Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська дипломна робота

на тему: «Розробка програмного забезпечення для централізованого збору оповіщень з використанням хмарних технологій»

Виконав: студент IV курсу
групи 3ПІ-18б
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Третяк М.І.
(прізвище та ініціали)
Керівник: д.т.н., проф. каф. ПЗ Ліщинська Л.Б.
(прізвище та ініціали)
Рецензент: д.т.н., проф. каф. КН Іванчук Я.В.
(прізвище та ініціали)

Допущено до захисту
Зав. кафедри __________ О.Н. Романюк
«____» __________ 2022 р.

ВНТУ– 2022

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший бакалаврський
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О. Н.

____
"25" __березня__ 2022 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Третяк Мирославі Ігорівні

1. Тема роботи – «Розробка програмного забезпечення для централізованого збору оповіщень з використанням хмарних технологій»

Керівник роботи: Ліщинська Людмила Броніславівна, д.т.н., професор кафедри ПЗ, затверджені наказом вищого навчального закладу від "24" _березня_ 2022 року №66.

2. Строк подання студентом роботи _13 червня 2022 року_.

3. Вихідні дані до роботи: модель розробки – ітеративна; вхідні дані – назва сервісу, правила аналізу онлайн-ресурсів, дані облікового запису; вихідні дані – текстові оповіщення від зареєстрованих сервісів, текстовий звіт, зображення діаграми звіту; середовище розробки – JetBrains PyCharm; мова програмування - Python.

4. Зміст розрахунково–пояснювальної записки: вступ; обґрунтування вибору методу розробки та постановка задачі дослідження; розробка методів збору та обробки оповіщень; розробка структури і програмних компонентів; тестування програми; висновки; список використаних джерел; додатки.

5. Перелік графічного матеріалу: блок-схема алгоритму отримання оповіщень з використанням вебхуків та публічних API; блок-схема алгоритму отримання оповіщень з використанням парсингу веб-ресурсів; блок-схема алгоритму генерації звітів на основі зібраних оповіщень; структура графічного інтерфейсу головного вікна додатку; структура графічного інтерфейсу вікна створення джерела сповіщень; структура графічного інтерфейсу вікна звіту.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Ліщинська Л.Б., д.т.н., професор кафедри ПЗ		

7. Дата видачі завдання _25 березня 2022 року_

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз завдання і вибір методу вирішення поставленої задачі дослідження	26.03.2022 – 01.04.2022	Вик.
2	Розробка методу та алгоритму отримання оповіщень з використанням вебхуків та публічних API	02.04.2022 – 09.04.2022	Вик.
3	Розробка методу та алгоритму отримання оповіщень з використанням парсингу веб-ресурсів	10.04.2022 – 18.04.2022	Вик.
4	Розробка методу та алгоритму генерації звітів на основі зібраних оповіщень	19.04.2022 – 26.04.2022	Вик.
5	Аналіз і вибір мови програмування та середовища розробки	27.04.2022 – 01.05.2022	Вик.
6	Програмна реалізація додатку	02.05.2022 – 18.05.2022	Вик.
7	Тестування програмного забезпечення	19.05.2022 – 28.05.2022	Вик.
8	Оформлення матеріалів до захисту БДР	29.05.2022 - 10.06.2022	Вик.

Студент ________ **Третяк М.І.**
(підпис) (прізвище та ініціали)

Керівник бакалаврської дипломної роботи ________ **Ліщинська Л.Б.**
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська дипломна робота складається з 104 сторінок формату А4, на яких є 35 рисунків, 4 таблиці, список використаних джерел містить 33 найменування.

У бакалаврській дипломній роботі проведено детальний аналіз стану програмного забезпечення для централізованого збору оповіщень. Встановлено об’єкт, предмет, завдання та методи дослідження. Сформульовано мету дослідження – підвищення ефективності роботи користувача з вхідними повідомленнями та оповіщеннями шляхом автоматизації та централізації процесу їх отримання. Для реалізації мети було розроблено алгоритми роботи, інтерфейс та програмну реалізацію додатку.

Розроблено алгоритми отримання оповіщень з використанням вебхуків та публічних API і з використанням парсингу веб-ресурсів, які враховують технічні особливості джерел даних та дозволяють збільшити ефективність отримання й обробки даних для кінцевого користувача. Запропоновано алгоритм генерації звітів на основі зібраних оповіщень, який покращує наочність отриманих даних і надає декілька варіантів їх представлення.

Додаток розроблено за допомогою мови програмування Python, бібліотеки Boto 3 для роботи з AWS SDK, бібліотеки Beautiful Soup для парсингу онлайн-ресурсів та середовища розробки JetBrains PyCharm. Для розробки серверної частини програмного забезпечення використано сервіси хмарного провайдера AWS. В результаті виконання бакалаврської дипломної роботи розроблено програмний засіб, працездатність і правильність роботи якого перевірено, підготовлена інструкція користувача.

Отримані в бакалаврській дипломній роботі результати можна використати для побудови ефективної системи автоматизованого збору оповіщень.

Ключові слова: інформаційні оповіщення, хмарні технології, онлайн-ресурси.

ABSTRACT

The bachelor's thesis consists of 104 A4 pages, which have 35 figures, 4 tables, the list of sources used contains 33 items.

A detailed analysis of the state of the software for centralized collection of notifications was done in the bachelor’s thesis. The object, subject, tasks and methods of research are established. The objectives of the study were formulated - to improve the user's efficiency with incoming messages and alerts by automating and centralizing the process of their acquisition. To achieve the goal, the algorithms, interface, and software implementation of the application for centralized receipt of notifications were developed.

The algorithms for getting notifications using webhooks and public API and parsing web resources, which consider the technical features of the data sources and increase the efficiency of receiving and processing data for the end user, were developed. Proposed an algorithm for generating reports based on the collected notifications, which improves the visibility of the received data and provides several options for its presentation.

The created application for centralized collection of notifications using serverless cloud technologies was developed with the help of Python programming language, Boto 3 library for working with AWS SDK, Beautiful Soup library for parsing online resources and JetBrains PyCharm development environment. For the development of the server part of the software the services of cloud provider AWS were used. As a result of the bachelor's thesis a software tool, its operability and correctness has been tested, a user manual has been prepared.

The results obtained in the bachelor's thesis can be used to build an effective system of automated notification collection.

Key words: information notifications, cloud technologies, online resources.

ЗМІСТ

ВСТУП.. 8
1 ОБҐРУНТУВАННЯ ВИБОРУ МЕТОДУ РОЗРОБКИ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ 12
1.1 Аналіз технологій хмарних безсерверних розрахунків................................ 12
1.2 Порівняльний аналіз аналогів 14
1.3 Аналіз методів отримання оповіщень 19
1.4 Постановка задач дослідження 23
1.5 Висновки 24
2 РОЗРОБКА МЕТОДІВ ЗБОРУ ТА ОБРОБКИ ОПОВІЩЕНЬ 25
2.1 Розробка методу і алгоритму отримання оповіщень з використанням вебхуків та публічних API 25
2.2 Розробка методу і алгоритму отримання оповіщень з використанням парсингу веб-ресурсів 28
2.3 Розробка методу і алгоритму генерації звітів на основі попередньо зібраних оповіщень 31
2.4 Висновки 34
3 РОЗРОБКА СТРУКТУРИ І ПРОГРАМНИХ КОМПОНЕНТІВ........................ 35
3.1 Варіантний аналіз і обґрунтування вибору мови програмування................ 35
3.2 Вибір середовища розробки 38
3.3 Розробка структури програмного забезпечення 42
3.4 Розробка структури графічного інтерфейсу 45
3.5 Програмна реалізація додатку 48
3.6 Висновки 55
4 ТЕСТУВАННЯ ПРОГРАМИ 56
4.1 Аналіз методів тестування програмного забезпечення................................. 56
4.2 Тестування розробленого програмного продукту 57
4.3 Розробка інструкції користувача 63
4.4 Вимоги до персонального комп’ютера 66

4.5 Висновки ... 67
ВИСНОВКИ ... 68
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ... 69
ДОДАТКИ ... 72
Додаток А. Технічне завдання ... 73
Додаток Б. Лістинг програми ... 77
Додаток В. Ілюстративний матеріал ... 95

ВСТУП

Обґрунтування вибору теми дослідження. Інформаційне поле кожної людини лише збільшується з кожним роком, нові технології дозволяють значно швидше та в більших об’ємах дізнаватися новини, спілкуватися з друзями та близькими людьми або ж обмінюватися робочою і навчальною інформацією [1]. Розвиток різноманітних інтернет-сервісів і програмного забезпечення значно скоротив необхідні зусилля та час для розповсюдження даних, а різноманітні системи оповіщення, автоматичні розсилки та інші інструменти сфокусувалися на наданні якомога більш комфортного сервісу для користувачів. Тому зараз можливо за короткий час отримати повідомлення про події в іншій частині світу, а голосне сповіщення не дасть можливості це пропустити.

Такий функціонал створювався заради комфорту кінцевих користувачів та для популяризації власних сервісів, адже ресурси з більшою швидкістю розповсюдження інформації мають більші шанси на подальшу популяризацію та розширення аудиторії. Але фактичне впровадження систем оповіщень у велику кількість програмного забезпечення та сервісів призвело до нової задачі – більшість людей зараз отримує та аналізує інформацію з декількох джерел або просто може спостерігати за різними інформаційними сферами, що в будь-якому випадку приводить до великої кількості джерел різноманітних повідомлень.

Велика кількість оповіщень може бути безпечною відразу через декілька причин. По-перше, автоматичні розсилки зазвичай не мають представлення про зайнятість та розклад дня кінцевого користувача. Push-оповіщення під час робочих або навчальних дзвінків швидко можуть почати надокучати та відволікати від теми розмови. Крім того, постійні звукові та візуальні сигнали, які звикли сприймати в вигляді «з’явилася нова інформація – необхідно її перевірити» можуть значно зменшувати запас концентрації людини, що впливає на якість роботи й навчання [2].

Просте рішення відключити або обмежити потік оповіщень підходить не для усіх користувачів, хоча воно є більш корисним [3], але частина людей боїться

пропустити важливу інформацію, а для деяких працівників швидка відповідь на нові повідомлення або інші джерела даних є складовою частиною роботи, яку неможливо обмежити. Альтернативний варіант вирішення задачі – це скористатися програмним забезпеченням для збору оповіщень. Наразі існує певна кількість продуктів для вирішення цієї задачі, але всі вони мають свої недоліки та обмеження, як-от відсутність функціоналу для створення власних інтеграцій з сервісами та генерації звітів або ж обмежена підтримка десктопних ОС. Використання хмарних технологій у свою чергу дозволяє спростити проектування архітектури додатку, зменшити витрати на інфраструктуру та полегшити масштабування та подальшу підтримку ПЗ.

Тому актуальним є питання підвищення ефективності роботи користувачів з вхідними оповіщеннями шляхом розробки сучасного програмного забезпечення для централізованого збору оповіщень, яке може використовуватися у сфері персонального менеджменту людини.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалася згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання дослідження. Метою дослідження є підвищення ефективності роботи користувача з вхідними повідомленнями та оповіщеннями шляхом автоматизації та централізації процесу їх отримання.

Відповідно до поставленої мети в бакалаврській дипломній роботі потрібно вирішити такі **завдання**:

– провести аналіз сучасних технологій хмарних безсерверних розрахунків;

– провести аналіз існуючих методів отримання оповіщень;

– розробити метод і алгоритм отримання оповіщень з використанням вебхуків та публічних API;

– розробити метод і алгоритм отримання оповіщень з використанням парсингу веб-ресурсів;

– розробити метод і алгоритм генерації звітів на основі зібраних оповіщень;

– розробити структуру та графічний інтерфейс користувача для створюваного програмного продукту;

– розробити програмні компоненти додатку для централізованого збору оповіщень;

– розробити модульні тести для автоматизованого тестування;

– провести ручне тестування розробленого програмного забезпечення;

– розробити інструкцію користувача.

Об’єкт дослідження – процеси отримання, зберігання та надсилання оповіщень від різноманітних сервісів та інтернет-ресурсів.

Предмет дослідження – методи та засоби централізованого збору оповіщень.

Методи дослідження. У процесі дослідження використовувались: аналіз вхідних і вихідних даних, дослідження і порівняння існуючих технічних рішень задачі, синтез отриманої інформації для постановки задач і розробки власного рішення; комп’ютерне моделювання для перевірки та аналізу теоретичних положень.

Наукова новизна отриманих результатів:

– удосконалено метод отримання оповіщень з використанням вебхуків та публічних API, який на відміну від існуючих базується на зборі, аналізі та збереженні даних шляхом тісної інтеграції з підтримуваними сервісами, що дозволяє збільшити ефективність процесу збору оповіщень за допомогою автоматизації отримання повідомлень при мінімальній конфігурації з боку користувача;

– удосконалено метод отримання оповіщень з використанням парсингу веб-ресурсів, який на відміну від існуючих дозволяє отримувати дані від непідтримуваних сервісів та онлайн-ресурсів з використанням довільних правил обробки, що збільшує ефективність та розширює можливості процесу збору оповіщень;

– удосконалено метод генерації звітів на основі зібраних оповіщень, який на відміну від існуючих пропонує декілька варіантів відображення інформації у

текстовому та графічному режимі, що дозволяє підвищити наочність звіту та функціональність процесу збору й обробки даних.

Практична цінність отриманих результатів. Практична цінність одержаних результатів полягає в тому, що на основі отриманих в бакалаврській дипломній роботі теоретичних положень запропоновано алгоритми та розроблено програмні засоби для централізованого збору оповіщень з використанням хмарних безсерверних технологій.

Особистий внесок здобувача. Усі наукові результати, які викладені у бакалаврській дипломній роботі, отримано автором самостійно. У наукових працях, опублікованих у співавторстві, автору належать такі результати: використання хмарних технологій для автоматизації потоків даних та реалізації персональних оповіщень [4], розробка алгоритму аналізу онлайн-ресурсів для створення оповіщень [5].

Апробація результатів роботи. Результати роботи доповідалися на L науково-технічній конференції підрозділів Вінницького національного технічного університету (2021 р., м. Вінниця) та LI науково-технічній конференції підрозділів Вінницького національного технічного університету (2022 р., м. Вінниця).

Публікації. Основні результати дослідження опубліковано в 2 наукових працях у збірниках матеріалів конференцій.

1 ОБҐРУНТУВАННЯ ВИБОРУ МЕТОДУ РОЗРОБКИ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Аналіз технологій хмарних безсерверних розрахунків

Створення та підтримка сучасного програмного забезпечення вимагає зараз значно більших фінансових витрат, ніж у минулому. Це зумовлено не тільки необхідністю платити за офісну інфраструктуру та роботу працівників, а й пов’язано зі значно більшими витратами на цифрову інфраструктуру для проектів [6]. Компанії мають значну потребу в серверних ресурсах, які можуть бути задіяні при розробці продукту (різноманітні CI/CD-інструменти, середовища для тестування, сховища для артефактів та інших загальнодоступних файлів) та при його безпосередньому використанні (хостинг та різноманітні розрахунки для онлайн-ресурсів).

Раніше популярним рішенням для описаних задач було використання власних серверів або навіть дата-центрів для великих компаній, але зараз такий варіант поступово втрачає свою актуальність [7]. Основними проблемами самостійного хостингу («self-hosting») є наступне:

– додаткові часові витрати на підтримку апаратного й програмного забезпечення;

– більші початкові витрати в порівнянні з хмарними сервісами;

– потреба додатково персоналу з відповідними навичками для підтримки серверів;

– більш складне масштабування ресурсів, адже для розширення необхідна покупка додаткового апаратного забезпечення;

– більш складне ліцензіювання для окремих видів ПЗ (наприклад, існують ліцензії з комплексними схеми розрахунку вартості на основі параметрів наявних серверів).

Описані недоліки ставали дедалі більш критичними через зростання технічної складності апаратного та програмного забезпечення, а також через потреби в значно потужнішому та дорожчому обладнанні. В результаті,

з'явилися різноманітні хмарні провайдери, які вирішили частину описаних проблем та надали доступ до нових можливостей. До переваг хмарних розрахунків можна віднести [8]:

– зменшення часу на розгортання та операції з сервісами – сучасні технології віртуалізації дозволяють за лічені хвилини створити віртуальний сервер з необхідними характеристиками та операційною системою;

– зменшення фінансових витрат – для хмарних сервісів оплачується лише фактичне використання ресурсів, а не повна вартість задіяної інфраструктури;

– простота масштабування – доступ до значних розрахункових потужностей дозволяє практично будь-якому проекту швидко адаптуватися до підвищення навантаження та нових потреб;

– безпека даних – хмарні провайдери надають широкі можливості по додатковому захисту та шифруванню даних, адже від цього залежить довіра їх клієнтів;

– простота інтегрування з іншими інструментами;

– можливість змішаного використання хмарних та власних ресурсів (гібридна хмара);

– автоматичні оновлення ПЗ для сервісів, якими керує хмарний провайдер.

Також важливо відмітити простоту використання та можливості по міграції від власних систем до хмарних рішень. Досить швидко можна створити необхідну кількість віртуальних серверів, перенести на них частину цифрової архітектури в такому вигляді, як вона є на момент міграції, а вже пізніше зайнятися інтеграцією з більш складними хмарними сервісами, що дозволить ще більше зекономити на фінансових та часових витратах [9].

Крім хмарних розрахунків, додаткові сервіси можуть представляти з себе різноманітні інструменти для зберігання та реплікації даних, готові рішення для керування базами даних, системи для збору статистики та показників інших сервісів, інструменти для створення та налаштування віртуальних мереж і багато іншого. Простота інтеграції між цими застосунками та перенесення

відповідальності за оновлення і підтримку на хмарного провайдера робить використання хмарних сервісів вигідним для компаній.

Окремою гілкою розвитку хмарних систем є безсерверні («serverless») розрахунки. Вони додають ще один рівень абстракції для клієнта та прибирають необхідність самостійно масштабувати ресурси або керувати віртуальними серверами [10].

Прикладами сервісів для безсерверних розрахунків є AWS Lambda, Azure Functions або GCP Cloud Functions. Вони дозволяють відразу завантажити необхідний для виконання код, а питання масштабування ресурсів та менеджменту ПЗ, що лежить в основі, залишити на стороні хмарного провайдера. Відповідно до цього, модель оплати також спрощується: зазвичай враховується лише загальний час виконання та об’єм використаної ОЗП.

Варто відмітити, що використання безсерверних розрахунків потребує певних змін в архітектурі, адже такі додатки не можуть локально зберігати власний стан та для збереження даних потребують використання додаткових сервісів. Наприклад, для хмарного провайдера AWS це може бути DynamoDB – NoSQL база даних від Amazon, яка доступна у вигляді сервісу та може автоматично масштабуватися в залежності від навантаження. З іншого боку, використання архітектури без збереження стану («stateless architecture») дозволяє створювати легко масштабовані додатки та мікросервіси [11].

Таким чином, розглянуто сучасні тенденції при роботі з апаратної та програмною інфраструктурою, наведено переваги використання хмарних сервісів і розглянуто особливості безсерверних розрахунків.

1.2 Порівняльний аналіз аналогів

Додатки для централізованого збору оповіщень зазвичай вирішують одну конкретну проблему – перевантаження уваги людини через занадто великий потік інформації з різноманітних джерел. У сучасному цифровому світі прості телефонні дзвінки та SMS-повідомлення відійшли на другий план, а ми в безкінечному потоці отримуємо оповіщення про повідомлення в месенджерах

(велика кількість людей користується відразу декількома), нагадування з календаря або таск-менеджера, новий лист в електронній скриньці, оповіщення з робочого додатку або фітнес-трекера та багато всього іншого.

Такий потік різносторонньої інформації швидко вичерпує запас уваги людини та починає погано впливати на її концентрацію. Просте відключення оповіщень часто не є рішенням, адже користувачі бояться пропустити щось важливе, тому й з’явилося програмне забезпечення для централізованого збору оповіщень. Задача таких додатків – систематизувати вхідні дані і таким чином зменшити інформаційне навантаження на користувача.

Найбільш популярними додатками для збору оповіщень є:

– Shift;

– Pushover;

– Notistory.

Розглянемо більш детально обрані аналоги.

Shift – додаток для операційних систем Mac та Windows від Shift Technologies. На рисунку 1.1 наведено приклад інтерфейсу додатку.

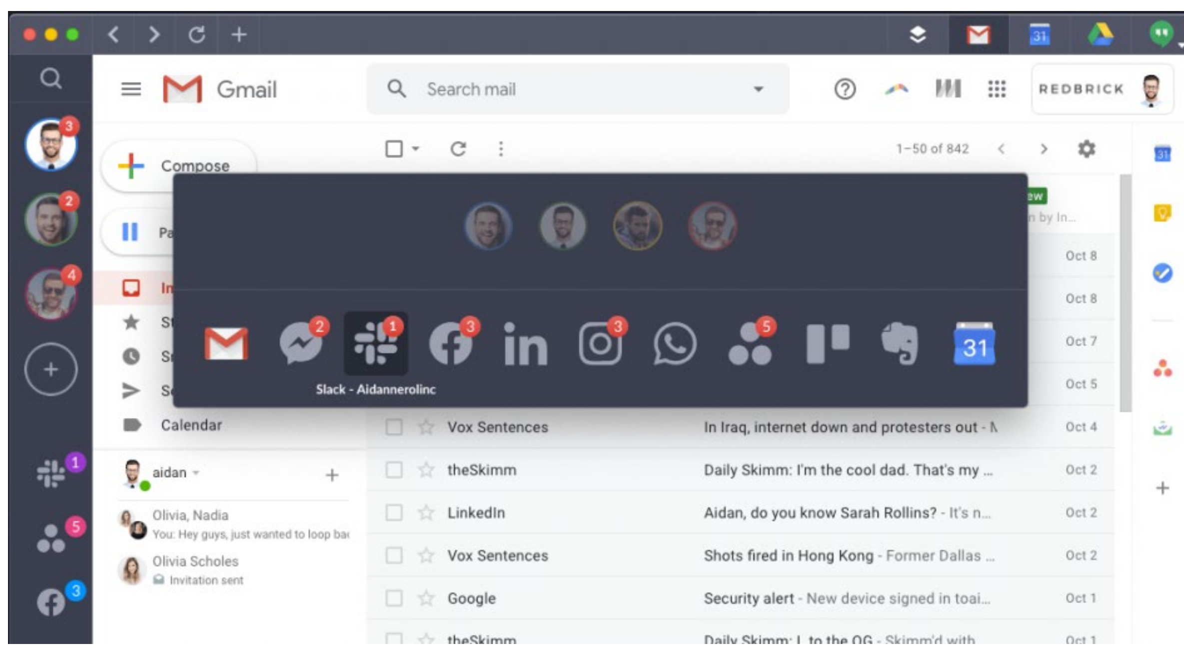


Рисунок 1.1 – Приклад інтерфейсу програми Shift

Shift позиціонується в якості програмного забезпечення для збору оповіщення та керування робочими додатками для підвищення продуктивності. Головними перевагами є обширний каталог готових інтеграцій з різноманітними сервісами, підтримка роботи з декількома акаунтами та можливість роботи з підтримуваними сервісами через веб-інтерфейс всередині самого додатку [12]. Основними недоліками є обмежена підтримка операційних систем та пристроїв для роботи, а також лімітований функціонал по роботі з оповіщеннями.

Pushover – популярний додаток для мобільних пристроїв від Pushover LLC. На рисунку 1.2 зображено приклад інтерфейсу програми.

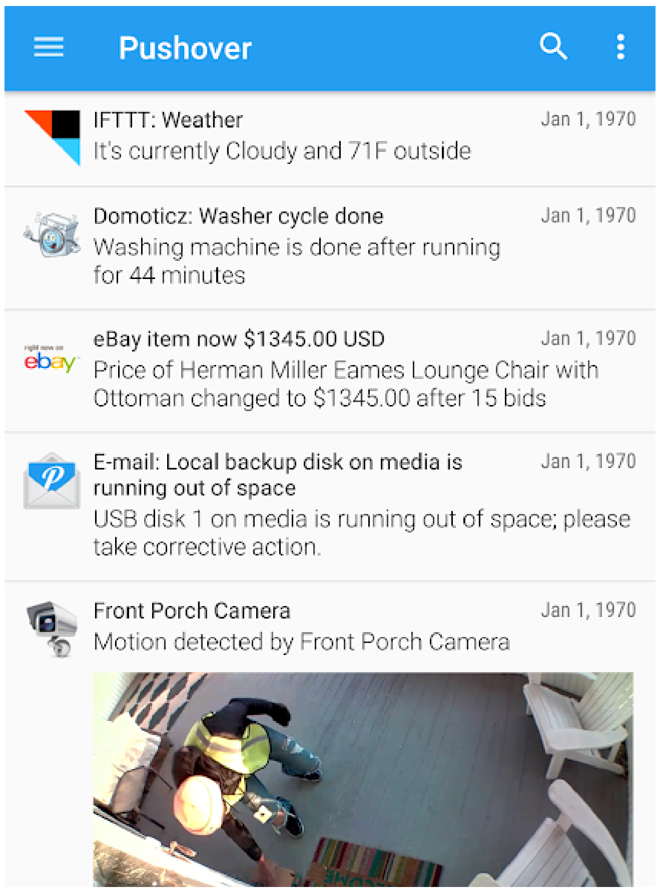


Рисунок 1.2 – Приклад інтерфейсу програми Pushover

Програмне забезпечення надає простий користувацький інтерфейс для перегляду отриманих оповіщень та дозволяє швидко розробляти власні інтеграції за допомогою публічного API. До переваг можна віднести простоту використання, вбудований функціонал для роботи в групах та обширний API, що

спрощує створення і підтримку інтеграцій для розробників сторонніх сервісів [13]. Важливими недоліками є малий каталог вже готових інтеграцій та орієнтація на користувачів мобільних пристроїв.

Notistory – мобільний додаток для централізованого збору оповіщень від whowhoLab. Приклад користувацького інтерфейсу наведено на рисунку 1.3.

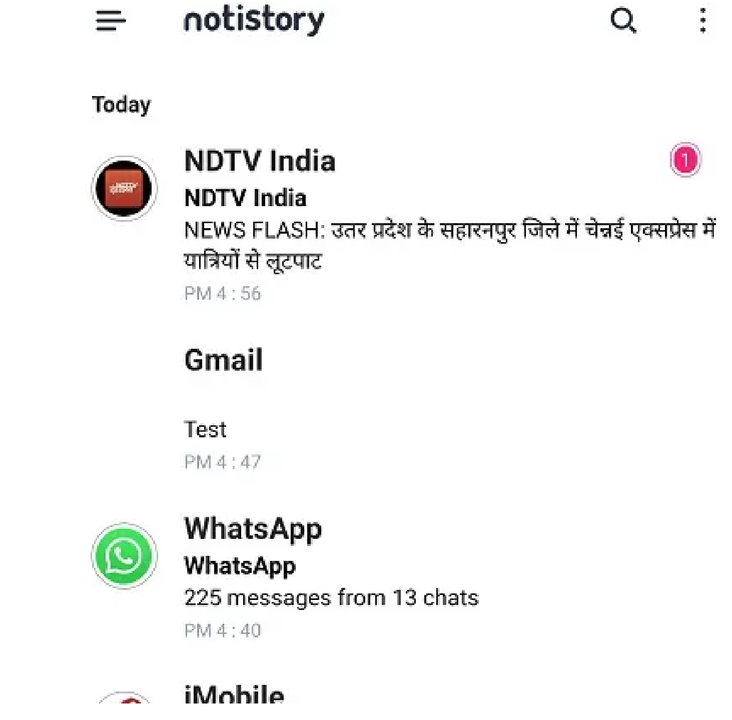


Рисунок 1.3 – Приклад інтерфейсу програми Notistory

Notistory займається керуванням оповіщень, що приходять на мобільний пристрій, тобто заміняє та покращує вбудований в ОС функціонал. Тому відсутня можливість отримання інформації від сторонніх сервісів (додатків, що не встановлені на смартфон, або ж веб-ресурсів), але наявний функціонал для менеджменту практично будь-яких системних оповіщень. Перевагами Notistory є широка інтеграція в операційну систему мобільного пристрою та можливість зберігання й подальшого пошуку по всім минулим оповіщенням [14]. Недоліками програми є робота лише з ОС Android та іншими системами, що базуються на його основі, а також відсутність підтримки сторонніх сервісів або веб-ресурсів.

Дослідження існуючих аналогів дозволило визначити переваги й недоліки в порівнянні зі створюваним програмних забезпеченням «Notification Collector». Результати аналізу наведені в таблиці 1.1.

Таблиця 1.1 – Порівняльні характеристики продуктів

Критерій	Shift	Pushover	Notistory	Notification Collector
Каталог готових інтеграцій	1	1	0,5	1
Створення власних інтеграцій	0	0,5	0	1
Підтримка десктопних ОС	1	0,5	0	1
Підтримка мобільних ОС	0	1	1	0
Генерація звітів на основі зібраних оповіщень	0	0	0,5	1
Підсумок	2	3	2	4

Розглянемо більш детально критерії оцінювання програмних продуктів.

Каталог готових інтеграцій – це список сервісів, з якими додаток може працювати відразу після встановлення, з мінімальним додатковим налаштуванням з боку користувача. В цій категорії лише Notistory отримує 0,5 балів через те, що програма передбачає роботу з оповіщеннями самої операційної системи.

Створення власних інтеграцій – це наявність функціоналу для підключення сторонніх сервісів до продукту. Наприклад, збирання оповіщень з маловідомого веб-ресурсу, який не входить в каталог підтримуваних інтеграцій. З уже існуючих аналогів лише Pushover має таку можливість, але при цьому вимагає знань розробки ПЗ для використання API та має певні обмеження у вигляді використання лише «Push» архітектури та обмеження кількості звернень до API.

Підтримка десктопних та мобільних ОС визначає на яких пристроях буде можливо використовувати програмний продукт. В цій категорії Pushover отримує 0,5 балів через те, що продукт передбачає роботу через веб-браузер, а не використання повноцінного додатку.

Генерація звітів передбачає подальшу обробку зібраних оповіщень. За цим критерієм Notistory не передбачає прямого створення звітів, але надає функціонал для зберігання старих оповіщень, їх сортування та пошуку, за що отримує 0,5 балів.

Таким чином, дослідження і порівняння існуючих аналогів дозволило визначити переваги й недоліки популярних рішень та проаналізувати за певними критеріями наявний функціонал. Результати аналізу доводять актуальність розробки власного програмного продукту.

1.3 Аналіз методів отримання оповіщень

Додатки для централізованого збору оповіщень використовують різні методи для отримання вхідних даних. Реалізація та технічна складність цих методів залежить від типу сервісу та відкритості доступу до нього. Часто використовують наступні варіанти:

– інтеграція з операційною системою пристрою;

– використання офіційного API для додатку або сервісу;

– перехоплення оповіщень або інших даних сторонніми засобами;

– аналіз або парсинг веб-ресурсів;

– використання вебхуків;

– створення власного сервісу оповіщень на основі шаблону «Публікація-підписка».

Розглянемо більш детально особливості кожного методу.

Інтеграція з ОС – спосіб, який використовується в Notistory та багатьох інших додатках для мобільних пристроїв. Він полягає в повній або частковій заміні функцій операційної системи по обробці вхідних оповіщень. Найбільш популярний приклад використання – це зміна логіки отримання оповіщень та інтерфейсу, а саме відмова від системного менеджера повідомлень на догоду власному GUI.

Програмне забезпечення на основі цього методу фактично виступає додатковим прошарком між ОС та кінцевим користувачем, тому обмежене

функціоналом самої системи та надає лише можливості постобробки даних. Таким чином, ОС Windows довгий час не мала підтримки сповіщень (до релізу Windows 10). Через це додатки з їх реалізацією поки не набули широкого розповсюдження, а продукти для збору оповіщень з використанням цього методу малопопулярні на десктопних системах. З іншого боку, програмне забезпечення для мобільних пристроїв навпаки часто використовує системні засоби для оповіщення користувача, через що даний метод є популярним на ОС Android, iOS та навіть macOS (через щільну інтеграцію зі смартфонами).

Значними недоліками при інтеграції з ОС залишаються обмеженість роботи системних оповіщень (лише встановлені додатки можуть надсилати повідомлення в систему) та потенційне дублювання даних на пристрої (одночасне зберігання і в самій ОС, і в розробленому додатку).

Використання офіційного API – це найпростіший спосіб інтеграції зі сторонніми сервісами для отримання повідомлень від них. Складність реалізації цілком залежить від доступних програмних бібліотек та їх технічних вимог [15]. Недоліком такого методу може стати складність підтримки створюваного додатку при великій кількості інтеграцій, адже кожен сервіс має свій власний спосіб підключення та роботи. Це не тільки значно збільшує кодову базу проекту, а й може збільшити навантаження на сервери додатку або пристрої користувача.

Перехоплення оповіщень або інших даних сторонніми засобами є протилежним до попереднього методом. Найчастіше він використовується для обмеженого ряду сервісів, які не надають доступ до API або ж забороняють отримання даних за допомогою інших методів через певну корпоративну політику. Збір оповіщень за допомогою сторонніх засобів може бути технічно складним та суперечити правилам й угодам користування певних сервісів, тому використовується рідко й зазвичай в неофіційних додатках.

Аналіз або парсинг веб-ресурсів – популярний спосіб отримання інформації з сайтів та інших платформ, які не надають інші методи програмного доступу. Наприклад, є сайт з новинами, ми хочемо отримувати сповіщення про

нові публікації, але ресурс не надає ніяких засобів для цього. В такому випадку можна скористатися різноманітними бібліотека для парсингу – автоматичного процесу вилучення великих об’ємів даних з веб-ресурсів. Технічна складність залежить від обраних мов програмування та бібліотек, хоча зазвичай їх використання намагаються максимально спростити. Крім того, підтримка великої кількості інтеграцій для збору сповіщень може бути проблемною через відмінності в структурах багатьох веб-ресурсів.

Всі попередні методи для збору інформації від сервісів використовують pull-архітектуру, але останні два базуються на push-архітектурі та мають інші принципи роботи. Для кращого розуміння розглянемо відмінності цих двох програмних архітектур. Схему взаємодії між пристроями для різних архітектур наведено на рисунку 1.4.

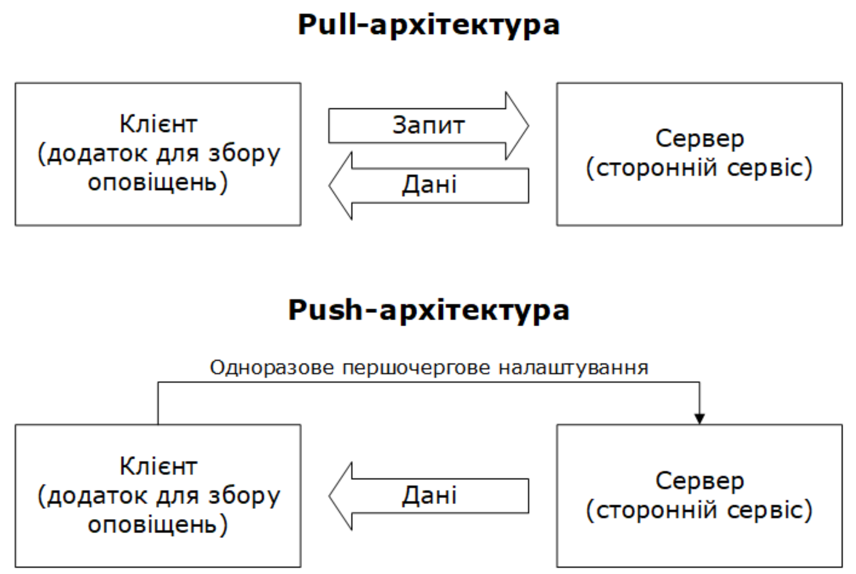


Рисунок 1.4 – Взаємодія пристроїв при використанні push та pull архітектур

Pull-архітектура передбачає формування запиту до сервера та отримання відповіді з необхідними даними на нього. Таким чином, для збору оповіщень зі стороннього сервісу додаток повинен надіслати запит до нього та обробити отриману відповідь. Для підтримання актуальності даних запити необхідно надсилати з певною періодичністю і клієнтський додаток наперед не знає чи доступні від сервісу нові повідомлення [16]. Тобто, програмне забезпечення буде

регулярно «опитувати» інші сервери на предмет наявності нових даних для нього, що підвищує навантаження на всі елементи такої мережі.

Push-архітектура вирішує проблему постійних запитів і працює за допомогою іншої концепції: коли відбувається певна подія або виконується попередньо визначена умова, то сервер самостійно надішле необхідні дані до клієнта. Це вимагає додаткової «реєстрації» клієнта на сервері, щоб він знав куди саме й яку інформацію необхідно передавати.

Прикладом використання push-архітектури є вебхуки. Додаток для збору оповіщень може за допомогою API або інших інструментів створити вебхук для певних подій на сторонньому сервісі й очікувати надходження даних [4]. Коли відповідна подія станеться – сервер самостійно надішле все необхідне в додаток, за яким залишається лише обробка отриманого повідомлення. Наприклад, сервіс електронної пошти Google Gmail з використанням вебхуків може надіслати HTTP-запит на стороннє зареєстроване програмне забезпечення при надходженні нових листів або інших подіях. Таким чином можна створити додаток, який буде створювати оповіщення про нові листи навіть без прямого доступу до веб-інтерфейсу сервісу або інших засобів. Недоліком є лише відсутність уніфікації API для роботи з вебхуками у різних ресурсів.

Останній метод збору оповіщень є більш комплексним та технічно складним – це створення власного сервісу оповіщень на основі шаблону «Публікація-підписка». Програмне забезпечення може надавати свій власний API для створення оповіщень, який інші сервіси будуть використовувати для публікації інформації, а на стороні додатку залишаються широкі можливості по подальшій обробці отриманих даних. Тобто, механізм схожий до роботи вебхуків, але процес реєстрації перенесено з боку сторонніх сервісів до клієнтського додатку. За таким принципом працює продукт Pushover, для якого доступно декілька бібліотек на різних мовах програмування, які розробники можуть використовувати в своїх сервісах для надсилання оповіщень. Основна проблема даного методу – це необхідність підтримки з боку сторонніх сервісів. Якщо необхідно отримати сповіщення про новий лист в скриньці – розробник

поштового сервісу повинен самостійно додати підтримку нашого ПЗ для збору оповіщень, що може бути нерелевантно для малопопулярних додатків через відсутність необхідної репутації та поширеності серед кінцевих користувачів.

Отже, було розглянуто шість різних способів взаємодії зі сторонніми ресурсами для збору оповіщень. Розглянуто технічні особливості кожного з них, визначено певні недоліки. Для розробки власного додатку вирішено використовувати відразу декілька методів: вебхуки та публічний API для підтримуваних сервісів та парсинг для довільних веб-ресурсів. Комбінація декількох варіантів дозволить створити більш гнучке та ефективне програмне забезпечення з широкими можливостями по інтеграції з різноманітними сервісами.

1.4 Постановка задач дослідження

Проаналізувавши переваги та недоліки існуючих додатків для збору оповіщень і дослідивши технічні особливості варіантів реалізації та стан хмарних технологій було визначено наступні задачі по розробці власного програмного продукту:

– провести аналіз сучасних технологій хмарних безсерверних розрахунків;

– провести аналіз існуючих методів отримання оповіщень;

– розробити метод і алгоритм отримання оповіщень з використанням вебхуків та публічних API;

– розробити метод і алгоритм отримання оповіщень з використанням парсингу веб-ресурсів;

– розробити метод і алгоритм генерації звітів на основі зібраних оповіщень;

– розробити структуру і графічний інтерфейс користувача для створюваного програмного продукту;

– розробити програмні компоненти додатку для централізованого збору оповіщень;

– розробити модульні тести для автоматизованого тестування;

– провести ручне тестування розробленого програмного забезпечення;

– розробити інструкцію користувача.

Технічне завдання на розробку наведено в додатку А.

Функціональні вимоги:

– користувач має змогу додати джерело оповіщення з підтримкою вебхуків;

– користувач має змогу додати джерело оповіщень з підтримкою парсингу онлайн-ресурсів;

– користувач має змогу переглянути зібрані оповіщення;

– користувач має змогу згенерувати звіт по зібраним оповіщенням.

Нефункціональні вимоги:

– підтримка ОС Windows 10, macOS Monterey, Ubuntu 20.04;

– клієнт-серверна взаємодія – через REST API;

– передача даних за допомогою захищеного протоколу HTTPS.

1.5 Висновки

У першому розділі було проаналізовано сучасний стан хмарних технологій, розглянуто особливості самостійного та хмарного хостингу програмних продуктів, наведено можливості та переваги сервісів для безсерверних розрахунків. Було проведено порівняльний аналіз аналогів, описано критерії оцінювання та досліджено переваги й недоліки популярних продуктів. Розглянуто методи розв’язання поставлених задач, досліджено їх технічні особливості та можливі недоліки, визначено методи, які можуть бути використані в розробці власного продукту. На основі отриманої інформації було сформовано перелік задач, які необхідно виконати для розробки власного програмного забезпечення для централізованого збору оповіщень з використанням хмарних безсерверних технологій. Розроблено постановку задачі та сформовано функціональні й нефункціональні вимоги.

2 РОЗРОБКА МЕТОДІВ ЗБОРУ ТА ОБРОБКИ ОПОВІЩЕНЬ

2.1 Розробка методу і алгоритму отримання оповіщень з використанням вебхуків та публічних API

В підрозділі 1.3 було розглянуто існуючі методи для збору оповіщень від сторонніх сервісів. І один з цих методів – це використання офіційних підтримуваних інструментів для збору даних, в якості яких можуть виступати вебхуки та API.

Запропонований метод отримання оповіщень з використанням вебхуків та публічних API базується на тісній взаємодії з сервісами, які дозволяють надсилати певні оновлення даних у вигляді повідомлень до сторонніх додатків. У випадку з вебхуками сервіси зазвичай оперують HTTP-запитами з повідомленнями в форматі JSON. Відповідно, серверна частина створюваного додатку повинна мати URL-адресу, на яку будуть приходити дані, а також включати в себе їх подальшу обробку. На відміну від існуючих методів передбачається використання єдиної URL-адреси та попередньо визначених правил для обробки оповіщень, так як їх структура може відрізнятися в залежності від сервісу й необхідно привести їх в один формат для подальшого відображення користувачеві та використання в інших методах. Повідомлення в форматі JSON представляють з себе комплексні об’єкти в вигляді «ключ-значення», що також підтримують вкладені структури, тому в якості формату опису правил обробки обрано запис шляху до необхідного ключа об’єкту. В результаті, запропонований метод дозволяє ефективно обробляти й зберігати вхідні повідомлення від сторонніх сервісів з підтримкою вебхуків.

Відповідно до цього було розроблено блок-схему алгоритму запропонованого методу, що наведена на рисунку 2.1. Розглянемо більш детально кроки алгоритму та принципи його роботи.

Крок 1. Початок алгоритму.

Крок 2. Функція в AWS Lambda отримує виклик від API Gateway зі стандартними вхідними змінними event і context, які мають всю необхідну

інформацію про вебхук або виклик API й відповідне оповіщення, а також додаткові відомості про середовище виконання.

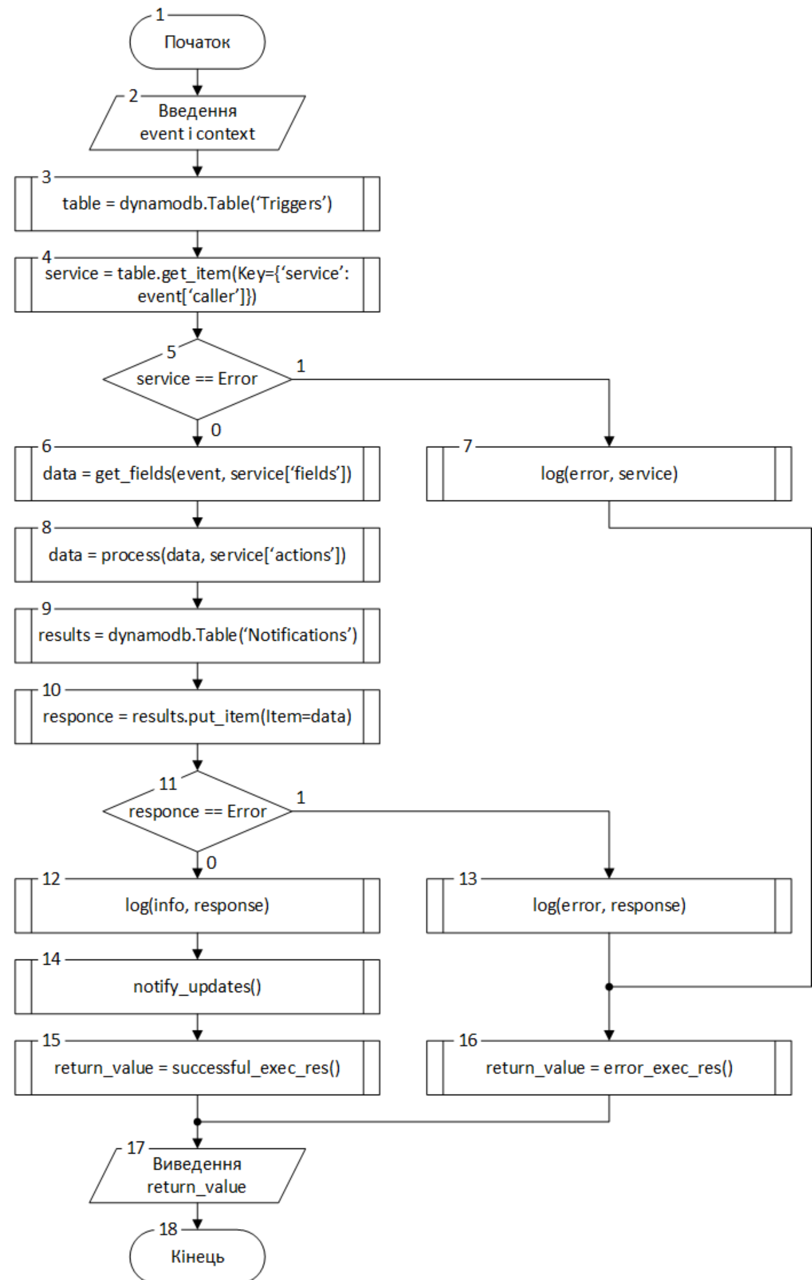


Рисунок 2.1 – Блок-схема алгоритму отримання оповіщень з використанням вебхуків та публічних API

Крок 3. Створюється об'єкт для роботи з таблицею бази даних, яка містить відомості про підтримувані сторонні сервіси та способи їх обробки.

Крок 4. Виконується пошук та отримання запису з таблиці. Необхідно дістати інформацію саме про той сервіс, від якого було отримано оповіщення.

Крок 5. Якщо при пошуку запису в таблиці виникла помилка – відбувається перехід до кроку 7. Інакше – перехід до кроку 6.

Крок 6. За допомогою функції get_fields відбувається добування необхідних даних з вхідної змінної event. Набір даних може бути різний для кожного веб-ресурсу або додатку, тому функція також вимагає інформацію про необхідний список полів з отриманого запису таблиці всіх сервісів.

Крок 7. Інформація про помилку, яка могла виникнути на кроці 5, записується в лог, після чого відбувається перехід на крок 16.

Крок 8. Після отримання даних з вхідної події може бути необхідна їх додаткова обробка. Для цього викликається функція process, яка перевіряє чи визначені в записі для поточного сервісу додаткові дії та за потреби виконує їх.

Крок 9. Створюється об’єкт для роботи з таблицею, яка містить вже оброблені оповіщення, що можна надсилати користувачу в додаток.

Крок 10. Оброблені дані оповіщення зберігаються в таблиці.

Крок 11. Якщо виникла помилка запису – відбувається перехід на крок 13. Інакше – перехід на крок 12.

Крок 12. Інформація про успішну обробку вебхука з оповіщенням записується в лог.

Крок 13. Інформація про помилку, яка могла виникнути на кроці 11, записується в лог, відбувається перехід до кроку 16.

Крок 14. Викликається додаткова функція notify_update, що сповіщає про закінчення обробки оповіщення та успішних запис даних в БД. Після цього в AWS Lambda може початись виклик іншої функції для надсилання даних в клієнтський додаток або ж акумулювання даних для генерації звітів, ці події залежать від персональних налаштувань користувача.

Крок 15. В змінну return_value за допомогою функції записується додаткова інформація про успішне отримання оповіщення.

Крок 16. Після логування помилок в return_value записується інформація про неуспішну обробку оповіщення та помилки, які виникли в процесі.

Крок 17. Відбувається виведення return_value, що сповіщає AWS Lambda про кінець виконання функції.

Крок 18. Кінець алгоритму.

Таким чином, в даному підрозділі було розглянуто блок-схему алгоритму отримання оповіщень з використанням вебхуків та публічних API. За допомогою псевдокоду та абстракцій, якими оперує сервіс AWS Lambda, було наведено принципи дії алгоритму. Детально описано кожен крок виконання.

2.2 Розробка методу і алгоритму отримання оповіщень з використанням парсингу веб-ресурсів

Наступний варіант отримання оповіщень – це парсинг веб-ресурсів. З технічної точки зору це є методом генерації оповіщень, а не їх обробки з інших джерел, так як відбувається збір інформації зі сторонніх сервісів, її обробка і порівняння з уже збереженими відомостями, після чого є можливість відправити отримані результати в клієнтський додаток.

Запропонований метод базується на технологія вебскрапінгу – перетворення у структуровані дані інформації з сторінок, які призначені для перегляду людиною за допомогою браузера. На відміну від існуючих методів передбачається використання попередньо описаних правил парсингу онлайн-ресурсів. Метод складається з трьох основних етапів: отримання даних сторінки, їх фільтрація на основі наданих правил та подальша обробка зі збереженням готових оповіщень. Найпростіший спосіб збору інформації з онлайн-ресурсів – це перегляд і обробка їх HTML-коду. Відповідно до цього в якості правил парсингу даних обрано CSS-селектори. Подальша обробка зібраних відомостей включає в себе формування тексту оповіщення. В результаті, метод дозволяє отримувати й зберігати вхідні повідомлення від онлайн-ресурсів з використанням довільних правил обробки від користувача.

Таким чином, для методу отримання оповіщень з використання парсингу веб-ресурсів було розроблено блок-схему алгоритму, що зображена на рисунку 2.2.

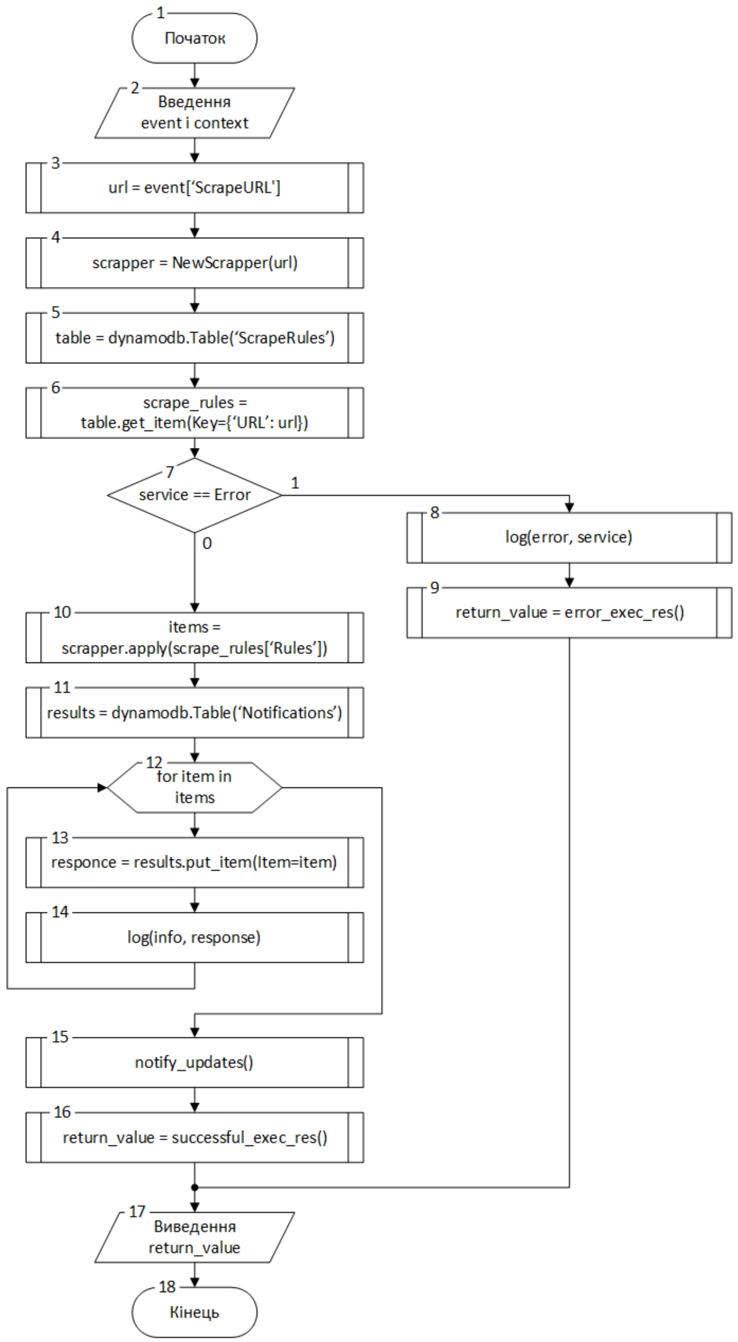


Рисунок 2.2 – Блок-схема алгоритму отримання оповіщень з використанням парсингу веб-ресурсів

Наприклад, цей алгоритм може використовувати для створення оповіщень про нові статті на улюбленому технічному порталі користувача. В такому випадку необхідно знати лише адресу сторінки, яка містить загальний список публікацій, та набір правил, які можна передати парсеру, щоб отримати цей список в зручному для обробки форматі [5]. Регулярно запускаючи процес парсингу та маючи результати попередніх спроб можна кожного разу легко

дізнатися чи список статей оновився і чи треба надіслати користувачу нове оповіщення. Розробка універсального алгоритму для цього процесу дозволяє використовувати підхід для практично будь-яких веб-ресурсів.

Розглянемо більш детально кроки розробленого алгоритму.

Крок 1. Початок алгоритму.

Крок 2. Введення всіх необхідних даних через змінні event та context, що автоматично генеруються при надходженні запитів в AWS Lambda.

Крок 3. З вхідних даних отримується адреса сторінки, з якої необхідно дістати інформацію і згенерувати оповіщення.

Крок 4. Створюється об'єкт-парсер на основі отриманої адреси. В цей момент в об'єкт також завантажується вміст відповідної сторінки.

Крок 5. Створюється об'єкт для доступу до таблиці БД зі збереженими правила для веб-парсера.

Крок 6. На основі адреси сторінки виконується спроба отримати відповідні правила обробки.

Крок 7. Якщо при отриманні правил виникла помилка – відбувається перехід до кроку 8. Інакше – перехід до кроку 10.

Крок 8. Інформація про помилку записується в лог.

Крок 9. В змінну return_value, що зберігає загальний результат виконання, записується інформація про неуспішне завершення операції.

Крок 10. Парсер за допомогою набору правил дістає необхідні дані та повертає їх у вигляді масиву, який зберігається у змінну items.

Крок 11. Створюється об'єкт для доступу до таблиці БД з готовими для користувача сповіщеннями.

Крок 12. Розпочинається цикл, який перебирає вміст масиву items. В даному контексті кожен елемент масиву – це окрема порція даних і нове сповіщення. Виконання тіла циклу розпочинається на кроці 13, закінчення перебору викликає перехід до кроку 15.

Крок 13. Поточний елемент масиву, тобто нове оповіщення, записується до таблиці.

Крок 14. Результат запису в таблицю зберігається в лог. Додаткової обробки помилок немає, так як може відбуватися робота з великим об’ємом даних. І успішні записи, і помилкові будуть збережені й доступні для можливого налагодження процесу.

Крок 15. Після запису всієї інформації відбувається додатковий виклик функції notify_update, що може бути використано для подальшої обробки оповіщень в клієнтському додатку.

Крок 16. В змінну return_value записується інформація про успішне виконання парсингу веб-ресурсу.

Крок 17. Відбувається виведення змінної return_value.

Крок 18. Кінець алгоритму.

Таким чином, було розглянуто блок-схему розробленого алгоритму для отримання оповіщень з використанням парсингу веб-ресурсів. Детально описано кожен крок виконання, розглянуто абстрактні етапи роботи з парсером та базою даних, які можуть допомогти в подальшій розробці програмного продукту.

2.3 Розробка методу і алгоритму генерації звітів на основі попередньо зібраних оповіщень

Користувачу може знадобитися не тільки простий збір усіх оповіщень з доданих сервісів та ресурсів, а й їх додаткова агрегація та аналіз. Згенерований звіт може допомогти краще зрозуміти потік щоденних вхідних даних користувача, тому варто виділити таку можливість в якості окремої функції продукту.

Запропонований метод удосконалює базові принципи генерації звіту шляхом відображення декількох різних представлень даних. Метод складається з двох основних етапів: простої обробки даних та відображення результатів. Обробка включає в себе отримання цільового списку оповіщень, їх сортування за категоріями (назвами сервісів) і підбиття підсумків за цими категоріями. Представлення даних на відміну від існуючих методів включає в себе три елементи: графічну діаграму, яка дозволяє легко зрозуміти співвідношення

кількості оповіщень від різних сервісів, короткий текстовий опис найбільш часто використовуваних сервісів, який може доповнювати графічне наповнення звіту, а також загальний список усіх оброблених оповіщень, що дозволяє більш детально дослідити оброблені дані.

Для процесу генерації звітів на основі зібраних оповіщень було розроблено алгоритм, який зображено на рисунку 2.4.

Розглянемо більш детально кроки виконання блок-схеми алгоритму.

Крок 1. Початок алгоритму.

Крок 2. Введення вхідних даних в форматі AWS Lambda, через змінні event та context. Для даного алгоритму ці змінні містять налаштування процесу генерації.

Крок 3. Зі змінної event отримується набір фільтрів для генерації.

Крок 4. Зі змінної event отримується ліміт кількості оповіщень.

Крок 5. Створюється об’єкт для роботи з таблицею сповіщень.

Крок 6. Здійснюється запит до таблиці, в якості фільтрів та ліміту максимальної кількості повідомлень використовуються вхідні дані алгоритму.

Крок 7. Якщо за результатами виконання запиту не було отримано жодного сповіщення для аналізу, то виконується перехід до кроку 17.

Крок 8. Створюється окремий об’єкт для підсумування кількості оповіщень для кожного сервісу або веб-ресурсу.

Крок 9. Розпочинається цикл, який перебирає усі сповіщення з отриманого набору даних. Виконання тіла циклу починається на кроці 10, а закінчення перебору – перехід до кроку 12.

Крок 10. Запис з даними сповіщення може містити великий набір додаткових відомостей, тому за допомогою окремої функції getService необхідно дістати лише назву сервісу для поточного повідомлення.

Крок 11. Отримана назва сервісу або веб-ресурсу для сповіщення додається в об’єкт-лічильник.

Крок 12. На основі всіх зібраних даних з об’єкта-лічильника будується діаграма.

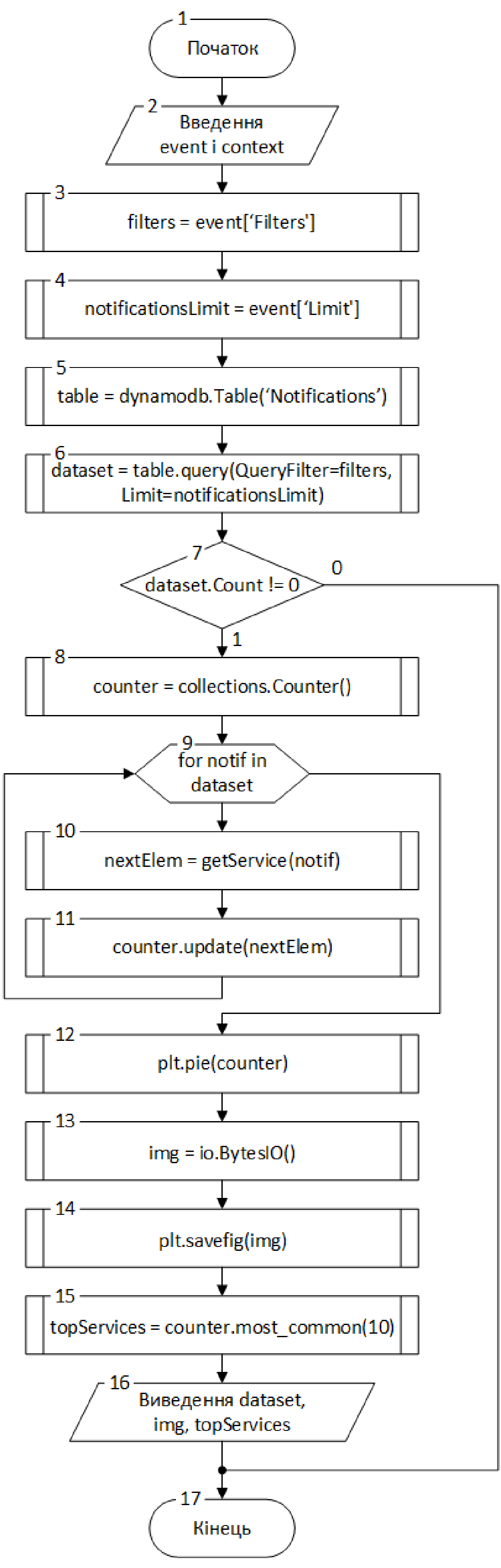


Рисунок 2.4 – Блок-схема алгоритму генерації звітів

на основі зібраних оповіщень

Крок 13. Для подальшої роботи зі згенерованою діаграмою створюється буфер байтів.

Крок 14. В буфер зберігається діаграма зі статистикою оповіщень.

Крок 15. В окрему змінну записується інформація про десять сервісів з найбільшою кількістю сповіщень.

Крок 16. Відбувається виведення вмісту буфера з зображенням, списку сервісів, які мають найбільше повідомлень, а також додатково виводиться весь отриманий набір даних, який може знадобитися для детального перегляду статистики в клієнтському додатку.

Крок 17. Кінець алгоритму.

Отже, в даному підрозділі було розглянуто розроблений алгоритм генерації звітів на основі зібраних оповіщень. Наведено блок-схему алгоритму, розглянуто кроки його виконання і роботу зі збереженими даними.

2.4 Висновки

У другому розділі було детально розглянуто й описано методи і алгоритми, що будуть використані при розробці додатку, а саме: метод і алгоритм отримання оповіщень з використанням вебхуків та публічних API; метод і алгоритм отримання оповіщень з використанням парсингу веб-ресурсів; метод і алгоритм генерації звітів на основі зібраних оповіщень. Для кожного з них розроблено блок-схему, наведено деталі виконання кожного кроку виконання.

3 РОЗРОБКА СТРУКТУРИ І ПРОГРАМНИХ КОМПОНЕНТІВ

3.1 Варіантний аналіз і обґрунтування вибору мови програмування

Вибір мови програмування при розробці додатків, що використовують хмарні сервіси, є важливим аспектом загальної швидкодії продукту та має помітний вплив на фінансові витрати за використану хмарну інфраструктуру. Хмарні провайдери зазвичай не обмежують вибір розробника й підтримують велику кількість мов. Наприклад, офіційна AWS SDK підтримує C++, Go, Java, JavaScript, C#, Node.js, PHP, Python та Ruby.

Такі сервіси, як AWS Lambda відразу дозволяють використовувати код на C#, Go, Java, Python та декількох інших мовах. Якщо розробник не знайшов потрібний варіант – є можливість створити своє власне середовище виконання для Lambda. Крім того, сервіси по типу EC2, ECS та EKS орієнтовані на виконання розрахунків будь-якого виду, на заваді можуть стати лише апаратні обмеження. Таким чином, використання хмарних безсерверних технологій не обмежує вибір інструментів розробки.

Проаналізуємо та порівняємо Java, C#, JavaScript та Python з метою визначення найбільш підходящої мови програмування для розробки додатку для централізованого збору оповіщень.

Java – мова програмування, що була розроблена компанією «Sun Microsystems» в 1995 року, а з 2009 розвивається та підтримується компанією «Oracle». Java є об’єктно-орієнтованою мовою зі схожим до C/C++ синтаксисом. Одна з найважливіших концепцій Java – це незалежність від архітектури, тому спосіб виконання програм на цій мові дещо вирізняється: компілятор транслює вихідний код ПЗ у байт-код, який не може відразу виконуватись на пристроях, але є простішим за звичайні машинні команди та дозволяє легко переносити програми [17]. Цей байт-код потім виконується за допомогою JVM (Java Virtual Machine), яка транслює команди в машинний код та надає доступ до функцій ОС та апаратного забезпечення. Таким чином, додаток на Java може виконуватись на будь-якому пристрої з тією чи іншою реалізацією JVM.

Перевагами та особливостями Java є:

– підтримка об'єктно-орієнтованої парадигми – синтаксис мови та її стандартна бібліотека передбачає використання ООП;

– звичний для більшості розробників С-подібний синтаксис та помірна складність вивчення мови;

– добре підходить для серверних та корпоративних рішень, багато компаній використовують Java саме в цій сфері;

– покращена безпека через використання JVM та відсутність прямої роботи з пам'яттю пристрою;

– легка переносимість програм між платформами.

C# – популярна мова програмування від Microsoft, яка вперше з'явилася о 2000 році, а з 2002 року C# версії 1.0 почали включати в склад Visual Studio та .NET. Напочатку мову часто порівнювали з Java та розглядали її в якості конкурента на корпоративному ринку, зараз же C# має свій власний напрям розвитку та сферу застосування. Це найбільш популярна мова програмування для розробки десктопних додатків для ОС Windows, а фреймворк .NET ще більше спрощує ці задачі. Для роботи з іншими платформами є інструменти .NET Core та Mono. C# є компільованою мовою, але код програм спочатку транслюється в формат Intermediate Language (IL), а вже потім повторно компілюється за допомогою Common Language Runtime (CLR) в машинний код на цільовій системі, що додатково підвищує переносимість створеного ПЗ.

Переваги та особливості C# наступні [18]:

– високорівнева мова програмування зі зручним та безпечним менеджментом пам'яті – є засоби роботи з вказівниками та іншими просунутими інструментами, але вбудований збірник сміття самостійно буде спостерігати за знищення невикористаних об'єктів у пам'яті;

– орієнтація на використання ООП;

– глибока інтеграція з .NET Framework, що містить багато готовий рішень та інструментів для реалізації десктопних додатків та веб-сервісів;

– зрозумілий та звичний С-подібний синтаксис;

– якісна документація та велика спільнота.

JavaScript – мова програмування, що з’явилася у 1995 році та була стандартизована під назвою ECMAScript у 1997. Спочатку активно використовувалась для програмування додаткового функціоналу клієнтської частини інтернет-ресурсів, але з розвитком технологій отримала багато нових сфер застосування. Наприклад, платформа Node.js дозволяє виконувати скрипти на цій мові програмування без використання браузера та надає доступ до API операційної системи [19], що дало початок великій кількості десктопного та серверного ПЗ.

До переваг та особливостей JavaScript можна віднести:

– JavaScript – це інтерпретована мова, яка не вимагає компіляції та ціною меншої швидкодії дозволяє відразу виконувати написаний код на будь-якому пристрої з сумісним інтерпретатором;

– широка підтримка спільноти – на даний момент це одна з найбільш популярних мов програмування, тому існує велика кількість матеріалів для її вивчення, а також значний каталог бібліотек для вирішення багатьох задач;

– використання в якості «Full-Stack» технології – сучасні версії JavaScript можуть виконуватись і на сервері, і на боці клієнта, що дозволяє використовувати одну мову програмування для розробки всього продукту.

Python – мова програмування, яка має довгу історію ще з кінця 1980-х років. Реліз версії 1.0 стався у 1994 році, після чого мова досі продовжує розвиватись. Python також є інтерпретованою мовою програмування, але на відміну від JavaScript з самого початку розвивалася в якості мови загального призначення. Зараз вона часто використання для розробки онлайн-ресурсів та додатків з використанням машинного навчання і нейронних мереж. Крім того, це популярний інструмент для математичних та статистичних досліджень [20].

Найважливішими перевагами та особливостями Python є:

– простота вивчення та використання – синтаксис Python значно простіший від інших розглянутих мов, а стандартна бібліотека дає доступ до

зручних абстракцій та синтаксичного «цукру», що дозволяють досягти бажаного результату за меншу кількість рядків коду;

– велика кількість альтернативних інтерпретаторів – для найбільш популярних ОС існує офіційний інтерпретатор (CPython), але є багато проектів, які дозволяють використовувати альтернативні середовища виконання (Jython, IronPython, PyPy та інші);

– велика кількість бібліотек та інтеграція з іншими мовами – Python має велику екосистему з бібліотеками для багатьох типових задач, а також дозволяє застосовувати програмні компоненти на Java, C, C++ та інших мовах.

Розглянуті мови програмування було порівняно за набором критеріїв, результати наведено в таблиці 3.1.

Таблиця 3.1 – Порівняння мов програмування

Критерій	Java	C#	JavaScript	Python
Підтримка ООП	1	1	0,5	1
Простота синтаксису	0	0	0,5	1
Вбудована підтримка на AWS Lambda	1	1	1	1
Бібліотеки для кросплатформної розробки	1	0,5	1	1
Широкий вибір IDE та інструментів	1	0,5	0,5	1
Підсумок	4	3	3,5	5

Мова програмування Python за результатами порівняння є більш підходящою для розробки додатку для централізованого збору оповіщень.

Отже, в даному підрозділі було розглянуто переваги та особливості мов програмування Java, C#, JavaScript та Python. Проаналізовано кожну мову за певним набором критеріїв, результати зведено в таблицю. Python обрано для розробки програмного забезпечення.

3.2 Вибір середовища розробки

Інтегроване середовище розробки – це програмне забезпечення, що об’єднує і спрощує різносторонні активності, які пов’язані з процесом розробки

програм. Часто назву такого ПЗ скорочують до короткого «IDE» (від Integrated Development Environment). IDE мають за мету підвищення продуктивності розробника за допомогою об'єднання найбільш часто використовуваних інструментів в один продукт.

Більшість середовищ розробки мають наступний функціонал:

– текстовий редактор коду – на відміну від звичайного редактора IDE зазвичай має підсвічування синтаксису та автодоповнення ключових слів мови програмування;

– глибока інтеграція з компілятором або інтерпретатором – зазвичай середовище розробки вміє самостійно викликати засоби трансляції коду без необхідності введення додаткових команд збірки з боку користувача, також часто IDE надає графічний інтерфейс для детального налаштування процесу;

– інтеграція з налагоджувачем – більшість IDE можуть не тільки зібрати додаток, а й запустити його в режимі налагодження для дослідження поведінки та пошуку помилок в процесі виконання.

Python є однією з найбільш популярних мов програмування у світі, тому вибір середовищ розробки для неї надзвичайно широкий. Розглянемо найбільш популярні з них, а саме: Spyder, PyDev та PyCharm.

Spyder – це безкоштовне та відкрите середовище розробки, що активно розвивається спільнотою. Воно розроблене на Python та спроектоване для потреб вчених, інженерів та інформаційних аналітиків, які використовують цю мову для аналізу великих об'ємів даних. Але середовище добре підходить і для інших задач, адже має потужні вбудовані інструменти для тестування та налагодження [21]. Крім того, IDE підтримує розширення функціоналу за допомогою плагінів, що спрощує роботу з популярними бібліотеками та додатковими засобами розробки. Наприклад, є інтеграція з Jupiter Notebook – одним з найбільш популярних засобів для візуалізації даних та генерації документів для математичних та статистичних досліджень. Графічний інтерфейс середовища розробки зображено на рисунку 3.1.

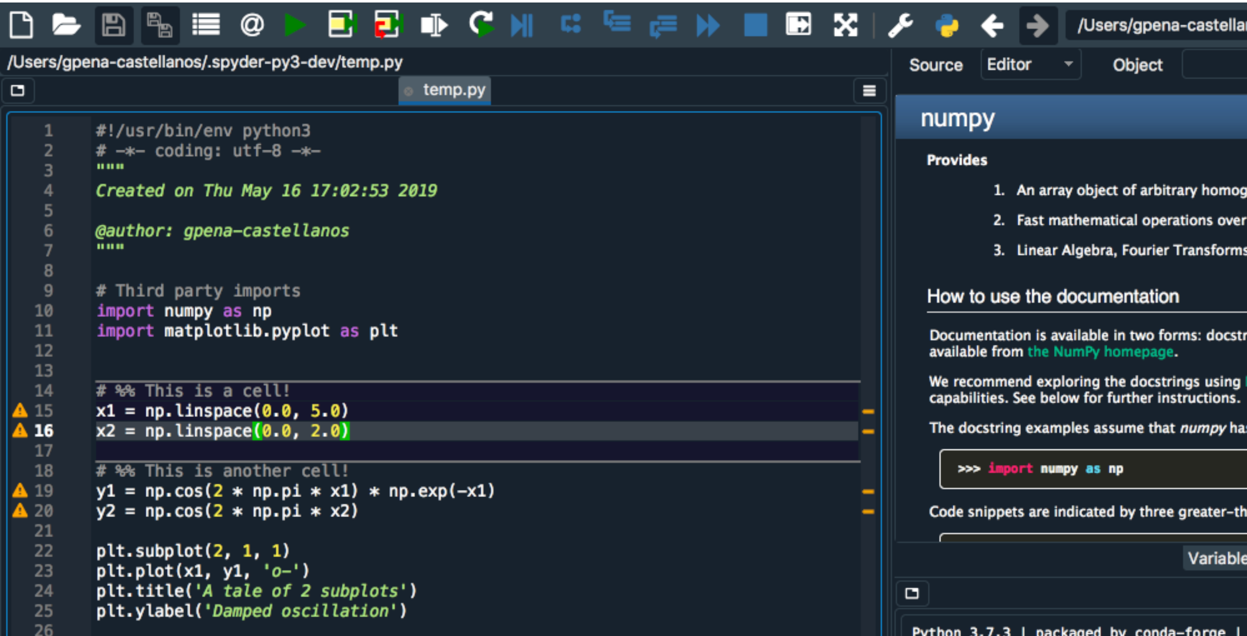


Рисунок 3.1 – Приклад графічного інтерфейсу Spyder

PyDev – це популярне середовище розробки, яке побудоване на платформі Eclipse. Eclipse – це модульне середовище, яке надає розробнику деякі базові інструменти та легко розширюється до повноцінної IDE. Тобто, PyDev включає в себе Eclipse та великий набір плагінів, які перетворюють цю платформу в потужну IDE для Python. Графічний інтерфейс PyDev наведено на рисунку 3.2.

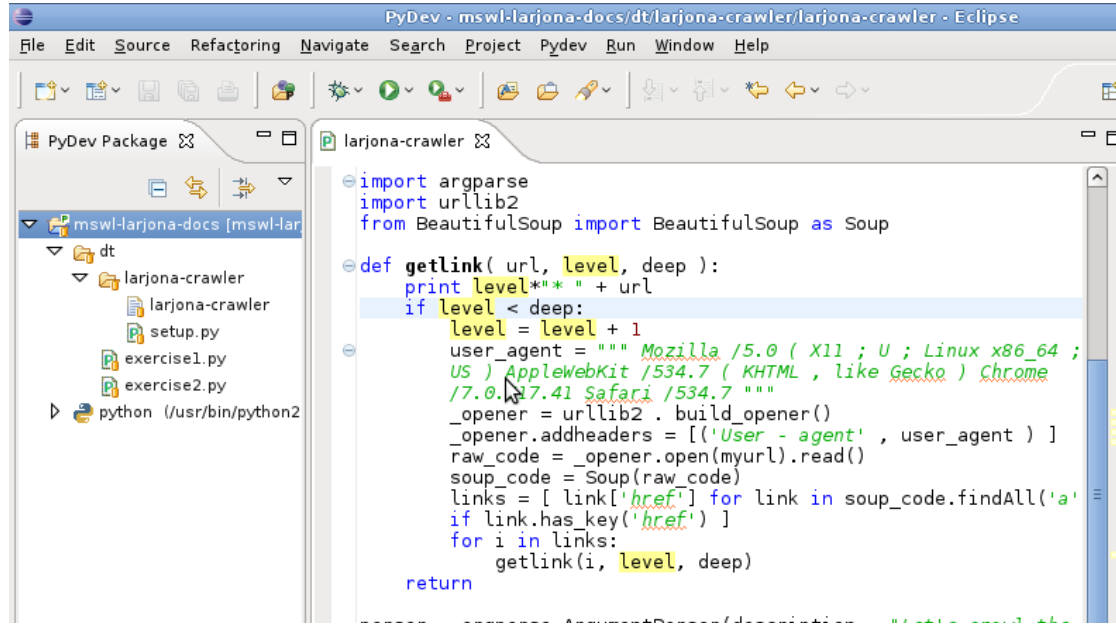


Рисунок 3.2 – Приклад графічного інтерфейсу PyDev

PyDev підтримує роботу з нестандартними інтерпретаторами, як-от Jython або IronPython. Також є підтримка популярних фреймворків [22] (наприклад, Django) та додаткових засобів дослідження коду (PyLint, MyPy та інших).

PyCharm – IDE від компанії JetBrains. Це середовище розробки працює з використанням засобів платформи Intellij IDEA, що дає їй відразу декілька переваг: один з найкращих аналізаторів коду, зручні інструменти рефакторингу, функціонал для швидкої навігації по коду, вбудовані засоби роботи з БД та популярними фреймворками для розробки онлайн-застосунків та тестування ПЗ [23]. Крім того, наявні інтеграції для роботи з науковими та інженерними інструментами, як-от IPython Notebook, Anaconda, Matplotlib та NumPy. Інтерфейс IDE зображено на рисунку 3.3.

Рисунок 3.3 – Приклад графічного інтерфейсу PyCharm

Кожна з розглянутих IDE має свої переваги та недоліки. Деякі середовища більш орієнтовані на застосування у певних прикладних сферах та передбачають використання разом зі спеціалізованими фреймворками та інструментами, а інші надають більш загальний функціонал для розробки класичних десктопних додатків. Для визначення найкращого інструменту для вирішення поставлених задач було проведено порівняння за певними критеріями. Результати порівняння та використані критерії наведено в таблиці 3.2.

Таблиця 3.2 – Порівняння середовищ розробки

Критерій	Spyder	PyDev	PyCharm
Універсальність функціоналу	0	0,5	1
Кросплатформність	1	1	1
Швидкодія та використання ресурсів системи	1	0	0,5
Інтеграція з БД та популярними фреймворками	0,5	0,5	1
Інструменти рефакторингу	0	0,5	1
Підсумок	2,5	2,5	4,5

Таким чином, за результатами порівняння PyCharm є найбільш інтегрованим середовищ розробки для розробки програмного забезпечення за поставлених умов і задач.

Отже, в даному підрозділі було розглянуто базові відомості про функціонал IDE. Детально розглянуто можливості Spyder, PyDev та PyCharm, наведено приклади інтерфейсу ПЗ. Було проведено порівняльний аналіз середовищ, для розробки додатку для централізованого зборку оповіщень було обрано PyCharm від JetBrains.

3.3 Розробка структури програмного забезпечення

Розробка структури для програмного забезпечення, яке будується для використання в хмарі та передбачає безсерверні розрахунки, потребує уважного аналізу всіх складових системи [24]. Це необхідно не тільки для кращого розуміння загального списку хмарних сервісів, які будуть використані, а й для оцінки грошових витрат на інфраструктуру.

Для програмного продукту для централізованого збору оповіщень було розроблено схему архітектури, що зображена на рисунку 3.4.

Відповідно до схеми, вся система складається з двох частин: клієнтської та серверної. Клієнтською частиною може виступати додаток для ПК і телефону або ж навіть бот для месенджера, що повністю позбавить кінцевого користувача потреби встановлювати додаткове ПЗ на свій пристрій. В рамках бакалаврської дипломної роботи в якості клієнтської частини розглянуто розробку програми

для десктопних платформ. Серверна частина системи побудована цілком на хмарних сервісах для безсерверних розрахунків. Розглянемо ці компоненти більш детально.

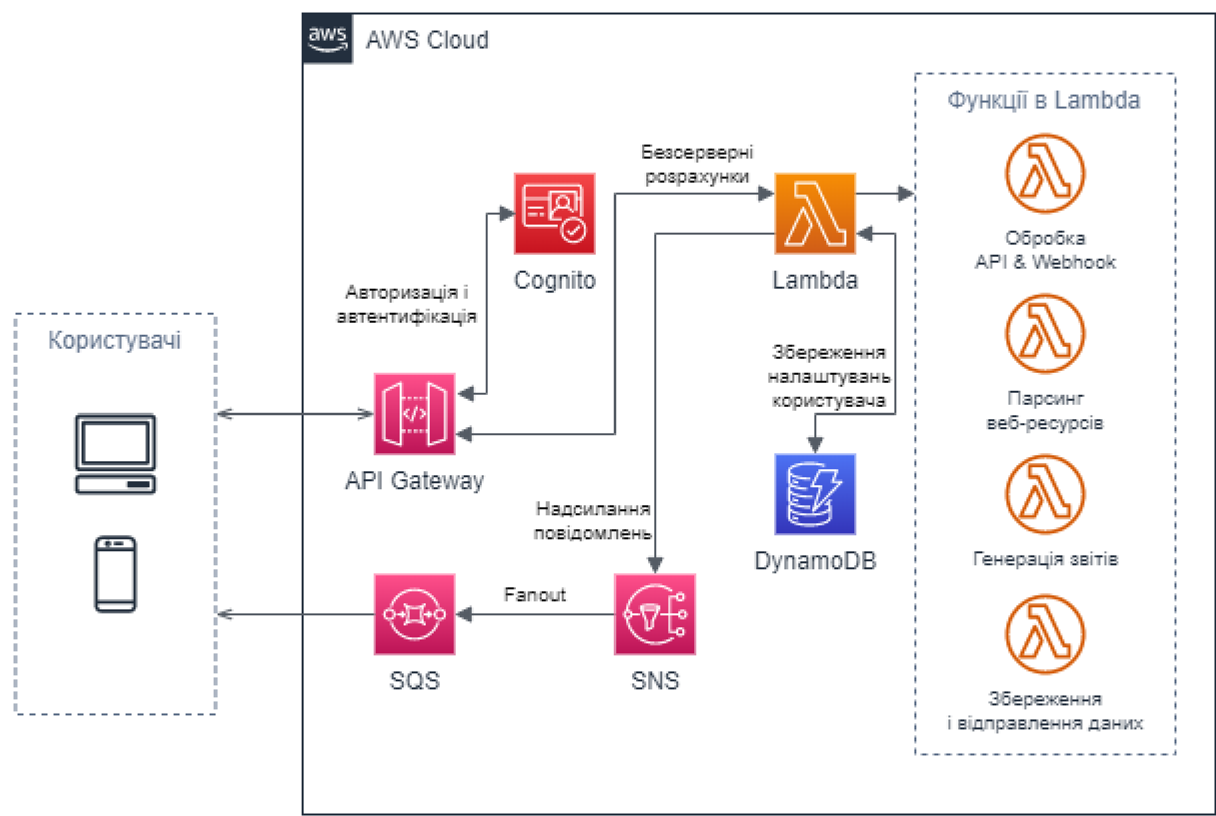


Рисунок 3.4 – Схема архітектури продукту

API Gateway – це сервіс для створення, підтримки, моніторингу та захисту власних прикладних програмних інтерфейсів (API). Для більшості безсерверних додатків API Gateway виступає в ролі єдиної вхідної точки, яка в залежності від запиту та інших параметрів розробленого API перенаправляє виклики в інші сервіси [25]. API Gateway має підтримку створення RESTful та WebSocket інтерфейсів, глибоку інтеграцію з різноманітними функціями AWS та навіть може генерувати клієнтські SDK для пришвидшення розробки.

Cognito – популярне рішення для авторизації, автентифікації та менеджменту користувачів. Сервіс спрощує реєстрацію нових користувачів та дозволяє скористатися уже готовими інтеграціями для використання акаунтів Facebook, Google та інших сторонніх мереж. Основна причина використання

цього сервісу – це хороша інтеграція з API Gateway [26], що дозволить швидко розробити якісний механізм захисту API та даних користувачів, а не писати самостійно чергове рішення для керування користувачами. Варто відмітити, що в даній архітектурі Cognito використовується лише для реєстрації та надання доступу кінцевим користувачам, а зберігання їх даних буде проводить в NoSQL базі даних DynamoDB.

Lambda – один з найважливіших хмарних сервісів AWS, який займається саме безсерверними розрахунками. Lambda дозволяє завантажити виконуваний код, налаштувати інтеграції для його запуску та відразу почати користуватися продуктом, без потреби додаткового хостингу середовища виконання коду. Зазвичай хмарні додатки розбиваються на велику кількість окремих частин [4], які всередині сервісу називаються функціями. Таким чином, у розробленій архітектурі кожній функції продукту відповідає функція всередині Lambda, яка в свою чергу може включати і використовувати інші, ще більш атомарні функції. Виконання коду в Lambda інтегроване з викликами до API Gateway – запит до конкретного методу інтерфейсу запустить відповідну йому функцію.

DynamoDB – безсерверна NoSQL база даних, яка орієнтована на зберігання документів або даних у форматі «ключ-значення». AWS автоматизує захист та реплікацію даних, а також весь менеджмент для підтримки БД [27]. На відміну від реляційних баз даних є можливість швидкого та простого масштабування навантаження на DynamoDB, а також функціонал для підвищення продуктивності за допомогою кешу DynamoDB Accelerator (DAX) та потокової реплікації зі створенням глобальних таблиць. Цей сервіс часто використовується разом з Lambda, так як безсерверні функції не можуть зберігати свій стан.

Крім того, для асинхронної (без взаємодії з API Gateway) відправки повідомлень до клієнта використовується зв’язка сервісів SNS та SQS. Simple Queue Service (SQS) – це розподілена система для черг повідомлень. Будь-який авторизований додаток може асинхронно отримати дані з черги. Simple Notification Service (SNS) – інструмент для реалізації Pub/Sub моделі («Публікація-підписка»). На відміну від черг SQS є можливість «підписатися»

відразу декільком клієнтам на одну тему, але якщо вони пропустили момент «публікації» повідомлення – ці дані втрачено. Часто використовується сценарій «Fanout», коли інформація надсилається на SNS, а звідти дані пересилаються в декілька SQS-черг [28]. Одна черга може відповідати, наприклад, за подальшу відправку даних на мобільні пристрої, а інша – на десктопні.

Отже, в даному підрозділі розглянуто архітектуру створюваного продукту. Наведено й описано схему компонентів, детально розглянуто сервіси хмарного провайдера AWS, які будуть використані для реалізації серверної частини додатку, а саме надано базове представлення про сервіси API Gateway, Cognito, Lambda, DynamoDB, SQS та SNS. Для кожного з сервісів описано можливості та функції, які можуть бути використані при створенні додатку.

3.4 Розробка структури графічного інтерфейсу

Графічний інтерфейс користувача (GUI) – це вид інтерфейсу, який дозволяє передавати інформацію кінцевому користувачу програмного забезпечення за допомогою графічних об’єктів, а не лише за допомогою тексту [29]. GUI може бути більш повільний за інтерфейс командного рядка (CLI) та вимагати більше часу й навичок для створення, але дозволяє пришвидшити роботу користувача та надає інструменти для комплексного графічного подання даних.

Проектування графічного інтерфейсу є важливим кроком, адже погано зроблений GUI може лише ускладнити взаємодію користувача з додатком. Таким чином, перед етапом реалізації додатку було додатково розроблено структурні схеми основних вікон. На рисунку 3.5 зображено структурну схему головного вікна програмного продукту.

Головне вікно включає наступні елементи інтерфейсу:

1) іконка програми;

2) стрічка заголовку;

3) системні кнопки керування вікном (кнопки «Згорнути», «Розгорнути» та «Закрити» для ОС Windows або ж інші в залежності від пристрою та системи);

4) кнопка для додавання нового джерела сповіщень;

5) кнопка для генерації звіту;

6) список останніх отриманих оповіщень.

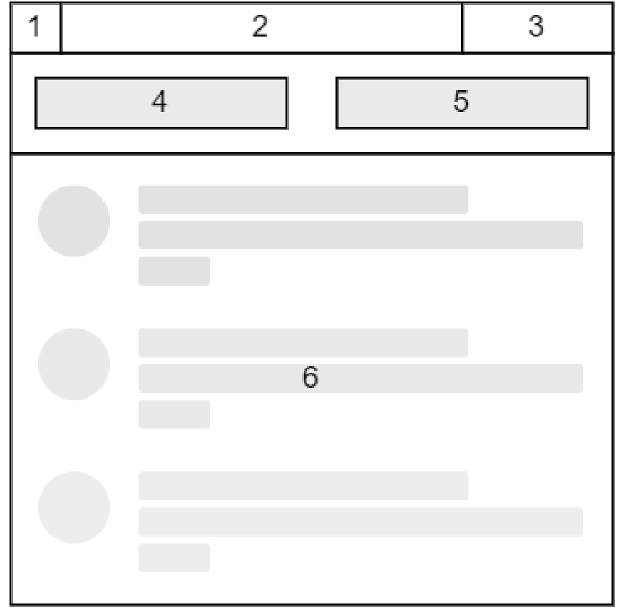


Рисунок 3.5 – Структурна схема головного вікна додатку

Кнопка для створення джерела сповіщень відкриває нове вікно з іншим набором графічних елементів. Структура цього вікна наведена на рисунку 3.6.

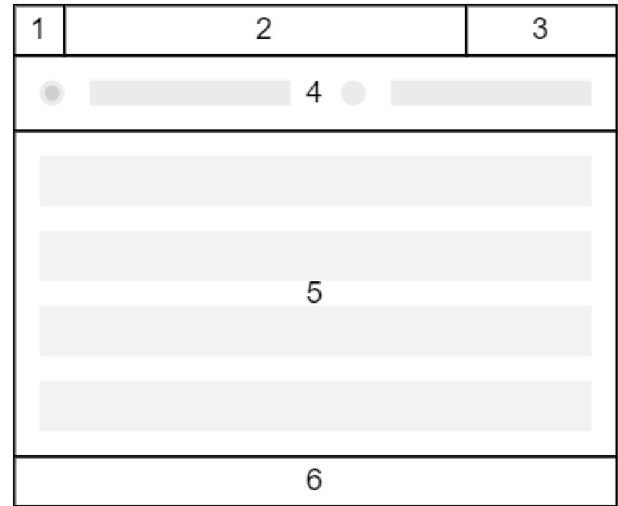


Рисунок 3.6 – Структурна схема вікна створення джерела сповіщень

Вікно включає в себе наступні елементи інтерфейсу:

1) іконка програми;

2) стрічка заголовку;

3) системні кнопки керування вікном;

4) радіокнопки для вибору типу джерела;

5) список полів для введення параметрів джерела;

6) рядок стану.

Так як передбачається робота з оповіщеннями, які можуть бути отримані і за допомого вебхуків, і через парсинг веб-ресурсів, то вікно включає радіокнопки для переключення між цими типами. В залежності від обраного типу джерела також змінюється набір полів для введення даних та кнопок для створення, тому ця динамічна область представлена одним елементом на схемі. Рядок стану повинен відображати результати додавання джерела або ж помилки, що можуть виникнути при перевірці введених даних та його створенні.

Кнопка для генерації звіту, що розташована на головному екрані додатку, відповідає за доступ до ще однієї функції ПЗ, яка вимагає додаткових розрахунків на сервері та відкриття нового вікна для відображення результатів. Структурна схема цього вікна показана на рисунку 3.7.

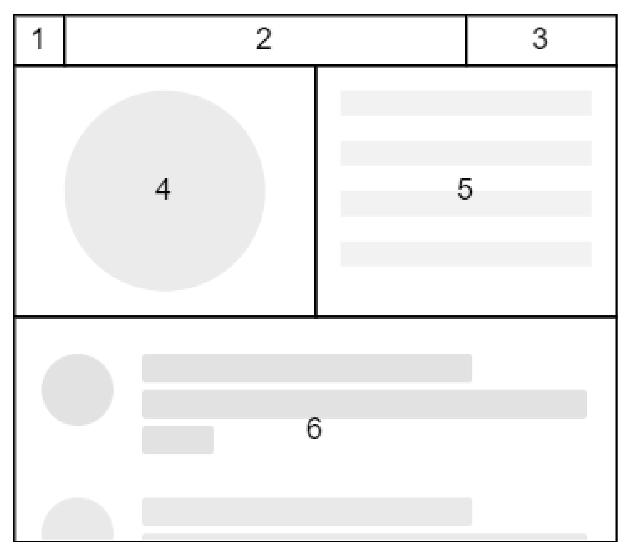


Рисунок 3.7 – Структурна схема вікна звіту

Вікно звіту включає такі елементи інтерфейсу:

1) іконка програми;

2) стрічка заголовку;

3) системні кнопки керування вікном;

4) секторна діаграма;

5) список найбільш кількісних джерел;

6) список оповіщень, що включені до звіту.

Секторна діаграма показує співвідношення кількості оповіщень з різних джерел. Також сервіси з найбільшою кількістю оповіщень перелічені в області справа від діаграми, а нижня область вікна дозволяє переглянути всі повідомлення, які були включені для аналізу.

Отже, в даному підрозділі було розглянуто базові відомості про графічний інтерфейс користувача. Розроблено структурні схеми інтерфейсу основних вікон додатку, детально описано їх елементи.

3.5 Програмна реалізація додатку

Для реалізації функціоналу додатку для збору оповіщень було розроблено велику кількість алгоритму, включаючи розглянуті в підрозділах 2.1, 2.2 та 2.3, а саме:

– алгоритм отримання оповіщень з використанням вебхуків та публічних API;

– алгоритм отримання оповіщень з використанням парсингу веб-ресурсів;

– алгоритм генерації звітів на основі зібраних оповіщень.

Розглянемо більш детально реалізацію програмного продукту.

Відповідно до описаної в підрозділі 3.3 архітектури весь код розділяється на дві частини: клієнтський додаток та серверну складову, що побудована на основі AWS-сервісів. Графічний інтерфейс користувача побудовано з використанням бібліотеки «Tkinter», що входить у склад стандартної бібліотеки мови Python [30]. Вхідною точкою додатку є функція «__main__» виконавчого файлу «main.py». При запуску клієнтського програмного забезпечення

розпочинається ініціація бібліотеки «Tkinter» та створення головного вікна, що наведено на рисунку 3.8.

```
if __name__ == '__main__':
    ctypes.windll.shcore.SetProcessDpiAwareness(2)

    root = tk.Tk()
    sv_ttk.use_light_theme()
    root.title('Notification collector')
    root.columnconfigure(0, weight=1)
    root.rowconfigure(0, weight=1)

    login = App(root)
    login.grid(column=0, row=0, sticky='nwes', padx=15, pady=15)
    login.columnconfigure(0, weight=1)
    login.rowconfigure(0, weight=1)

    root.mainloop()
```

Рисунок 3.8 – Частина лістингу виконавчого файлу «main.py»

За графічний інтерфейс користувача відповідає клас «App», екземпляр якого створюється в функції «__main__». Даний клас містить код для створення віджетів бібліотеки «Tkinter», які будуть відображатися у вікнах додатку, а також включає в себе методи для взаємодії з серверною частиною додатку. Наприклад, за допомогою бібліотеки «boto3», яка є офіційним AWS SDK для Python та дозволяє працювати з API в парадигмі ООП [31], здійснюється реєстрація та вхід користувачів, за що відповідають методи «sign_up» та «sign_in». Лістинг коду, що відповідає за вхід користувачів в додаток, наведено на рисунку 3.9.

Алгоритм отримання оповіщень з використанням вебхуків та публічних API реалізовано в вигляді окремої функції в сервісі безсерверних розрахунків AWS Lambda. Доступ до функції надається через сервіс API Gateway, який дозволяє налаштувати шлях та HTTP-методи доступу, а також за потреби може надати додаткові засоби безпеки для API. Наприклад, це може бути обмеження доступу лише для зареєстрованих користувачів. У випадку з обробкою вебхуків необхідно навпаки надати вільний доступ до адресу Lambda-функції, щоб зовнішні сервіси могли надсилати свої дані.

```python
def sign_in(self):
    if not self.username.get().strip() or not self.password.get().strip():
        messagebox.showerror('Error', 'Please fill in all fields!')
        return

    try:
        response = cognito.initiate_auth(
            ClientId=client_id,
            AuthFlow='USER_PASSWORD_AUTH',
            AuthParameters={
                'USERNAME': self.username.get(),
                'PASSWORD': self.password.get()
            }
        )
        print(response)
        self.token = response['AuthenticationResult']['AccessToken']
        messagebox.showinfo('Sign-in info', 'Successfully logged in!')
    except Exception as e:
        messagebox.showerror('Sign-in error', e)
```

Рисунок 3.9 – Лістинг методу «sign_in»

Обробка вебхуків починається з отримання вхідних даних, що передаються в якості параметрів всередині URL. Якщо необхідні параметри відсутні – подальше виконання неможливе. Лістинг цієї частини коду наведено на рисунку 3.10.

```python
# Get params
try:
    service = event['queryStringParameters']['service']
    user_id = event['queryStringParameters']['user_id']
except KeyError as e:
    print('Failed to get input parameters: ' + str(e))
    return {
        'statusCode': 403
    }
```

Рисунок 3.10 – Лістинг коду, що відповідає за отримання вхідних даних

Далі виконується перевірка даних користувача в відповідній таблиці в сервісі DynamoDB. Якщо запис користувача містить сервіс-джерело, яке надіслало повідомлення на URL вебхука, то скрипт виконується далі, інакше – обробка зупиняється. Це необхідно для того, щоб відсікти помилкові запити, коли вебхуки з боку користувача налаштовані некоректно та можливе випадкове

надсилання оповіщень сторонній людині. Лістинг цієї частини коду наведено на рисунку 3.11.

```
# Check user_id
try:
    users = dynamodb.Table('Users')
    query_res = users.query(
        KeyConditionExpression=boto3.dynamodb.conditions.Key('sub').eq(user_id)
    )

    if query_res['Count'] != 1:
        print('Failed to get user with sub: ' + user_id)
        return {
            'statusCode': 403
        }

    if service not in query_res['Items'][0]['webhooks']:
        print('Service ' + service + ' is not in user webhooks list: ' + user_id)
        return {
            'statusCode': 403
        }
```

Рисунок 3.11 – Лістинг коду, що відповідає за перевірку списку сервісів користувача

Після цього виконується основна частина алгоритму – обробка отриманих даних на основі списку полів та дій, що були отримані в DynamoDB з окремої таблиці «Triggers». Лістинг цього коду зображено на рисунку 3.12.

```
triggers = dynamodb.Table('Triggers')

service_trigger = triggers.get_item(Key={'Service': service})
fields = service_trigger['Item']['fields']
actions = service_trigger['Item']['actions']

for k, v in fields.items():
    fields[k] = eval(v, {}, {'input': json.loads(event['body'])})

notification_text = eval(actions, {}, fields)
```

Рисунок 3.12 – Лістинг коду, що відповідає за обробку даних від сервісів

При успішній обробці даних створюється новий запис в таблиці «Notifications» в DynamoDB, після чого оповіщення може бути надіслане в

клієнтський додаток. Код, що відповідає за збереження даних, наведено на рисунку 3.13.

```
# Create notification
try:
    results = dynamodb.Table('Notifications')
    results.put_item(Item={
        'id': str(uuid.uuid4()),
        'service': service,
        'user_id': user_id,
        'text': notification_text
    })
except Exception as e:
    print('Failed to add new notification: ' + str(e))
    return {
        'statusCode': 500
    }
```

Рисунок 3.13 – Лістинг коду, що відповідає за збереження даних оповіщення

Алгоритм отримання оповіщень з використанням парсингу веб-ресурсів також реалізовано з використанням безсерверних розрахунків сервісу Lambda. Проте ця функція не очікує вхідних даних від зовнішніх сервісів, а самостійно запускається з певним проміжком у часі. Перший етап виконання – це отримання вмісту сторінки та його парсинг з урахуванням переданих правил, які містяться в змінній «payload». Лістинг цього коду наведено на рисунку 3.14.

```
# Parse page
page = requests.get(payload['url'])
soup = BeautifulSoup(page.content, 'html.parser')
container = soup.select(payload['container_selector'])
notif_list = []
for item in container:
    notif_text = ""
    for rule in payload['items']:
        rule_res = item.select_one(rule).string
        if rule_res is not None:
            notif_text += (rule_res + ' ')
    notif_list.append(notif_text.strip())
```

Рисунок 3.14 – Лістинг, коду що відповідає за перевірку вмісту сторінки

Після формування списку всіх потенційних оповіщень, що можуть міститися на сторінці, відбувається порівняння з уже існуючим списком, щоб виключити ті оповіщення, які вже надсилалися користувачеві при попередньому запуску функції. Якщо фінальний список має ненульову довжину, то він містить повідомлення, що необхідно надіслати в клієнтський додаток. Крім того, необхідно додатково зберегти результати парсингу сторінки, щоб забезпечити коректність наступних запусків Lambda-функції. Фрагмент відповідного коду наведено на рисунку 3.15.

```
# Send new items
try:
    results_table = dynamodb.Table('Notifications')
    for notif in diff:
        results_table.put_item(Item={
            'id': str(uuid.uuid4()),
            'service': payload['url'],
            'user_id': payload['user_id'],
            'text': notif
        })
except Exception as e:
    print('Failed to add new notification: ' + str(e))

# Update previous list
try:
    rules_table = dynamodb.Table('ScrapeRules')
    rules_table.update_item(
        Key={'rule_id': payload['rule_id']},
        UpdateExpression='set previous = :r',
        ExpressionAttributeValues={
            ':r': notif_list,
        }
    )
except Exception as e:
    print('Failed to update previous list: ' + str(e))
```

Рисунок 3.15 – Фрагмент лістингу коду, що відповідає за збереження даних

Реалізація алгоритму генерації звітів на основі зібраних оповіщень розділено на дві частини: серверну та клієнтську. Таке розділення необхідне для більш ефективного використання ресурсів – зображення діаграми для звіту набагато швидше й простіше згенерувати на стороні клієнта з використанням уже підготовлених даних, аніж реалізовувати весь процес всередині хмарних сервісів та додатково вирішувати задачу зберігання і надсилання зображень.

Обробка даних для звіту також реалізована в вигляді Lambda-функції. Спочатку функція отримує з DynamoDB список оповіщень, для чого додатково використовуються передані користувачем налаштування та фільтри, що показано на рисунку 3.16.

```
input_data = json.loads(event['body'])
count_limit = input_data.get('limit', 200)
filters = boto3.dynamodb.conditions.Attr('user_id').eq(user_id)

response = None
try:
    n_table = dynamodb.Table('Notifications')
    response = n_table.scan(
        Limit=count_limit,
        FilterExpression=filters
    )
except Exception as e:
    print('Failed to get rules from DynamoDB: ' + str(e))
    return {
        'statusCode': 200,
        'body': json.dumps({
            'count': 0,
            'msg': 'Failed to get notifications'
        })
    }
```

Рисунок 3.16 – Лістинг коду для отримання списку з DynamoDB

Якщо отриманий список не є пустим, що може статися при передачі некоректних даних від користувача, то відбувається групування оповіщень за сервісом-джерелом. Після цього всі ці дані надсилаються в клієнтський додаток, що показано на рисунку 3.17.

```
counter = collections.Counter()
for notif in response['Items']:
    counter.update({notif['service']: 1})

return {
    'statusCode': 200,
    'body': json.dumps({
        'count': response['Count'],
        'msg': 'Success',
        'top': counter.most_common(10),
        'items': response['Items']
    }, ensure_ascii=False)
}
```

Рисунок 3.17 – Лістинг коду для формування і надсилання даних

Подальша обробка даних для звіту на стороні клієнта є мінімальною: загальний список оповіщень відразу показується користувачеві, а список з групуванням на основі назви сервісу використовується для генерації діаграми. Для створення графічного представлення використовується бібліотека «matplotlib», після чого воно відображається в GUI з використанням віджетів бібліотеки «Tkinter». Цей код наведено на рисунку 3.18.

```
# Pie chart
fig = Figure()
colors = ['#EDE3DB', '#ECB99C', '#BE9593', '#F1D7D6', '#6E6D71']
ax = fig.add_subplot(111)
ax.pie(data, labels=labels, autopct='%1.1f%%',
       startangle=90, colors=colors,
       pctdistance=0.8)

centre_circle = Circle((0, 0), 0.6, fc='white')
ax.add_patch(centre_circle)

chart_pie = FigureCanvasTkAgg(fig, frame_charts)
chart_pie.get_tk_widget().pack()
```

Рисунок 3.18 – Лістинг коду, що відповідає за створення і відображення діаграми

Отже, в підрозділі 3.5 було описано програмний код основних складових частин програмного продукту. Було детально описано реалізацію алгоритмів додатку. Повний лістинг коду ПЗ наведено в додатку Б.

3.6 Висновки

У даному розділі було проведено аналіз мов програмування Java, C#, JavaScript та Python. За результатами порівняння було обрано мову Python для розробки програмного забезпечення. Було розглянуто інтегровані середовища розробки Spyder, PyDev та PyCharm. Порівняння характеристик IDE дозволило обрати PyCharm в якості середовища для розробки продукту. Було детально розглянуто структуру ПЗ, розроблено структурні схеми графічного інтерфейсу, розглянуто програмну реалізацію додатку.

4 ТЕСТУВАННЯ ПРОГРАМИ

4.1 Аналіз методів тестування програмного забезпечення

Тестування – це процес перевірки роботи програмного забезпечення з метою виявлення помилок і встановлення відповідності між вимогами до продукту та його реалізацією [32]. Виявлення усіх помилок та проблем у додатку не є практично можливим, але добре налаштований процес тестування, що йде з перших етапів розробки, може значно зменшити їх кількість та підвищити загальну якість продукту й різноманітні функціональні та нефункціональні характеристики.

Класифікація видів тестування є умовною, адже різні методики можуть бути тісно пов’язані одне з одним та переслідувати схожі цілі. Всі види тестування програмного забезпечення, залежно від установленої мети, можна розділити на наступні загальні групи:

– функціональні (functional testing);

– нефункціональні (non-functional testing);

– пов’язані зі змінами (regression testing).

За знанням системи класифікація є наступною:

– тестування «чорного ящика» (black box);

– тестування «білого ящика» (white box);

– тестування «сірого ящика» (gray box).

Функціональні види тестування розглядають зовнішню поведінку системи. Функціональні тести базуються на функціях і особливостях, а також взаємодії з іншими системами, і можуть бути представлені на всіх рівнях тестування. Найбільш поширеними видами функціонального тестування є: компонентне або модульне тестування (unit testing), інтеграційне тестування (integration testing), регресивне тестування (regression testing), системне тестування (system testing) та інші.

Нефункціональне тестування перевіряє характеристики програмного забезпечення, що напряму не пов’язані з функціоналом, як-от продуктивність та

ефективність роботи продукту, зручність використання, стійкість до довготривалих навантажень або навантажень вище проектного максимуму, а також багато інших. В цілому, це тестування того, як саме система працює.

Для перевірки працездатності програмного продукту було обрано два методи: автоматизоване модульне тестування та ручне тестування різноманітних сценаріїв за методом «чорного ящика». Модульне тестування відноситься до категорії функціональних видів тестування та перевіряє окремі найменші функції і методи ПЗ. Так як воно покладається на роботу з внутрішнім кодом системи, а не з доступними зовнішніми інтерфейсами, то такий вид тестування відносять до тестування «білого ящика». Різні види тестування «чорного ящика» зазвичай не повинні знати про внутрішню структуру й роботу продукту, а тестування «сірого ящика» передбачає лише часткове розкриття деталей внутрішньої роботи.

Таким чином, у підрозділі було розглянуто загальні відомості про тестування програмного забезпечення та описано найбільш популярні способи класифікації видів тестування. Для перевірки функціонування розробленого ПЗ обрано модульне тестування та тестування «чорного ящика»

4.2 Тестування розробленого програмного продукту

Клієнтська частина додатку «Notification Collector» відповідає лише за відображення графічного інтерфейсу та обробку дій користувача. Решта функціоналу побудована за допомогою сервісів AWS, як-от API Gateway для простого й захищеного доступу до API або Lambda для виконання безсерверних розрахунків. Всередині Lambda-функцій на мові Python реалізовано основні алгоритми ПЗ, а також побудовано додаткові інструменти для взаємодії з клієнтською частиною продукту. З метою тестування даного функціоналу було розроблено набір з 8 модульних тестів, які перевіряють:

– створення запису користувача після реєстрації акаунту через сервіс AWS Cognito;

– створення інтеграції на основі вебхуків;

– створення інтеграції з використанням парсера інтернет-ресурсів;

– генерацію звітів;

– отримання загального списку оповіщень;

– обробку загального списку правил для парсера;

– обробку окремих правил для парсера;

– обробку вхідних даних вебхуків.

Модульне тестування було автоматизовано за допомогою бібліотеки «PyTest», яка спрощує написання тестів та дозволяє їх пакетне виконання, щоб мати можливість за один запуск перевірити весь необхідний функціонал [33]. Більшість тестів включають у себе підготовку вхідних даних, виклик функції, яка реалізує необхідний функціонал, порівняння очікуваних і отриманих результатів. На рисунку 4.1 наведено приклад коду модульного тесту для перевірки створення запису.

```
def test_add_user():
    test_uuid = str(uuid.uuid4())
    event = {'request': {'userAttributes': {'sub': test_uuid}}}
    response = add_user.lambda_function.lambda_handler(event, None)
    assert test_uuid == response['request']['userAttributes']['sub']
```

Рисунок 4.1 – Лістинг тесту для створення запису користувача

Модульне тестування продукту не виявило жодних проблем. Результати запуску всіх розроблених тестів наведено на рисунку 4.2.

```
============================= test session starts ==============================
collecting ... collected 8 items

test.py::test_add_user PASSED                                            [ 12%]
test.py::test_create_wh_int PASSED                                       [ 25%]
test.py::test_create_sc_int PASSED                                       [ 37%]
test.py::test_generate_report PASSED                                     [ 50%]
test.py::test_get_all PASSED                                             [ 62%]
test.py::test_scrapper PASSED                                            [ 75%]
test.py::test_scrapper_targets PASSED                                    [ 87%]
test.py::test_webhooks PASSED                                            [100%]

============================== 8 passed in 4.61s ===============================
```

Рисунок 4.2 – Загальні результати модульного тестування

Для тестування за методом «чорного ящика» було розроблено набір тест-кейсів, які покривають сценарії використання продукту кінцевим користувачем. Розглянемо їх більш детально.

Тест-кейс №1 «Реєстрація нового користувача»:

1) відкрити клієнтський додаток;

2) у вікні авторизації обрати режим «Реєстрація»;

3) заповнити поля імені користувача, електронної адреси та паролю;

4) натиснути кнопку «Реєстрація».

Очікуваним результатом є підтвердження реєстрації та відкриття вікна з повідомленням про необхідність перевірки електронної пошти користувача. Результат виконання тест-кейсу наведено на рисунку 4.3.

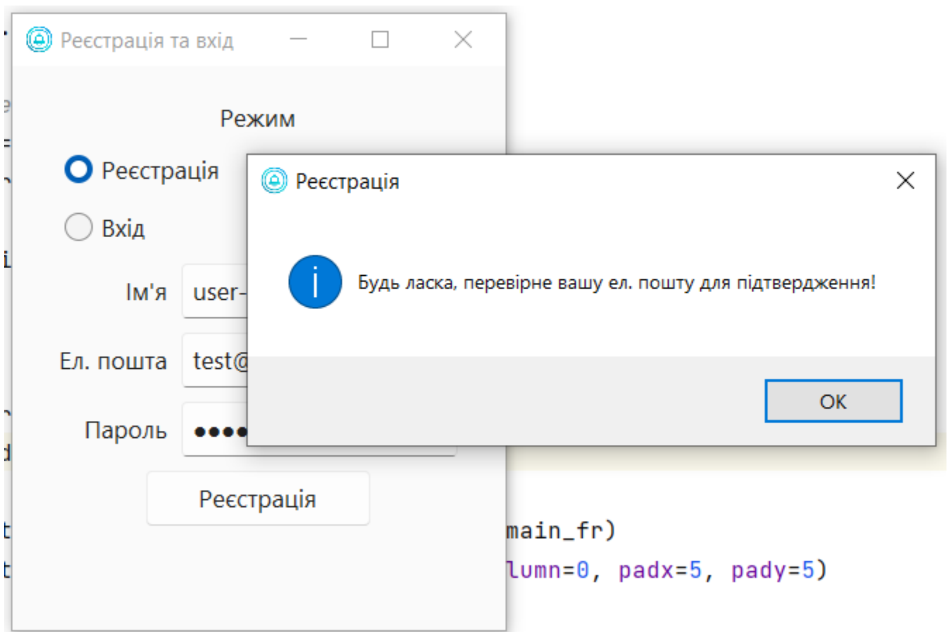


Рисунок 4.3 – Результат виконання тест-кейсу №1

Тест-кейс №2 «Додавання джерела з підтримкою вебхуків»:

1) відкрити клієнтський додаток;

2) ввійти з використанням попередньо зареєстрованого акаунту;

3) натиснути кнопку «Додати джерело»;

4) обрати тип «Webhook»;

5) обрати з випадаючого списку будь-який підтримуваний сервіс;

6) натиснути кнопку «Створити».

Очікуваним результатом є відкриття вікна з підтвердженням створення нового джерела для акаунту та генерація посилання, яке необхідно використовувати всередині відповідного сервісу для реєстрації вебхука. Згенероване посилання доступне для копіювання в окремому полі інтерфейсу, що доступне лише для зчитування. Результат виконання тест-кейсу наведено на рисунку 4.4.

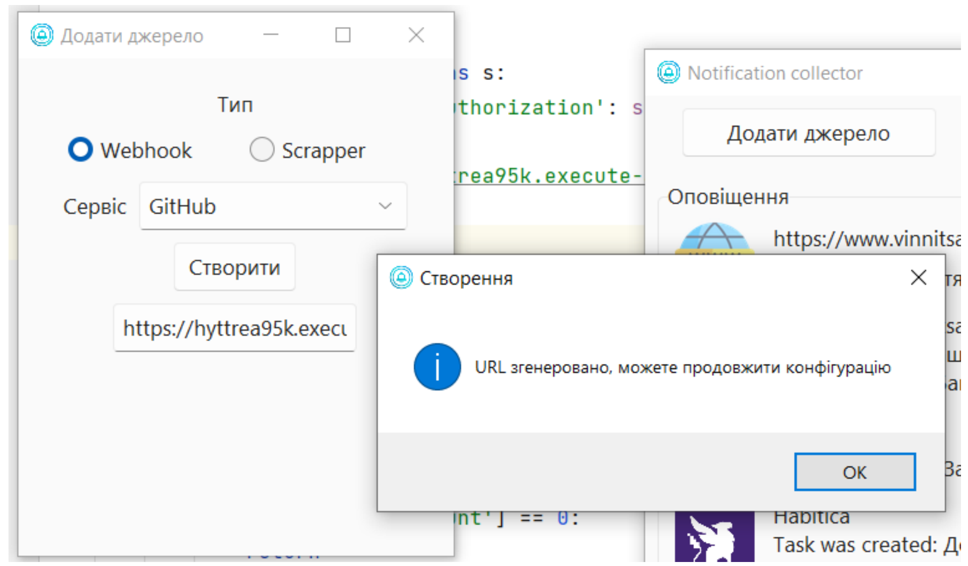


Рисунок 4.4 – Результат виконання тест-кейсу №2

Тест-кейс №3 «Додавання джерела з використанням парсера»:

1) відкрити клієнтський додаток;

2) ввійти з використанням попередньо зареєстрованого акаунту;

3) натиснути кнопку «Додати джерело»;

4) обрати тип «Scrapper»;

5) заповнити поля «URL», «Контейнер» та «Елемент»;

6) натиснути кнопку «Створити».

Очікуваним результатом є відкриття вікна з підтвердженням створення нового джерела для акаунту. Якщо введені параметри є коректними, то додаткові дії зі сторони користувача непотрібні. Результат виконання тест-кейсу наведено на рисунку 4.5.

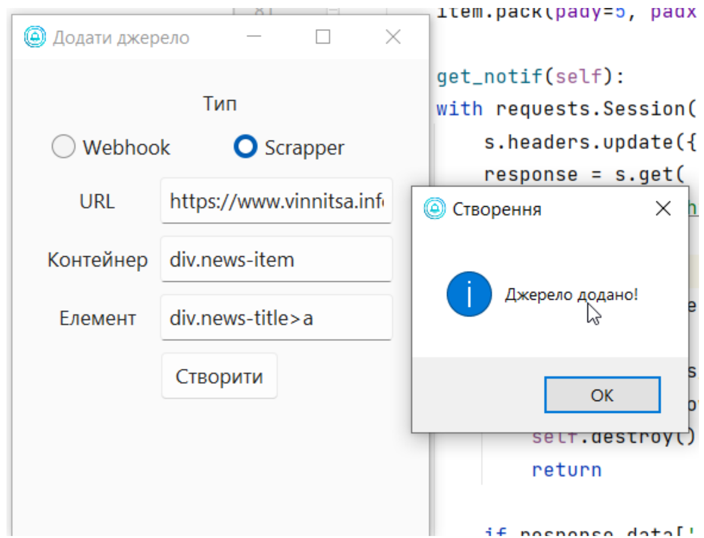


Рисунок 4.5 – Результат виконання тест-кейсу №3

Тест-кейс №4 «Отримання оповіщень»:

1) відкрити клієнтський додаток;

2) ввійти з використанням попередньо зареєстрованого акаунту;

3) додати одне або декілька джерел відповідно до тест-кейсів №2 або №3;

4) згенерувати події-оповіщення за допомогою взаємодії з сервісами, які реєструвались в якості джерел на кроці 3;

5) перевірити список оповіщень в головному вікні додатку.

Очікуваним результатом є оновлення головного вікна й відображення списку нових повідомлень. Результат виконання наведено на рисунку 4.5.

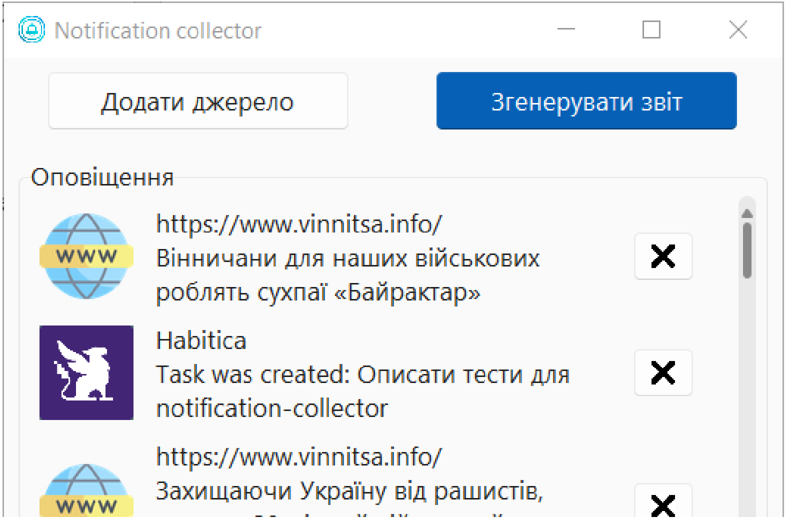


Рисунок 4.5 – Результат виконання тест-кейсу №4

Тест-кейс №5 «Генерація звіту»:

1) відкрити клієнтський додаток;

2) ввійти з використанням попередньо зареєстрованого акаунту;

3) виконати кроки 3-4 тест-кейсу №4;

4) в головному вікні додатку натиснути кнопку «Згенерувати звіт».

Очікуваним результатом є відкриття нового вікна зі згенерованою діаграмою, списком джерел, які мають найбільше оповіщень, а також відображення загального списку проаналізованих повідомлень. Результат виконання наведено на рисунку 4.6.

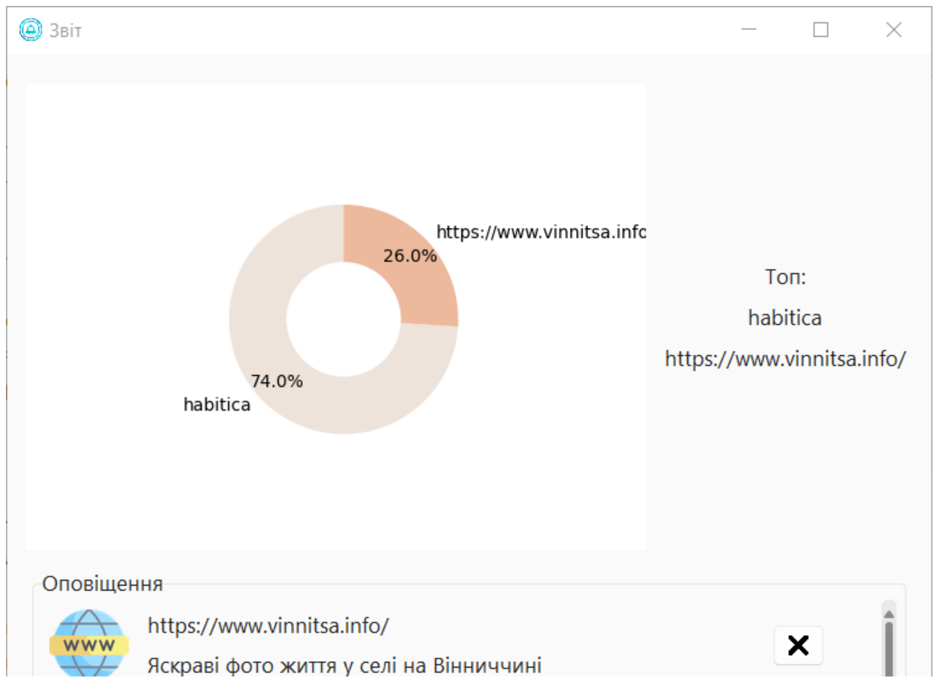


Рисунок 4.6 – Результат виконання тест-кейсу №5

Фактичні результати виконання тест-кейсів №1-5 відповідають очікуваним, усі тести пройдено успішно.

Таким чином, у даному підрозділі було розглянуто розроблені автоматизовані модульні тести та ручні тести за методом «чорного ящика». Наведено список функцій, для яких було розроблено модульні тести, розглянуто результати їх виконання. Було детально описано розроблені тест-кейси. Тестування додатку «Notification Collector» не виявило жодних проблем.

4.3 Розробка інструкції користувача

Для початку використання «Notification Collector» користувачеві необхідно завантажити клієнтський додаток та мати стабільне інтернет-підключення для використання усіх функцій ПЗ. Відразу після запуску додатку відкривається вікно роботи з акаунтом, яке зображене на рисунку 4.7. Воно надає можливість зареєструватися або ввійти під уже існуючим обліковим записом.

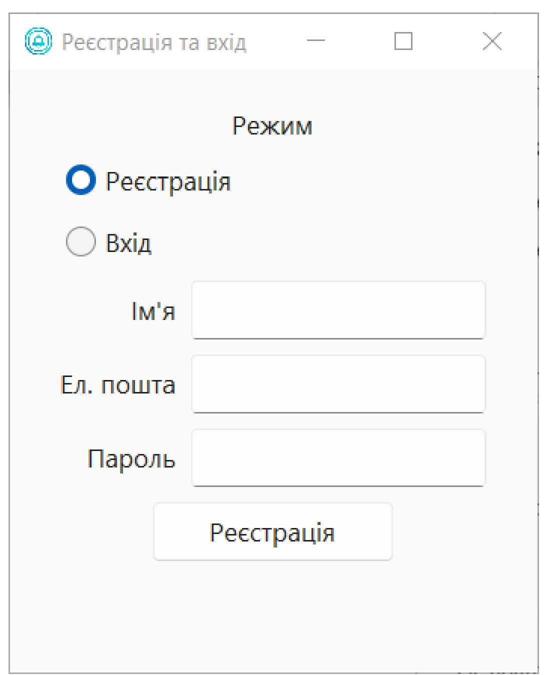


Рисунок 4.7 – Вікно реєстрації та входу в акаунт

При реєстрації нового облікового запису користувачеві необхідно підтвердити свою електронну адресу – після натискання кнопки «Реєстрація» автоматично надсилається лист з відповідним посиланням. Далі користувач може змінити режим на «Вхід» за допомогою відповідної радіокнопки в GUI та ввійти в cткорений акаунт. Якщо вхід виконано успішно, то автоматично відкривається основне робоче вікно додатку «Notification Collector», що зображене на рисунку 4.8.

Головне вікно надає доступ до основних фукнцій програмного забезпечення. Воно розділене на дві складові частини: верхня містить кнопки для додавання нового джерела оповіщень і генерування звіту, а нижня відображає

список останніх отриманих оповіщень. Список оновлюється при надходженні нових повідомлень, кожен пункт списку описує окреме сповіщення і включає в себе назву джерела, його логотип, отриману інформацію та кнопку видалення зі списку.

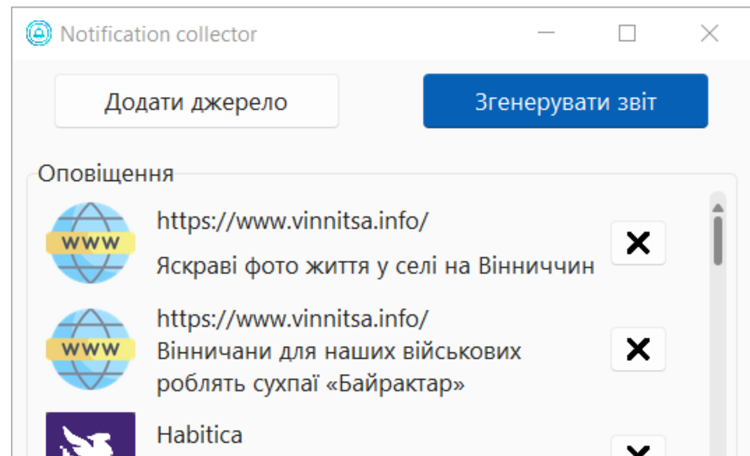


Рисунок 4.8 – Головне вікно додатку

При натисканні кнопки «Додати джерело» відкривається нове вікно, що наведене на рисунку 4.9, яке необхідне для створення нової інтеграції для збору оповіщень.

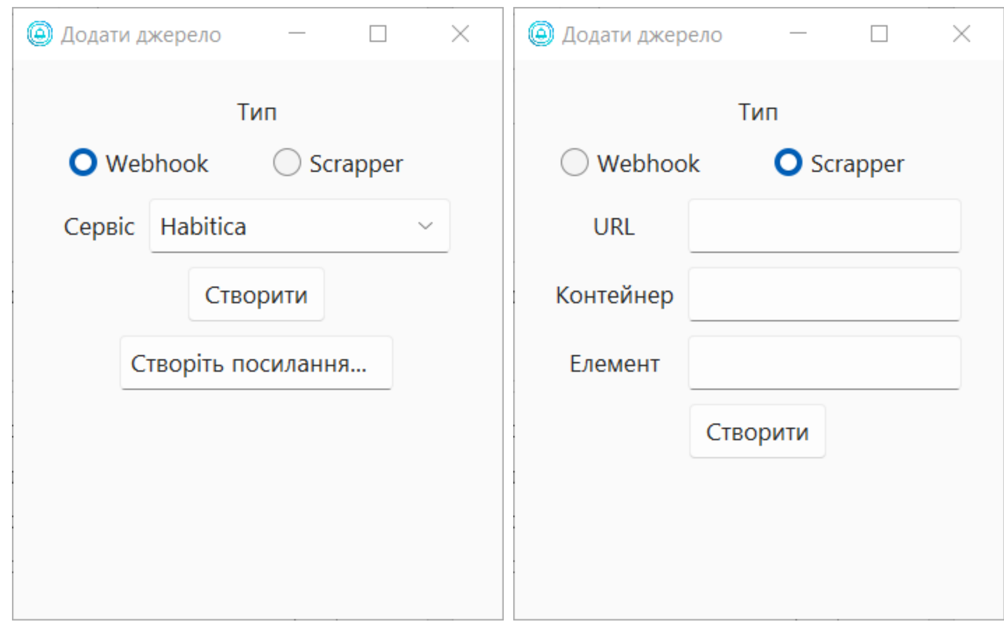


Рисунок 4.9 – Вікно додавання нового джерела для різних типів інтеграцій

Інтерфейс вікна автоматично оновлюється відповідно до обраного типу інтеграції. Користувач може згенерувати посилання для роботи з вебхуками, яке потім необхідно додатково зареєструвати в відповідному підтримуваному сервісі. Ця процедура сильно відрізняється в залежності від ресурсу, тому не може бути автоматизована всередині додатку.

Другий варіант створення інтеграції – це використання парсера для отримання даних від будь-якого інтернет-ресурсу. Для цього користувачеві необхідно ввести URL сайту, а також вказати CSS-селектори для загального списку та для вибірки окремих елементів (наприклад, «div.news-item» та «div.news-title>a» відповідно). Дані, які отримані за допомогою цих селекторів, використовуються для створення оповіщень. Такий спосіб отримання інформації вимагає додаткового досвіду та знань з боку користувача, але надає значно більше свободи та можливість використовувати ПЗ з практично будь-яким інтернет-ресурсом.

При натисканні кнопки «Згенерувати звіт» відкривається нове вікно, що наведене на рисунку 4.10.

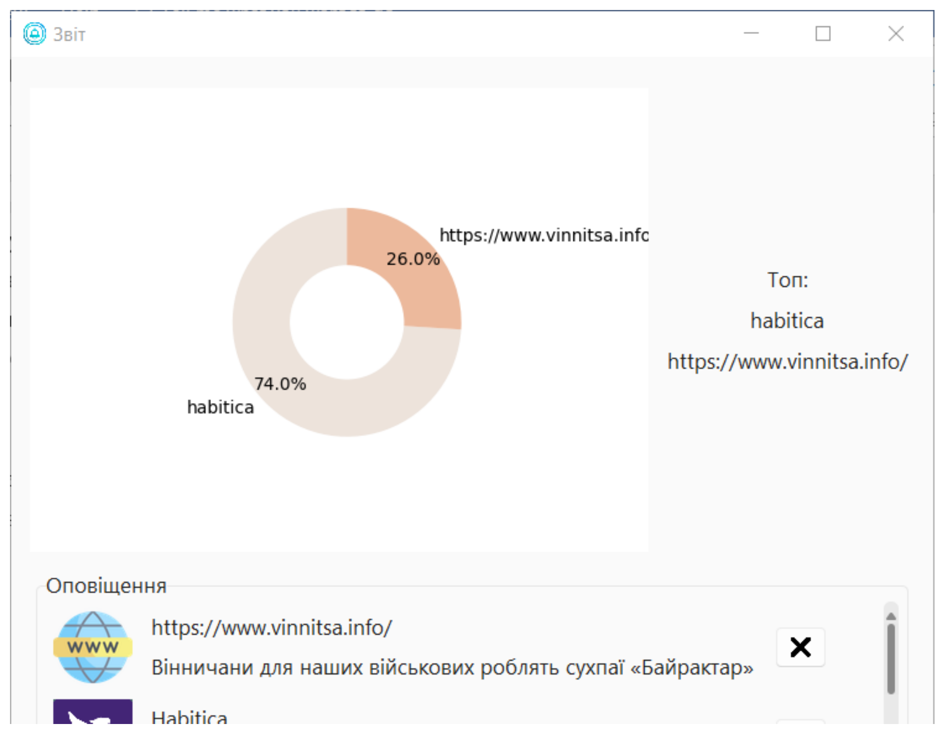


Рисунок 4.10 – Вікно зі звітом

Додаток автоматично збирає відомості про останні отримані сповіщення та генерує короткий звіт на їх основі. В лівій частині вікна користувач може переглянути секторну діаграму, яка відображає співвідоношення оповіщень від різних джерел. В правій частині вікна можна переглянути які сервіси надсилають найбільше даних. Нижня частина містить докладний список усіх оповіщень, що були використані для аналізу.

Таким чином, у даному підрозділі було розглянуто розроблену інструкцію користувача. Описано процес реєстрації та підтвердження облікового запису, розглянуто всі базові функції додатку, що доступні кінцевому користувачеві.

4.4 Вимоги до персонального комп’ютера

Додаток було розроблено з використанням інтерпретованої мови програмування Python та ряду бібліотек, що спростили роботу з AWS SDK і допомогли в створенні графічного інтерфейсу користувача. Зазвичай для розповсюдження і виконання Python-скриптів на пристрої кінцевого користувача необхідно додатково встановити інтерпретатор та пакети всіх залежних бібліотек, що значно ускладнює роботу з ПЗ. Для вирішення проблеми було використано бібліотеку «PyInstaller», яка автоматично збирає усі необхідні компоненти в готовий для використання виконавчий файл. Також можливо додатково налаштувати роботу цього інструменту та включити в результуючу збірку ПЗ усі необхідні сторонні файли, як-от зображення, конфігураційні файли, файли з налаштуваннями теми інтерфейсу та інші.

Мінімальна та рекомендована конфігурації персонального комп’ютера для використання додатку наведені в таблиці 4.1.

Таблиця 4.1 – Мінімальна та рекомендована конфігурації ПК

Параметр	Мінімальна конфігурація	Рекомендована конфігурація
Центральний процесор	Процесор на архітектурі x86-64 з тактовою частотою 1,6 ГГц	Процесор на архітектурі x86-64 з тактовою частотою 2,4 ГГц
Об’єм ОЗП	4 ГБ DDR3	8 ГБ DDR4

Продовження таблиці 4.1

Параметр	Мінімальна конфігурація	Рекомендована конфігурація
Накопичувач	HDD, 200 МБ вільного місця	SSD, 200 МБ вільного місця
Графічний процесор	Інтегрований графічний пристрій	Інтегрований графічний пристрій
ОС	Windows 7, macOS High Sierra, Ubuntu 18.04	Windows 10, macOS Monterey, Ubuntu 20.04
Мережа	Стабільне інтернет-з’єднання зі швидкістю 10 Мбіт/с	Стабільне інтернет-з’єднання зі швидкістю 50 Мбіт/с або більше

Таким чином, було описано використання бібліотеки «PyInstaller» для розповсюдження додатку для централізованого збору оповіщень. Наведено мінімальну та рекомендовану конфігурації персонального комп’ютера.

4.5 Висновки

У даному розділі було розглянуто базові відомості про тестування програмного забезпечення. Наведено спрощену класифікацію видів тестування, обрано модульне тестування та тестування «чорного ящика» в якості методів перевірки роботи додатку, розглянуто особливості цих видів тестування. Було розроблено 8 модульних тестів для різноманітних функцій ПЗ та 5 тест-кейсів для ручного тестування за методом «чорного ящика». За результатами тестування не було виявлено жодних проблем з програмним продуктом. Розроблено детальну інструкцію користувача, розглянуто мінімальну та рекомендовану конфігурації персонального комп’ютера для використання додатку.

ВИСНОВКИ

У бакалаврській дипломній роботі було розроблено програмний додаток для централізованого збору оповіщень з використанням хмарних безсерверних технологій під назвою «Notification Collector». Розроблене програмне забезпечення призначене для автоматизації отримання, зберігання та відображення оповіщень від різнотипних джерел, що повинно спростити процес роботи з великою кількістю вхідних даних для кінцевого користувача.

Проведено аналіз стану сучасних хмарних технологій, розглянуто переваги використання безсерверних розрахунків. Досліджено існуючі рішення для збору оповіщень, порівняно їх характеристики, підтверджено доцільність розробки власного програмного продукту. Проведено аналіз методів розв'язання поставлених завдань, виконано постановку задач розробки власного додатку.

Удосконалено метод отримання оповіщень з використанням вебхуків та публічних API, удосконалено метод отримання оповіщень з використанням парсингу веб-ресурсів, а також удосконалено метод генерації звітів на основі зібраних оповіщень. Для описаних методів розроблено відповідні алгоритми та їх блок-схеми.

Проведено варіантний аналіз та обґрунтування вибору програмних засобів для розробки програми, обрано мову програмування Python та середовище розробки JetBrains PyCharm. Докладно описано архітектуру серверної частини ПЗ. Розроблено та описано структурні схеми вікон додатку. Розглянуто реалізацію основних функцій додатку.

Розглянуто класифікацію видів тестування, обрано модульне тестування та тестування «чорного ящика» для перевірки роботи програмного продукту. Розроблено набір автоматизованих модульних тестів та ручних тест-кейсів. Тестування додатку довело працездатність основних функцій та відповідність поставленому технічному завданню. Розроблено інструкцію користувача та сформовано вимоги до персонального комп'ютера.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Jacob T. How data centres are enabling remote working. DATACOM. URL: https://datacom.com/au/en/discover/articles/blog-how-data-centres-are-enabling-remote-working.

2. Rodrigues A. Our brains are being overloaded with push notifications about nothing. Vice. URL: https://www.vice.com/en/article/a3a848/facebook-notification-overload.

3. Pielot M., Rello L. Productive, anxious, lonely - 24 hours without push notifications. MobileHCI '17: proceedings of the 19th international conference on human-computer interaction with mobile devices and services : Conference on Mobile Human-Computer Interaction, Vienna, 4–7 September 2017. New York, 2017. URL: https://pielot.org/pubs/PielotRello2017-MHCI-DoNotDisturb.pdf.

4. Третяк М. І., Миргородський А. В. Використання хмарних технологій для автоматизації потоків даних та реалізації персональних оповіщень. L Науково-технічна конференція підрозділів Вінницького національного технічного університету, м. Вінниця, 10–12 берез. 2021 р. Вінниця, 2021. URL: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2021/paper/view/11875/9948.

5. Третяк М. І., Ліщинська Л. Б. Розробка алгоритму аналізу онлайн-ресурсів для створення оповіщень. LI Науково-технічна конференція підрозділів Вінницького національного технічного університету, м. Вінниця, 31 трав. 2022 р. Вінниця, 2022. URL: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2022/paper/view/15497/13102.

6. Angell B. The hidden costs of your aging it infrastructure. Juriba. URL: https://blog.juriba.com/the-hidden-costs-of-an-aging-it-infrastructure.

7. Erl T., Puttini R., Mahmood Z. Cloud computing: concepts, technology & architecture. Upper Saddle River : Pearson, 2013. 528 p.

8. Carvalho L., Marden M. Fostering business and organizational transformation to generate business value with amazon web services. IDC. URL:

https://pages.awscloud.com/rs/112-TZM-766/images/AWS-BV%20IDC%202018.pdf?aliId=1614258770.

9. Daconta M. The great cloud migration: your roadmap to cloud computing, big data and linked data. Denver : Outskirts Press, 2013. 218 p.

10. Katzer J. Learning serverless: design, develop, and deploy with confidence. Sebastopol : O'Reilly Media, 2020. 232 p.

11. Dwyer G. Stateful vs stateless architecture: why stateless won. Virtasant. URL: https://www.virtasant.com/blog/stateful-vs-stateless-architecture-why-stateless-won.

12. Shift brings all of your work together. Shift. URL: https://tryshift.com/features/.

13. Pushover message API. Pushover. URL: https://pushover.net/api.

14. Notistory – you can read messages without notifying sender. URL: https://free-apps-android.com/notistory/.

15. Jin B., Sahni S., Shevat A. Designing web apis: building apis that developers love. Sebastopol : O'Reilly Media, 2018. 232 p.

16. Chia W. Push vs. pull in gitops: is there really a difference?. The New Stack. URL: https://thenewstack.io/push-vs-pull-in-gitops-is-there-really-a-difference/.

17. Horstmann C. S. Core java, volume I: fundamentals. Boston : Oracle Press, 2021. 944 p.

18. Price M. J. C# 8.0 and .NET core 3.0 – modern cross-platform development: build applications with C#, .NET core, entity framework core, ASP.NET core, and ML.NET using visual studio code. 4th ed. Birmingham : Packt Publishing, 2019. 818 p.

19. Casciaro M., Mammino L. Node.js Design Patterns: Design and implement production-grade Node.js applications using proven patterns and techniques. 3rd ed. Birmingham : Packt Publishing, 2020. 660 p.

20. Lubanovic B. Introducing python: modern computing in simple packages. Sebastopol : O'Reilly Media, 2019. 630 p.

21. Urooj W. What is python spyder IDE and how to use it?. URL: https://medium.com/edureka/spyder-ide-2a91caac4e46.

22. Zadrozny F. Navigating code using PyDev. Eclipse Foundation. URL: https://www.eclipse.org/community/eclipse_newsletter/2015/august/article4.php.

23. Rahmonov J. Pycharm for productive python development (guide). Real Python. URL: https://realpython.com/pycharm-guide/#pycharm-professional-features.

24. Reznik P., Dobson J., Gienow M. Cloud native transformation: practical patterns for innovation. Sebastopol : O'Reilly Media, 2019. 540 p.

25. DeBrie A. A detailed overview of AWS API gateway. URL: https://www.alexdebrie.com/posts/api-gateway-elements/.

26. Ryan P. Securing AWS API gateway with cognito user pools. SPR. URL: https://spr.com/securing-aws-api-gateway-with-congito-user-pools/.

27. Culkin J., Zazon M. AWS cookbook: recipes for success on AWS. Sebastopol : O'Reilly Media, 2021. 358 p.

28. Gifrin C., Dengler S. Messaging fanout pattern for serverless architectures using amazon SNS. AWS Compute Blog. URL: https://aws.amazon.com/ru/blogs/compute/messaging-fanout-pattern-for-serverless-architectures-using-amazon-sns/.

29. Galitz W. O. The essential guide to user interface design: an introduction to GUI design principles and techniques. 3rd ed. Indianapolis : Wiley, 2007. 888 p.

30. Chaudhary B. Tkinter GUI application development blueprints. Birmingham : Packt Publishing, 2015. 340 p.

31. Boto3 documentation. Boto3 Docs. URL: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.

32. What is QA in software testing?. 3Pillar Global. URL: https://www.3pillarglobal.com/insights/what-is-qa-in-software-testing/.

33. Oliveira B. PyTest Quick Start Guide: Write better Python code with simple and maintainable tests. Birmingham : Packt Publishing, 2018. 160 p.

ДОДАТКИ

Додаток А. Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедрою ПЗ
д.т.н., проф.
___________ О. Н. Романюк
"__" ____________ 2022 р.

Технічне завдання
на бакалаврську дипломну роботу «Розробка програмного забезпечення для централізованого збору оповіщень з використанням хмарних технологій» за спеціальністю
121 – Інженерія програмного забезпечення

Керівник бакалаврської дипломної роботи:
_____________ д.т.н., професор Л.Б. Ліщинська
"__" ____________ 2022 р.
Виконала:
__________ студент гр. 3ПІ-18б М.І. Третяк
"__" ____________ 2022 р.

Вінниця – 2022 року

1. Найменування та галузь застосування

Бакалаврська дипломна робота: «Розробка програмного забезпечення для централізованого збору оповіщень з використанням хмарних технологій».

Галузь застосування – персональний менеджмент.

2. Підстава для розробки.

Підставою для виконання бакалаврської дипломної роботи (БДР) є індивідуальне завдання на БДР та наказ № від «24» березня 2022 р. ректора по ВНТУ про закріплення тем БДР.

3. Мета та призначення розробки.

Метою бакалаврської роботи є підвищення ефективності роботи користувача з вхідними повідомленнями та оповіщеннями шляхом автоматизації та централізації процесу їх отримання.

Призначення роботи – розробка та програмна реалізація додатку для централізованого збору оповіщень.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БДР.

1. Reznik P., Dobson J., Gienow M. Cloud native transformation: practical patterns for innovation. Sebastopol : O'Reilly Media, 2019. 540 p.

2. Lubanovic B. Introducing python: modern computing in simple packages. 2nd ed. Sebastopol : O'Reilly Media, 2019. 630 p.

3. Chaudhary B. Tkinter GUI application development blueprints. Birmingham : Packt Publishing, 2015. 340 p.

4. Boto3 documentation. Boto3 Docs. URL: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.

5. Технічні вимоги

Модель розробки – ітеративна; вхідні дані – назва сервісу, правила аналізу онлайн-ресурсів, дані облікового запису; вихідні дані – текстові оповіщення від зареєстрованих сервісів, текстовий звіт, зображення діаграми звіту; середовище розробки – JetBrains PyCharm; мова програмування – Python.

6. Конструктивні вимоги

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред’являється по закінченню робіт:

1. Пояснювальна записка до БДР;
2. Технічне завдання;
3. Лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз завдання і вибір методу вирішення поставленої задачі дослідження	26.03.2022 – 01.04.2022	Вик.
2	Розробка алгоритму отримання оповіщень з використанням вебхуків та публічних API	02.04.2022 – 09.04.2022	Вик.
3	Розробка алгоритму отримання оповіщень з використанням парсингу веб-ресурсів	10.04.2022 – 18.04.2022	Вик.
4	Розробка алгоритму генерації звітів на основі зібраних оповіщень	19.04.2022 – 26.04.2022	Вик.
5	Аналіз і вибір мови програмування та середовища розробки	27.04.2022 – 01.05.2022	Вик.
6	Програмна реалізація додатку	02.05.2022 – 18.05.2022	Вик.
7	Тестування програмного забезпечення	19.05.2022 – 28.05.2022	Вик.
8	Оформлення матеріалів до захисту БДР	29.05.2022 – 10.06.2022	Вик.

10. Порядок контролю та прийняття

Виконання етапів бакалаврської дипломної роботи контролюється керівником згідно з графіком виконання роботи.

Прийняття бакалаврської дипломної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

Додаток Б. Протокол перевірки кваліфікаційної роботи на наявність текстових запозичень

ПРОТОКОЛ
ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: «Розробка програмного забезпечення для централізованого збору оповіщень з використанням хмарних технологій»

Тип роботи: БДР

Підрозділ : кафедра програмного забезпечення, ФІТКІ

Науковий керівник: Ліщинська Л. Б.

Оригінальність	97,8%
Схожість	2,2%

Аналіз звіту подібності

■ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**

□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку ____________ Черноволик Г. О.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck

Автор роботи ____________ Третяк М. І.

Керівник роботи ____________ Ліщинська Л. Б.

Додаток В. Лістинг програми

main.py

```
import ctypes
import tkinter as tk

import sv_ttk
from PIL import Image, ImageTk

from app import App

if __name__ == '__main__':
    ctypes.windll.shcore.SetProcessDpiAwareness(2)

    app_img = Image.open('icon.png')

    root = tk.Tk()
    sv_ttk.use_light_theme()
    root.wm_iconphoto(True, ImageTk.PhotoImage(app_img))
    root.title('Notification collector')

    app = App(root)
    app.pack()

    root.update_idletasks()
    root.minsize(root.winfo_width(), root.winfo_height())

    root.mainloop()
```

app.py

```
import time
import tkinter as tk
import uuid
from tkinter import ttk, messagebox

import requests
from PIL import Image, ImageTk
from PIL.ImageTk import PhotoImage

import create
import report
import user


class App(ttk.Frame):
    def __init__(self, parent: tk.Misc | None):
        ttk.Frame.__init__(self, parent)
        self.parent = parent

        self.web_img = ImageTk.PhotoImage(Image.open('world-wide-
web.png').resize((64, 64), Image.ANTIALIAS))
        self.github_img =
ImageTk.PhotoImage(Image.open('github.png').resize((64, 64), Image.ANTIALIAS))
        self.habitica_img = ImageTk.PhotoImage(Image.open('habitica-
logo.png').resize((64, 64), Image.ANTIALIAS))
        self.close_img = ImageTk.PhotoImage(Image.open('close.png').resize((16,
16), Image.ANTIALIAS))

        ttk.Button(self, text='Додати джерело', width=20, command=lambda:
create.CreateForm(self.parent, self.token)) \
```

```
            .grid(row=0, column=0, padx=10, pady=10)
        ttk.Button(self, text='Згенерувати звіт', width=20,
style='Accent.TButton',
                   command=lambda: report.ReportForm(self.parent, self.token)) \
            .grid(row=0, column=1, padx=10, pady=10)

        self.notif_frame = ttk.LabelFrame(self, text='Оповіщення', height=400)
        self.notif_frame.grid_propagate(False)
        self.notif_frame.grid(row=1, column=0, columnspan=2, sticky='nwes',
padx=10, pady=10)

        self.scroll_canvas = tk.Canvas(self.notif_frame, height=400)
        self.scroll_frame = ttk.Frame(self.scroll_canvas)
        self.scrollbar = ttk.Scrollbar(self.notif_frame, orient=tk.VERTICAL,
command=self.scroll_canvas.yview)
        self.scroll_canvas.configure(yscrollcommand=self.scrollbar.set)

        self.scrollbar.pack(side=tk.RIGHT, fill=tk.Y, padx=2, pady=2)
        self.scroll_canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=2,
pady=2)
        self.scroll_canvas.create_window(0, 0, window=self.scroll_frame,
anchor=tk.NW)

        self.scroll_frame.bind('<Configure>', self._scroll_frame_conf)
        self.scroll_frame.bind('<Enter>', self._bound_to_mousewheel)
        self.scroll_frame.bind('<Leave>', self._unbound_to_mousewheel)

        self.parent.update_idletasks()
        login_win = user.UserForm(self.parent)
        self.token = login_win.get_token()

        self.get_notif()

    def _scroll_frame_conf(self, event):

self.scroll_canvas.configure(scrollregion=self.scroll_canvas.bbox(tk.ALL))

    def _bound_to_mousewheel(self, event):
        self.scroll_canvas.bind_all("<MouseWheel>", self._on_mousewheel)

    def _unbound_to_mousewheel(self, event):
        self.scroll_canvas.unbind_all("<MouseWheel>")

    def _on_mousewheel(self, event):
        self.scroll_canvas.yview_scroll(int(-1 * (event.delta / 120)), tk.UNITS)

    def add_notif(self, service, text):
        img: PhotoImage = self.web_img
        if service == 'habitica':
            service = 'Habitica'
            img = self.habitica_img
        elif service == 'github':
            service = 'GitHub'
            img = self.github_img

        item = ttk.Frame(self.scroll_frame)
        ttk.Label(item, image=img).grid(row=0, column=0, padx=5, rowspan=2)
        ttk.Label(item, text=service, width=35).grid(row=0, column=1, padx=10)
        ttk.Label(item, text=text, width=35, wraplength=320).grid(row=1,
column=1, padx=10)
        ttk.Button(item, image=self.close_img, command=lambda:
item.destroy()).grid(row=0, column=2, rowspan=2)

        item.pack(pady=5, padx=5)
```

```
    def get_notif(self):
        with requests.Session() as s:
            s.headers.update({'Authorization': self.token})
            response = s.get(
                url='https://hyttrea95k.execute-api.eu-central-
1.amazonaws.com/notifications'
            )

            response_data = response.json()

            if response.status_code != 200:
                messagebox.showerror('Помилка', 'Виникла помилка при виклику
API: ' + str(response_data))
                self.destroy()
                return

            if response_data['count'] == 0:
                return

            for item in response_data['items']:
                self.add_notif(item['service'], item['text'])
```

user.py

```
import tkinter as tk
from tkinter import ttk, messagebox

import boto3 as boto3
from mypy_boto3_cognito_idp import CognitoIdentityProviderClient


class UserForm(tk.Toplevel):
    def __init__(self, parent):
        tk.Toplevel.__init__(self, parent)

        self.cognito: CognitoIdentityProviderClient = boto3.client('cognito-
idp', 'eu-central-1')
        self.client_id = '7f141iku26a83rou7l9h1td877'

        self.title("Реєстрація та вхід")
        self.geometry('350x400')

        self.token = None

        self.main_fr = ttk.Frame(self)
        self.main_fr.pack(padx=10, pady=20)

        self.mode = tk.IntVar()
        self.mode.trace_add('write', self.change_mode)
        ttk.Label(self.main_fr, text='Режим', anchor='w').grid(row=0, column=0,
pady=5, padx=5)
        ttk.Radiobutton(self.main_fr, text='Реєстрація', value=0,
variable=self.mode)\
            .grid(row=1, column=0, sticky=tk.W, pady=5, padx=5)
        ttk.Radiobutton(self.main_fr, text='Вхід', value=1, variable=self.mode)\
            .grid(row=2, column=0, sticky=tk.W, pady=5, padx=5)

        self.form_fr = ttk.Frame(self.main_fr)
        self.form_fr.grid(row=3, column=0)

        ttk.Label(self.form_fr, text='Ім\'я').grid(row=1, column=0, sticky='e')
        self.username = tk.StringVar()
```

```
        ttk.Entry(self.form_fr, width=20,
textvariable=self.username).grid(row=1, column=1)

        self.em_lb = ttk.Label(self.form_fr, text='Ел. пошта')
        self.em_lb.grid(row=2, column=0, sticky='e')
        self.email = tk.StringVar()
        self.em_ent = ttk.Entry(self.form_fr, width=20, textvariable=self.email)
        self.em_ent.grid(row=2, column=1)

        ttk.Label(self.form_fr, text='Пароль').grid(row=3, column=0, sticky='e')
        self.password = tk.StringVar()
        ttk.Entry(self.form_fr, width=20, show='\u25CF',
textvariable=self.password).grid(row=3, column=1)

        self.act_btn = ttk.Button(self.form_fr, text='Реєстрація',
command=self.sign_up, width=15)
        self.act_btn.grid(row=4, column=0, columnspan=2)

        for child in self.form_fr.winfo_children():
            child.grid_configure(padx=5, pady=5)

        self.grab_set()

    def change_mode(self, var, index, mode):
        if self.mode.get() == 0:
            self.act_btn.config(text='Реєстрація', command=self.sign_up)
            self.em_lb.grid()
            self.em_ent.grid()
        else:
            self.act_btn.config(text='Вхід', command=self.sign_in)
            self.em_lb.grid_remove()
            self.em_ent.grid_remove()

    def sign_up(self):
        if (not self.username.get().strip()
                or not self.email.get().strip()
                or not self.password.get().strip()):
            messagebox.showerror('Помилка', 'Будь ласка, заповність усі поля
форми!')
            return

        try:
            response = self.cognito.sign_up(
                ClientId=self.client_id,
                Username=self.username.get(),
                Password=self.password.get(),
                UserAttributes=[
                    {
                        'Name': 'email',
                        'Value': self.email.get()
                    }
                ]
            )
            print(response)
            if response['UserConfirmed'] is False:
                messagebox.showinfo('Реєстрація', 'Будь ласка, перевірне вашу
ел. пошту для підтвердження!')
        except Exception as e:
            messagebox.showerror('Помилка реєсрації', e)

    def sign_in(self):
        if not self.username.get().strip() or not self.password.get().strip():
            messagebox.showerror('Помилка', 'Будь ласка, заповність усі поля
форми!')
```

```
                return

            try:
                response = self.cognito.initiate_auth(
                    ClientId=self.client_id,
                    AuthFlow='USER_PASSWORD_AUTH',
                    AuthParameters={
                        'USERNAME': self.username.get(),
                        'PASSWORD': self.password.get()
                    }
                )
                print(response)
                self.token = response['AuthenticationResult']['AccessToken']
                messagebox.showinfo('Вхід', 'Вхід успішно виконано!')
                self.destroy()
            except Exception as e:
                messagebox.showerror('Помилка входу', e)

    def get_token(self):
        self.wait_window()
        return self.token
```

create.py

```
import json
import tkinter as tk
from tkinter import ttk, messagebox

import requests


class CreateForm(tk.Toplevel):
    def __init__(self, parent, token):
        tk.Toplevel.__init__(self, parent)

        self.title("Додати джерело")
        self.geometry('350x400')

        self.token = token

        self.main_fr = ttk.Frame(self)
        self.main_fr.pack(padx=10, pady=20)

        self.mode = tk.IntVar()
        self.mode.trace_add('write', self.change_mode)
        ttk.Label(self.main_fr, text='Тип', anchor='w').grid(row=0, column=0,
columnspan=2, pady=5, padx=5)
        ttk.Radiobutton(self.main_fr, text='Webhook', value=0,
variable=self.mode) \
            .grid(row=1, column=0, sticky=tk.W, pady=5, padx=5)
        ttk.Radiobutton(self.main_fr, text='Scrapper', value=1,
variable=self.mode) \
            .grid(row=1, column=1, sticky=tk.W, pady=5, padx=5)

        self.form_fr = ttk.Frame(self.main_fr)
        self.form_fr.grid(row=3, column=0, columnspan=2)

        # Webhook form
        self.webhook_fr = ttk.Frame(self.form_fr)
        self.webhook_fr.grid()

        ttk.Label(self.webhook_fr, text='Сервіс').grid(row=0, column=0, padx=5,
pady=5)
        self.service_cb = ttk.Combobox(self.webhook_fr, values=['Habitica',
```

```
'GitHub'])
        self.service_cb.current(0)
        self.service_cb.grid(row=0, column=1, padx=5, pady=5)
        ttk.Button(self.webhook_fr, text='Створити', command=self.create_wh_int)
\
            .grid(row=1, column=0, columnspan=2, padx=5, pady=5)
        self.wh_url = tk.StringVar(value='Створіть посилання...')
        ttk.Entry(self.webhook_fr, textvariable=self.wh_url, state='readonly') \
            .grid(row=2, column=0, columnspan=2, padx=5, pady=5)

        # Scrapper form
        self.scrapper_fr = ttk.Frame(self.form_fr)
        self.scrapper_fr.grid()

        ttk.Label(self.scrapper_fr, text='URL').grid(row=0, column=0, padx=5,
pady=5)
        self.scrapper_url = tk.StringVar()
        ttk.Entry(self.scrapper_fr, textvariable=self.scrapper_url).grid(row=0,
column=1, padx=5, pady=5)

        ttk.Label(self.scrapper_fr, text='Контейнер').grid(row=1, column=0,
padx=5, pady=5)
        self.scrapper_container = tk.StringVar()
        ttk.Entry(self.scrapper_fr,
textvariable=self.scrapper_container).grid(row=1, column=1, padx=5, pady=5)

        ttk.Label(self.scrapper_fr, text='Елемент').grid(row=2, column=0,
padx=5, pady=5)
        self.scrapper_item = tk.StringVar()
        ttk.Entry(self.scrapper_fr, textvariable=self.scrapper_item).grid(row=2,
column=1, padx=5, pady=5)
        ttk.Button(self.scrapper_fr, text='Створити',
command=self.create_sc_int) \
            .grid(row=3, column=0, columnspan=2, padx=5, pady=5)

        self.scrapper_fr.grid_remove()

        self.grab_set()

    def change_mode(self, var, index, mode):
        if self.mode.get() == 0:
            self.scrapper_fr.grid_remove()
            self.webhook_fr.grid()
        else:
            self.webhook_fr.grid_remove()
            self.scrapper_fr.grid()

    def create_wh_int(self):
        if self.token is None:
            messagebox.showerror('Доступ', 'Будь ласка, ввійдіть в акаунт для
використання додатку!')
            return

        service = 'habitica' if self.service_cb.current() == 0 else 'github'

        with requests.Session() as s:
            s.headers.update({'Authorization': self.token})
            s.headers.update({'Content-Type': 'application/json'})
            response = s.post(
                url='https://hyttrea95k.execute-api.eu-central-
1.amazonaws.com/create/webhook',
                data=json.dumps({
                    "service": service
                })
```

```
            )

            response_data = response.json()

            if response.status_code != 201:
                messagebox.showerror('Створення', 'Виникла помилка при виклику
API: ' + str(response_data))
                return

            self.wh_url.set(response_data['url'])
            messagebox.showinfo('Створення', 'URL згенеровано, можете продовжити
конфігурацію')

    def create_sc_int(self):
        if self.token is None:
            messagebox.showerror('Доступ', 'Будь ласка, ввійдіть в акаунт для
використання додатку!')
            return

        if (not self.scrapper_url.get().strip()
                or not self.scrapper_container.get().strip()
                or not self.scrapper_item.get().strip()):
            messagebox.showerror('Дані', 'Будь ласка, заповність усі поля
форми!')
            return

        with requests.Session() as s:
            s.headers.update({'Authorization': self.token})
            s.headers.update({'Content-Type': 'application/json'})
            response = s.post(
                url='https://hyttrea95k.execute-api.eu-central-
1.amazonaws.com/create/scrapper',
                data=json.dumps({
                    "url": self.scrapper_url.get().strip(),
                    "container": self.scrapper_container.get().strip(),
                    "item": self.scrapper_item.get().strip(),
                })
            )

            response_data = response.json()

            if response.status_code != 201:
                messagebox.showerror('Створення', 'Виникла помилка при виклику
API: ' + str(response_data))
                return

            messagebox.showinfo('Створення', 'Джерело додано!')
```

report.py

```
import json
import tkinter as tk
from tkinter import ttk, messagebox

import requests
from PIL import ImageTk, Image
from PIL.ImageTk import PhotoImage
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
from matplotlib.patches import Circle


class ReportForm(tk.Toplevel):
    def __init__(self, parent, token):
```

```
        tk.Toplevel.__init__(self, parent)

        self.web_img = ImageTk.PhotoImage(Image.open('world-wide-
web.png').resize((64, 64), Image.ANTIALIAS))
        self.github_img =
ImageTk.PhotoImage(Image.open('github.png').resize((64, 64), Image.ANTIALIAS))
        self.habitica_img = ImageTk.PhotoImage(Image.open('habitica-
logo.png').resize((64, 64), Image.ANTIALIAS))
        self.close_img = ImageTk.PhotoImage(Image.open('close.png').resize((16,
16), Image.ANTIALIAS))

        self.title("Звіт")
        # self.geometry('350x400')

        self.token = token

        labels, data, items = self.get_data()

        self.main_fr = ttk.Frame(self)
        self.main_fr.pack(padx=10, pady=20)

        # Pie chart
        fig = Figure(figsize=(4, 3))
        colors = ['#EDE3DB', '#ECB99C', '#BE9593', '#F1D7D6', '#6E6D71']
        ax = fig.add_subplot(111)
        ax.pie(data, labels=labels, autopct='%1.1f%%',
               startangle=90, colors=colors,
               pctdistance=0.8, textprops={'fontsize': 8}, radius=0.8)

        centre_circle = Circle((0, 0), 0.4, fc='white')
        ax.add_patch(centre_circle)

        chart_pie = FigureCanvasTkAgg(fig, self.main_fr)
        chart_pie.get_tk_widget().grid(row=0, column=0, padx=5, pady=5)

        # Top
        top_fr = ttk.Frame(self.main_fr)
        ttk.Label(top_fr, text='Топ:').pack(padx=5, pady=5)
        for lab in labels:
            ttk.Label(top_fr, text=lab).pack(padx=5, pady=5)
        top_fr.grid(row=0, column=1, padx=5, pady=5)

        # All
        self.notif_frame = ttk.LabelFrame(self.main_fr, text='Оповіщення',
height=400)
        self.notif_frame.grid_propagate(False)
        self.notif_frame.grid(row=1, column=0, columnspan=2, sticky='nwes',
padx=10, pady=10)

        self.scroll_canvas = tk.Canvas(self.notif_frame, height=400)
        self.scroll_frame = ttk.Frame(self.scroll_canvas)
        self.scrollbar = ttk.Scrollbar(self.notif_frame, orient=tk.VERTICAL,
command=self.scroll_canvas.yview)
        self.scroll_canvas.configure(yscrollcommand=self.scrollbar.set)

        self.scrollbar.pack(side=tk.RIGHT, fill=tk.Y, padx=2, pady=2)
        self.scroll_canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=2,
pady=2)
        self.scroll_canvas.create_window(0, 0, window=self.scroll_frame,
anchor=tk.NW)

        self.scroll_frame.bind('<Configure>', self._scroll_frame_conf)
        self.scroll_frame.bind('<Enter>', self._bound_to_mousewheel)
        self.scroll_frame.bind('<Leave>', self._unbound_to_mousewheel)
```

```
        for item in items:
            self.add_notif(item['service'], item['text'])

        self.grab_set()

    def _scroll_frame_conf(self, event):
self.scroll_canvas.configure(scrollregion=self.scroll_canvas.bbox(tk.ALL))

    def _bound_to_mousewheel(self, event):
        self.scroll_canvas.bind_all("<MouseWheel>", self._on_mousewheel)

    def _unbound_to_mousewheel(self, event):
        self.scroll_canvas.unbind_all("<MouseWheel>")

    def _on_mousewheel(self, event):
        self.scroll_canvas.yview_scroll(int(-1 * (event.delta / 120)), tk.UNITS)

    def add_notif(self, service, text):
        item = ttk.Frame(self.scroll_frame)

        img: PhotoImage = self.web_img
        if service == 'habitica':
            service = 'Habitica'
            img = self.habitica_img
        elif service == 'github':
            service = 'GitHub'
            img = self.github_img

        ttk.Label(item, image=img).grid(row=0, column=0, padx=5, rowspan=2)
        ttk.Label(item, text=service, width=55).grid(row=0, column=1, padx=10)
        ttk.Label(item, text=text, width=55, wraplength=500).grid(row=1,
column=1, padx=10)
        ttk.Button(item, image=self.close_img, command=lambda:
item.destroy()).grid(row=0, column=2, rowspan=2)

        item.pack(pady=5, padx=5)

    def get_data(self):
        with requests.Session() as s:
            s.headers.update({'Authorization': self.token})
            s.headers.update({'Content-Type': 'application/json'})
            response = s.post(
                url='https://hyttrea95k.execute-api.eu-central-
1.amazonaws.com/report',
                data=json.dumps({
                    "limit": 50
                })
            )

            response_data = response.json()

            if response.status_code != 200:
                messagebox.showerror('Помилка', 'Виникла помилка при виклику
API: ' + str(response_data))
                self.destroy()
                return

            if response_data['count'] == 0:
                messagebox.showerror('Дані', 'Недостатньо даних для формування
звіту!')
                self.destroy()
                return
```

```
            labels = []
            data = []
            for record in response_data['top']:
                labels.append(record[0])
                data.append(record[1])

            return labels, data, response_data['items']
```

add_user\lambda_function.py

```
import boto3

dynamodb = boto3.resource('dynamodb')


def lambda_handler(event, context):
    try:
        users = dynamodb.Table('Users')
        users.put_item(Item={
            'sub': event['request']['userAttributes']['sub'],
            'webhooks': [],
            'scrappers': []
        })

    except Exception as e:
        print(str(e))

    return event
```

create_wh_int\lambda_function.py

```
import json
import boto3

def lambda_handler(event, context):
    dynamodb = boto3.resource('dynamodb')

    user_id = event['requestContext']['authorizer']['jwt']['claims']['sub']

    request_body = json.loads(event['body'])
    service = request_body['service']

    try:
        users_table = dynamodb.Table('Users')
        users_table.update_item(
            Key={'sub': user_id},
            UpdateExpression='set webhooks = list_append(webhooks, :i)',
            ExpressionAttributeValues={
                ':i': [service],
            }
        )
    except Exception as e:
        print('Failed to update user: ' + str(e))
        return {
            'statusCode': 500,
            'body': json.dumps({'message': 'Failed to update user data'})
        }

    responce = {
        'message': 'Success',
        'url': 'https://hyttrea95k.execute-api.eu-central-
1.amazonaws.com/webhook-collector?user_id=' + user_id + '&service=' + service
    }
```

```
    return {
        'statusCode': 201,
        'body': json.dumps(responce)
    }
```

create_sc_int\lambda_function.py

```
import json
import uuid

import boto3

dynamodb = boto3.resource('dynamodb')


def lambda_handler(event, context):
    user_id = event['requestContext']['authorizer']['jwt']['claims']['sub']

    request_body = json.loads(event['body'])
    url = request_body['url']
    container = request_body['container']
    item = request_body['item']

    try:
        rules_table = dynamodb.Table('ScrapeRules')
        rules_table.put_item(Item={
            'rule_id': str(uuid.uuid4()),
            'user_id': user_id,
            'url': url,
            'container_selector': container,
            'items': [item],
            'previous': [],
        })
    except Exception as e:
        print('Failed to add new rule: ' + str(e))
        return {
            'statusCode': 500,
            'body': json.dumps({'message': 'Failed to create new rule'})
        }

    return {
        'statusCode': 201,
        'body': json.dumps({'message': 'Success'})
    }
```

generate_report\lambda_function.py

```
import collections
import json

import boto3

dynamodb = boto3.resource('dynamodb')


def lambda_handler(event, context):
    user_id = event['requestContext']['authorizer']['jwt']['claims']['sub']

    input_data = json.loads(event['body'])
    count_limit = input_data.get('limit', 200)
    filters = boto3.dynamodb.conditions.Attr('user_id').eq(user_id)

    response = None
```

```
    try:
        n_table = dynamodb.Table('Notifications')
        response = n_table.scan(
            Limit=count_limit,
            FilterExpression=filters
        )
    except Exception as e:
        print('Failed to get rules from DynamoDB: ' + str(e))
        return {
            'statusCode': 200,
            'body': json.dumps({
                'count': 0,
                'msg': 'Failed to get notifications'
            })
        }

    if response['Count'] == 0:
        return {
            'statusCode': 200,
            'body': json.dumps({
                'count': 0,
                'msg': 'Not enough notification for analysis'
            })
        }

    counter = collections.Counter()
    for notif in response['Items']:
        counter.update({notif['service']: 1})

    return {
        'statusCode': 200,
        'body': json.dumps({
            'count': response['Count'],
            'msg': 'Success',
            'top': counter.most_common(10),
            'items': response['Items']
        }, ensure_ascii=False)
    }
```

get\lambda_function.py

```
import collections
import json

import boto3

dynamodb = boto3.resource('dynamodb')


def lambda_handler(event, context):
    user_id = event['requestContext']['authorizer']['jwt']['claims']['sub']

    filters = boto3.dynamodb.conditions.Attr('user_id').eq(user_id)

    response = None
    try:
        n_table = dynamodb.Table('Notifications')
        response = n_table.scan(
            Limit=200,
            FilterExpression=filters
        )
    except Exception as e:
        print('Failed to get rules from DynamoDB: ' + str(e))
        return {
```

```
            'statusCode': 200,
            'body': json.dumps({
                'count': 0,
                'msg': 'Failed to get notifications'
            })
        }

    return {
        'statusCode': 200,
        'body': json.dumps({
            'count': response['Count'],
            'msg': 'Success',
            'items': response['Items']
        }, ensure_ascii=False)
    }
```

scrapper\lambda_function.py

```
import ast
import uuid

import boto3
import requests
from bs4 import BeautifulSoup

dynamodb = boto3.resource('dynamodb')


def lambda_handler(event, context):
    for record in event['Records']:
        # Get payload
        payload = ast.literal_eval(record["body"])

        # Parse page
        page = requests.get(payload['url'])
        soup = BeautifulSoup(page.content, 'html.parser')
        container = soup.select(payload['container_selector'])
        notif_list = []
        for item in container:
            notif_text = ""
            for rule in payload['items']:
                rule_res = item.select_one(rule).string
                if rule_res is not None:
                    notif_text += (rule_res + ' ')
            notif_list.append(notif_text.strip())

        # Get only new items
        diff = [x for x in notif_list if x not in payload['previous']]
        if len(diff) == 0:
            print('No new items')
            return {
                'statusCode': 200
            }

        # Send new items
        try:
            results_table = dynamodb.Table('Notifications')
            for notif in diff:
                results_table.put_item(Item={
                    'id': str(uuid.uuid4()),
                    'service': payload['url'],
                    'user_id': payload['user_id'],
                    'text': notif
                })
```

```
        except Exception as e:
            print('Failed to add new notification: ' + str(e))

        # Update previous list
        try:
            rules_table = dynamodb.Table('ScrapeRules')
            rules_table.update_item(
                Key={'rule_id': payload['rule_id']},
                UpdateExpression='set previous = :r',
                ExpressionAttributeValues={
                    ':r': notif_list,
                }
            )
        except Exception as e:
            print('Failed to update previous list: ' + str(e))

    return {
        'statusCode': 200
    }
```

scrapper_targets\lambda_function.py

```
import boto3

dynamodb = boto3.resource('dynamodb')
sqs = boto3.resource('sqs')


def lambda_handler(event, context):
    # Get all rules
    rules = None
    try:
        rules_table = dynamodb.Table('ScrapeRules')
        response = rules_table.scan()

        if response['Count'] == 0:
            print('No rules in table')
            return

        rules = response['Items']

    except Exception as e:
        print('Failed to get rules from DynamoDB: ' + str(e))

    # Send items to SQS
    try:
        queue = sqs.get_queue_by_name(QueueName='notification-collector-
targets')

        max_batch_size = 10
        chunks = [rules[x:x + max_batch_size] for x in range(0, len(rules),
max_batch_size)]
        for chunk in chunks:
            entries = []
            for x in chunk:
                entry = {
                    'Id': x['rule_id'],
                    'MessageBody': str(x)
                }
                entries.append(entry)
            resp = queue.send_messages(Entries=entries)

            if len(resp.get('Failed', [])) != 0:
                print('Failed to send some rules to SQS!')
```

```
    except Exception as e:
        print('Failed to send rules to SQS: ' + str(e))
```

webhooks\lambda_function.py

```
import json
import boto3
import uuid

dynamodb = boto3.resource('dynamodb')


def lambda_handler(event, context):
    # Parameters
    service = None
    user_id = None

    # Get params
    try:
        service = event['queryStringParameters']['service']
        user_id = event['queryStringParameters']['user_id']
    except KeyError as e:
        print('Failed to get input parameters: ' + str(e))
        return {
            'statusCode': 403
        }

    # Check user_id
    try:
        users = dynamodb.Table('Users')
        query_res = users.query(

KeyConditionExpression=boto3.dynamodb.conditions.Key('sub').eq(user_id)
        )

        if query_res['Count'] != 1:
            print('Failed to get user with sub: ' + user_id)
            return {
                'statusCode': 403
            }

        if service not in query_res['Items'][0]['webhooks']:
            print('Service ' + service + ' is not in user webhooks list: ' +
user_id)
            return {
                'statusCode': 403
            }

    except Exception as e:
        print('Failed to query users: ' + str(e))
        return {
            'statusCode': 500
        }

    # Get fields and actions for service
    notification_text = None
    try:
        triggers = dynamodb.Table('Triggers')

        service_trigger = triggers.get_item(Key={'Service': service})
        fields = service_trigger['Item']['fields']
        actions = service_trigger['Item']['actions']
```

```
    for k, v in fields.items():
        fields[k] = eval(v, {}, {'input': json.loads(event['body'])})

    notification_text = eval(actions, {}, fields)
except Exception as e:
    print('Failed to process notification: ' + str(e))
    return {
        'statusCode': 500
    }

# Create notification
try:
    results = dynamodb.Table('Notifications')
    results.put_item(Item={
        'id': str(uuid.uuid4()),
        'service': service,
        'user_id': user_id,
        'text': notification_text
    })
except Exception as e:
    print('Failed to add new notification: ' + str(e))
    return {
        'statusCode': 500
    }

return {
    'statusCode': 200
}
```

Додаток Г. Ілюстративний матеріал

ГРАФІЧНА ЧАСТИНА

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЦЕНТРАЛІЗОВАНОГО ЗБОРУ ОПОВІЩЕНЬ З ВИКОРИСТАННЯМ ХМАРНИХ ТЕХНОЛОГІЙ

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення

Бакалаврська дипломна робота на тему:
«РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЦЕНТРАЛІЗОВАНОГО ЗБОРУ ОПОВІЩЕНЬ З ВИКОРИСТАННЯМ ХМАРНИХ ТЕХНОЛОГІЙ»

Автор:
студент групи ЗПІ-18б Третяк М.І.
Науковий керівник:
д.т.н., професор каф. ПЗ Ліщинська Л.Б.

Вінниця - 2022

Рисунок В.1 – Титульний слайд

Актуальність теми

Наразі існує певна кількість продуктів для збору оповіщень, але всі вони мають свої недоліки та обмеження, як-от відсутність функціоналу для створення власних інтеграцій з сервісами та генерації звітів або ж обмежена підтримка десктопних ОС. Використання хмарних технологій у свою чергу дозволяє спростити проектування архітектури додатку, зменшити витрати на інфраструктуру та полегшити масштабування та подальшу підтримку ПЗ.

Тому актуальним є питання підвищення ефективності роботи користувачів з вхідними оповіщеннями шляхом розробки сучасного програмного забезпечення для централізованого збору оповіщень, яке може використовуватися у сфері персонального менеджменту людини.

2

Рисунок В.2 – Актуальність теми

Мета, об’єкт та предмет дослідження

- **Мета дослідження** – підвищення ефективності роботи користувача з вхідними повідомленнями та оповіщеннями шляхом автоматизації та централізації процесу їх отримання.
- **Об’єкт дослідження** – процеси отримання, зберігання та надсилання оповіщень від різноманітних сервісів та інтернет-ресурсів.
- **Предмет дослідження** – методи та засоби централізованого збору оповіщень.

3

Рисунок В.3 – Мета, об’єкт та предмет дослідження

Задачі дослідження

Відповідно до поставленої мети в бакалаврській дипломній роботі потрібно вирішити такі завдання:

- провести аналіз сучасних технологій хмарних безсерверних розрахунків;
- провести аналіз існуючих методів отримання оповіщень;
- розробити метод і алгоритм отримання оповіщень з використанням вебхуків та публічних API;
- розробити метод і алгоритм отримання оповіщень з використанням парсингу веб-ресурсів;
- розробити метод і алгоритм генерації звітів на основі зібраних оповіщень;
- розробити структуру та графічний інтерфейс користувача для створюваного програмного продукту;
- розробити програмні компоненти додатку для централізованого збору оповіщень;
- розробити модульні тести для автоматизованого тестування;
- провести ручне тестування розробленого програмного забезпечення;
- розробити інструкцію користувача.

4

Рисунок В.4 – Задачі дослідження

Рисунок В.5 – Наукова новизна

Рисунок В.6 – Практична цінність одержаних результатів

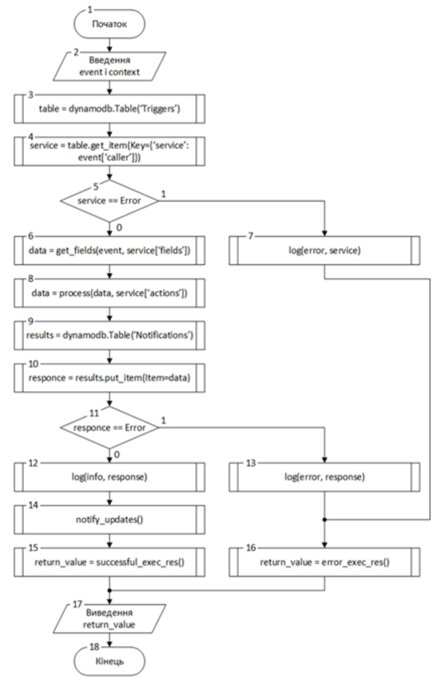
Порівняльний аналіз аналогів

Критерій	Shift	Pushover	Notistory	Notification Collector
Каталог готових інтеграцій	1	1	0,5	1
Створення власних інтеграцій	0	0,5	0	1
Підтримка десктопних ОС	1	0,5	0	1
Підтримка мобільних ОС	0	1	1	0
Генерація звітів на основі зібраних оповіщень	0	0	0,5	1
Підсумок	2	3	2	4

7

Рисунок В.7 – Порівняльний аналіз аналогів

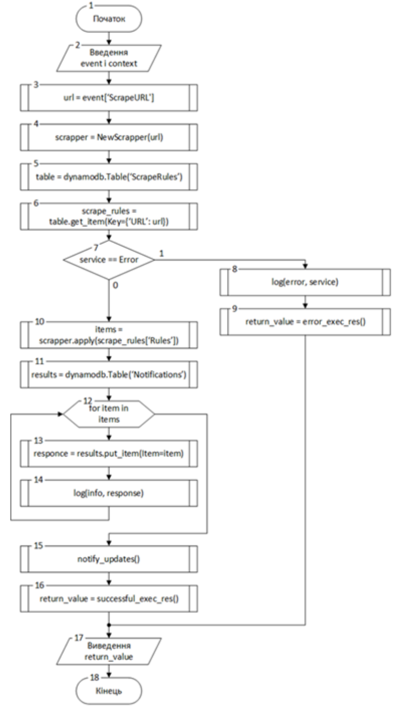
Блок-схема алгоритму отримання оповіщень з використанням вебхуків та публічних API



8

Рисунок В.8 – Блок-схема алгоритму отримання оповіщень з використанням вебхуків та публічних API

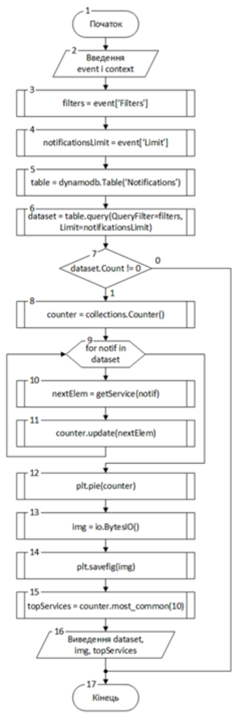
Блок-схема алгоритму отримання оповіщень з використанням парсингу веб-ресурсів



9

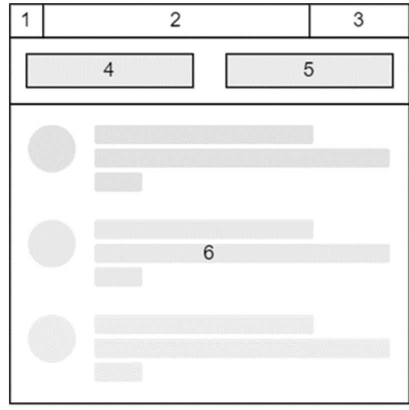
Рисунок В.9 – Блок-схема алгоритму отримання оповіщень з використанням парсингу вебресурсів

Блок-схема алгоритму генерації звітів на основі зібраних оповіщень



10

Рисунок В.10 – Блок-схема алгоритму генерації звітів на основі зібраних оповіщень



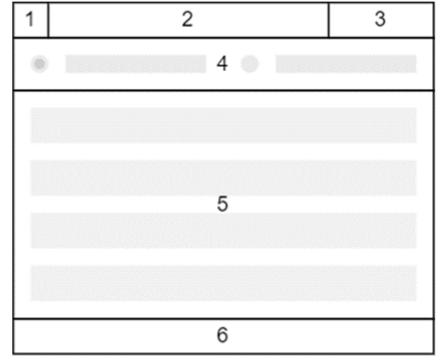
Структура графічного інтерфейсу головного вікна додатку

Головне вікно включає наступні елементи інтерфейсу:

1) іконка програми;
2) стрічка заголовку;
3) системні кнопки керування вікном;
4) кнопка для додавання нового джерела сповіщень;
5) кнопка для генерації звіту;
6) список останніх отриманих оповіщень.

11

Рисунок В.11 – Структура графічного інтерфейсу головного вікна додатку



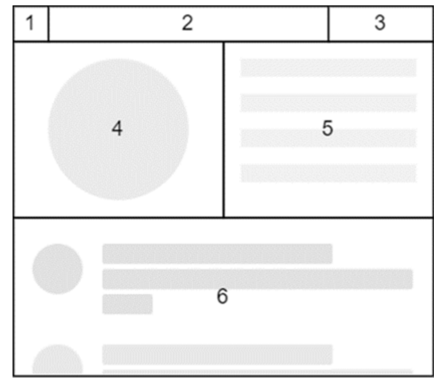
Структура графічного інтерфейсу вікна створення джерела сповіщень

Вікно включає в себе наступні елементи інтерфейсу:

1) іконка програми;
2) стрічка заголовку;
3) системні кнопки керування вікном;
4) радіокнопки для вибору типу джерела;
5) список полів для введення параметрів джерела;
6) рядок стану.

12

Рисунок В.12 – Структура графічного інтерфейсу вікна створення джерела сповіщень



Структура графічного інтерфейсу вікна звіту

Вікно звіту включає такі елементи інтерфейсу:

1) іконка програми;
2) стрічка заголовку;
3) системні кнопки керування вікном;
4) секторна діаграма;
5) список найбільш кількісних джерел;
6) список оповіщень, що включені до звіту.

13

Рисунок В.13 – Структура графічного інтерфейсу вікна звіту

```python
def sign_in(self):
    if not self.username.get().strip() or not self.password.get().strip():
        messagebox.showerror('Error', 'Please fill in all fields!')
        return

    try:
        response = cognito.initiate_auth(
            ClientId=client_id,
            AuthFlow='USER_PASSWORD_AUTH',
            AuthParameters={
                'USERNAME': self.username.get(),
                'PASSWORD': self.password.get()
            }
        )
        print(response)
        self.token = response['AuthenticationResult']['AccessToken']
        messagebox.showinfo('Sign-in info', 'Successfully logged in!')
    except Exception as e:
        messagebox.showerror('Sign-in error', e)
```

Лістинг

Метод «sign_in» відповідає за вхід користувача в обліковий запис.

Відбувається перевірка введених даних та спроба ввійти в акаунт за допомогою функцій Boto3 – офіційної AWS SDK для Python.

14

Рисунок В.14 – Лістинг

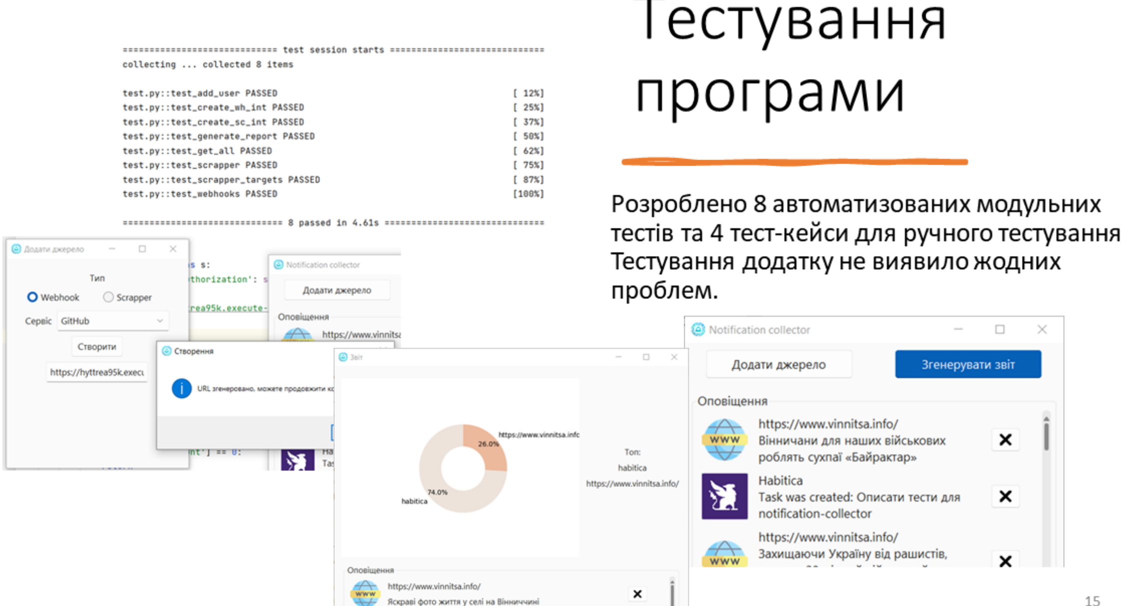


Рисунок В.15 – Тестування програми

Апробація та публікації результатів роботи

Результати роботи доповідалися на:

- **L Науково-технічній конференції підрозділів Вінницького національного технічного університету** (2021 р., м. Вінниця);
- **LI Науково-технічній конференції підрозділів Вінницького національного технічного університету** (2022 р., м. Вінниця).

За тематикою дослідження опубліковано 2 наукових праці у збірниках матеріалів конференцій.

16

Рисунок В.16 – Апробація та публікації результатів роботи

Висновки

- У бакалаврській дипломній роботі було розроблено програмний додаток для централізованого збору оповіщень з використанням хмарних безсерверних технологій під назвою «Notification Collector».
- Проведено аналіз стану сучасних хмарних технологій та методів отримання оповіщень.
- Удосконалено метод отримання оповіщень з використанням вебхуків та публічних API.
- Удосконалено метод отримання оповіщень з використанням парсингу веб-ресурсів.
- Удосконалено метод генерації звітів на основі зібраних оповіщень.
- Використано мову програмування Python та середовище розробки JetBrains PyCharm для створення додатку.
- Розроблено набір автоматизованих модульних тестів та ручних тест-кейсів. Тестування додатку довело працездатність основних функцій та відповідність поставленому технічному завданню.

17

Рисунок В.17 – Висновки