     value); }
    public string LastChecked     { get => _lastChecked;
set => SetField(ref _lastChecked,     value); }
    public string ServiceStatus   { get => _serviceStatus;
set => SetField(ref _serviceStatus,   value); }

    //  Commands
    public AsyncRelayCommand StartServiceCommand  { get; }
    public AsyncRelayCommand StopServiceCommand   { get; }
    public AsyncRelayCommand ProcessNowCommand    { get; }
    public AsyncRelayCommand RefreshStatsCommand  { get; }
    public RelayCommand      NavigateCommand      { get; }

    //  Child ViewModels
    public DashboardViewModel          DashboardVm   { get; }
    public EmailAccountsViewModel  AccountsVm  { get; }
    public ProcessingRulesViewModel RulesVm    { get; }
    public ProcessingActionsViewModel ActionsVm { get; }
    public ProcessedMessagesViewModel MessagesVm { get; }
    public AttachmentsViewModel AttachmentsVm  { get; }
    public ProcessingLogsViewModel LogsVm      { get; }
    public SettingsViewModel SettingsVm        { get; }

    public MainViewModel(
        BackgroundPollingService polling,
        IDbContextFactory<AppDbContext> dbFactory,
        DashboardViewModel         dashboardVm,
        EmailAccountsViewModel accountsVm,
        ProcessingRulesViewModel rulesVm,
        ProcessingActionsViewModel actionsVm,
        ProcessedMessagesViewModel messagesVm,
        AttachmentsViewModel attachmentsVm,
        ProcessingLogsViewModel logsVm,
        SettingsViewModel settingsVm)
    {
        _polling   = polling;
        _dbFactory = dbFactory;
```

```
        AccountsVm    = accountsVm;
        RulesVm       = rulesVm;
        ActionsVm     = actionsVm;
        MessagesVm    = messagesVm;
        AttachmentsVm = attachmentsVm;
        LogsVm        = logsVm;
        SettingsVm    = settingsVm;
        DashboardVm = dashboardVm;

        _currentPage  = AccountsVm; // default page

        // Build navigation sidebar items
        NavItems.Add(new NavItem("Dashboard",          "ti-
layout-dashboard", DashboardVm));
        NavItems.Add(new NavItem("Email Accounts",     "ti-
mail",             AccountsVm));
        NavItems.Add(new NavItem("Processing Rules",   "ti-
filter",           RulesVm));
        NavItems.Add(new NavItem("Actions",            "ti-
bolt",             ActionsVm));
        NavItems.Add(new NavItem("Messages",           "ti-
inbox",            MessagesVm));
        NavItems.Add(new NavItem("Attachments",        "ti-
paperclip",        AttachmentsVm));
        NavItems.Add(new NavItem("Logs",               "ti-
list-details",     LogsVm));
        NavItems.Add(new NavItem("Settings",           "ti-
settings",         SettingsVm));

        // Wire up background service events
        //  Background service wiring
        _polling.StateChanged += async (_, _) =>
        {
            await
Application.Current.Dispatcher.InvokeAsync(OnServiceStateChan
ged);

            // Auto-refresh the dashboard after each
completed cycle
            if (!_polling.IsRunning || _polling.StatusText ==
"Running")
                await
Application.Current.Dispatcher.InvokeAsync(
                    () => _ = DashboardVm.LoadAsync());
        };

        //  Commands —
        StartServiceCommand = new AsyncRelayCommand(
            async () => { _polling.Start(); await
DashboardVm.LoadAsync(); },
            () => !_polling.IsRunning);

        StopServiceCommand = new AsyncRelayCommand(
```

```
            async () => { _polling.Stop(); await
DashboardVm.LoadAsync(); },
            () => _polling.IsRunning);

        ProcessNowCommand = new AsyncRelayCommand(async () =>
        {
            IsBusy = true;
            try   { await _polling.TriggerNowAsync(); }
            finally { IsBusy = false; }
            await DashboardVm.LoadAsync();
        });

        NavigateCommand = new RelayCommand(vm =>
        {
            if (vm is ViewModelBase page) CurrentPage = page;
        });
    }

    //  Initialisation

    public async Task InitialiseAsync()
    {
        await DashboardVm.LoadAsync();
        await AccountsVm.LoadCommand.ExecuteAsync(null);
        await RulesVm.LoadCommand.ExecuteAsync(null);
        await SettingsVm.LoadCommand.ExecuteAsync(null);
        await ActionsVm.LoadCommand.ExecuteAsync(null);
        await MessagesVm.LoadCommand.ExecuteAsync(null);
        await LogsVm.SearchCommand.ExecuteAsync(null);
        await AttachmentsVm.LoadCommand.ExecuteAsync(null);
    }

    //  Stats refresh —

    private async Task RefreshStatsAsync()
    {
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();

            TotalProcessed  = await
db.EmailMessages.CountAsync();
            TotalSuccessful = await db.EmailAttachments
                .CountAsync(a => a.Status ==
Domain.Enums.AttachmentStatus.Saved
                              || a.Status ==
Domain.Enums.AttachmentStatus.Sent);
            TotalFailed     = await db.ProcessingLogs
                .CountAsync(l => l.Status ==
Domain.Enums.LogStatus.Failure);

            var latest = await db.EmailAccounts
```

```
                .Where(a => a.LastCheckedAt.HasValue)
                .MaxAsync(a => (DateTime?)a.LastCheckedAt);

            LastChecked = latest.HasValue
                ? latest.Value.ToLocalTime().ToString("yyyy-
MM-dd HH:mm")
                : "Never";
        }
        catch (Exception ex)
        {
            SetError(ex);
        }
    }

    private void OnServiceStateChanged()
    {
        ServiceStatus = _polling.StatusText;
        StartServiceCommand.RaiseCanExecuteChanged();
        StopServiceCommand.RaiseCanExecuteChanged();
    }
}

//  Navigation item

public sealed record NavItem(string Title, string Icon,
ViewModelBase? ViewModel);

using EmailProcessor.Domain.Enums;
using EmailProcessor.Domain.Interfaces;
using EmailProcessor.Domain.Models;
using EmailProcessor.Infrastructure.Data;
using EmailProcessor.WPF.Commands;
using Microsoft.EntityFrameworkCore;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace EmailProcessor.WPF.ViewModels;

// —
// ObservableObject
// —

/// <summary>Base class that implements
INotifyPropertyChanged.</summary>
public abstract class ObservableObject :
INotifyPropertyChanged
{
    public event PropertyChangedEventHandler?
PropertyChanged;

    protected void OnPropertyChanged([CallerMemberName]
string? name = null) =>
```

```
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(name));

    protected bool SetField<T>(ref T field, T value,
[CallerMemberName] string? name = null)
    {
        if (EqualityComparer<T>.Default.Equals(field, value))
return false;
        field = value;
        OnPropertyChanged(name);
        return true;
    }
}

// —
// ViewModelBase
// —

/// <summary>
/// Base ViewModel with common UI-state properties:
/// IsBusy, StatusMessage, and an error-display helper.
/// </summary>
public abstract class ViewModelBase : ObservableObject
{
    private bool _isBusy;
    private string _statusMessage = string.Empty;

    public bool IsBusy
    {
        get => _isBusy;
        set => SetField(ref _isBusy, value);
    }

    public string StatusMessage
    {
        get => _statusMessage;
        set => SetField(ref _statusMessage, value);
    }

    protected void SetStatus(string message) => StatusMessage
= message;

    protected void SetError(Exception ex)
    {
        StatusMessage = $"Error: {ex.Message}";
        IsBusy = false;
    }
}


//
═══════════════════════════════════════════════════════════
═══════════
```

```
// EmailAccountsViewModel
//
═══════════════════════════════════════════════════════════
═══════════════

/// <summary>CRUD management for email accounts, with
connection-test support.</summary>
public sealed class EmailAccountsViewModel : ViewModelBase
{
    private readonly IDbContextFactory<AppDbContext>
_dbFactory;
    private readonly IEmailService _emailService;
    private readonly IEncryptionService _encryption;

    public ObservableCollection<EmailAccount> Accounts { get;
} = [];

    private EmailAccount? _selected;
    public EmailAccount? Selected
    {
        get => _selected;
        set
        {
            SetField(ref _selected, value);
            EditCopy = value is null ? null :
CloneAccount(value);
            IsEditing = false;
        }
    }

    private EmailAccount? _editCopy;
    public EmailAccount? EditCopy
    {
        get => _editCopy;
        set => SetField(ref _editCopy, value);
    }

    private bool _isEditing;
    public bool IsEditing
    {
        get => _isEditing;
        set => SetField(ref _isEditing, value);
    }

    // Separate plain-text password field — never bound to
the stored encrypted value
    private string _plainPassword = string.Empty;
    public string PlainPassword
    {
        get => _plainPassword;
        set => SetField(ref _plainPassword, value);
    }
```

```
    public AsyncRelayCommand LoadCommand { get; }
    public AsyncRelayCommand SaveCommand { get; }
    public AsyncRelayCommand DeleteCommand { get; }
    public AsyncRelayCommand TestCommand { get; }
    public RelayCommand NewCommand { get; }
    public RelayCommand EditCommand { get; }
    public RelayCommand CancelEditCommand { get; }


    public EmailAccountsViewModel(
        IDbContextFactory<AppDbContext> dbFactory,
        IEmailService emailService,
        IEncryptionService encryption)
    {
        _dbFactory = dbFactory;
        _emailService = emailService;
        _encryption = encryption;

        LoadCommand = new AsyncRelayCommand(LoadAsync);
        SaveCommand = new AsyncRelayCommand(SaveAsync, () =>
IsEditing && EditCopy is not null);
        DeleteCommand = new AsyncRelayCommand(DeleteAsync, ()
=> Selected is not null);
        TestCommand = new
AsyncRelayCommand(TestConnectionAsync, () => Selected is not
null);

        NewCommand = new RelayCommand(() => { EditCopy =
NewAccount(); IsEditing = true; });
        EditCommand = new RelayCommand(() => IsEditing =
true, () => Selected is not null);
        CancelEditCommand = new RelayCommand(() => { EditCopy
= Selected is null ? null : CloneAccount(Selected); IsEditing
= false; });
    }

    private async Task LoadAsync()
    {
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();
            var list = await db.EmailAccounts.OrderBy(a =>
a.Name).ToListAsync();
            Accounts.Clear();
            foreach (var a in list) Accounts.Add(a);
        }
        finally { IsBusy = false; }
    }

    private async Task SaveAsync()
    {
```

```
        if (EditCopy is null) return;
        IsBusy = true;
        try
        {
            // Encrypt password if the user typed one in
            if (!string.IsNullOrWhiteSpace(PlainPassword))
                EditCopy.EncryptedPassword =
_encryption.Encrypt(PlainPassword);

            await using var db = await
_dbFactory.CreateDbContextAsync();

            if (EditCopy.Id == 0)
                db.EmailAccounts.Add(EditCopy);
            else
                db.EmailAccounts.Update(EditCopy);

            await db.SaveChangesAsync();
            PlainPassword = string.Empty;
            IsEditing = false;
            await LoadAsync();
            SetStatus("Account saved.");
        }
        catch (Exception ex) { SetError(ex); }
        finally { IsBusy = false; }
    }

    private async Task DeleteAsync()
    {
        if (Selected is null) return;
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();
            await db.EmailAccounts.Where(a => a.Id ==
Selected.Id).ExecuteDeleteAsync();
            await LoadAsync();
            SetStatus("Account deleted.");
        }
        catch (Exception ex) { SetError(ex); }
        finally { IsBusy = false; }
    }

    private async Task TestConnectionAsync()
    {
        if (Selected is null) return;
        IsBusy = true;
        SetStatus("Testing connection…");
        try
        {
            var ok = await
_emailService.TestConnectionAsync(Selected);
```

```
            SetStatus(ok ? "Connection successful ✓" :
"Connection failed ✗");
        }
        catch (Exception ex) { SetError(ex); }
        finally { IsBusy = false; }
    }

    private static EmailAccount CloneAccount(EmailAccount a)
=> new()
    {
        Id = a.Id,
        Name = a.Name,
        EmailAddress = a.EmailAddress,
        ImapServer = a.ImapServer,
        ImapPort = a.ImapPort,
        UseSsl = a.UseSsl,
        Username = a.Username,
        EncryptedPassword = a.EncryptedPassword,
        MonitoredFolder = a.MonitoredFolder,
        PollingIntervalSeconds = a.PollingIntervalSeconds,
        IsActive = a.IsActive
    };

    private static EmailAccount NewAccount() => new()
    {
        ImapPort = 993,
        UseSsl = true,
        MonitoredFolder = "INBOX",
        PollingIntervalSeconds = 300,
        IsActive = true
    };
}

//
============================================================
============
// ProcessingRulesViewModel
//
============================================================
============

public sealed class ProcessingRulesViewModel : ViewModelBase
{
    private readonly IDbContextFactory<AppDbContext>
_dbFactory;

    public ObservableCollection<ProcessingRule> Rules { get;
} = [];

    private ProcessingRule? _selected;
    public ProcessingRule? Selected { get => _selected; set {
SetField(ref _selected, value); EditCopy = value is null ?
null : CloneRule(value); IsEditing = false; } }
```

```
    private ProcessingRule? _editCopy;
    public ProcessingRule? EditCopy { get => _editCopy; set
=> SetField(ref _editCopy, value); }

    private bool _isEditing;
    public bool IsEditing { get => _isEditing; set =>
SetField(ref _isEditing, value); }

    public AsyncRelayCommand LoadCommand { get; }
    public AsyncRelayCommand SaveCommand { get; }
    public AsyncRelayCommand DeleteCommand { get; }
    public RelayCommand NewCommand { get; }
    public RelayCommand EditCommand { get; }
    public RelayCommand CancelEditCommand { get; }

    public
ProcessingRulesViewModel(IDbContextFactory<AppDbContext>
dbFactory)
    {
        _dbFactory = dbFactory;
        LoadCommand = new AsyncRelayCommand(LoadAsync);
        SaveCommand = new AsyncRelayCommand(SaveAsync, () =>
IsEditing && EditCopy is not null);
        DeleteCommand = new AsyncRelayCommand(DeleteAsync, ()
=> Selected is not null);
        NewCommand = new RelayCommand(() => { EditCopy = new
ProcessingRule { Priority = 100, IsActive = true }; IsEditing
= true; });
        EditCommand = new RelayCommand(() => IsEditing =
true, () => Selected is not null);
        CancelEditCommand = new RelayCommand(() => { EditCopy
= Selected is null ? null : CloneRule(Selected); IsEditing =
false; });
    }

    private async Task LoadAsync()
    {
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();
            var list = await db.ProcessingRules.Include(r =>
r.Actions).OrderBy(r => r.Priority).ToListAsync();
            Rules.Clear();
            foreach (var r in list) Rules.Add(r);
        }
        finally { IsBusy = false; }
    }

    private async Task SaveAsync()
    {
```

```
        if (EditCopy is null) return;
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();
            if (EditCopy.Id == 0)
db.ProcessingRules.Add(EditCopy);
            else db.ProcessingRules.Update(EditCopy);
            await db.SaveChangesAsync();
            IsEditing = false;
            await LoadAsync();
            SetStatus("Rule saved.");
        }
        catch (Exception ex) { SetError(ex); }
        finally { IsBusy = false; }
    }

    private async Task DeleteAsync()
    {
        if (Selected is null) return;
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();
            await db.ProcessingRules.Where(r => r.Id ==
Selected.Id).ExecuteDeleteAsync();
            await LoadAsync();
            SetStatus("Rule deleted.");
        }
        catch (Exception ex) { SetError(ex); }
        finally { IsBusy = false; }
    }

    private static ProcessingRule CloneRule(ProcessingRule r)
=> new()
    {
        Id = r.Id,
        Name = r.Name,
        IsActive = r.IsActive,
        Priority = r.Priority,
        SenderContains = r.SenderContains,
        SubjectContains = r.SubjectContains,
        BodyContains = r.BodyContains,
        DateFrom = r.DateFrom,
        DateTo = r.DateTo,
        MustHaveAttachments = r.MustHaveAttachments,
        AttachmentExtensions = r.AttachmentExtensions,
        AttachmentNameContains = r.AttachmentNameContains
    };
}
```

```
//
═══════════════════════════════════════════════════════════
═══════════
// ProcessingActionsViewModel
//
═══════════════════════════════════════════════════════════
═══════════

public sealed class ProcessingActionsViewModel :
ViewModelBase
{
    private readonly IDbContextFactory<AppDbContext>
_dbFactory;

    public ObservableCollection<ProcessingRule> Rules { get;
} = [];
    public ObservableCollection<ProcessingAction> Actions {
get; } = [];

    private ProcessingRule? _selectedRule;
    public ProcessingRule? SelectedRule
    {
        get => _selectedRule;
        set { SetField(ref _selectedRule, value); _ =
LoadActionsAsync(); }
    }

    private ProcessingAction? _selected;
    public ProcessingAction? Selected { get => _selected; set
{ SetField(ref _selected, value); EditCopy = value is null ?
null : CloneAction(value); IsEditing = false; } }

    private ProcessingAction? _editCopy;
    public ProcessingAction? EditCopy { get => _editCopy; set
=> SetField(ref _editCopy, value); }

    private bool _isEditing;
    public bool IsEditing { get => _isEditing; set =>
SetField(ref _isEditing, value); }

    public IReadOnlyList<ActionType> ActionTypes { get; } =
Enum.GetValues<ActionType>();

    public AsyncRelayCommand LoadCommand { get; }
    public AsyncRelayCommand SaveCommand { get; }
    public AsyncRelayCommand DeleteCommand { get; }
    public RelayCommand NewCommand { get; }
    public RelayCommand EditCommand { get; }
    public RelayCommand CancelEditCommand { get; }

    public
ProcessingActionsViewModel(IDbContextFactory<AppDbContext>
dbFactory)
```

```
    {
        _dbFactory = dbFactory;
        LoadCommand = new AsyncRelayCommand(LoadRulesAsync);
        SaveCommand = new AsyncRelayCommand(SaveAsync, () =>
IsEditing && EditCopy is not null);
        DeleteCommand = new AsyncRelayCommand(DeleteAsync, ()
=> Selected is not null);
        NewCommand = new RelayCommand(() =>
        {
            if (SelectedRule is null) return;
            EditCopy = new ProcessingAction { RuleId =
SelectedRule.Id, IsActive = true, ActionType =
ActionType.SaveToFolder };
            IsEditing = true;
        }, () => SelectedRule is not null);
        EditCommand = new RelayCommand(() => IsEditing =
true, () => Selected is not null);
        CancelEditCommand = new RelayCommand(() => { EditCopy
= Selected is null ? null : CloneAction(Selected); IsEditing
= false; });
    }

    private async Task LoadRulesAsync()
    {
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();
            var list = await db.ProcessingRules.OrderBy(r =>
r.Priority).ToListAsync();
            Rules.Clear();
            foreach (var r in list) Rules.Add(r);
        }
        finally { IsBusy = false; }
    }

    private async Task LoadActionsAsync()
    {
        if (SelectedRule is null) { Actions.Clear(); return;
}
        await using var db = await
_dbFactory.CreateDbContextAsync();
        var list = await db.ProcessingActions.Where(a =>
a.RuleId == SelectedRule.Id).ToListAsync();
        Actions.Clear();
        foreach (var a in list) Actions.Add(a);
    }

    private async Task SaveAsync()
    {
        if (EditCopy is null) return;
        IsBusy = true;
```

```
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();
            if (EditCopy.Id == 0)
db.ProcessingActions.Add(EditCopy);
            else db.ProcessingActions.Update(EditCopy);
            await db.SaveChangesAsync();
            IsEditing = false;
            await LoadActionsAsync();
            SetStatus("Action saved.");
        }
        catch (Exception ex) { SetError(ex); }
        finally { IsBusy = false; }
    }

    private async Task DeleteAsync()
    {
        if (Selected is null) return;
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();
            await db.ProcessingActions.Where(a => a.Id ==
Selected.Id).ExecuteDeleteAsync();
            await LoadActionsAsync();
            SetStatus("Action deleted.");
        }
        catch (Exception ex) { SetError(ex); }
        finally { IsBusy = false; }
    }

    private static ProcessingAction
CloneAction(ProcessingAction a) => new()
    {
        Id = a.Id,
        RuleId = a.RuleId,
        Name = a.Name,
        ActionType = a.ActionType,
        IsActive = a.IsActive,
        TargetFolder = a.TargetFolder,
        ApiUrl = a.ApiUrl,
        HttpMethod = a.HttpMethod,
        ApiHeaders = a.ApiHeaders,
        AuthToken = a.AuthToken,
        SendAsMultipart = a.SendAsMultipart,
        SendMetadataAsJson = a.SendMetadataAsJson,
        MoveToFolder = a.MoveToFolder
    };
}
```

```
//
================================================================
==============
// ProcessedMessagesViewModel
//
================================================================
==============

public sealed class ProcessedMessagesViewModel :
ViewModelBase
{
    private readonly IDbContextFactory<AppDbContext>
_dbFactory;

    public ObservableCollection<EmailMessage> Messages { get;
} = [];
    private string _searchText = string.Empty;
    public string SearchText { get => _searchText; set {
SetField(ref _searchText, value); _ = LoadAsync(); } }

    public AsyncRelayCommand LoadCommand { get; }

    public
ProcessedMessagesViewModel(IDbContextFactory<AppDbContext>
dbFactory)
    {
        _dbFactory = dbFactory;
        LoadCommand = new AsyncRelayCommand(LoadAsync);
    }

    private async Task LoadAsync()
    {
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();
            var q = db.EmailMessages
                .Include(m => m.Account)
                .AsNoTracking()
                .OrderByDescending(m => m.ReceivedAt)
                .AsQueryable();

            if (!string.IsNullOrWhiteSpace(SearchText))
                q = q.Where(m =>
m.Subject.Contains(SearchText) ||
m.Sender.Contains(SearchText));

            var list = await q.Take(500).ToListAsync();
            Messages.Clear();
            foreach (var m in list) Messages.Add(m);
        }
        finally { IsBusy = false; }
```

```
    }
}

//
================================================================
================
// AttachmentsViewModel
//
================================================================
================

public sealed class AttachmentsViewModel : ViewModelBase
{
    private readonly IDbContextFactory<AppDbContext>
_dbFactory;
    public ObservableCollection<EmailAttachment> Attachments
{ get; } = [];

    public AsyncRelayCommand LoadCommand { get; }

    public
AttachmentsViewModel(IDbContextFactory<AppDbContext>
dbFactory)
    {
        _dbFactory = dbFactory;
        LoadCommand = new AsyncRelayCommand(LoadAsync);
    }

    private async Task LoadAsync()
    {
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();
            var list = await db.EmailAttachments
                .Include(a => a.Message)
                .AsNoTracking()
                .OrderByDescending(a => a.ProcessedAt)
                .Take(500)
                .ToListAsync();
            Attachments.Clear();
            foreach (var a in list) Attachments.Add(a);
        }
        finally { IsBusy = false; }
    }
}

//
================================================================
================
// ProcessingLogsViewModel
```

```
//
═══════════════════════════════════════════════════════════
═══════════

public sealed class ProcessingLogsViewModel : ViewModelBase
{
    private readonly ILoggingService _loggingService;

    public ObservableCollection<ProcessingLog> Logs { get; }
= [];

    // Filter properties
    private DateTime? _dateFrom;
    private DateTime? _dateTo;
    private string _filterSender = string.Empty;
    private string _filterSubject = string.Empty;
    private int? _filterStatus;

    public DateTime? FilterDateFrom { get => _dateFrom; set
=> SetField(ref _dateFrom, value); }
    public DateTime? FilterDateTo { get => _dateTo; set =>
SetField(ref _dateTo, value); }
    public string FilterSender { get => _filterSender; set =>
SetField(ref _filterSender, value); }
    public string FilterSubject { get => _filterSubject; set
=> SetField(ref _filterSubject, value); }
    public int? FilterStatus { get => _filterStatus; set =>
SetField(ref _filterStatus, value); }

    public AsyncRelayCommand SearchCommand { get; }
    public RelayCommand ClearFiltersCommand { get; }

    public ProcessingLogsViewModel(ILoggingService
loggingService)
    {
        _loggingService = loggingService;
        SearchCommand = new AsyncRelayCommand(SearchAsync);
        ClearFiltersCommand = new RelayCommand(ClearFilters);
    }

    private async Task SearchAsync()
    {
        IsBusy = true;
        try
        {
            var query = new LogQuery
            {
                DateFrom = FilterDateFrom,
                DateTo = FilterDateTo,
                Sender = FilterSender,
                Subject = FilterSubject,
                Status = FilterStatus
            };
```

```
            var results = await
_loggingService.QueryAsync(query);
            Logs.Clear();
            foreach (var l in results) Logs.Add(l);
        }
        catch (Exception ex) { SetError(ex); }
        finally { IsBusy = false; }
    }

    private void ClearFilters()
    {
        FilterDateFrom = null;
        FilterDateTo = null;
        FilterSender = string.Empty;
        FilterSubject = string.Empty;
        FilterStatus = null;
    }
}

//
═══════════════════════════════════════════════════════════════
═══════════
// SettingsViewModel
//
═══════════════════════════════════════════════════════════════
═══════════

public sealed class SettingsViewModel : ViewModelBase
{
    private readonly IDbContextFactory<AppDbContext>
_dbFactory;

    private string _maxAttachmentSizeMb = "25";
    private string _allowedExtensions =
".pdf,.xlsx,.csv,.docx,.xml,.zip";
    private string _blockedExtensions =
".exe,.bat,.cmd,.vbs,.ps1,.sh";
    private bool _markMessagesAsRead = true;

    public string MaxAttachmentSizeMb { get =>
_maxAttachmentSizeMb; set => SetField(ref
_maxAttachmentSizeMb, value); }
    public string AllowedExtensions { get =>
_allowedExtensions; set => SetField(ref _allowedExtensions,
value); }
    public string BlockedExtensions { get =>
_blockedExtensions; set => SetField(ref _blockedExtensions,
value); }
    public bool MarkMessagesAsRead { get =>
_markMessagesAsRead; set => SetField(ref _markMessagesAsRead,
value); }
```

```
    public AsyncRelayCommand LoadCommand { get; }
    public AsyncRelayCommand SaveCommand { get; }

    public SettingsViewModel(IDbContextFactory<AppDbContext>
dbFactory)
    {
        _dbFactory = dbFactory;
        LoadCommand = new AsyncRelayCommand(LoadAsync);
        SaveCommand = new AsyncRelayCommand(SaveAsync);
    }

    private async Task LoadAsync()
    {
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();
            var settings = await
db.ApplicationSettings.ToDictionaryAsync(s => s.Key, s =>
s.Value);

            if (settings.TryGetValue("MaxAttachmentSizeMb",
out var v)) MaxAttachmentSizeMb = v;
            if (settings.TryGetValue("AllowedExtensions", out
var e)) AllowedExtensions = e;
            if (settings.TryGetValue("BlockedExtensions", out
var b)) BlockedExtensions = b;
            if (settings.TryGetValue("MarkMessagesAsRead",
out var m)) MarkMessagesAsRead = bool.TryParse(m, out var
parsed) && parsed;
        }
        finally { IsBusy = false; }
    }

    private async Task SaveAsync()
    {
        IsBusy = true;
        try
        {
            await using var db = await
_dbFactory.CreateDbContextAsync();

            async Task UpsertAsync(string key, string value)
            {
                var s = await
db.ApplicationSettings.FirstOrDefaultAsync(x => x.Key ==
key);
                if (s is null) db.ApplicationSettings.Add(new
ApplicationSetting { Key = key, Value = value });
                else s.Value = value;
            }
```

```
            await UpsertAsync("MaxAttachmentSizeMb",
MaxAttachmentSizeMb);
            await UpsertAsync("AllowedExtensions",
AllowedExtensions);
            await UpsertAsync("BlockedExtensions",
BlockedExtensions);
            await UpsertAsync("MarkMessagesAsRead",
MarkMessagesAsRead.ToString().ToLower());

            await db.SaveChangesAsync();
            SetStatus("Settings saved.");
        }
        catch (Exception ex) { SetError(ex); }
        finally { IsBusy = false; }
    }
}
```

ДОДАТОК В. Опис програмного забезпечення

Розроблене програмне забезпечення призначене для автоматизації процесу отримання, фільтрації та обробки повідомлень електронної пошти. Застосунок забезпечує підключення до поштових скриньок за протоколом IMAP, отримання нових листів, перевірку повідомлень за налаштованими правилами, обробку вкладених файлів, виконання дій із ними та збереження результатів роботи в базі даних.

Основним користувачем програмного забезпечення є відповідальна особа або адміністратор, який налаштовує поштові облікові записи, створює правила обробки повідомлень, визначає дії після спрацювання правил і контролює результати роботи системи. Після початкового налаштування застосунок може працювати у фоновому режимі, автоматично перевіряючи активні поштові скриньки відповідно до заданих інтервалів.

До складу програмного забезпечення входять такі основні функціональні частини:

– модуль керування поштовими обліковими записами;

– модуль підключення до поштових скриньок;

– модуль правил обробки повідомлень;

– модуль дій після спрацювання правил;

– модуль обробки вкладень;

– модуль передавання даних до зовнішнього API;

– модуль журналювання;

– модуль фонового опитування поштових скриньок;

– модуль роботи з базою даних;

– користувацький інтерфейс для налаштування та контролю роботи системи.

Модуль керування поштовими обліковими записами призначений для

додавання, редагування, перегляду, деактивації та видалення поштових скриньок, які перевіряються застосунком. Для кожного облікового запису користувач задає назву, адресу електронної пошти, IMAP-сервер, порт, логін, пароль або пароль застосунку, папку для моніторингу, інтервал перевірки та ознаку активності.

Модуль підключення до поштових скриньок забезпечує взаємодію з поштовим сервером за протоколом IMAP. Він відповідає за встановлення з’єднання, відкриття визначеної поштової папки, отримання нових повідомлень, позначення листів як прочитаних і переміщення повідомлень до іншої папки. Для роботи з електронною поштою використовується бібліотека MailKit.

Модуль правил обробки повідомлень дозволяє визначати умови, за яких електронний лист повинен бути оброблений системою. Правила можуть враховувати адресу відправника, тему повідомлення, текст листа, дату надсилання, наявність вкладень, розширення вкладених файлів і частину назви вкладення. Правила мають пріоритет і ознаку активності, що дозволяє керувати порядком їх перевірки.

Модуль дій після спрацювання правил визначає, які операції має виконати система після знаходження відповідного повідомлення. До підтримуваних дій належать збереження вкладення у визначену папку, передавання вкладення до зовнішнього API, передавання метаданих повідомлення у форматі JSON і переміщення листа до іншої папки поштової скриньки. Одне правило може мати декілька дій, що дозволяє реалізовувати складніші сценарії автоматичної обробки.

Модуль обробки вкладень відповідає за перевірку вкладених файлів, очищення назв файлів від недопустимих символів, блокування потенційно небезпечних розширень, формування унікальних імен файлів і збереження інформації про вкладення в базі даних. Завантажені файли не запускаються автоматично, а лише зберігаються або передаються до зовнішньої системи відповідно до налаштованих правил.

Модуль передавання даних до зовнішнього API забезпечує інтеграцію

застосунку з іншими інформаційними системами. Він формує HTTP-запити на основі параметрів дії, додає необхідні заголовки, токен авторизації, метадані повідомлення та, за потреби, вкладений файл. Передавання вкладення може виконуватися у форматі multipart/form-data, а службова інформація про лист може передаватися у форматі JSON.

Модуль журналювання призначений для фіксації результатів роботи системи. У журналі зберігаються успішні операції, попередження та помилки, що виникають під час підключення до поштової скриньки, отримання повідомлень, перевірки правил, збереження вкладень, передавання даних до API або роботи з базою даних. Запис журналу містить дату й час події, поштовий обліковий запис, відправника, тему листа, правило, вкладення, виконану дію, статус і текст помилки за наявності.

Модуль фонового опитування поштових скриньок забезпечує автоматичну роботу застосунку без постійної участі користувача. Він періодично перевіряє активні поштові облікові записи відповідно до заданого інтервалу та запускає повний цикл обробки повідомлень. Крім автоматичного режиму, користувач може виконати ручний запуск перевірки пошти через інтерфейс застосунку.

Для збереження даних використовується база даних Microsoft SQL Server. У базі даних зберігаються поштові облікові записи, правила обробки, дії, отримані повідомлення, вкладення, записи журналу та загальні налаштування застосунку. Доступ до бази даних реалізовано за допомогою Entity Framework Core через клас AppDbContext.

Структура бази даних включає таблиці EmailAccounts, ProcessingRules, ProcessingActions, EmailMessages, EmailAttachments, ProcessingLogs і ApplicationSettings. Таблиця EmailAccounts містить параметри поштових скриньок. Таблиця ProcessingRules зберігає умови обробки повідомлень. Таблиця ProcessingActions містить дії, які виконуються після спрацювання правил. Таблиця EmailMessages зберігає метадані отриманих листів. Таблиця EmailAttachments містить інформацію про вкладені файли. Таблиця ProcessingLogs використовується для журналювання результатів обробки.

Таблиця ApplicationSettings призначена для збереження загальних параметрів системи.

Для запобігання повторній обробці листів у програмному забезпеченні використовується IMAP UID повідомлення в межах конкретного поштового облікового запису. Комбінація ідентифікатора поштового облікового запису та IMAP UID є унікальною, що дозволяє системі визначати, які повідомлення вже були отримані та оброблені.

З метою захисту облікових даних паролі поштових скриньок не зберігаються у відкритому вигляді. Перед збереженням вони шифруються з використанням механізму Windows Data Protection API. Під час підключення до поштової скриньки пароль розшифровується і використовується лише для автентифікації на поштовому сервері.

Розроблене програмне забезпечення забезпечує автоматизацію повторюваних операцій, пов’язаних з обробкою електронної пошти. Його використання дозволяє зменшити обсяг ручної роботи, скоротити час обробки вкладених файлів, уникнути повторної обробки повідомлень і забезпечити контрольованість процесу завдяки веденню журналу подій.

ДОДАТОК Г. Інструкції програмісту, оператору та користувачеві програмного забезпечення

Програмне забезпечення реалізовано як настільний застосунок для операційної системи Windows. Для роботи застосунку необхідна наявність доступу до поштового сервера з підтримкою протоколу IMAP, бази даних Microsoft SQL Server, локальної або мережевої папки для збереження вкладень, а також, за потреби, доступу до зовнішнього API для передавання вкладень або метаданих повідомлень.

Інструкція програмісту

Перед початком експлуатації необхідно перевірити, що на робочому комп'ютері встановлено необхідне програмне середовище, налаштовано підключення до бази даних, а також забезпечено доступ до поштової скриньки, яка буде оброблятися системою. Для поштових сервісів, які не дозволяють використовувати основний пароль облікового запису для зовнішніх застосунків, необхідно створити окремий пароль застосунку.

Програміст виконує встановлення, супровід, оновлення та, за потреби, модифікацію програмного забезпечення. Перед початком роботи з програмним кодом необхідно отримати актуальну версію проєкту, відкрити його в середовищі розробки та перевірити наявність усіх необхідних залежностей.

Для відкриття проєкту необхідно:

– скопіювати або завантажити вихідний код програмного забезпечення;

– відкрити файл рішення в середовищі Visual Studio;

– відновити NuGet-пакети, які використовуються у проєкті;

– перевірити параметри підключення до бази даних;

– виконати побудову проєкту;

– запустити застосунок у режимі налагодження або звичайного виконання.

Основними зовнішніми бібліотеками та технологіями, які

використовуються в програмному забезпеченні, є WPF, Entity Framework Core, MailKit, MimeKit, HttpClient і механізм Windows DPAPI. WPF використовується для створення графічного інтерфейсу, Entity Framework Core — для роботи з базою даних, MailKit і MimeKit — для отримання та аналізу електронних повідомлень, HttpClient — для взаємодії із зовнішніми API, а DPAPI — для захищеного зберігання паролів поштових облікових записів.

На рисунку Г.1 показано структуру проєкту програмного забезпечення в середовищі розробки. Програміст може використовувати її для швидкого орієнтування в основних частинах застосунку: інтерфейсному шарі, ViewModel-класах, сервісах прикладної логіки, доменних сутностях, класах доступу до бази даних і конфігураційних файлах. Такий поділ допомагає виконувати супровід програмного забезпечення без порушення логіки взаємодії між шарами.

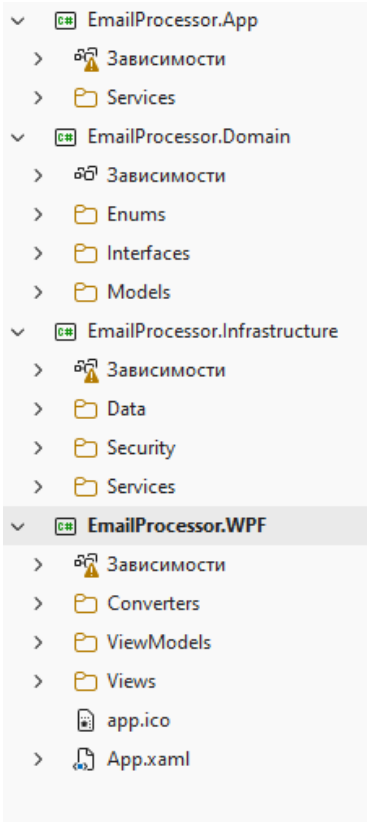


Рисунок Г.1 – Структура проєкту програмного забезпечення в середовищі розробки

Під час внесення змін до структури бази даних необхідно оновити

відповідні класи сутностей, конфігурації Entity Framework Core та міграції бази даних. Після зміни структури таблиць потрібно перевірити, чи не порушено зв’язки між сутностями, індекси, обмеження унікальності та правила видалення пов’язаних записів. Особливу увагу слід приділяти унікальному обмеженню для пари AccountId та ImapUid у таблиці EmailMessages, оскільки воно використовується для запобігання повторній обробці повідомлень.

У разі зміни механізму отримання електронної пошти необхідно працювати з реалізацією інтерфейсу IEmailService. Поточна реалізація MailKitEmailService використовує протокол IMAP. Якщо в майбутньому буде додано інший механізм отримання повідомлень, наприклад через інший поштовий сервіс або API, бажано створити нову реалізацію IEmailService без зміни класів, які працюють із цим інтерфейсом.

Під час додавання нових типів дій після спрацювання правил необхідно оновити модель ProcessingAction, логіку виконання дій у сервісах обробки вкладень або API-запитів, а також відповідні елементи інтерфейсу користувача. Нові дії повинні фіксувати результат виконання в журналі, щоб оператор міг контролювати їхню роботу.

Перед передачею оновленої версії програмного забезпечення в експлуатацію програміст повинен виконати перевірку основних функцій: запуск застосунку, підключення до бази даних, додавання поштового облікового запису, перевірка підключення до IMAP-сервера, створення правила, налаштування дії, ручний запуск перевірки пошти, збереження вкладення, запис результатів у журнал і відображення даних в інтерфейсі користувача.

Інструкція оператору

Оператор або адміністратор відповідає за початкове налаштування програмного забезпечення та контроль його роботи. Перед використанням застосунку оператор повинен мати дані для підключення до поштової скриньки, доступ до бази даних, шлях до папки для збереження вкладень і, за потреби, параметри зовнішнього API.

Після запуску застосунку оператору необхідно перейти до розділу

керування поштовими обліковими записами та створити новий обліковий запис. Для цього потрібно вказати назву поштової скриньки, адресу електронної пошти, IMAP-сервер, порт, логін, пароль або пароль застосунку, папку для моніторингу та інтервал автоматичної перевірки. Після введення параметрів потрібно виконати тестове підключення. Якщо підключення успішне, обліковий запис можна зберегти та активувати.

На рисунку Г.2 наведено приклад вікна налаштування поштового облікового запису. Оператор у цьому вікні задає назву поштової скриньки, адресу електронної пошти, IMAP-сервер, порт, логін, пароль або пароль застосунку, папку для моніторингу та інтервал автоматичної перевірки. Після введення параметрів необхідно виконати перевірку підключення, щоб переконатися в правильності налаштувань і доступності поштового сервера.

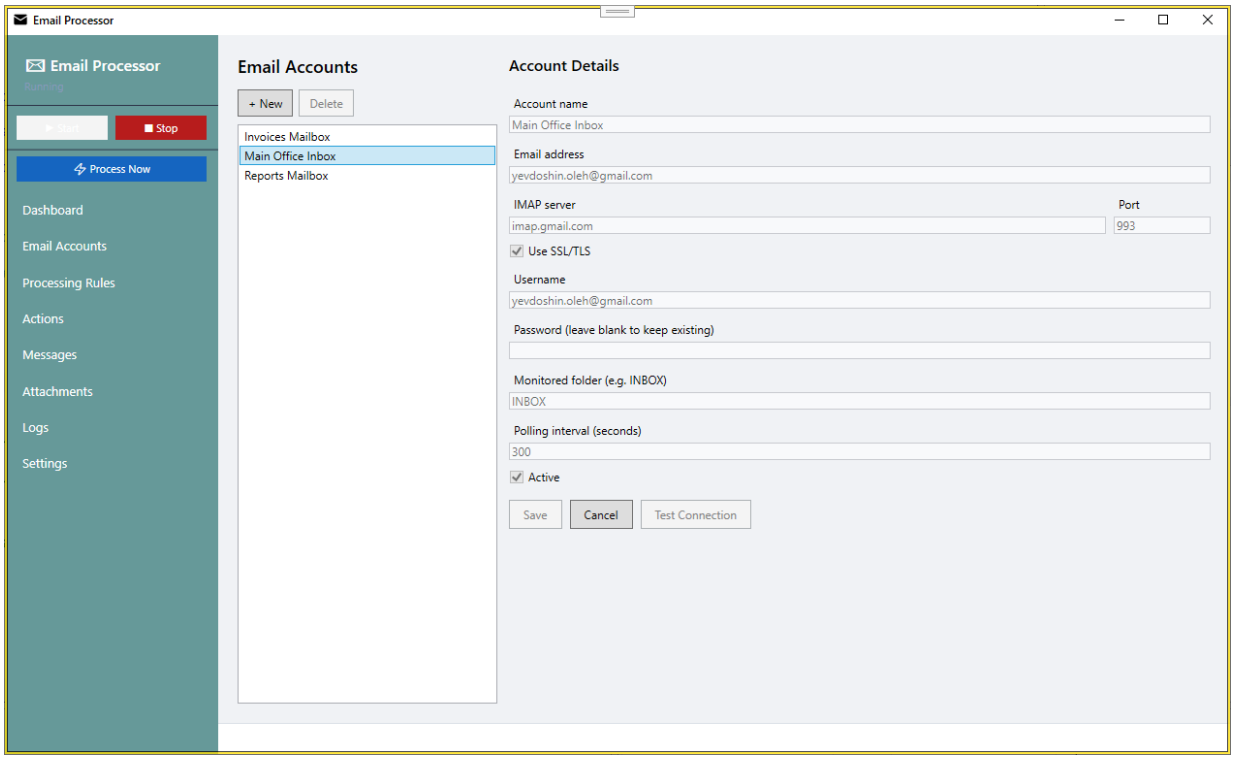


Рисунок Г.2 – Налаштування поштового облікового запису

У разі помилки підключення оператору необхідно перевірити правильність адреси IMAP-сервера, порту, логіна, пароля, наявність доступу до мережі та дозвіл поштового сервісу на використання зовнішніх застосунків.

Якщо поштовий сервіс вимагає окремий пароль застосунку, необхідно створити його в налаштуваннях поштового акаунта та використати замість основного пароля.

Після налаштування поштової скриньки оператор створює правила обробки повідомлень. Для цього необхідно відкрити розділ правил і додати нове правило. У правилі вказується назва, ознака активності, пріоритет та умови спрацювання. Умови можуть стосуватися відправника, теми листа, тексту повідомлення, дати надсилання, наявності вкладень, розширення файлів або частини назви вкладення.

Під час створення правил оператору рекомендується задавати пріоритети таким чином, щоб більш конкретні правила перевірялися раніше, а загальні — пізніше. Наприклад, правило для листів від конкретного постачальника з PDF-рахунками повинно мати вищий пріоритет, ніж загальне правило для всіх повідомлень із вкладеннями. Це дозволяє уникнути неправильного вибору правила для обробки листа.

Після створення правила оператор повинен налаштувати дії, які виконуватимуться після його спрацювання. Для кожного правила можна створити одну або декілька дій. Якщо потрібно зберігати вкладення, оператор задає цільову папку. Якщо потрібно передавати дані до зовнішнього API, оператор указує адресу API, HTTP-метод, заголовки, токен авторизації та спосіб передавання даних. Якщо потрібно переміщувати листи після обробки, оператор зазначає папку поштової скриньки, до якої має бути переміщено повідомлення.

Після налаштування поштових облікових записів, правил і дій оператор може виконати ручну перевірку пошти. Це доцільно зробити перед увімкненням постійного фонового режиму, щоб переконатися, що система правильно отримує листи, знаходить відповідні правила, обробляє вкладення та записує результати в журнал. Якщо під час ручної перевірки виникають помилки, необхідно переглянути журнал обробки та виправити налаштування.

Для запуску автоматичної роботи оператор вмикає фоновий сервіс. Після цього система самостійно перевіряє активні поштові скриньки відповідно до

заданих інтервалів. Оператор повинен періодично контролювати головну панель і журнал обробки, щоб переконатися у відсутності помилок і своєчасному виконанні операцій.

У разі виникнення помилки оператор повинен відкрити журнал обробки та переглянути запис із відповідним статусом. У журналі вказуються дата й час події, поштова скринька, відправник, тема листа, назва правила, назва вкладення, виконана дія, статус і текст помилки. На основі цієї інформації оператор може визначити причину проблеми: неправильні параметри пошти, недоступність API, відсутність доступу до папки, заборонене розширення файлу або помилку бази даних.

За необхідності оператор може деактивувати поштовий обліковий запис або окреме правило. Це корисно під час технічного обслуговування, тимчасового припинення обробки певної скриньки або перевірки нових налаштувань. Перед внесенням значних змін до правил рекомендується тимчасово зупинити фоновий сервіс, щоб уникнути обробки повідомлень за неповними або некоректними параметрами.

Інструкція користувачеві

Користувач працює із застосунком через графічний інтерфейс і контролює результати автоматичної обробки електронної пошти. Для початку роботи необхідно запустити програмне забезпечення.

На рисунку Г.3 показано головне вікно програмного забезпечення, яке використовується для контролю загального стану системи. У цьому вікні користувач може переглянути зведену інформацію про кількість оброблених повідомлень, успішні та помилкові операції, стан фонового сервісу та останні записи журналу. Головне вікно дає змогу швидко оцінити, чи працює система коректно, без переходу до детального журналу обробки.

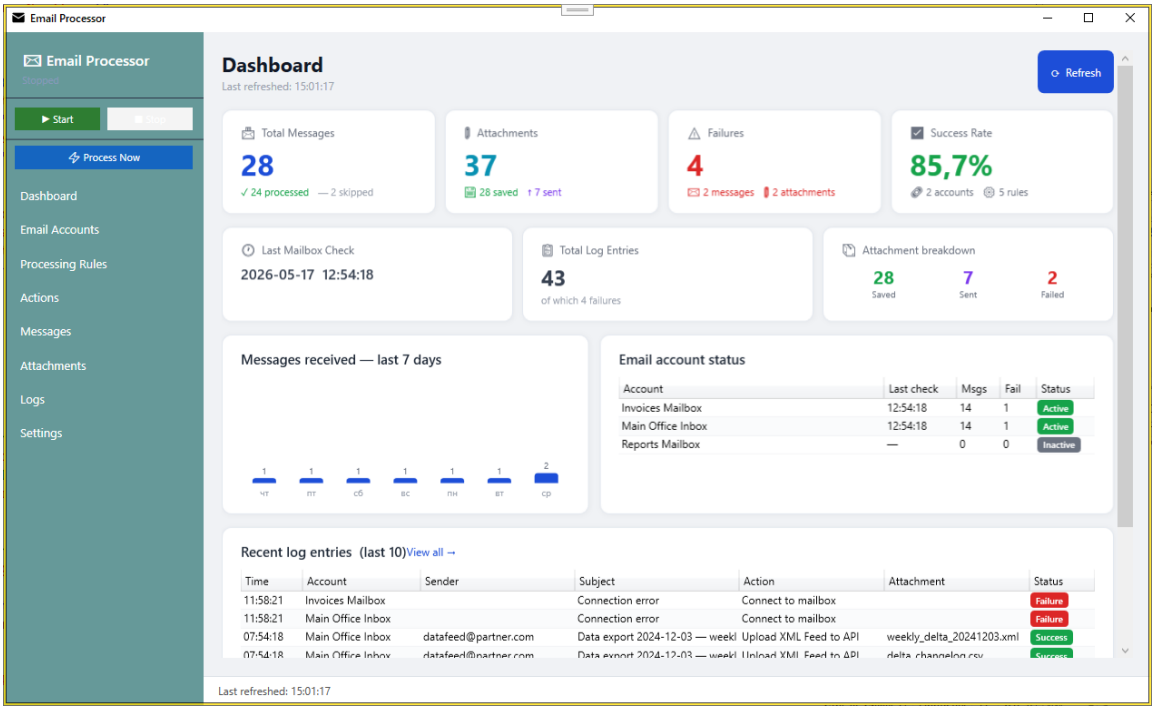


Рисунок Г.3 – Головне вікно програмного забезпечення

Якщо користувачу необхідно переглянути налаштовані поштові скриньки, він переходить до розділу Email accounts. У цьому розділі відображається список поштових облікових записів, їхній стан, адреса електронної пошти, інтервал перевірки та час останньої обробки. За наявності відповідних прав користувач може додати новий обліковий запис, змінити параметри наявного або виконати перевірку підключення.

Для ручного запуску обробки користувач натискає кнопку перевірки пошти. Після цього система запускає обробку активних поштових облікових записів без очікування чергового автоматичного запуску. Після завершення перевірки результати відображаються на головній панелі та в журналі обробки.

Якщо користувач хоче переглянути повідомлення, які були отримані системою, він відкриває розділ оброблених повідомлень. У цьому розділі можна побачити тему листа, відправника, дату отримання, поштовий обліковий запис і статус обробки. Якщо повідомлення було оброблене з помилкою, користувач може переглянути текст помилки або перейти до відповідного запису журналу.

Для перегляду вкладених файлів користувач відкриває розділ Attachments. У ньому відображаються назви вкладень, розширення файлів, розмір, шлях

збереження, статус обробки та текст помилки за наявності. Якщо вкладення було успішно збережене, користувач може знайти його у відповідній локальній або мережевій папці, зазначеній у налаштуваннях дії.

Для контролю роботи системи користувач використовує журнал обробки. На рисунку Г.4 наведено приклад вікна журналу обробки. У журналі користувач може переглядати результати отримання та обробки повідомлень, збереження вкладень, виконання API-запитів і виникнення помилок. Для зручності аналізу записи журналу можуть фільтруватися за датою, поштовим обліковим записом, відправником, темою, правилом або статусом. Це дозволяє швидко знайти інформацію про конкретний лист або визначити причину помилки.

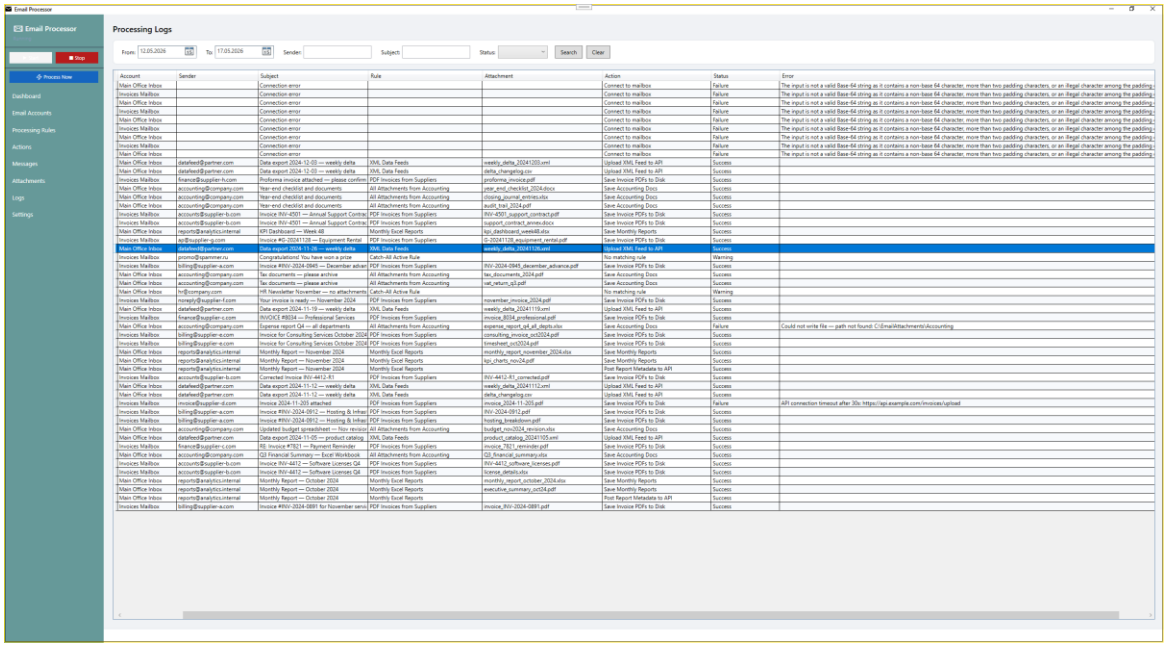


Рисунок Г.4 – Перегляд журналу обробки повідомлень

Якщо в журналі з’явився запис зі статусом помилки, користувачу необхідно переглянути його деталі. Якщо помилка пов’язана з неправильними налаштуваннями поштової скриньки, правила або дії, потрібно звернутися до оператора або адміністратора. Якщо помилка пов’язана з недоступністю зовнішнього API або файлової папки, необхідно перевірити доступність відповідного ресурсу. Якщо причина помилки незрозуміла, потрібно передати інформацію програмісту або відповідальній особі з технічного супроводу.

Під час роботи з програмним забезпеченням користувачу не потрібно вручну відкривати кожен лист або завантажувати вкладення з поштової скриньки. Основні операції виконуються автоматично відповідно до налаштованих правил. Користувач повинен лише контролювати результати роботи, переглядати журнал і за потреби повідомляти про помилки відповідальну особу.

Для завершення роботи із застосунком користувач може зупинити фоновий сервіс, якщо це передбачено його правами, після чого закрити програму стандартним способом. Якщо фоновий режим не зупинено, перед закриттям застосунку необхідно переконатися, що в цей момент не виконується обробка повідомлень. Це дозволяє уникнути переривання поточного циклу обробки пошти.

Дотримання наведених інструкцій дозволяє правильно встановлювати, налаштовувати, супроводжувати та використовувати програмне забезпечення для автоматичної обробки повідомлень електронної пошти. Програміст відповідає за технічну підтримку й розвиток системи, оператор — за її налаштування та контроль, а користувач — за щоденну роботу з інтерфейсом і перегляд результатів обробки.

ДОДАТОК Д. Програма та методика випробувань програмного забезпечення

Об’єктом випробувань є настільний WPF-застосунок, призначений для підключення до поштових скриньок за протоколом IMAP, отримання нових повідомлень, перевірки листів за налаштованими правилами, обробки вкладень, виконання дій після спрацювання правил, взаємодії із зовнішніми API та ведення журналу результатів обробки.

Метою випробувань є встановлення того, що програмне забезпечення:

– запускається в середовищі Windows;

– забезпечує підключення до бази даних Microsoft SQL Server;

– дозволяє створювати та редагувати поштові облікові записи;

– виконує перевірку підключення до поштової скриньки;

– отримує нові повідомлення з поштового сервера за протоколом IMAP;

– запобігає повторній обробці вже отриманих листів;

– дозволяє створювати правила обробки повідомлень;

– коректно визначає повідомлення, які відповідають умовам правил;

– виконує налаштовані дії після спрацювання правил;

– обробляє вкладення та зберігає інформацію про них;

– зберігає вкладені файли у визначену папку;

– передає вкладення або метадані повідомлення до зовнішнього API;

– записує результати роботи до журналу;

– відображає повідомлення, вкладення та записи журналу в інтерфейсі користувача;

– коректно обробляє помилки без аварійного завершення роботи.

Випробування проводяться в тестовому середовищі. Для перевірки роботи програмного забезпечення необхідно підготувати комп’ютер з операційною системою Windows, встановленим середовищем виконання .NET, доступом до Microsoft SQL Server, тестовою поштовою скринькою з підтримкою IMAP, тестовими електронними повідомленнями з вкладеннями різних типів, папкою

для збереження вкладень і, за потреби, тестовим API- endpoint для перевірки передавання даних.

Перед початком випробувань необхідно:

– встановити або запустити програмне забезпечення;

– перевірити наявність доступу до бази даних;

– створити тестову базу даних або підготувати порожні таблиці;

– підготувати тестову поштову скриньку;

– увімкнути доступ до поштової скриньки за протоколом IMAP;

– створити пароль застосунку, якщо це необхідно для поштового сервісу;

– підготувати тестові листи з вкладеннями;

– створити папку для збереження вкладених файлів;

– підготувати тестовий API або імітацію зовнішнього сервісу;

– переконатися, що застосунок має права на запис у вибрану папку.

Випробування виконуються шляхом послідовної перевірки основних функціональних можливостей програмного забезпечення. Для кожного тесту визначаються умови виконання, вхідні дані, порядок дій, очікуваний результат і фактичний результат (табл. Д.1 – Д.25). Якщо фактичний результат збігається з очікуваним, тест вважається успішним. Якщо результат відрізняється, необхідно зафіксувати помилку, умови її виникнення та повідомлення, яке було відображене системою або записане до журналу.

Таблиця Д.1 – Перевірка запуску програмного забезпечення

Параметр	**Опис**
Мета перевірки	Переконатися, що застосунок запускається без помилок
Початкові умови	Програмне забезпечення встановлено або зібрано з вихідного коду
Вхідні дані	Відсутні
Порядок виконання	Запустити застосунок стандартним способом
Очікуваний результат	Відкривається головне вікно програмного забезпечення, аварійне завершення не відбувається
Критерій успішності	Головне вікно відображається коректно, користувач може перейти до основних розділів

Таблиця Д.2 – Перевірка підключення до бази даних

Параметр	Опис
Мета перевірки	Переконатися, що застосунок має доступ до бази даних
Початкові умови	Налаштовано рядок підключення до Microsoft SQL Server
Вхідні дані	Параметри підключення до бази даних
Порядок виконання	Запустити застосунок і відкрити розділ, який використовує дані з бази
Очікуваний результат	Дані зчитуються або порожній список відображається без помилки
Критерій успішності	Помилки підключення до бази даних не виникають

Таблиця Д.3 – Перевірка додавання поштового облікового запису

Параметр	Опис
Мета перевірки	Перевірити створення нового поштового облікового запису
Початкові умови	Застосунок запущено, база даних доступна
Вхідні дані	Назва скриньки, email-адреса, IMAP-сервер, порт, логін, пароль, папка для моніторингу, інтервал перевірки
Порядок виконання	Відкрити розділ поштових облікових записів, додати новий запис і зберегти його
Очікуваний результат	Новий обліковий запис з’являється у списку поштових скриньок
Критерій успішності	Дані збережено в базі, пароль не відображається у відкритому вигляді

Таблиця Д.4 – Перевірка тестового підключення до поштової скриньки

Параметр	Опис
Мета перевірки	Переконатися, що застосунок може підключитися до поштового сервера
Початкові умови	Додано поштовий обліковий запис із коректними параметрами
Вхідні дані	Параметри IMAP-сервера, логін і пароль
Порядок виконання	Натиснути кнопку перевірки підключення
Очікуваний результат	Система повідомляє про успішне підключення
Критерій успішності	Підключення встановлено, помилка автентифікації не виникає

Таблиця Д.5 – Перевірка обробки неправильних параметрів пошти

Параметр	Опис
Мета перевірки	Перевірити реакцію системи на неправильні облікові дані
Початкові умови	Створено поштовий обліковий запис
Вхідні дані	Неправильний пароль або неправильний IMAP-сервер
Порядок виконання	Змінити параметри облікового запису на некоректні та виконати перевірку підключення
Очікуваний результат	Система відображає повідомлення про помилку підключення
Критерій успішності	Застосунок не завершує роботу аварійно, помилка фіксується або відображається користувачу

Таблиця Д.6 – Перевірка створення правила обробки

Параметр	Опис
Мета перевірки	Перевірити можливість створення правила обробки повідомлень
Початкові умови	Застосунок запущено, база даних доступна
Вхідні дані	Назва правила, активність, пріоритет, умови щодо відправника, теми, вкладень або розширень
Порядок виконання	Відкрити розділ правил, створити нове правило та зберегти його
Очікуваний результат	Правило відображається у списку правил
Критерій успішності	Умови правила зберігаються та доступні для подальшого редагування

Таблиця Д.7 – Перевірка створення дії після спрацювання правила

Параметр	Опис
Мета перевірки	Перевірити налаштування дії, яка виконується після спрацювання правила
Початкові умови	Створено правило обробки
Вхідні дані	Тип дії, назва дії, цільова папка або API-адреса
Порядок виконання	Додати до правила дію збереження вкладення або передавання даних до API
Очікуваний результат	Дія зберігається і відображається як пов’язана з правилом
Критерій успішності	Під час спрацювання правила система може виконати налаштовану дію

Таблиця Д.8 – Перевірка ручного запуску обробки пошти

Параметр	Опис
Мета перевірки	Перевірити можливість ручного запуску обробки активних поштових скриньок
Початкові умови	Налаштовано активну поштову скриньку, правило та дію
Вхідні дані	Тестовий лист у поштовій скриньці
Порядок виконання	Натиснути кнопку ручної перевірки пошти
Очікуваний результат	Система підключається до поштової скриньки, отримує лист і запускає його обробку
Критерій успішності	Результат обробки відображається в журналі та списку повідомлень

Таблиця Д.9 – Перевірка отримання нового повідомлення

Параметр	Опис
1	**2**
Мета перевірки	Перевірити, що система отримує нові листи з поштової скриньки
Початкові умови	У тестовій поштовій скриньці є нове повідомлення
Вхідні дані	Тестове електронне повідомлення
Порядок виконання	Запустити обробку пошти вручну або через фоновий сервіс

Кінець таблиці Д.9

1	2
Очікуваний результат	Повідомлення зберігається в таблиці EmailMessages
Критерій успішності	У списку повідомлень відображається тема, відправник, дата та статус обробки

Таблиця Д.10 – Перевірка запобігання повторній обробці повідомлення

Параметр	Опис
Мета перевірки	Переконатися, що одне й те саме повідомлення не обробляється повторно
Початкові умови	Повідомлення вже було отримане та збережене системою
Вхідні дані	Те саме повідомлення в поштовій скриньці
Порядок виконання	Повторно запустити перевірку пошти
Очікуваний результат	Повідомлення не обробляється повторно
Критерій успішності	У базі даних не створюється дубль запису з однаковими AccountId та ImapUid

Таблиця Д.11 – Перевірка спрацювання правила за темою повідомлення

Параметр	Опис
Мета перевірки	Перевірити, що правило коректно визначає повідомлення за темою
Початкові умови	Створено активне правило з умовою SubjectContains
Вхідні дані	Лист із темою, що містить заданий фрагмент
Порядок виконання	Запустити обробку пошти
Очікуваний результат	Повідомлення відповідає правилу та передається на обробку
Критерій успішності	У журналі фіксується спрацювання відповідного правила

Таблиця Д.12 – Перевірка спрацювання правила за відправником

Параметр	Опис
Мета перевірки	Перевірити фільтрацію повідомлень за адресою відправника
Початкові умови	Створено правило з умовою SenderContains
Вхідні дані	Лист від відправника, адреса якого відповідає умові
Порядок виконання	Запустити обробку пошти
Очікуваний результат	Повідомлення розпізнається як таке, що відповідає правилу
Критерій успішності	Система виконує дії, пов’язані з правилом

Таблиця Д.13 – Перевірка повідомлення, що не відповідає правилам

Параметр	Опис
1	2
Мета перевірки	Перевірити поведінку системи для повідомлень, які не відповідають жодному правилу
Початкові умови	У системі немає правила, яке відповідає тестовому повідомленню
Вхідні дані	Лист із довільною темою та відправником
Порядок виконання	Запустити обробку пошти

Кінець таблиці Д.13

1	2
Очікуваний результат	Повідомлення позначається як пропущене
Критерій успішності	Статус повідомлення встановлено як Skipped, у журналі створено відповідний запис

Таблиця Д.14 – Перевірка збереження вкладення у папку

Параметр	Опис
Мета перевірки	Перевірити збереження вкладеного файлу у файлову систему
Початкові умови	Створено правило та дію SaveToFolder
Вхідні дані	Лист із вкладенням дозволеного типу
Порядок виконання	Запустити обробку пошти
Очікуваний результат	Вкладення зберігається у визначену папку
Критерій успішності	Файл існує в цільовій папці, запис про вкладення створено в базі даних

Таблиця Д.15 – Перевірка формування унікального імені файлу

Параметр	Опис
Мета перевірки	Переконатися, що система не перезаписує файл з однаковою назвою
Початкові умови	У цільовій папці вже існує файл із такою самою назвою
Вхідні дані	Лист із вкладенням, назва якого збігається з наявним файлом
Порядок виконання	Запустити обробку пошти
Очікуваний результат	Новий файл зберігається з унікальною назвою
Критерій успішності	Початковий файл не перезаписано, новий файл збережено окремо

Таблиця Д.16 – Перевірка блокування небезпечних розширень

Параметр	Опис
Мета перевірки	Переконатися, що потенційно небезпечні вкладення не обробляються
Початкові умови	Створено правило, яке обробляє повідомлення з вкладеннями
Вхідні дані	Лист із вкладенням типу .exe, .bat або .ps1
Порядок виконання	Запустити обробку пошти
Очікуваний результат	Вкладення не зберігається та не передається до API
Критерій успішності	У журналі створено запис про пропуск або помилку обробки небезпечного вкладення

Таблиця Д.17 – Перевірка передавання вкладення до зовнішнього API

Параметр	Опис
1	2
Мета перевірки	Перевірити виконання API-дії для вкладеного файлу
Початкові умови	Створено дію SendAttachmentToApi з тестовою API-адресою

Кінець таблиці Д.17

1	2
Вхідні дані	Лист із вкладенням дозволеного типу
Порядок виконання	Запустити обробку пошти
Очікуваний результат	Система формує HTTP-запит і передає вкладення до API
Критерій успішності	API-запит завершується успішно, у журналі створено запис зі статусом Success

Таблиця Д.18 – Перевірка передавання метаданих повідомлення до API

Параметр	Опис
Мета перевірки	Перевірити передавання службових даних повідомлення у форматі JSON
Початкові умови	Створено дію SendMetadataToApi
Вхідні дані	Лист, який відповідає правилу
Порядок виконання	Запустити обробку пошти
Очікуваний результат	Система надсилає до API JSON із метаданими повідомлення
Критерій успішності	API отримує дані, у журналі створено запис про успішне виконання дії

Таблиця Д.19 – Перевірка обробки помилки зовнішнього API

Параметр	Опис
Мета перевірки	Перевірити реакцію системи на недоступність або помилку зовнішнього API
Початкові умови	Створено API-дію з некоректною або недоступною адресою
Вхідні дані	Лист, який відповідає правилу
Порядок виконання	Запустити обробку пошти
Очікуваний результат	Система фіксує помилку API, але не завершує роботу аварійно
Критерій успішності	У журналі створено запис зі статусом Failure і текстом помилки

Таблиця Д.20 – Перевірка журналювання результатів обробки

Параметр	Опис
Мета перевірки	Переконатися, що система створює записи журналу для виконаних операцій
Початкові умови	Виконано обробку хоча б одного повідомлення
Вхідні дані	Результати обробки повідомлення
Порядок виконання	Відкрити розділ журналу обробки
Очікуваний результат	У журналі відображаються записи про виконані дії
Критерій успішності	Записи містять дату, поштовий обліковий запис, тему, відправника, дію, статус і помилку за наявності

Таблиця Д.21 – Перевірка фільтрації журналу

Параметр	Опис
Мета перевірки	Перевірити пошук і фільтрацію записів журналу
Початкові умови	У журналі є записи з різними статусами та датами
Вхідні дані	Фільтр за датою, статусом, відправником або темою
Порядок виконання	Встановити значення фільтрів і застосувати пошук
Очікуваний результат	Відображаються тільки записи, що відповідають заданим умовам
Критерій успішності	Результати фільтрації відповідають обраним параметрам

Таблиця Д.22 – Перевірка автоматичної роботи фонового сервісу

Параметр	Опис
Мета перевірки	Переконатися, що система автоматично перевіряє поштові скриньки
Початкові умови	Налаштовано активну поштову скриньку з інтервалом перевірки
Вхідні дані	Новий лист у поштовій скриньці
Порядок виконання	Запустити фоновий сервіс і дочекатися настання інтервалу перевірки
Очікуваний результат	Система автоматично отримує й обробляє повідомлення
Критерій успішності	Результат обробки з’являється без ручного запуску перевірки

Таблиця Д.23 – Перевірка зупинки фонового сервісу

Параметр	Опис
Мета перевірки	Перевірити можливість зупинити автоматичне опитування пошти
Початкові умови	Фоновий сервіс запущено
Вхідні дані	Команда зупинки фонового сервісу
Порядок виконання	Натиснути кнопку зупинки фонового сервісу
Очікуваний результат	Сервіс припиняє автоматичну перевірку поштових скриньок
Критерій успішності	Нові листи не обробляються до повторного запуску сервісу

Таблиця Д.24 – Перевірка відображення вкладень в інтерфейсі

Параметр	Опис
Мета перевірки	Переконатися, що інформація про вкладення доступна користувачу
Початкові умови	Оброблено повідомлення з вкладенням
Вхідні дані	Запис про вкладення в базі даних
Порядок виконання	Відкрити розділ перегляду вкладень
Очікуваний результат	У списку відображається назва, розширення, розмір, шлях збереження та статус вкладення
Критерій успішності	Дані про вкладення відповідають фактичним результатам обробки

Таблиця Д.25 – Перевірка роботи із загальними налаштуваннями

Параметр	Опис
Мета перевірки	Перевірити збереження та використання загальних параметрів застосунку
Початкові умови	Застосунок запущено, база даних доступна
Вхідні дані	Значення параметрів, наприклад максимальний розмір вкладення або стандартний інтервал перевірки
Порядок виконання	Змінити параметр у налаштуваннях і зберегти зміни
Очікуваний результат	Нове значення параметра зберігається та використовується застосунком
Критерій успішності	Після перезапуску застосунку значення параметра не втрачається

За результатами виконання випробувань було сформовано таблицю результатів, у якій наведено фактичний результат кожної перевірки та статус її проходження.

Таблиця Д.26 – Результати випробувань програмного забезпечення

№ тесту	Назва перевірки	Фактичний результат	Статус
1	**2**	**3**	**4**
1	Перевірка запуску програмного забезпечення	Головне вікно програмного забезпечення відкривається коректно, аварійне завершення не відбувається	Успішно пройдено
2	Перевірка підключення до бази даних	Застосунок успішно підключається до бази даних, дані зчитуються та відображаються без помилок	Успішно пройдено
3	Перевірка додавання поштового облікового запису	Новий поштовий обліковий запис зберігається в базі даних і відображається у списку поштових скриньок	Успішно пройдено
4	Перевірка тестового підключення до поштової скриньки	Підключення до поштової скриньки виконується успішно, повідомлення про помилку не з’являється	Успішно пройдено
5	Перевірка створення правила обробки	Створене правило зберігається в базі даних і відображається у списку правил обробки	Успішно пройдено
6	Перевірка створення дії після спрацювання правила	Дія зберігається та коректно пов’язується з відповідним правилом обробки	Успішно пройдено
7	Перевірка ручного запуску обробки пошти	Після ручного запуску система виконує перевірку активних поштових скриньок і запускає обробку повідомлень	Успішно пройдено
8	Перевірка отримання нового повідомлення	Нове повідомлення отримується з поштової скриньки, зберігається в базі даних і відображається у списку повідомлень	Успішно пройдено

Кінець таблиці Д.26

1	2	3	4
9	Перевірка запобігання повторній обробці повідомлення	Повторний запуск перевірки не призводить до дублювання вже обробленого повідомлення	Успішно пройдено
10	Перевірка збереження вкладення у папку	Вкладений файл зберігається у визначену папку, а інформація про нього записується в базу даних	Успішно пройдено
11	Перевірка передавання даних до зовнішнього API	Система формує HTTP-запит і передає вкладення або метадані повідомлення до зовнішнього API	Успішно пройдено
12	Перевірка журналювання результатів обробки	У журналі створюються записи про виконані дії, успішні операції та помилки	Успішно пройдено
13	Перевірка фільтрації журналу	Після застосування фільтрів у журналі відображаються тільки записи, що відповідають заданим умовам	Успішно пройдено
14	Перевірка автоматичної роботи фонового сервісу	Фоновий сервіс автоматично перевіряє активні поштові скриньки відповідно до заданого інтервалу	Успішно пройдено
15	Перевірка обробки помилкових ситуацій	У разі виникнення помилок система не завершує роботу аварійно, а фіксує інформацію про помилку в журналі	Успішно пройдено

За результатами виконання програми та методики випробувань можна зробити висновок про відповідність розробленого програмного забезпечення основним функціональним вимогам. Успішне проходження наведених перевірок підтверджує, що застосунок може використовуватися для автоматичної обробки повідомлень електронної пошти, вкладених файлів, виконання дій за правилами та контролю результатів через журнал обробки.