МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА СИСТЕМИ УПРАВЛІННЯ ВЕБХУКАМИ З МОЖЛИВІСТЮ АВТОНОМНОГО РОЗГОРТАННЯ»**

Виконав: студент 4 курсу, групи 6.1212-пі-2
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
Б.К. Грачов
(ініціали та прізвище)

Керівник декан математичного факультету, професор, д.т.н. Гоменюк С.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Грачову Богдану Костянтиновичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка системи управління вебхуками з можливістю автономного розгортання

керівник роботи Гоменюк Сергій Іванович, д.т.н., професор
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1741-с

2. Строк подання студентом роботи 01.06.2026 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області та програмних засобів.
2. Проєктування системи.
3. Програмна реалізація системи.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 09.12.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	19.12.2025	
2.	Збір вихідних даних.	19.01.2026	
3.	Обробка методичних та теоретичних	17.02.2026	
	джерел.		
4.	Розробка першого та другого розділу.	16.03.2026	
5.	Розробка третього розділу.	12.05.2026	
6.	Оформлення та нормоконтроль	25.05.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	17.06.2026	

Студент _____ Б.К. Грачов
(підпис) (ініціали та прізвище)

Керівник роботи _____ С.І. Гоменюк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка системи управління вебхуками з можливістю автономного розгортання»: 63 с., 41 рис., 4 табл., 10 джерел, 1 додаток.

АСИНХРОННА ОБРОБКА, ВЕБХУКИ, ДОСТАВКА ПОВІДОМЛЕНЬ, КОНТЕЙНЕРИЗАЦІЯ, ПОВТОРНІ СПРОБИ ДОСТАВКИ, САМОСТІЙНЕ РОЗГОРТАННЯ, СПОСТЕРЕЖУВАНІСТЬ СИСТЕМИ, ЧЕРГИ ПОВІДОМЛЕНЬ, HMAC-ПІДПИС, REST API.

Об’єкт дослідження – процес розробки системи управління вебхуками з можливістю автономного розгортання.

Мета роботи: розробка застосунку для керування вебхуками з можливістю автономного розгортання.

Методи дослідження – методи об’єктно-орієнтованого програмування, методи проєктування REST API та версіонування інтерфейсів, методи асинхронної обробки повідомлень із використанням брокера черг, методи забезпечення надійної доставки на основі механізмів повторних спроб та експоненційного відступу.

У роботі проведено аналіз існуючих рішень у сфері управління вебхуками – як хмарних сервісів (Svix, Hookdeck, Convoy), так і самостійно розгорнуваних альтернатив. Встановлено, що більшість комерційних платформ не надають повного контролю над інфраструктурою та накладають обмеження на обсяг доставлень. На основі аналізу спроєктовано систему з модульною монолітною архітектурою, розподіленою на чотири ізольовані обмежені контексти з власними базами даних PostgreSQL.

Для гарантованого доставлення реалізовано патерн Transactional Outbox із чергою RabbitMQ та патерн Circuit Breaker із персистованим станом.

SUMMARY

Bachelor's qualifying paper "Development of a Webhook Management System with Autonomous Deployment Capability": 63 pages, 41 figures, 4 tables, 10 references, 1 supplement.

ASYNCHRONOUS PROCESSING, CONTAINERIZATION, DELIVERY RETRY MECHANISM, HMAC SIGNING, MESSAGE DELIVERY, MESSAGE QUEUES, REST API, SELF-HOSTED DEPLOYMENT, SYSTEM OBSERVABILITY, WEBHOOKS.

The object of the study is the process of developing a webhook management system with autonomous deployment capability.

The aim of the study is to develop an application for a webhook management system with autonomous deployment capability.

The methods of research include object-oriented programming methods, REST API design and interface versioning methods, asynchronous message processing methods using a message broker, and reliable delivery assurance methods based on retry mechanisms and exponential backoff.

The thesis presents an analysis of existing webhook management solutions, including cloud-based services (Svix, Hookdeck, Convoy) and self-hosted alternatives. It was found that most commercial platforms restrict infrastructure control and impose delivery volume limitations. Based on the analysis, a system with a modular monolith architecture divided into four isolated bounded contexts with independent PostgreSQL databases was designed and implemented.

To guarantee delivery, the Transactional Outbox pattern with a RabbitMQ queue and the Circuit Breaker pattern with persisted state were implemented.

ЗМІСТ

Завдання на кваліфікаційну роботу…….…….. 2
Реферат 4
Summary 5
Вступ 7
1 Аналіз предметної області та обґрунтування технічних рішень 9
1.1 Аналіз предметної області 9
1.2 Огляд існуючих рішень 10
1.3 Вимоги до системи 12
1.4 Вибір та обґрунтування технологічного стеку 13
2 Проєктування системи 16
2.1 Архітектура системи 16
2.2 Проєктування бази даних 19
2.3 Проєктування REST API 22
2.4 Проєктування механізму доставки повідомлень 23
2.5 Проєктування інтерфейсу користувача 26
3 Програмна реалізація системи 31
3.1 Реалізація серверної частини 31
3.2 Реалізація доменної моделі 34
3.3 Реалізація шару доступу до даних 37
3.4 Реалізація системи надійного доставлення 41
3.5 Реалізація автентифікації та авторизації 46
3.6 Реалізація обмеження частоти запитів 51
3.7 Тестування системи 55
Висновки 59
Перелік посилань 61
Додаток А Інструкція з розгортання системи Aether 62

ВСТУП

Інтеграція програмних систем між собою є невід'ємною частиною сучасної розробки програмного забезпечення. Через популярності мікросервісної архітектури виникає необхідність у надійних, масштабованих механізмах передачі подій між сервісами. Вебхуки стали стандартним такої взаємодії, дозволяючи одній системі миттєво асинхронно сповіщати іншу про настання певної події.

Незважаючи на широке застосування вебхуків, ринок інструментів для їх керування переважно представлений хмарними комерційними платформами. Такі рішення передбачають передачу даних через інфраструктуру третіх сторін, що створює обмеження у сферах безпеки, відповідності регуляторним вимогам та контролю над витратами. Відсутність зручної self-hosted альтернативи з повноцінним набором функцій – надійною доставкою, повторними спробами, підписом запитів та спостережуваністю – визначає актуальність даної роботи.

Мета – розробка застосунку для керування вебхуками з можливістю автономного розгортання.

Для досягнення поставленої мети необхідно вирішити такі задачі:

- проаналізувати наявні рішення у сфері керування та доставки вебхуків, визначити їх переваги та недоліки;
- обґрунтувати вибір технологічного стеку для реалізації серверної та клієнтської частин системи;
- спроєктувати архітектуру застосунку з урахуванням вимог до надійності та масштабованості;
- реалізувати серверну частину з підтримкою асинхронної обробки подій, механізму повторних спроб, HMAC-підпису та API-ключів;
- розробити клієнтський вебінтерфейс для керування колекціями подій, вебхуками, журналами доставки та аналітикою;

– налаштувати виробниче розгортання системи засобами контейнеризації з автоматичним забезпеченням TLS;
– провести тестування розробленої системи.

Об’єкт дослідження – процес розробки системи управління вебхуками з можливістю автономного розгортання.

Предмет дослідження – методи та засоби побудови надійної системи доставки вебхуків на основі платформи .NET 10, фреймворку ASP.NET Core, брокера повідомлень RabbitMQ, реляційної бази даних PostgreSQL та інструментів контейнеризації Docker і Caddy.

Методи дослідження – методи об’єктно-орієнтованого програмування, методи проєктування REST API та версіонування інтерфейсів, методи асинхронної обробки повідомлень із використанням брокера черг, методи забезпечення надійної доставки на основі механізмів повторних спроб та експоненційного відступу.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, переліку посилань та 1 додатка. Загальний обсяг роботи становить 63 сторінки. У переліку посилань 10 використаних джерел.

Перший розділ містить аналіз предметної області та огляд існуючих рішень для керування вебхуками, а також обґрунтування вибору технологій. У другому розділі представлено проєктування системи: структуру бази даних, архітектуру компонентів та опис програмного інтерфейсу. У третьому розділі описано практичну реалізацію серверної та клієнтської частин, механізми асинхронної доставки, систему автентифікації та виробниче розгортання застосунку.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ ТЕХНІЧНИХ РІШЕНЬ

1.1 Аналіз предметної області

Сучасні програмні системи рідко функціонують як ізольовані застосунки – вони взаємодіють між собою, обмінюються даними та реагують на події, що відбуваються в суміжних сервісах. Одним із ключових механізмів такої взаємодії є вебхуки – HTTP-запити, що надсилаються однією системою до іншої у відповідь на настання певної події. На відміну від традиційного підходу, за якого клієнт самостійно опитує сервер для отримання оновлень, вебхуки реалізують модель «push»: сервер сам ініціює передачу даних у момент, коли подія відбувається.

Вебхуки широко застосовуються у платіжних системах, CRM-платформах, системах контролю версій, сервісах автоматизації та інших галузях, де важлива своєчасна реакція на зовнішні події. Наприклад, платіжний провайдер може надсилати вебхук після підтвердження транзакції, а система керування репозиторієм – після злиття гілки коду.

Для розуміння переваг вебхуків доцільно порівняти їх з альтернативними підходами до міжсервісної комунікації. Polling – найпростіший підхід, за якого клієнт з певною періодичністю надсилає запити до сервера для перевірки наявності нових даних. Головним недоліком є неефективне використання ресурсів: більшість запитів повертає порожній результат. WebSocket забезпечує двосторонній канал зв’язку між клієнтом і сервером у реальному часі, проте вимагає постійного з’єднання та є надлишковим для сценаріїв односторонньої передачі подій. Server-Sent Events (SSE) дозволяють серверу передавати потік подій клієнту через одне HTTP-з’єднання, однак підходять переважно для браузерних клієнтів і не вирішують задачу сповіщення зовнішніх серверних систем. Вебхуки позбавлені цих

недоліків: вони не вимагають постійного з’єднання, легко інтегруються з будь-яким HTTP-сервером і масштабуються незалежно від кількості підписників.

Типовий життєвий цикл доставки вебхука охоплює кілька етапів. Спочатку в системі-відправнику відбувається певна подія – наприклад, зміна статусу замовлення. Система формує HTTP POST-запит із JSON-тілом, що описує подію, і надсилає його на зареєстровану URL-адресу отримувача. Отримувач обробляє запит і повертає HTTP-відповідь. У разі недоступності отримувача або отримання відповіді з кодом помилки система повинна повторити спробу доставки через певний інтервал часу.

Надійна реалізація цього процесу вимагає вирішення ряду технічних задач: забезпечення гарантованої доставки навіть при тимчасових збоях, захисту цілісності запитів за допомогою криптографічного підпису, журналювання кожної спроби доставки та надання інструментів для моніторингу й повторного відтворення невдалих запитів. Саме ці вимоги визначають функціональне ядро розроблюваної системи.

1.2 Огляд існуючих рішень

На ринку існує ряд платформ для керування та доставки вебхуків, які умовно можна поділити на дві категорії: хмарні комерційні сервіси та рішення з відкритим кодом для самостійного розгортання.

Серед хмарних платформ найбільш відомими є Svix, Hookdeck та Convoy. Svix – це спеціалізований сервіс для відправки вебхуків, орієнтований насамперед на розробників SaaS-продуктів. Він пропонує зручний API, повторні спроби доставки та портал для розробників, однак є повністю хмарним рішенням і вимагає передачі даних через інфраструктуру третіх сторін. Hookdeck позиціонується як платформа для отримання, моніторингу та маршрутизації вхідних вебхуків. Сервіс надає широкі можливості для налагодження та повторного відтворення запитів, проте також не передбачає розгортання на власній інфраструктурі. Convoy є рішенням з відкритим кодом,

що підтримує self-hosted розгортання та надає базовий набір функцій для доставки вебхуків, однак відрізняється складністю налаштування та обмеженим набором вбудованих інструментів для аналітики й спостережуваності.

Для систематизації порівняння розглянутих рішень складено таблицю 1.1.

Таблиця 1.1 – Порівняння критеріїв існуючих систем

Критерій	**Svix**	**Hookdeck**	**Convoy**	**Aether**
Self-hosted	–	–	+	+
Відкритий код	–	–	+	+
Повторні спроби	+	+	+	+
HMAC-підпис	+	+	+	+
Вбудована аналітика	+	+	–	+
Безкоштовне використання	Частково	Частково	+	+

Аналіз наявних рішень дозволяє виділити спільний недолік хмарних платформ – залежність від зовнішньої інфраструктури, що унеможливлює їх застосування у середовищах із підвищеними вимогами до конфіденційності або регуляторними обмеженнями на передачу даних. Існуючі self-hosted альтернативи, зокрема Convoy, вирішують цю проблему лише частково, поступаючись хмарним рішенням за рівнем зручності використання та повнотою вбудованого функціоналу.

Таким чином, існує практична потреба у self-hosted платформі, яка б поєднувала надійну асинхронну доставку вебхуків, криптографічний підпис запитів, автоматичні повторні спроби з експоненційним відступом, повноцінний журнал доставки та зручний вебінтерфейс – і при цьому могла розгортатися на власній інфраструктурі без залежності від зовнішніх сервісів. Усунення цієї прогалини є основною мотивацією для розробки системи Aether.

1.3 Вимоги до системи

Перед початком проєктування та реалізації будь-якої інформаційної системи необхідно чітко визначити вимоги, яким вона має відповідати. Вимоги прийнято поділяти на функціональні, що описують поведінку системи, та нефункціональні, що визначають якісні характеристики її роботи.

До **функціональних вимог** розроблюваної системи належать:

- *керування колекціями подій*: користувач повинен мати можливість створювати, редагувати та видаляти колекції, що є логічними групами споріднених подій;
- *керування подіями*: система має підтримувати визначення типізованих подій у межах колекцій із можливістю задання схеми корисного навантаження;
- *реєстрація вебхуків*: для кожної події користувач повинен мати змогу зареєструвати один або кілька вебхуків із зазначенням цільової URL-адреси, довільних HTTP-заголовків та алгоритму HMAC-підпису;
- *асинхронна доставка подій*: ініційовані події мають передаватися до черги повідомлень і оброблятися незалежно від основного потоку виконання застосунку;
- *автоматичні повторні спроби*: у разі невдалої доставки система повинна автоматично повторювати спробу з експоненційним збільшенням інтервалу між спробами;
- *журналювання доставки*: кожна спроба доставки має фіксуватися із зазначенням HTTP-статусу, затримки, тіла запиту та відповіді;
- *API-ключі*: система має підтримувати програмне ініціювання подій зовнішніми сервісами через автентифікацію за допомогою API-ключів;
- *аналітика*: користувач повинен мати доступ до агрегованої статистики доставок, включаючи показники успішності та середню затримку.

До **нефункціональних вимог** належать:

– *надійність*: система має забезпечувати гарантовану доставку повідомлень навіть за умов тимчасової недоступності цільових сервісів;
– *безпека*: усі запити до вебхуків мають підписуватися криптографічним підписом HMAC для забезпечення цілісності та автентичності даних; доступ до API системи має захищатися за допомогою JWT-токенів та API-ключів;
– *спостережуваність*: система повинна надавати повну інформацію про стан кожної доставки, що дозволяє оперативно виявляти та усувати збої;
– *автономність розгортання*: система має розгортатися на власній інфраструктурі без залежності від зовнішніх хмарних сервісів, з автоматичним отриманням TLS-сертифікатів;
– *зручність використання*: вебінтерфейс має забезпечувати інтуїтивно зрозуміле керування всіма компонентами системи без необхідності безпосередньої роботи з API.

Визначений перелік вимог є основою для подальшого проєктування архітектури системи та вибору відповідного технологічного стеку.

1.4 Вибір та обґрунтування технологічного стеку

Вибір технологій для реалізації системи є одним із ключових архітектурних рішень, що безпосередньо впливає на продуктивність, надійність та зручність подальшої підтримки застосунку. Вибір здійснювався на основі вимог, визначених у попередньому підрозділі, з урахуванням зрілості технологій, наявності активної спільноти та відповідності поставленим задачам.

Для реалізації серверної частини обрано мову програмування C# у поєднанні з платформою .NET 10. Дана платформа є кросплатформеною,

підтримує сучасні асинхронні патерни програмування та забезпечує високу продуктивність при обробці HTTP-запитів. Як вебфреймворк використано ASP.NET Core – зрілий і широко застосовуваний інструмент для побудови REST API, що надає вбудовані засоби маршрутизації, валідації, автентифікації та документування інтерфейсів. Для взаємодії з реляційною базою даних застосовано Entity Framework Core – об'єктно-реляційний mapper, що дозволяє описувати схему даних засобами мови C# та керувати міграціями бази даних у рамках єдиного проєкту.

Як система керування базами даних обрано PostgreSQL – відкриту реляційну СУБД із широкими можливостями, високою надійністю та активною підтримкою спільноти. PostgreSQL добре зарекомендувала себе у виробничих середовищах і повністю відповідає вимогам до зберігання структурованих даних системи: користувачів, колекцій, подій, вебхуків та журналів доставки.

Для реалізації асинхронної черги повідомлень обрано RabbitMQ – брокер повідомлень із підтримкою протоколу AMQP, що забезпечує надійну передачу повідомлень між компонентами системи. RabbitMQ дозволяє відокремити процес ініціювання події від її доставки, що є ключовою вимогою для забезпечення відмовостійкості та масштабованості. Вбудовані механізми підтвердження доставки повідомлень гарантують, що жодна подія не буде втрачена навіть у разі збою у процесі обробки.

Клієнтська частина застосунку реалізована у вигляді односторінкового застосунку з використанням бібліотеки React 19 та мови TypeScript. React забезпечує компонентний підхід до побудови інтерфейсу, спрощує управління станом і є одним із найпоширеніших інструментів у сучасній фронтенд-розробці. TypeScript доповнює JavaScript статичною типізацією, що суттєво знижує кількість помилок на етапі розробки та полегшує супровід коду. Як інструмент збірки використано Vite – сучасний bundler із швидким часом запуску та підтримкою гарячого оновлення модулів у режимі розробки.

Виробниче розгортання системи реалізовано засобами Docker та Docker Compose, що забезпечує ізоляцію компонентів, відтворюваність середовища

та спрощує процес оновлення застосунку. Як зворотний проксі-сервер обрано Caddy – сучасний вебсервер із вбудованою підтримкою автоматичного отримання та оновлення TLS-сертифікатів через протокол ACME. Це дозволяє налаштувати захищене HTTPS-з’єднання без ручного керування сертифікатами.

Обраний технологічний стек забезпечує відповідність усім визначеним функціональним та нефункціональним вимогам і створює надійне підґрунтя для проєктування та реалізації системи в наступних розділах роботи.

2 ПРОЄКТУВАННЯ СИСТЕМИ

Проєктування є фундаментальним етапом розробки програмного забезпечення, що передує безпосередній реалізації та визначає загальну структуру майбутньої системи. Якість прийнятих на цьому етапі рішень безпосередньо впливає на надійність, масштабованість та зручність супроводу готового продукту [1].

У межах даного розділу розглянуто архітектуру розроблюваної системи, спроєктовано структуру бази даних, описано взаємодію між компонентами застосунку та визначено структуру програмного інтерфейсу. Прийняті проєктні рішення ґрунтуються на вимогах, сформульованих у першому розділі, та обраному технологічному стеку.

2.1 Архітектура системи

Розроблювана система побудована за багаторівневою архітектурою, що передбачає чіткий розподіл відповідальності між компонентами [2, 3]. Такий підхід забезпечує незалежність рівнів один від одного, спрощує тестування окремих частин системи та дозволяє масштабувати компоненти незалежно залежно від навантаження.

Систему складають такі основні компоненти: клієнтський вебзастосунок, серверний HTTP API, брокер повідомлень, споживач черги та реляційна база даних. Взаємодія між ними організована таким чином, що жоден компонент не звертається до бази даних безпосередньо, оминаючи серверну логіку, а доставка вебхуків повністю відокремлена від обробки вхідних HTTP-запитів.

Клієнтський вебзастосунок реалізований як односторінковий застосунок і взаємодіє з системою виключно через HTTP API. Він відповідає за відображення інтерфейсу користувача та не містить жодної бізнес-логіки.

Серверний HTTP API є центральним компонентом системи. Він приймає запити від клієнта та зовнішніх сервісів, виконує автентифікацію та авторизацію, застосовує бізнес-логіку і зберігає дані до бази даних. У момент ініціювання події API формує повідомлення та публікує його до черги брокера RabbitMQ, після чого негайно повертає відповідь клієнту, не очікуючи завершення доставки.

Брокер повідомлень RabbitMQ виступає посередником між API та споживачем черги. Він забезпечує надійне зберігання повідомлень до моменту їх обробки та гарантує, що жодне повідомлення не буде втрачене навіть у разі тимчасової недоступності споживача.

Споживач черги – це окремий фоновий сервіс, що працює в межах того самого серверного застосунку. Він отримує повідомлення з черги, формує HTTP-запит до зареєстрованої URL-адреси вебхука, додає HMAC-підпис та надсилає запит до цільового сервісу. Результат кожної спроби доставки фіксується у базі даних. У разі отримання відповіді з кодом помилки або відсутності відповіді споживач повторює спробу відповідно до налаштованої стратегії експоненційного відступу.

База даних PostgreSQL забезпечує постійне зберігання всіх даних системи: облікових записів користувачів, колекцій, подій, вебхуків, API-ключів та журналів доставки. Доступ до бази даних здійснюється виключно через серверний застосунок за допомогою Entity Framework Core.

Зовнішні сервіси взаємодіють із системою через HTTP API, використовуючи API-ключі для автентифікації. Це дозволяє програмно ініціювати події безпосередньо зі сторонніх застосунків без використання вебінтерфейсу.

Далі на рисунках 2.1 та 2.2 наведено діаграму компонентів та діаграму розгортання відповідно.

Така архітектура забезпечує асинхронність доставки, відмовостійкість при тимчасових збоях цільових сервісів та незалежність між компонентами, що відповідає нефункціональним вимогам, визначеним у першому розділі.

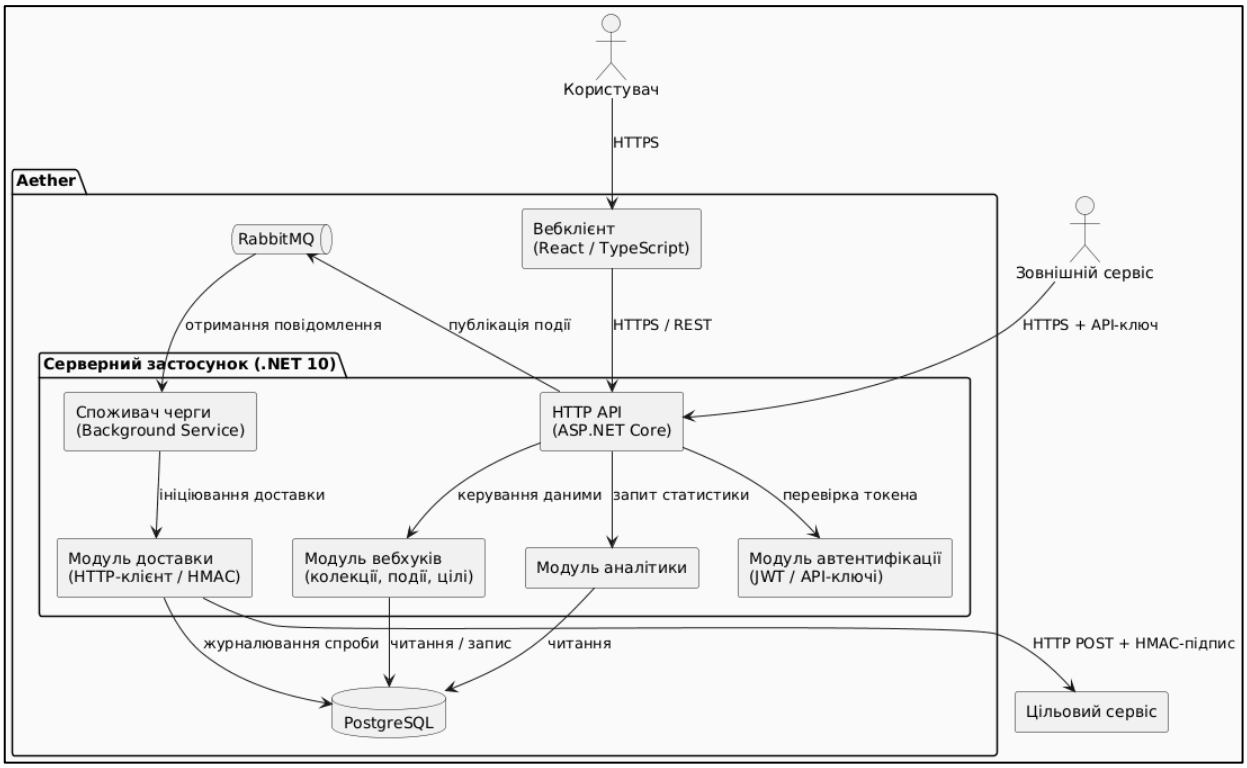


Рисунок 2.1 – Діаграма компонентів

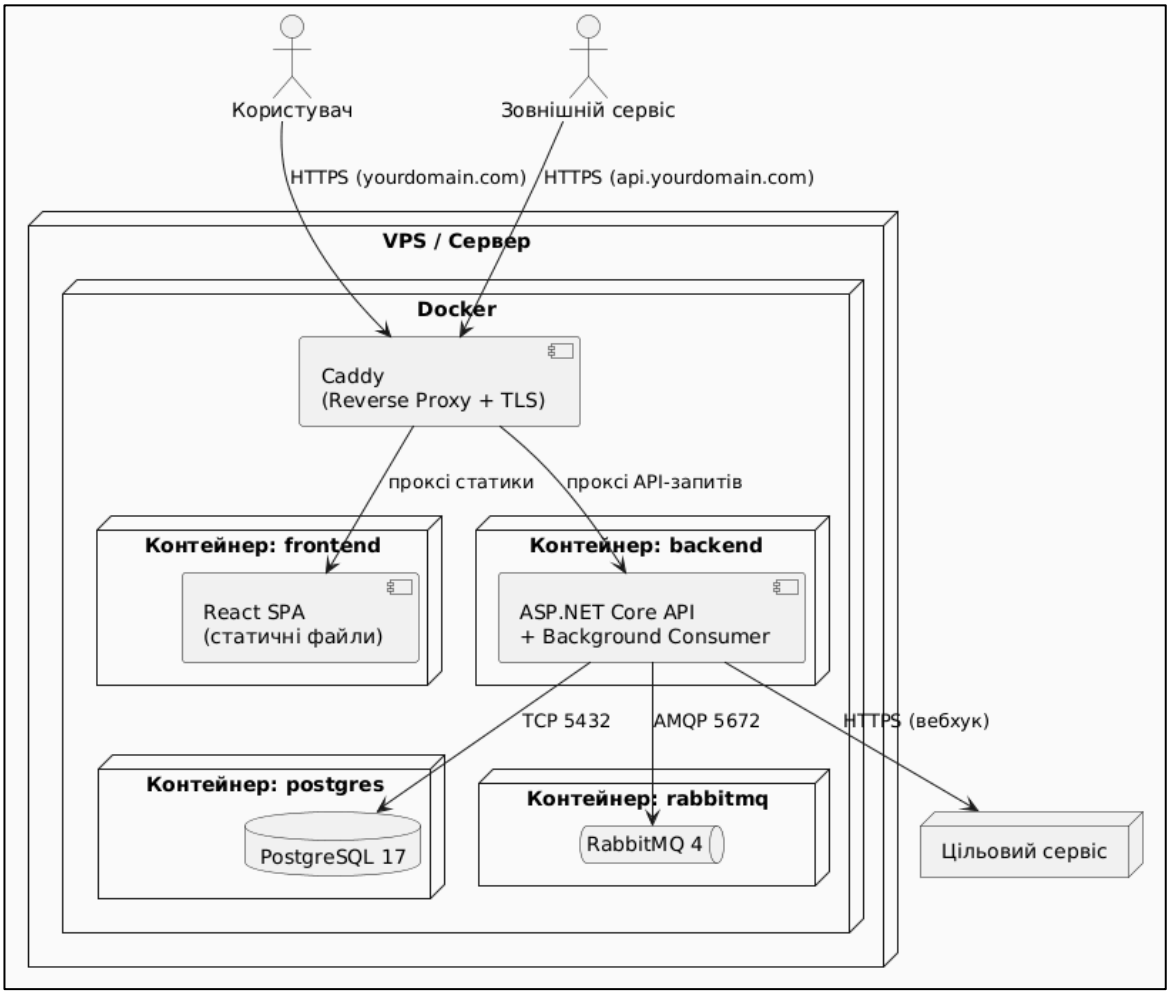


Рисунок 2.2 – Діаграма розгортання

2.2 Проєктування бази даних

Структура бази даних системи організована відповідно до модульної архітектури серверного застосунку. Кожен модуль має власний контекст бази даних, що керує лише тими сутностями, за які він відповідає. Така організація забезпечує слабку зв’язаність між модулями та спрощує їх незалежне тестування. Усього в системі визначено чотири контексти: IdentityDbContext, WebhookDbContext, DeliveryDbContext та AnalyticsDbContext.

Модуль ідентифікації оперує двома таблицями. Таблиця Users зберігає облікові записи користувачів. Кожен запис містить унікальний ідентифікатор типу UUID, унікальну електронну адресу, хеш пароля, роль, стан облікового запису, мітку безпеки та часові мітки створення й оновлення. Вбудований API-ключ зберігається як залежна сутність у межах того самого запису – хеш ключа та його останні чотири символи збережено у стовпцях ApiKeyHash та ApiKeyLastFour. Таблиця UserApiKeys зберігає іменовані API-ключі для програмного доступу. Кожен ключ пов’язаний із користувачем, має унікальний хеш, останні чотири символи для ідентифікації та прапорець відкликання IsRevoked. Унікальний індекс на хеш ключа забезпечує швидкий пошук при автентифікації запитів.

Модуль вебхуків містить п’ять таблиць, що утворюють ієрархічну структуру. Таблиця EventsCollections представляє колекції подій, що належать конкретному користувачеві. Кожна колекція має ім’я та часові мітки. При видаленні колекції всі пов’язані події видаляються каскадно. Таблиця Events зберігає типізовані події в межах колекцій. Кожна подія має ім’я, необов’язковий опис, колір у форматі HEX, прапорець активності та необов’язковий шаблон тіла запиту. Видалення події каскадно поширюється на пов’язані параметри та вебхуки. Таблиця EventParameters описує параметри схеми корисного навантаження події – ім’я, тип, ознаку обов’язковості та необов’язкове значення за замовчуванням. Таблиця Webhooks зберігає зареєстровані вебхуки для конкретних подій. Кожен вебхук містить ім’я, цільову URL-адресу, необов’язковий секрет для підпису, алгоритм підпису,

назву заголовка підпису, необов’язковий шаблон тіла, прапорець активності, максимальну кількість повторних спроб із значенням за замовчуванням 5 та стратегію відступу із значенням за замовчуванням «Exponential». Таблиця WebhookHeaders зберігає довільні HTTP-заголовки, що додаються до кожного запиту вебхука, – ключ довжиною до 128 символів і значення до 1024 символів.

Модуль доставки містить три таблиці. Таблиця DeliveryAttempts є журналом кожної спроби доставки. Запис фіксує ідентифікатор вебхука, події та користувача, необов’язкове посилання на попередню спробу RetryOf, статус, номер спроби, URL запиту, тіло запиту, необов’язкове тіло відповіді обсягом до 65 536 символів, необов’язкове повідомлення про помилку та часову мітку. Для прискорення типових запитів визначено два складені індекси: за полями UserId, IsTest, TriggeredAtUtc для вибірки журналу та за UserId, IsTest, Status для агрегованих запитів аналітики. Таблиця OutboxMessages реалізує патерн «скринька вихідних повідомлень»: перед публікацією до черги RabbitMQ повідомлення записується до цієї таблиці, що гарантує його збереження навіть у разі збою брокера [5]. Запис містить усі дані, необхідні для доставки: URL вебхука, секрет, алгоритм підпису, заголовок підпису, довільні заголовки, корисне навантаження, статус, номер спроби та час запланованої відправки. Складений індекс на Status та ScheduledAtUtc забезпечує ефективне отримання наступних повідомлень до обробки. Таблиця CircuitBreakerStates зберігає стан автоматичного вимикача для кожного вебхука – поточний стан, кількість послідовних невдалих спроб та час останнього оновлення. Унікальний індекс на WebhookId гарантує наявність лише одного запису для кожного вебхука.

Модуль аналітики містить таблицю UserAnalytics, що зберігає агреговану статистику для кожного користувача. Унікальний індекс на UserId забезпечує наявність лише одного запису на користувача.

Усі первинні ключі в системі мають тип UUID і генеруються на стороні застосунку, а не бази даних. Ідентифікатори та значення доменних об’єктів

зберігаються у базі даних у примітивних типах через конвертери значень Entity Framework Core, що дозволяє зберігати строгу типізацію на рівні доменної моделі без впливу на схему бази даних.

Далі на рисунках 2.3 та 2.4 представлено ER діаграму та діаграму модулів бази даних відповідно.

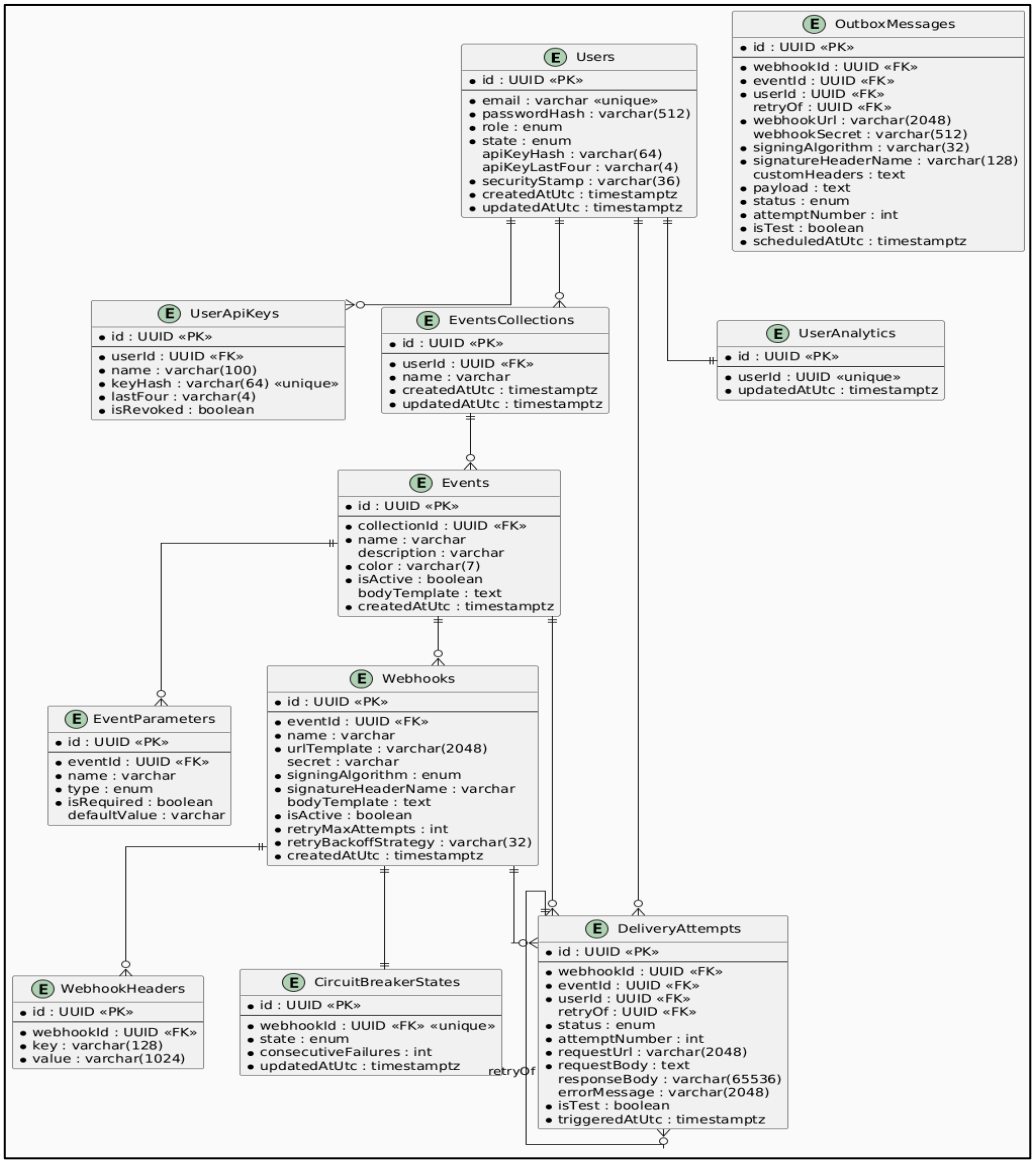


Рисунок 2.3 – ER діаграма

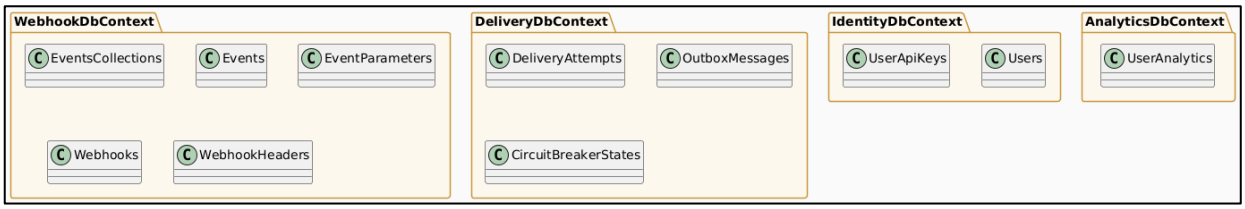


Рисунок 2.4 – Діаграма модулів бази даних

2.3 Проєктування REST API

Програмний інтерфейс системи реалізовано відповідно до принципів REST і побудовано на базі механізму Minimal API платформи ASP.NET Core. Усі маршрути об’єднано під спільним префіксом /api/v1, де v1 позначає версію інтерфейсу [6]. Версіонування забезпечує можливість введення нових версій API у майбутньому без порушення сумісності з наявними клієнтами.

API розділено на дві зони доступу. Захищена зона охоплює маршрути, що вимагають автентифікації за допомогою JWT-токена. Публічна зона /api/v1/public призначена для зовнішніх сервісів і автентифікується за допомогою API-ключа, що передається у заголовку запиту. Такий розподіл дозволяє зовнішнім системам ініціювати події без необхідності отримання JWT-токена [7].

Усі маршрути API організовано у вісім груп відповідно до функціональних модулів системи. Приклад груп та їх маршрутів для авторизації наведено у таблиці 2.1.

Таблиця 2.1 – Маршрути API Identity

Метод	**Маршрут**	**Опис**
POST	/api/v1/identity/register	Реєстрація нового облікового запису
GET	/api/v1/identity/confirm-email	Підтвердження електронної адреси
POST	/api/v1/identity/login	Вхід та отримання токенів
POST	/api/v1/identity/refresh	Оновлення пари токенів

Формат відповідей є уніфікованим: у разі успіху повертається JSON-об’єкт із відповідними даними, у разі помилки – об’єкт зі стандартизованою структурою, що містить код помилки та повідомлення. Для документування API застосовано специфікацію OpenAPI, що автоматично генерується на основі опису маршрутів. Інтерактивна документація доступна через інтерфейс Scalar за адресою /scalar/v1 та є публічно доступною без автентифікації.

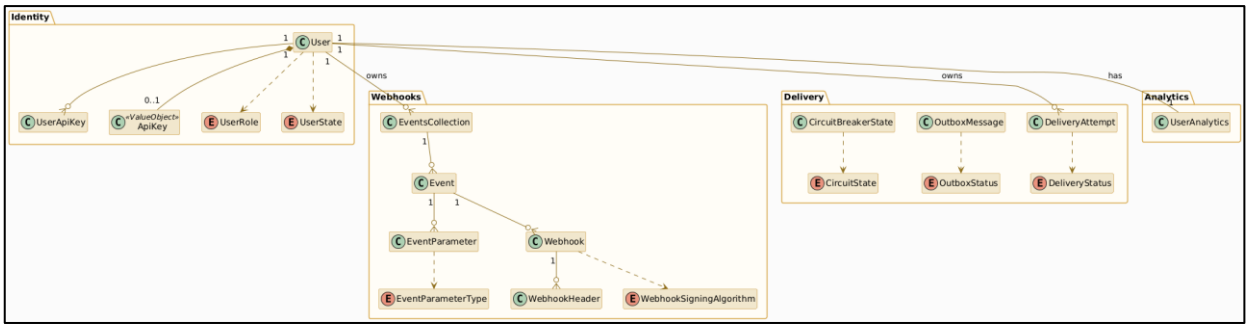


Рисунок 2.5 – Діаграма класів

2.4 Проєктування механізму доставки повідомлень

Механізм доставки є центральним компонентом системи, що забезпечує надійну асинхронну передачу вебхуків від моменту ініціювання події до отримання відповіді від цільового сервісу. Він реалізований як конвеєр з кількох незалежних компонентів, кожен з яких виконує чітко визначену роль.

Загальний потік доставки. Ініціювання події відбувається через HTTP POST-запит до ендпоінту /api/v1/public/deliveries/trigger. Після валідації запиту система визначає всі активні вебхуки для вказаної події та для кожного з них формує запис OutboxMessage зі статусом Pending і негайно зберігає його до бази даних. Відповідь клієнту повертається одразу після збереження, не очікуючи фактичної доставки. Такий підхід реалізує патерн «скринька вихідних повідомлень» (Outbox Pattern), що гарантує збереження повідомлення навіть у разі тимчасової недоступності брокера.

OutboxRelayWorker. Фоновий сервіс опитує базу даних кожну секунду та вибирає до 10 повідомлень зі статусом Pending, час відправки яких настав. Перед публікацією кожне повідомлення блокується на 30 секунд, що запобігає його повторній обробці у разі паралельного запуску кількох екземплярів сервісу. Після успішної публікації до RabbitMQ статус повідомлення змінюється на Published.

WebhookDispatchConsumerWorker. Фоновий сервіс підписується на чергу RabbitMQ та обробляє повідомлення типу DispatchWebhookMessage.

Для кожного повідомлення він викликає HttpWebhookDispatcher, зберігає результат доставки у таблиці DeliveryAttempts та оновлює стан автоматичного вимикача для відповідного вебхука. Після обробки результат публікується у потік живих оновлень (SSE) для відображення у реальному часі в інтерфейсі користувача.

HttpWebhookDispatcher. Безпосередньо виконує HTTP POST-запит до цільової URL-адреси. Тіло запиту передається у форматі JSON з типом вмісту application/json. Якщо для вебхука налаштовано секрет, обчислюється HMAC-підпис корисного навантаження з використанням обраного алгоритму – HMAC-SHA256, HMAC-SHA384 або HMAC-SHA512 [8]. Підпис додається до запиту у заголовку з налаштованою назвою у форматі sha256=<hex>. До запиту також додаються довільні заголовки, зареєстровані для вебхука. Таймаут запиту становить 30 секунд. Тіло відповіді зчитується з обмеженням у 16 КБ.

Стратегії повторних спроб. RetrySchedulerWorker опитує базу даних кожні 10 секунд та знаходить спроби доставки зі статусом Failed. Для кожної невдалої спроби перевіряється максимально дозволена кількість спроб (за замовчуванням 5). Якщо ліміт вичерпано, спроба позначається як Dead. В іншому випадку створюється новий запис OutboxMessage із часом відправки, розрахованим відповідно до обраної стратегії відступу. Система підтримує три стратегії: експоненційну (30 с → 2 хв → 10 хв → 30 хв), фіксовану (5 хвилин між спробами) та лінійну (2 хв, 4 хв, 6 хв і далі).

Автоматичний вимикач (Circuit Breaker) [4]. Для кожного вебхука система веде лічильник послідовних невдалих доставок. Після досягнення порогу у 5 послідовних помилок автоматичний вимикач переходить у стан Open і блокує нові спроби доставки на 1 хвилину. Після закінчення часу охолодження вимикач переходить у стан HalfOpen, допускаючи одну пробну спробу. У разі її успіху стан повертається до Closed із обнуленням лічильника. Цей механізм захищає цільові сервіси від надмірного навантаження під час тривалих збоїв та зберігає ресурси черги.

Далі на рисунках 2.6, 2.7 та 2.8 наведено діаграму станів для автоматичного вимикача, діаграму станів для спроб доставки та діаграму послідовності відповідно.

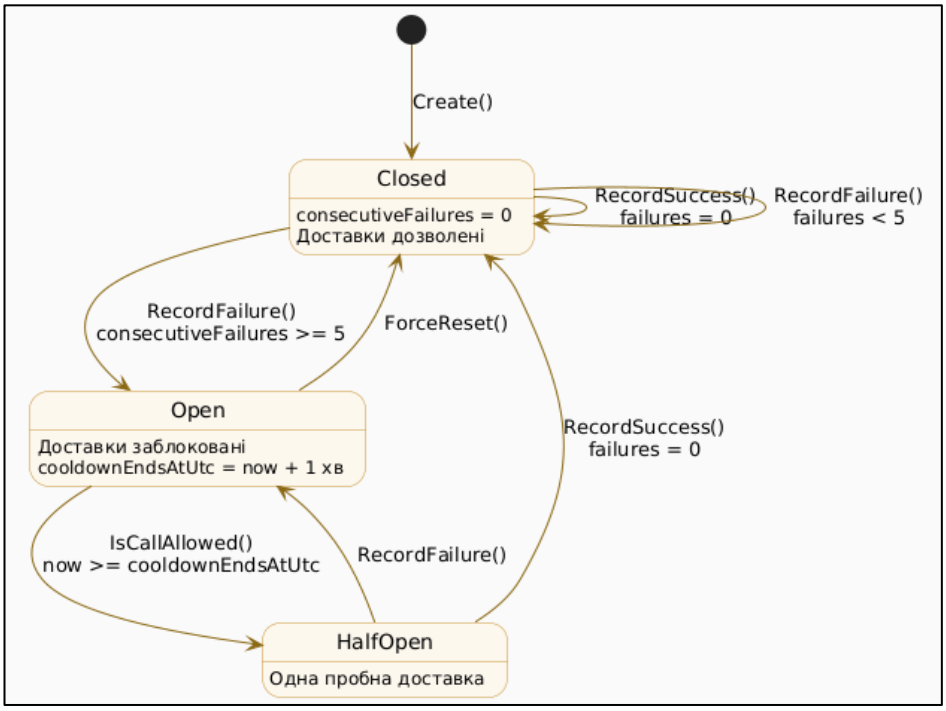


Рисунок 2.6 – Діаграма станів для автоматичного вимикача

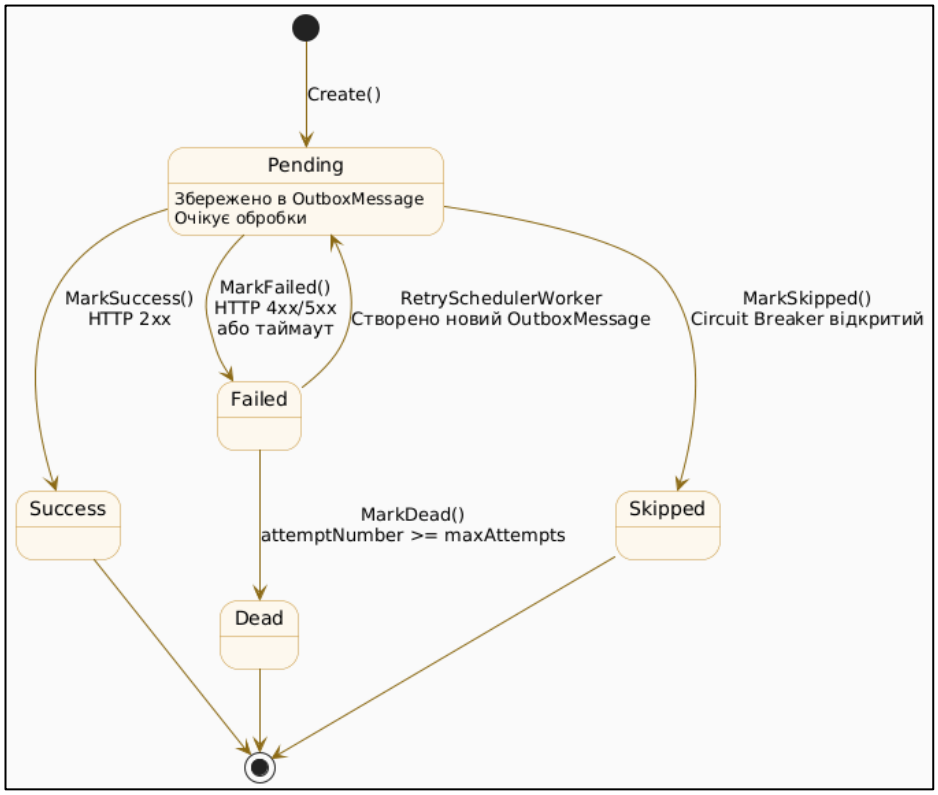


Рисунок 2.7 – Діаграма станів доставки

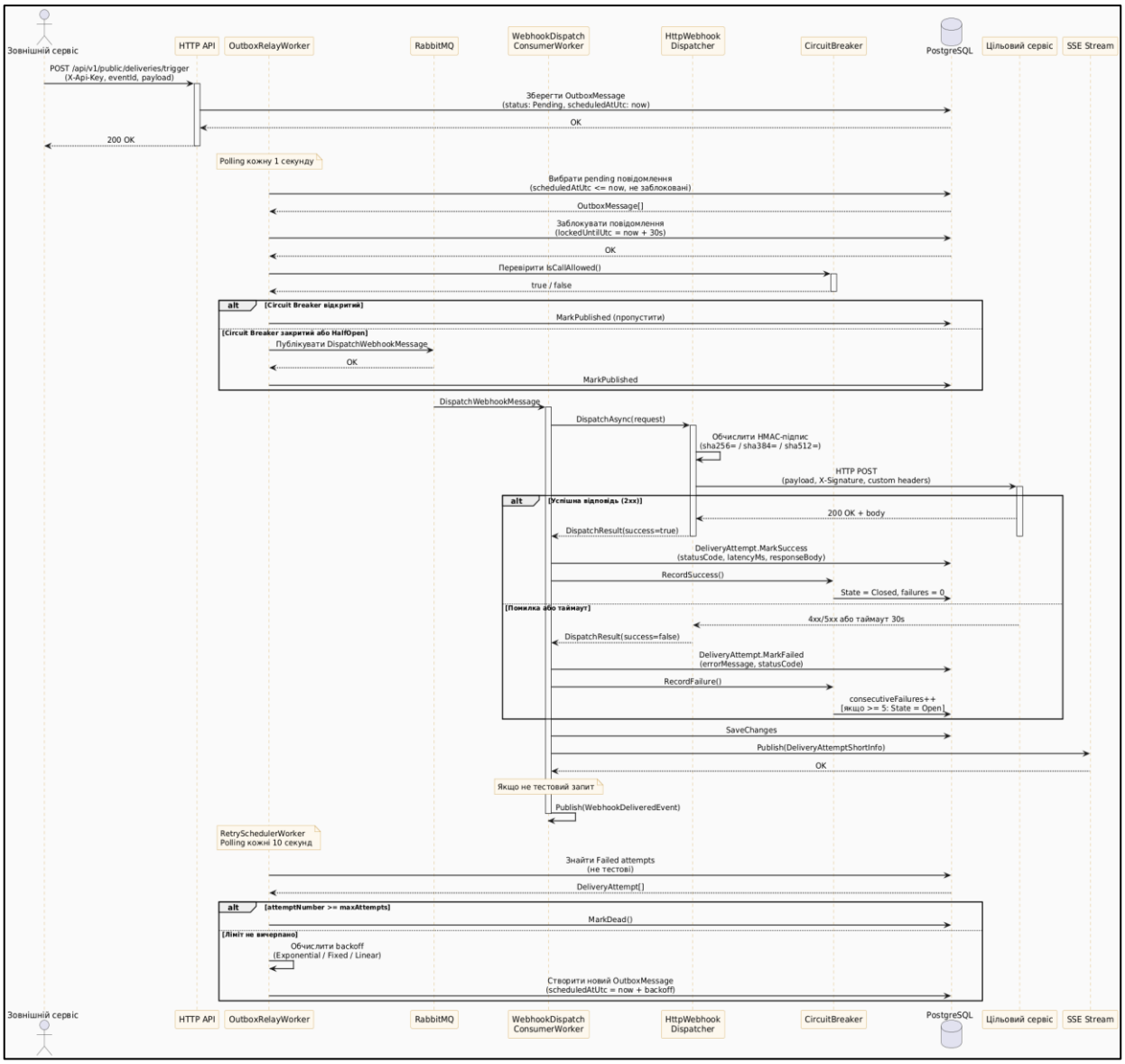


Рисунок 2.8 – Діаграма послідовності

2.5 Проєктування інтерфейсу користувача

Клієнтська частина системи реалізована як односторінковий застосунок (SPA) з використанням бібліотеки React 19 та мови TypeScript. Маршрутизація побудована на основі React Router v6 і організована у вигляді ієрархічного дерева маршрутів. Застосунок розподілено на чотири зони з різною логікою доступу та власним компонентом розкладки.

Публічна зона охоплює лендингову сторінку за адресою /, що доступна як неавтентифікованим відвідувачам, так і авторизованим користувачам. Вона

містить опис системи, перелік ключових функцій та посилання на дії залежно від стану автентифікації. Зона автентифікації включає сторінки входу /login, реєстрації /register та підтвердження електронної адреси /confirm-email. Ці сторінки не вимагають авторизації і використовують спрощений компонент розкладки AuthLayout з центрованою формою. Захищена зона користувача охоплює всі маршрути після успішного входу та відображається в межах компонента DashboardLayout, що містить бічну панель навігації. Зона адміністрування доступна лише користувачам з роллю Admin і використовує окремий компонент AdminLayout.

Бічна панель навігації DashboardLayout містить такі розділи: «Dashboard» із загальною статистикою, «Webhooks» зі списком усіх вебхуків, «Events» зі списком колекцій подій, «Logs» із журналом доставок та «Settings» з налаштуваннями облікового запису. У нижній частині панелі відображається аватар, ім’я та електронна адреса поточного користувача, а також кнопка виходу із системи.

Сторінка Dashboard є головним екраном після входу. Вона відображає агреговану статистику користувача: загальну кількість доставок, відсоток успішних, середню затримку, кількість активних подій та вебхуків. Також представлено графік тренду доставок за останні дні та список останніх спроб доставки в режимі реального часу через SSE.

Сторінка Webhooks відображає повний список вебхуків користувача. Сторінка WebhookDetail показує повну конфігурацію вебхука: URL-шаблон, алгоритм підпису, кастомні заголовки, налаштування повторних спроб. Тут доступні функції редагування, активації, деактивації, тестового запуску та видалення.

Сторінка Webhooks відображає повний список вебхуків користувача. Сторінка WebhookDetail показує повну конфігурацію вебхука: URL-шаблон, алгоритм підпису, кастомні заголовки, налаштування повторних спроб. Тут доступні функції редагування, активації, деактивації, тестового запуску та видалення.

Сторінка Events відображає список колекцій поточного користувача. Перехід до колекції відкриває сторінку CollectionDetail зі списком подій. Для кожної події доступна сторінка EventDetail, що відображає параметри схеми, налаштування шаблону тіла та список прив’язаних вебхуків. На цій сторінці можна активувати або деактивувати подію, відредагувати її параметри та видалити її.

Сторінка Logs є журналом усіх спроб доставки з підтримкою фільтрації та пошуку. Сторінка DeliveryAttemptDetail відображає деталі конкретної спроби: HTTP-статус, затримку, тіло запиту та відповіді, повідомлення про помилку, а також дозволяє ініціювати повторну доставку.

Сторінка Settings містить дві вкладки: керування API-ключами та інформацію про обліковий запис.

Адмін-панель AdminLayout надає доступ до розділів керування користувачами, колекціями, подіями та вебхуками в межах усієї платформи. Сторінка AdminOverview відображає системну статистику, стани автоматичних вимикачів та журнал усіх доставок.

Далі на рисунках 2.9 – 2.14 наведено знімки основних сторінок сайту.

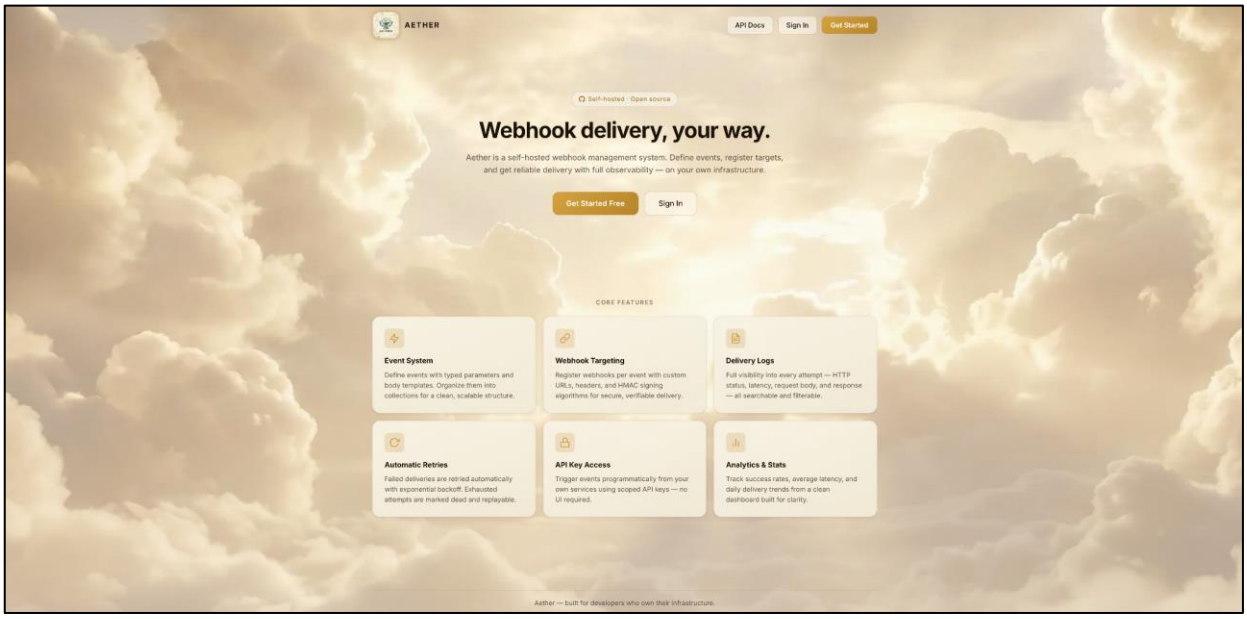


Рисунок 2.9 – “Landing” сторінка

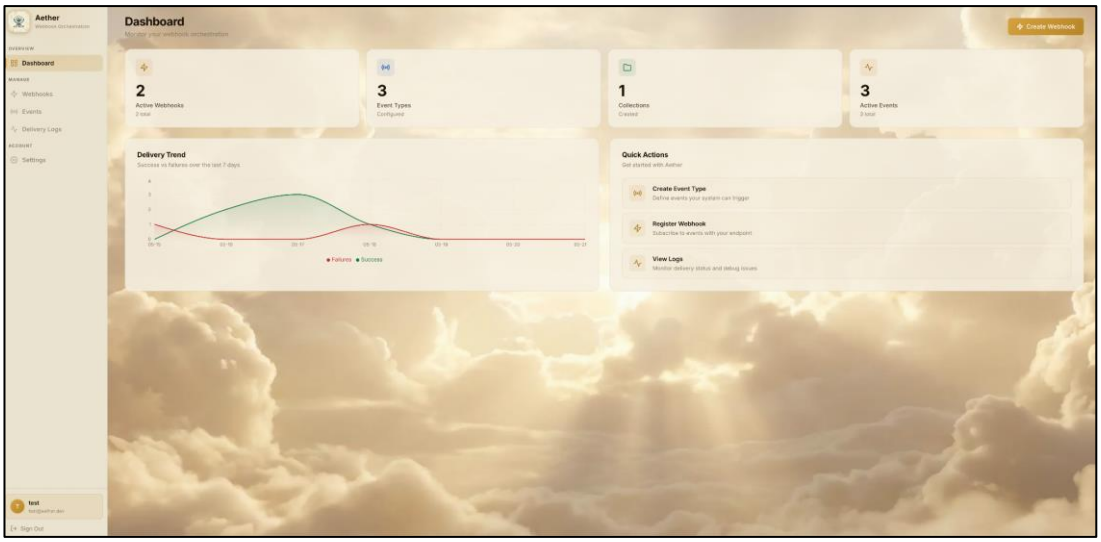


Рисунок 2.10 – “Dashboard” сторінка

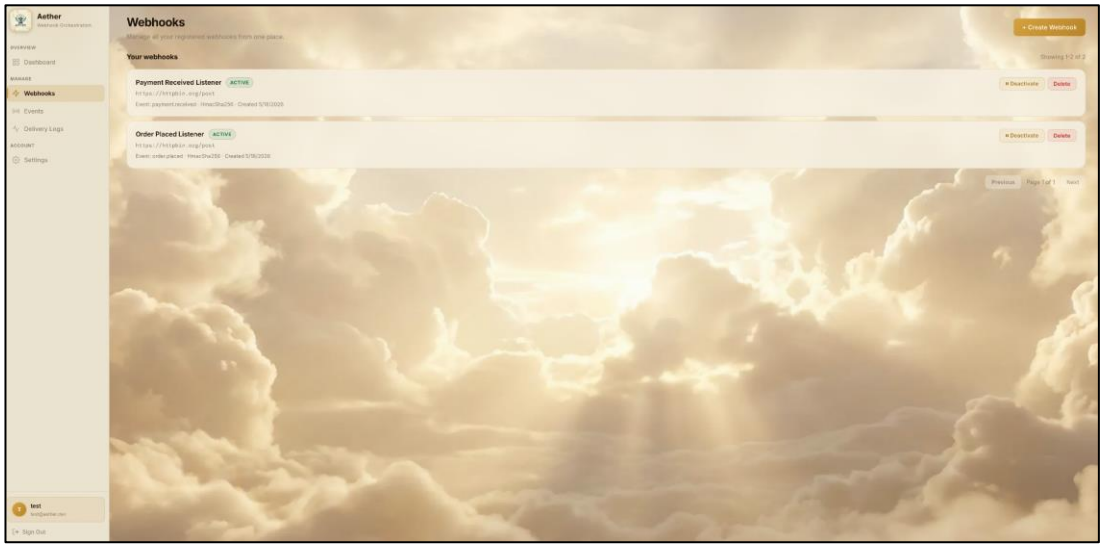


Рисунок 2.11 – “Webhooks” сторінка



Рисунок 2.12 – “Events” сторінка

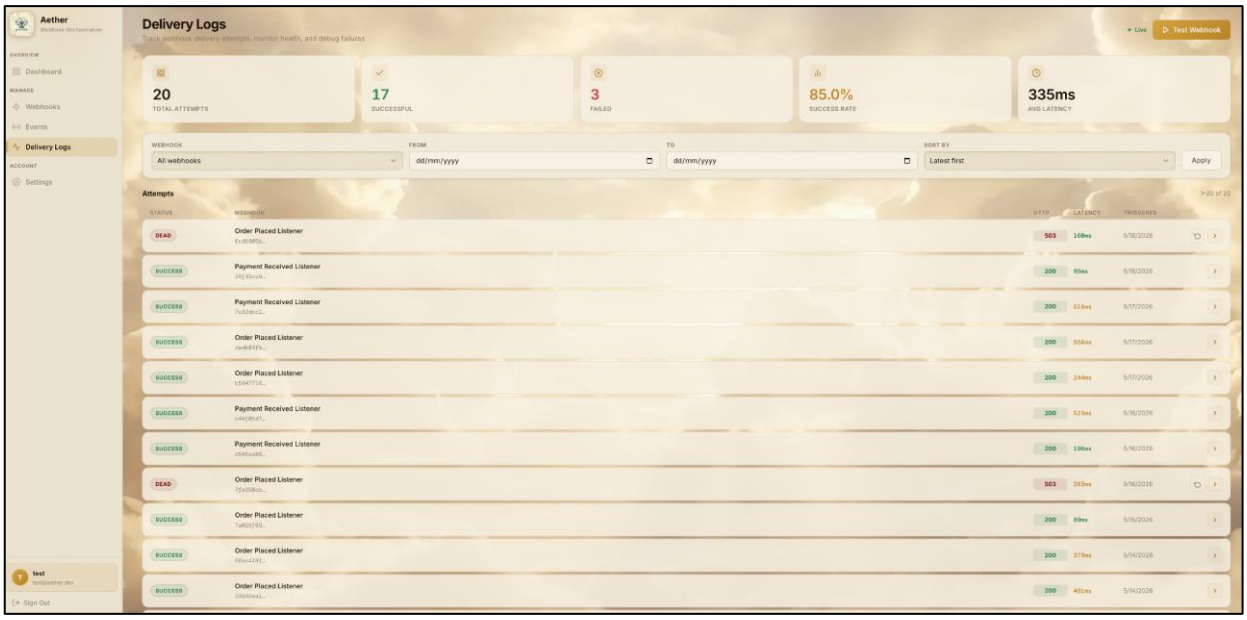


Рисунок 2.13 – “Delivery Logs” сторінка

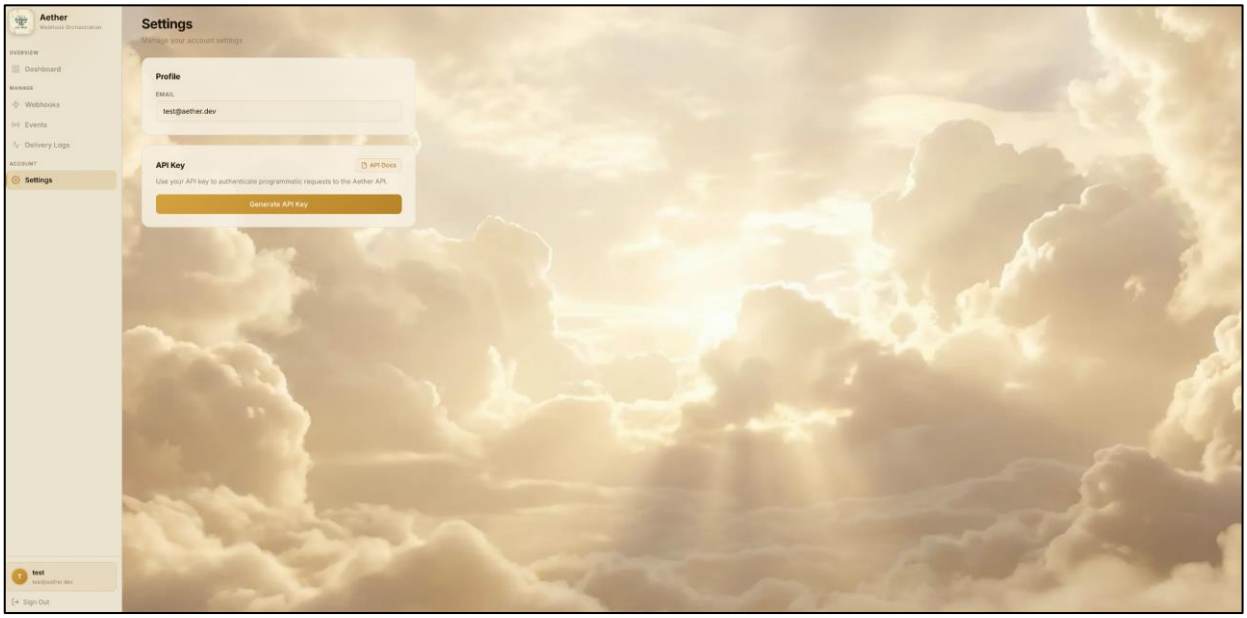


Рисунок 2.14 – “Settings” сторінка

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Реалізація серверної частини

Серверна частина системи реалізована як монолітний застосунок із модульною внутрішньою структурою на платформі .NET 10. Кодова база організована у вигляді окремих проєктів для кожного модуля, що дотримуються чіткого розподілу на шари: доменний, прикладний та інфраструктурний. Модулі системи – Identity, Webhooks, Delivery та Analytics – не залежать один від одного безпосередньо: взаємодія між ними відбувається виключно через інтерфейси контрактів, визначені у спільному проєкті Aether.Contracts. Точкою входу застосунку є проєкт Aether.HttpApi.

Реєстрація залежностей виконується у файлі Program.cs за допомогою послідовного виклику методів-розширень, кожен з яких інкапсулює реєстрацію сервісів одного модуля. Такий підхід дозволяє зберігати точку входу компактною і зрозумілою, не перетворюючи її на монолітний блок конфігурацій. Кожен модуль реєструє власні сервіси у двох методах: AddXxxApplication() для прикладного рівня та AddXxxInfrastructure() для інфраструктурного. Зовнішні налаштування – рядки підключення до баз даних, параметри RabbitMQ, JWT, SMTP – зчитуються з конфігурації через строго типізовані об’єкти-облікові дані, що реалізують інтерфейс ICredential.

Конвеєр обробки HTTP-запитів налаштований з урахуванням порядку виконання middleware. Спочатку застосовується ForwardedHeadersMiddleware для коректного визначення IP-адреси клієнта за зворотним проксі Caddy. Далі виконується GlobalExceptionHandler, що перехоплює всі необроблені винятки та перетворює їх на стандартизовані відповіді у форматі RFC 9457 Problem Details. Зокрема, ValueObjectValidationException перетворюється на статус 400, DomainException – на 422, UnauthorizedException – на 401. Після нього виконується SecurityHeadersMiddleware, що додає до кожної відповіді захисні

HTTP-заголовки: X-Content-Type-Options: nosniff, X-Frame-Options: DENY, Strict-Transport-Security для HTTPS-з’єднань та інші. Далі в конвеєрі йдуть CORS, автентифікація, BannedUserMiddleware та авторизація.

BannedUserMiddleware виконується після автентифікації та перевіряє, чи не заблоковано поточного автентифікованого користувача. Для зменшення кількості звернень до бази даних стан блокування кешується у пам’яті через IUserBanCache. У разі виявлення заблокованого користувача повертається відповідь 401 без передачі запиту далі по конвеєру.

Маршрутизація реалізована за допомогою Minimal API ASP.NET Core. Усі маршрути згруповані за модулями через MapGroup() з відповідними префіксами /api/v1/.... Кожна група налаштовує спільні для неї вимоги до авторизації та обмеження запитів безпосередньо на рівні групи, що усуває необхідність повторення цих атрибутів на кожному ендпоінті окремо. Група /api/v1/admin вимагає наявності ролі Admin, решта захищених груп вимагають лише автентифікованого користувача.

Обмеження частоти запитів реалізовано засобами вбудованого RateLimiter ASP.NET Core з кількома різними стратегіями для різних груп ендпоінтів. Для ендпоінтів автентифікації застосовано фіксоване вікно: не більше 20 запитів за 60 секунд з ідентифікацією за IP-адресою. Для стандартних захищених ендпоінтів – ковзне вікно на 300 запитів за 60 секунд з ідентифікацією за ідентифікатором користувача. Для публічного ендпоінту ініціювання подій застосовано алгоритм токен-відра: 60 токенів із поповненням 10 токенів кожні 10 секунд. Для тестового запуску вебхука та повторної доставки встановлено окремі фіксовані вікна – 10 і 20 запитів на хвилину відповідно. У разі перевищення ліміту повертається стандартизована відповідь 429 з заголовком Retry-After.

Документація API генерується автоматично на основі OpenAPI-специфікації з використанням бібліотеки Scalar. Інтерактивний інтерфейс доступний за адресою /scalar/v1 і зареєстрований до виклику UseAuthentication(), що робить його публічно доступним без автентифікації.

Специфікація містить опис схеми автентифікації Bearer, що дозволяє виконувати тестові запити безпосередньо з інтерфейсу документації.

Далі на рисунках 3.1 та 3.2 наведено схему структури проєкту та конвеєру middleware.

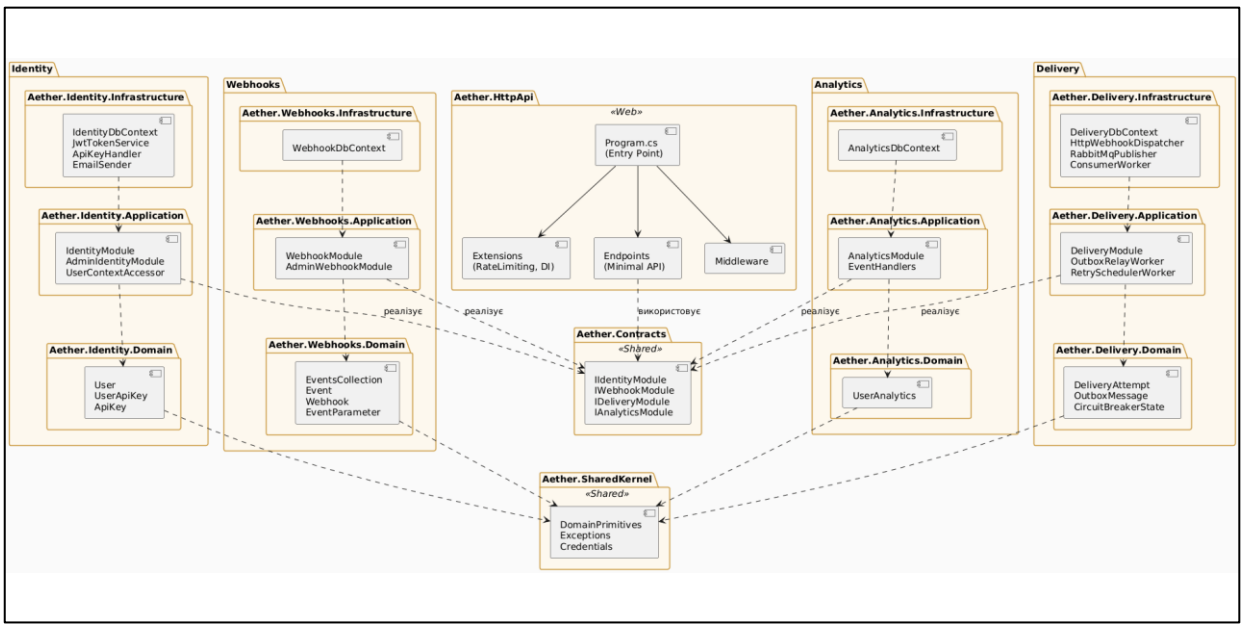


Рисунок 3.1 – Схема проєкту

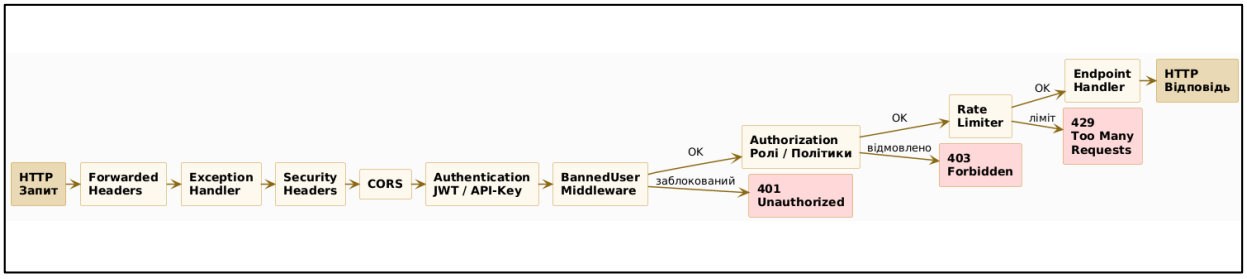


Рисунок 3.2 – Схема middleware

Інтерактивна документація API реалізована за допомогою бібліотеки Scalar та доступна за адресою api.aether-hooks.me/scalar/v1 без необхідності автентифікації (див. рис. 3.3).

Документація генерується автоматично на основі OpenAPI-специфікації та містить повний перелік ендпоінтів із описами, схемами запитів і відповідей. Інтерфейс підтримує виконання тестових запитів безпосередньо з браузера із використанням JWT-токена або API-ключа.

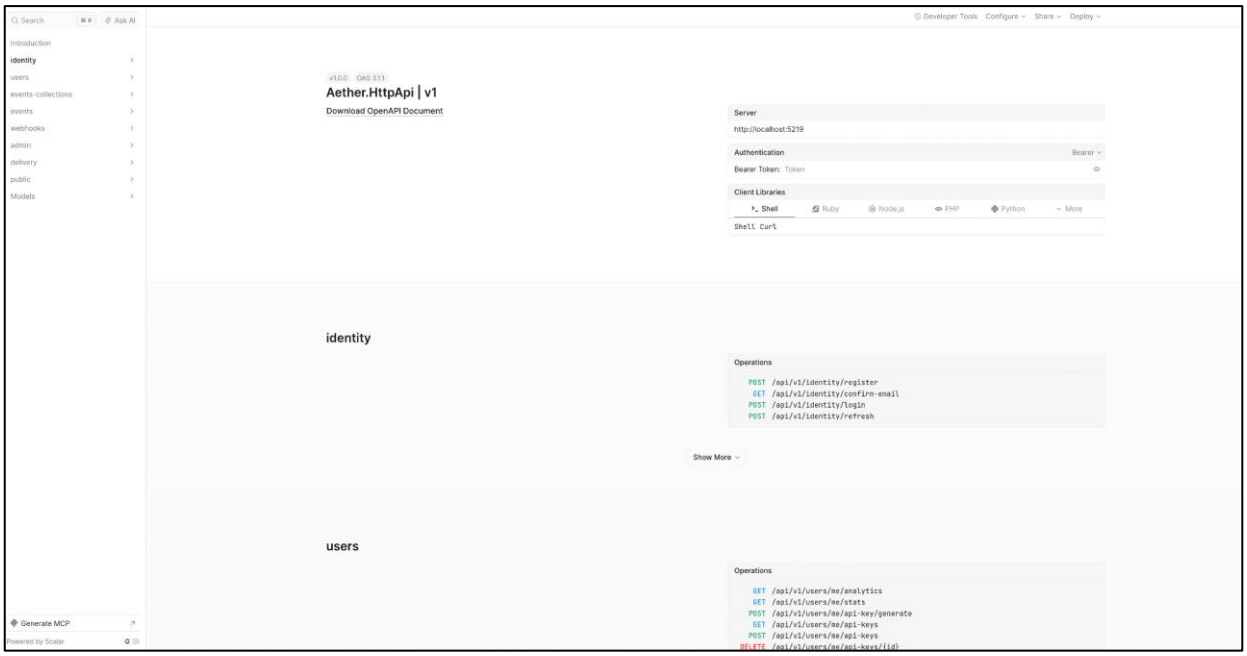


Рисунок 3.3 – Scalar API документація

3.2 Реалізація доменної моделі

Доменна модель системи реалізована з дотриманням принципів інкапсуляції та єдиної відповідальності. Усі бізнес-правила зосереджені безпосередньо в доменних об’єктах, а не розпорошені між сервісами прикладного рівня. Це забезпечує єдине місце перевірки інваріантів і унеможливлює переведення об’єкта у неприпустимий стан через зовнішній код.

Для типізації ідентифікаторів сутностей застосовано підхід доменних примітивів на основі readonly record struct. Кожна сутність має власний тип ідентифікатора: UserId, EventId, WebhookId, DeliveryAttemptId тощо. Такий підхід унеможливлює на рівні компілятора помилки виду «переплутав ідентифікатор події з ідентифікатором колекції», що є типовою проблемою при використанні звичайних типів Guid. Кожен примітив містить статичний метод New() для генерації нового значення та TryFrom() для парсингу з рядка.

Для доменних об’єктів-значень (Value Objects) визначено абстрактний базовий клас ValueObject, що реалізує порівняння за значенням через метод

GetAtomicValues(). Похідні класи перевизначають цей метод, перераховуючи складові значення об’єкта. Це дозволяє коректно порівнювати два об’єкти типу EmailAddress або HexColor без явного визначення Equals у кожному класі. Перевантажені оператори == та != делегують виклик до Equals.

Кожен об’єкт-значення має закритий конструктор і два статичних методи: TryCreate() та FromValid(). Метод TryCreate() виконує повну валідацію вхідних даних і у разі порушення інваріанту кидає ValueObjectValidationException. Метод FromValid() призначений виключно для відновлення об’єкта з уже перевірених даних, що зберігаються у базі даних, і пропускає перевірку. Така дворівнева конструкція є свідомим рішенням: вона забезпечує строгу перевірку на вході системи без зайвих витрат при читанні зі сховища.

Клас EmailAddress нормалізує адресу до нижнього регістру та перевіряє наявність рівно одного символу @ і обмеження довжини у 256 символів. Клас PasswordHash перевіряє мінімальну довжину паролю у 8 символів та делегує хешування зовнішньому сервісу через інтерфейс IPasswordHasher, зберігаючи доменний об’єкт незалежним від конкретної реалізації хешування. Клас EventName обмежує допустимі символи до малих латинських літер та крапки, що забезпечує сумісність з форматом collection.event_name. Клас HexColor перевіряє відповідність значення формату #RRGGBB за допомогою скомпільованого регулярного виразу.

Особливої уваги заслуговує реалізація WebhookUrl. Окрім стандартної перевірки формату URI, клас реалізує захист від SSRF-атак (Server-Side Request Forgery). При створенні об’єкта перевіряється, що хост не входить до списку заблокованих (localhost, 0.0.0.0 тощо), а якщо хост є IP-адресою – перевіряється належність до приватних діапазонів: 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, 169.254.0.0/16 (зокрема, адреса метаданих AWS) та інших зарезервованих діапазонів. IPv6-mapped IPv4-адреси нормалізуються перед перевіркою.

Доменні сутності використовують модифікатор private set або private init для всіх властивостей, що унеможливлює їх зміну ззовні. Будь-яка зміна стану

виконується виключно через відкриті методи сутності, які містять відповідні перевірки. Наприклад, метод User.Ban() перевіряє, що користувач ще не заблокований, і разом зі зміною стану інвалідує SecurityStamp, що автоматично анулює всі активні JWT-токени. Метод CircuitBreakerState.RecordFailure() інкапсулює логіку відстеження порогу відмов і самостійно приймає рішення про відкриття вимикача.

Виняткові ситуації доменного рівня оброблюються через ієрархію спеціалізованих класів винятків: DomainException для порушення бізнес-правил, ValueObjectValidationException для некоректних значень, UnauthorizedException для несанкціонованого доступу та ConflictException для конфліктів даних. Кожен тип маппується у GlobalExceptionHandler на відповідний HTTP-статус, що дозволяє доменному рівню залишатись незалежним від HTTP-специфіки.

Далі на рисунках 3.4 та 3.5 представлено діаграму ієрархії Value Objects та діаграму доменних примітивів відповідно.

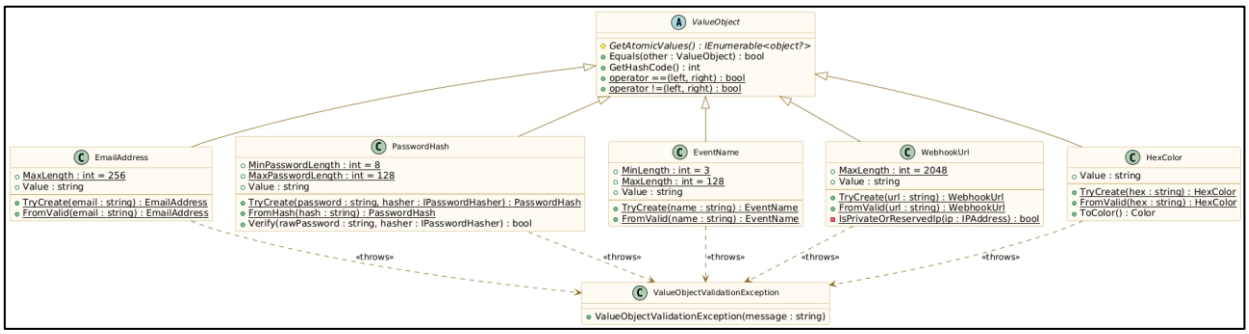


Рисунок 3.4 – Діаграма Value Objects

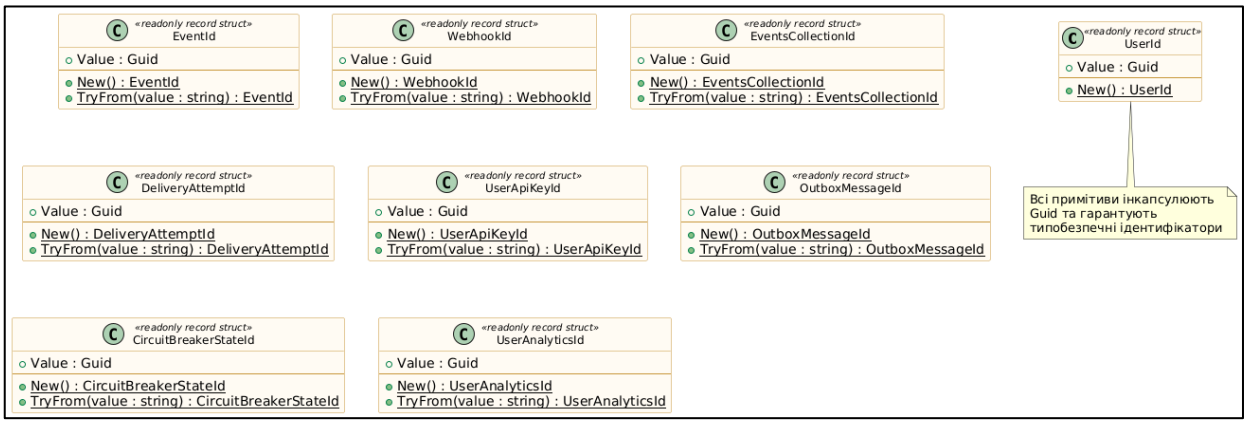


Рисунок 3.5 – Діаграма доменних примітивів

На рисунку 3.6 наведено код класу ValueObject.



```
19 usages  14 inheritors  YSecretY  0+39 exposing APIs
public abstract class ValueObject
{
    3 usages  13 overrides  YSecretY
    protected abstract IEnumerable<object?> GetAtomicValues();

    1 usage  YSecretY
    public bool Equals(ValueObject? other)
    {
        if (other is null || other.GetType() != GetType())
            return false;

        return GetAtomicValues().SequenceEqual(other.GetAtomicValues());
    }

    YSecretY
    public override bool Equals(object? obj) =>
        obj is ValueObject other && Equals(other);

    YSecretY
    public override int GetHashCode() =>
        GetAtomicValues() // IEnumerable<object?>
            .Aggregate(default(HashCode), (hash, value) =>
            {
                hash.Add(value);
                return hash;
            }) // HashCode
            .ToHashCode(); // int

    YSecretY
    public static bool operator ==(ValueObject? left, ValueObject? right) =>
        Equals(left, right);

    YSecretY
    public static bool operator !=(ValueObject? left, ValueObject? right) =>
        !Equals(left, right);
}
```

Рисунок 3.6 – Код класу ValueObject

3.3 Реалізація шару доступу до даних

Шар доступу до даних системи Aether реалізований засобами Entity Framework Core з використанням PostgreSQL як основного сховища. Ключовою архітектурною рішенням є відмова від єдиної бази даних на користь чотирьох ізольованих баз – по одній на кожен модуль системи: Identity, Webhooks, Delivery та Analytics. Такий підхід повністю узгоджується

з принципами архітектури обмежених контекстів (Bounded Context): кожен модуль є автономним щодо схеми, еволюції моделі та підключення, що унеможливлює міжмодульні з'єднання (JOIN) на рівні бази даних та забезпечує незалежне масштабування.

Кожен модуль містить власний клас DbContext, що реалізує відповідний інтерфейс, визначений у шарі застосування. Наприклад, WebhookDbContext реалізує IWebhookDbContext, IdentityDbContext – IIdentityDbContext тощо. Подібна структура забезпечує принципову залежність: шар застосування не має жодної прямої залежності від Entity Framework Core – він бачить лише інтерфейс із набором DbSet<T> та методом SaveChangesAsync. Це спрощує підміну реальної бази даних SQLite-контекстом у тестах без зміни будь-якого коду бізнес-логіки.

Усі чотири контексти побудовані за ідентичним шаблоном: конфігурація через ApplyConfigurationsFromAssembly автоматично виявляє та застосовує всі класи, що реалізують IEntityTypeConfiguration<T>, з відповідної збірки інфраструктури. Таким чином, додавання нової сутності не вимагає змін у самому DbContext – достатньо додати новий клас конфігурації.

Реєстрація контекстів у контейнері залежностей виконується у статичних методах розширення кожного модуля (див. рис. 3.7).

```
YSecretY
private void AddDatabase(WebhookPostgresCredential postgresCredential) =>
    services.AddDbContext<WebhookDbContext>(options => options.UseNpgsql(postgresCredential.ConnectionString)) // IServiceCollection
        .TryAddScoped<IWebhookDbContext>( implementationFactory: provider => provider.GetRequiredService<WebhookDbContext>());
```

Рисунок 3.7 – Реєстрація WebhookDbContext у DI

Метод TryAddScoped реєструє інтерфейс як псевдонім до того самого конкретного екземпляра в межах запиту. Завдяки цьому незалежно від того, де в коді запитується IWebhookDbContext чи WebhookDbContext – обидва вирішуються до одного й того самого об'єкта, зберігаючи транзакційну цілісність у межах одного HTTP-запиту.

Для кожної сутності визначено окремий клас конфігурації, що реалізує IEntityTypeConfiguration<T>. Всі конфігурації є internal – вони не видимі за межами інфраструктурної збірки та реєструються автоматично.

Принциповою особливістю є використання строго типізованих ідентифікаторів. Усі первинні та зовнішні ключі у доменній моделі є value-типами (наприклад, UserId, WebhookId, EventId), а не примітивними Guid. На рівні EF Core для кожного такого типу визначено явний конвертер значень, що перетворює доменний примітив у Guid при запису та відновлює назад при читанні. Для всіх ідентифікаторів встановлено ValueGeneratedNever(), що означає генерацію ідентифікаторів на стороні клієнта (у доменному шарі) через статичний метод New(), а не засобами бази даних.

Для полів, що відповідають value objects, конфігурація явно задає обмеження довжини, що відповідають константам доменної моделі. Наприклад, для WebhookUrl.MaxLength = 2048 та DeliveryAttempt.ResponseBody встановлено HasMaxLength(65536).

Індексування в системі є цілеспрямованим – кожен індекс відповідає конкретному запиту застосування, а не додається «на всяк випадок». Так, у конфігурації DeliveryAttemptConfigurations визначено два складені індекси:

– (UserId, IsTest, TriggeredAtUtc) – покриває основний запит журналу доставлень із фільтрацією та сортуванням;
– (UserId, IsTest, Status) – покриває запит статистики доставлень за статусом.

Для OutboxMessage визначено складений індекс (Status, ScheduledAtUtc), що забезпечує ефективне читання черги повідомлень фоновими сервісами OutboxRelayWorker та RetrySchedulerWorker.

Кожен модуль містить власну ізольовану папку міграцій, згенеровану засобами EF Core CLI. На момент розробки кожен модуль має одну міграцію Initial, що описує повну початкову схему відповідної бази даних. Застосування міграцій відбувається автоматично під час запуску застосування через виклик MigrateAsync для кожного контексту (див. рис. 3.8).

```
await using var scope = app.Services.CreateAsyncScope();
await IdentityModule.MigrateAsync(scope.ServiceProvider);
await WebhooksModule.MigrateAsync(scope.ServiceProvider);
await DeliveryModule.MigrateAsync(scope.ServiceProvider);
await AnalyticsModule.MigrateAsync(scope.ServiceProvider);
```

Рисунок 3.8 – Міграції бази даних

Такий підхід забезпечує узгодженість схеми бази даних зі станом моделі при кожному розгортанні без ручного втручання.

У системі свідомо відсутній патерн Unit of Work у його класичному вигляді. Модульні класи (WebhookModule, DeliveryModule тощо) взаємодіють безпосередньо з IXxxDbContext, а межі транзакцій визначаються викликом SaveChangesAsync у межах одного запиту. Вбудований механізм відстеження змін EF Core (Change Tracker) слугує неявною одиницею роботи в межах одного екземпляра DbContext зі скоупованим часом життя.

Далі на рисунку 3.9 наведено ER діаграму бази даних, а також додано приклади коду та реалізації необхідних модулей на рисунках 3.10, 3.11 та 3.12.

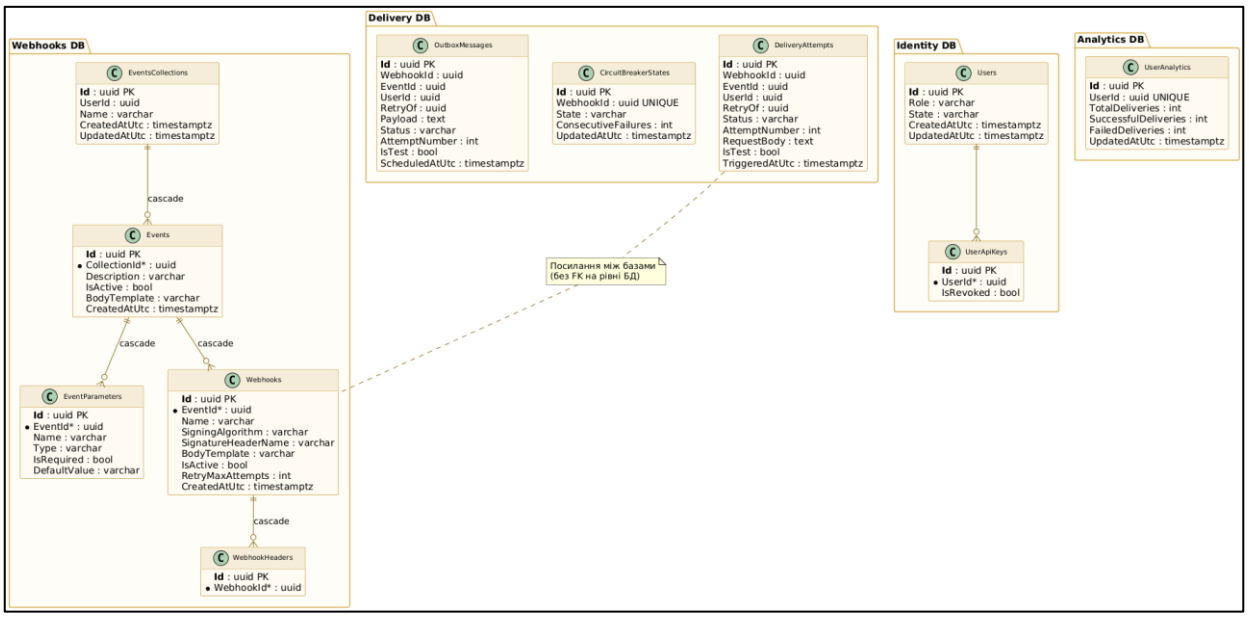


Рисунок 3.9 – ER діаграма

```
YSecretY
private void AddDatabase(WebhookPostgresCredential postgresCredential) =>
    services.AddDbContext<WebhookDbContext>(options => options.UseNpgsql(postgresCredential.ConnectionString)) // IServiceCollection
        .TryAddScoped<IWebhookDbContext>( implementationFactory: provider => provider.GetRequiredService<WebhookDbContext>());
```

Рисунок 3.10 – Реєстрація WebhookDbContext у DI

```
9 usages  1 inheritor  YSecretY  0+1 exposing APIs
public interface IWebhookDbContext
{
    11 usages  1 implementation  YSecretY
    public DbSet<EventsCollection> EventsCollections { get; }

    24 usages  1 implementation  YSecretY
    public DbSet<Event> Events { get; }

    25 usages  1 implementation  YSecretY
    public DbSet<Webhook> Webhooks { get; }

    17 usages  YSecretY
    public Task<int> SaveChangesAsync(CancellationToken cancellationToken = default);
}
```

Рисунок 3.11 – Інтерфейс IWebhookDbContext

```
YSecretY
internal class UserAnalyticsConfigurations : IEntityTypeConfiguration<UserAnalytics>
{
    YSecretY
    public void Configure(EntityTypeBuilder<UserAnalytics> builder)
    {
        builder.HasKey(x :UserAnalytics => x.Id);

        builder.Property(x :UserAnalytics => x.Id)
            .ValueGeneratedNever()
            .HasConversion(id => id.Value, guid => new UserAnalyticsId(guid));

        builder.Property(x :UserAnalytics => x.UserId)
            .IsRequired()
            .ValueGeneratedNever()
            .HasConversion(id => id.Value, guid => new UserId(guid));

        builder.Property(x :UserAnalytics => x.UpdatedAtUtc)
            .IsRequired();

        builder.HasIndex(x :UserAnalytics => x.UserId).IsUnique();
    }
}
```

Рисунок 3.12 – Конфігурація моделі UserAnalytics

3.4 Реалізація системи надійного доставлення

Надійність доставлення вебхуків є ключовою вимогою до системи: жодне тригерне повідомлення не повинно бути втрачене навіть у разі тимчасової недоступності цільового сервісу або збою в інфраструктурі. Для досягнення цієї мети в системі Aether реалізовано комбінацію трьох патернів:

Transactional Outbox, Circuit Breaker та стратегічний повтор із відкладеним плануванням.

Центральним елементом архітектури доставлення є патерн Transactional Outbox. Суть патерну полягає в тому, що замість безпосереднього надсилання HTTP-запиту до цільового сервісу в момент тригера, система записує повідомлення для доставлення в таблицю OutboxMessages у межах тієї самої транзакції бази даних, що й основна бізнес-операція. Це гарантує атомарність: або і тригерна подія, і запис у чергу збережені, або не збережено нічого.

Клас DeliveryModule.TriggerAsync при отриманні тригера знаходить усі активні вебхуки для відповідної події та для кожного з них створює окремий запис OutboxMessage із повним контекстом доставлення – URL, секрет, алгоритм підпису, заголовки, тіло запиту та час запланованого виконання (див. рис. 3.13).

```
foreach (var webhook in webhooks)
{
    var payload :string  = BuildPayload(webhook.Event.Name.Value, rng);

    var headers :string? = webhook.Headers.Count > 0
        ? JsonSerializer.Serialize(webhook.Headers.Select(h :WebhookHeader  => new { key = h.Key, value = h.Value }))
        : null;

    var message = OutboxMessage.Create(
        webhook.Id,
        webhook.EventId,
        user.Id,
        webhook.UrlTemplate.Value,
        webhook.Secret?.Value,
        webhook.SigningAlgorithm.ToString(),
        webhook.WebhookSignatureHeaderName.Value,
        headers,
        payload,
        attemptNumber: 1,
        isTest: false,
        retryOf: null,
        scheduledAtUtc: now
    );

    deliveryDb.OutboxMessages.Add(message);
}
```

Рисунок 3.13 – Приклад коду TriggerAsync

Клас OutboxRelayWorker є BackgroundService, що безперервно опитує таблицю OutboxMessages з інтервалом одна секунда. При кожній ітерації він

вибирає до десяти повідомлень зі статусом Pending, у яких ScheduledAtUtc <= now та LockedUntilUtc не встановлено або вже минув. Перед публікацією кожного повідомлення сервіс встановлює блокування на 30 секунд (LockedUntilUtc = now + 30s) та зберігає зміни в базі даних. Це забезпечує ідемпотентність у разі горизонтального масштабування: якщо два екземпляри сервісу одночасно опитують чергу, кожне повідомлення буде оброблено лише одним із них.

Перед публікацією в RabbitMQ сервіс перевіряє стан Circuit Breaker для відповідного вебхука. Якщо автоматичний вимикач відкритий, повідомлення позначається як Published без фактичного надсилання, що рівнозначно пропуску спроби в поточному циклі. Якщо автоматичний вимикач дозволяє виклик, повідомлення публікується у RabbitMQ через IDeliveryMessagePublisher.

Обробка повідомлень та HTTP-доставлення. WebhookDispatchConsumerWorker є RabbitMQ-споживачем, що отримує DispatchWebhookMessage та координує фактичне HTTP-доставлення. Послідовність дій для кожного повідомлення:

- виклик HttpWebhookDispatcher.DispatchAsync – надсилання HTTP POST із таймаутом 30 секунд;
- обчислення HMAC-підпису (SHA-256, SHA-384 або SHA-512) тіла запиту та додавання його до налаштованого заголовка;
- створення запису DeliveryAttempt з результатом: HTTP-статус, латентність, тіло відповіді або повідомлення про помилку;
- оновлення стану Circuit Breaker: RecordSuccess або RecordFailure;
- збереження DeliveryAttempt та оновленого CircuitBreakerState до бази даних;
- публікація результату в DeliveryStreamService для живого оновлення дашборду.

RetrySchedulerWorker – другий фоновий сервіс, що запускається кожні 10 секунд. Він знаходить усі спроби доставлення зі статусом Failed та для

кожної визначає, чи вичерпано максимальну кількість спроб. Якщо вичерпано: спроба позначається як Dead. Якщо ні – створюється новий запис OutboxMessage із розрахованим відкладенням відповідно до обраної стратегії:

– Exponential (за замовчуванням) – затримки 30 с, 2 хв, 10 хв, 30 хв для спроб 1-4;
– Linear – затримка = 2 хв × номер спроби;
– Fixed – фіксована затримка 5 хвилин між кожною спробою.

Завдяки полю ScheduledAtUtc новий запис не буде підхоплений OutboxRelayWorker раніше розрахованого часу, що реалізує відкладений повтор без жодного механізму sleep у коді.

Circuit Breaker реалізовано як доменну сутність CircuitBreakerState, персистовану у PostgreSQL. Для кожного вебхука існує рівно один запис стану. Автоматичний вимикач підтримує три стани (див. рис. 3.14).

– Closed – нормальна робота, всі виклики дозволені;
– Open – після 5 послідовних невдач вимикач відкривається на 1 хвилину, всі нові спроби пропускаються;
– HalfOpen – після закінчення cooldown-періоду один пробний виклик дозволяється; успіх переводить у Closed, невдача – знову у Open.

```
11 usages  YSecretY
public void RecordFailure(TimeProvider timeProvider)
{
    var now :DateTimeOffset = timeProvider.GetUtcNow();
    ConsecutiveFailures++;
    LastFailureAtUtc = now;
    UpdatedAtUtc = now;

    if (ConsecutiveFailures >= FailureThreshold && State != CircuitState.Open)
    {
        State = CircuitState.Open;
        OpenedAtUtc = now;
        CooldownEndsAtUtc = now.Add(CooldownDuration);
    }
}
```

Рисунок 3.14 – Приклад коду RecordFailure

Персистування стану у базі даних дозволяє зберігати його між перезапусками сервісу та забезпечує коректну роботу в багатоекземплярному розгортанні.

Окремо реалізовано механізм тестового пострілу (TestFireWebhookAsync), що повністю обходить чергу Outbox та RabbitMQ. HTTP-запит надсилається синхронно безпосередньо через IWebhookDispatcher, результат зберігається як DeliveryAttempt із прапорцем IsTest = true.

Тестові спроби виключені з усіх запитів статистики, повторів та дашборду, проте відображаються в журналі з відповідним фільтром.

Далі на рисунках 3.15, 3.16 та 3.17 наведено діаграму послідовностей доставки, Circuit Breaker та RetrySchedulerWorker відповідно.

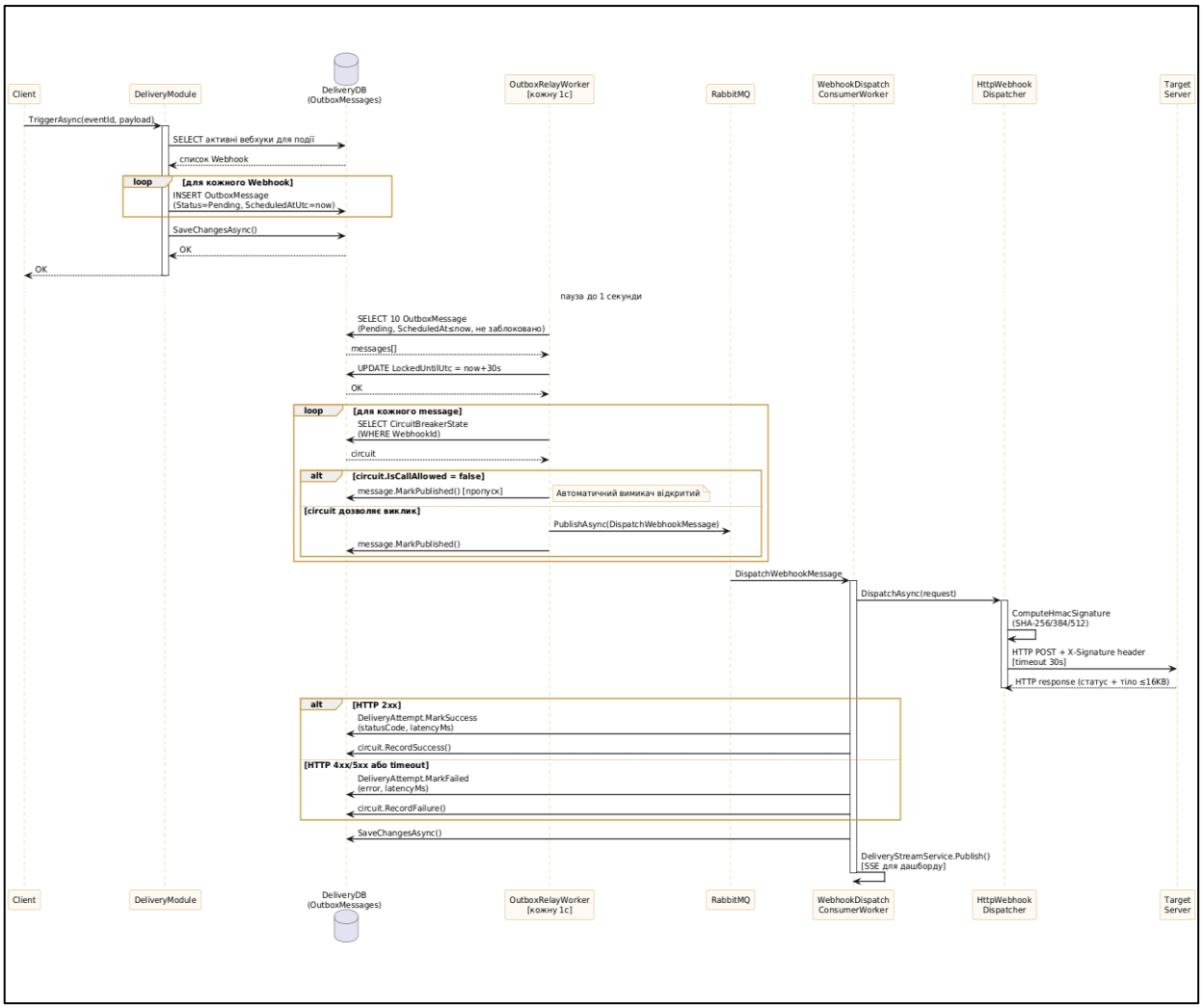


Рисунок 3.15 – Діаграма Sequence-Delivery

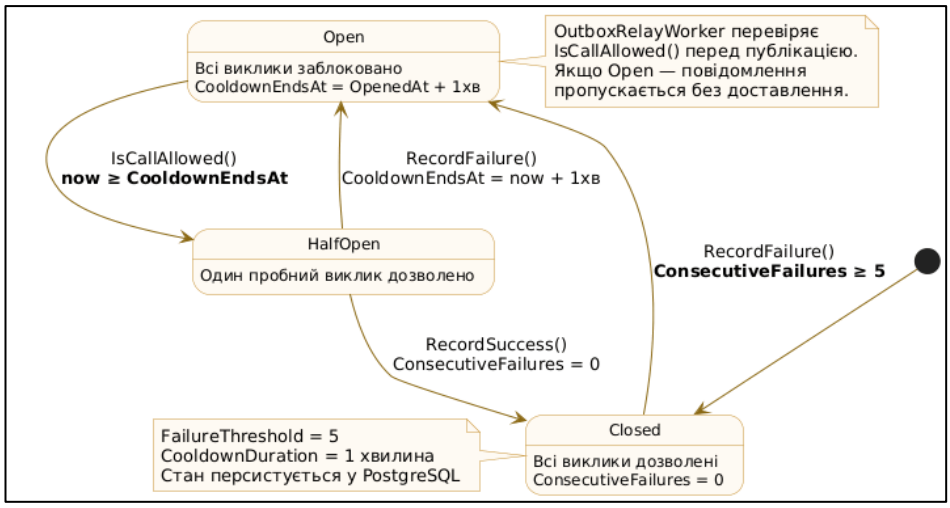


Рисунок 3.16 – Діаграма Circuit-Breaker

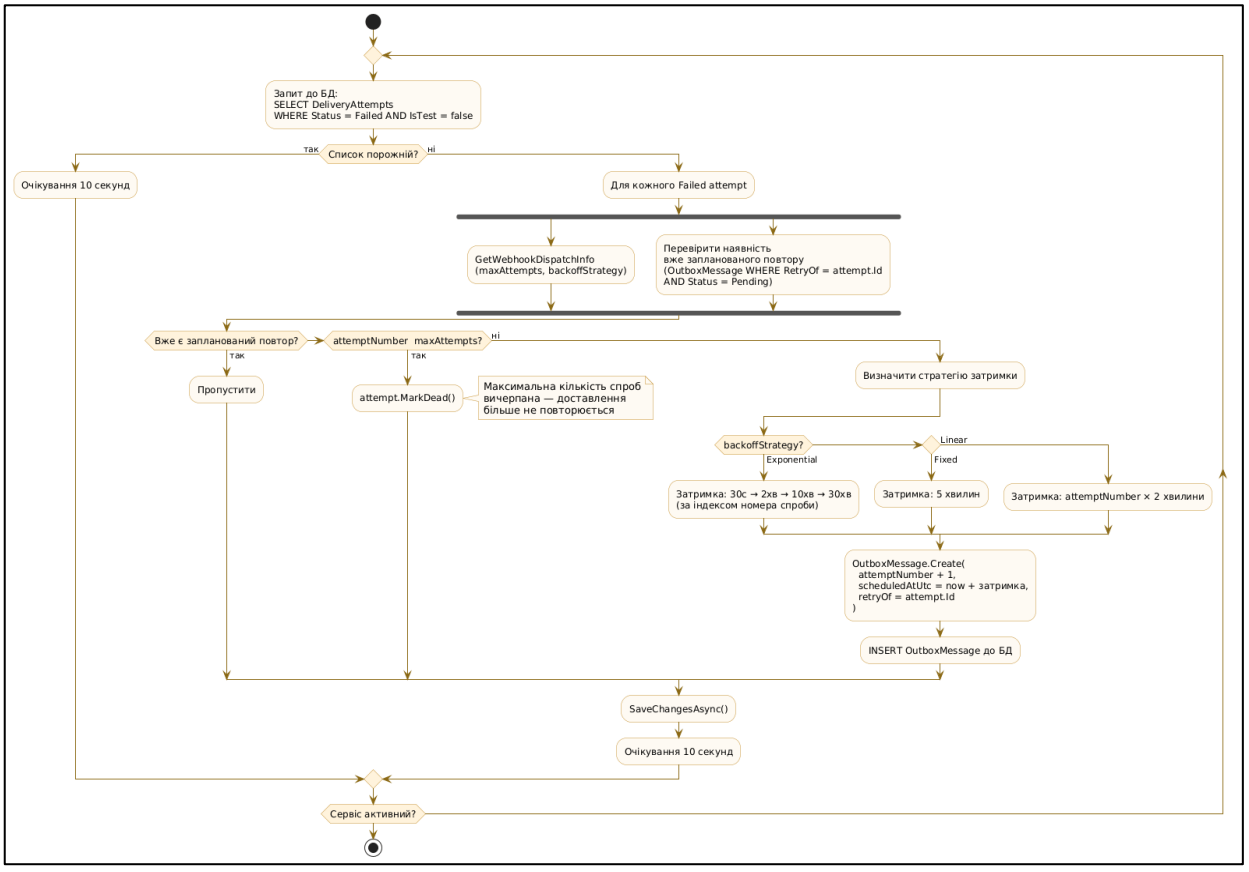


Рисунок 3.17 – Діаграма RetrySchedulerWorker

3.5 Реалізація автентифікації та авторизації

Система Aether підтримує два рівноправні механізми автентифікації: JWT Bearer-токени для інтерактивної роботи через веббраузер та API-ключі

для програмного доступу зі сторонніх сервісів. Обидва механізми реалізовані як окремі схеми автентифікації ASP.NET Core і об'єднані єдиною політикою авторизації за замовчуванням.

Сервіс JwtTokenService генерує пару токенів – токен доступу та токен оновлення – за алгоритмом HMAC-SHA256 з симетричним ключем. Токен доступу містить такі claims: ідентифікатор користувача (ClaimTypes.NameIdentifier), email, роль ("role") та кастомний claim "token_type": "Access". Час дії становить 15 хвилин. Токен оновлення містить ідентифікатор користувача, "token_type": "Refresh" та "security_stamp" – рядок, що слугує серверним механізмом інвалідації сесій (див. рис. 3.18).

```
4 usages  YSecretY
public class JwtCredential : ICredential
{
    0+2 usages  YSecretY
    public static string SectionName => "Jwt";

    3 usages
    public required string SecretKey { get; init; }

    4 usages
    public required string Issuer { get; init; }

    4 usages
    public required string Audience { get; init; }

    1 usage
    public int AccessTokenExpirationMinutes { get; init; } = 15;

    1 usage
    public int RefreshTokenExpirationDays { get; init; } = 7;
}
```

Рисунок 3.18 – Приклад коду JwtCredential

Оновлення токена виконується за класичною схемою: клієнт надсилає refresh-токен, сервер перевіряє його підпис, термін дії та token_type, після чого видає нову пару токенів.

Ключовим елементом безпеки є SecurityStamp – рядок у форматі GUID, що зберігається у записі користувача та включається до кожного refresh-токена. При кожному запиті на оновлення сервер перевіряє збіг штампу з поточним значенням у базі даних. Штамп змінюється автоматично при будь-

якій чутливій операції: зміні пароля, зміні email або блокуванні облікового запису. Завдяки цьому всі видані refresh-токени стають недійсними без потреби у відстеженні чорного списку або зберіганні стану токенів на сервері.

Для програмного доступу реалізовано кастомний обробник ApiKeyAuthenticationHandler, що успадковує AuthenticationHandler<ApiKeyAuthenticationOptions> і зареєстрований під схемою "ApiKey". Обробник зчитує заголовок X-API-Key та виконує такі перевірки:

– обчислення SHA-256 хешу отриманого ключа та пошук відповідного запису UserApiKey у базі даних;
– перевірка, що ключ не відкликаний (IsRevoked = false) та не прострочений (ExpiresAtUtc);
– завантаження пов’язаного користувача та перевірка статусу Active.

Оновлення поля LastUsedAtUtc при кожній успішній автентифікації. API-ключі зберігаються виключно у вигляді SHA-256 хешу. У відкритому вигляді зберігаються лише останні чотири символи (LastFour) для ідентифікації ключа користувачем. Система підтримує до 10 активних ключів на одного користувача з можливістю іменування та відкликання кожного.

Хешування паролів виконується за алгоритмом Argon2 через бібліотеку Isopoh.Cryptography.Argon2 [9]. Доменна модель PasswordHash примусово перевіряє довжину пароля (від 8 до 128 символів) до передачі хашеру, що унеможливлює зберігання паролів, що не відповідають вимогам.

Порядок проміжного програмного забезпечення є принципово важливим для коректної роботи механізмів безпеки (див. рис. 3.19):

– UseForwardedHeaders;
– UseExceptionHandler;
– SecurityHeadersMiddleware;
– UseCors;
– UseHttpsRedirection;
– UseAuthentication;

– BannedUserMiddleware;
– UseAuthorization;
– UseRateLimiter.

```
builder.Services // IServiceCollection
    .AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.MapInboundClaims = false;
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtCredential.SecretKey)),
            ValidateIssuer = true,
            ValidIssuer = jwtCredential.Issuer,
            ValidateAudience = true,
            ValidAudience = jwtCredential.Audience,
            ValidateLifetime = true,
            ClockSkew = TimeSpan.Zero,
            RoleClaimType = "role",
            NameClaimType = "sub"
        };
    })
    .AddScheme<ApiKeyAuthenticationOptions, ApiKeyAuthenticationHandler>(
        ApiKeyAuthenticationHandler.SchemeName,
        configureOptions: _ => { }
    );

builder.Services.AddAuthorizationBuilder()
    .SetDefaultPolicy(new AuthorizationPolicyBuilder(
            params authenticationSchemes: JwtBearerDefaults.AuthenticationScheme,
            ApiKeyAuthenticationHandler.SchemeName)
        .RequireAuthenticatedUser() // AuthorizationPolicyBuilder
        .Build())
    .AddPolicy(ApiKeyAuthenticationHandler.SchemeName, configurePolicy: policy => policy
        .AddAuthenticationSchemes(ApiKeyAuthenticationHandler.SchemeName)
        .RequireAuthenticatedUser());
```

Рисунок 3.19 – Приклад реєстрації сервісів авторизації

Далі на рисунку 3.20 наведено діаграму послідовностей процесу автентифікації та оновлення токенів. Діаграма відображає повний цикл – від входу користувача до інвалідації сесії через ротацію SecurityStamp. Після успішної перевірки облікових даних сервіс JwtTokenService генерує пару токенів: короткоживучий токен доступу з терміном дії 15 хвилин та токен оновлення терміном 7 днів із вбудованим SecurityStamp. При кожному запиті на оновлення сервер звіряє штамп із поточним значенням у базі даних – у разі зміни пароля, email або блокування облікового запису штамп ротується, що миттєво інвалідує всі видані refresh-токени без потреби у чорному списку.

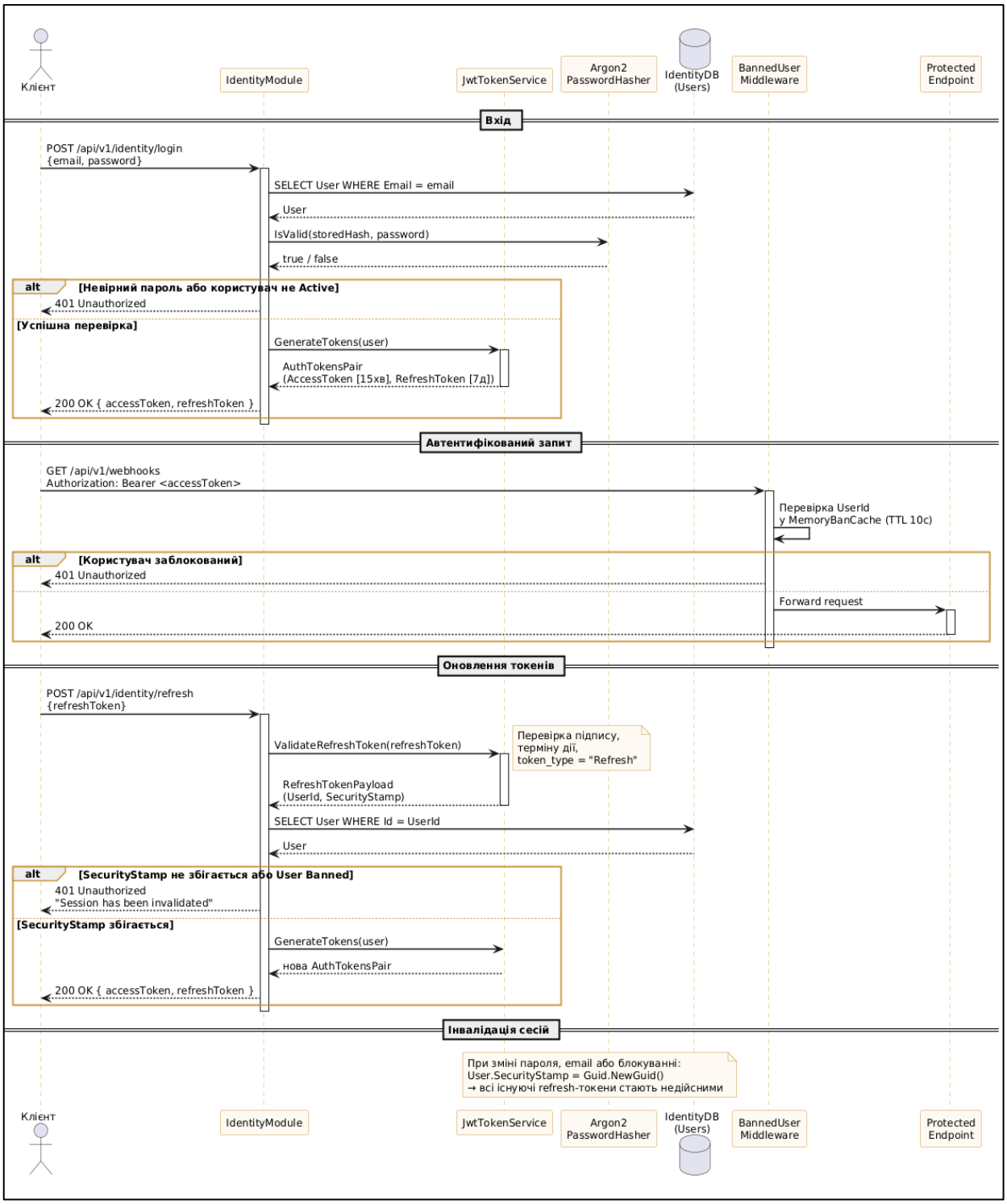


Рисунок 3.20 – Діаграма послідовностей авторизації

При вході користувач надсилає облікові дані, сервер перевіряє пароль через Argon2 та генерує пару токенів: короткоживучий токен доступу (15 хвилин) і токен оновлення (7 днів) із вбудованим SecurityStamp. Кожен автентифікований запит проходить через BannedUserMiddleware, що перевіряє статус користувача через кеш із TTL 10 секунд до передачі запиту обробнику.

При оновленні токенів сервер додатково звіряє SecurityStamp із поточним значенням у базі даних. Зміна пароля, email або блокування облікового запису ротує штамп, миттєво інвалідуючи всі видані refresh-токени без потреби у чорному списку.

3.6 Реалізація обмеження частоти запитів

Обмеження частоти запитів (rate limiting) є важливим механізмом захисту API від перевантаження, зловживань та DoS-атак. У системі Aether реалізовано диференційовану стратегію обмеження: для кожної категорії маршрутів обрано окремий алгоритм та параметри, що відповідають характеру та очікуваному навантаженню відповідних операцій.

Система використовує три різних алгоритми обмеження, що надаються вбудованим механізмом ASP.NET Core (System.Threading.RateLimiting) [10]. Для кожного типу маршрутів визначено іменовану політику з індивідуальними параметрами (див. табл. 3.1).

Таблиця 3.1 – Порівняння політик обмеження кількості запитів

Політика	**Алгоритм**	**Ліміт**	**Вікно**
auth	Fixed Window	20 запитів	60 с
default	Sliding Window	300 запитів	60 с (6 сегментів)
public-read	Sliding Window	120 запитів	60 с (6 сегментів)
public-trigger	Token Bucket	60 токенів	+10 токенів / 10 с
test-fire	Fixed Window	10 запитів	60 с
replay	Fixed Window	20 запитів	60 с

Fixed Window застосовується для рідкісних або чутливих операцій: автентифікація, тестовий постріл, повтор доставлення. Простота алгоритму та жорсткі межі унеможливлюють брутфорс облікових даних та зловживання ресурсномісткими операціями.

Sliding Window використовується для основних CRUD-операцій. На відміну від Fixed Window, ковзне вікно рівномірно розподіляє навантаження в часі завдяки поділу вікна на 6 сегментів, що усуває ефект «сплеску» на межі вікна.

Token Bucket застосовується для публічного тригера подій (public-trigger). Алгоритм найкраще відповідає природі тригерних запитів: дозволяє короткочасні піки активності (до 60 токенів накопичено) при стабільному поповненні (10 токенів кожні 10 секунд), що відповідає сценарію пакетного надсилання подій з зовнішніх систем.

Принципово важливим є вибір ключа розподілу (partition key) – критерію, за яким лічильник запитів прив'язується до конкретного клієнта (див. рис. 3.21).

```
limiter.AddPolicy(RateLimitPolicy.Auth, partitioner: ctx:HttpContext =>
    RateLimitPartition.GetFixedWindowLimiter(
        IpKey(ctx),
        factory: _ => new FixedWindowRateLimiterOptions
        {
            PermitLimit = opts.Auth.PermitLimit,
            Window = TimeSpan.FromSeconds(opts.Auth.WindowSeconds),
            QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
            QueueLimit = 0
        }));
```

Рисунок 3.21 – Приклад реєстрації FixedWindowLimiter

Для маршруту auth використовується IP-адреса, оскільки на момент обробки запиту користувач ще не автентифікований. Для всіх інших захищених маршрутів ключем є ідентифікатор користувача з JWT-токена, що забезпечує справедливий розподіл лімітів між користувачами незалежно від їх

IP-адрес – важливо в умовах NAT або проксі-серверів. Якщо ідентифікатор недоступний (API-ключ або анонімний запит), використовується IP-адреса.

Конфігурація та відповідь при перевищенні ліміту. Усі числові параметри винесено до конфігураційного класу RateLimitingOptions, що зчитується з розділу RateLimiting файлу конфігурації; це дозволяє налаштовувати ліміти без перекомпіляції застосування (див. рис. 3.22).

```
3 usages  YSecretY
internal sealed class RateLimitingOptions
{
    public const string SectionName = "RateLimiting";

    2 usages
    public FixedWindowConfig Auth { get; init; } = new(PermitLimit: 20, WindowSeconds: 60);
    3 usages
    public SlidingWindowConfig Default { get; init; } = new(PermitLimit: 300, WindowSeconds: 60, SegmentsPerWindow: 6);
    3 usages
    public SlidingWindowConfig PublicRead { get; init; } = new(PermitLimit: 120, WindowSeconds: 60, SegmentsPerWindow: 6);
    3 usages
    public TokenBucketConfig PublicTrigger { get; init; } = new(TokenLimit: 60, TokensPerPeriod: 10, ReplenishmentPeriodSeconds: 10);
    2 usages
    public FixedWindowConfig TestFire { get; init; } = new(PermitLimit: 10, WindowSeconds: 60);
    2 usages
    public FixedWindowConfig Replay { get; init; } = new(PermitLimit: 20, WindowSeconds: 60);
}

3 usages  YSecretY  3 exposing APIs
internal sealed record FixedWindowConfig(int PermitLimit, int WindowSeconds);

2 usages  YSecretY  2 exposing APIs
internal sealed record SlidingWindowConfig(int PermitLimit, int WindowSeconds, int SegmentsPerWindow);

1 usage  YSecretY  1 exposing API
internal sealed record TokenBucketConfig(int TokenLimit, int TokensPerPeriod, int ReplenishmentPeriodSeconds);
```

Рисунок 3.22 – RateLimitingOptions

При перевищенні ліміту клієнт отримує відповідь HTTP 429 з заголовком Retry-After та тілом у форматі application/problem+json відповідно до RFC 9110, що дозволяє клієнтським застосуванням коректно обробляти throttling і автоматично повторювати запити після вказаного інтервалу.

Паралельно з основними модулями в системі реалізовано аналітичний модуль, що акумулює статистику доставлень у реальному часі. Модуль побудований на основі подієво-орієнтованої архітектури: замість прямих викликів між модулями використовуються доменні події через IPublisher (Mediator).

При кожному доставленні WebhookDispatchConsumerWorker публікує подію WebhookDeliveredEvent. Обробник WebhookDeliveredEventHandler в

аналітичному модулі оновлює агреговану статистику користувача: кількість успішних та невдалих доставлень, сумарний та середній час відповіді, часові мітки останніх операцій. Аналогічно, зміни ресурсів (створення або видалення вебхуків, подій, колекцій) публікуються як WebhookResourceChangedEvent та обробляються WebhookResourceChangedEventHandler, що оновлює лічильники ресурсів через дельта-механізм.

Такий підхід повністю розв’язує модулі між собою: модуль доставлення не залежить від аналітичного модуля і не знає про його існування. Аналітика є пасивним спостерігачем, що реагує на події системи.

Діаграма послідовностей аналітичного модуля представлена на рисунку 3.23.

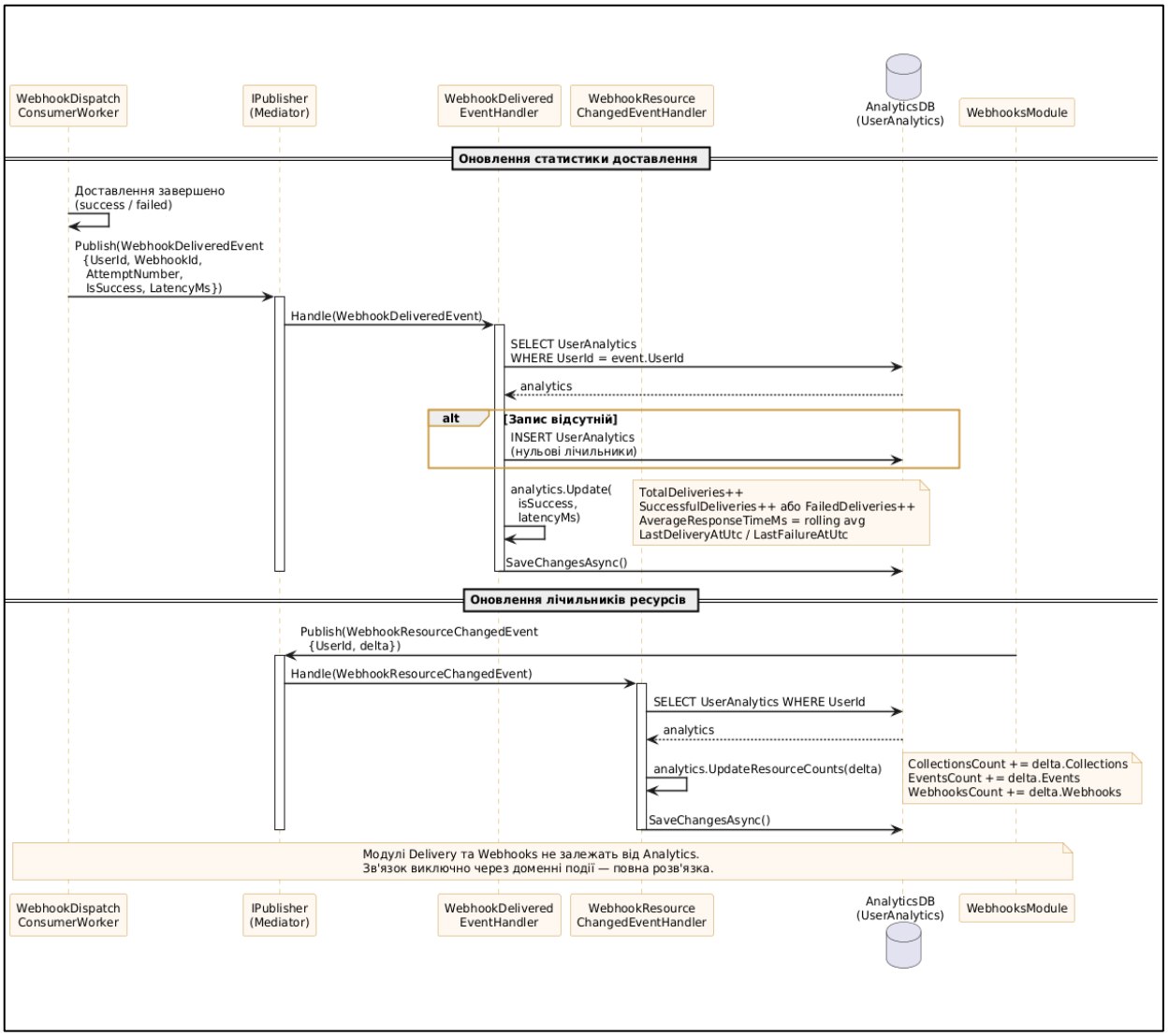


Рисунок 3.23

3.7 Тестування системи

Тестування системи Aether організовано у вигляді трирівневої піраміди: від ізольованих модульних тестів доменної логіки до інтеграційних тестів повного стека. Загалом проєкт містить сім тестових збірок, розподілених за трьома категоріями: Units, Integration та Exploratory. Основним фреймворком є xUnit, для підміни залежностей використовується бібліотека NSubstitute.

Модульні тести охоплюють три збірки: Aether.Webhooks.Domain.Tests, Aether.Delivery.Domain.Tests та Aether.Delivery.Infrastructure.Tests. Вони не мають залежностей від бази даних або DI-контейнера і виконуються повністю ізольовано.

Тести доменних сутностей перевіряють коректність бізнес-логіки value objects та агрегатів. Для кожного value object визначено окремий тестовий клас, що покриває: відхилення порожніх та null-значень, граничні довжини рядків, нормалізацію (trim, lowercase), формати (hex-кольори, URL-схеми) та рівність об’єктів. Очікуваним винятком є ValueObjectValidationException.

Окремо тестується SSRF-захист у WebhookUrl – параметризовані тести перевіряють блокування приватних діапазонів IP: RFC1918 (10.x, 172.16–31.x, 192.168.x), link-local (169.254.x.x), loopback (127.x), carrier-grade NAT (100.64.x), а також заблокованих хостів (localhost, 0.0.0.0) (див. рис. 3.24).

```
[Theory]
[InlineData(url: "http://localhost/hook")]
[InlineData(url: "https://localhost:8080/hook")]
[InlineData(url: "http://ip6-localhost/hook")]
[InlineData(url: "http://0.0.0.0/hook")]
YSecretY
public void TryCreate_WithBlockedHostname_ThrowsValidationException(string url)
{
    Assert.Throws<ValueObjectValidationException>(() => WebhookUrl.TryCreate(url));
}
```

Рисунок 3.24 – Приклад тестування SSRF

Тести Circuit Breaker використовують вбудований патерн FakeTimeProvider – внутрішній клас, що успадковує абстрактний TimeProvider

з методом AdvanceTo(DateTimeOffset). Це дозволяє детерміновано керувати часом у тестах без зовнішніх залежностей (див. рис. 3.25).

```
[Fact]
YSecretY
public void IsCallAllowed_WhenHalfOpen_ReturnsTrue()
{
    var cb = NewClosed();
    TripBreaker(cb);

    var afterCooldown = new FixedTimeProvider(cb.CooldownEndsAtUtc!.Value.AddSeconds(1));
    cb.IsCallAllowed(afterCooldown);

    Assert.Equal(CircuitState.HalfOpen, cb.State);
    Assert.True(cb.IsCallAllowed(afterCooldown));
}
```

Рисунок 3.25 – Керування часом у тесті CircuitBreaker

Тести інфраструктури (Aether.Delivery.Infrastructure.Tests) перевіряють HttpWebhookDispatcher за допомогою кастомних підкласів HttpMessageHandler. Тести верифікують коректність обчислення HMAC-підпису для трьох алгоритмів (SHA-256, SHA-384, SHA-512), наявність заголовка підпису, обробку HTTP-помилок, таймаут 30 секунд та запис латентності.

Інтеграційні тести використовують SQLite з базою даних в оперативній пам’яті (Data Source=:memory:) замість PostgreSQL. Кожен тестовий клас створює власний ізольований контекст через статичний метод Create(), що забезпечує повну незалежність тестів між собою.

Тести застосункового шару (Aether.Webhooks.Application.Tests, Aether.Delivery.Application.Tests) створюють реальні модульні класи (WebhookModule, DeliveryModule) з реальним EF Core контекстом на SQLite та підміненими зовнішніми залежностями через NSubstitute. Збірка Aether.Delivery.Application.Tests містить 28 тестів для DeliveryModule та окремі тести для фонових сервісів OutboxRelayWorker і RetrySchedulerWorker з повноцінним DI-контейнером (див. рис. 3.26).

```
[Fact]
YSecretY More...
public async Task ProcessBatch_WithLockedMessage_SkipsIt()
{
    using var ctx = DeliveryWorkerContext.Create();
    var msg :OutboxMessage = MakeMessage(ctx.UserId, scheduledAt: DateTimeOffset.UtcNow.AddSeconds(-1));
    msg.Lock(DateTimeOffset.UtcNow.AddSeconds(30));
    ctx.Db.OutboxMessages.Add(msg);
    await ctx.Db.SaveChangesAsync();

    await ctx.CreateRelayWorker().ProcessBatchAsync(CancellationToken.None);

    await ctx.Publisher.DidNotReceive() // IDeliveryMessagePublisher
        .PublishDispatchAsync( message: Arg.Any<DispatchWebhookMessage>(), cancellationToken: Arg.Any<CancellationToken>()); // Task
}
```

Рисунок 3.26 – Приклад інтеграційного тесту

Тести API-шару (Aether.Api.Tests) використовують WebApplicationFactory<Program> з підміною JWT-автентифікації кастомним TestAuthHandler, що зчитує ідентифікатор користувача із заголовка X-Test-UserId без реального JWT.

ExploratoryWebAppFactory запускає реальний ASP.NET Core застосунок з чотирма окремими SQLite in-memory базами даних (див. табл. 3.2).

Таблиця 3.2 – Покриття тестами

Категорія	**Збірки**	**Тести**	**Покриття**
Unit (domain)	2	~60	Domain
Unit(infrastructure)	1	~14	HTTP, HMAC
Integration	2	~55	Модулі
Integration (API)	1	~14	HTTP endpoints
Exploratory	1	~4	Повний стек

Рисунок 3.27 демонструє результати виконання тестового набору системи. Усі тести завершуються успішно, що підтверджує коректність реалізації доменної логіки, інфраструктурних компонентів та API-шару.

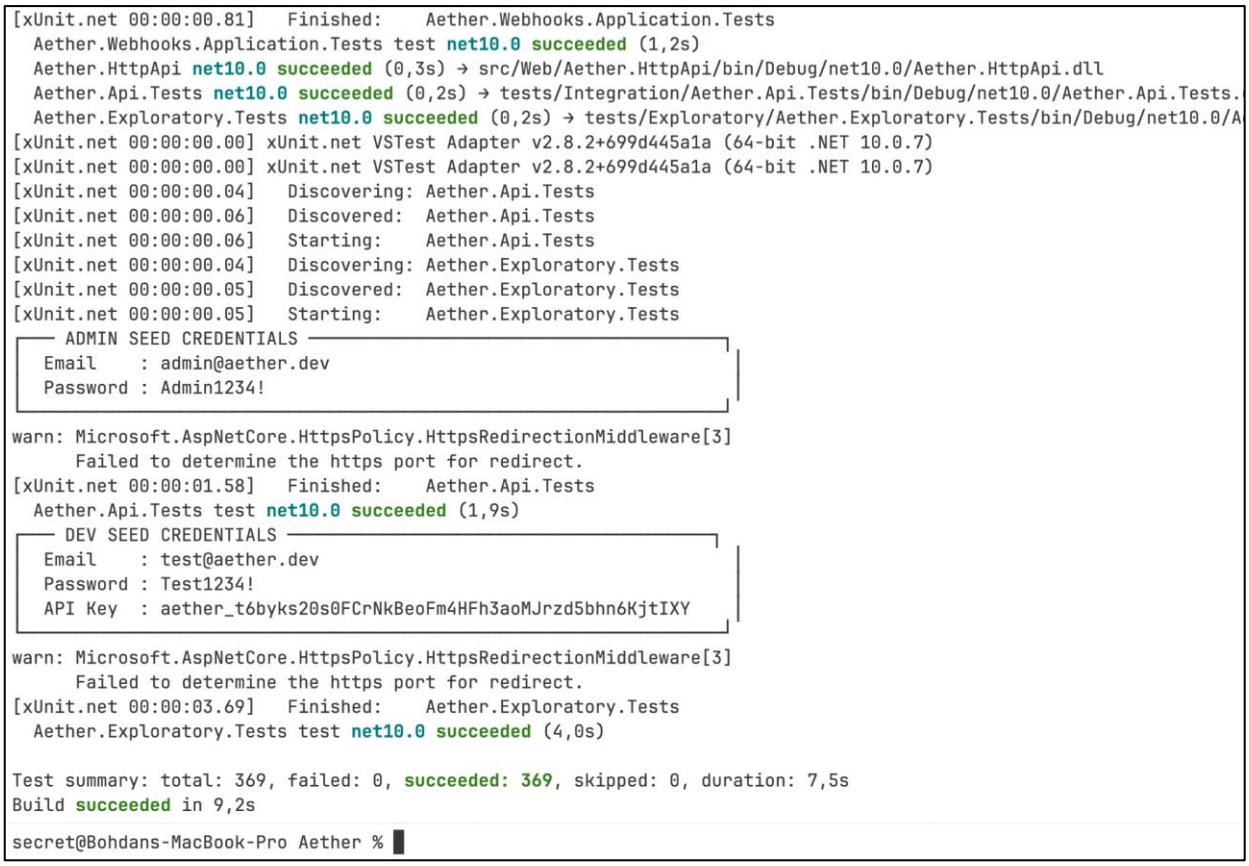


Рисунок 3.27 – Проходження тестів

Наявність багаторівневої тестової піраміди забезпечує впевненість у стабільності системи при подальшому розвитку та рефакторингу коду.

ВИСНОВКИ

У кваліфікаційній роботі розроблено та реалізовано систему управління вебхуками Aether – самостійно розгорнуваний програмний комплекс, що забезпечує надійне доставлення подій між розподіленими сервісами з повною спостережуваністю та гнучкою конфігурацією.

У першому розділі проведено аналіз предметної області вебхуків як механізму міжсервісної взаємодії, розглянуто існуючі комерційні та відкриті рішення. Встановлено, що наявні продукти або є хмарними сервісами з обмеженою гнучкістю розгортання, або не забезпечують повного набору функцій надійного доставлення. Обґрунтовано доцільність розробки власного самостійно розгорнуваного рішення.

У другому розділі спроєктовано архітектуру системи на основі принципів модульного монолі-ту з чотирма ізольованими обмеженими контекстами: Identity, Webhooks, Delivery та Analytics. Розроблено структуру доменної моделі, визначено технологічний стек та спосіб розгортання на базі Docker та Caddy.

У третьому розділі реалізовано всі компоненти системи:

– доменна модель побудована з використанням патернів Value Object та типобезпечних ідентифікаторів на основі readonly record struct, що унеможливлює некоректне використання примітивних типів у бізнес-логіці; реалізовано SSRF-захист на рівні доменного об’єкта WebhookUrl;
– шар доступу до даних реалізовано засобами Entity Framework Core з чотирма незалежними базами даних PostgreSQL; конфігурація сутностей виконана через IEntityTypeConfiguration з явними конвертерами значень для строго типізованих ідентифікаторів; індексування побудовано відповідно до конкретних запитів застосування;

- система надійного доставлення побудована на патерні Transactional Outbox із чергою повідомлень RabbitMQ; реалізовано патерн Circuit Breaker з персистованим станом та трьома стратегіями повтору (Exponential, Fixed, Linear); механізм песимістичного блокування записів забезпечує коректну роботу в багатоекземплярному розгортанні;
- автентифікація реалізована через два рівноправні механізми – JWT Bearer із 15-хвилинним токеном доступу та API-ключі зі зберіганням у вигляді SHA-256 хешу; інвалідація сесій реалізована через ротацію SecurityStamp без потреби у чорному списку токенів; паролі хешуються алгоритмом Argon2;
- обмеження частоти запитів реалізовано через шість іменованих політик із трьома алгоритмами (Fixed Window, Sliding Window, Token Bucket), підібраними відповідно до характеру навантаження кожної категорії маршрутів;
- тестування організовано у вигляді трирівневої піраміди: модульні тести доменних сутностей та інфраструктурних сервісів, інтеграційні тести застосункового та API-шарів на базі SQLite in-memory, а також дослідницькі наскрізні тести повного стека.

Практична цінність розробленої системи полягає в тому, що вона надає організаціям повний контроль над інфраструктурою доставлення подій без залежності від хмарних провайдерів. Система розгортається засобами Docker Compose, автоматично отримує TLS-сертифікати через Caddy та Let’s Encrypt, підтримує горизонтальне масштабування API-компоненти.

Усі поставлені завдання виконано в повному обсязі. Результати роботи оформлено у вигляді відкритого програмного забезпечення, розміщеного у публічному репозиторії.

ПЕРЕЛІК ПОСИЛАНЬ

1. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston : Addison-Wesley Professional, 2003. 560 p.

2. Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Boston : Prentice Hall, 2017. 432 p.

3. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley Professional, 2002. 560 p.

4. Fowler M. CircuitBreaker. *martinfowler.com*. URL: https://martinfowler.com/bliki/CircuitBreaker.html (дата звернення: 03.04.2026).

5. Fowler M. Transactional Outbox Pattern. *microservices.io*. URL: https://microservices.io/patterns/data/transactional-outbox.html (дата звернення: 03.04.2026).

6. Richardson C. Microservices Patterns: With examples in Java. Shelter Island : Manning Publications, 2018. 520 p.

7. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). RFC 7519. *Internet Engineering Task Force*. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 08.04.2026).

8. Krawczyk H., Bellare M., Canetti R. HMAC: Keyed-Hashing for Message Authentication. RFC 2104. *Internet Engineering Task Force*. URL: https://datatracker.ietf.org/doc/html/rfc2104 (дата звернення: 08.04.2026).

9. Biryukov A., Dinu D., Khovratovich D. Argon2: the memory-hard function for password hashing and other applications. University of Luxembourg. URL: https://www.password-hashing.net/argon2-specs.pdf (дата звернення: 09.04.2026).

10. Microsoft. ASP.NET Core documentation. *Microsoft Learn*. URL: https://learn.microsoft.com/en-us/aspnet/core (дата звернення: 11.04.2026).

ДОДАТОК А

Інструкція з розгортання системи Aether

А.1 Системні вимоги

Для розгортання системи необхідно:

– операційна система Linux (Ubuntu 22.04+, Debian 12+) або macOS;
– Docker Engine 24.0+ та Docker Compose 2.20+;
– зареєстроване доменне ім’я з DNS-записами, що вказують на сервер (для автоматичного отримання TLS-сертифіката).

А.2 Отримання вихідного коду

```
git clone https://github.com/YSecretY/aether.git
cd aether
```

А.3 Налаштування змінних середовища

Скопіювати файл прикладу та заповнити значення:

```
cp .env.example .env
```

Обов’язкові змінні у файлі .env:

```
DOMAIN=your-domain.com
POSTGRES_PASSWORD=strong_password_here
RABBITMQ_DEFAULT_USER=aether
RABBITMQ_DEFAULT_PASS=strong_password_here
JWT_SECRET_KEY=min_32_chars_secret_key_here
JWT_ISSUER=https://api.your-domain.com
```

JWT_AUDIENCE=https://your-domain.com
VITE_API_URL=https://api.your-domain.com

А.4 Запуск системи

docker compose -f docker-compose.prod.yml up -d

Після запуску Docker Compose автоматично:

– збере образи frontend (React + Nginx) та backend (.NET 10);
– запустить PostgreSQL з чотирма базами даних та RabbitMQ;
– запустить Caddy, який отримає TLS-сертифікат від Let’s Encrypt та налаштує HTTPS.

А.5 Перевірка стану

```
# Перевірка стану контейнерів
docker compose -f docker-compose.prod.yml ps
# Перегляд логів API
docker compose -f docker-compose.prod.yml logs api --tail=50
```

Після успішного запуску система доступна за адресами:
https://your-domain.com – вебінтерфейс;
https://api.your-domain.com/scalar/v1 – інтерактивна документація API.

А.6 Зупинка та видалення

```
# Зупинка без видалення даних
docker compose -f docker-compose.prod.yml down
# Зупинка з видаленням volumes (всі дані будуть втрачені)
docker compose -f docker-compose.prod.yml down -v
```