Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра автоматизації та інтелектуальних інформаційних технологій

Бакалаврська кваліфікаційна робота на тему:

«РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ АВТОМАТИЗОВАНОЇ РЕЄСТРАЦІЇ ТА УПРАВЛІННЯ АКАУНТАМИ НА ПЛАТФОРМІ GRASS»

Виконав: студент IV курсу, групи 1ІСТ-21б
спеціальності 126 – Інформаційні системи та технології
(шифр і назва спеціальності)
Олександр ТАНАСКОВ
(прізвище та ініціали)
Керівник: к.т.н., доцент кафедри КСУ
Ольга СОФИНА
(прізвище та ініціали)
«16» 06 2025 р.
Рецензент: к.т.н., доцент кафедри КСУ
Олег КОВАЛЮК
(прізвище та ініціали)
«17» 06 2025 р.

Допущено до захисту
завідувач кафедри АІТ
Олег БІСІКАЛО
«19» 06 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
(повне найменування вищого навчального закладу)

Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра автоматизації та інтелектуальних інформаційних технологій
Рівень вищої освіти перший (бакалаврський)
Галузь знань 12 – Інформаційні технології
Спеціальність 126 – Інформаційні системи та технології
Освітньо-професійна програма Інтелектуальні інформаційні системи

ЗАТВЕРДЖУЮ
зав. кафедри АІТ
д.т.н., проф. Олег БІСІКАЛО
«19» 06 2025 року

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Танаскову Олександру Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Розробка інформаційної системи автоматизованої реєстрації та управління акаунтами на платформі Grass»
керівник роботи Бісікало Олег Володимирович, д.т.н, професор
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від «20» березня 2025 року №97
2. Термін подання студентом роботи 10.06.2025 р.
3. Вихідні дані до роботи:
API-документація платформи Grass, формати облікових записів та проксі, вимоги до структури багатопотокового фармінгу, сервісні обмеження антикапча, специфікації для підключення WebSocket та REST-запитів.
4. Провести аналіз сучасних методів автоматизації взаємодії з веб-платформами, реалізувати програмний комплекс для реєстрації, підтвердження акаунтів, підключення криптогаманців та збирання поінтів. Розробити архітектуру системи, модулі обробки сесій, логування та захищеного зберігання даних. Провести тестування функціональності, продуктивності та безпеки.
5. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень): алгоритм роботи програми, загальна структурна схема програми, UML-діаграми розробленої системи, діаграма потоків обробки акаунтів, знімки екрану програмного інтерфейсу та логів виконання

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Софина О. Ю., доц. каф. АІІТ		

7. Дата видачі завдання 20.03.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Термін виконання		Примітки
		початок	закінчення	
1	Вибір, узгодження та затвердження теми БКР	11.02.2025	20.03.2025	вик
2	Аналіз літературних джерел. Розробка загальної структури системи	17.03.2025	26.04.2025	вик
3	Розробка технічного завдання (ТЗ)	05.05.2025	10.05.2025	вик
4	Реалізація модулів реєстрації, авторизації та підтвердження акаунтів	12.05.2025	17.05.2025	вик
5	Розробка структурної схеми	19.05.2025	20.05.2025	вик
6	Побудова UML-діаграм, тестування та аналіз результатів	21.05.2025	22.05.2025	вик
7	Оформлення пояснювальної записки та додатків	22.05.2025	24.05.2025	вик
8	Перевірка ПЗ на відповідність вимогам	02.06.2025	07.06.2025	вик
9	Попередній захист, рецензування, доопрацювання	10.06.2025	15.06.2025	вик
10	Захист БКР	16.06.2025	23.06.2025	вик

Студент _____ Олександр ТАНАСКОВ
(підпис) (прізвище та ініціали)

Керівник роботи _____ Ольга СОФИНА
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська дипломна робота складається з 86 сторінок формату А4 в тому числі і 3 додатків, на яких є 22 рисунків, 1 таблиць, список використаних джерел містить 20 найменувань.

У дипломній роботі розглянуто процес розробки інформаційної системи автоматизованої реєстрації та управління акаунтами на платформі Grass. Представлено архітектуру програмного забезпечення, реалізовано ключові модулі для масової реєстрації облікових записів, підключення криптогаманців, підтвердження поштових адрес, а також автоматичного фармінгу поінтів. Система підтримує багатопоточну обробку, що дозволяє ефективно масштабувати кількість акаунтів. Для зберігання даних використано комбінований підхід – текстові файли та база даних SQLite. Проведено тестування стабільності, продуктивності та безпеки програмного продукту. Наведено порівняння з існуючими аналогами та обґрунтовано переваги розробленої системи в умовах реального навантаження.

Ключові слова: Автоматизація, Масова реєстрація акаунтів, Багатопоточність, Фармінг поінтів, Grass-платформа, Підключення криптогаманця, CAPTCHA-обхід, WebSocket, REST API.

ANNOTATION

The bachelor’s thesis consists of 86 pages in A4 format, including 3 appendices, which contain 22 pictures, 1 table, and the reference list includes 20 entries.

The thesis covers the process of developing an information system for automated registration and account management on the Grass platform. It presents the software architecture, implements key modules for mass account registration, cryptocurrency wallet [10] connection, email address verification, and automatic points farming. The system supports multithreaded processing, allowing efficient scaling of the number of accounts. A combined approach is used for data storage – text files and an SQLite database. Stability, performance, and security testing of the software product have been conducted. A comparison with existing analogs is provided, and the advantages of the developed system in real-world load conditions are justified.

Keywords: Automation, Mass account registration, Multithreading, Points farming, Grass platform, Cryptocurrency wallet connection, CAPTCHA bypass, WebSocket, REST API.

ЗМІСТ

ВСТУП

Актуальність розробки інформаційних систем, що автоматизують взаємодію з вебплатформами, стрімко зростає в умовах цифрової економіки, зокрема на тлі розвитку блокчейн-технологій, Web3-інфраструктур та нових підходів до децентралізації інтернету. У новій реальності цифрових взаємодій все більшу популярність здобувають платформи, які дозволяють користувачам монетизувати власні ресурси – як фізичні (інтернет-з’єднання), так і інформаційні (активність, участь у програмах). Яскравим прикладом такої платформи є Grass, яка створена як децентралізована екосистема для збору публічних веб-даних із подальшим використанням їх у сфері штучного інтелекту, та яка винагороджує користувачів токенами за участь у процесі збору.

З урахуванням високої популярності платформи Grass та великого попиту на участь в її екосистемі, виникає об'єктивна потреба у масовій автоматизації процесів, пов’язаних із реєстрацією акаунтів, підключенням Web3-гаманців, верифікацією поштових адрес, а також запуском механізмів фармінгу поінтів. При цьому особливої ваги набуває масштабованість – здатність обробляти сотні або навіть тисячі акаунтів одночасно з мінімальним залученням людини.

Враховуючи обмеження, що запроваджує сама платформа Grass (наприклад, контроль сесій, перевірки IP, верифікація через email або wallet), стає очевидним, що ефективна участь без автоматизації майже неможлива. З іншого боку, надмірна автоматизація без урахування цих обмежень призводить до блокування акаунтів, втрати поінтів або виключення з аірдропу. Тому на перший план виходить завдання створення інтелектуальної системи, яка буде не лише автоматизувати дії користувача, а й адаптуватися до умов самої платформи, дотримуючись її політик безпеки.

Метою цієї роботи є підвищення ефективності автоматизованої взаємодії з платформою Grass шляхом розробки відповідного програмного комплексу.

Вона повинна підтримувати:

- масову реєстрацію акаунтів із використанням антикапча-сервісів;

- імпорт та керування проксі;
- підтвердження email через IMAP-протокол;
- підключення Web3-гаманців;
- запуск процесу фармінгу через WebSocket-з’єднання;
- збереження статистики та логування всіх дій.

Для досягнення мети були поставлені та реалізовані такі **завдання**:

- досліджено предметну галузь, зокрема платформу Grass, її архітектуру, токеноміку та вимоги до акаунтів;
- здійснено аналіз існуючих рішень у сфері автоматизації Web3-реєстрацій та фармінгу;
- спроєктовано власну архітектуру системи з поділом на модулі реєстрації (на Python) та майнінгу (на Node.js);
- реалізовано логіку масштабованого запуску до 65 000 акаунтів одночасно;
- проведено тестування системи в реальному середовищі та оцінено її продуктивність.

Об’єктом дослідження виступає процес масової автоматизованої взаємодії з платформою Grass.

Предмет дослідження – технології розробки масштабованих багатопотокових систем для роботи з децентралізованими Web3-сервісами.

Практичне значення цієї роботи полягає в тому, що створена система може бути використана не лише як засіб особистого фармінгу, а й як інструмент для командного заробітку, проведення маркетингових кампаній, побудови «ферм акаунтів» або навіть створення SaaS-сервісу для оренди акаунтів. Вона є гнучкою, масштабованою, адаптованою до нових вимог і побудована на відкритих технологіях, що дає змогу швидко модифікувати її під нові платформи або оновлення самої Grass.

1 ОСОБЛИВОСТІ ОБ’ЄКТА ДОСЛІДЖЕННЯ

1.1 Платформа Grass та принцип її роботи

Платформа Grass являє собою інноваційну децентралізовану систему, яка надає користувачам можливість монетизувати невикористану пропускну здатність власного інтернет-з’єднання. Замість того щоб безоплатно ділитися своїми ресурсами з великими компаніями, учасники екосистеми можуть отримувати винагороду у вигляді токенів GRASS за участь у функціонуванні мережі.

Ключовий принцип платформи полягає у побудові децентралізованої інфраструктури для збору та обробки публічних веб-даних, які згодом використовуються для навчання моделей штучного інтелекту. Встановивши клієнтський додаток Grass, користувач надає мережі дозвіл використовувати частину свого інтернет-каналу для отримання даних, при цьому зберігаючи повний контроль над персональною інформацією та дотриманням конфіденційності.

Архітектура платформи базується на блокчейні Solana, що забезпечує високу швидкодію, низькі комісії та ефективну масштабованість. Додатково, впровадження технології Zero-Knowledge proofs (ZK) дозволяє здійснювати перевірку достовірності даних без необхідності розкривати особисті відомості користувачів, що суттєво підвищує рівень безпеки.

Таким чином, Grass формує нову модель цифрової взаємодії, у межах якої користувачі отримують можливість пасивного заробітку, одночасно сприяючи розвитку технологій штучного інтелекту й зберігаючи контроль над власними цифровими ресурсами.

Платформа Grass являє собою інноваційну децентралізовану систему, яка надає користувачам можливість монетизувати невикористану пропускну здатність власного інтернет-з’єднання. Замість того щоб безоплатно ділитися своїми

ресурсами з великими компаніями, учасники екосистеми можуть отримувати винагороду у вигляді токенів GRASS за участь у функціонуванні мережі.

Ключовий принцип платформи полягає у побудові децентралізованої інфраструктури для збору та обробки публічних веб-даних, які згодом використовуються для навчання моделей штучного інтелекту. Встановивши клієнтський додаток Grass, користувач надає мережі дозвіл використовувати частину свого інтернет-каналу для отримання даних, при цьому зберігаючи повний контроль над персональною інформацією та дотриманням конфіденційності.

Архітектура платформи базується на блокчейні Solana, що забезпечує високу швидкодію, низькі комісії та ефективну масштабованість. Додатково, впровадження технології Zero-Knowledge proofs (ZK) дозволяє здійснювати перевірку достовірності даних без необхідності розкривати особисті відомості користувачів, що суттєво підвищує рівень безпеки.

Таким чином, Grass формує нову модель цифрової взаємодії, у межах якої користувачі отримують можливість пасивного заробітку, одночасно сприяючи розвитку технологій штучного інтелекту й зберігаючи контроль над власними цифровими ресурсами.

Платформа Grass функціонує як децентралізована інфраструктура, основне завдання якої забезпечення ефективного збору та обробки публічних веб-даних з метою підтримки розвитку штучного інтелекту (ШІ). Її ключова ідея полягає у залученні користувачів до надання невикористаної частини інтернет-трафіку, за що вони отримують винагороду у вигляді токенів GRASS. Такий підхід дозволяє створити масштабовану систему, здатну генерувати якісні дата-сети, необхідні для навчання сучасних моделей ШІ.

Платформа побудована на основі блокчейну Solana, використовуючи Layer 2-рішення для досягнення високої продуктивності, масштабованості та мінімальних затримок при обробці транзакцій. Архітектура Grass складається з кількох ключових компонентів:

- Grass Nodes [5]– це пристрої кінцевих користувачів, які надають частину своєї пропускної здатності для виконання функцій децентралізованих веб-скреперів.

Вони збирають публічні дані з відкритих джерел, виступаючи основною точкою збору інформації.

- Роутери – елементи, що відповідають за маршрутизацію трафіку між вузлами та валідаторами. Їхнє завдання – забезпечити контрольований, безпечний та цілісний обіг даних у межах мережі.
- Валідатори – вузли, що здійснюють перевірку достовірності зібраних даних. Для цього використовуються криптографічні механізми, зокрема zk-SNARKs, які дозволяють підтверджувати автентичність інформації без розкриття її вмісту.
- ZK Processor – спеціалізований модуль, що обробляє криптографічні докази, створені валідаторами, і фіксує їх у блокчейні. Завдяки цьому формується незмінний запис про походження та цілісність кожного пакета даних.
- Grass Data Ledger – децентралізований реєстр, у якому зберігаються структуровані масиви даних разом із супровідними доказами достовірності. Це забезпечує прозорість, а також можливість перевірки джерела і цілісності інформації.
- Edge Embedding Models – модулі, що виконують попередню обробку зібраних даних. Вони перетворюють неструктуровану інформацію у структурований формат, придатний для подальшого використання у навчанні моделей штучного інтелекту.

Завдяки такій архітектурі на рисунку 1.1 Grass забезпечує поєднання високої ефективності збору даних, дотримання принципів децентралізації та збереження конфіденційності користувачів, що робить її перспективною платформою в контексті розвитку Web3 та ШІ-сервісів.

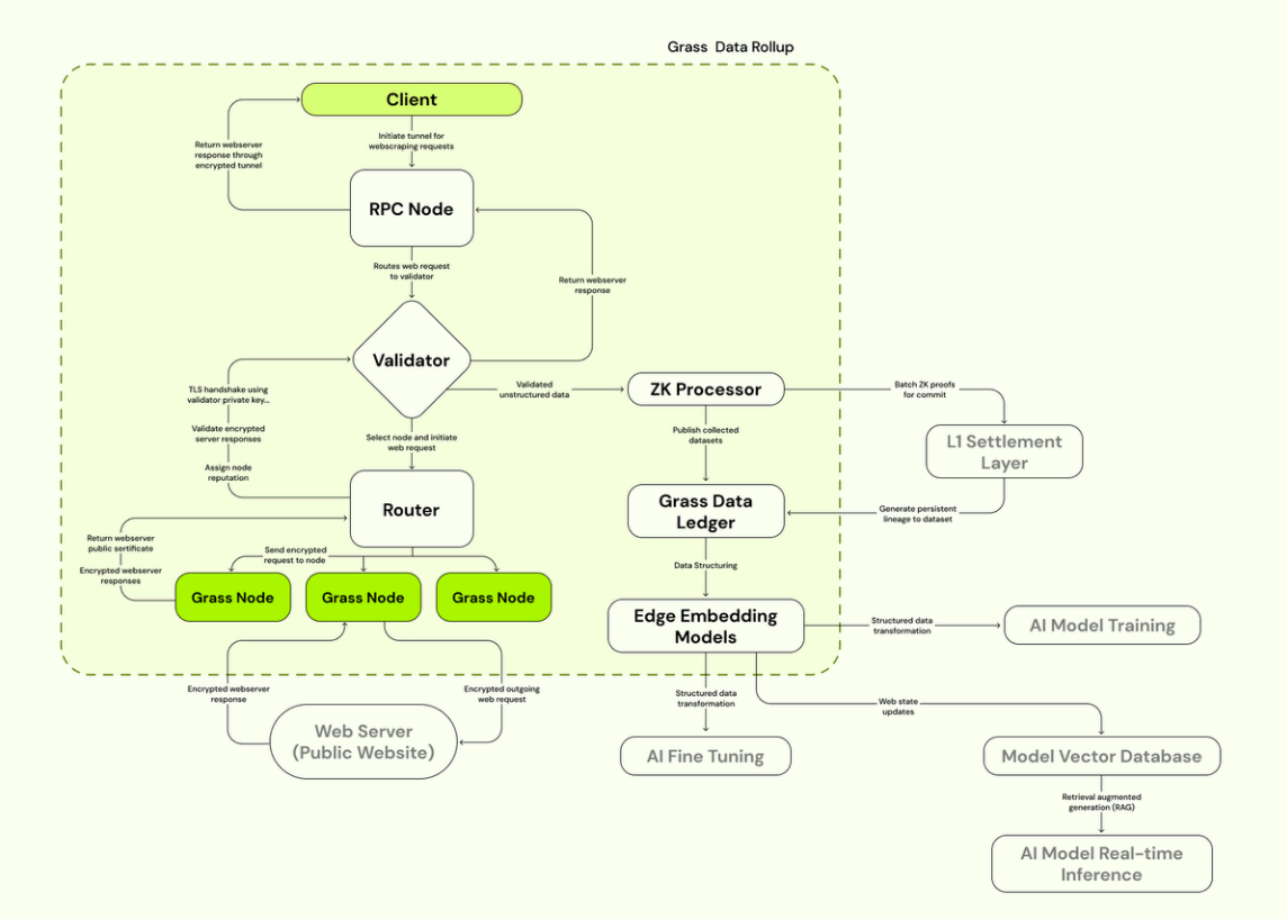

Рисунок 1.1 – Суверенна архітектура зведення даних

Токеноміка платформи Grass є одним із ключових компонентів її децентралізованої інфраструктури. Вона виконує кілька критично важливих функцій: стимулювання активної участі користувачів, забезпечення розвитку мережі та підтримка її довгострокової стійкості й масштабованості.

Загальна емісія токенів GRASS встановлена на рівні 1 000 000 000 одиниць. Розподіл токенів структуровано таким чином, щоб забезпечити баланс між інтересами інвесторів, розробників, користувачів та учасників екосистеми:

- Інвестори – 252 000 000 токенів (25,2%) призначено для ранніх інвесторів. З метою уникнення спекулятивного тиску на ринку ці активи підлягають річному періоду блокування з подальшим поступовим розблокуванням протягом ще одного року. Такий підхід сприяє стабільності та зменшує ймовірність короткострокових коливань цін.

- Учасники та розробники – 220 000 000 токенів (22%) виділено ключовим учасникам мережі, включно з командою розробників. Блокування передбачено на перший рік, після чого розблокування відбувається поступово протягом наступних трьох років. Це створює довгострокову мотивацію для підтримки і розвитку платформи.

- Фонд екосистеми та розвиток – 228 000 000 токенів (22,8%) зарезервовано для стратегічних потреб платформи: фінансування досліджень, технічних оновлень, партнерських програм та DAO-керованих ініціатив. Ці ресурси відіграють важливу роль у підтриманні динаміки зростання екосистеми Grass.

- Майбутні стимули – 170 000 000 токенів (17%) передбачено для ретроактивних винагород. Вони використовуються для заохочення ранніх учасників і творців корисного контенту або інструментів, що зробили значний внесок у розвиток мережі.

- Перший аірдроп – 100 000 000 токенів (10%) було розподілено в рамках масштабної кампанії з метою формування широкої користувацької бази. Цей крок став початком реалізації концепції «Інтернету, що належить користувачам».

- Винагороди для маршрутизаторів – 30 000 000 токенів (3%) виділено для стимулювання учасників, які виконують роль маршрутизаторів, сприяючи оптимізації передачі даних і зниженню затримок у мережі.

Таким чином, токеноміка Grass на рисунку 1.2 побудована з урахуванням принципів сталого розвитку, прозорості й інклюзивності, створюючи умови для гармонійного зростання екосистеми як з технічної, так і з соціальної точки зору.

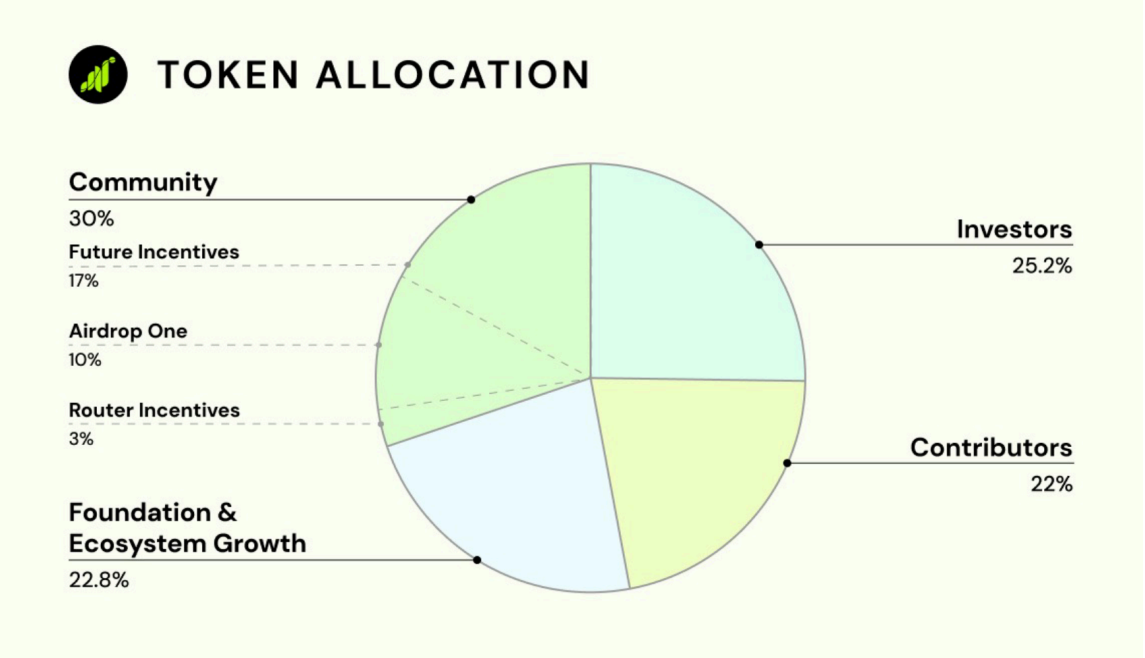

Рисунок 1.2 – Тортова діаграма токенів GRASS

У межах другого сезону функціонування платформи Grass передбачено виділення 5,5% від загальної емісії токенів GRASS на винагороди для активних учасників екосистеми. Це рішення спрямоване на підтримку залучення нових користувачів, а також стимулювання тих, хто вже здійснює внесок у стабільну роботу мережі.

Функціональне призначення токенів GRASS

Токени GRASS відіграють багатоаспектну роль у межах децентралізованої інфраструктури платформи. Основні напрями їх використання включають:

- Стейкінг – користувачі мають можливість блокувати власні токени задля підтримки операційної стабільності мережі. Учасники, що здійснюють стейкінг, отримують додаткові винагороди, що стимулює довгострокову участь у проєкті.
- Участь в управлінні – власники токенів беруть участь у голосуваннях, які впливають на ключові аспекти розвитку платформи. Таким чином реалізується принцип децентралізованого управління (governance), за яким спільнота має реальний вплив на майбутнє Grass.

- Винагороди для спільноти – токени також виступають інструментом стимулювання учасників, які розгортають вузли, забезпечують стабільну передачу даних і загалом підтримують інфраструктуру платформи.

У підсумку, токеноміка GRASS у другому сезоні зберігає свою стратегічну спрямованість на формування життєздатної, стійкої та децентралізованої екосистеми, де кожен активний учасник може отримувати винагороди відповідно до свого вкладу у функціонування мережі.

Платформа Grass відкриває перед користувачами унікальну можливість отримувати винагороди за надання невикористаної пропускної здатності Інтернету. Такий підхід не лише сприяє формуванню децентралізованої цифрової інфраструктури, але й забезпечує низку практичних переваг для учасників екосистеми.

1. Пасивний дохід

Учасники платформи отримують Grass Points, які згодом можуть бути конвертовані у токени $GRASS. Це дозволяє монетизувати власні ресурси без необхідності активної участі, що фактично формує стабільне джерело пасивного доходу.

2. Справедлива система винагород

Модель винагород базується на кількох параметрах: обсязі наданої пропускної здатності, географічному розташуванні та тривалості участі. Такий підхід забезпечує пропорційний і справедливий розподіл винагород, стимулюючи широку та стабільну участь у мережі.

3. Прозорість і безпека

Grass використовує блокчейн-технології для забезпечення прозорості фінансових операцій та збереження конфіденційності користувачів. Особисті дані не зберігаються та не передаються третім сторонам, що гарантує високий рівень безпеки.

4. Можливість впливати на розвиток екосистеми

Власники токенів $GRASS отримують право голосу у питаннях, що стосуються розвитку платформи. Це забезпечує децентралізоване управління та залучення спільноти до прийняття стратегічно важливих рішень.

5. Простота та доступність

Інтерфейс платформи розроблено таким чином, щоб навіть користувачі без технічного досвіду могли легко підключитися до мережі [2] та розпочати отримання винагород. Як показано на рисунку 1.3, після успішного з'єднання вузла користувач має змогу моніторити свої доходи та підключення, що забезпечує прозорість і доступність для широкого кола учасників.

6. Сприяння розвитку штучного інтелекту

Передаючи частину свого інтернет-трафіку [4], користувачі беруть участь у зборі даних для навчання моделей ШІ. Це створює додану цінність і сприяє технологічному прогресу в галузі інновацій.

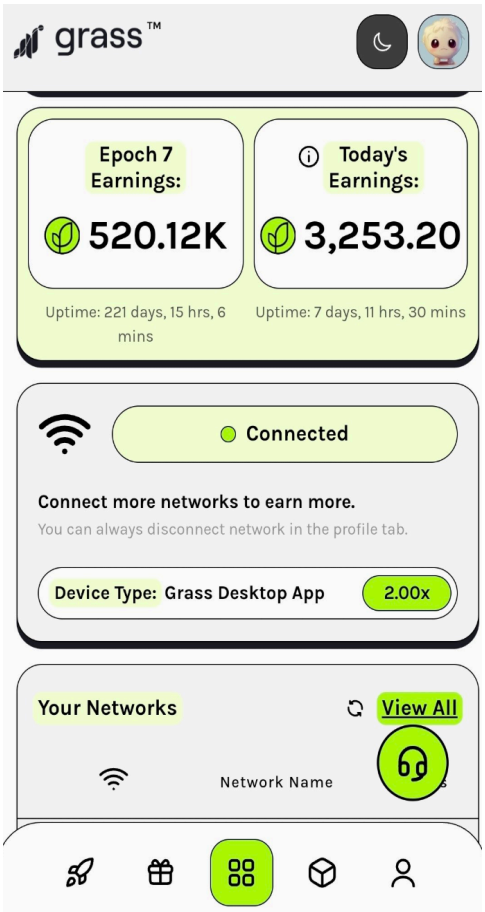

Рисунок 1.3 – Приклад успішного з'єднання вузла у додатку

1.2 Структура акаунта на платформі Grass

Акаунт користувача в екосистемі Grass є основною одиницею участі в мережі [7] та ключем до взаємодії з децентралізованою інфраструктурою платформи. Від правильного налаштування акаунта залежить ефективність фармінгу, участь у сезонних активностях, а також можливість отримання винагород у вигляді токенів $GRASS.

- Електронна пошта – використовується як базовий засіб для реєстрації акаунта. На цю адресу надходять підтвердження, інструкції щодо активації профілю, а також сповіщення про нарахування винагород. Email також може слугувати каналом зв’язку в разі відновлення доступу або технічних питань.
- Пароль – задається користувачем під час реєстрації або автоматично генерується у випадку масового створення акаунтів за допомогою автоматизованих скриптів. Він забезпечує захист облікового запису від несанкціонованого доступу.
- Нікнейм (username) – публічне ім’я користувача в системі, яке може відображатися у дашборді, партнерській програмі або інших публічних компонентах платформи. Хоча не є критично важливим з технічної точки зору, нікнейм відіграє роль у персоналізації досвіду користувача.
- Реферальний код – унікальний ідентифікатор, який надається кожному акаунту для залучення нових користувачів. У межах партнерської програми Grass за кожного успішного реферала нараховуються додаткові поінти, що стимулює органічне зростання мережі.
- Прив’язаний гаманець (Web3 wallet) – один із центральних елементів акаунта, який забезпечує можливість отримання токенів $GRASS. Платформа підтримує інтеграцію з такими гаманцями, як Rabby, MetaMask та іншими сумісними Web3-рішеннями. Прив’язка гаманця є обов’язковою умовою для участі в дистрибуції токенів та управлінні ресурсами в межах екосистеми.

Grass не використовує KYC-процедури для звичайних користувачів, однак безпека акаунтів забезпечується за рахунок:

1. Підтвердження електронної пошти (email verification).

2. Зв’язку акаунта з унікальним гаманцем (wallet binding).

3. Додаткових обмежень для повторного створення акаунтів з одного пристрою/IP.

Також система фіксує IP-адреси, User-Agent, fingerprint браузера та інші технічні характеристики з метою захисту від ботів та масового зловживання. Як показано на рисунку 1.4, платформа Grass реалізує механізм обмеження доступу для уникнення ситуацій, коли один користувач намагається створити велику кількість акаунтів з одного пристрою чи мережі. Система виявляє та блокує підозрілу активність, що забезпечує чесну участь у фармінгу поінтів.

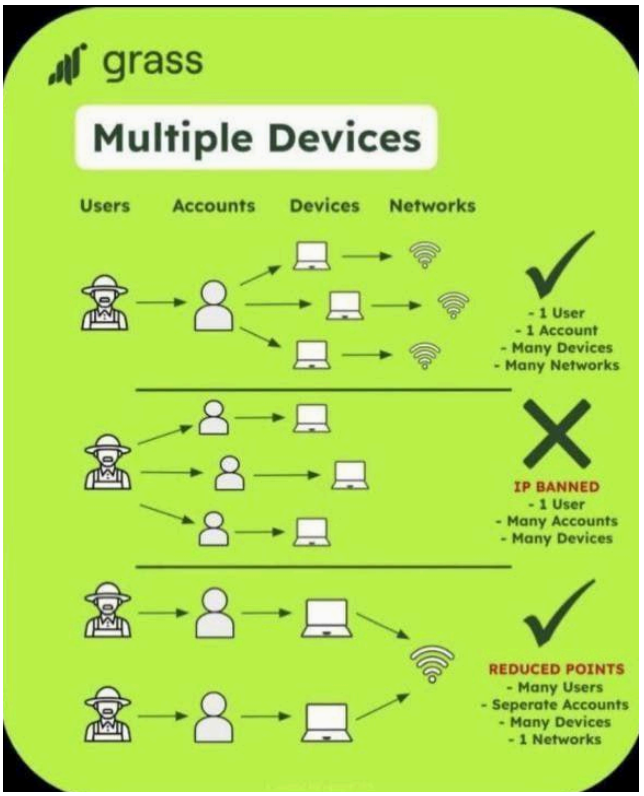

Рисунок 1.4 – Офіційні дозволи підключень

Після створення та активації акаунта користувач отримує доступ до персонального кабінету, який містить низку функціональних розділів:

- Загальна статистика – відображає кількість накопичених Grass Points, поточний прогрес у сезоні, а також статус активності фармінгу.
- Реферальна система – містить інформацію про запрошених користувачів, кількість зароблених бонусів і ефективність реферальної кампанії.

- Wallet binding – інформує про стан прив’язки гаманця, з можливістю його змінити або відв’язати.
- Розділ винагород (Rewards) – надає доступ до історії нарахувань, а також умов, необхідних для конвертації GP у токени GRASS у майбутньому.

Для забезпечення масової реєстрації акаунтів із використанням headless-браузерів або скриптів, необхідно передбачити ряд обов’язкових параметрів:

- email, password – базові облікові дані,
- ref_code – для активації бонусів за реферальною програмою,
- anti-captcha API key – для обходу CAPTCHA,
- proxy – забезпечення унікальності сесії та захист від блокувань.

Ці параметри формують так званий «рядок акаунта», який може оброблятись скриптом у повністю автоматизованому режимі.

Оскільки платформа Grass активно протидіє автоматизованим діям з боку бот-мереж, кожен акаунт повинен відповідати певним умовам унікальності. Серед ключових вимог:

- унікальна електронна пошта;
- окрема браузерна сесія або профіль;
- проксі з різними геолокаціями;
- реалістичний User-Agent.

Недотримання цих вимог може призвести до нульового нарахування поінтів або блокування акаунтів.

1.3 Механізм фармінгу поінтів та роль гаманців

Grass Points (GP) є основною одиницею винагород у рамках платформи Grass. Вони нараховуються користувачам за надання невикористаної пропускної здатності

через браузерне розширення або десктопний додаток. Кількість зароблених GP залежить від тривалості активного з'єднання, стабільності Інтернету та географічного розташування. Всі поінти накопичуються в реальному часі та формують основу для майбутньої конвертації в токени GRASS, зокрема під час аірдропів.

Для отримання GRASS токенів у рамках аірдропів необхідно прив’язати криптовалютний гаманець до акаунта Grass. Платформа підтримує гаманці, сумісні з мережею Solana (наприклад, Phantom або Solflare). Прив’язка гаманця дає змогу точно ідентифікувати користувача та гарантувати безпечну передачу токенів. Важливою умовою є наявність прив’язаного гаманця на момент знімка (snapshot), інакше користувач не отримає винагород.

Grass реалізує багаторівневу реферальну програму, що дозволяє заробляти додаткові GP через залучення нових користувачів:

1. Прямі реферали – 20% від GP, зароблених запрошеним користувачем.
2. Непрямі реферали – 10% з другого рівня та 5% з третього рівня.
3. Бонуси – за досягнення рефералом 100 годин активного з'єднання користувач отримує 2 500 GP, а сам реферал – 5 000 GP.

Така система стимулює активне залучення нових учасників і побудову мережі.

Для максимізації ефективності фармінгу рекомендується:

- використання VPS – дозволяє забезпечити безперервну роботу додатку;

мультиакаунтинг – запуск декількох облікових записів з різними IP-адресами збільшує загальний дохід;

- автоматизація – застосування скриптів для моніторингу та управління значно спрощує процес.

Попри ефективність таких методів, варто зазначити, що їх використання суперечить політиці платформи, отже – офіційно не підтримується.

Grass поєднує у собі блокчейн, Web3 та ШІ-технології, надаючи користувачам можливість заробітку шляхом надання мережевих ресурсів. Високий рівень інтересу до платформи зумовлює потребу в рішенні, здатному забезпечити масовий запуск та обслуговування акаунтів.

Grass, як частина сегменту DePIN (Decentralized Physical Infrastructure Networks), активно використовує механізм нарахування поінтів, що згодом конвертуються у токени. З урахуванням успішного першого аірдропу та анонсованого другого сезону, інтерес користувачів до масової участі суттєво зріс. Це призводить до збільшення кількості акаунтів, що потребують ефективного управління.

Ручне створення та обслуговування акаунтів є трудомістким, схильним до помилок і неефективним у масштабі. Проблеми із втратою доступу, дублюванням облікових записів або неузгодженістю дій знижують загальний результат. Тому виникає об’єктивна потреба в автоматизованих рішеннях, які централізовано керують процесами реєстрації, верифікації, прив’язки гаманців і запуску фармінгу.

Застосування багатопотокових скриптів, які автоматично проходять всі етапи взаємодії з платформою, дозволяє:

- масово створювати акаунти,
- запускати фармінг одночасно для сотень користувачів,
- мінімізувати людське втручання,
- ефективно розподіляти ресурси (VPS, проксі, антикапча API).

Сучасна автоматизація враховує політики безпеки платформи. Серед обов’язкових заходів – ротація проксі, зміна User-Agent, використання окремих сесій та email-адрес, а також інтеграція із сервісами розв’язання CAPTCHA. Це знижує ризики блокування, виключення з аірдропу та зменшує ймовірність виявлення масових дій.

Окрім особистого використання, Grass дедалі частіше розглядається як платформа для організації прибуткових ферм акаунтів. Масова автоматизація дозволяє запускати сотні акаунтів на одному сервері [3], що суттєво підвищує дохідність і дозволяє швидко окупити інвестиції. Цей підхід використовують як індивідуальні ентузіасти, так і цілі команди з повністю автоматизованими скриптами.

1.4. Аналіз ризиків автоматизованих систем взаємодії з веб-платформами

Автоматизація процесів взаємодії з веб-сервісами, зокрема в контексті масової реєстрації акаунтів, збору поінтів або виконання інших повторюваних дій, супроводжується низкою технічних і організаційних ризиків. Ці ризики можуть вплинути як на стабільність роботи системи, так і на саму можливість її використання у довготривалій перспективі.

Найпоширенішим ризиком є виявлення автоматизованої поведінки з боку платформи. Веб-сайти, зокрема платформи типу Grass, можуть використовувати антибот-захисти, такі як аналіз частоти запитів, використання технологій Cloudflare, перевірка User-Agent, геолокаційне порівняння IP, аналіз вікна активності, повторення однакових дій у межах одного сеансу. У разі виявлення підозрілої активності платформа може заблокувати акаунт, обмежити IP або навіть відкинути усі поінти.

Ще одним фактором ризику є обмеження або бан IP-адрес проксі-серверів, особливо якщо використовується багато акаунтів з одного діапазону. Це може призвести до масового відключення сесій або до неможливості проходження капчі.

Також існує ризик, пов’язаний із змінами у внутрішній логіці сайту, включаючи зміну API, структури HTML, або параметрів WebSocket-з’єднання [8]. Подібні зміни можуть призвести до порушення стабільної роботи системи або потребувати оперативного оновлення скриптів.

Крім того, варто враховувати ризики, пов’язані з навантаженням на поштові сервіси – якщо велика кількість акаунтів використовує поштові адреси з одного сервера (наприклад, GMX, ProtonMail), платформа може заблокувати доставку листів або відмітити їх як спам.

Для зменшення зазначених ризиків у системі було реалізовано кілька технічних захистів: використання ротації User-Agent, затримки між діями, окремий проксі для кожного акаунта, обмеження кількості одночасних потоків, симуляція реального браузерного WebSocket-з’єднання. Завдяки цьому вдалось мінімізувати

ймовірність виявлення автоматизованої активності та забезпечити стабільність роботи.

1.5 Висновки до розділу 1

У першому розділі було проведено детальний аналіз платформи Grass як об’єкта дослідження. Розглянуто її архітектуру, принципи роботи, токеноміку, переваги участі для користувачів, а також структуру облікового запису й механізми нарахування винагород. Особливу увагу приділено актуальності масової автоматизації акаунтів, що дозволяє ефективно масштабувати участь у фармінгу поінтів і забезпечити максимальну вигоду в умовах конкурентного середовища. Отримані висновки створюють підґрунтя для технічної реалізації автоматизованої системи, що буде розглянута у наступному розділі.

2 ПОСТАНОВКА ЗАДАЧІ ТА ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Алгоритм автоматизованої реєстрації акаунта

Програмна логіка автоматизованої реєстрації на платформі Grass реалізується як послідовність технічно чітко визначених етапів. Кожен з них виконується за допомогою headless-браузера у поєднанні з системою генерації та обробки даних:

- Генерація облікових даних: на початковому етапі створюються унікальні значення для ключових полів – електронної пошти, імені користувача та пароля. Ці значення зберігаються у базі даних або лог-файлі для подальшого використання й моніторингу.
- Ініціалізація headless-сесії: запускається браузерний інстанс з попередньо заданим проксі-сервером і зміненим User-Agent, що дозволяє імітувати звичайного користувача. Додатково застосовуються параметри, які відповідають типовим поведінковим моделям реальних користувачів.
- Заповнення форми: автоматизований скрипт переходить за адресою https://app.getgrass.io/register, де вводить згенеровані дані у відповідні поля реєстраційної форми.
- Розв’язання CAPTCHA: для обходу Google reCAPTCHA v2 здійснюється інтеграція з антикапча-сервісами (наприклад, 2Captcha або CapSolver). Через API передається ключ сайту, після чого отримується токен із розв’язанням, який вставляється у відповідне поле форми.
- Підтвердження відправки: після введення всіх даних і розв’язання CAPTCHA, скрипт імітує натискання кнопки реєстрації та очікує відповіді сервера [3] про успішність або невдачу створення облікового запису.

Обробка результатів: результат (успішна або неуспішна реєстрація) записується у лог-файл, з можливістю подальшого аналізу, повторної спроби або виключення пошкоджених облікових записів.

Щоб уникнути блокування з боку систем виявлення підозрілої активності, кожна реєстраційна сесія використовує окремий проксі-сервер. Це дозволяє розподіляти IP-адреси відповідно до заданої географії, що імітує доступ до платформи з різних країн. Окрім цього, headless-браузери здійснюють підміну технічних параметрів, зокрема:

- мови браузера (Accept-Language);
- часової зони (Timezone Spoofing);
- браузерного "відбитка" (Fingerprint Spoofing);
- доступних розширень і шрифтів;
- роздільної здатності екрана.

Завдяки цьому система Grass розцінює активність як дії реального користувача, що дозволяє мінімізувати ймовірність виявлення автоматизації.

Платформа Grass застосовує Google reCAPTCHA v2 як основний елемент захисту від ботів. Для його подолання автоматизована система інтегрується з зовнішніми сервісами антикапчі. Принцип дії такий:

1. Скрипт отримує sitekey зі сторінки реєстрації.
2. Через API звертається до сервісу розв’язання CAPTCHA (наприклад, 2Captcha).
3. Після завершення розпізнавання сервіс повертає токен, який вводиться у форму.
4. Форма надсилається з активованим токеном, що дозволяє пройти перевірку.

Завдяки цій процедурі автоматизована система може проходити реєстрацію без ручного втручання, дотримуючись при цьому вимог антибот-захисту та знижуючи ризик блокування. Як видно на рисунку 2.1, система витримує навантаження при роботі з 20 000 активними акаунтами, забезпечуючи ефективність та стабільність під час масової реєстрації.

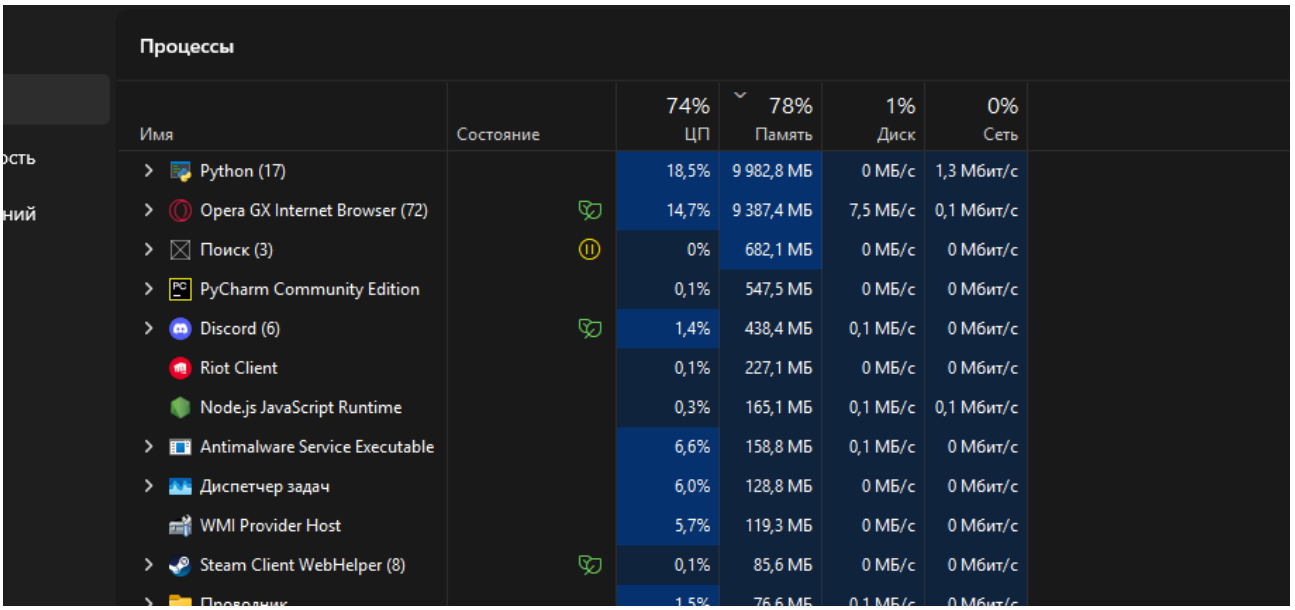

Рисунок 2.1 – Навантаження на систему при 20к активних акаунтів

Процес автоматизованої реєстрації облікового запису на платформі Grass реалізується як послідовність чітко структурованих кроків, які дозволяють досягти високої ефективності при масовому створенні акаунтів.

- Генерація облікових даних. На першому етапі відбувається створення унікальних значень для електронної пошти, імені користувача та пароля. Згенеровані дані записуються у внутрішню базу або лог-файл для подальшого використання в процесах верифікації та авторизації.
- Запуск headless-сесії. Ініціалізується безголовий браузер із заданими параметрами, включно з унікальним User-Agent та проксі-сервером. Це забезпечує імітацію дій реального користувача з урахуванням географічної розподіленості та варіативності пристроїв.
- Заповнення форми. Система автоматично переходить на реєстраційну сторінку за адресою https://app.getgrass.io/register, де вводить попередньо згенеровані дані у відповідні поля.
- Розв’язання CAPTCHA. Для обходу Google reCAPTCHA v2 реалізується інтеграція з антикапча-сервісами, такими як 2Captcha або CapSolver, за допомогою API-ключів. Скрипт передає ключ сайту, очікує на результат і вставляє отриманий токен у форму.

- Підтвердження відправки. Після заповнення форми та активації CAPTCHA скрипт виконує імітацію натискання кнопки «Sign up» та очікує відповідь сервера.
- Обробка результатів. Залежно від відповіді платформи, система фіксує статус реєстрації (успішно або з помилкою) у лог-файл, що дозволяє вести облік усіх створених акаунтів.

Щоб уникнути блокувань з боку систем захисту, кожна реєстрація здійснюється із застосуванням унікального проксі-сервера. Це дозволяє моделювати доступ із різних IP-адрес та географічних регіонів. Окрім проксі, headless-браузери застосовують техніки імітації користувацької поведінки, зокрема:

- підміну timezone, мовних параметрів і роздільної здатності;
- зміни у WebGL-рендерінгу та Canvas fingerprint;
- підключення фіктивних розширень і шрифтів;
- модифікацію navigator-об’єктів для обману ботозахисту.

Завдяки цьому вдається знизити ризик виявлення автоматизованої активності.

З метою запобігання масовим реєстраціям ботів, платформа Grass використовує Google reCAPTCHA v2. Для обходу цього механізму автоматизована система інтегрується з перевіреними антикапча-сервісами, які виконують обчислення на стороні зовнішнього сервера. Алгоритм дії передбачає:

1. Отримання sitekey з реєстраційної сторінки.
2. Надсилання запиту до антикапча-провайдера із зазначеним sitekey та URL
3. Очікування на рішення (токен підтвердження)
4. Вставку отриманого токена у форму на сайті.

Цей процес дозволяє обійти антибот-захист без втрати швидкодії, що є критично важливим для реалізації масової автоматизації. На рисунку 2.2 представлено інтерфейс сервісу 2Captcha, який використовується для обходу механізму Google reCAPTCHA. Цей сервіс допомагає автоматизувати процес підтвердження через captcha, що значно спрощує реєстрацію та взаємодію з платформою.

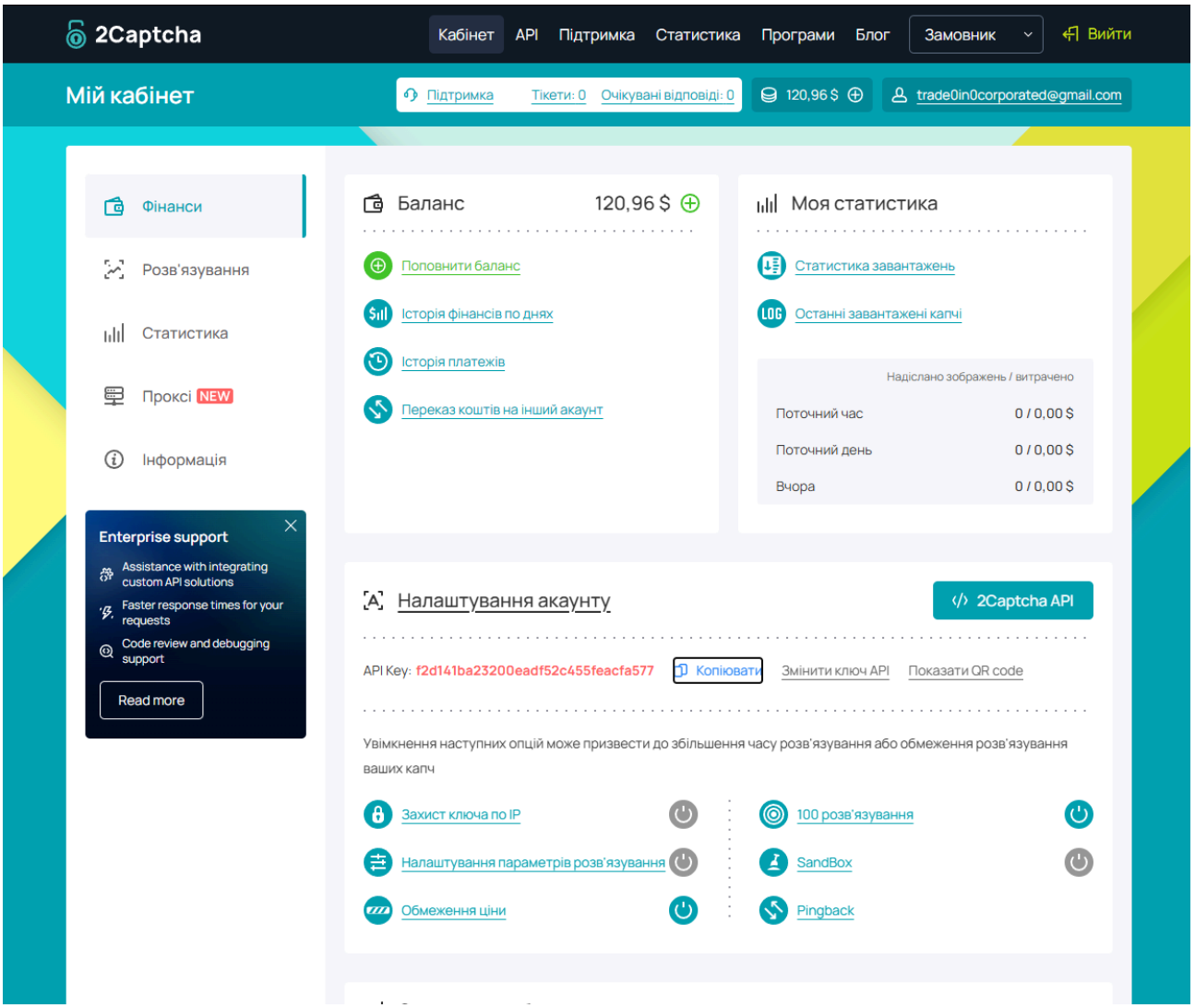

Рисунок 2.2 – Скріншот сервісу 2Captch

Однією з ключових переваг автоматизації реєстрації через headless-браузери є можливість паралельного виконання сотень реєстраційних потоків. Для цього використовуються асинхронні засоби програмування – зокрема, бібліотека asyncio у поєднанні з Semaphore, яка дозволяє контролювати кількість одночасно активних сесій та рівномірно розподіляти навантаження між ними.

Кожен потік функціонує як незалежна одиниця, що проходить повний цикл реєстрації: від генерації даних до взаємодії з сайтом та обробки відповіді. У разі виникнення помилок – таких як блокування проксі, збій антикапча-сервісу або відмова в реєстрації – відповідний потік автоматично перезапускається з новим набором параметрів. Такий підхід забезпечує стійкість і надійність навіть за умов високого навантаження або часткових збоїв у зовнішніх сервісах.

Для ефективного моніторингу й аналізу роботи автоматизованої системи впроваджено механізм детального логування результатів кожної сесії. Після завершення процесу реєстрації скрипт фіксує результат у відповідному файлі:

- success.txt – містить список успішно створених акаунтів, включаючи ID, email та інші параметри;
- failed.txt – містить інформацію про неуспішні спроби реєстрації, з можливістю подальшого аналізу причин помилок.

Такий підхід дозволяє:

1. Відслідковувати ефективність реєстраційних потоків у реальному часі;

2. Виконувати повторну реєстрацію невдалих акаунтів без втрати даних;

3.Забезпечити масштабовану підтримку всієї системи при зростанні кількості облікових записів.

Механізми мультипоточності та логування виступають важливими елементами автоматизованої архітектури, що дозволяє системі працювати стабільно навіть під високими навантаженнями. Як видно на рисунку 2.3, логи у файлах success.txt та failed.txt надають детальну інформацію про результати реєстрацій, що дозволяє здійснювати аналіз причин помилок та повторне використання акаунтів для реєстрації.

На рисунку 2.4 зображено загальний лог після завершення виконання програми, де відображається зібрана інформація за всю сесію, що дозволяє оцінити загальний стан системи і підготувати її до подальших операцій.

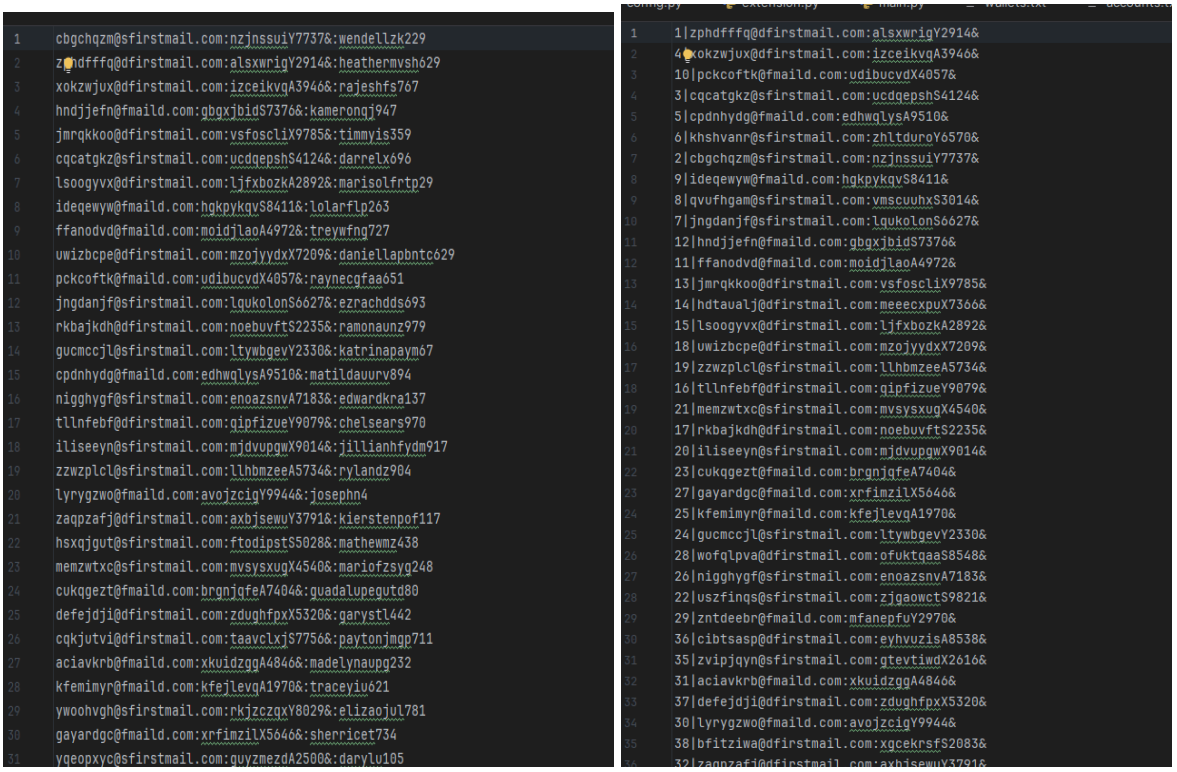

Рисунок 2.3 – Логування у файлах success.txt і failed.txt

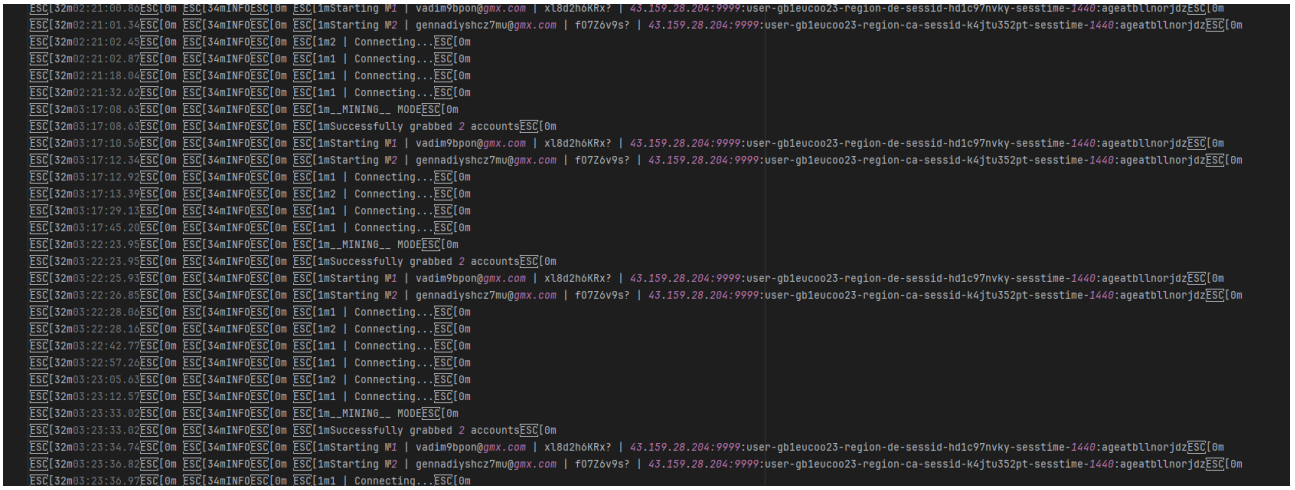

Рисунок 2.4 – Загальний лог після закінчення виконання програми

2.2 Підключення гаманця та підтвердження пошти

Для повноцінної активації облікового запису на платформі Grass необхідно виконати два критично важливі етапи: підтвердження електронної пошти та прив’язку Web3-гаманця. Обидва компоненти є обов’язковими для участі в програмі

фармінгу Grass Points та отримання токенів $GRASS у межах аірдропів. Наявність лише зареєстрованого акаунта без цих підтверджень не дає змоги користувачеві брати участь в економіці платформи.

Підтвердження email-адреси виступає елементом первинної верифікації користувача та є технічно необхідним кроком для активації акаунта рисунок 2.5. Починаючи з 12 березня 2025 року, платформа Grass оновила політику доступу: без підтвердженого email рисунок 2.6 обліковий запис не має змоги заробляти поінти або брати участь у будь-яких винагородах, включно з аірдропами.

Хоча у серверній частині системи [6] частково збереглася можливість створення акаунтів через прямі HTTP-запити до REST-ендпойнтів (наприклад, POST /api/register із параметрами email, password, referral_code), акаунти, створені в обхід основного інтерфейсу, часто отримують внутрішній статус limited. Такий статус суттєво обмежує функціональність: платформа може відмовити у нарахуванні винагород, блокувати активність або повністю виключити акаунт із сезонної програми.

З огляду на це, рекомендованим і безпечним підходом є реєстрація з обов’язковим підтвердженням електронної пошти, яка реалізується за допомогою IMAP-протоколу. Це дозволяє повністю автоматизувати процес без потреби у ручному втручанні.

Технічна схема підтвердження через IMAP:

1. Після відправки форми реєстрації система очікує вхідний лист від сервера Grass на вказану електронну адресу.
2. Встановлюється захищене IMAP-з’єднання з поштовим сервером користувача.
3. У папці "Вхідні" здійснюється пошук листа за ключовими ознаками – зокрема, відправником та темою.
4. Із вмісту HTML-листа парситься гіперпосилання активації акаунта.
5. Посилання викликається програмно через HTTP-запит, після чого обліковий запис отримує статус active.

Перевагою цього підходу є повна автоматизація – усі дії виконуються у фоновому режимі. У разі збою (наприклад, затримка листа, недоступність поштового сервера, помилки під час парсингу), система здійснює кілька повторних спроб із затримками та веде журнал подій для подальшого аналізу.

Таким чином, хоча технічно ще можливе створення акаунтів без підтвердження пошти, використання офіційного механізму активації через IMAP є найнадійнішим варіантом, що гарантує повний доступ до всіх функцій акаунта та значно знижує ризики блокування або втрати винагород.

Рисунок 2.5 – Реєстрація до 12.03.2025

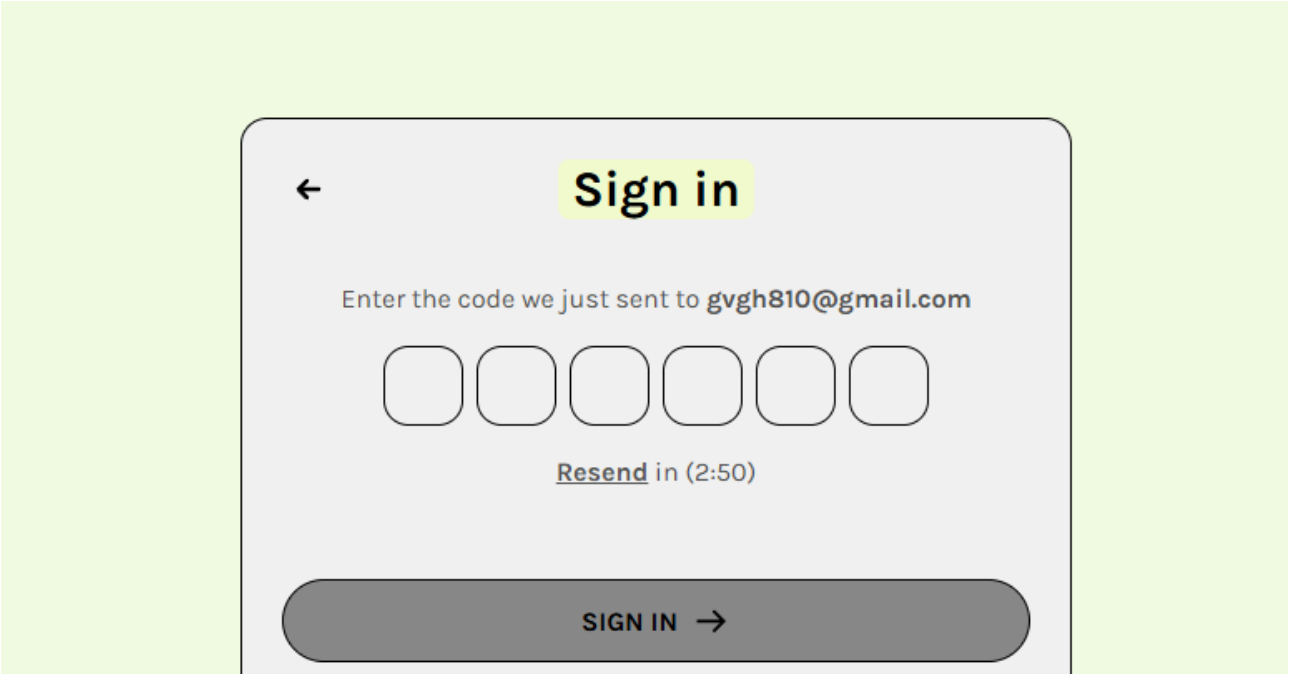

Рисунок 2.6 – Реєстрація після 12.03.2025

Після активації облікового запису за допомогою підтвердження електронної пошти наступним обов’язковим кроком є прив’язка криптовалютного Web3-гаманця. Цей етап є критично важливим для повноцінної участі в економіці платформи Grass, оскільки саме на прив’язаний гаманець здійснюється розподіл токенів $GRASS за підсумками сезону.

Оскільки Grass функціонує в екосистемі блокчейну Solana, платформа підтримує гаманці, сумісні з цією мережею. Найпоширенішими серед них є:

- Phantom Wallet – браузерне розширення, яке є де-факто стандартом для взаємодії з dApp-середовищем Solana;
- Rabby Wallet – універсальний гаманець з підтримкою імпорту приватних ключів;
- Solflare, Trust Wallet та інші гаманці, що мають інтеграцію з Web3 та підтримку Solana.

Процедура підключення Web3-гаманця реалізується за наступним сценарієм:

1. Користувач ініціює процес через інтерфейс персонального кабінету, натиснувши кнопку "Connect Wallet".

2. Відкривається модальне вікно, яке активує Web3-з'єднання за допомогою Wallet Adapter.

3. Гаманець користувача відображає запит на підтвердження підключення.

4. Після підтвердження система зчитує публічну адресу (publicKey) та закріплює її за відповідним обліковим записом.

5. У профілі з’являється статус "Wallet connected", що свідчить про успішне підключення.

Варто зауважити, що у разі повторного використання гаманця, який уже був прив’язаний до іншого акаунта, система або відмовляє у прив’язці, або надає акаунту обмежений функціонал. У рамках автоматизованих систем підключення зазвичай використовуються попередньо згенеровані seed-фрази або приватні ключі, які імпортуються локально на рівні браузерного середовища перед процедурою підключення.

У новій архітектурі платформи Grass було запроваджено додатковий рівень верифікації Web3-гаманця – через електронну пошту. Це зроблено для посилення безпеки акаунтів і запобігання зловживанням мультиакаунтингом або автоматизованими маніпуляціями.

Після підключення гаманця система автоматично надсилає на електронну пошту користувача листа з підтверджувальним посиланням. Для завершення процесу прив’язки користувачеві необхідно відкрити це посилання. Без підтвердження акаунт вважається неповністю активованим і не допускається до нарахування винагород.

Алгоритм підтвердження відповідає стандартному IMAP-процесу верифікації:

1. Встановлюється з’єднання з поштовим сервером через IMAP-протокол.

2. Здійснюється пошук листа з темою на кшталт "Confirm your wallet".

3. Із HTML-вмісту парситься відповідне гіперпосилання.

4. Посилання відкривається – вручну або програмно через HTTP-запит.

5. Акаунт отримує статус "Wallet verified", що відображається у системі.

Цей механізм є важливим елементом ідентифікації, оскільки дозволяє зв’язати акаунт, електронну адресу та гаманець в єдину цілісну сутність, що належить

одному користувачу. Лише після проходження цього етапу акаунт вважається повністю активованим (active/verified) і допускається до участі в сезонному фармінгу та розподілі токенів, як показано на рисунку 2.7.

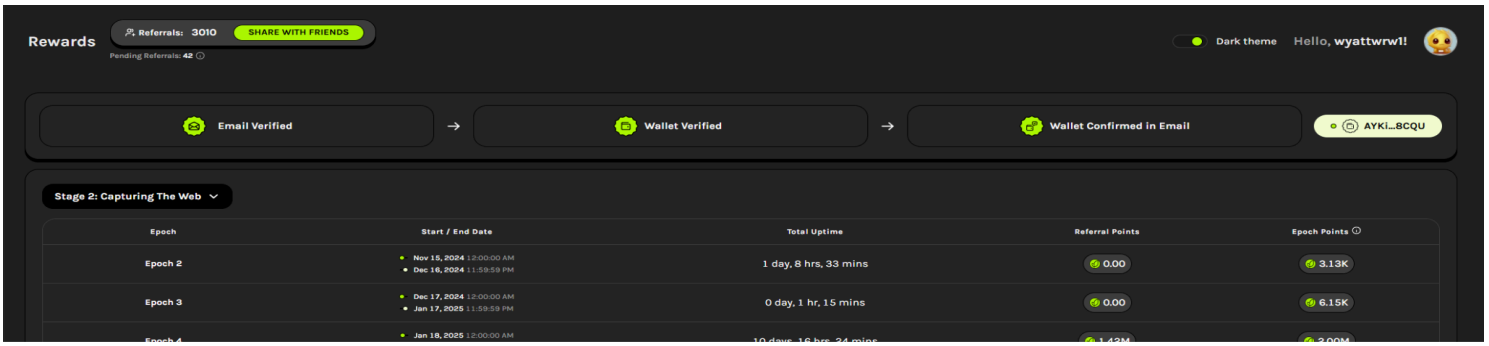

Рисунок 2.7 – Повністю підєднанний аккаунт

2.3 Збір і фармінг поінтів у багатопоточному режимі

Фармінг Grass Points на платформі Grass реалізується через постійне утримання активного WebSocket-з’єднання з сервером. Ці поінти нараховуються за стабільну присутність акаунта в мережі, а також за участь у зборі відкритих веб-даних. Для масштабованого керування сотнями акаунтів одночасно застосовується багатопоточна архітектура, що використовує асинхронну взаємодію, перевірку стану сесій та механізми відновлення з’єднань у разі збою.

На відміну від стандартних WebSocket-реалізацій, платформа Grass не надає відкритої або очевидної адреси для підключення через клієнтський інтерфейс сайту. При спробі аналізу мережевих запитів через вкладку Network > WS у браузерах Chrome або Firefox не фіксується жодне активне з’єднання після логіну. Це свідчить про наміри платформи приховати або зашифрувати процес ініціалізації WebSocket, з метою ускладнення автоматизованого доступу.

Grass використовує обфускацію JavaScript-коду [9] та проксійні скрипти, що робить стандартні засоби моніторингу неефективними. Усі ключові запити або

зашифровані, або проходять через непрозорі внутрішні API, не залишаючи слідів відкритого WebSocket-трафіку у стандартних інструментах розробника.

Для виявлення фактичної точки підключення до WebSocket-сервера необхідно використовувати мережеві аналізатори з можливістю перехоплення HTTPS-з’єднань. Серед ефективних інструментів, що застосовуються у таких випадках:

- Burp Suite (у режимі HTTPS MITM);
- mitmproxy;
- Fiddler;
- Wireshark;
- Proxyman.

Ці інструменти дозволяють:

- створити довірений сертифікат CA та підключити його до системи/браузера для перехоплення зашифрованого трафіку;
- зчитувати повний вміст HTTPS-запитів, включаючи заголовки, токени автентифікації, параметри браузера;
- ідентифікувати момент ініціалізації WebSocket-з’єднання, орієнтуючись на заголовок Upgrade: websocket;
- зафіксувати фактичну адресу підключення (wss://...) разом із ключовими параметрами – такими як session_token, browser_id, user_id, fingerprint тощо.

У рамках технічного експерименту Grass було запущено у браузері, після чого весь трафік було спрямовано через Burp Suite, попередньо налаштований на перехоплення HTTPS. Після входу в акаунт і проходження автентифікації, в одному з фонових запитів було зафіксовано ініціацію WebSocket-з’єднання з характерним заголовком Upgrade: websocket. Запит містив набір параметрів, зокрема:

- адресу підключення (wss://farm.getgrass.io/connect);
- маркери сесії (token=, uid=, ts=);
- мета-дані браузера, що ідентифікують пристрій.

Результати такого аналізу дозволяють відтворити структуру WebSocket-з'єднання [11] програми, що є фундаментом для подальшого створення асинхронного клієнта фармінгу. Як показано на рисунку 2.8, дані, отримані через

Burp Suite, демонструють запит, що використовується для встановлення WebSocket-з'єднання з сервером платформи Grass.

```
GET /socket.io/?EIO=4&transport=websocket&browser_id=xxxxxxxxx
Host: api.getgrass.io
Upgrade: websocket
Connection: Upgrade
Origin: https://app.getgrass.io
User-Agent: ...
```

Рисунок 2.8 – Дані отримані через Burp Suite

Після успішного встановлення WebSocket-з’єднання та проходження авторизації, клієнт переходить у режим активного фармінгу, в рамках якого відбувається постійна взаємодія з сервером через так званий heartbeat-механізм. Його суть полягає в обміні контрольними повідомленнями:

- клієнт періодично надсилає сигнал ping, що засвідчує активність сесії;
- сервер у відповідь повертає pong або сам ініціює ping;
- у разі відсутності відповіді протягом встановленого тайм-ауту (зазвичай 60–120 секунд), з’єднання розривається.

Окрім heartbeat, клієнт може надсилати запити, які імітують активність акаунта: mine, activity, presence, а також періодично оновлювати баланс або перевіряти статус фармінгу.

У разі розриву з’єднання реалізується автоматичне перепідключення з оновленням ключових параметрів – зокрема browser_id, проксі-сервером, User-Agent тощо.

Для успішної ініціалізації WebSocket-сесії клієнт повинен передати набір параметрів, які зазвичай отримуються у відповідь на REST-запити (наприклад, GET /api/init, GET /api/user):

- browser_id – ідентифікатор сесії;
- user_id, session_token або JWT – для авторизації;
- region, version, fingerprint – додаткові службові параметри.

Ці дані передаються у заголовках запиту або у формі query-параметрів, і повинні бути достовірно згенеровані або отримані з API-платформи. Важливу роль відіграють:

- Origin – має строго відповідати https://app.getgrass.io;
- User-Agent – типовий сучасний браузер (наприклад, Chrome 120+);
- Proxy IP – бажано географічно релевантний (Tier-1 регіони).

Недотримання хоча б одного з цих параметрів зазвичай призводить до відмови в з'єднанні або миттєвого його розриву.

Щоб уникнути блокувань і зберегти стабільність з'єднання, кожен акаунт обслуговується через унікальний проксі-сервер (HTTP або SOCKS5). Це дозволяє:

- уникати бану за повторне використання IP-адрес;
- дотримуватись географічної диверсифікації;
- тестувати швидкість і якість з'єднання (наприклад, через /proxy-score).

При виявленні низької продуктивності або блокування, система автоматично виконує динамічну заміну проксі з оновленням параметрів сесії.

Для одночасної обробки десятків або сотень акаунтів використовується асинхронна модель з підтримкою паралельних потоків. Найефективніші реалізації базуються на:

- бібліотеці asyncio (у Python);
- ThreadPoolExecutor або Task-групах;
- обмеженні кількості потоків через Semaphore.

Кожен акаунт виконується у власному таску, із незалежним життєвим циклом: підключення → фармінг → перевірка балансу → очікування → перепідключення (у разі розриву).

Для ефективного управління масштабованою системою реалізується збирання та збереження статистики в структурованих таблицях:

- Таблиця акаунтів – з email, user_id, поточним статусом, прив'язаним проксі, останньою активністю;
- Таблиця поінтів – з актуальним балансом GP, часом останнього оновлення;

- Таблиця проксі – з IP-адресою, оцінкою швидкості, географією.

Ці дані дозволяють виявляти неактивні акаунти, проблемні проксі та оптимізувати розподіл навантаження при масштабуванні

2.4 Конкурентні рішення

На ринку Grass-автоматизації існують кілька типових альтернатив, що пропонують різний рівень функціональності та контролю.

Слотові сервіси пропонують оренду вже активованих акаунтів із підключеним гаманцем. Користувач сплачує за "слот" доступу та отримує віконце до статистики – зазвичай через Telegram-бот або просту панель.

Переваги:

- не потребує технічних знань;
- повністю готове до фармінгу середовище.

Недоліки:

- висока вартість ($1–$2/акаунт/міс);
- відсутність доступу до внутрішніх даних;
- ризики втрати доступу чи витоку даних (база знаходиться на стороні сервісу).

Такий підхід підходить для початкового тестування, однак не є оптимальним для досліджень або побудови власної інфраструктури.

Grass-Bot рисунок 2.9 – це open-source-проєкт на базі Node.js [5], що частково реалізує функціонал фармінгу через WebSocket. Він дозволяє запускати кілька акаунтів паралельно та працює зі справжніми сесіями авторизації. [12]

Перевагами проєкту є:

- відкритий код;
- доступність для модифікації;
- базовий багатопоточний режим.

Однак, у поточному стані він має обмежену документацію, не покриває весь цикл від реєстрації до верифікації гаманця та не передбачає складну логіку моніторингу або заміни проксі.

Рисунок 2.9 – Інтерфейс Grass-Bot від Jaammerr

Переваги:

- відкритий код, що дозволяє вносити зміни під свої потреби;
- підтримка мультиоблікової роботи;

- активна спільнота на GitHub.

Недоліки:

- відсутній зручний інтерфейс – програма працює лише з командного рядка;
- нестабільне WebSocket-з'єднання, що використовує стару структуру з частими розривами;
- підтримуються лише вузли типу 1x та 1.25x, що обмежує масштабування та швидкодію;
- немає вбудованої реєстрації акаунтів, антикапчі та поштової перевірки.

Загальна оцінка: підходить для технічно підкованих користувачів, але вимагає доопрацювання та ручного контролю.

Самостійне автоматизоване скриптування (на базі Selenium / Playwright)

Деякі користувачі обирають створення власних скриптів для керування браузером із використанням headless-інструментів, таких як Playwright або Selenium. Цей підхід дозволяє вручну автоматизувати кожен етап: заповнення форм, обхід CAPTCHA, прив'язку гаманця, фармінг поінтів.

Переваги:

- повний контроль над логікою;
- можливість реалізувати будь-який сценарій.

Недоліки:

- потребує значного технічного досвіду;
- низька стабільність при великій кількості потоків;
- відсутність універсального рішення, усе тримається на самописному коді;
- складна підтримка при оновленнях DOM-структури сайту.

Загальна оцінка: найгнучкіший, але найскладніший у реалізації підхід. Підходить для експериментів, але не для серійного використання.

2.5 Обґрунтування вибору власного рішення

У процесі створення інформаційної системи для масової автоматизації реєстрації, управління акаунтами та фармінгу Grass Points було прийнято рішення застосувати гібридний технологічний стек, поєднавши Python [1] (для роботи з реєстрацією та підготовкою акаунтів) та Node.js (для реалізації високонавантаженого модуля фармінгу). Така архітектура дозволила досягти оптимального співвідношення продуктивності, гнучкості й масштабованості системи.

Функціонал реєстрації облікових записів, обробки поштових підтверджень через IMAP, прив'язки рефералів, підключення Web3-гаманців, імпорту проксі та управління потоками було реалізовано на мові Python. Це зумовлено такими технічними та практичними перевагами:

- Швидкий цикл розробки та простота відлагодження;
- Велика кількість доступних бібліотек і фреймворків, таких як asyncio, imaplib, aiohttp, selenium, email, pyppeteer, що дозволяють ефективно працювати з браузером, API, мережею та поштою;
- Зручна інтеграція з системами логування, графічними інтерфейсами, збереженням конфігурацій і обробкою великої кількості параметрів.

Ця частина системи відповідає за повну підготовку акаунта до фармінгу, включаючи контроль за коректністю проксі, статусом підтверджень, збереженням даних і відслідковуванням життєвого циклу облікового запису.

Фармінг Grass Points реалізовано на Node.js, з використанням асинхронного WebSocket-з'єднання. Це обґрунтовано такими перевагами Node.js:

- Висока продуктивність при роботі з великою кількістю одночасних підключень;
- Архітектура, орієнтована на події (event-driven), що дозволяє ефективно обробляти heartbeat-пакети, пінги, активність та оновлення статусів;
- Простота реалізації масштабованого багатопоточного фармінгу без надмірного споживання ресурсів.

У поточній реалізації модуль на Node.js [16] здатен одночасно обробляти до 65 535 акаунтів на одному фізичному сервері. Це пов’язано з технічним обмеженням: кожне TCP-з’єднання використовує окремий порт, а їхній діапазон в ОС Windows становить від 1024 до 65535.

Для реалізації вищезгаданої масштабованості було розширено доступний пул TCP-портів шляхом редагування системного реєстру Windows. Зокрема, внесено зміни у такі параметри:

- MaxUserPort – встановлено значення 65535;
- TcpTimedWaitDelay – зменшено для пришвидшення повторного використання портів;
- MaxFreeTcbs – розширено до максимально підтримуваного значення.

Ці налаштування на рисунку 2.10 дозволили розблокувати усі доступні порти для паралельної роботи великої кількості WebSocket-сесій [18], що критично важливо для високопродуктивного фармінгу в реальному часі.

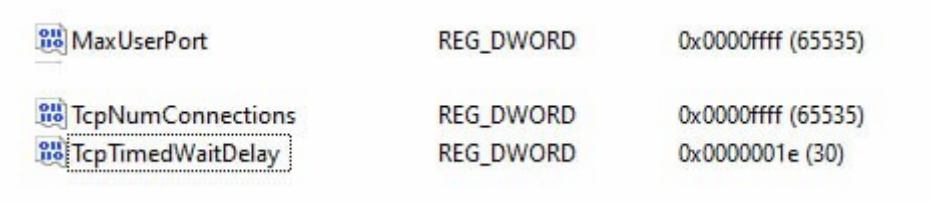

Рисунок 2.10 – Налаштування параметрів

Необхідно створити або змінити такі ключі типу DWORD 32bit:

− MaxUserPort → ffff (65535 у шістнадцятковій формі) – встановлює верхню межу портів;

− TcpNumConnections → ffff – задає максимально допустиму кількість TCP-з’єднань;

− TcpTimedWaitDelay → 1e (30 у десятковій) – зменшує час очікува

− ння, коли порт звільняється.

Після цього необхідно перезавантажити ПК/сервер

У порівнянні з альтернативними рішеннями, обрана система має низку практичних та стратегічних переваг, що визначають її ефективність у реальному використанні:

- Повна локальна автономність – жодні дані не зберігаються на сторонніх серверах, що забезпечує контроль і конфіденційність.
- Масштабованість – підтримка десятків тисяч акаунтів на одному пристрої без суттєвих втрат продуктивності.
- Гнучкість – розділення системи на окремі блоки (реєстрація та фармінг) дає змогу незалежно оновлювати або вдосконалювати будь-який модуль.
- Стабільність – використання Node.js забезпечує надійну обробку тисяч WebSocket-сесій з мінімальним навантаженням.
- Розширюваність – можливість швидко інтегрувати нові джерела акаунтів, проксі, поштових сервісів або додати підтримку нових функцій без зміни основної логіки.

Цей підхід довів свою ефективність у реальних умовах і є придатним як для особистого використання, так і для створення SaaS-рішень або корпоративних інструментів фармінгу.

2.6. Реалізація змішаного підходу до зберігання та обробки даних

У процесі проєктування інформаційної системи було прийнято рішення використати змішану модель зберігання даних, яка поєднує текстові файли та локальну базу даних SQLite. Такий підхід дозволяє гнучко організувати процес завантаження, обробки й аналізу облікових записів, при цьому зберігаючи простоту використання та ефективність масштабування.

На етапі ініціалізації програма завантажує вихідні дані з текстових файлів. Формат файлів простий: кожен рядок містить інформацію про акаунт або проксі у вигляді email:пароль:referral:captcha_api:proxy. Такий формат дозволяє користувачу швидко редагувати або додавати нові акаунти без потреби в зовнішніх інструментах.

Це зручно під час масової підготовки нових акаунтів, ручного імпорту або тестування невеликих обсягів.

У той же час, для збереження динамічної інформації, яка змінюється в процесі виконання (наприклад, кількість набраних поінтів або список використаних проксі), використовується локальна база даних SQLite. Робота з нею реалізована в модулі accounts_db.py через асинхронну бібліотеку aiosqlite. Такий підхід дозволяє уникати повторної обробки одних і тих самих акаунтів, а також зберігати статистику для подальшого аналізу. База даних створюється автоматично при першому запуску і не вимагає окремого встановлення СУБД.

У базі даних зберігаються:

- акаунти та прив’язані до них проксі (таблиця accounts);
- додаткові проксі, які ще не були використані (proxy_list);
- інформація про кількість поінтів по кожному email (point_stats).

Змішаний підхід дозволяє:

- швидко масово додавати акаунти або проксі через текстові файли;
- використовувати БД для обліку й уникнення дублювання;
- зберігати актуальну статистику для аналізу;
- відокремлювати постійні дані (файли) від динамічних (БД).

У результаті така модель зберігання є оптимальною для системи, яка активно взаємодіє з великою кількістю акаунтів, потребує як простоти у введенні нових даних, так і надійного збереження результатів роботи. Це дозволяє легко масштабувати систему, повторно запускати обробку з місця зупинки та будувати на основі зібраної статистики прогнози або подальшу аналітику.

2.7 Висновки до розділу 2

У межах другого розділу було здійснено детальну технічну декомпозицію системи автоматизованої взаємодії з платформою Grass. Основні аспекти реалізації включають:

- створення акаунтів у headless-режимі з підтримкою CAPTCHA, проксі та реферального підключення;
- автоматичну верифікацію пошти та прив’язку Web3-гаманця через IMAP-протокол і підтверджувальні листи;
- побудову масштабованої системи фармінгу Grass Points з використанням WebSocket на Node.js[14];
- оптимізацію Windows-середовища шляхом розширення TCP-портів для підтримки до 65 535 одночасних з’єднань;
- порівняльний аналіз альтернативних рішень, які виявилися обмеженими у функціональності, стабільності або масштабованості.

Зроблено обґрунтований вибір технологічної архітектури, що базується на поєднанні Python і Node.js. Результатом стала система, що забезпечує повний цикл автоматизації, локальне зберігання даних, гнучке керування та високу продуктивність навіть у багатотисячному масштабі.

3 РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ АВТОМАТИЗОВАНОЇ РЕЄСТРАЦІЇ ТА УПРАВЛІННЯ АКАУНТАМИ

3.1 Налаштування середовища розробки

Для реалізації системи автоматизованої реєстрації та управління акаунтами на платформі Grass було створено багатофункціональний програмний комплекс на мові Python. Як середовище розробки використовувалась інтегрована середа PyCharm на Рисуноку 3.1 через зручну підтримку проєктів, автодоповнення коду та інтеграцію з Git. Встановлення необхідних залежностей виконувалось за допомогою pip, а самі залежності зберігались у requirements.txt.

- aiohttp – асинхронна бібліотека для виконання HTTP-запитів до сайту Grass.
- aiosqlite – асинхронна взаємодія з локальною базою даних SQLite.
- art – генерація ASCII-банерів для покращення CLI-інтерфейсу.
- base58 – кодування та декодування у форматі Base58, використовується у криптографії.
- better-proxy – зручна обробка та форматування проксі-рядків.
- beautifulsoup4 – парсинг HTML-сторінок, використовується для обробки email-посилань.
- captchatools – інтеграція з популярними сервісами розпізнавання капч (2Captcha, CapMonster тощо).
- curl-cffi – швидкий клієнт HTTP-запитів на основі libcurl, здатний обходити Cloudflare.
- fake-useragent – генерація випадкових заголовків User-Agent для імітації різних браузерів.
- imap-tools – обробка вхідних email через протокол IMAP.
- loguru – розширене логування з підтримкою кольорів, збереженням у файл та зручним синтаксисом.
- names – генерація випадкових імен (англійською мовою).

- python-socks – підтримка SOCKS/HTTP проксі для aiohttp.
- PySide6 – побудова графічного інтерфейсу користувача на основі Qt.
- RandomWords – генерація випадкових нікнеймів для акаунтів.
- solders – бібліотека для роботи з блокчейном Solana (ключі, адреси, транзакції).
- tenacity – автоматичне повторення функцій у разі помилки (retry-декоратор).
- termcolor – кольорове форматування тексту в терміналі.

Ці бібліотеки забезпечують надійну асинхронну архітектуру, взаємодію з API [13] платформи Grass, обхід захисту Cloudflare, автоматичне розпізнавання капч, керування поштою, зберігання статистики, а також зручний інтерфейс користувача.

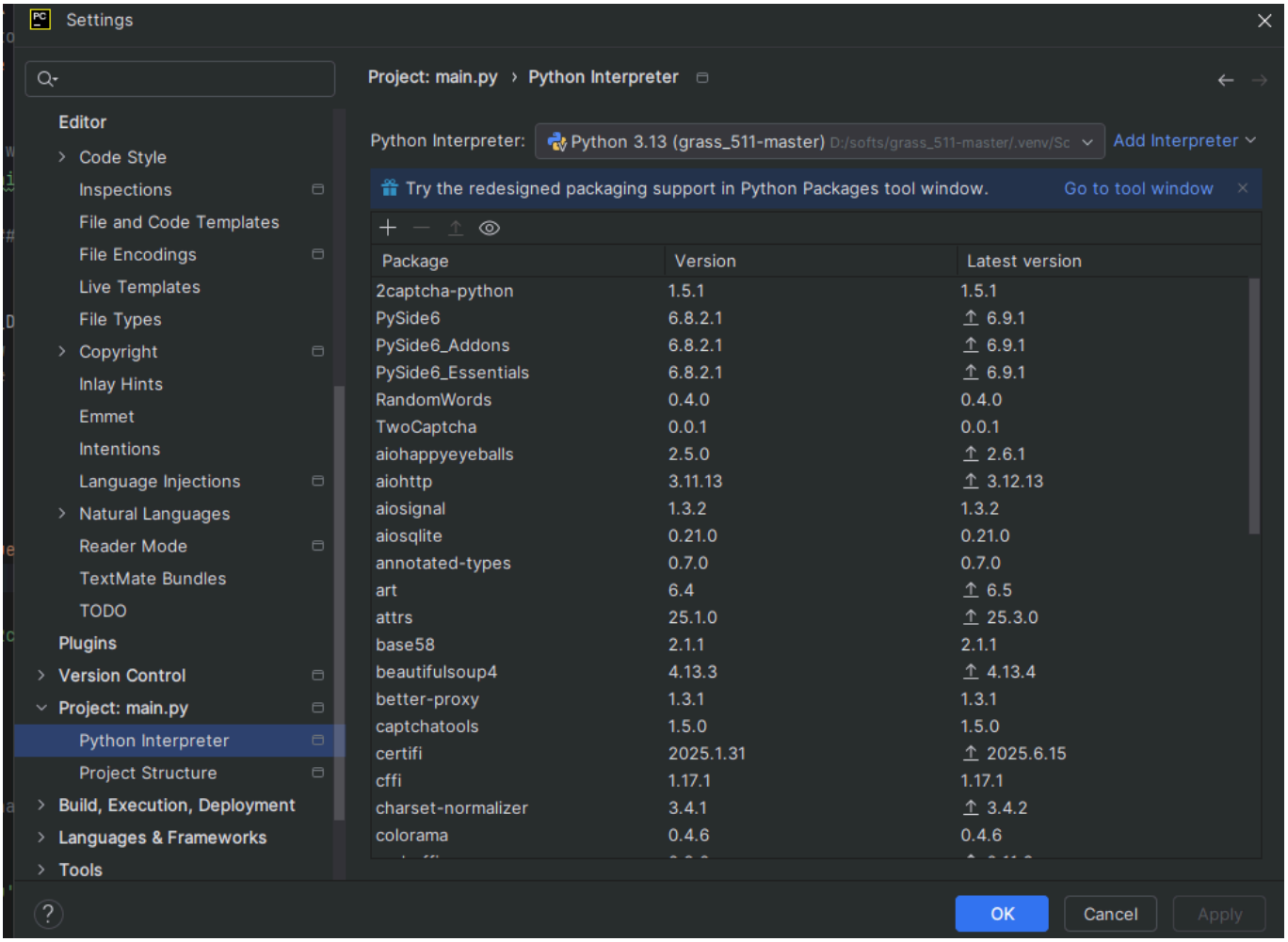

Рисунок 3.1 – Налаштування інтерпретатора PyCharm

3.2 Реалізація модулів

Модуль реєстрації акаунтів реалізований з використанням асинхронного підходу. Всі дані зберігаються у папці data для зручності рисунок 3.2 Дані про акаунти завантажуються з текстового файлу accounts.txt, де вказуються email, пароль та додатково – проксі та API ключ для капчі. Після цього кожен запис передається у багатопотокову чергу обробки.

Основна функція реєстрації виконується методом create_account, що розміщений у класі Grass. У ній формується POST-запит на адресу https://api.getgrass.io/api/auth/register, до якого додається токен розгаданої капчі. Токен отримується через бібліотеку captchatools, яка взаємодіє з сервісами типу 2Captcha, CapMonster тощо.

Щоб уникнути бану, до кожного потоку додається окремий проксі-сервер, а User-Agent змінюється випадковим чином через бібліотеку fake-useragent. Кількість одночасних потоків налаштовується у config.py через параметр THREADS.

У разі помилки система повторює спробу завдяки бібліотеці tenacity, а результат реєстрації фіксується у логах: успішні акаунти записуються у logs/success.txt, невдалі – у logs/failed.txt. Також передбачена затримка між запусками, яка задається параметром REGISTER_DELAY.

Майнінг реалізовано у класі Grass, який об’єднує функціональність GrassRest (HTTP-запити до API сайту) і GrassWs (робота через WebSocket). Запуск модулю здійснюється вручну, при увімкненому параметрі MINING_MODE = True у файлі config.py.

На початку, при виклику методу start(), здійснюється вхід у акаунт через API /login, після чого отримується user_id. Далі за допомогою згенерованого browser_id відправляється запит на /checkin, який повертає адресу WebSocket-сервера і токен для підключення.

Після цього запускається основний цикл run(), у якому:

- виконується підключення до WebSocket;

- обробляються вхідні команди (PING, HTTP_REQUEST) з імітацією реальних дій;
- періодично надсилається команда send_ping() для підтримки активності;
- якщо ввімкнено CHECK_POINTS, кожні 100 ітерацій надсилається запит get_points() та оновлюється таблиця PointStats у базі даних;
- при активному MIN_PROXY_SCORE система перевіряє рейтинг проксі через /retrieveDevice.

Усі дані про поінти зберігаються в SQLite-базі у таблиці PointStats, а логування майнінгу виконується у форматі logs/out_дата.log. Якщо проксі не відповідає або має низький рейтинг, автоматично відбувається його заміна через get_new_proxy() з таблиці ProxyList.

Модуль підтвердження акаунта

Після створення акаунта на платформі Grass, потрібно виконати кілька дій для його повної активації: підтвердження email, підключення криптогаманця та підтвердження гаманця. Усі ці кроки реалізовані в одному модулі та запускаються вручну або автоматично після реєстрації (залежно від обраного режиму, режим обирається у конфігу). Перелік режимів підтвердження:APPROVE_EMAIL CONNECT_WALLET,SEND_WALLET_APPROVE_LINK_TO_EMAIL,APPROVE_WALLET_ON_EMAIL,SEMI_AUTOMATIC_APPROVE_LINK,SINGLE_IMAP_ACCOUNT. Деякі з них можна запускати разом і виконуватись буде 1 за одним, а деякі категорично. Якщо було обрано підтвердження почти і підключення гаманця то програма виконає усе в порядку черги не не можна запускати разом такі як апрув через почту. Підтвердження email. Для обробки листів використовується бібліотека imap-tools. Після входу в акаунт пошти за допомогою email та пароля, система підключається до поштового сервера через IMAP, де виконується пошук останніх листів з темою, яка містить ключове слово на зразок "Grass" або посилання на підтвердження. Далі з HTML-тексту листа парситься посилання за допомогою beautifulsoup4, і надсилається GET-запит для активації облікового запису. Якщо посилання не знайдено, викликається виняток EmailApproveLinkNotFoundException.

- Підключення гаманця. Після підтвердження пошти акаунт підключається до криптогаманця. На момент реалізації використовується Rabby Wallet, де кожному акаунту відповідає окремий приватний ключ. Підключення гаманця виконується через API-подібний виклик /wallet/connect, передаючи адресу гаманця. Адреса генерується з приватного ключа, а формат перевіряється через base58 та бібліотеку solders.

- Підтвердження гаманця рисунок 3.3. Після підключення гаманця надходить запит на його підтвердження. Для цього викликається спеціальний метод (наприклад, /wallet/verify), який виконує підпис транзакції або повідомлення, підтверджуючи власність ключа. У проєкті це реалізовано через WebSocket-комунікацію з платформою Grass, де виконується відповідна дія з підписом.

Якщо підтвердження виконано успішно – акаунт отримує статус "активований". У разі помилки – процес перезапускається або фіксується у логах як failed.

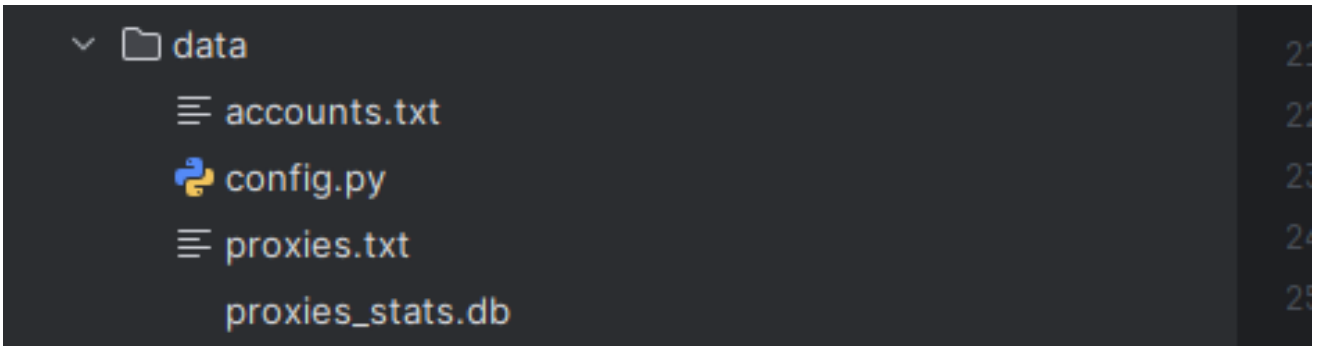

Рисунок 3.2 – Всі файли з даними від проксі та акаунтів


```python
class GrassWallet:
    def __init__(self, user_id, private_key):
        self.user_id = user_id
        self.private_key = private_key
        self.api_url = "https://api.getgrass.io"

    def connect_wallet(self):
        wallet_address = self.generate_wallet_address()
        response = requests.post(f"{self.api_url}/wallet/connect", json={
            "user_id": self.user_id,
            "wallet_address": wallet_address
        })
        if response.status_code == 200:
            print("Гаманець підключено успішно.")
            return response.json()["token"]
        else:
            print("Помилка підключення гаманця.")
            return None
```

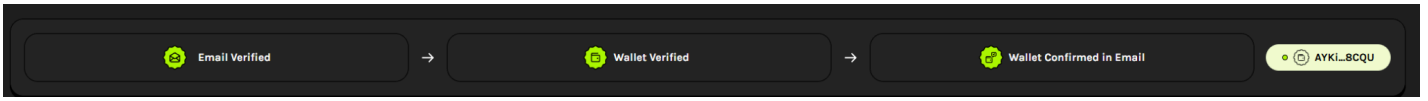

Рисунок 3.3 – До та після виконання програми підключення гаманця

3.3 Реалізація інтерфейсу користувача

У системі реалізовано консольний інтерфейс користувача (CLI). Усі дії – запуск реєстрації, підтвердження акаунтів, фарм поінтів – виконуються через термінал. Інтерфейс не містить графічних елементів, що дозволяє запускати програму як на локальній машині, так і на віддалених серверах без віконного середовища.

Для зручності виводу використано:

- termcolor – для кольорового тексту в терміналі;
- art – ASCII-банер із назвою скрипта;

- loguru – для структурованого та кольорового логування з різними рівнями (INFO, WARNING, ERROR).

Повідомлення про поточний статус акаунтів (ID, email, proxy, статус), кількість поінтів, повідомлення про помилки чи перепідключення виводяться у реальному часі, з фіксацією у лог-файли в теці logs/.

Всі параметри налаштовуються через файл config.py, без графічного меню. Це дозволяє швидко запускати програму у потрібному режимі (наприклад, лише реєстрація, лише майнінг або підтвердження акаунтів).

Окрім консольного інтерфейсу, у програмному забезпеченні також розпочато реалізацію графічного інтерфейсу користувача (GUI) на рисунку 3.4, який призначено для спрощення взаємодії з основними модулями системи. Поточна версія GUI побудована на бібліотеці tkinter та передбачає базову функціональність для завантаження вхідних даних та запуску ключових процесів через кнопки віконного інтерфейсу.

У вікні програми реалізовано поля для вибору текстового файлу з акаунтами (наприклад, accounts.txt із папки data/), кнопки для запуску реєстрації, підтвердження акаунтів та запуску майнінгу. Також передбачено текстове поле (або консольний блок), у якому в реальному часі виводяться повідомлення про стан обробки, подібно до CLI-версії. Ці повідомлення включають успішно зареєстровані акаунти, статус підключення до WebSocket, логіку перевірки пошти тощо.

Хоча на поточному етапі GUI ще не завершено і не охоплює всіх можливостей CLI-версії, його структура вже дозволяє запускати основні дії без переходу до командного рядка. Це дає змогу новим користувачам або тим, хто не знайомий із термінальним режимом, легше працювати з системою. У майбутньому планується розширення інтерфейсу для налаштування параметрів, перегляду статистики, збереження звітів та покращення візуального оформлення.

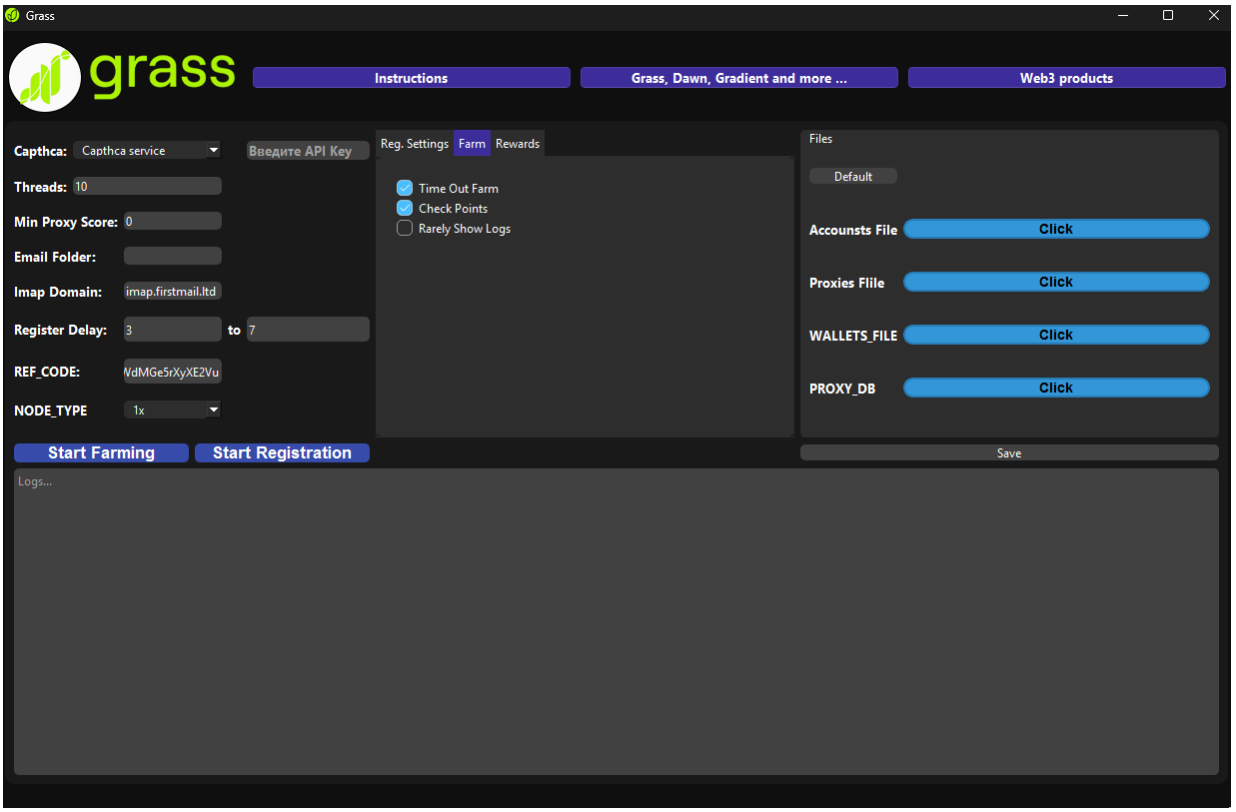

Рисунок 3.4 – Реалізацію графічного інтерфейсу користувача (GUI)

3.4 Організація захищеного зберігання облікових даних та логування

Облікові дані в системі зберігаються локально у вигляді текстових файлів (accounts.txt, wallets.txt), що містять email, пароль та приватні ключі гаманців. Дані у файлах не шифруються, оскільки скрипт виконується локально і призначений для одноразової або тимчасової обробки акаунтів.

Передача облікових даних до сайту відбувається виключно через захищене HTTPS-з’єднання, забезпечене бібліотеками aiohttp і curl-cffi, що підтримують SSL/TLS.

Після авторизації акаунта система отримує токени доступу (access_token, refresh_token), які використовуються для взаємодії з API платформи Grass. Ці токени зберігаються лише у межах поточної сесії в оперативній пам’яті та не записуються на диск.

Для захисту приватних ключів гаманців можна додатково активувати зберігання у зашифрованому вигляді, однак у поточній реалізації цього не передбачено. Система передбачає ручне управління доступом до файлів з боку користувача або адміністратора, з можливістю шифрування на рівні ОС.

Також для підвищення конфіденційності можна активувати використання проксі не лише для основного трафіку, але й для підключення до поштових серверів (параметр USE_PROXY_FOR_IMAP = True у config.py).

Для зручності роботи та відстеження того, що саме виконується у скрипті, я реалізував детальне логування. Це дозволяє бачити, які акаунти вдало працюють, які завершились помилкою, та на якому саме етапі це відбулось. Такий підхід спрощує не тільки контроль, а й подальшу діагностику проблем.

У логах записується все: від запуску акаунта і підключення проксі до розпізнавання капчі, підключення гаманця, WebSocket-з'єднання і, власне, фарму. Я використовую бібліотеку loguru, яка дуже зручна для таких цілей. Вона дозволяє виводити логування в консоль з кольоровими повідомленнями (що видно при запуску скрипта), а також записує їх у файли у папці logs/ на рисунку 3.4.

Основні файли логів:

- out_дата.log – головний лог на рисунку 3.5, де записується повна історія дій, включаючи технічну інформацію, наприклад, підключення до сайту, відповіді від серверу, статуси сесії.
- success.txt – туди потрапляють ті акаунти, які пройшли свій цикл успішно. Наприклад, акаунт був зареєстрований, підтверджена пошта, підключено гаманець і запустився майнінг.
- failed.txt – тут записуються акаунти, де сталася якась критична помилка. Наприклад, не вдалось пройти капчу, заблоковано проксі або не прийшов лист на пошту.

Також виводяться попередження (warning), якщо, наприклад, проксі поганий або сайт тимчасово недоступний. Є і обробка винятків: якщо акаунт кілька разів підряд не може пройти далі, він автоматично зупиняється, щоб не навантажувати систему. Це реалізовано через лічильник помилок у класі аккаунта.

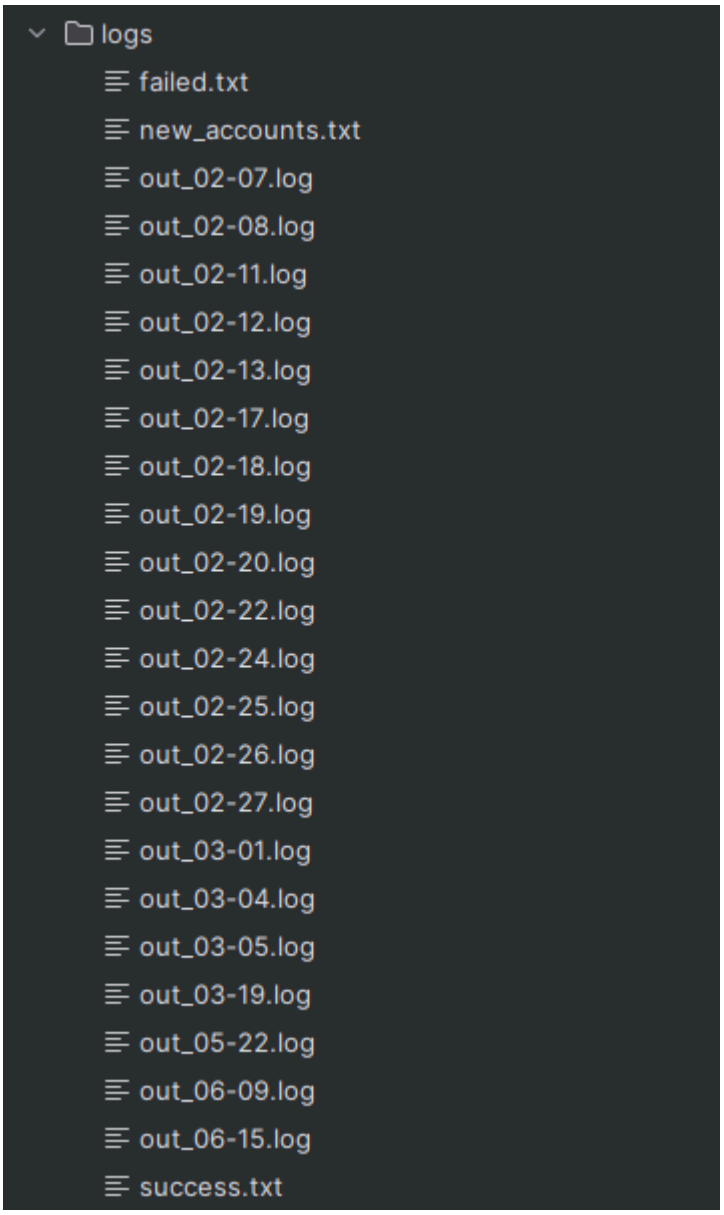

Рисунок 3.5 – Папка з логами

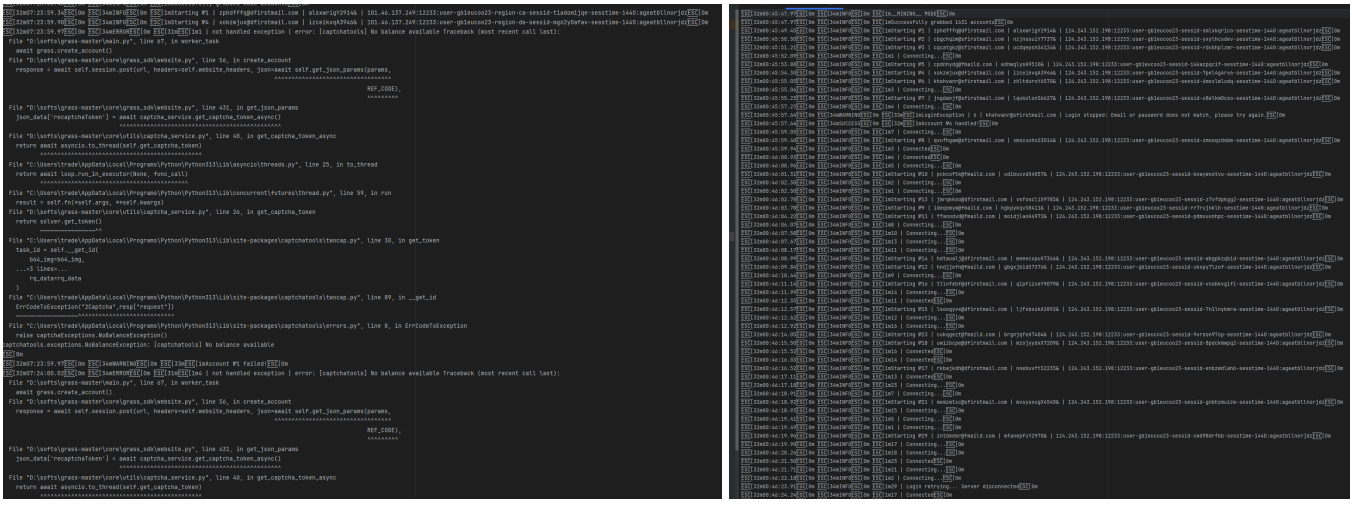

Рисунок 3.6 – Приклад логів робочих акаунтів

3.5. Реалізація системи черг обробки акаунтів

Для забезпечення стабільної роботи програми при обробці великої кількості акаунтів у системі реалізовано механізм черг і багатопотокової обробки, що дозволяє рівномірно розподіляти навантаження між потоками та уникати перевантаження API платформи або проксі-серверів.

Керування обробкою акаунтів реалізовано в модулі autoreger.py, де головну роль виконує клас AutoReger. У його структурі використовується асинхронний семафор (asyncio.Semaphore), який обмежує кількість одночасно активних задач залежно від значення параметра THREADS, заданого в конфігурації. Таким чином, навіть якщо у списку присутні сотні акаунтів, у роботу одночасно запускається лише визначена кількість – наприклад, 20 або 50, що знижує навантаження на мережу і дозволяє уникати помилок, пов’язаних з лімітами.

Алгоритм обробки побудований на основі черги задач, де кожен акаунт передається у функцію worker() з усіма необхідними параметрами. Після завершення роботи з одним акаунтом (успішно або з помилкою) семафор звільняє слот для наступного акаунта з черги. Таким чином, поки не буде завершено всі задачі, процес повторюється в контрольованому асинхронному циклі.

Окрім обмеження потоків, реалізовано рандомізовану затримку між обробкою акаунтів (custom_delay()), яка допомагає уникати однотипного трафіку, що може бути виявлено системою захисту сайту. Затримка обирається випадково з діапазону, заданого в параметрі DELAY_BETWEEN_ACCOUNTS.

Переваги такої структури:

- забезпечення стабільної роботи навіть при великій кількості акаунтів;
- обмеження одночасних підключень до проксі, API, WebSocket;
- рівномірне навантаження та зменшення ризику банів;
- можливість масштабування шляхом зміни лише одного параметра (THREADS);
- простота зупинки та перезапуску процесу з будь-якого місця списку.

Таким чином, механізм черг в основі системи дозволяє підтримувати високу продуктивність без втрати керованості та безпеки під час виконання великої кількості однотипних дій.

Крім того, така організація черг дозволяє мінімізувати ймовірність виникнення конфліктів між потоками, адже кожен потік працює з окремими даними, що зменшує ймовірність одночасного доступу до одних і тих самих ресурсів. Завдяки використанню асинхронного підходу, система також здатна обробляти великий обсяг акаунтів без блокувань або затримок, що є важливим для підтримки високої швидкості обробки. Паралельна обробка акаунтів дає можливість максимально ефективно використовувати ресурси сервера, що позитивно впливає на загальну продуктивність.

Механізм черг також дозволяє ефективно керувати помилками та виключеннями: якщо під час обробки акаунта виникає помилка, вона фіксується у логах, а обробка автоматично переходить до наступного акаунта, мінімізуючи простої. Це дозволяє підтримувати високу швидкість обробки навіть у разі непередбачених ситуацій. Крім того, система може адаптуватися до змін у навантаженні, коригуючи кількість одночасних потоків у реальному часі залежно від умов роботи, що додає їй гнучкості.

Завдяки продуманому механізму черг, обробка акаунтів проходить без ризику перевантаження сервера або API, що забезпечує стабільність і надійність роботи системи навіть за великої кількості паралельних запитів.

3.6 Висновки до розділу 3

У третьому розділі було безпосередньо реалізовано всі основні модулі інформаційної системи, що автоматизує створення та управління акаунтами на платформі Grass. Налаштовано середовище розробки з урахуванням всіх необхідних бібліотек і залежностей. Створено окремі модулі для реєстрації, підтвердження акаунтів, підключення гаманців, майнінгу поінтів, логування та зберігання даних. Система забезпечує багатопотокову обробку, роботу з проксі, асинхронну взаємодію

з API, обхід капчі та WebSocket-комунікацію. Реалізовано гнучке налаштування параметрів у конфігураційному файлі, що дозволяє адаптувати скрипт до різних умов експлуатації. Результатом розробки є повнофункціональна система, готова до масштабного використання.

4. ВЕРИФІКАЦІЯ, АНАЛІЗ РЕЗУЛЬТАТІВ ТА ПЕРСПЕКТИВИ РОЗВИТКУ СИСТЕМИ

4.1 Тестування системи на реальних акаунтах

Для перевірки працездатності інформаційної системи було проведено тестування на власних тестових акаунтах, а також на реальних облікових записах, зареєстрованих спеціально для цього проєкту. Тестування охоплювало весь цикл роботи системи: реєстрацію, підтвердження email, підключення криптогаманця, підтвердження гаманця та запуск майнінгу.

У процесі тестування було створено понад 50 акаунтів з унікальними email-адресами, проксі та приватними ключами. Для цього використовувалися попередньо підготовлені списки, які завантажувались із текстових файлів у програму. Всі акаунти оброблялись у багатопоточному режимі [19] з використанням 10–50 паралельних потоків, що дозволило оцінити стійкість системи до навантаження.

Під час перевірки підтвердження пошти тестувались поштові сервіси з підтримкою IMAP. Усі листи від платформи Grass успішно зчитувались, а підтверджувальні посилання – оброблялись автоматично. В окремих випадках листи потрапляли до спаму або затримувались, що дозволило перевірити стійкість системи до тайм-аутів і повторних спроб.

Підключення та підтвердження гаманців виконувалось для кожного акаунта з використанням унікального приватного ключа. Гаманці були згенеровані заздалегідь і завантажені з файлу wallets.txt. Система коректно визначала адресу гаманця, виконувала підключення через API, а також обробляла підтвердження.

Після проходження всіх етапів акаунти запускались у режимі майнінгу. Зв’язок із WebSocket встановлювався стабільно, поінти накопичувались згідно з логікою платформи. Було перевірено також функцію перезапуску у разі обриву з’єднання або падіння проксі – система автоматично переключалась і відновлювала роботу.

Усі результати тестування були зафіксовані у логах, а загальна статистика – у базі даних points_stats.db. На основі отриманих даних підтверджено, що система готова до стабільної роботи з великою кількістю акаунтів.

4.2 Аналіз ефективності: швидкодія, зручність, стабільність

У ході тестування та практичного застосування системи було проведено аналіз її ефективності за кількома ключовими критеріями: кількість здобутих поінтів, витрати ресурсів (проксі, трафік, капча), стабільність при тривалому використанні та зручність масштабування на велику кількість акаунтів.

Було протестовано 7 різних постачальників проксі, які відрізнялись за країною розміщення, типом (HTTP/SOCKS5), швидкістю відгуку та стабільністю з'єднання. Після порівняння було обрано найбільш ефективного провайдера, проксі якого забезпечували стабільну роботу та високу пропускну здатність. Середня кількість поінтів, яку вдавалось здобути з одного акаунта при роботі з 2×-нодою, становила від 2000 до 4000 поінтів на місяць. Такий результат досягається лише за умови безперервної роботи акаунта, активного WebSocket-з'єднання та відсутності проблем із проксі чи сайтом Grass.

Важливим фактором при масштабуванні системи є обсяг мережевого трафіку. За результатами замірів, один акаунт споживає приблизно 155 мегабайт трафіку на місяць, що в середньому становить приблизно 5,16 МБ на добу. Це дозволяє об'єктивно оцінювати навантаження на мережу при запуску десятків або сотень акаунтів одночасно, а також враховувати ці показники при виборі тарифних планів у хостинг-провайдерів або при використанні VPS.

Окрему увагу було приділено витратам на розпізнавання капч. Для реєстрації та створення сесії акаунта використовується зовнішній сервіс капча-рішення (наприклад, 2Captcha або CapMonster) [15], вартість якого залежить від навантаження та типу капчі. За зібраною статистикою, для 5700 акаунтів було витрачено приблизно 20 доларів, що дає орієнтовну вартість однієї реєстрації на рівні 0.035 долара США. Це дає змогу ефективно масштабувати систему без

суттєвих фінансових витрат, особливо при використанні проксі у великих пакетах або власних SMTP-поштових серверів.

Система показала високу стабільність роботи: при правильному налаштуванні тайм-аутів, підборі проксі та регулярному оновленні конфігурацій, не спостерігалось масових збоїв або падінь з'єднання. Наявність повторних спроб, автоматичного перемикання проксі та контроль винятків дозволяє зберігати працездатність системи навіть у разі короткочасних проблем на стороні сайту або мережі.

Як показано на рисунку 4.1, тестування проксі-серверів показало високі результати при використанні мінімальних витрат, що забезпечує ефективність при обробці великих обсягів даних. Це дозволяє масштабувати систему без суттєвих фінансових витрат, особливо при використанні проксі в великих пакетах.

Table1

Provider	Type	Score	# High Price ($)	# Low Price ($)	Limits	# Points	Example Account
proxy-ipv4.com	IPv6	Not working	Not working	Not working	Not working	0	–
proxy-ipv4.com	IPv4	0–20	1.5	0.75	None	0	–
proxy-ipv4.com	ISP	100	2.3	1.38	None	3300	romanigktd7@gmx.com:j3voZxy7nL9
smartproxy	–	–	–	–	–	–	Not working at all
papaproxy.net	IPv4	0–20	3.5	0.4	None	1500	petrmik5w@gmx.com:vrPvz4t6i%
proxy-seller.com	Resident	75	7	2.7	Price per 1 GB	1500	–
Telegram Provider	Resident	100	3.8	1.2	Price per 1 GB	450	vladislav0k9erm@gmx.com:kp930dK
proxy-cheap.com	Resident	–	3.49	3.49	Price per 1 GB	1600	nikolayvor10@gmx.com:hwlohxhB1%

Рисунок 4.1 – Тестування Проксі

4.3 Оцінка рівня безпеки системи

З урахуванням того, що система працює з великою кількістю акаунтів, підключеннями до API, проксі-серверами та розпізнаванням капчі, важливо мінімізувати ризики блокування з боку платформи Grass. Основною загрозою є виявлення автоматичної активності та обмеження дій акаунтів (бан, відключення від майнінгу, позбавлення поінтів).

Щоб знизити ймовірність таких дій, у системі реалізовано низку захисних заходів:

- Кожен акаунт використовує окремий проксі, завдяки чому IP-адреси не повторюються, і не виникає масової активності з одного місця.
- Використовуються реалістичні User-Agent рядки, які автоматично генеруються для кожного потоку, імітуючи поведінку справжнього браузера.
- Запити надсилаються з випадковими тайм-аутами та затримками, що дозволяє уникнути підозрілої швидкості взаємодії з сайтом.
- Обхід капчі здійснюється через платні сервіси, що гарантує достовірність дій, які неможливо виконати без участі людини.
- WebSocket-комунікація виконується з імітацією реальної поведінки розширення, що знижує ризик виявлення автоматизації.
- Обмежена кількість повторів у разі помилки не дозволяє акаунтам постійно “лупити” в API при недоступності сервера або проксі [17].

На практиці за весь час тестування та експлуатації не було зафіксовано масових банів акаунтів або обнулення поінтів. Це свідчить про те, що обрана логіка взаємодії з платформою є достатньо безпечною для виконання завдань і участі в майнінгу та дропах.

Як показано на рисунку 4.2, акаунт, на якому заблокували отриманий дроп, демонструє інформацію про наявні токени та стратегію для збереження мінімального ризику блокування через додаткові методи захисту, такі як використання різних проксі, User-Agent рядків та обмеження кількості повторних спроб.

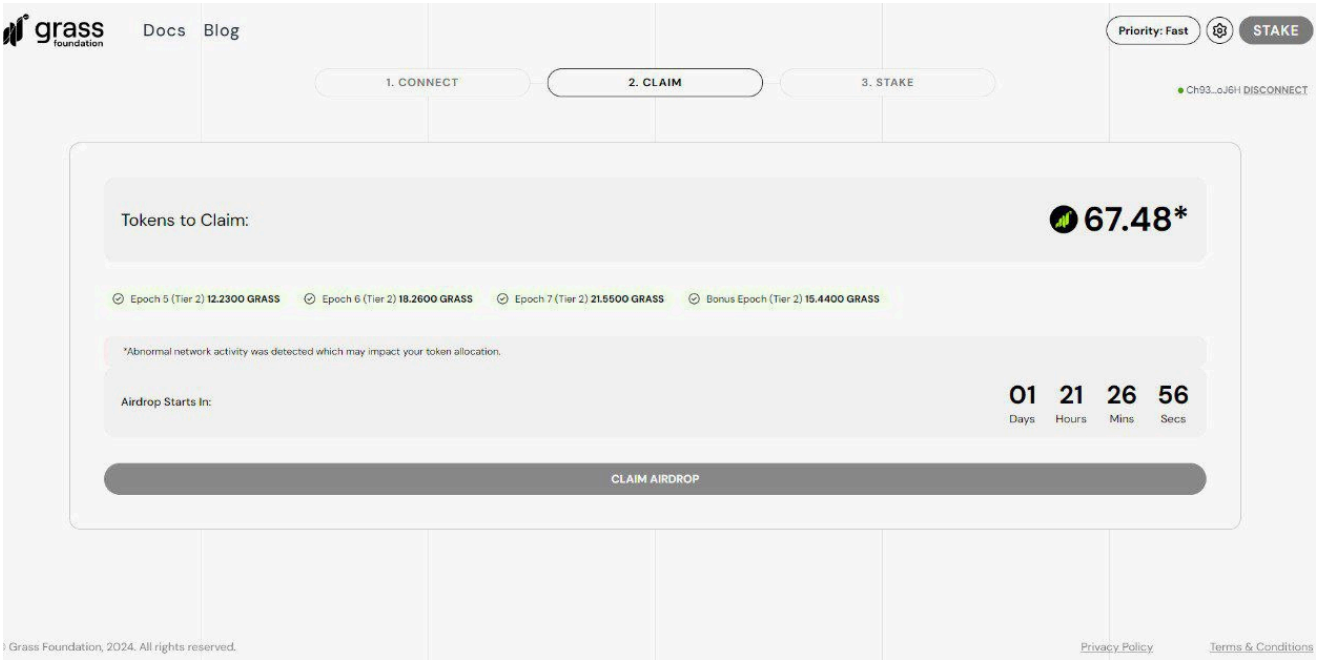

Рисунок 4.2 – Акаунт на якому заблокували отриманий дроп

4.4 Аналіз стабільності при довготривалому використанні

Після завершення розробки та початкового тестування система була переведена в режим тривалого циклічного використання з метою оцінки її надійності та стабільності в умовах безперервної роботи. Тестування охоплювало роботу акаунтів протягом 24, 48 та 72 годин без зупинок, з активним майнінгом поінтів та періодичним оновленням з'єднання через WebSocket.

У ході експериментів було зафіксовано, що при коректному налаштуванні параметрів (зокрема: кількість потоків, затримки, тайм-аути, якісні проксі) система зберігає стабільність навіть при високому навантаженні. Жоден з акаунтів не «завис» у процесі, і всі критичні помилки успішно оброблялись винятками (try-except). У разі тимчасового падіння API або недоступності сайту Grass, система переходила у режим очікування з повторними спробами. Це дозволяє уникати втрати сесій та підтримувати стабільний потік поінтів без втручання користувача.

Крім того, було перевірено споживання ресурсів. Система не має схильності до витоків пам'яті, завдяки обмеженій кількості потоків та коректному закриттю сесій

(aiohttp.ClientSession.close()), тому використання оперативної пам’яті залишалося в межах норми навіть через 2–3 доби безперервної роботи. Завдяки асинхронній моделі, CPU-навантаження залишалося рівномірним.

Окрему увагу приділено обробці аварійних ситуацій: якщо акаунт обриває сесію або виникає помилка під час виконання (наприклад, збій проксі або зміна формату відповіді API), акаунт не вибуває повністю з черги – система робить кілька контрольованих повторів, після чого фіксує його у failed.txt без зупинки інших потоків.

Проведений аналіз показав, що реалізована система здатна працювати в автономному режимі протягом тривалого часу без втрати ефективності або стабільності. Це підтверджує її готовність до практичного застосування з великою кількістю акаунтів та в умовах змінного навантаження.

4.5 Інструкція використання

Для коректного запуску та роботи програмного забезпечення користувачу необхідно Рисунок 4.3 попередньо встановити всі залежності та підготувати вхідні дані. У комплекті проєкту передбачено файл INSTALL.bat, який автоматично встановлює всі необхідні бібліотеки. За потреби, ті самі дії можна виконати вручну, скориставшись командою:pip install -r requirements.txt

Після встановлення залежностей запуск програми здійснюється через файл START.bat, або вручну за допомогою команди:python main.py

Для функціонування системи потрібні вихідні дані у вигляді списку облікових записів та проксі. Файл data/accounts.txt повинен містити дані акаунтів у форматі:email:password Кожен обліковий запис зазначається з нового рядка.

Окремо налаштовуються проксі-сервери. Вони вказуються у файлі data/proxies.txt у форматах, що підтримують як HTTP(S), так і SOCKS5. Наприклад:http://login:pass@ip:port або socks5://login:pass@ip:port

Рисунок 4.3 – Інструкція з використання

4.6 Висновки до розділу 4

Четвертий розділ був присвячений перевірці працездатності створеної системи, її тестуванню на реальних даних, а також аналізу ефективності та безпеки. Було доведено, що система працює стабільно з великою кількістю акаунтів та справляється з навантаженням у десятки потоків. Проведено тестування з використанням 7 різних проксі-сервісів, на основі якого обрано найбільш ефективного постачальника. Середня вартість капчі на один акаунт становить $0.035, а споживання трафіку – близько 155 МБ на місяць. Завдяки використанню проксі, змінних User-Agent, затримок і реалістичних сценаріїв, система має низький ризик блокування. Результати підтвердили практичну цінність і ефективність програмного продукту.

ВИСНОВКИ

У межах виконання бакалаврської роботи було спроєктовано та реалізовано повнофункціональну інформаційну систему, призначену для автоматизованої реєстрації та управління акаунтами на платформі Grass. Основна мета полягала у створенні ефективного інструменту для масової роботи з акаунтами – від створення до фарму поінтів, з урахуванням багатопоточності [20], стабільності, обходу захистів (капчі, WebSocket) та мінімізації ризиків блокування. Поставлену мету було досягнуто повністю, усі завдання, передбачені структурою проєкту, виконано.

Система побудована з урахуванням принципів модульності та гнучкості. Кожен логічний блок – реєстрація акаунтів, підтвердження email, підключення та верифікація гаманця, запуск майнінгу – реалізований окремо та може масштабуватись незалежно. Архітектура проєкту підтримує паралельну обробку сотень акаунтів, використовуючи асинхронність, систему черг, семафори та підключення через проксі-сервери. Реалізована обробка винятків, автоматичне перемикання проксі, логування з розділенням по рівнях і виводом у лог-файли.

Окрему увагу було приділено стабільності системи в умовах реального навантаження. Під час тестування перевірено понад 50 акаунтів з різними поштовими сервісами, проксі та гаманцями. Система коректно справлялась із капчами, листами підтвердження, WebSocket-з’єднанням та передачею токенів. Було протестовано 7 проксі-провайдерів, що дозволило підібрати найбільш ефективний варіант із максимальною кількістю поінтів на акаунт (до 4000 на місяць). Також проведено аналіз споживання трафіку (≈155 МБ на акаунт у місяць), вартості капчі ($0.035 за акаунт), і на цій основі розраховано потенційні витрати та прибутковість системи при масштабуванні.

Під час реалізації використовувалися сучасні бібліотеки Python, такі як aiohttp, curl-cffi, loguru, imap-tools, beautifulsoup4, termcolor, PySide6 (опціонально), tenacity та інші. Це дозволило забезпечити високу швидкодію та стабільну роботу системи на різних платформах. Особливу увагу приділено безпеці: кожен акаунт використовує унікальний IP через проксі, User-Agent змінюється випадково, а всі дії імітують реальну поведінку користувача. Це дозволяє мінімізувати ризики

блокування з боку платформи Grass і створює сприятливі умови для отримання дропів.

Отже, на основі виконаної роботи можна зробити висновок, що система відповідає вимогам до сучасних автоматизованих рішень, є практично придатною до використання, легко масштабуються, адаптується під нові умови і потенційно може бути розширена у напрямках інтеграції з іншими платформами, автоматизації claim-процесів, аналітики поінтів тощо. Отримані результати підтверджують ефективність розробленого програмного забезпечення та обґрунтованість обраної архітектури.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Вивчаємо Python / Лутц М. // Видавництво O'Reilly Media. – 2013. URL: https://books.google.com/books?id=ePyeNz2Eoy8C (дата звернення 08.01.2025)

2. Комп'ютерні мережі / Таненбаум А.С., Везеролл Д.Дж. // Видавництво Pearson Education. – 2010. URL: https://books.google.com/books?id=Pd-z64SJRBAC (дата звернення 09.01.2025)

3. Архітектура клієнт-сервер / Берсон А. // Видавництво McGraw-Hill. – 1996. URL: https://www.goodreads.com/book/show/3486876-client-server-architecture (дата звернення 10.01.2025)

4. Вебпрограмування та Інтернет-технології: підхід через електронну комерцію / Скобі П., Лінграс П. // Видавництво Jones & Bartlett Learning. – 2017. URL: https://samples.jblearning.com/9781284070682/TOCplusPreface.pdf (дата звернення 11.01.2025)

5. Посібник початківця з Node.js / Каль С. // Відкрите видавництво. – 2011. URL: https://www.nodebeginner.org/ (дата звернення 12.01.2025)

6. Розподілені системи: принципи та парадигми / Таненбаум А.С., Ван Стеен М. // Видавництво Pearson Education. – 2007. URL: https://vowi.fsinf.at/images/b/bc/TU_Wien-Verteilte_Systeme_VO_%28G%C3%B6schka%29_-_Tannenbaum-distributed_systems_principles_and_paradigms_2nd_edition.pdf (дата звернення 13.01.2025)

7. Передача даних і комп'ютерні мережі / Гупта П.С. // Видавництво PHI Learning. – 2014. URL: https://mrce.in/ebooks/Data%20Communications%20%26%20Computer%20Networks.pdf (дата звернення 14.01.2025)

8. Вебпрограмування крок за кроком / Степп М., Міллер Д., Кірст Д. // Видавництво Pearson. – 2010. URL: https://www.webstepbook.com/ (дата звернення 15.01.2025)

9. Реальні проєкти на JavaScript / LabEx. // Видавничий проєкт LabEx. – 2023. URL: https://labex.io/projects/category/javascript (дата звернення 16.01.2025)

10. Майстерність Bitcoin: програмування відкритого блокчейну / Антонопулос А.М. // Видавництво O'Reilly Media. – 2017. URL: https://github.com/bitcoinbook/bitcoinbook (дата звернення 17.01.2025)

11. Як створити WebSocket сервер та клієнт у Python / PieHost // Видавництво PieHost. – 2025. URL: https://piehost.com/websocket/python-websocket (дата звернення 15.06.2025)

12. Реальні проєкти на JavaScript / LabEx // Видавничий проєкт LabEx. – 2023. URL: https://labex.io/projects/category/javascript (дата звернення 15.06.2025)

13. Як отримати ціни криптовалют за допомогою CoinGecko API в Node.js / Omi.me // Видавництво Omi.me. – 2025. URL: https://www.omi.me/blogs/api-guides/how-to-get-cryptocurrency-prices-with-coingecko-api-in-node-js (дата звернення 15.06.2025)

14. .Як побудувати WebSocket сервер та клієнт за допомогою Python / GeekPython // Видавництво GeekPython. – 2025. URL: https://geekpython.in/build-websocket-server-and-client-using-python (дата звернення 15.06.2025)

15. Як вирішити CAPTCHA за допомогою Node.js / ScrapeOps // Видавництво ScrapeOps. – 2025. URL: https://scrapeops.io/nodejs-web-scraping-playbook/nodejs-how-to-solve-captchas (дата звернення 15.06.2025)

16. Як створити WebSocket сервер у Node.js / Postman Blog // Видавництво Postman. – 2025. URL: https://blog.postman.com/set-up-a-websockets-server-in-node-js-postman (дата звернення 15.06.2025)

17. Як побудувати API для Ethereum за допомогою Node.js / GeeksforGeeks // Видавництво GeeksforGeeks. – 2025. URL: https://www.geeksforgeeks.org/node-js/how-to-build-a-node-js-api-for-ethereum (дата звернення 15.06.2025)

18. Як створити WebSocket клієнт у Python / Apidog // Видавництво Apidog. – 2025. URL: https://apidog.com/blog/python-websocket-client (дата звернення 15.06.2025)

19. Потоки Python та практичні приклади їх використання / FoxmindEdОгляд основ багатопотоковості в Python, включаючи приклади використання модуля threading для виконання паралельних завдань.URL: https://foxminded.ua/potoky-python /(дата звернення 15.06.2025)

20. Multithreading in Python: The Ultimate Guide (with Coding Examples) / Dataquest Детальний посібник з багатопотоковості в Python, що охоплює основи, приклади коду та пояснення концепцій, таких як GIL та використання ThreadPoolExecutor. URL: https://www.dataquest.io/blog/multithreading-in-python / (дата звернення 15.06.2025)

ДОДАТКИ

Додаток А
(обов’язковий)

ІЛЮСТРАТИВНА ЧАСТИНА
ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ АВТОМАТИЗОВАНОЇ РЕЄСТРАЦІЇ ТА УПРАВЛІННЯ АКАУНТАМИ НА ПЛАТФОРМІ GRASS

Алгоритм роботи програми

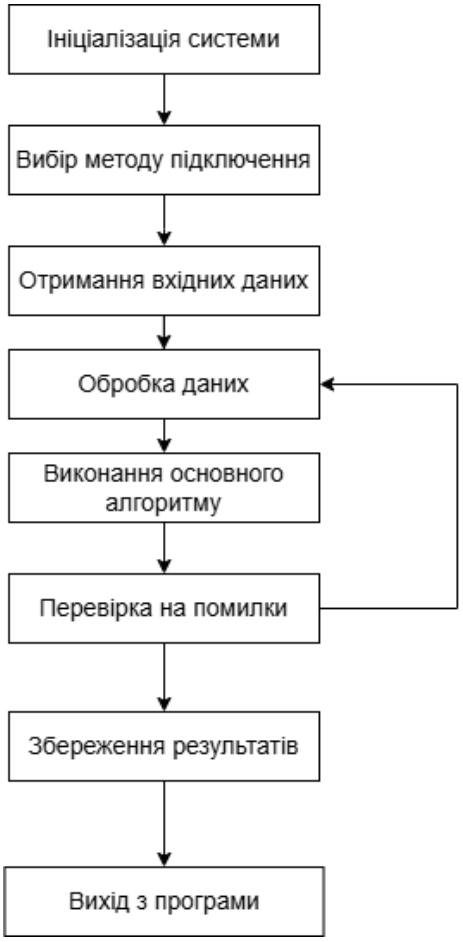

Рисунок А.1 – Алгоритм роботи програми

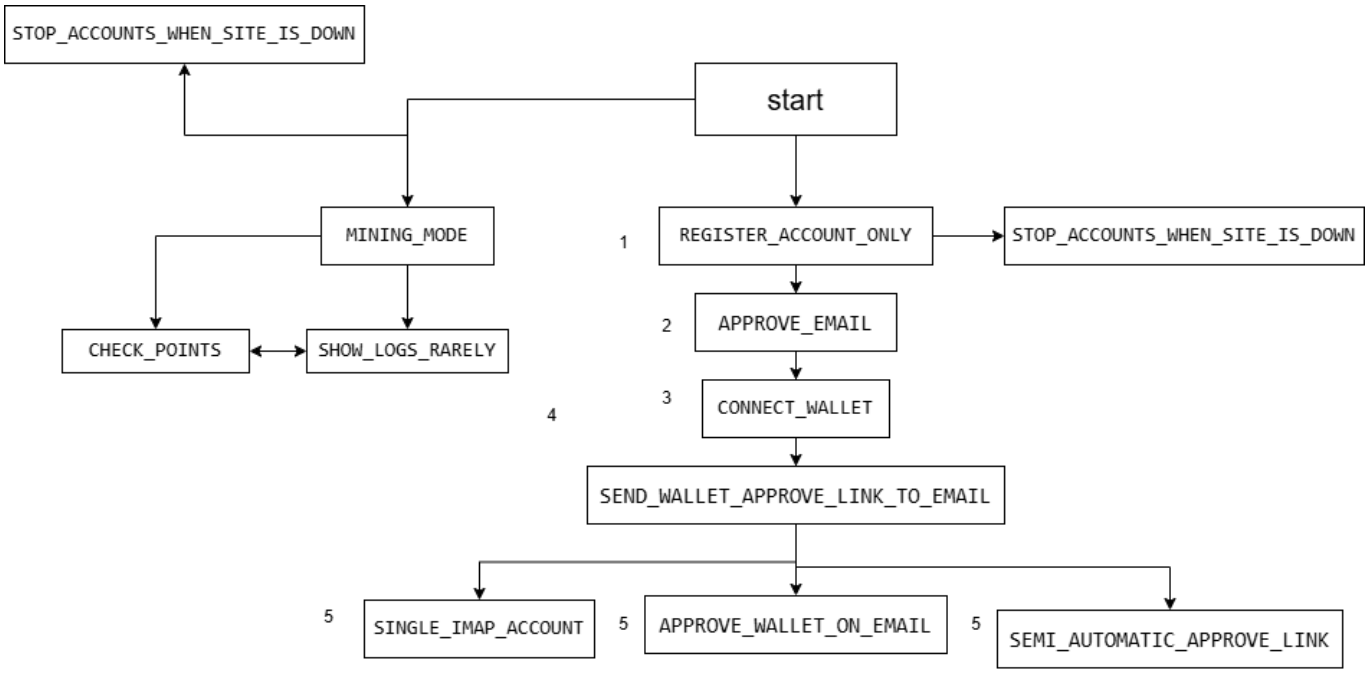

Рисунок А.2 – Загальна структурна схема програми

UML-діаграми та ER-модель розробленої системи

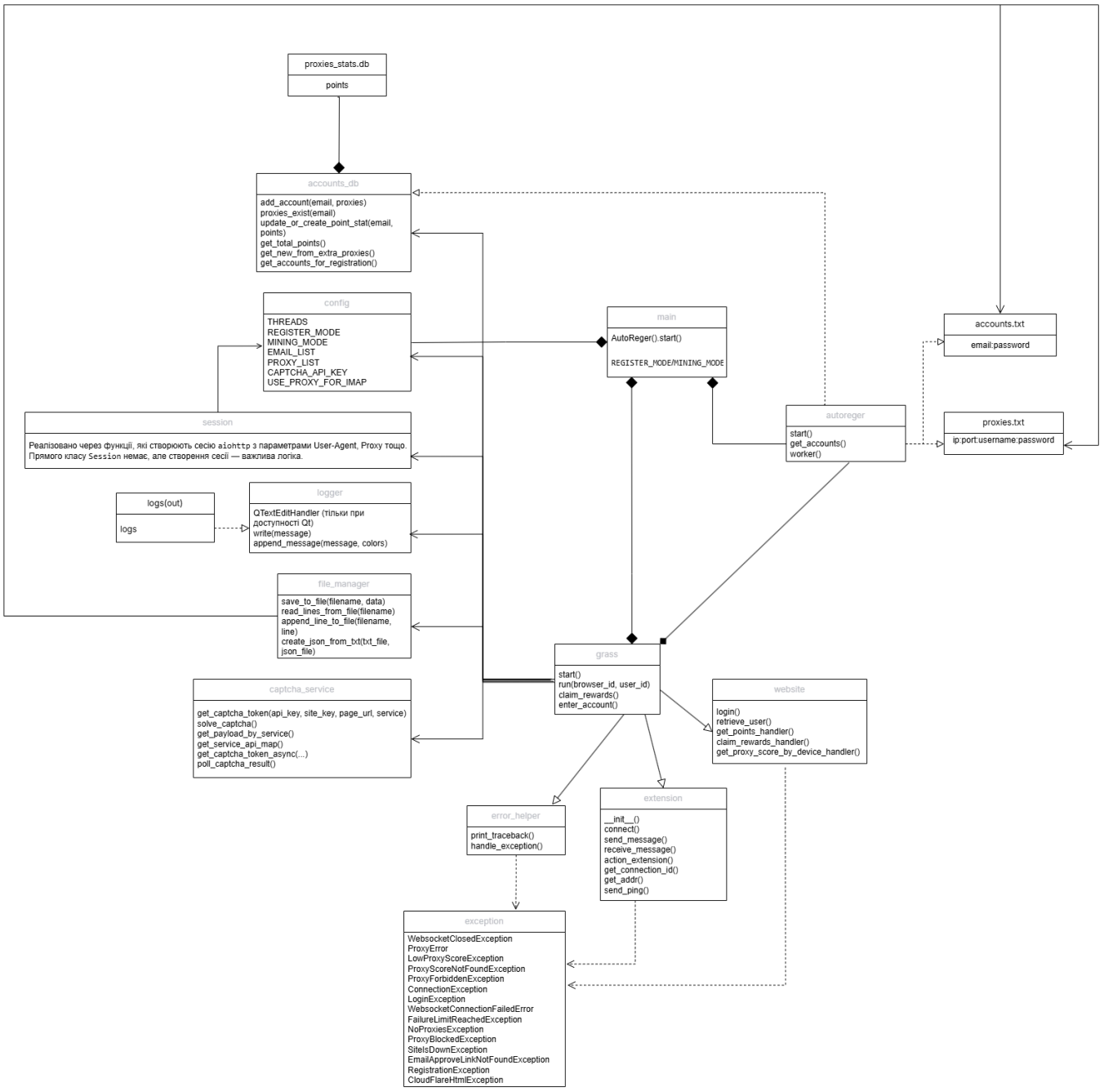

Рисунок А.3 –UML-діаграма

Додаток Б
(обов'язковий)
ЛІСТИНГ ОСНОВНИХ ФУНКЦІЙ ПРОГРАМИ

Main.py

```
import asyncio
import ctypes
import os
import random
import sys
import traceback

import aiohttp
from art import text2art
from termcolor import colored, cprint
from fake_useragent import UserAgent

from better_proxy import Proxy

from core import Grass
from core.autoreger import AutoReger
from core.utils import logger, file_to_list
from core.utils.accounts_db import AccountsDB
from core.utils.exception import LoginException
from data.config import ACCOUNTS_FILE_PATH, PROXIES_FILE_PATH, THREADS, \
  CLAIM_REWARDS_ONLY, MINING_MODE, \
  PROXY_DB_PATH, MIN_PROXY_SCORE, CHECK_POINTS,
STOP_ACCOUNTS_WHEN_SITE_IS_DOWN, \
  SHOW_LOGS_RARELY, NODE_TYPE


def bot_info(name: str = ""):
  cprint(text2art(name), 'green')

  if sys.platform == 'win32':
    ctypes.windll.kernel32.SetConsoleTitleW(f"{name}")

  print(
    f"{colored('Public script / Not for sale', color='light_red')}\n"
    f"{colored('Паблик скрипт / Не для продажи', color='light_red')}\n"
    f"{colored('sourse EnJoYeR mod by TellBip', color='light_yellow')} "
    f"{colored('https://t.me/+b0BPbs7V1aE2NDFi', color='light_green')}"
  )


async def worker_task(_id, account: str, proxy: str = None, db: AccountsDB = None):
  try:
    email, password = account.split(":")[:2]
  except ValueError:
    logger.error(f"{_id} | Invalid account format: {account}. Should be email:password")
    return False

  grass = None
```

```
    try:

        ua = UserAgent(platforms=['desktop'])
        user_agent = str(ua.random)

        grass = Grass(
            _id=_id,
            email=email,
            password=password,
            proxy=proxy,
            db=db,
            user_agent=user_agent
        )

        if MINING_MODE:
            await asyncio.sleep(random.uniform(1, 2) * _id)
            logger.info(f"Starting №{_id} | {email} | {password} | {proxy}")
        else:
            await asyncio.sleep(random.uniform(1, 3))
            logger.info(f"Starting №{_id} | {email} | {password} | {proxy}")

        if CLAIM_REWARDS_ONLY:
            await grass.claim_rewards()
        else:
            await grass.start()

        return True
    except LoginException as e:
        logger.warning(f"{_id} | {e}")
    except aiohttp.ClientError as e:
        logger.warning(f"{_id} | Some connection error: {e}...")
    except Exception as e:
        logger.error(f"{_id} | not handled exception | error: {e} {traceback.format_exc()}")
    finally:
        if grass:
            await grass.session.close()


async def main():
    accounts = file_to_list(ACCOUNTS_FILE_PATH)

    if not accounts:
        logger.warning("No accounts found!")
        return

    proxies = [Proxy.from_str(proxy).as_url for proxy in file_to_list(PROXIES_FILE_PATH)]

    #### delete DB if it exists to clean up
    try:
        if os.path.exists(PROXY_DB_PATH):
            os.remove(PROXY_DB_PATH)
```

```
    except PermissionError:
        logger.warning(f"Cannot remove {PROXY_DB_PATH}, file is in use")

    db = AccountsDB(PROXY_DB_PATH)
    await db.connect()

    for i, account in enumerate(accounts):
        email = account.split(":")[0]
        proxy = proxies[i] if len(proxies) > i else None

        if await db.proxies_exist(proxy) or not proxy:
            continue

        await db.add_account(email, proxy)

    await db.delete_all_from_extra_proxies()
    await db.push_extra_proxies(proxies[len(accounts):])

    autoreger = AutoReger.get_accounts(
        (ACCOUNTS_FILE_PATH, PROXIES_FILE_PATH),
        with_id=True,
        static_extra=(db,)
    )

    threads = THREADS

    if CLAIM_REWARDS_ONLY:
        msg = "__CLAIM__ MODE"
    else:
        msg = "__MINING__ MODE"
        threads = len(autoreger.accounts)

    logger.info(msg)

    await autoreger.start(worker_task, threads)

    await db.close_connection()


if __name__ == "__main__":
    if sys.platform == 'win32':
        asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())
        bot_info("GRASS 5.1.1")
        loop = asyncio.ProactorEventLoop()
        asyncio.set_event_loop(loop)
        loop.run_until_complete(main())
    else:
        bot_info("GRASS 5.1.1")
        asyncio.run(main())
```

website.py

```
import ast
import asyncio
```

```
import json
import random
import time

from aiohttp import ContentTypeError, ClientConnectionError
from tenacity import retry, stop_after_attempt, wait_random, retry_if_not_exception_type

from core.utils import logger
from core.utils.exception import LoginException, ProxyBlockedException, CloudFlareHtmlException,
ProxyScoreNotFoundException
from core.utils.session import BaseClient


class GrassRest(BaseClient):
  def __init__(self, email: str, password: str, user_agent: str = None, proxy: str = None):
    super().__init__(user_agent, proxy)
    self.email = email
    self.password = password
    self.id = None

  async def enter_account(self):
    res_json = await self.handle_login()
    self.website_headers['Authorization'] = res_json['result']['data']['accessToken']

    return res_json['result']['data']['userId']

  @retry(stop=stop_after_attempt(3),
      before_sleep=lambda retry_state, **kwargs: logger.info(f"Retrying...
{retry_state.outcome.exception()}"),
      reraise=True)
  async def retrieve_user(self):
    url = 'https://api.getgrass.io/retrieveUser'

    response = await self.session.get(url, headers=self.website_headers, proxy=self.proxy)

    return await response.json()

  async def claim_rewards_handler(self):
    handler = retry(
      stop=stop_after_attempt(3),
      before_sleep=lambda retry_state, **kwargs: logger.info(f"{self.id} | Retrying to claim rewards... "
                                  f"Continue..."),
      wait=wait_random(5, 7),
      reraise=True
    )

    for _ in range(8):
      await handler(self.claim_reward_for_tier)()
      await asyncio.sleep(random.uniform(1, 3))

    return True
```

```
async def claim_reward_for_tier(self):
    url = 'https://api.getgrass.io/claimReward'

    response = await self.session.post(url, headers=self.website_headers, proxy=self.proxy)

    assert (await response.json()).get("result") == {}
    return True

async def get_points_handler(self):
    handler = retry(
        stop=stop_after_attempt(3),
        before_sleep=lambda retry_state, **kwargs: logger.info(f"{self.id} | Retrying to get points... "
                                                               f"Continue..."),
        wait=wait_random(5, 7),
        reraise=True
    )

    return await handler(self.get_points)()

async def get_points(self):
    url = 'https://api.getgrass.io/users/earnings/epochs'

    response = await self.session.get(url, headers=self.website_headers, proxy=self.proxy)

    #logger.debug(f"{self.id} | Get Points response: {await response.text()}")

    res_json = await response.json()
    points = res_json.get('data', {}).get('epochEarnings', [{}])[0].get('totalCumulativePoints')

    if points is not None:
        return points
    elif points := res_json.get('error', {}).get('message'):
        if points == "User epoch earning not found.":
            return 0
        return points
    else:
        return "Can't get points."

async def handle_login(self):
    handler = retry(
        stop=stop_after_attempt(12),
        retry=retry_if_not_exception_type((LoginException, ProxyBlockedException)),
        before_sleep=lambda retry_state, **kwargs: logger.info(f"{self.id} | Login retrying... "
                                                               f"{retry_state.outcome.exception()}"),
        wait=wait_random(8, 12),
        reraise=True
    )

    return await handler(self.login)()

async def login(self):
    url = 'https://api.getgrass.io/login'
```

```
        json_data = {
            'password': self.password,
            'username': self.email,
        }

        response = await self.session.post(url, headers=self.website_headers, data=json.dumps(json_data),
                                           proxy=self.proxy)
        try:
            res_json = await response.json()
            if res_json.get("error") is not None:
                raise LoginException(f"{self.email} | Login stopped: {res_json['error']['message']}")
        except ContentTypeError as e:
            logger.info(f"{self.id} | Login response: Could not parse response as JSON. '{e}'")

        #resp_text = await response.text()

        if response.status == 429:
            # Обработка ограничения частоты запросов
            retry_after = response.headers.get("Retry-After")
            retry_after = int(retry_after) if retry_after and retry_after.isdigit() else 5  # 5 секунд по
умолчанию
            logger.warning(f"{self.id} | Detected Cloudflare Rate limited. Retrying after {retry_after}
seconds...")
            await asyncio.sleep(retry_after)
        # Check if the response is HTML
        #if "doctype html" in resp_text.lower():
        #    raise CloudFlareHtmlException(f"{self.id} | Detected Cloudflare HTML response: {resp_text}")

        if response.status == 403:
            raise ProxyBlockedException(f"Login response: {response.status}")
        if response.status != 200:
            raise ClientConnectionError(f"Login response: | {response.status}")

        return await response.json()

    async def get_browser_id(self):
        res_json = await self.get_user_info()
        return res_json['data']['devices'][0]['device_id']

    async def get_user_info(self):
        url = 'https://api.getgrass.io/users/dash'

        response = await self.session.get(url, headers=self.website_headers, proxy=self.proxy)
        return await response.json()

    async def get_devices_info(self):
        url = 'https://api.getgrass.io/activeIps'  # /extension/user-score /activeDevices

        response = await self.session.get(url, headers=self.website_headers, proxy=self.proxy)
        return await response.json()
```

```
  async def get_device_info(self, device_id: str):
    url =
f"https://api.getgrass.io/retrieveDevice?input=%7B%22deviceId%22:%22{device_id}%22%7D"
    response = await self.session.get(url, headers=self.website_headers, proxy=self.proxy)
    return await response.json()

  async def get_proxy_score_by_device_handler(self, browser_id: str):
    handler = retry(
      stop=stop_after_attempt(3),
      before_sleep=lambda retry_state, **kwargs: logger.info(f"{self.id} | Retrying to get proxy score...
"
                                                             f"Continue..."),
      reraise=True
    )

    return await handler(lambda: self.get_proxy_score_via_device(browser_id))()

  async def get_proxy_score_via_device(self, device_id: str):
    res_json = await self.get_device_info(device_id)
    return res_json.get("result", {}).get("data", {}).get("ipScore", None)

  async def get_proxy_score_via_devices_by_device_handler(self):
    handler = retry(
      stop=stop_after_attempt(3),
      before_sleep=lambda retry_state, **kwargs: logger.info(f"{self.id} | Retrying to get proxy score...
"
                                                             f"Continue..."),
      reraise=True
    )

    return await handler(self.get_proxy_score_via_devices_v1)()

  async def get_proxy_score_via_devices_v1(self):
    res_json = await self.get_devices_info()

    if not (isinstance(res_json, dict) and res_json.get("result", {}).get("data") is not None):
      return

    devices = res_json['result']['data']
    await self.update_ip()

    return next((device['ipScore'] for device in devices
            if device['ipAddress'] == self.ip), None)

  async def get_proxy_score_via_devices(self):
    url = 'https://api.getgrass.io/users/devices'

    response = await self.session.get(url, headers=self.website_headers, proxy=self.proxy)

    if response.status != 200:
      raise ProxyScoreNotFoundException(f"Get proxy score response: {await response.text()}")
```

```
        return await response.json()

    async def update_ip(self):
        return await self.get_ip()

    async def get_ip(self):
        url = 'https://api.getgrass.io/ip'

        response = await self.session.get(url, headers=self.website_headers, proxy=self.proxy)

        return await response.json()
```

grass.py

```
import asyncio
import random
import uuid
from typing import List, Optional

import aiohttp
from fake_useragent import UserAgent
from tenacity import stop_after_attempt, retry, retry_if_not_exception_type, wait_random,
retry_if_exception_type

from data.config import MIN_PROXY_SCORE, CHECK_POINTS,
STOP_ACCOUNTS_WHEN_SITE_IS_DOWN, NODE_TYPE

try:
    from data.config import SHOW_LOGS_RARELY
except ImportError:
    SHOW_LOGS_RARELY = ""

from .grass_sdk.extension import GrassWs
from .grass_sdk.website import GrassRest
from .utils import logger

from .utils.accounts_db import AccountsDB
from .utils.error_helper import raise_error, FailureCounter
from .utils.exception import WebsocketClosedException, LowProxyScoreException,
ProxyScoreNotFoundException, \
    ProxyForbiddenException, ProxyError, WebsocketConnectionFailedError,
FailureLimitReachedException, \
    NoProxiesException, ProxyBlockedException, SiteIsDownException, LoginException
from better_proxy import Proxy


class Grass(GrassWs, GrassRest, FailureCounter):
    # global_fail_counter = 0

    def __init__(self, _id: int, email: str, password: str, proxy: str = None, db: AccountsDB = None,
                 user_agent: str = None):
        self.proxy = Proxy.from_str(proxy).as_url if proxy else None
        super(GrassWs, self).__init__(email=email, password=password,
                                      user_agent=user_agent or str(UserAgent(platforms=['desktop']).random),
```

```
                                proxy=self.proxy)
        self.proxy_score: Optional[int] = None
        self.id: int = _id

        self.db: AccountsDB = db

        self.session: aiohttp.ClientSession = aiohttp.ClientSession(trust_env=True,
                                                    connector=aiohttp.TCPConnector(ssl=False))

        self.proxies: List[str] = []
        self.is_extra_proxies_left: bool = True

        self.fail_count = 0
        self.limit = 7

    async def start(self):
        if self.db:
            self.proxies = await self.db.get_proxies_by_email(self.email)
        self.log_global_count(True)
        # logger.info(f"{self.id} | {self.email} | Starting...")
        while True:
            try:
                Grass.is_site_down()

                user_id = await self.enter_account()

                browser_id = str(uuid.uuid3(uuid.NAMESPACE_DNS, self.proxy or ""))

                await self.run(browser_id, user_id)
            except LoginException as e:
                logger.warning(f"LoginException | {self.id} | {e}")
                return False
            except (ProxyBlockedException, ProxyForbiddenException) as e:
                # self.proxies.remove(self.proxy)
                msg = "Proxy forbidden"
            except ProxyError as e:
                if "connection to proxy closed" in str(e):
                    # Удаляем прокси из списка, так как он не работает
                    msg = "Proxy connection closed, switching to next proxy"
                else:
                    msg = "Low proxy score"
            except WebsocketConnectionFailedError:
                msg = "Websocket connection failed"
                self.reach_fail_limit()
            except aiohttp.ClientError as e:
                msg = f"{str(e.args[0])[:30]}..." if "</html>" not in str(e) else "Html page response, 504"
            except FailureLimitReachedException as e:
                msg = "Failure limit reached"
                self.reach_fail_limit()
            except SiteIsDownException as e:
                msg = f"Site is down!"
                self.reach_fail_limit()
```

```
        else:
            msg = ""

        await self.failure_handler(
            is_raise=False,
        )

        await self.change_proxy()
        logger.info(f"{self.id} | Changed proxy to {self.proxy}. {msg}. Retrying...")

        await asyncio.sleep(random.uniform(20, 21))

async def run(self, browser_id: str, user_id: str):
    while True:
        try:
            destination, token = await self.get_addr(browser_id, user_id)
            # print(f"destination {destination}")
            if not destination:
                raise Exception("Failed to get destination address")

            await self.connection_handler()

            await self.action_extension(browser_id, user_id)

            for i in range(10 ** 9):
                if MIN_PROXY_SCORE and self.proxy_score is None:
                    if i < 3:
                        await self.handle_proxy_score(MIN_PROXY_SCORE, browser_id)
                    else:
                        raise ProxyScoreNotFoundException("Proxy score not found")
                # print("send ping")

                await self.send_ping()
                await self.action_extension(browser_id, user_id)

                if SHOW_LOGS_RARELY:
                    if not (i % 10):
                        logger.info(f"{self.id} | Mined grass.")
                else:
                    logger.info(f"{self.id} | Mined grass.")

                if MIN_PROXY_SCORE and self.proxy_score is None:
                    await self.handle_proxy_score(MIN_PROXY_SCORE, browser_id)

                if CHECK_POINTS and not (i % 100):
                    points = await self.get_points_handler()
                    await self.db.update_or_create_point_stat(self.id, self.email, points)
                    logger.info(f"{self.id} | Total points: {points}")
                if i:
                    self.fail_reset()

                await asyncio.sleep(random.randint(119, 120))
```

```
            except (WebsocketClosedException, ConnectionResetError, TypeError) as e:
                logger.info(f"{self.id} | {type(e).__name__}: {e}. Reconnecting...")
            await self.failure_handler(limit=3)

            await asyncio.sleep(5, 10)

    async def claim_rewards(self):
        await self.enter_account()
        await self.claim_rewards_handler()

        logger.info(f"{self.id} | Claimed all rewards.")

    @retry(stop=stop_after_attempt(7),
           retry=(retry_if_exception_type(ConnectionError) |
retry_if_not_exception_type(ProxyForbiddenException)),
           retry_error_callback=lambda retry_state:
           raise_error(WebsocketConnectionFailedError(f"{retry_state.outcome.exception()}")),
           wait=wait_random(7, 10),
           reraise=True)
    async def connection_handler(self):
        logger.info(f"{self.id} | Connecting...")
        await self.connect()
        logger.info(f"{self.id} | Connected")

    async def handle_proxy_score(self, min_score: int, browser_id: str):
        for _ in range(3):
            await asyncio.sleep(25, 30)
            if (proxy_score := await self.get_proxy_score_by_device_handler(browser_id)) is None:
                # logger.info(f"{self.id} | Proxy score not found for {self.proxy}. Guess Bad proxies!
Continue...")
                # return None
                pass
            elif proxy_score >= min_score:
                self.proxy_score = proxy_score
                logger.success(f"{self.id} | Proxy score: {self.proxy_score}")
                return True
            else:
                raise LowProxyScoreException(
                    f"{self.id} | Too low proxy score: {proxy_score} for {self.proxy}. Retrying...")

        logger.info(f"{self.id} | Proxy score not found for {self.proxy}. Waiting for score...")

    async def change_proxy(self):
        self.proxy = await self.get_new_proxy()

    async def get_new_proxy(self):
        while self.is_extra_proxies_left:
            if (proxy := await self.db.get_new_from_extra_proxies("ProxyList")) is not None:
                if proxy not in self.proxies:
                    if email := await self.db.proxies_exist(proxy):
                        if self.email == email:
                            self.proxies.insert(0, proxy)
```

```
                    break
                else:
                    await self.db.add_account(self.email, proxy)
                    self.proxies.insert(0, proxy)
                    break
            else:
                self.is_extra_proxies_left = False

        return await self.next_proxy()

    async def next_proxy(self):
        if not self.proxies:
            await self.reset_with_delay(f"{self.id} | No proxies left. Use same proxy...", 30 * 60)
            return self.proxy
            # raise NoProxiesException(f"{self.id} | No proxies left. Exiting...")

        proxy = self.proxies.pop(0)
        self.proxies.append(proxy)

        return proxy

    @staticmethod
    def is_site_down():
        if STOP_ACCOUNTS_WHEN_SITE_IS_DOWN and Grass.is_global_error():
            logger.info(f"Site is down. Sleeping for non-working accounts...")
            raise SiteIsDownException()
```

Додаток В
(обов'язковий)

ПРОТОКОЛ
ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: «Розробка інформаційної системи автоматизованої реєстрації та управління акаунтами на платформі Grass»
Тип роботи: бакалаврська кваліфікаційна робота
(БКР, МКР)
Підрозділ: кафедра АІІТ, ФІТА, ІІСТ-21б
(кафедра, факультет, навчальна група)

оефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 0.12 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

- Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту
- У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.
- У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:
Олег БІСІКАЛО, зав. кафедри АІІТ _____
(прізвище, ініціали, посада) (підпис)
Любов ЯНЧУК, секретар кафедри АІІТ _____
(прізвище, ініціали, посада) (підпис)
Особа, відповідальна за перевірку _____ Костянтин ОВЧИННИКОВ
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)
Керівник _____ Ольга СОФИНА
(підпис) (прізвище, ініціали, посада)
Здобувач _____ Олександр ТАНАСКОВ
(підпис) (прізвище, ініціали)