ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Система керування користувацькими peer-підключеннями WireGuard із графічним інтерфейсом на Python (PyQt6)»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Богдан ЛАВРЕНЮК
(підпис)

Виконав: здобувач(ка) вищої освіти групи ТЦР-42
Богдан ЛАВРЕНЮК

Керівник: Олексій АНАНЧЕНКО
кандидат техн. наук

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Технологій цифрового розвитку
_____ Вікторія ЖЕБКА
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Лавренюку Богдану Юрійовичу

1. Тема кваліфікаційної роботи: «Система керування користувацькими peer-підключеннями WireGuard із графічним інтерфейсом на Python (PyQt6) »
керівник кваліфікаційної роботи кандидат техн.наук, доцент Олексій АНАНЧЕНКО,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «_20__» лютого 2026 року № _51___.

2. Строк подання кваліфікаційної роботи «___» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи обробки природньої мови, опис методів вилучення інформації з тексту та формування тематичних словників, технічна документація з описом бібліотек для обробки природньої мови.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд та аналіз існуючих методів та технологій обробки природньої мови Аналіз предметної області та існуючих технологічних підходів до побудови desktop-застосунків для керування мережевими конфігураціями.

2. Проєктування архітектури та інтерфейсу системи керування користувацькими WireGuard-профілями.
3. Реалізація програмного засобу керування WireGuard-профілями на базі Python і Qt та опис принципів його функціонування.
4. Тестування та оцінка працездатності розробленого застосунку.

5. Перелік графічного матеріалу: *презентація*
   1. Аналіз аналогів.
   2. Вимоги до програмного забезпечення.
   3. Програмні засоби реалізації.
   4. Діаграма варіантів використання.
   5. Алгоритм роботи застосунку.
   6. Діаграма класів.
   7. Екранні форми.
   8. Апробація результатів дослідження

6. Дата видачі завдання «__20__» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	20.03.2026-23.03.2026	
2	Аналіз та дослідження існуючих аналогів	24.03-13.03.2026	
3	Огляд існуючих алгоритмів обробки тексту та методів і технологій побудови тематичного словника	14.03–25.03.2026	
4	Проектування застосунку для автоматизованої побудови тематичного словника навчальної дисципліни	26.03–15.04.2026	
5	Програмна реалізація застосунку	16.04–25.04.2026	
6	Тестування застосунку	26.04–28.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	29.04-04.05.2026	
8	Розробка демонстраційних матеріалів	05.05-10.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувач(ка) вищої освіти _____ Богдан ЛАВРЕНЮК
(підпис)

Керівник
кваліфікаційної роботи _____ Олексій АНАНЧЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 56 стор., 1 табл., 31 рис., 21 джерел.

Мета роботи – підвищення зручності та ефективності керування користувацькими peer-підключеннями WireGuard.

Об'єкт дослідження – процес локального керування конфігураційними профілями WireGuard у desktop-середовищі.

Предмет дослідження – програмні засоби, архітектурні рішення та методи реалізації desktop-застосунку для керування WireGuard-профілями на основі Python, Qt і локального сховища даних.

Короткий зміст роботи: У кваліфікаційній роботі розв'язано задачу розроблення desktop-застосунку для локального керування користувацькими peer-підключеннями WireGuard. Виконано аналіз існуючих технологічних підходів до реалізації систем цього класу та обґрунтовано доцільність використання мови програмування Python і фреймворку Qt. У межах роботи спроєктовано програмну архітектуру застосунку, графічний інтерфейс користувача та структуру локальної бази даних. Реалізовано модулі роботи з профілями, що охоплюють операції створення, імпорту, модифікації, видалення та візуалізації конфігураційних даних, а також засоби їх попередньої валідації. Результати роботи підтверджено апробацією базових функціональних сценаріїв і перевіркою працездатності розробленого програмного засобу.

Сфера використання - локальні системи конфігурування та супроводу мережевих профілів у задачах організації захищених підключень.

КЛЮЧОВІ СЛОВА: WIREGUARD, VPN, DESKTOP-ЗАСТОСУНОК, PYTHON, QT, PYSIDE6, SQLITE, ГРАФІЧНИЙІНТЕРФЕЙС, КОНФІГУРАЦІЙНИЙ ПРОФІЛЬ, ВАЛІДАЦІЯ ДАНИХ.

ЗМІСТ

ВСТУП

Актуальність теми У сучасних умовах цифровізації інформаційних процесів та зростання потреби в захищеному мережевому доступі все більшого поширення набувають VPN-технології, зокрема WireGuard, який вирізняється відносною простотою конфігурування, високою продуктивністю та широким застосуванням як у персональному, так і в корпоративному середовищі. Разом із тим практичне використання WireGuard для кінцевого користувача або адміністратора часто супроводжується необхідністю ручної роботи з конфігураційними файлами, збереженням декількох профілів підключення, контролем їх актуальності та виправленням помилок, що виникають під час редагування параметрів.

Існуючі засоби роботи з WireGuard-конфігураціями здебільшого орієнтовані або на серверне адміністрування, або на використання готових конфігураційних файлів без достатньо зручного локального графічного інтерфейсу для централізованого керування профілями. Унаслідок цього виникає потреба у створенні desktop-застосунку, який забезпечував би зручне локальне керування конфігураційними профілями, дозволяв би створювати, імпортувати, редагувати, зберігати та переглядати параметри підключення у впорядкованому вигляді.

Розроблення такого програмного засобу є актуальним, оскільки воно сприяє підвищенню зручності роботи з мережевими конфігураціями, зменшенню кількості помилок під час ручного введення даних, покращенню організації профілів підключення та створенню основи для подальшого розвитку локальних систем керування VPN-конфігураціями.

Об'єктом дослідження є процес локального керування конфігураційними профілями WireGuard у desktop-середовищі.

Предметом дослідження є програмні засоби, архітектурні рішення та методи реалізації desktop-застосунку для керування WireGuard-профілями на основі Python, Qt і локального сховища даних.

Метою роботи є підвищення зручності та ефективності керування користувацькими peer-підключеннями WireGuard шляхом розроблення desktop-застосунку з графічним інтерфейсом для створення, імпорту, редагування, збереження та перегляду конфігураційних профілів.

Досягнення поставленої мети передбачає реалізацію таких завдань:

1. Провести аналіз існуючих підходів до побудови desktop-застосунків для керування мережевими конфігураціями та обґрунтувати вибір засобів реалізації.
2. Дослідити можливості використання Python і Qt для створення локального застосунку з графічним інтерфейсом.
3. Спроєктувати архітектуру програмного засобу, структуру графічного інтерфейсу та модель локального зберігання даних.
4. Реалізувати функціональні модулі для створення, імпорту, редагування, видалення та відображення WireGuard-профілів.
5. Забезпечити перевірку коректності введених конфігураційних даних перед їх збереженням.
6. Реалізувати локальне збереження профілів у базі даних та механізми їх подальшого завантаження і відображення.
7. Провести тестування працездатності основних функціональних сценаріїв застосунку.

*Практичне значення результатів.*У межах роботи розроблено desktop-застосунок для локального керування конфігураційними профілями WireGuard. У програмному засобі реалізовано ініціалізацію локальної бази даних, створення, імпорт, редагування, видалення та відображення профілів, а також перевірку коректності введених даних перед збереженням. Застосунок побудовано за модульним принципом із розподілом функцій між рівнями графічного інтерфейсу, прикладної логіки та доступу до даних; для зберігання профілів використано локальну SQLite-базу даних, а взаємодію з користувачем реалізовано через графічний інтерфейс на основі Qt/Python

Практична значущість результатів полягає у створенні зручного інструмента для локальної організації та супроводу WireGuard-конфігурацій, який може бути використаний у навчальних, прикладних і адміністративних сценаріях. Розроблений застосунок дозволяє впорядковувати профілі підключення, зменшує залежність від ручного редагування конфігураційних файлів і створює основу для подальшого розширення функціональності, зокрема в напрямах ведення історії сесій, додавання статистичних модулів та реалізації механізмів connect/disconnect. Отримані результати можуть бути використані як основа для подальшої розробки локальних систем керування VPN-профілями та як приклад реалізації desktop-застосунку конфігураційного типу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Характеристика предметної галузі

Предметна галузь роботи пов’язана з керуванням конфігураційними профілями захищених мережевих підключень у локальному середовищі. У сучасних умовах використання VPN-технологій важливого значення набуває не лише забезпечення безпечного з’єднання, а й зручність роботи з параметрами підключення, їх збереженням, редагуванням і повторним використанням.

Технологія WireGuard є одним із сучасних засобів побудови віртуальних приватних мереж, однак практичне використання її конфігурацій часто передбачає роботу з текстовими файлами, що ускладнює супровід декількох профілів і підвищує ризик помилок під час ручного редагування. У зв’язку з цим виникає потреба в застосуванні програмних засобів, які дозволяють організувати централізовану локальну роботу з профілями підключення через графічний інтерфейс.

Предметна галузь охоплює поєднання мережевих технологій, засобів розробки desktop-застосунків, механізмів валідації конфігураційних даних та локального зберігання профілів. У межах цієї галузі актуальними є рішення, що забезпечують створення, імпорт, редагування, збереження й відображення конфігураційних профілів у зручній для користувача формі. Саме тому розроблення desktop-застосунку для локального керування WireGuard-профілями є актуальним і доцільним напрямом дослідження.

1.2 Аналіз існуючих аналогів систем керування WireGuard-профілями

У сфері організації захищених мережевих підключень спостерігається зростання інтересу до програмних засобів, що спрощують роботу з конфігураціями WireGuard та надають зручні інтерфейси для керування

профілями підключення. У цьому підрозділі проведено аналіз існуючих рішень, які частково або повністю реалізують функціональність, близьку до тієї, що розробляється в межах даної кваліфікаційної роботи. Зокрема, розглянуто системи wg-easy, wg-access-server та WGDashboard. Платформа wg-easy позиціонується як рішення формату WireGuard + Web UI з підтримкою створення, редагування, видалення клієнтів, QR-кодів, статистики підключень і додаткових адміністративних можливостей . Система wg-access-server є сервером доступу з web-інтерфейсом, підтримкою автентифікації користувачів, one-click реєстрації пристроїв і централізованого керування VPN-доступом . Рішення WGDashboard орієнтоване на спрощення перегляду та керування конфігураціями WireGuard, а також моніторинг стану підключень через web-інтерфейс . Аналіз зазначених аналогів дає змогу визначити їх сильні сторони, обмеження та обґрунтувати доцільність розроблення локального desktop-застосунку для керування WireGuard-профілями.

1.2.1 Аналіз програмного рішення wg-easy

Система wg-easy є open-source програмним рішенням для розгортання WireGuard VPN із вбудованим web-інтерфейсом адміністрування. В офіційній документації та репозиторії проєкт позиціонується як засіб, що поєднує функції WireGuard і Web UI в одному продукті та орієнтований на спрощення встановлення й подальшого керування VPN-конфігураціями. Проєкт має окрему версійну документацію та поширюється під ліцензією AGPL-3.0-only.

З архітектурної точки зору wg-easy належить до серверно-орієнтованих web-рішень. Його типове впровадження передбачає наявність керованого хоста, доменного імені або публічної IP-адреси, а також контейнерного середовища, зокрема Docker Compose або Podman. Офіційна документація прямо вказує Docker Compose як базовий спосіб розгортання, а для безпечного доступу до web-інтерфейсу рекомендує використовувати reverse proxy. Таким чином, wg-easy орієнтований не на локальний desktop-сценарій, а на централізоване серверне адміністрування WireGuard через браузер.

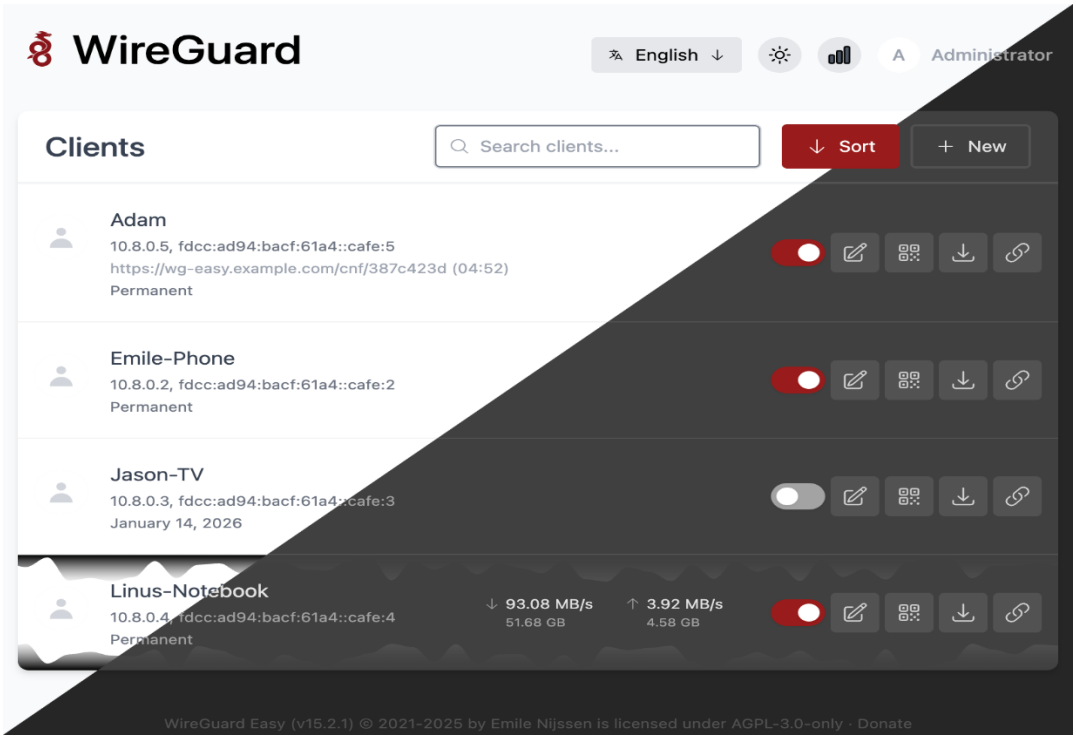


Рис.1.1 Веб-інтерфейс головної сторінки wg-easy

Функціональні можливості wg-easy охоплюють не лише базове керування конфігураціями, а й ширший набір адміністративних операцій. Згідно з офіційним описом, система підтримує створення, редагування, видалення, увімкнення та вимкнення клієнтів, відображення QR-кодів, завантаження конфігураційних файлів клієнтів, статистику підключених клієнтів, графіки передавання трафіку Tx/Rx, мультимовність, одноразові посилання, термін дії клієнтів, Prometheus metrics, підтримку IPv6, CIDR і двофакторну автентифікацію. За своїм функціональним наповненням це рішення є повноцінною системою web-керування WireGuard-сервером.

До основних переваг wg-easy доцільно віднести високий рівень функціональної готовності, зручність web-адміністрування, наявність засобів візуалізації стану клієнтів і трафіку, а також підтримку сучасних засобів інтеграції та моніторингу. Важливою перевагою є й те, що система зводить значну частину ручної роботи з WireGuard-конфігураціями до керованих операцій через браузер, що суттєво покращує користувацький досвід у порівнянні з ручним редагуванням конфігураційних файлів. Крім того, офіційна документація містить окремі інструкції для базового встановлення,

використання Podman, Caddy, Traefik та інших суміжних сценаріїв, що свідчить про практичну зрілість рішення.

Водночас wg-easy має й суттєві обмеження з погляду задачі, що розглядається в межах цієї роботи. По-перше, система вимагає серверної інфраструктури та контейнерного середовища, що підвищує поріг впровадження в порівнянні з локальним desktop-застосунком. По-друге, її архітектура орієнтована насамперед на адміністрування WireGuard-сервера і клієнтів у web-середовищі, а не на персональне локальне керування профілями в межах одного робочого місця. По-третє, використання AGPL-ліцензії є більш суворим з юридичного погляду, ніж застосування пермісивних ліцензій, наприклад MIT або Apache 2.0. Отже, wg-easy є потужним серверним аналогом, однак за своєю архітектурною природою належить до іншого класу систем, ніж локальний desktop-інструмент конфігураційного типу.

1.2.2 Аналіз програмного рішення wg-access-server

Програмне рішення wg-access-server є open-source системою, призначеною для побудови WireGuard VPN-сервера з web-інтерфейсом керування пристроями та підключеннями. В офіційному описі проєкт позиціонується як all-in-one WireGuard VPN solution with a web UI for connecting devices, тобто як комплексний сервер доступу, що поєднує функції VPN, web-керування та підтримки клієнтських підключень у межах єдиного програмного продукту. У репозиторії проєкт поширюється під MIT license, що є пермісивною ліцензією та спрощує його використання й модифікацію в порівнянні з більш суворими open-source ліцензіями.

З архітектурної точки зору wg-access-server належить до серверно-орієнтованих web-рішень, орієнтованих на централізоване керування доступом через браузер. Офіційна документація підкреслює, що система реалізована як single binary, який забезпечує WireGuard VPN server і device management web UI. Водночас проєкт підтримує кілька моделей розгортання: через Docker, Docker Compose та Helm/Kubernetes, що свідчить про його орієнтацію на

інфраструктурні сценарії використання і можливість інтеграції у більш складні серверні середовища. Зокрема, Helm-документація прямо вказує на підтримку Kubernetes-розгортання, а також зазначає, що за замовчуванням пристрої не зберігаються між перезапусками pod без спеціальної конфігурації.

Функціонально wg-access-server реалізує ширший набір можливостей, ніж простий менеджер конфігурацій. В офіційному описі вказано, що продукт підтримує user authentication, 1-click device registration, сумісність із Mac, Linux, Windows, iOS і Android, роботу з QR-кодами, а також різні network isolation modes для кращого контролю доступу. Отже, система поєднує засоби керування WireGuard-конфігураціями з механізмами централізованої автентифікації та реєстрації пристроїв. Такий функціональний підхід дозволяє розглядати wg-access-server не просто як інтерфейс до конфігурацій, а як сервер доступу з елементами політики безпеки та адміністрування користувачів.

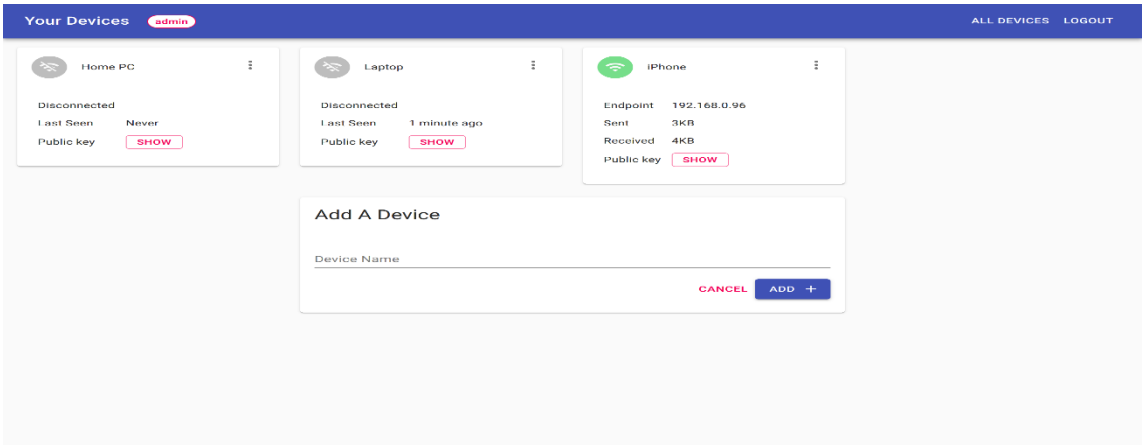


Рис 1.2 Веб-інтерфейс головної сторінки wg-access-server

Окремої уваги заслуговує технологічна організація рішення. В офіційній документації зазначено, що система складається з Golang Server і React App, а клієнт взаємодіє із сервером через gRPC-Web. Це свідчить про використання сучасного розподіленого web-підходу, де backend відповідає за серверну логіку та роботу з мережею, а frontend - за інтерфейс керування в браузері. З академічної точки зору це вказує на більш складну багатошарову архітектуру в порівнянні з локальними desktop-рішеннями конфігураційного типу.

До переваг wg-access-server доцільно віднести його комплексність, підтримку автентифікації, централізовану модель керування пристроями,

наявність web-інтерфейсу та підтримку різних способів розгортання. Суттєвою сильною стороною є орієнтація на багатокористувацький сценарій, коли доступ до VPN-сервера організовується через єдину точку керування, а підключення клієнтів може здійснюватися за спрощеним сценарієм із використанням QR-кодів і web-інтерфейсу. Також важливо, що проєкт підтримує середовища Docker, Compose і Kubernetes, тобто придатний до інтеграції в сучасну DevOps-інфраструктуру.

Рис 1.3 вікно аутифікації wg-access-server

Разом із тим wg-access-server має низку обмежень у контексті задачі, що розглядається в даній роботі. По-перше, це рішення є серверною системою, а не локальним desktop-застосунком, тому його впровадження потребує інфраструктурної підготовки, зокрема наявності серверного середовища, контейнерної платформи або Kubernetes-кластера. По-друге, його функціональність виходить далеко за межі задачі персонального локального керування профілями, оскільки охоплює автентифікацію користувачів, централізовану реєстрацію пристроїв і політики мережевої ізоляції. По-третє, багатокомпонентна архітектура на базі Go, React і gRPC-Web є значно складнішою в реалізації та супроводі, ніж архітектура локального desktop-застосунку, побудованого на Python, Qt і SQLite. Таким чином, з позиції нашої

роботи wg-access-server виступає функціонально потужним, але архітектурно іншим класом систем.

1.2.3 Аналіз програмного рішення WGDashboard

Програмне рішення WGDashboard є open-source системою для конфігурування, моніторингу та керування WireGuard VPN через web-інтерфейс. На офіційному сайті проєкт описується як simple and easy-to-use dashboard to manage your WireGuard VPN server(s), а GitHub-репозиторій характеризує його як simple dashboard for WireGuard VPN written in Python & Vue.js. Таким чином, WGDashboard належить до класу серверно-орієнтованих web-рішень, призначених для спрощення адміністрування WireGuard-конфігурацій без прямої роботи з командним рядком.

З погляду технологічної реалізації WGDashboard побудовано на поєднанні Python і Vue.js, що прямо зазначено в офіційних матеріалах проєкту, а поширюється він під ліцензією Apache License 2.0. Така технологічна основа свідчить про використання web-архітектури з серверною частиною, що відповідає за роботу з конфігураціями та службовими операціями WireGuard, і клієнтською частиною, яка забезпечує взаємодію з адміністратором через браузер. Важливо також, що проєкт має окрему документацію, офіційний сайт, демо-середовище та активний цикл релізів, що свідчить про достатньо зрілий рівень підтримки.

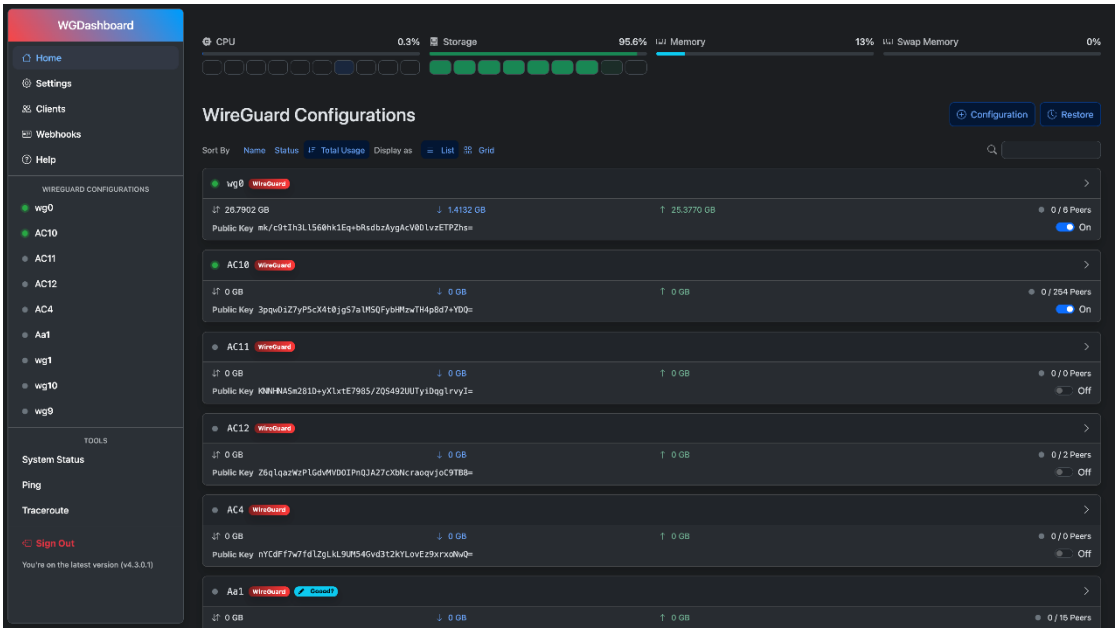


Рис 1.4 Домашня сторінка в WGDashboard

Архітектурно WGDashboard орієнтований на серверну експлуатацію. Документація вказує, що система може працювати на bare-metal Linux, у Docker-контейнері, а також інтегруватися в інші середовища, зокрема Proxmox. У документації з Docker Image прямо зазначено підтримку контейнерного розгортання, використання окремих томів для /etc/wireguard і /data, відкриття окремих портів для web-інтерфейсу й WireGuard, а також автоматичне створення конфігурацій під час першого запуску. Отже, WGDashboard є насамперед інструментом адміністрування WireGuard на стороні сервера, а не локальним desktop-застосунком користувача.

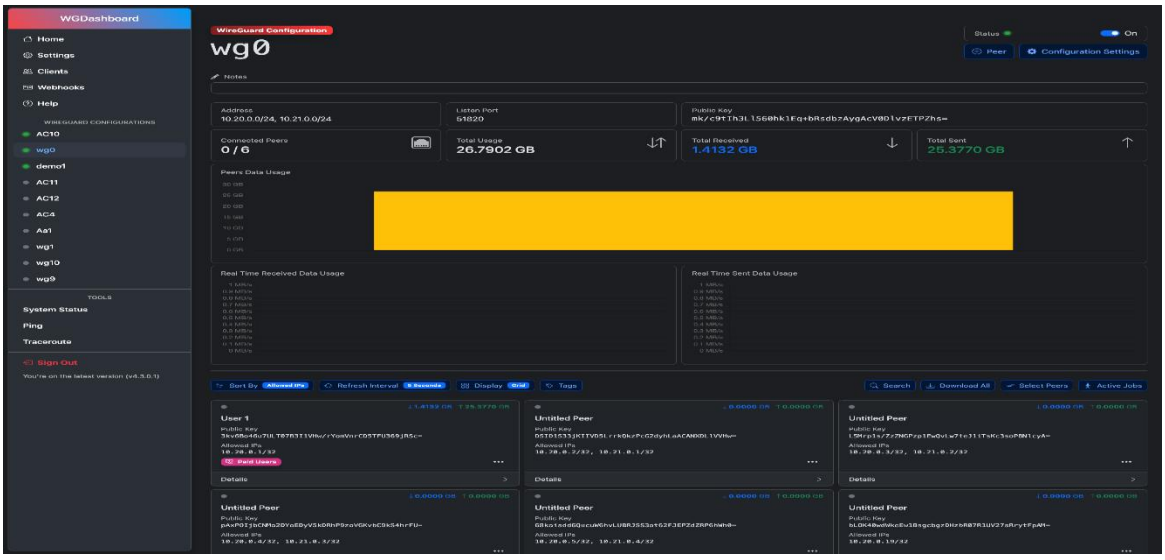


Рис 1.5 Сторінка з дешбордами та станом тунелів

За функціональними можливостями WGDashboard забезпечує достатньо широкий набір засобів керування. Згідно з офіційним переліком можливостей, система автоматично знаходить наявні WireGuard-конфігурації, надає захищений доступ до dashboard через credentials і TOTP, дозволяє додавати peer-и поодинці або пакетно, редагувати й видаляти їх, обмежувати доступ, генерувати QR-коди й .conf-файли, ділитися ними через публічне посилання, планувати задачі видалення чи обмеження peer-ів, призначати peer-и клієнтам, переглядати їхній статус у реальному часі, а також виконувати ping і traceroute. Таким чином, WGDashboard поєднує функції конфігуратора, інструмента моніторингу та адміністративної панелі керування.

До основних переваг WGDashboard доцільно віднести, по-перше, його зручність у візуальному керуванні WireGuard-конфігураціями. Сам проєкт був створений саме як відповідь на незручність ручного моніторингу через wg show, що прямо зазначено в GitHub-README. По-друге, суттєвою перевагою є підтримка реального часу спостереження за статусом peer-ів, а також наявність діагностичних інструментів, зокрема ping і traceroute. По-третє, продукт має розвинені можливості роботи з peer-ами: генерацію QR-кодів, .conf-файлів, посилань, обмеження доступу та планування задач, що робить його практично корисним для адміністрування реальної WireGuard-інфраструктури. Окремою перевагою є підтримка багатомовності та багатофакторної автентифікації.

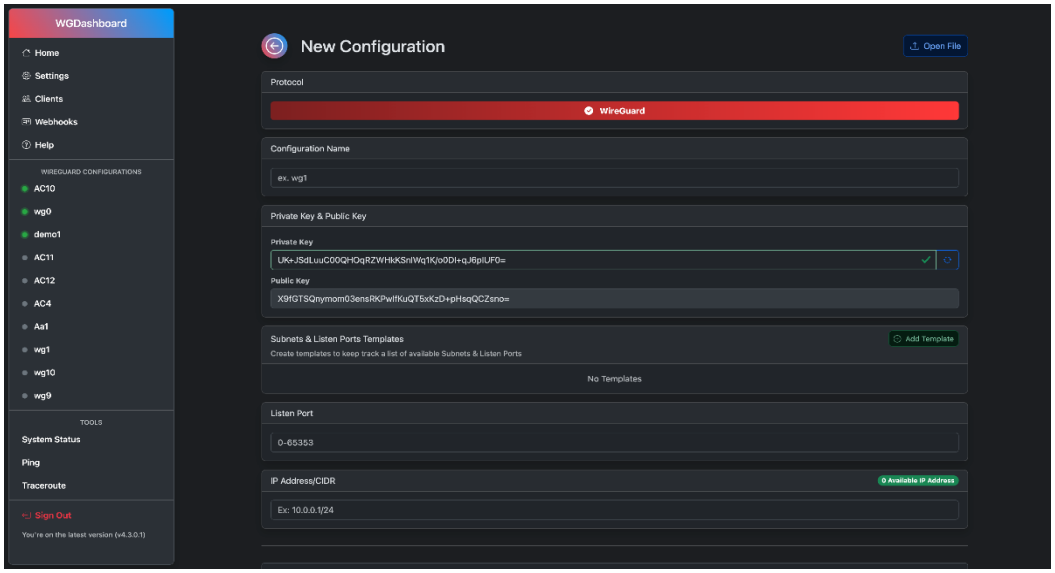


Рис 1.6 сторітка з додаванням нового peer-підлючення

Разом із тим WGDashboard має й низку обмежень у контексті задачі, що розв’язується в межах даної кваліфікаційної роботи. Насамперед це рішення належить до web/server-класу систем, а отже потребує серверного середовища, відповідного мережевого налаштування та постійного супроводу з боку адміністратора. На відміну від локального desktop-підходу, WGDashboard орієнтований на управління існуючими конфігураціями WireGuard на сервері, а не на персональне локальне зберігання й редагування профілів у межах одного робочого місця. Крім того, функціональна насиченість системи, зокрема наявність багатофакторної автентифікації, тестових інструментів, контейнерного розгортання та планувальника задач, робить її архітектурно складнішою порівняно з локальним desktop-застосунком конфігураційного типу. Додатково в офіційному README є попередження про необхідність своєчасного оновлення версій, розміщених у публічному інтернеті, що підкреслює важливість безпекового супроводу для такого класу web-рішень.

Проведений аналіз програмних рішень wg-easy, wg-access-server та WGDashboard показав, що всі розглянуті системи є функціонально розвиненими засобами керування WireGuard, орієнтованими переважно на серверне розгортання та web-адміністрування. Їх спільними рисами є наявність інтерфейсу керування, засобів роботи з конфігураціями та peer-підключеннями, а також орієнтація на централізоване адміністрування WireGuard-інфраструктури через браузер. Водночас ці рішення потребують серверного середовища, додаткових мережевих налаштувань і, як правило, контейнерної або іншої інфраструктурної підтримки, що зумовлює вищу складність впровадження та експлуатації.

На відміну від розглянутих аналогів, розроблюваний у межах даної кваліфікаційної роботи програмний засіб має іншу концепцію побудови: він реалізується як локальний desktop-застосунок, орієнтований на персональне керування конфігураційними профілями WireGuard без необхідності розгортання окремого сервера або web-інфраструктури. Його архітектура побудована за модульним принципом із розподілом на рівні графічного

інтерфейсу, прикладної логіки та доступу до даних, а поточна реалізація вже підтримує сценарії створення, імпорту, редагування, видалення, збереження та валідації профілів у локальному сховищі. Таким чином, запропонований підхід не дублює існуючі server/web-рішення, а займає окрему нішу локальних систем керування WireGuard-профілями, що робить його доцільним і практично обґрунтованим у межах поставленої задачі.

1.3 Постановка завдання на розробку системи

На основі проведеного аналізу предметної галузі, а також результатів дослідження існуючих аналогів систем керування WireGuard-конфігураціями, встановлено, що наявні програмні рішення орієнтовані переважно на серверне розгортання та web-адміністрування WireGuard-інфраструктури. Такі системи забезпечують розвинені засоби централізованого керування, однак потребують окремого серверного середовища, додаткового мережевого налаштування та інфраструктурної підтримки. Водночас для задач локального персонального керування конфігураційними профілями доцільним є інший підхід - створення desktop-застосунку, що не потребує окремого серверного розгортання та забезпечує зручну роботу з профілями безпосередньо у локальному середовищі користувача.

У межах кваліфікаційної роботи ставиться завдання розробити desktop-застосунок для локального керування користувацькими peer-підключеннями WireGuard, який забезпечуватиме базові сценарії роботи з конфігураційними профілями через графічний інтерфейс. Система повинна бути орієнтована на локальне використання, використовувати вбудоване сховище даних і не вимагати розгортання окремого web-сервера або серверної інфраструктури.

Розроблювана система повинна забезпечувати:

- ініціалізацію локальної бази даних під час запуску застосунку;
- зчитування та відображення збережених профілів у графічному інтерфейсі;

- створення нового профілю шляхом ручного введення параметрів;
- імпорт конфігураційного профілю з файлу формату .conf;
- редагування та видалення існуючих профілів;
- перевірку коректності та повноти введених конфігураційних даних перед збереженням;
- збереження профілів у локальному сховищі та автоматичне оновлення інтерфейсу після внесення змін.

1.4 Визначення проблем предметної галузі

Предметна галузь, пов’язана з керуванням конфігураційними профілями WireGuard, характеризується низкою практичних труднощів. Основною з них є необхідність ручної роботи з конфігураційними файлами, що ускладнює створення, редагування, збереження та повторне використання профілів підключення. За наявності декількох профілів зростає складність їх упорядкування, контролю актуальності та перевірки правильності введених параметрів.

Додатковою проблемою є те, що більшість існуючих рішень для керування WireGuard орієнтовані переважно на серверне розгортання та web-адміністрування, а не на локальне персональне використання. Унаслідок цього користувачі, які потребують простого засобу для роботи з профілями в межах одного комп’ютера, не мають достатньо зручного desktop-інструмента. Це зумовлює потребу у створенні локального застосунку, який забезпечував би централізоване зберігання профілів, перевірку конфігураційних даних і керування ними через графічний інтерфейс.

1.5 Визначення вимог до застосунку

Вимоги до розроблюваного застосунку сформовано на основі аналізу предметної галузі, існуючих аналогів та поставленого завдання на розробку

системи. З урахуванням цільового призначення програмного засобу доцільно виокремити функціональні та нефункціональні вимоги.

Функціональні вимоги

До основних функціональних вимог застосунку належать такі:

1. Ініціалізація локального сховища даних. Застосунок повинен автоматично створювати та ініціалізувати локальну базу даних під час запуску.

2. Відображення профілів у графічному інтерфейсі. Застосунок повинен завантажувати збережені WireGuard-профілі та відображати їх у вигляді списку або карток.

3. Створення та імпорт профілю. Застосунок повинен забезпечувати створення нового профілю шляхом ручного введення параметрів, а також імпорт профілю з конфігураційного файлу .conf.

4. Редагування та видалення профілю. Користувач повинен мати можливість змінювати параметри існуючого профілю та видаляти непотрібні записи зі сховища.

5. Перевірка коректності введених даних. Застосунок повинен виконувати валідацію введених параметрів профілю перед збереженням і повідомляти користувача про виявлені помилки.

6. Збереження змін та оновлення інтерфейсу. Після створення, імпорту, редагування або видалення профілю зміни повинні зберігатися в локальній базі даних, а відображення профілів у графічному інтерфейсі повинно автоматично оновлюватися.

2 АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ ТА СТРУКТУРИ ПРОГРАМНОЇ СИСТЕМИ

2.1 Середовище розробки

Для розроблення програмного засобу було використано середовище Visual Studio Code, яке забезпечує зручні засоби роботи з вихідним кодом, навігації проєктом, редагування файлів, запуску застосунку та налагодження програмної логіки. Вибір цього середовища зумовлений його універсальністю, підтримкою мови програмування Python, можливістю інтеграції з розширеннями для роботи з Qt-проєктами, а також зручністю організації структури desktop-застосунку.

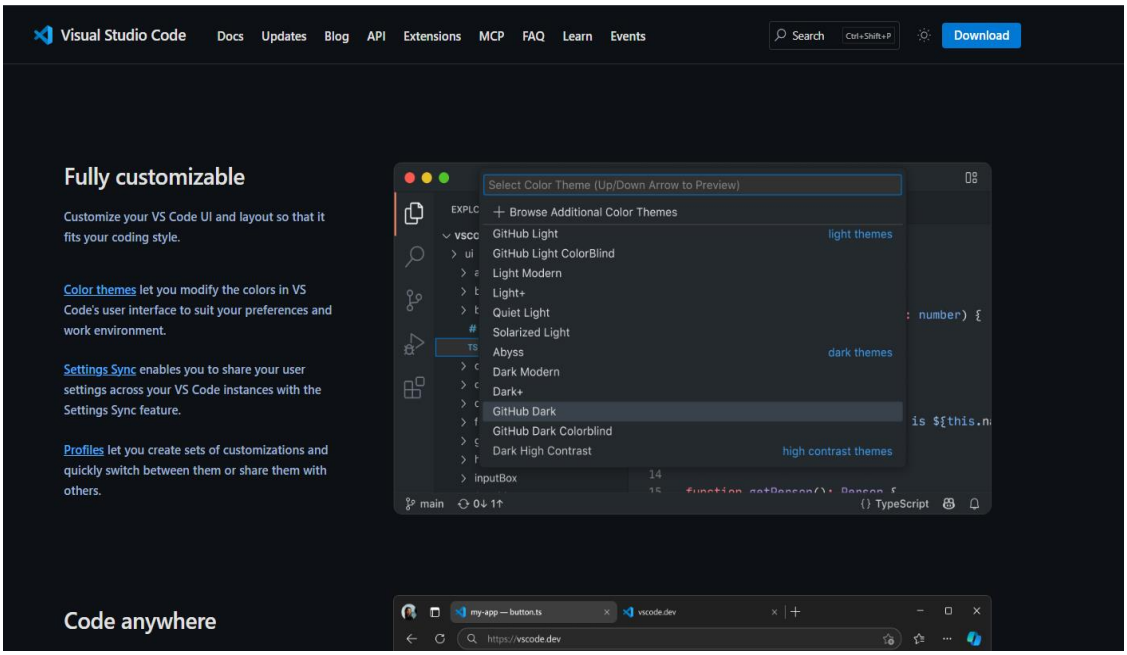


Рис 2.1 Інтерфейс середовища Visual Studio code

У процесі розробки Visual Studio Code використовувалося для створення та редагування програмних модулів, перевірки логіки роботи застосунку, запуску окремих компонентів системи та тестування базових сценаріїв взаємодії з користувачем. Середовище дозволило ефективно організувати роботу з багатофайловим проєктом, у межах якого були реалізовані інтерфейсні компоненти, модулі прикладної логіки та засоби доступу до даних.

Як основну мову програмування було використано Python, а для побудови графічного інтерфейсу - PySide6 як Python-прив’язку до Qt. Для візуального

проєктування окремих форм інтерфейсу застосовувався Qt Designer, що дало змогу відокремити структуру інтерфейсу від прикладної логіки програми. Для локального зберігання даних було використано SQLite, доступ до якої реалізовано засобами стандартного модуля sqlite3.

2.2 Вибір мови програмування та фреймворку

Вибір мови програмування та фреймворку для реалізації розроблюваного застосунку здійснювався з урахуванням характеру поставленої задачі. Оскільки система розробляється як локальний desktop-застосунок конфігураційного типу, для неї визначальними є такі критерії: наявність зручних засобів побудови графічного інтерфейсу, підтримка подієво-орієнтованої моделі взаємодії, можливість швидкої реалізації прикладної логіки, інтеграція з локальним сховищем даних, а також доцільність використання в межах навчального та прикладного проєкту.

Як основну мову програмування було обрано Python. Такий вибір зумовлений тим, що Python є високорівневою мовою загального призначення, яка забезпечує відносно невисоку трудомісткість розроблення, добру читабельність коду та зручність реалізації прикладної логіки. Для задач, пов’язаних із роботою з конфігураційними даними, валідацією введення, взаємодією з файлами та локальними базами даних, Python є достатньо виразним і практичним інструментом. Крім того, використання Python добре узгоджується з модульною побудовою програмного засобу, де окремо реалізуються рівні інтерфейсу, прикладної логіки та доступу до даних. У поточній реалізації це вже проявляється через поділ на MainWindow, ProfileDialog, service layer, repository layer та SQLite-сховище.

Для побудови графічного інтерфейсу було обрано Qt for Python (PySide6). Офіційна документація визначає Qt for Python як bindings-проєкт, що надає доступ до повного Qt framework із Python. Це означає, що розробник отримує у своє розпорядження зрілий набір засобів для побудови нативного графічного

інтерфейсу, включно з віджетами, подієвою моделлю, сигналами й слотами, діалоговими вікнами та інструментами проєктування форм. Саме такі можливості є ключовими для системи, у якій основна взаємодія з користувачем відбувається через введення, перегляд, зміну та перевірку конфігураційних параметрів.

Вибір PySide6 замість інших GUI-підходів є обґрунтованим також з погляду архітектурної відповідності задачі. На відміну від Electron, який будує desktop-застосунки на базі JavaScript, HTML, CSS, Node.js і Chromium, тобто фактично реалізує desktop-оболонку навколо web-інтерфейсу, Qt/PySide6 дозволяє створювати нативний GUI без browser-runtime, що є більш природним для прикладного конфігураційного застосунку. На відміну від Tauri, який поєднує web-frontend із Rust-backend і системним WebView, PySide6 не вимагає побудови двошарової web/Rust-моделі та забезпечує пряміший шлях до реалізації форм, діалогів і прикладної логіки в межах єдиного технологічного середовища. Порівняно з .NET MAUI, що позиціонується як multi-platform framework для mobile і desktop apps на C# та XAML, підхід Python + Qt краще відповідає задачі локального desktop-застосунку, оскільки не орієнтує систему на ширший mobile/desktop стек і дозволяє зосередитися саме на прикладному GUI конфігураційного типу.

Додатковою перевагою PySide6 є використання відкритої ліцензії LGPL v3, що робить його придатним для реалізації навчальних і прикладних проєктів за умови дотримання умов розповсюдження. Для порівняння, Electron і .NET MAUI поширюються під ліцензією MIT, а Tauri - під MIT або Apache 2.0, однак саме з погляду функціональної відповідності поставленій задачі вирішальним є не лише ліцензійний аспект, а насамперед архітектурна та інтерфейсна придатність фреймворку.

3 ПРОЄКТУВАННЯ СИСТЕМИ

3.1 Архітектура застосунку

Архітектура програмного застосунку визначає його структурну організацію, принципи взаємодії між компонентами, розподіл відповідальності між окремими модулями та загальну логіку обробки даних. Для системи локального керування WireGuard-профілями архітектурне рішення повинно забезпечувати не лише зручну роботу з графічним інтерфейсом і локальним сховищем конфігурацій, а й підтримку керування тунелями, перевірки стану підключення, ведення історії сесій та формування зведених показників для інформаційної панелі.

У межах даної роботи реалізовано багаторівневу модульну desktop - архітектуру, орієнтовану на розподіл функцій між інтерфейсом користувача, прикладною логікою, рівнем доступу до даних і локальним сховищем. Такий підхід забезпечує ізоляцію відповідальностей між компонентами, полегшує супровід програмного засобу та створює основу для подальшого розвитку системи. На відміну від монолітного підходу, де логіка інтерфейсу, перевірки даних і SQL-операції зосереджуються безпосередньо в коді вікон, у поточній реалізації ці функції розмежовані між окремими модулями та рівнями архітектури

У загальному вигляді застосунок складається з таких основних рівнів: Presentation layer- рівень графічного інтерфейсу користувача; Service layer - рівень прикладної логіки, перевірки даних, керування тунелями, сесіями та агрегованими показниками; Repository layer - рівень доступу до локальної бази даних; Data layer - локальна база даних SQLite;

Рівень представлення реалізує взаємодію користувача із системою. До нього належать головне вікно застосунку та діалогова форма профілю. Центральним елементом цього рівня є MainWindow, яке під час ініціалізації викликає створення схеми бази даних, завантажує графічний інтерфейс із .ui-файлу, прив'язує віджети, налаштовує сигнали й обробники подій, а також забезпечує навігацію між основними сторінками Dashboard, Profiles і History.

На рівні представлення користувач отримує доступ до основних функціональних сценаріїв системи: створення нового профілю, імпорту конфігурації, редагування і видалення записів, пошуку профілів, підключення і відключення тунелю, перегляду історії сесій, очищення журналу, а також перегляду агрегованих показників інформаційної панелі. Крім того, саме на цьому рівні реалізовано перемикання теми оформлення, локалізації інтерфейсу та автоматичне оновлення окремих візуальних елементів за допомогою QTimer

Окремим елементом рівня представлення є ProfileDialog, який використовується як універсальна форма для створення нового профілю, імпорту конфігурації та редагування вже існуючого запису. У межах діалогу реалізовано режими NEW, IMPORT і EDIT, що дозволяє використовувати одну форму для кількох близьких сценаріїв взаємодії. Діалог підтримує імпорт .conf-файлу, його парсинг через configparser, автоматичне заповнення полів форми, а також первинну інтерфейсну валідацію введених значень

Особливістю архітектури рівня представлення є те, що він не працює з локальною базою даних напряму. Інтерфейс відповідає за збір введених даних, відображення результатів і виклик відповідних сервісів, тоді як бізнес-логіка, збереження та робота із системними компонентами винесені до окремих модулів. Це дозволяє зменшити зв'язність між UI і прикладним кодом та покращує підтримуваність застосунку.

Рівень прикладної логіки представлений сукупністю сервісних модулів: profile_service, wireguard_service, session_service, history_service та dashboard_service. Саме ці модулі реалізують основні правила функціонування

системи, координують взаємодію між графічним інтерфейсом, локальною базою даних і системними утилітами WireGuard for Windows.

Модуль profile_service відповідає за бізнес-валідацію, нормалізацію та підготовку даних профілю перед збереженням у базі. У ньому реалізовано перевірку коректності назви профілю, форматів ключів, адреси інтерфейсу, кінцевої точки та числових параметрів конфігурації. Окрім цього, сервіс перетворює низькорівневі помилки обмежень бази даних, наприклад порушення унікальності назви, у зрозумілі для інтерфейсу повідомлення.

Модуль wireguard_service реалізує інтеграцію із системними компонентами WireGuard for Windows. Він відповідає за пошук wireguard.exe і wg.exe, побудову runtime-конфігурації, запис конфігураційного файлу, встановлення та видалення тунелю як Windows-сервісу, перевірку його стану через sc query, а також зчитування часу останнього handshake і статистики переданого трафіку. Завдяки цьому прикладний рівень взаємодіє не лише з локальною БД, а й із зовнішнім системним середовищем операційної системи.

Модуль session_service координує життєвий цикл VPN-сесії. Після успішного запуску тунелю він відкриває новий запис у таблиці sessions, запускає фонову перевірку handshake та за результатами оновлює стан сесії. Якщо handshake відсутній, сервіс автоматично відключає тунель і закриває сесію зі статусом no_handshake. Якщо handshake зафіксовано, у базу записуються статус здоров’я тунелю, час останнього handshake та значення RX/TX. Таким чином, сервіс сесій виконує не лише роль посередника між UI та CLI, а й роль координаційного механізму моніторингу підключення.

Модуль history_service використовується для читання, фільтрації, видалення та очищення історії сесій. Він формує прикладний рівень над журналом підключень і надає інтерфейсу готові операції для сторінки History, не дозволяючи UI працювати з таблицею sessions безпосередньо

Модуль dashboard_service призначений для побудови агрегованих показників інформаційної панелі. Він обчислює кількість профілів, активних тунелів, загальну кількість сесій, число проблемних підключень, а також формує дані про

поточний тунель і перелік останніх подій. Таким чином, прикладний рівень охоплює не лише операційні функції системи, а й аналітичний функціонал відображення зведеної інформації.

Рівень доступу до даних реалізовано через модулі profile_repository та session_repository, які інкапсулюють взаємодію з локальною базою даних SQLite. Винесення SQL-логіки в окремі репозиторії дозволяє ізолювати операції збереження, читання, оновлення та видалення від інтерфейсного й сервісного рівнів, що відповідає принципу розділення відповідальностей.

profile_repository забезпечує повний набір CRUD-операцій над таблицею profiles: створення нового запису, отримання списку профілів, читання окремого профілю, оновлення та видалення. Усі операції над конфігураційними профілями проходять саме через цей модуль, що спрощує локалізацію змін у схемі або механізмах збереження.

session_repository реалізує повний життєвий цикл роботи з таблицею sessions. Він відкриває та закриває сесії, зберігає параметри стану тунелю, надає історію підключень із фільтрами, дозволяє видаляти записи, очищати журнал та виконувати агрегуючі запити для сторінки Dashboard. Таким чином, цей модуль виконує роль сховища для підсистеми журналювання та статистики.

Як локальне сховище даних використано SQLite, що є доцільним рішенням для desktop-застосунку, орієнтованого на автономну роботу без окремого серверу бази даних. У модулі ініціалізації схеми застосовано sqlite3.Row, PRAGMA foreign_keys = ON і journal_mode = WAL, що підвищує цілісність і надійність локальної роботи з даними.

Схема локальної бази даних містить дві основні таблиці: profiles і sessions. Таблиця profiles зберігає конфігураційні параметри WireGuard-профілів, зокрема назву, опис, параметри інтерфейсу та peer-з’єднання, а також службові поля auto_connect і save_history. Таблиця sessions призначена для фіксації життєвого циклу підключень і містить посилання на профіль, ім’я тунелю, час початку та завершення сесії, статус, health-стан, час останнього handshake, значення RX/TX і службові діагностичні відомості. Між цими таблицями реалізовано зв’язок

один-до-багатьох через поле profile_id з правилом ON DELETE CASCADE, що забезпечує автоматичне видалення залежних сесій при видаленні профілю.

Взаємодія компонентів застосунку організована за послідовним сценарієм. Під час запуску програми ініціалізується схема локальної бази даних, після чого MainWindow завантажує графічний інтерфейс і через сервісний рівень отримує профілі, історію та dashboard-дані. За запитом користувача відкривається ProfileDialog у режимі створення, імпорту або редагування профілю. Діалогова форма виконує первинну перевірку введення, а в режимі імпорту додатково зчитує та парсить .conf-файл. Далі дані передаються до profile_service, де проходять бізнес-валідацію й нормалізацію, після чого profile_repository виконує операції збереження або оновлення у SQLite.

У разі підключення профілю session_service викликає wireguard_service, який формує runtime-конфігурацію, запускає тунель як Windows-сервіс і відкриває новий запис сесії. Після цього в окремому потоці виконується перевірка handshake, а результат записується до sessions. Для сторінки History MainWindow використовує history_service, який повертає записи журналу відповідно до обраних фільтрів. Для сторінки Dashboard інтерфейс використовує dashboard_service, який формує агреговані показники та поточний стан тунелю. Така схема взаємодії демонструє наявність у застосунку вже реалізованого робочого вертикального зрізу, в якому всі основні рівні архітектури поєднані в єдину систему функціональних сценаріїв.

На структурному рівні проєкт організовано за функціональними модулями, кожен із яких відповідає за свою категорію логіки: модулі інтерфейсу та вікон, модулі діалогових форм, сервіси прикладної логіки, репозиторії доступу до даних, а також модулі ініціалізації бази даних і взаємодії з WireGuard for Windows. Такий підхід дозволяє розробляти й аналізувати окремі частини системи більш ізольовано та підвищує керованість коду.

На відміну від ранніх етапів розроблення, поточна архітектура вже охоплює не лише базові сценарії створення, імпорту, редагування та збереження профілів, а й пошук профілів, ведення та фільтрацію історії сесій, формування

статистичних показників для інформаційної панелі, а також механізми підключення й відключення тунелів із перевіркою їхнього стану. Таким чином, архітектура застосунку не лише створює основу для подальшого розвитку, а й уже забезпечує реалізацію ключових функціональних сценаріїв локальної системи керування WireGuard-профілями.

До основних переваг реалізованої архітектури належать модульність, оскільки логіка поділена між окремими рівнями та сервісами; зручність супроводу, оскільки зміни в одному рівні мінімально впливають на інші; розширюваність, оскільки структура допускає додавання нових функціональних модулів; зменшення зв’язності, оскільки інтерфейс не працює безпосередньо з SQL-операціями або CLI-командами; а також зручність аналізу та тестування прикладної логіки, оскільки сервіси й репозиторії можуть розглядатися окремо від UI.

3.2 Формалізація вимог до програмного забезпечення

Формалізація вимог до програмного забезпечення є ключовим етапом проєктування, на якому здійснюється перехід від концептуального бачення системи до чітко структурованих специфікацій, придатних для реалізації, верифікації та подальшого супроводу. Для настільного застосунку, що призначений для керування VPN-профілями WireGuard у середовищі Windows, включаючи моніторинг стану тунелів у реальному часі, ведення журналу сесій та надання зручного графічного інтерфейсу користувача, моделювання вимог дозволяє досягти однозначності між описаними функціями та їхньою технічною реалізацією.

Основою для побудови моделі вимог стали функціональні та нефункціональні вимоги, що охоплюють такі аспекти, як управління профілями, налаштування VPN-з'єднань, відстеження сесій, перегляд та фільтрація історії підключень, відображення зведених показників на інформаційній панелі, а також

параметри користувацького інтерфейсу, включаючи підтримку локалізації та зміну теми оформлення.

У ході моделювання вимог було визначено логічну класифікацію функціональностей, що об'єднані у функціональні блоки відповідно до сценаріїв використання системи. Такий підхід забезпечує цілісне структурування вимог і дозволяє встановити чіткі межі відповідальності кожного модуля.

Блок управління профілями охоплює операції зі створення, редагування, імпорту та видалення конфігурацій WireGuard. Користувач може створити профіль вручну, заповнивши всі необхідні поля конфігурації (приватний ключ, IP-адреса інтерфейсу, публічний ключ однорангового вузла, кінцева точка, дозволені IP-адреси тощо), або імпортувати готовий файл .conf, що автоматично розбирається системою та заповнює відповідні поля. Для кожного профілю передбачена можливість встановлення прапорців автопідключення та збереження журналу сесій. Крім того, реалізована функція текстового пошуку профілів у реальному часі за назвою, що забезпечує зручну навігацію при великій кількості конфігурацій. При видаленні профілю система каскадно вилучає всі пов'язані з ним записи сесій відповідно до механізму зовнішніх ключів бази даних.

Блок управління VPN-підключеннями реалізує взаємодію з системними компонентами WireGuard for Windows через командний рядок. На цьому етапі використовується вже попередньо перевірений і збережений профіль: валідація конфігураційних параметрів виконується на рівні діалогової форми та сервісу профілів під час створення або редагування запису. Під час підключення система формує runtime-конфігурацію у форматі .conf, записує її до локального каталогу конфігурацій та викликає wireguard.exe для встановлення тунелю як Windows-сервісу. Після запуску сервісу здійснюється опитування його стану через sc query, що дозволяє підтвердити активність тунелю. При відключенні виконується виклик команди видалення тунельного сервісу. Ідентифікатор тунелю формується як wd_<id профілю>, що гарантує його унікальність у межах системи.

Блок моніторингу стану тунелю забезпечує перевірку стану з'єднання після встановлення тунелю. Асинхронний механізм перевірки стану запускається після підключення з затримкою у п'ять секунд і фіксує результат через виклик команди wg show latest-handshakes. Якщо між однорангами було встановлено handshake - сесія позначається як здорова (healthy); якщо handshake відсутній, система автоматично розриває з'єднання та позначає сесію статусом no_handshake. Поточний стан тунелю, включаючи кількість прийнятих та переданих байт (RX/TX) та час останнього handshake, відображається в режимі реального часу.

Блок відстеження та управління сесіями відповідає за ведення записів про активні та завершені VPN-підключення. Під час підключення система автоматично створює запис сесії у базі даних із початковими параметрами. При відключенні або зміні стану тунелю фіксуються час завершення, тривалість у секундах, кінцевий статус (connected, disconnected, failed, no_handshake), статус стану підключення (healthy, no_handshake, unknown), час останнього handshake та обсяги переданого трафіку. Усі ці дані збираються через виклик утиліти wg та зберігаються у локальній базі даних SQLite.

Блок перегляду та фільтрації історії надає користувачу можливість переглядати повну хронологію сесій у вигляді таблиці з дев'ятьма стовпцями: назва профілю, час початку та завершення сесії, тривалість, статус, стан підключення, обсяги прийнятого та переданого трафіку, а також кнопка видалення окремого запису. Реалізовано фільтрацію за часовим діапазоном (з вибором дат початку та кінця) і за профілем. Над таблицею відображаються зведені статистичні показники: загальний час підключення, загальний обсяг отриманого та переданого трафіку для відфільтрованих записів. Передбачені функції видалення окремих записів та масового очищення журналу.

Блок інформаційної панелі (Dashboard) призначений для відображення поточного стану системи у режимі реального часу. Інформаційна панель містить чотири зведені картки із ключовими показниками: загальна кількість профілів, кількість активних тунелів, загальна кількість сесій та кількість сесій із відсутнім

handshake. У центральній частині відображається блок поточного тунелю із назвою профілю, кінцевою точкою, статусом підключення та стану, часом останнього handshake і поточними показниками трафіку. Нижче відображається список п'яти останніх активностей. Панель автоматично оновлюється кожні п'ять секунд засобами таймера Qt, проте лише тоді, коли активна вкладка Dashboard, що виключає зайве навантаження.

Вимоги до інтерфейсу користувача охоплюють три основні сторінки навігації: Dashboard, Profiles і History. Перемикання між ними здійснюється через бічне меню навігації. Система підтримує дві теми оформлення - темну та світлу (з кольоровою схемою Catppuccin) - із можливістю перемикання в реальному часі за допомогою відповідної кнопки. Реалізована підтримка двох мов інтерфейсу: англійської та української, із перемиканням через випадаючий список у верхній панелі. Всі рядки інтерфейсу визначені у централізованому словнику перекладів із резервним перемиканням на англійську мову за відсутності ключа. Компоненти інтерфейсу побудовані на основі Qt Designer XML, що відокремлює представлення від логіки.

Суттєвою вимогою до інтерфейсу є також підтримка локалізації. У поточній реалізації передбачено дві мови інтерфейсу: англійську та українську, із перемиканням через випадаючий список у верхній панелі. Така організація локалізації є доцільною, оскільки дозволяє змінювати мову без прихованих налаштувань і забезпечує прямий доступ до цієї функції з будь-якої сторінки застосунку.

Таким чином, формалізація вимог до програмного забезпечення дала змогу подати розроблюваний застосунок як систему взаємопов’язаних функціональних блоків, кожен із яких має визначене призначення, набір сценаріїв використання та відповідність конкретним програмним модулям. Такий підхід забезпечує узгодженість між вимогами, архітектурою і фактичною реалізацією застосунку, а також створює основу для подальшого тестування, оцінювання повноти реалізації та розвитку програмного засобу.

У табл. 3.1 наведено узагальнення ключових функціональних блоків, їх логічне призначення та пов'язані з ними модулі системи.

Таблиця 3.1

Функціональні блоки системи

№	Функціональний блок	Призначення	Пов’язані модулі
1	Управління профілями	Створення, редагування, імпорт, видалення VPN-конфігурацій	profile_service.py, profile_repository.py
2	Управління підключеннями	Встановлення та розрив VPN-тунелів через WireGuard CLI	wireguard_service.py, command_runner.py,
3	Моніторинг стану тунелю	Перевірка handshake, відстеження трафіку, автовідключення	session_service.py,
4	Відстеження сесій	Ведення записів про підключення у базі даних	session_service.py, session_repository.py
5	Журнал та фільтрація історії	Перегляд, пошук і видалення записів про минулі сесії	history_service.py
6	Інформаційна панель	Відображення агрегованих показників у режимі реального часу	dashboard_service.py, ui_main_window.py
7	Інтерфейс користувача	Навігація, теми оформлення, локалізація, пошук	ui_main_window.py, translations.py, profile_dialog.py

На Рис. А.1. Діаграма варіантів використання подано діаграму варіантів використання, яка відображає основні сценарії взаємодії користувача з системою. Спроектована діаграма варіантів використання, що відображає основні сценарії взаємодії користувача з системою, наведена у Додатку А, ст. 67

Отже, наведені положення формують цілісне уявлення про структуру, призначення та напрям подальшого розвитку розроблюваного програмного засобу. Сукупність визначених компонентів і функціональних можливостей створює основу для реалізації локального desktop-застосунку керування WireGuard-профілями, а також забезпечує узгодженість між архітектурними рішеннями, функціональними вимогами та практичною реалізацією системи. Це, у свою чергу, дозволяє перейти до подальшого детального розгляду окремих модулів, діаграм і сценаріїв функціонування застосунку.

3.3 Проєктування бази даних

Проєктування бази даних є важливим етапом розроблення програмного забезпечення, оскільки саме структура зберігання даних визначає спосіб організації інформації, її цілісність, доступність та можливість подальшого розвитку системи. Для desktop-застосунку, призначеного для локального керування профілями WireGuard, база даних повинна забезпечувати не лише збереження конфігураційних параметрів профілів, а й фіксацію фактів підключення, збереження історії сесій та підтримку агрегованих даних для інформаційної панелі.

У межах розроблюваного застосунку для організації локального сховища використано SQLite, що є доцільним рішенням для автономного desktop-продукту, який не потребує окремого серверного середовища. Підключення до бази даних реалізоване засобами стандартного модуля sqlite3; при цьому використовується sqlite3.Row, режим PRAGMA foreign_keys = ON та journal_mode = WAL, що підвищує надійність роботи з даними і забезпечує коректну підтримку зовнішніх ключів .

Фактична схема локальної бази даних застосунку містить дві основні таблиці: profiles і sessions . Така структура відповідає реалізованій логіці системи, у якій профіль виступає основною конфігураційною сутністю, а сесія -

записом про конкретний цикл роботи тунелю, його стан, тривалість і параметри обміну даними.

Таблиця profiles призначена для збереження конфігураційних параметрів WireGuard-профілів. У ній зберігаються не лише базові значення, необхідні для побудови .conf-конфігурації, а й службові параметри застосунку. До складу таблиці входять поля: id, name, description, auto_connect, save_history, private_key, address, dns, listen_port, mtu, public_key, preshared_key, allowed_ips, endpoint і persistent_keepalive . Саме ця таблиця використовується в сценаріях створення, імпорту, редагування, пошуку та видалення профілів, що реалізовано через profile_repository і profile_service .

Таблиця sessions призначена для збереження інформації про життєвий цикл підключень. У ній фіксуються ідентифікатор сесії, прив’язка до профілю (profile_id), ім’я тунелю, час початку та завершення сесії, її тривалість, статус підключення, стан здоров’я тунелю, час останнього handshake, обсяги вхідного та вихідного трафіку, а також службові деталі виконання . На практиці ця таблиця використовується не лише як журнал історії, а й як джерело даних для моніторингу поточного стану тунелю та побудови зведених показників Dashboard.

Зв’язок між таблицями реалізовано через зовнішній ключ sessions.profile_id → profiles.id з правилом ON DELETE CASCADE . Це означає, що у разі видалення профілю всі пов’язані з ним записи сесій автоматично вилучаються з бази даних. Такий підхід забезпечує цілісність схеми та дозволяє уникнути накопичення “висячих” записів історії, які більше не пов’язані з жодним профілем.

Логіка роботи з таблицею profiles реалізована в profile_repository.py, де забезпечено повний набір CRUD-операцій: створення профілю, отримання списку профілів, читання окремого запису, оновлення та видалення . Поверх цього шару працює profile_service.py, який виконує бізнес-валідацію конфігураційних даних, нормалізацію полів і перетворення технічних помилок бази даних у зрозумілі повідомлення для графічного інтерфейсу .

Таблиця sessions використовується через session_repository.py, де реалізовано відкриття сесії, її закриття, оновлення даних про стан тунелю, вибірку відкритих і завершених сесій, а також агрегуючі запити для статистики й історії. На рівні сервісів ця таблиця інтегрована з session_service.py, який координує життєвий цикл сесії під час підключення й відключення тунелю, запускає асинхронну перевірку handshake та записує результати моніторингу в базу даних . Крім того, history_service.py використовує ці дані для побудови журналу підключень і фільтрації історії , a dashboard_service.py - для формування зведених показників і поточного стану системи.

На рисунку 3.2 зображено логічну ER-діаграму, яка відображає основні сутності та їхні зв’язки у структурі бази даних.

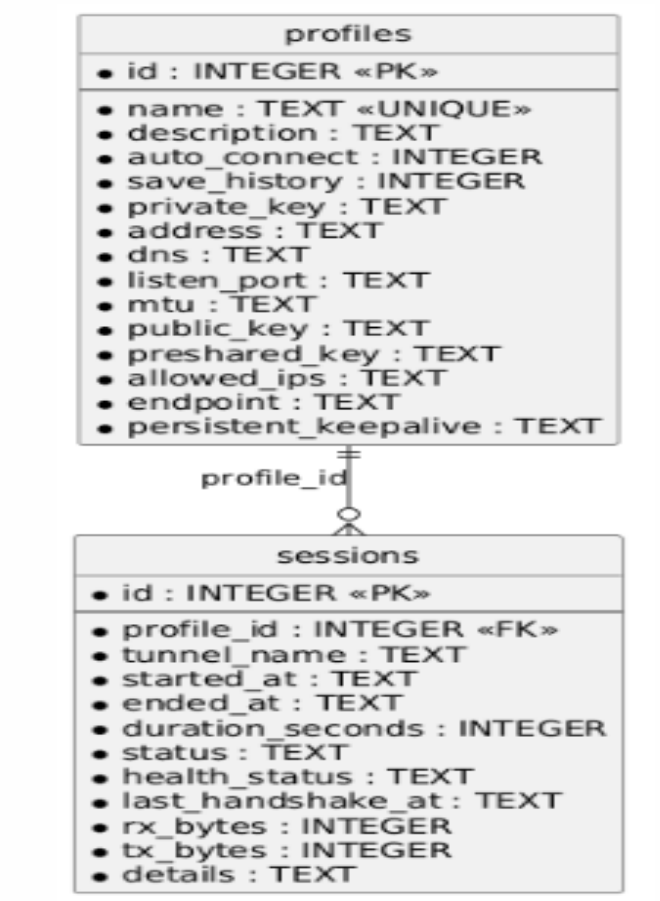


Рис. 3.2 ER-діаграма логічної структури зберігання даних

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Діаграма класів

Діаграма класів є одним із базових засобів об’єктно-орієнтованого проєктування, оскільки вона дає змогу відобразити основні програмні сутності системи, їх атрибути, методи, а також залежності й зв’язки між окремими компонентами архітектури. Для розроблюваного desktop-застосунку керування WireGuard-профілями діаграма класів має особливе значення, оскільки дозволяє наочно показати, яким чином організовано взаємодію між графічним інтерфейсом, прикладною логікою, механізмами доступу до даних і компонентами керування VPN-тунелями.

На відміну від систем, у яких предметна область реалізується переважно через велику кількість окремих класів-моделей, у даному застосунку використано змішаний архітектурний підхід. Частина функціональності представлена у вигляді класів інтерфейсного рівня та допоміжних типів, тоді як значна частина прикладної логіки реалізована у вигляді окремих сервісних і репозиторних модулів. Саме тому в межах цієї роботи діаграма класів повинна відображати не лише буквально наявні Python-класи, а й логічні компоненти сервісного та репозиторного рівнів, які доцільно подати в UML як окремі класи зі стереотипами <<service>> і <<repository>>. Такий підхід забезпечує коректне відображення реальної структури системи та зберігає її архітектурну зрозумілість.

У структурі діаграми класів доцільно виділити кілька логічних груп компонентів:

класи рівня представлення, допоміжні типи, сервісні компоненти, репозиторії доступу до даних, а також логічні сутності зберігання даних.

На рисунку 4.1 буде подано повну UML-діаграму класів, що демонструє ключові сутності, зв’язки та методи.

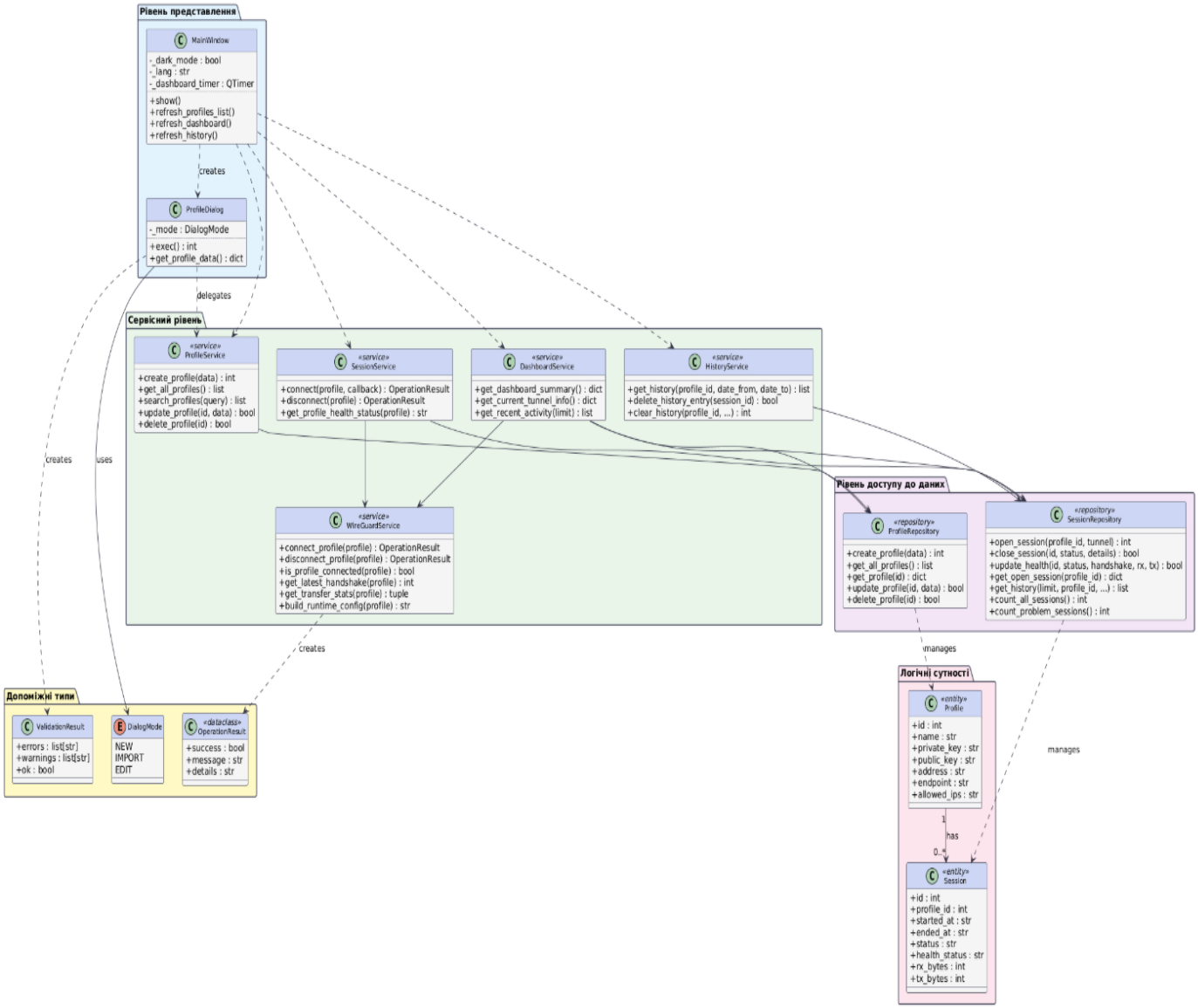


Рис 4.1 UML-діаграма класів архітектури застосунку

Класи рівня представлення:

Центральним елементом рівня представлення є клас MainWindow, який виконує роль координатора графічного інтерфейсу застосунку. Саме цей клас ініціалізує схему локальної бази даних при запуску програми, завантажує інтерфейс із .ui-файлу, організовує навігацію між сторінками Dashboard, Profiles і History, а також забезпечує виклик сервісів для оновлення списку профілів, історії сесій і показників інформаційної панелі. Крім того, MainWindow обробляє

основні дії користувача, пов’язані зі створенням, імпортом, редагуванням і відображенням профілів, переглядом історії та оновленням dashboard-даних.

Другим ключовим класом інтерфейсного рівня є ProfileDialog, який реалізує універсальне діалогове вікно для створення, імпорту та редагування профілю. У межах цього класу підтримуються три режими роботи: NEW, IMPORT і EDIT. Він відповідає за зчитування даних із форми, імпорт .conf-файлу, розбір секцій [Interface] і [Peer], первинну перевірку введення та повернення сформованої структури даних до прикладного рівня. Таким чином, ProfileDialog виконує роль спеціалізованого інтерфейсного контролера для роботи з конфігураційними параметрами профілю

Допоміжні типи:

Окрему групу на діаграмі класів становлять допоміжні типи, які забезпечують структуровану взаємодію між компонентами системи. До таких типів належить DialogMode, який визначає режими роботи діалогу профілю, а також ValidationResult, що використовується для накопичення помилок і попереджень під час первинної перевірки даних у ProfileDialog.

Важливим допоміжним типом є також OperationResult, реалізований як dataclass у wireguard_service.py. Цей тип використовується для уніфікованого повернення результатів операцій підключення та відключення тунелю. Він містить ознаку успішності, коротке повідомлення та службову діагностичну інформацію. Використання такого типу спрощує взаємодію між сервісами й графічним інтерфейсом та забезпечує стандартизовану форму подання результатів системних дій.

Сервісні компоненти:

На прикладному рівні доцільно виділити кілька основних сервісних компонентів.

Компонент ProfileService відповідає за бізнес-валідацію, нормалізацію даних профілю та взаємодію з репозиторієм профілів. Саме він виконує перевірку назви профілю, ключів, адреси інтерфейсу, кінцевої точки та числових параметрів конфігурації, а також перетворює технічні помилки бази даних на

зрозумілі повідомлення для користувача. У структурі системи цей компонент забезпечує другий рівень перевірки даних після первинної валідації у ProfileDialog.

Компонент WireGuardService реалізує взаємодію із системними утилітами WireGuard for Windows. Він відповідає за формування runtime-конфігурації, встановлення і видалення тунелю як Windows-сервісу, перевірку факту активності сервісу, зчитування часу останнього handshake та статистики переданого трафіку. У діаграмі класів цей компонент доцільно подати як окремий сервісний блок, що поєднує внутрішню логіку застосунку із зовнішнім системним середовищем ОС та утилітами wireguard.exe, wg.exe і sc.

Компонент SessionService координує життєвий цикл VPN-сесій. Він поєднує виклики WireGuardService із записами в таблицю sessions, відкриває нову сесію після підключення, запускає асинхронну перевірку handshake, записує health-статус і показники трафіку, а також завершує сесію у випадку відключення або відсутності handshake. З погляду UML цей компонент є одним із центральних елементів системи, оскільки саме через нього з’єднуються логіка керування тунелем і механізм журналювання сесій.

Компонент HistoryService відповідає за читання, фільтрацію, видалення та очищення історії завершених сесій. Він функціонує як окремий прикладний шар над репозиторієм сесій і забезпечує логіку роботи сторінки History.

Компонент DashboardService призначений для агрегування статистики, формування зведених показників, визначення поточного стану тунелю та побудови списку останніх активностей. Цей компонент використовує як дані репозиторіїв, так і живі показники WireGuard, тому в діаграмі класів він виступає як сервіс аналітичного рівня, що об’єднує інформацію з кількох джерел для подальшого відображення на сторінці Dashboard.

Репозиторії доступу до даних:

На рівні доступу до даних основними компонентами системи є ProfileRepository та SessionRepository.

ProfileRepository забезпечує CRUD-операції над таблицею profiles, зокрема створення профілю, отримання списку профілів, читання окремого запису, оновлення та видалення. У межах діаграми класів цей компонент доцільно розглядати як ізольований шлюз доступу до конфігураційних даних профілю, що приховує SQL-логіку від інтерфейсного та сервісного рівнів

SessionRepository реалізує роботу з таблицею sessions, включаючи відкриття та закриття сесій, оновлення health-даних, вибірку історії, очищення журналу та агрегуючі запити для dashboard-показників. У класовій діаграмі цей компонент слід подати як окремий репозиторій журналу сесій, який використовується сервісами моніторингу, історії та аналітики.

Логічні сутності даних:

Хоча у поточній Python-реалізації сутності профілю та сесії не винесені в окремі повноцінні класи предметної області, а в багатьох місцях передаються у вигляді словників, на діаграмі класів доцільно відобразити їх як логічні сутності - Profile і Session. Такий підхід є виправданим із погляду аналітичного моделювання, оскільки ці сутності мають чітко визначені атрибути, зберігаються в окремих таблицях та лежать в основі більшості функціональних сценаріїв системи.

Сутність Profile містить такі основні атрибути: id, name, description, auto_connect, save_history, private_key, address, dns, listen_port, mtu, public_key, preshared_key, allowed_ips, endpoint, persistent_keepalive. Сутність **Session** включає id, profile_id, tunnel_name, started_at, ended_at, duration_seconds, status, health_status, last_handshake_at, rx_bytes, tx_bytes та details. Між цими сутностями існує зв’язок один-до-багатьох, оскільки одному профілю може відповідати багато сесій підключення.

Характер зв’язків на діаграмі:

Для поточної реалізації характерними є не стільки ієрархії наслідування, скільки зв’язки залежності, використання та асоціації. MainWindow залежить від ProfileDialog і звертається до сервісних компонентів. ProfileDialog використовує DialogMode і ValidationResult, а після підтвердження передає дані до

ProfileService. ProfileService взаємодіє з ProfileRepository. SessionService залежить від WireGuardService, SessionRepository і типу OperationResult. HistoryService працює із SessionRepository. DashboardService використовує ProfileRepository, SessionRepository та WireGuardService. Отже, діаграма класів відображає передусім координацію між рівнями системи, а не класичні ієрархії успадкування.

4.2 Реалізація функцій

Програмна реалізація функціональних можливостей desktop-застосунку виконана відповідно до шаруватої архітектури із чітким розділенням логіки представлення, прикладних сервісів та доступу до даних. Основною мовою розробки є Python 3, графічний інтерфейс реалізовано засобами фреймворку PySide6 (Qt for Python). Усі функціональні блоки реалізовано у вигляді окремих модулів - сервісів і репозиторіїв, - що забезпечує незалежність компонентів, повторне використання коду та зручність супроводу. Взаємодія з операційною системою здійснюється через виклики системних утиліт WireGuard for Windows за допомогою стандартного модуля subprocess. Нижче описано ключові реалізовані функції.

1. Управління профілями VPN

Функціонал створення та редагування профілів реалізований у модулі services/profile_service.py. Публічна функція create_profile() виконує двоетапну

```
91  def create_profile(data: dict) -> int:
92
93      _validate(data)
94      clean = _normalise(data)
95      try:
96          return repo.create_profile(clean)
97      except sqlite3.IntegrityError as exc:
98          _guard_integrity(exc, clean["name"])
```

Рис 4.2 Публічна функція create_profile()

обробку вхідних даних: спочатку викликається внутрішня функція _validate(), яка перевіряє коректність усіх обов'язкових полів конфігурації WireGuard за допомогою регулярних виразів,

```
_B64_RE   = re.compile(r"^[A-Za-z0-9+/]{42,44}={0,2}$")
_CIDR_RE  = re.compile(r"^\d{1,3}(\.\d{1,3}){3}/\d{1,2}$")
_HOST_RE  = re.compile(r"^.+:\d{1,5}$")

Windsurf: Refactor | Explain | Generate Docstring | ×
def _validate(self) -> ValidationResult:
    result = ValidationResult()
    v = result

    name = self.input_profile_name.text().strip()
    if not name:
        v.errors.append("Profile Name is required.")
    elif len(name) < 2:
        v.errors.append("Profile Name must be at least 2 characters.")

    pk = self.input_private_key.text().strip()
    if not pk:
        v.errors.append("Private Key is required.")
    elif not self._B64_RE.match(pk):
        v.errors.append("Private Key does not look like a valid Base64 WireGuard key.")

    addr = self.input_address.text().strip()
    if not addr:
        v.errors.append("Address is required.")
    elif not self._CIDR_RE.match(addr.split(",")[0].strip()):
        v.warnings.append("Address format looks unusual (expected x.x.x.x/prefix).")

    pubkey = self.input_public_key.text().strip()
    if not pubkey:
        v.errors.append("Peer Public Key is required.")
    elif not self._B64_RE.match(pubkey):
        v.errors.append("Peer Public Key does not look like a valid Base64 WireGuard key.")

    allowed = self.input_allowed_ips.text().strip()
    if not allowed:
        v.errors.append("Allowed IPs is required.")

    endpoint = self.input_endpoint.text().strip()
    if not endpoint:
        v.errors.append("Endpoint is required.")
    elif not self._HOST_RE.match(endpoint):
        v.warnings.append("Endpoint format looks unusual (expected host:port).")

    psk = self.input_preshared_key.text().strip()
    if psk and not self._B64_RE.match(psk):
        v.warnings.append("Preshared Key does not look like a valid Base64 key.")

    port = self.input_listen_port.text().strip()
    if port and not port.isdigit():
        v.errors.append("Listen Port must be a number.")

    mtu = self.input_mtu.text().strip()
    if mtu and not mtu.isdigit():
        v.errors.append("MTU must be a number.")

    ka = self.input_persistent_keepalive.text().strip()
    if ka and not ka.isdigit():
        v.errors.append("Keepalive must be a number (seconds).")

    return result
```

Рис 4.3 внутрішня функція _validate()

а потім функція _normalise() приводить дані до канонічного вигляду перед збереженням у базі.

```
def _normalise(data: dict) -> dict:

    return {
        "name":                 data.get("name", "").strip(),
        "description":          data.get("description", "").strip(),
        "auto_connect":         int(bool(data.get("auto_connect", False))),
        "save_history":         int(bool(data.get("save_history", True))),
        "private_key":          data.get("private_key", "").strip(),
        "address":              data.get("address", "").strip(),
        "dns":                  data.get("dns", "").strip(),
        "listen_port":          data.get("listen_port", "").strip(),
        "mtu":                  data.get("mtu", "").strip(),
        "public_key":           data.get("public_key", "").strip(),
        "preshared_key":        data.get("preshared_key", "").strip(),
        "allowed_ips":          data.get("allowed_ips", "").strip(),
        "endpoint":             data.get("endpoint", "").strip(),
        "persistent_keepalive": data.get("persistent_keepalive", "").strip(),
    }
```

Рис 4.4 функція _normalise()

Перевірці підлягають: відповідність приватного та публічного ключів формату Base64, коректність IP-адреси у нотації CIDR, формат кінцевої точки у вигляді host:port, а також числовий тип необов'язкових параметрів MTU, ListenPort і PersistentKeepalive. У разі порушення обмеження унікальності на рівні бази даних технічний виняток sqlite3.IntegrityError перехоплюється і конвертується у зрозуміле для користувача повідомлення через допоміжну функцію _guard_integrity().

```python
def _guard_integrity(exc: sqlite3.IntegrityError, name: str) -> None:
    """Convert a SQLite UNIQUE violation into a user-friendly ProfileValidationError."""
    msg = str(exc).lower()
    if "unique" in msg and "name" in msg:
        raise ProfileValidationError(
            f"A profile named '{name}' already exists. Choose a different name."
        )
    raise ProfileValidationError(f"Database constraint error: {exc}") from exc
```

Рис 4.5 функція _guard_integrity()

2. Встановлення VPN-підключення

Функція підключення до VPN-тунелю реалізована у модулі services/wireguard_service.py у вигляді функції connect_profile().

```python
def connect_profile(profile: dict) -> OperationResult:

    tunnel_name = get_tunnel_name(profile)
    conf_path   = _config_path(tunnel_name)
    exe         = _find_wireguard_exe()

    if is_profile_connected(profile):
        return OperationResult(
            success=True,
            message=f"Tunnel '{tunnel_name}' is already connected.",
        )

    try:
        conf_path.write_text(build_runtime_config(profile), encoding="utf-8")
    except OSError as exc:
        return OperationResult(
            success=False,
            message=f"Failed to write config for '{tunnel_name}'.",
            details=str(exc),
        )

    ok, details = _run(exe, "/installtunnelservice", str(conf_path))
    if not ok:
        return OperationResult(
            success=False,
            message=f"wireguard.exe failed to install tunnel '{tunnel_name}'.",
            details=details,
        )

    if not _poll_for_running(tunnel_name):
        return OperationResult(
            success=False,
            message=f"Tunnel '{tunnel_name}' was installed but did not reach RUNNING state.",
            details=details,
        )

    return OperationResult(
        success=True,
        message=f"Tunnel '{tunnel_name}' connected.",
        details=details,
    )
```

Рис 4.6 функція connect_profile()

Логіка виконується у чотири послідовні кроки. На першому кроці перевіряється, чи тунель вже активний, що забезпечує ідемпотентність операції. На другому кроці формується runtime-конфігурація за допомогою функції build_runtime_config() та записується у файл у директорії data/configs/.

```python
def build_runtime_config(profile: dict) -> str:
    """
    Render a WireGuard .conf string from a profile dict.

    Only non-empty optional fields are included so the config stays minimal.
    """
    lines: list[str] = ["[Interface]"]
    lines.append(f"PrivateKey = {profile['private_key']}")
    lines.append(f"Address = {profile['address']}")

    if profile.get("dns"):
        lines.append(f"DNS = {profile['dns']}")
    if profile.get("listen_port"):
        lines.append(f"ListenPort = {profile['listen_port']}")
    if profile.get("mtu"):
        lines.append(f"MTU = {profile['mtu']}")

    lines.append("")
    lines.append("[Peer]")
    lines.append(f"PublicKey = {profile['public_key']}")
    lines.append(f"AllowedIPs = {profile['allowed_ips']}")
    lines.append(f"Endpoint = {profile['endpoint']}")

    if profile.get("preshared_key"):
        lines.append(f"PresharedKey = {profile['preshared_key']}")
    if profile.get("persistent_keepalive"):
        lines.append(f"PersistentKeepalive = {profile['persistent_keepalive']}")

    return "\n".join(lines) + "\n"
```

Рис 4.7 функція build_runtime_config()

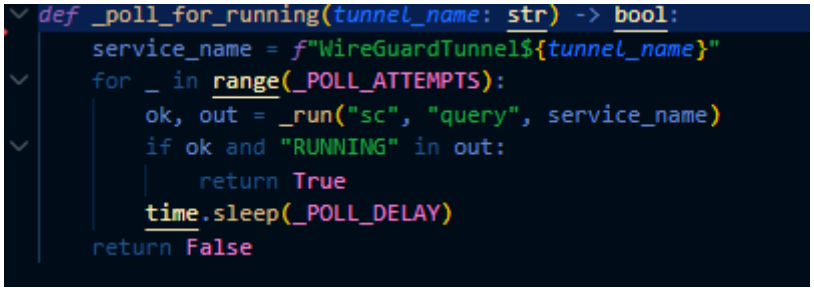
На третьому кроці здійснюється виклик wireguard.exe /installtunnelservice, який реєструє тунель як Windows-сервіс.

```python
def _run(*args: str) -> tuple[bool, str]:
    ts = datetime.now().strftime("%H:%M:%S")
    cmd_str = " ".join(str(a) for a in args)
    try:
        result = subprocess.run(
            args,
            capture_output=True,
            text=True,
            timeout=15,
        )
        output = (result.stdout + result.stderr).strip()
        details = f"[{ts}] $ {cmd_str}\n{output}" if output else f"[{ts}] $ {cmd_str}"
        return result.returncode == 0, details
    except FileNotFoundError:
        return False, f"[{ts}] Command not found: {args[0]}"
    except subprocess.TimeoutExpired:
        return False, f"[{ts}] Command timed out: {cmd_str}"
    except OSError as exc:
        return False, f"[{ts}] OS error running '{cmd_str}': {exc}"
```

Рис 4.8 функція _run()

Оскільки запуск сервісу є асинхронним, на четвертому кроці виконується циклічне опитування стану через sc query внутрішньою функцією

_poll_for_running() до отримання статусу RUNNING. Результат кожного кроку повертається у вигляді об'єкта OperationResult.



```python
def _poll_for_running(tunnel_name: str) -> bool:
    service_name = f"WireGuardTunnel${tunnel_name}"
    for _ in range(_POLL_ATTEMPTS):
        ok, out = _run("sc", "query", service_name)
        if ok and "RUNNING" in out:
            return True
        time.sleep(_POLL_DELAY)
    return False
```

Рис 4.9 функція _poll_for_running()

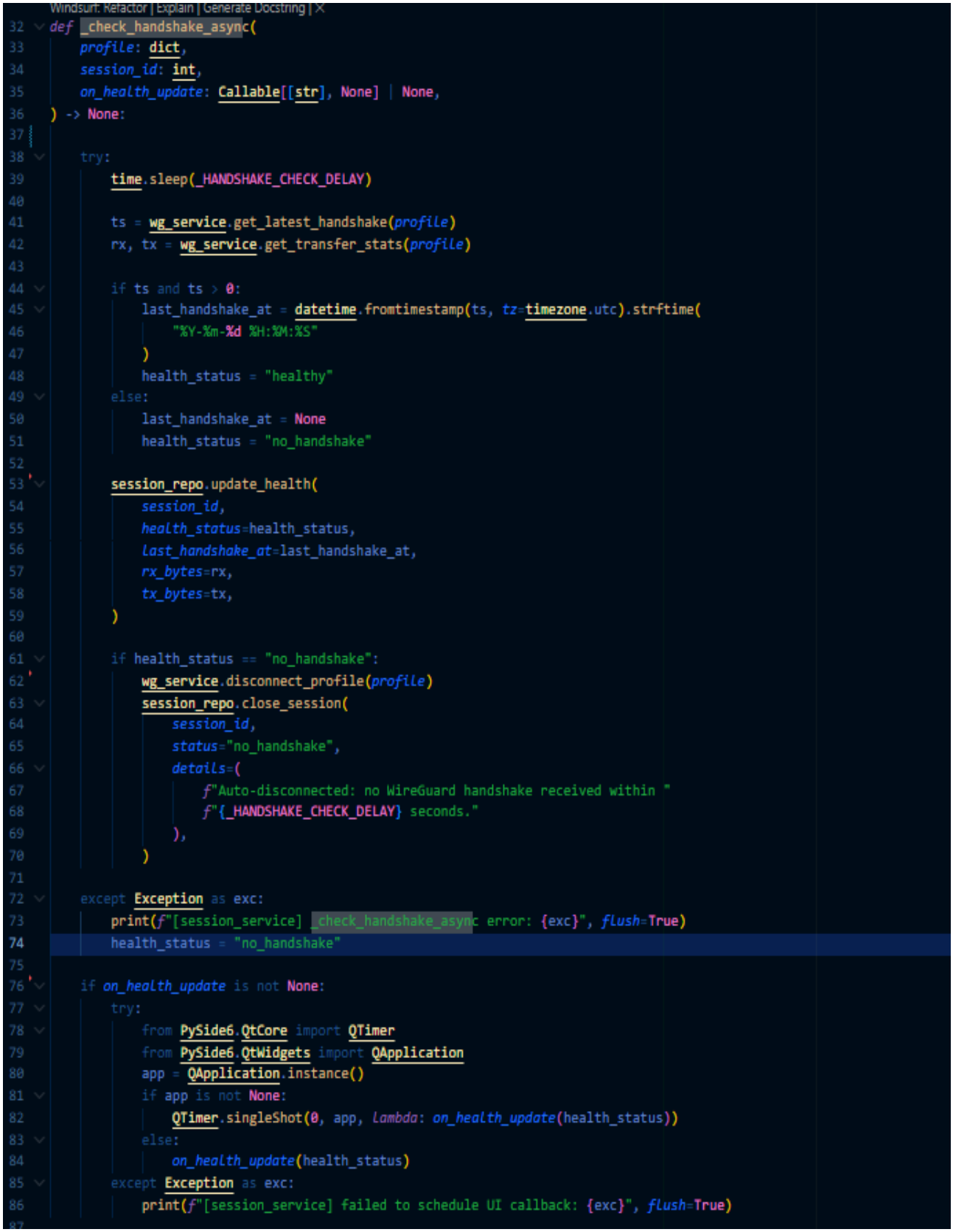
3. Відстеження сесій та асинхронна перевірка handshake

Управління життєвим циклом VPN-сесій реалізоване у модулі services/session_service.py. Публічна функція connect() координує виклик WireGuardService та запис сесії через SessionRepository.

```python
def connect(
    profile: dict,
    on_health_update: Callable[[str], None] | None = None,
) -> OperationResult:
    profile_id   = profile["id"]
    tunnel_name = wg_service.get_tunnel_name(profile)

    if wg_service.is_profile_connected(profile):
        return OperationResult(
            success=True,
            message=f"Tunnel '{tunnel_name}' is already connected.",
        )

    result = wg_service.connect_profile(profile)

    if result.success:
        session_id = session_repo.open_session(profile_id, tunnel_name)
        threading.Thread(
            target=_check_handshake_async,
            args=(profile, session_id, on_health_update),
            daemon=True,
        ).start()
    else:
        session_id = session_repo.open_session(profile_id, tunnel_name)
        session_repo.close_session(
            session_id,
            status="failed",
            details=result.details,
        )

    return result
```

Рис 4.10 функція connect()

Після успішного підключення у окремому фоновому потоці threading.Thread(daemon=True) запускається функція _check_handshake_async(),



```
Windsurf: Refactor | Explain | Generate Docstring | ×
def _check_handshake_async(
    profile: dict,
    session_id: int,
    on_health_update: Callable[[str], None] | None,
) -> None:

    try:
        time.sleep(_HANDSHAKE_CHECK_DELAY)

        ts = wg_service.get_latest_handshake(profile)
        rx, tx = wg_service.get_transfer_stats(profile)

        if ts and ts > 0:
            last_handshake_at = datetime.fromtimestamp(ts, tz=timezone.utc).strftime(
                "%Y-%m-%d %H:%M:%S"
            )
            health_status = "healthy"
        else:
            last_handshake_at = None
            health_status = "no_handshake"

        session_repo.update_health(
            session_id,
            health_status=health_status,
            last_handshake_at=last_handshake_at,
            rx_bytes=rx,
            tx_bytes=tx,
        )

        if health_status == "no_handshake":
            wg_service.disconnect_profile(profile)
            session_repo.close_session(
                session_id,
                status="no_handshake",
                details=(
                    f"Auto-disconnected: no WireGuard handshake received within "
                    f"{_HANDSHAKE_CHECK_DELAY} seconds."
                ),
            )

    except Exception as exc:
        print(f"[session_service] _check_handshake_async error: {exc}", flush=True)
        health_status = "no_handshake"

    if on_health_update is not None:
        try:
            from PySide6.QtCore import QTimer
            from PySide6.QtWidgets import QApplication
            app = QApplication.instance()
            if app is not None:
                QTimer.singleShot(0, app, lambda: on_health_update(health_status))
            else:
                on_health_update(health_status)
        except Exception as exc:
            print(f"[session_service] failed to schedule UI callback: {exc}", flush=True)
```

Рис 4.11 функція _check_handshake_async()

яка через п'ять секунд зчитує час останнього WireGuard-handshake та обсяг переданого трафіку за допомогою утиліти wg.exe. Якщо handshake підтверджено - сесія отримує статус healthy; якщо handshake відсутній - система автоматично

розриває тунель, закриває сесію зі статусом no_handshake та повертає виклик до головного потоку Qt через QTimer.singleShot() для оновлення інтерфейсу.

4. Завершення сесії та відключення тунелю

Функція відключення від VPN реалізована у двох взаємодіючих функціях: disconnect_profile() у services/wireguard_service.py та disconnect() у services/session_service.py. Перша виконує виклик wireguard.exe /uninstalltunnelservice та підтверджує зупинку сервісу через _poll_for_stopped().

```
Windsurf: Refactor | Explain | Generate Docstring | X
def disconnect_profile(profile: dict) -> OperationResult:
    tunnel_name = get_tunnel_name(profile)
    exe         = _find_wireguard_exe()

    if not is_profile_connected(profile):
        return OperationResult(
            success=True,
            message=f"Tunnel '{tunnel_name}' is already disconnected.",
        )

    ok, details = _run(exe, "/uninstalltunnelservice", tunnel_name)
    if not ok:
        return OperationResult(
            success=False,
            message=f"wireguard.exe failed to remove tunnel '{tunnel_name}'.",
            details=details,
        )

    if not _poll_for_stopped(tunnel_name):
        return OperationResult(
            success=False,
            message=f"Tunnel '{tunnel_name}' uninstall command succeeded but service is still running.",
            details=details,
        )

    return OperationResult(
        success=True,
        message=f"Tunnel '{tunnel_name}' disconnected.",
        details=details,
    )
```

Рис 4.12 функція disconnect_profile()

Друга знаходить відкритий запис сесії через SessionRepository.get_open_session та закриває його з відповідним статусом, при цьому тривалість у секундах

обчислюється безпосередньо засобами SQLite через функцію julianday(), що виключає розбіжність між системним часом Python та базою даних.

```python
def disconnect(profile: dict) -> OperationResult:
    profile_id   = profile["id"]
    tunnel_name = wg_service.get_tunnel_name(profile)

    if not wg_service.is_profile_connected(profile):
        open_session = session_repo.get_open_session(profile_id)
        if open_session:
            session_repo.close_session(
                open_session["id"],
                status="disconnected",
                details="Session closed: tunnel was already stopped.",
            )
        return OperationResult(
            success=True,
            message=f"Tunnel '{tunnel_name}' is already disconnected.",
        )

    result = wg_service.disconnect_profile(profile)

    if result.success:
        open_session = session_repo.get_open_session(profile_id)
        if open_session:
            session_repo.close_session(
                open_session["id"],
                status="disconnected",
                details=result.details,
            )

    return result
```

Рис 4.13 функція disconnect()

```python
def _poll_for_stopped(tunnel_name: str) -> bool:
    service_name = f"WireGuardTunnel${tunnel_name}"
    for _ in range(_POLL_ATTEMPTS):
        ok, out = _run("sc", "query", service_name)
        # Service removed entirely (sc returns non-zero) or explicitly stopped
        if not ok or "RUNNING" not in out:
            return True
        time.sleep(_POLL_DELAY)
    return False
```

Рис 4.14 функція _poll_for_stopped()

5. Фільтрація та відображення журналу сесій

Функціонал перегляду та фільтрації історії реалізований у модулі services/history_service.py, який делегує запити до функції get_history()

```python
Windsurf: Refactor | Explain | ×
def get_history(
    limit: int | None = None,
    profile_id: int | None = None,
    date_from: str | None = None,
    date_to: str | None = None,
) -> list[dict]:
    """
    Return closed sessions (ended_at IS NOT NULL) with optional filters.

    *date_from* / *date_to* are ISO date strings (YYYY-MM-DD); *date_to* is
    inclusive through 23:59:59 of that day.
    """
    conditions: list[str] = ["s.ended_at IS NOT NULL"]
    params: list = []

    if profile_id is not None:
        conditions.append("s.profile_id = ?")
        params.append(profile_id)
    if date_from:
        conditions.append("s.started_at >= ?")
        params.append(date_from)
    if date_to:
        conditions.append("s.started_at <= ?")
        params.append(f"{date_to} 23:59:59")

    where = " AND ".join(conditions)
    limit_clause = "LIMIT ?" if limit is not None else ""
    if limit is not None:
        params.append(limit)

    sql = f"""
        SELECT s.*, p.name AS profile_name
        FROM sessions s
        LEFT JOIN profiles p ON p.id = s.profile_id
        WHERE {where}
        ORDER BY s.started_at DESC
        {limit_clause}
    """
    with get_connection() as conn:
        rows = conn.execute(sql, params).fetchall()
    return [_row_to_dict(r) for r in rows]
```

Рис 4.15 функція get_history() репозиторію database/repositories/session_repository.py.

Функція підтримує одночасне застосування трьох незалежних фільтрів: ідентифікатор профілю, нижня та верхня межа діапазону дат. SQL-запит формується динамічно - умови додаються до списку лише за наявності відповідного параметра, що дозволяє виконувати як повну вибірку без фільтрів, так і точково відфільтровані запити без дублювання коду.

6. Агрегування даних інформаційної панелі

Функціонал інформаційної панелі реалізований у модулі services/dashboard_service.py. Функція get_dashboard_payload() об'єднує результати трьох окремих функцій в одному виклику.

```
97
      Windsurf: Refactor | Explain | X
98  def get_dashboard_payload() -> dict:
99      """Return the full dashboard data in one call."""
100     return {
101         "summary":         get_dashboard_summary(),
102         "current_tunnel":  get_current_tunnel_info(),
103         "recent_activity": get_recent_activity(),
104     }
```

Рис 4.16 функція get_dashboard_payload()

Функція get_dashboard_summary() повертає чотири зведені показники: загальну кількість профілів, кількість активних тунелів, загальну кількість сесій та кількість проблемних сесій зі статусом failed або no_handshake.

```
def get_dashboard_summary() -> dict:
    return {
        "total_profiles":  len(profile_repo.get_all_profiles()),
        "active_tunnels":  len(session_repo.get_open_sessions()),
        "total_sessions":  session_repo.count_all_sessions(),
        "problem_sessions": session_repo.count_problem_sessions(),
    }
```

Рис 4.17 функція get_dashboard_summary()

Функція get_current_tunnel_info() спочатку читає збережені дані з відкритої сесії, а потім перезаписує їх актуальними показниками трафіку (rx_bytes, tx_bytes) та

часом handshake, отриманими безпосередньо від wg.exe у реальному часі. У разі недоступності утиліти автоматично використовуються резервні значення з бази даних.

```python
def get_current_tunnel_info() -> dict | None:
    session = session_repo.get_open_session_with_profile()
    if session is None:
        return None

    # Start with stored values as fallback
    rx_bytes          = session.get("rx_bytes")
    tx_bytes          = session.get("tx_bytes")
    last_handshake_at = session.get("last_handshake_at")

    # Overwrite with live WireGuard data when possible
    profile = profile_repo.get_profile(session["profile_id"])
    if profile:
        try:
            ts       = wg_service.get_latest_handshake(profile)
            rx, tx   = wg_service.get_transfer_stats(profile)
            rx_bytes = rx
            tx_bytes = tx
            if ts and ts > 0:
                last_handshake_at = datetime.fromtimestamp(ts).strftime(
                    "%Y-%m-%d %H:%M:%S"
                )
        except Exception:
            pass  # keep stored fallback values

    return {
        "profile_name":      session.get("profile_name") or "—",
        "endpoint":          session.get("profile_endpoint") or "—",
        "status":            session.get("status", "connected"),
        "health_status":     session.get("health_status", "unknown"),
        "last_handshake_at": last_handshake_at,
        "rx_bytes":          rx_bytes,
        "tx_bytes":          tx_bytes,
    }
```

Рис 4.18 функція get_current_tunnel_info()

4.3 Тестування

Після завершення реалізації основних функціональних модулів desktop-застосунку було проведено комплексне тестування з метою перевірки працездатності, стабільності та відповідності програмного засобу поставленим функціональним вимогам. Основною метою тестування було підтвердження

коректної роботи ключових сценаріїв користувача, виявлення можливих помилок у взаємодії між інтерфейсом, сервісним рівнем, репозиторіями та локальною базою даних, а також перевірка реакції системи на некоректне введення даних і проблемні ситуації під час керування VPN-тунелем.

Тестування виконувалося переважно у формі ручної інтерактивної перевірки із моделюванням типових практичних сценаріїв використання системи. У межах перевірки оцінювалися такі групи функцій: створення профілю, імпорт конфігураційного файлу .conf, редагування та видалення профілів, локальне збереження даних у SQLite, запуск і зупинка WireGuard-тунелю, асинхронна перевірка handshake, формування журналу сесій, фільтрація історії, відображення агрегованих показників на інформаційній панелі, а також перемикання теми й мови інтерфейсу

Особлива увага приділялася не лише позитивним сценаріям, а й ситуаціям помилкового введення або часткової недоступності функціонального середовища. Зокрема, перевірялася поведінка системи при введенні некоректних ключів, неправильного формату IP-адреси або кінцевої точки, повторного використання вже існуючої назви профілю, відсутності handshake після підключення, а також у разі спроби видалення профілю, пов’язаного з журналом сесій. Такий підхід дозволив оцінити не лише номінальну працездатність застосунку, а й його стійкість до типових помилок користувача й аномальних ситуацій виконання.

На рисунку 4.19 наведено сторінку Profiles, яка використовувалася під час тестування сценаріїв створення, редагування, пошуку та видалення конфігураційних профілів WireGuard. У межах перевірки було підтверджено коректне відображення списку профілів, оновлення інтерфейсу після внесення змін, а також узгодженість між вмістом локальної бази даних і графічним представленням записів.

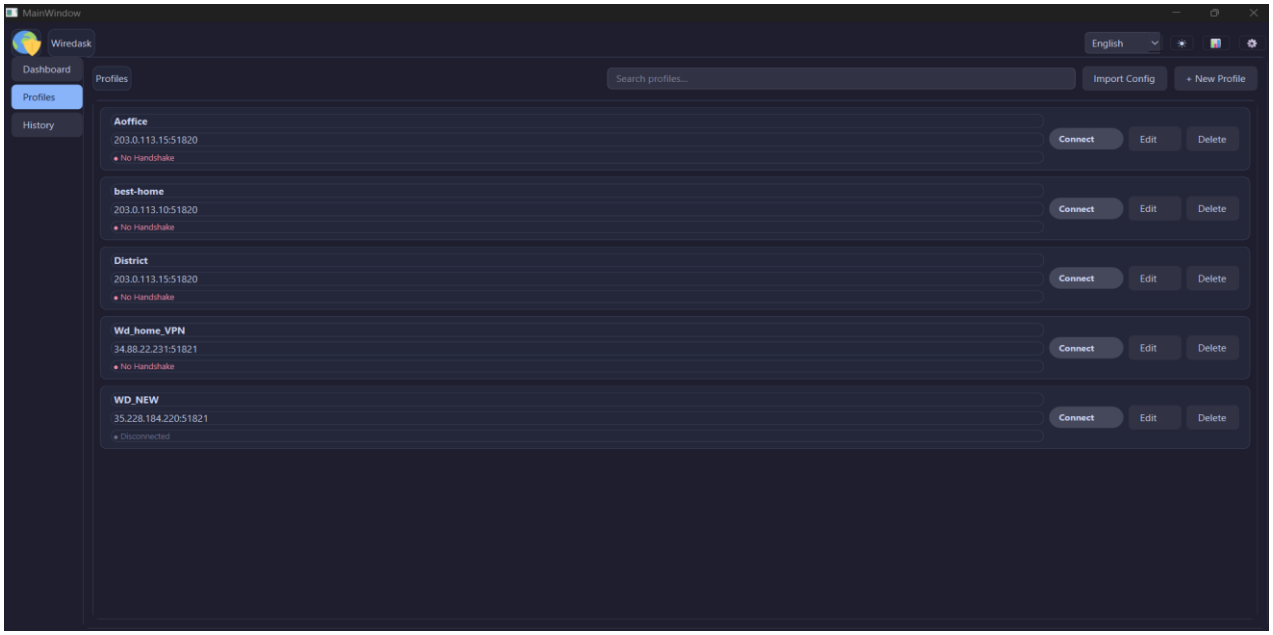


Рис 4.19 Екранні форми сторінки Profiles з VPN профілями

На рисунку 4.20 подано сторінку History, що використовувалася для перевірки функцій перегляду, фільтрації та видалення записів журналу сесій. Під час тестування було встановлено, що система коректно відображає параметри завершених підключень, підтримує відбір записів за заданими критеріями та забезпечує актуальне формування зведених статистичних показників для вибраної вибірки.

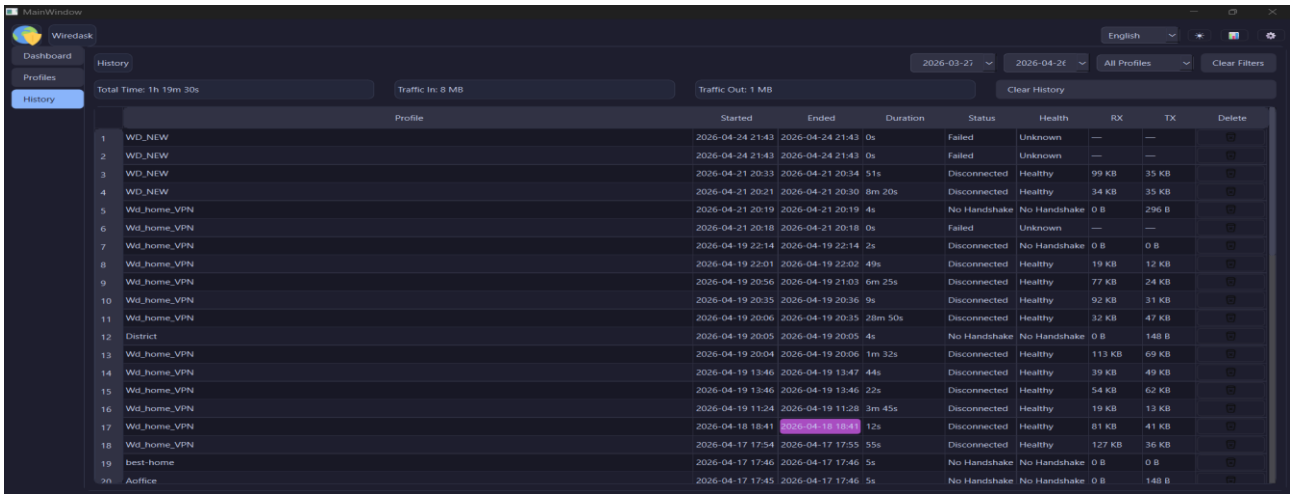


Рис 4.20 Екранні форми сторінки History з історією сесій

На рисунку 4.21 зображено сторінку Dashboard, яка застосовувалася для перевірки правильності формування агрегованих показників системи. У межах тестування було підтверджено коректне відображення кількості профілів,

активних тунелів, загальної кількості сесій, проблемних підключень, а також інформації про поточний тунель і останню активність користувача.

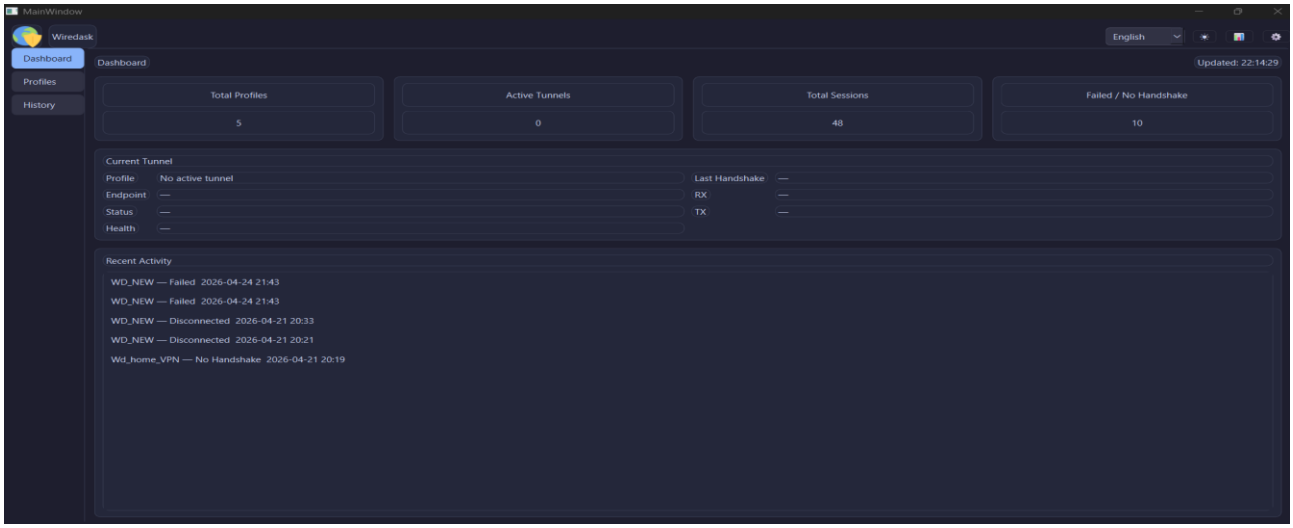


Рис 4.21 Сторінка Dashboard із зведеними статистичними показниками

На рисунку 4.22 наведено діалогові вікна NEW PROFILE, IMPORT PROFILE та EDIT PROFILE, що використовувалися під час тестування сценаріїв введення та модифікації конфігураційних даних. Перевірка показала, що система коректно підтримує різні режими роботи форми, автоматичне заповнення полів під час імпорту .conf-файлу та первинну валідацію введених параметрів перед їх передачею на рівень прикладної логіки.

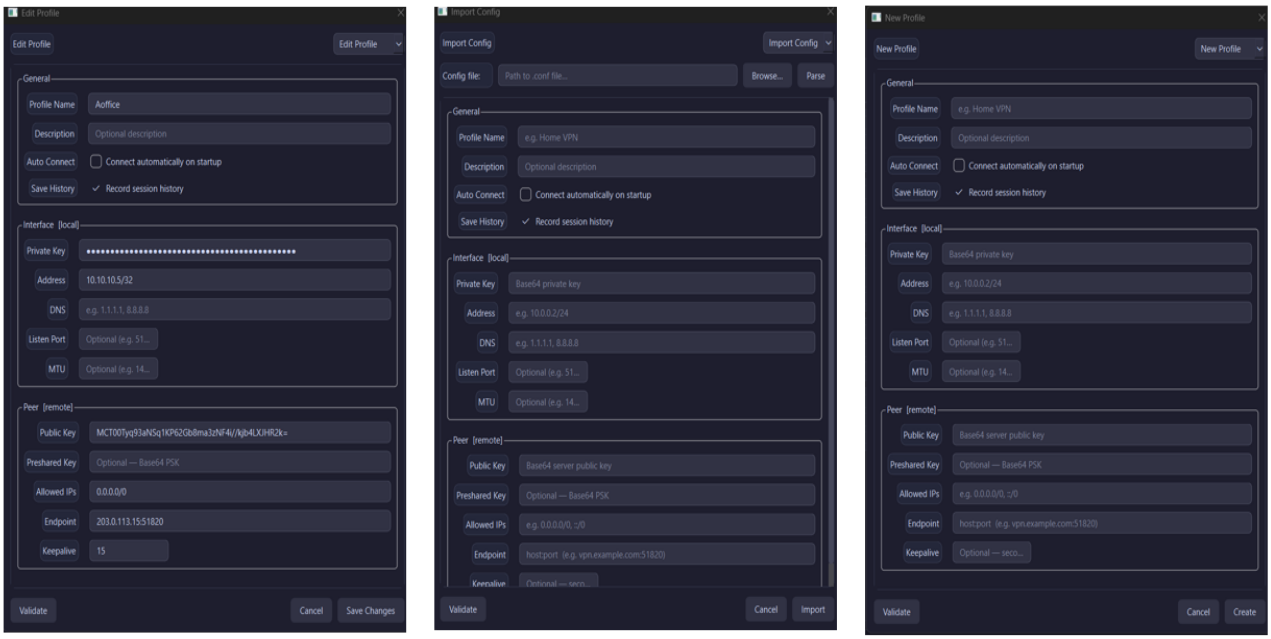


Рис 4.22 Екранні форми вікон EDIT,IMPORT,NEW PROFILE

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розв’язано актуальну задачу розроблення desktop-застосунку для локального керування користувацькими peer-підключеннями WireGuard із графічним інтерфейсом на основі Python і Qt. Основна мета роботи полягала у підвищенні зручності та ефективності роботи з конфігураційними профілями WireGuard шляхом створення програмного засобу, який дозволяє централізовано виконувати їх створення, імпорт, редагування, збереження, перегляд і супровід у межах одного робочого середовища.

У ході виконання роботи було проведено аналіз предметної галузі та досліджено існуючі програмні рішення для керування WireGuard-конфігураціями. Аналіз аналогів показав, що сучасні системи цього класу переважно орієнтовані на серверне розгортання та web-адміністрування, потребують окремої інфраструктури та додаткового мережевого налаштування. Це підтвердило доцільність обраного напряму розроблення локального desktop-застосунку, який не залежить від окремого web-сервера та забезпечує персональне керування профілями безпосередньо на робочій станції користувача.

У роботі було обґрунтовано вибір мови програмування Python, фреймворку PySide6, середовища розробки Visual Studio Code та локальної бази даних SQLite. Такий стек виявився доцільним для побудови програмного засобу конфігураційного типу, оскільки поєднує відносну простоту реалізації, зручність створення графічного інтерфейсу, достатню виразність прикладної логіки та можливість автономної роботи без зовнішнього серверного середовища.

У межах кваліфікаційної роботи було спроєктовано багаторівневу модульну архітектуру застосунку, яка охоплює рівень представлення, рівень прикладної логіки, рівень доступу до даних і рівень локального зберігання. Такий підхід забезпечив розмежування відповідальностей між компонентами системи, підвищив зручність супроводу коду та створив основу для подальшого

розвитку функціональності. Окремо було спроєктовано структуру локальної бази даних, що включає таблиці профілів і сесій, пов’язані між собою відношенням один-до-багатьох.

У процесі реалізації було створено основні функціональні модулі системи. Застосунок підтримує створення нових профілів, імпорт типових конфігураційних файлів .conf, редагування та видалення записів, їх локальне збереження у SQLite, а також відображення у графічному інтерфейсі. Реалізовано дворівневу валідацію конфігураційних даних, що охоплює первинну перевірку на рівні форми та повторну бізнес-валідацію на рівні сервісного шару. Це дозволило зменшити ймовірність помилок під час введення конфігураційних параметрів і підвищити надійність роботи системи.

Важливим результатом роботи стала реалізація не лише функцій локального керування профілями, а й базових механізмів керування WireGuard-тунелями. Застосунок формує runtime-конфігурацію, взаємодіє з WireGuard for Windows через системні утиліти, виконує підключення та відключення тунелю, перевіряє його стан, контролює наявність handshake і фіксує результати в журналі сесій. Додатково реалізовано відображення історії підключень, фільтрацію журналу сесій та формування агрегованих показників для інформаційної панелі. Це дозволяє розглядати створений програмний засіб не лише як редактор конфігурацій, а як цілісну локальну систему керування WireGuard-профілями й пов’язаними з ними сесіями.

У ході тестування було перевірено працездатність основних функціональних сценаріїв застосунку. Зокрема, підтверджено коректність створення, імпорту, редагування, збереження та видалення профілів, правильність роботи локального сховища, механізмів підключення й відключення тунелю, перевірки handshake, журналювання сесій, відображення історії та dashboard-показників. Окремо було перевірено реакцію системи на помилкове введення даних і проблемні ситуації під час встановлення з’єднання.

Результати тестування засвідчили, що розроблений програмний засіб загалом відповідає поставленим функціональним вимогам і демонструє працездатність у межах обраної моделі використання.

Практичне значення отриманих результатів полягає у створенні зручного локального інструмента для організації, супроводу та контролю конфігураційних профілів WireGuard. Розроблений застосунок може бути використаний у навчальних, прикладних і адміністративних сценаріях, де необхідно впорядковувати профілі підключення, зменшувати залежність від ручного редагування конфігураційних файлів та забезпечувати більш наочну взаємодію з параметрами VPN-з’єднання.

Таким чином, у межах кваліфікаційної роботи досягнуто поставленої мети та виконано основні завдання дослідження. Розроблений desktop-застосунок реалізує базовий набір необхідних функцій для локального керування WireGuard-профілями, має модульну архітектуру, узгоджену з сучасними підходами до побудови прикладного програмного забезпечення, і може розглядатися як основа для подальшого функціонального розширення. Перспективами розвитку програмного засобу є вдосконалення механізмів адміністрування, розширення статистичних можливостей, поглиблення моніторингу стану тунелів, підтримка додаткових сценаріїв імпорту та експорту конфігурацій, а також подальше вдосконалення інтерфейсу користувача.

Робота пройшла апробацію на двох наукових конференціях, за результатами апробації опубліковано тези доповідей:

1. Лавренюк Б. Ю., Ананченко О. Є. Програмна система керування WireGuard VPN-профілями: архітектурні рішення та механізми захисту інформації. *VII Всеукраїнська науково-технічна конференція* «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях» (Київ, 23 квітня 2026 р., напрям «Безпека програм та даних»). (Подано до друку).
2. Лавренюк Б. Ю., Ананченко О. Є. Модульна архітектура настільного додатка для керування локальними мережевими профілями. Всеукраїнська

науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація» (Київ, 1–3 травня 2026 р., напрям «Програмні системи та архітектури»). (Подано до друку).

ПЕРЕЛІК ПОСИЛАНЬ

1. Pressman R. S., Maxim B. R. *Software Engineering: A Practitioner's Approach*. 8th ed. New York: McGraw-Hill Education, 2015. 992 p.
2. Sommerville I. *Software Engineering*. 10th ed. Boston: Pearson, 2015. 816 p.
3. Fowler M. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, 2002. 533 p.
4. Martin R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River: Prentice Hall, 2008. 464 p.
5. Martin R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston: Prentice Hall, 2017. 432 p.
6. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1994. 395 p.
7. Python Software Foundation. *Python 3 Documentation*. URL: https://docs.python.org/3/ (дата звернення: 26.04.2026).
8. Python Software Foundation. *sqlite3 - DB-API 2.0 interface for SQLite databases*. URL: https://docs.python.org/3/library/sqlite3.html (дата звернення: 26.04.2026).
9. Python Software Foundation. *subprocess - Subprocess management*. URL: https://docs.python.org/3/library/subprocess.html (дата звернення: 26.04.2026).
10. Python Software Foundation. *threading - Thread-based parallelism*. URL: https://docs.python.org/3/library/threading.html (дата звернення: 26.04.2026).
11. Qt Company. *Qt for Python Documentation*. URL: https://doc.qt.io/qtforpython-6/ (дата звернення: 26.04.2026).
12. Qt Company. *Qt Designer Manual*. URL: https://doc.qt.io/qt-6/qtdesigner-manual.html (дата звернення: 26.04.2026).
13. SQLite Consortium. *SQLite Documentation*. URL: https://www.sqlite.org/docs.html (дата звернення: 26.04.2026).

14. SQLite Consortium. *SQLite Foreign Key Support*. URL: https://www.sqlite.org/foreignkeys.html (дата звернення: 26.04.2026).

15. WireGuard Development Team. *WireGuard: Fast, Modern, Secure VPN Tunnel*. URL: https://www.wireguard.com/ (дата звернення: 26.04.2026).

16. WireGuard Development Team. *WireGuard Quick Start Guide*. URL: https://www.wireguard.com/quickstart/ (дата звернення: 26.04.2026).

17. Donenfeld J. A. *WireGuard: Next Generation Kernel Network Tunnel*. URL: https://www.wireguard.com/papers/wireguard.pdf (дата звернення: 26.04.2026).

18. *wg-easy GitHub Repository*. URL: https://github.com/wg-easy/wg-easy (дата звернення: 26.04.2026).

19. freifunkMUC. *wg-access-server GitHub Repository*. URL: https://github.com/freifunkMUC/wg-access-server (дата звернення: 26.04.2026).

20. WGDashboard. *Official Documentation*. URL: https://wgdashboard.dev/ (дата звернення: 26.04.2026).

21. Donald Zou. *WGDashboard GitHub Repository*. URL: https://github.com/donaldzou/WGDashboard (дата звернення: 26.04.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

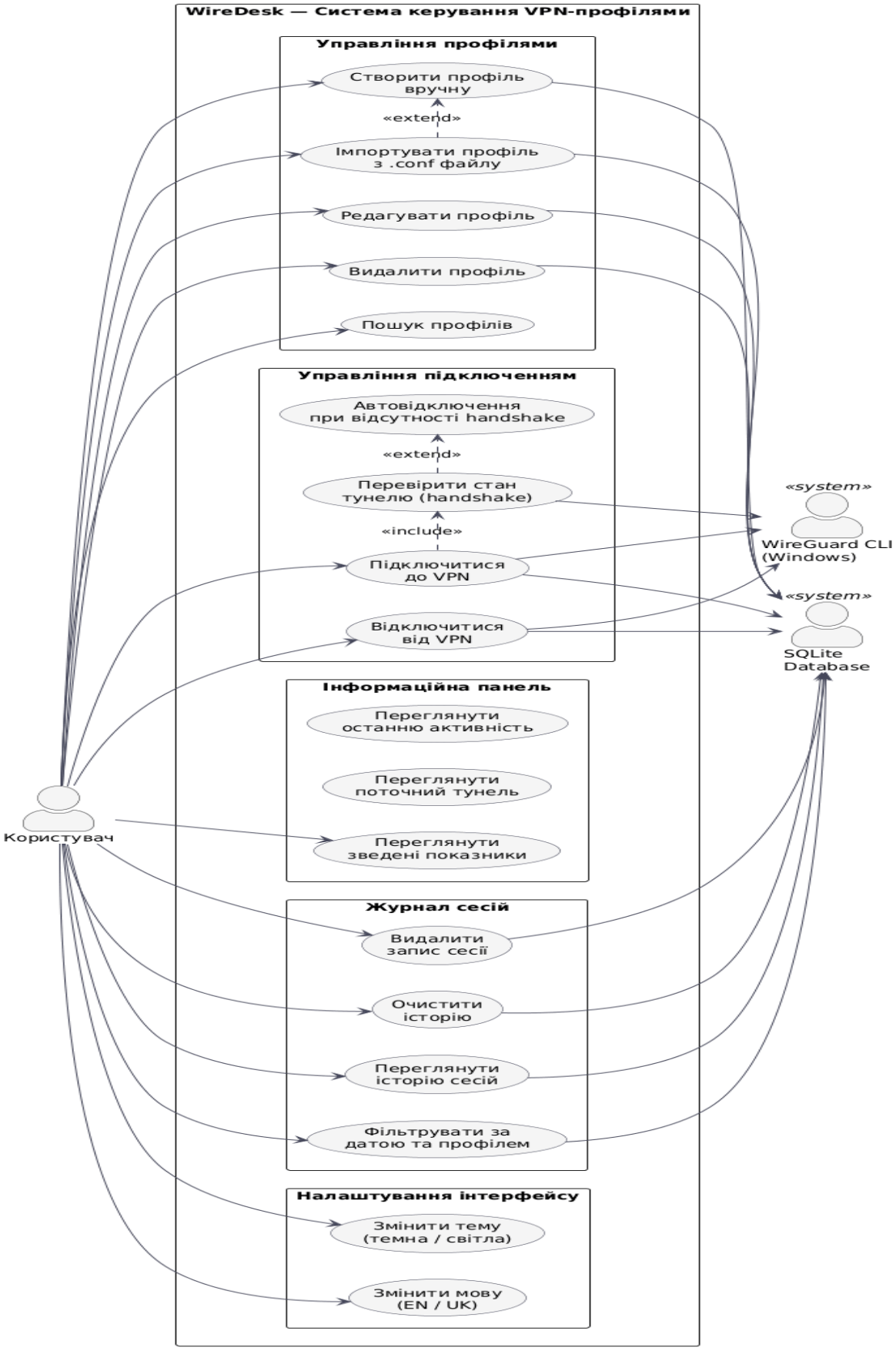


Рис. А.1. Діаграма варіантів використання

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ТЕХНОЛОГІЇ ЦИФОРВОГО РОЗВИТКУ

Система керування користувацькими peer-підключеннями WireGuard із графічним інтерфейсом на Python (PyQt6)

Виконав студент 4 курсу
групи ТЦР-42
Лавренюк Богдан Юрійович
Керівник роботи:
доцент кафедри, кандидат технічних наук
Ананченко Олексій Євгенович

2026

1

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета дослідження** — підвищення зручності та ефективності керування користувацькими peer-підключеннями WireGuard шляхом розробки спеціалізованого програмного засобу з графічним інтерфейсом.
- **Об'єкт дослідження** — процеси конфігурування, збереження та адміністрування peer-підключень WireGuard.
- **Предмет дослідження** — принципи побудови, програмні компоненти та функціональні можливості системи керування WireGuard-підключеннями на базі Python і Qt.

2

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Проаналізувати особливості технології WireGuard та сучасні підходи до організації керування користувацькими peer-підключеннями в програмних системах.
2. Дослідити існуючі підходи до побудови desktop-застосунків для керування мережевими конфігураціями та обґрунтувати вибір засобів реалізації, зокрема Python і Qt.
3. Сформувати вимоги до програмної системи керування WireGuard-підключеннями, визначити її основні функціональні можливості та сценарії використання.
4. Спроєктувати архітектуру застосунку, передбачивши розподіл на модулі графічного інтерфейсу, бізнес-логіки, доступу до даних і локального збереження конфігурацій.
5. Розробити структуру локальної бази даних для збереження профілів WireGuard, а також реалізувати механізми створення, читання, редагування та видалення записів.
6. Реалізувати графічний інтерфейс користувача для створення, імпорту, редагування та перегляду профілів, а також механізми перевірки коректності введених конфігураційних даних.
7. Виконати апробацію розробленого програмного засобу на рівні основних користувацьких сценаріїв і оцінити можливості його подальшого розширення.

3

АНАЛІЗ АНАЛОГІВ

Аналог	Основа / технології	Переваги	Недоліки	Ключові відмінності від запропонованої системи
wg-easy	• Docker Compose або Podman • WireGuard + веб-інтерфейс • Серверний стек backend + frontend	• Висока функціональна готовність • Зручний веб-інтерфейс • Швидке керування клієнтами • Візуальний контроль підключень і трафіку	• Потребує серверної інфраструктури • Орієнтований переважно на адміністрування сервера • Ускладнює розгортання та супровід	• Є web/server-рішенням, а не локальним desktop-застосунком • Орієнтується на централізоване адміністрування WireGuard-сервера • Передбачає контейнерне розгортання, тоді як наша реалізація працює локально
wg-access-server	• Go (backend) • React (frontend) • gRPC-Web API • Docker, Docker Compose, Kubernetes	• Комплексне рішення для багатокористувацьких VPN-середовищ • Централізоване керування доступом • Підходить для корпоративних сценаріїв • Підтримує кілька клієнтських платформ	• Значно складніше за локальний desktop-інструмент • Потребує backend-інфраструктури • Функціональність надлишкова для персонального керування профілями	• Реалізує сервер доступу з web-інтерфейсом • Має автентифікацію, керування користувачами й пристроями • Орієнтований на багатокористувацьке середовище, а не на локальне персональне використання
WGDashboard	• Python (backend) • Vue.js (frontend) • Linux або Docker • Модель веб-дашборду	• Зручний візуальний контроль стану системи • Dashboard і засоби моніторингу • Діагностика та перегляд логів • Добре підходить для адміністрування існуючого сервера	• Потребує веб/серверного середовища • Орієнтований на адміністрування сервера • Не призначений для локального збереження та редагування профілів у desktop-форматі	• Найближчий за ідеєю моніторингу та dashboard-підходу • Водночас належить до класу web/server-рішень • Не є локальним desktop-конфігуратором профілів

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

- 1. Ініціалізація локального сховища.
- 2. Відображення профілів.
- 3. Створення та імпорт профілю.
- 4. Редагування та видалення профілю.
- 5. Валідація введених даних.
- 6. Збереження змін і оновлення інтерфейсу.

Нефункціональні вимоги:

- 1. Локальний режим роботи.
- 2. Надійність збереження даних
- 3. Зручність користувацького інтерфейсу.
- 4. Прийнятний час відгуку.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Visual Studio Code

Середовище розробки, що використовувалося для написання, редагування та супроводу програмного коду застосунку. Забезпечує зручну роботу з багатофайловим Python-проєктом.

Python

Основна мова програмування, на якій реалізовано логіку застосунку, обробку даних, взаємодію між модулями та загальну функціональність системи.

Git / GitHub

Інструменти контролю версій, що застосовувалися для збереження історії змін, керування версіями проєкту та організації спільної роботи над кодом.

Qt Designer

Засіб проєктування графічного інтерфейсу, що використовувався для створення та редагування форм, діалогових вікон і візуальних компонентів застосунку.

PySide6

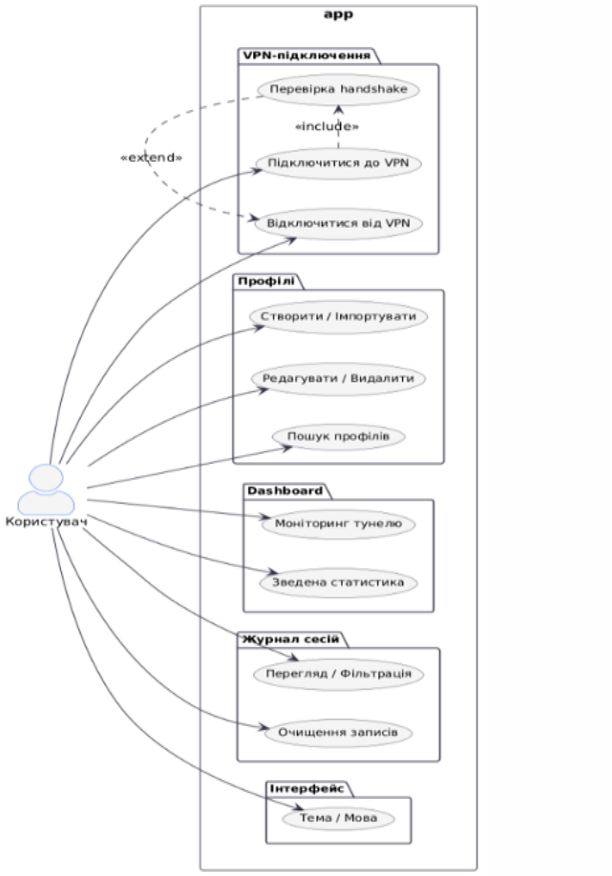
Бібліотека Python для Qt, яка застосовувалася для побудови графічного інтерфейсу користувача, обробки подій та інтеграції логіки застосунку з UI.

SQLite

Легка вбудована система керування базами даних, що використовувалася для локального збереження профілів, конфігурацій та пов'язаних записів застосунку.

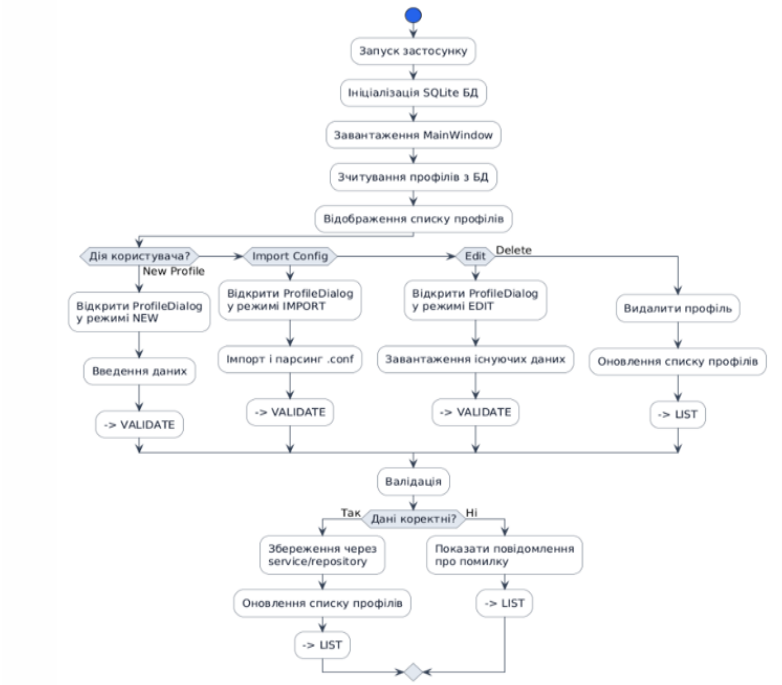
6

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



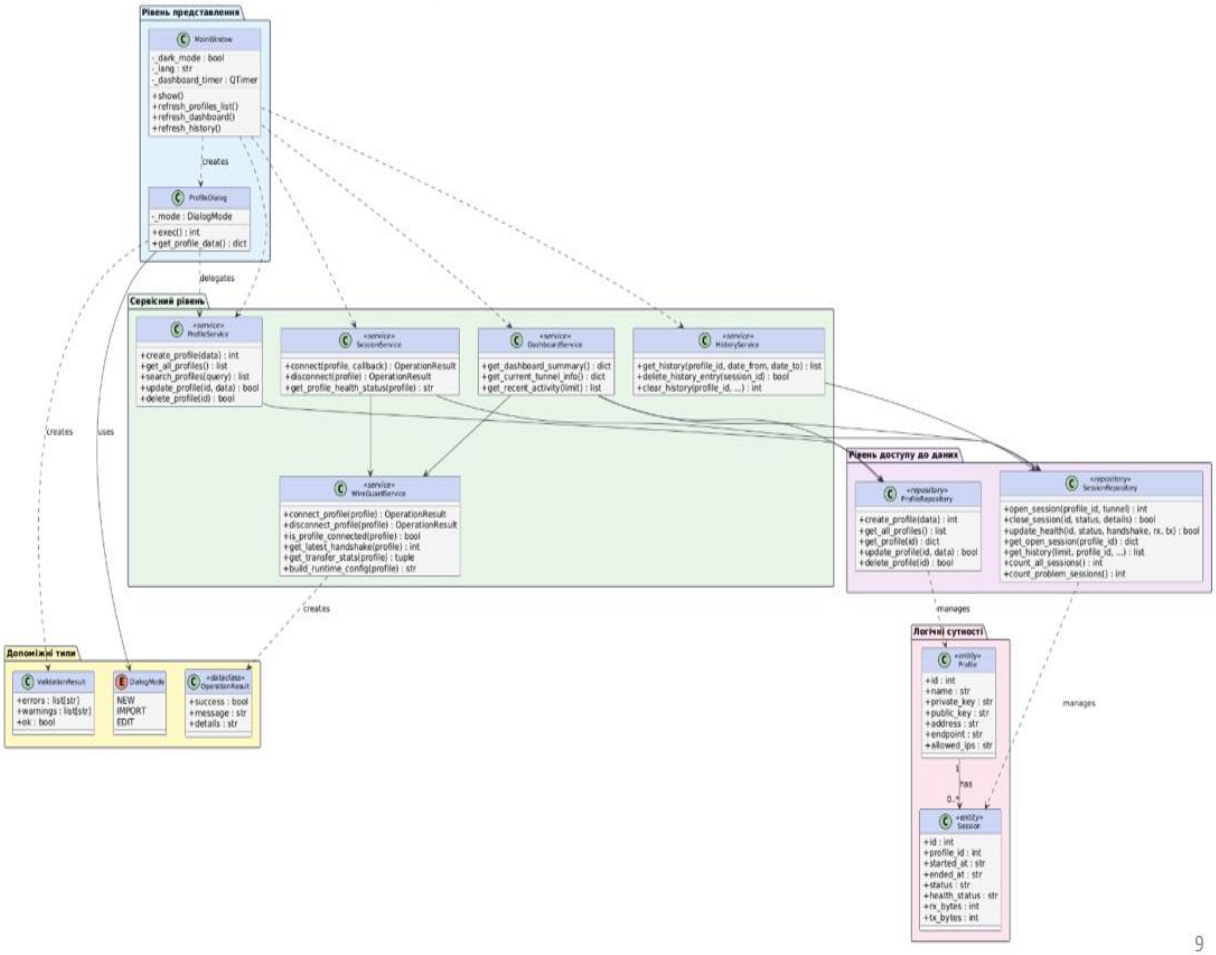
7

АЛГОРИТМ РОБОТИ ЗАСТОСУНКУ



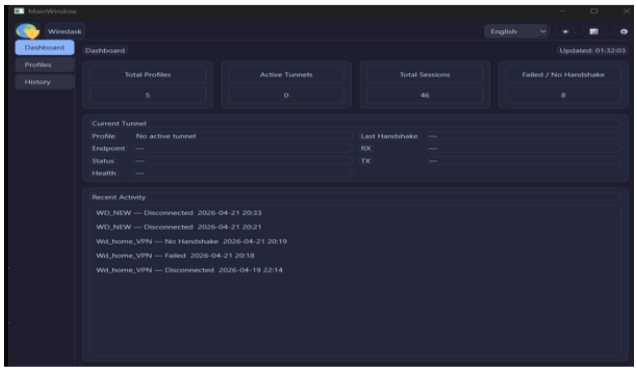
8

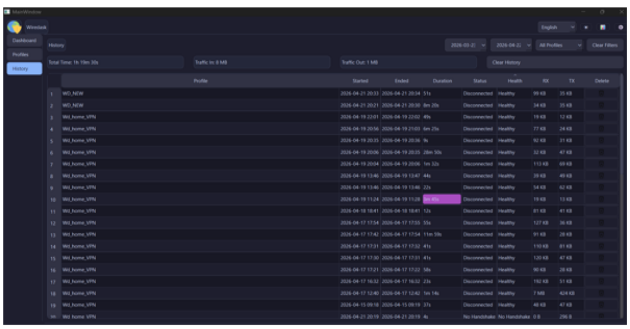
ДІАГРАМА КЛАСІВ



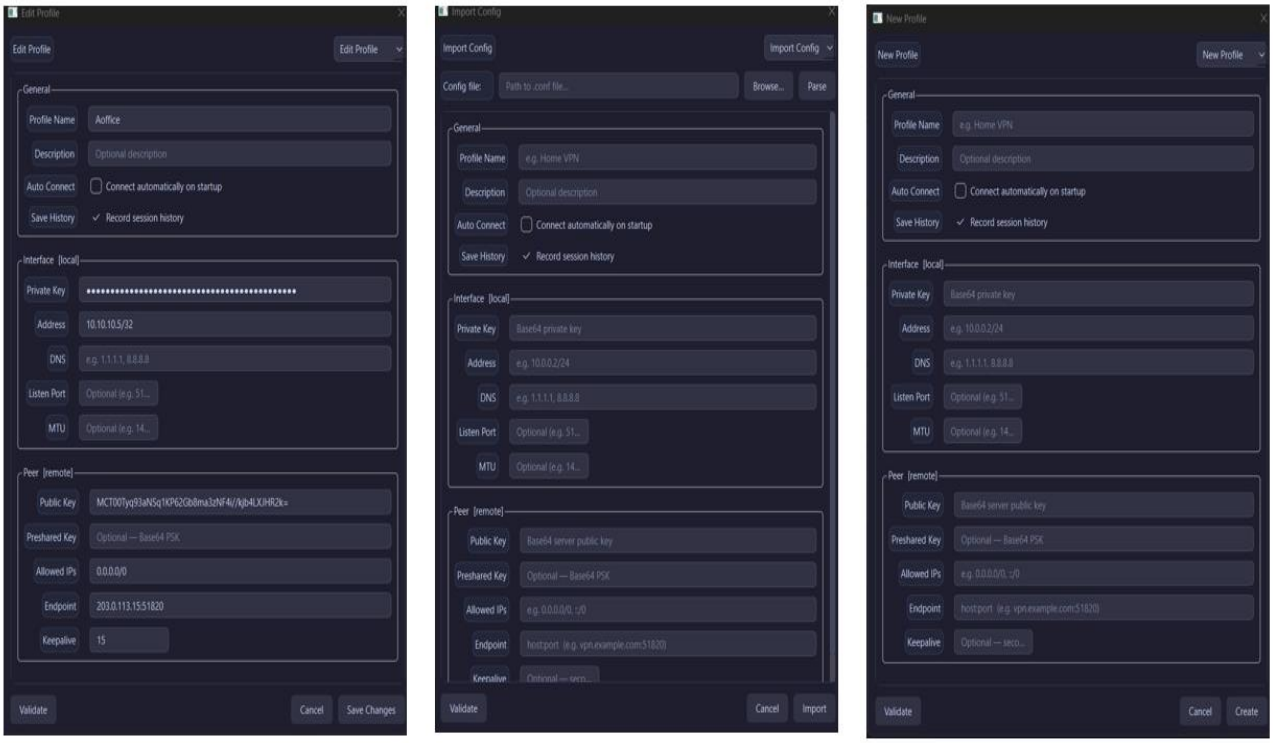
9

ЕКРАННІ ФОРМИ





ЕКРАННІ ФОРМИ



АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1) Результати роботи апробовано на VII Всеукраїнській науково-технічній конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях»(Київ, 23 квітня 2026 р., напрям «Безпека програм та даних»). Тема доповіді: « Програмна система керування WireGuard VPN-профілями:архітектурні рішення та механізми захисту інформацій» (Подано до друку).

2) Результати роботи апробовано на Всеукраїнській науково-технічній конференції «Технології цифрового розвитку: програмні системи та децентралізація» (Київ, 1-3 травня 2026 р., напрям «Програмні системи та архітектури»). Тема доповіді: «Модульна архітектура настільного додатка для керування локальними мережевими профілями » (Подано до друку).

ВИСНОВКИ

1. У ході роботи проаналізовано предметну галузь і сучасні аналоги, що дало змогу обґрунтувати доцільність локального desktop-підходу замість server/web-рішень.
2. Спроєктовано модульну архітектуру застосунку, яка охоплює рівень графічного інтерфейсу, прикладної логіки, доступу до даних і локального зберігання.
3. Реалізовано ключові функції системи: створення, імпорт, редагування, збереження та пошук профілів, підключення і відключення WireGuard-тунелю, перевірку handshake, журнал сесій, історію та інформаційну панель.
4. Результати тестування підтвердили працездатність основних сценаріїв роботи та доцільність подальшого розвитку застосунку як локальної системи керування WireGuard-профілями.

ДЯКУЮ ЗА УВАГУ!

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

```
"""
database/schema.py — DDL for all tables. Run once on
startup via init_db().
"""

from __future__ import annotations

from database.db import get_connection

_CREATE_PROFILES = """
CREATE TABLE IF NOT EXISTS profiles (
    id                  INTEGER PRIMARY KEY
AUTOINCREMENT,
    name                TEXT    NOT NULL UNIQUE,
    description         TEXT    NOT NULL DEFAULT '',
    auto_connect        INTEGER NOT NULL DEFAULT 0,
    save_history        INTEGER NOT NULL DEFAULT 1,
    private_key         TEXT    NOT NULL,
    address             TEXT    NOT NULL,
    dns                 TEXT    NOT NULL DEFAULT '',
    listen_port         TEXT    NOT NULL DEFAULT '',
    mtu                 TEXT    NOT NULL DEFAULT '',
    public_key          TEXT    NOT NULL,
    preshared_key       TEXT    NOT NULL DEFAULT '',
    allowed_ips         TEXT    NOT NULL,
    endpoint            TEXT    NOT NULL,
    persistent_keepalive TEXT   NOT NULL DEFAULT ''
);
"""

# status values:        'connected' | 'disconnected' |
'failed'
# health_status values: 'unknown'   | 'healthy'      |
'no_handshake'
# ended_at / duration_seconds are NULL while the
session is still open.
_CREATE_SESSIONS = """
CREATE TABLE IF NOT EXISTS sessions (
    id              INTEGER PRIMARY KEY
AUTOINCREMENT,
    profile_id      INTEGER NOT NULL REFERENCES
profiles(id) ON DELETE CASCADE,
    tunnel_name     TEXT    NOT NULL,
    started_at      TEXT    NOT NULL DEFAULT
(datetime('now')),
    ended_at        TEXT,
    duration_seconds INTEGER,
    status          TEXT    NOT NULL DEFAULT
'connected',
    health_status   TEXT    NOT NULL DEFAULT
'unknown',
    last_handshake_at TEXT,
    rx_bytes        INTEGER,
    tx_bytes        INTEGER,
    details         TEXT    NOT NULL DEFAULT ''
);

CREATE INDEX IF NOT EXISTS idx_sessions_profile_id ON
sessions (profile_id);
CREATE INDEX IF NOT EXISTS idx_sessions_status     ON
sessions (status);
"""

# Columns removed from profiles after the initial
schema — DROP is idempotent via try/except.
_PROFILES_MIGRATIONS: list[str] = [
    "ALTER TABLE profiles DROP COLUMN created_at",
    "ALTER TABLE profiles DROP COLUMN updated_at",
]

# Columns added/removed after the initial schema —
applied once to existing databases.
_SESSIONS_MIGRATIONS: list[str] = [
    "ALTER TABLE sessions ADD COLUMN health_status
TEXT NOT NULL DEFAULT 'unknown'",
    "ALTER TABLE sessions ADD COLUMN last_handshake_at
TEXT",
    "ALTER TABLE sessions ADD COLUMN rx_bytes
INTEGER",
    "ALTER TABLE sessions ADD COLUMN tx_bytes
INTEGER",
    "ALTER TABLE sessions DROP COLUMN created_at",
    "ALTER TABLE sessions DROP COLUMN updated_at",
]


def init_db() -> None:
    """Create all tables and apply any pending column
migrations."""
    with get_connection() as conn:
        conn.executescript(_CREATE_PROFILES +
_CREATE_SESSIONS)
        for stmt in _PROFILES_MIGRATIONS +
_SESSIONS_MIGRATIONS:
            try:
                conn.execute(stmt)
            except Exception:
                pass
```

```
"""
database/repositories/profile_repository.py — CRUD for
the profiles table.
"""

from __future__ import annotations

from typing import Optional

from database.db import get_connection

_PROFILE_FIELDS = (
    "name", "description", "auto_connect",
"save_history",
    "private_key", "address", "dns", "listen_port",
    return [_row_to_dict(r) for r in rows]


def get_profile(profile_id: int) -> Optional[dict]:
    """Return a single profile by id, or None if not
found."""
    sql = "SELECT * FROM profiles WHERE id = ?"
    with get_connection() as conn:
        row = conn.execute(sql,
(profile_id,)).fetchone()
    return _row_to_dict(row) if row else None


def update_profile(profile_id: int, data: dict) ->
bool:
```

```
"mtu",
    "public_key", "preshared_key", "allowed_ips",
"endpoint",
    "persistent_keepalive",
)


def _row_to_dict(row) -> dict:
    return dict(row)


def create_profile(data: dict) -> int:
    """Insert a new profile. Returns the new profile
id."""
    sql = """
        INSERT INTO profiles
            (name, description, auto_connect,
save_history,
             private_key, address, dns, listen_port,
mtu,
             public_key, preshared_key, allowed_ips,
endpoint, persistent_keepalive)
        VALUES
            (:name, :description, :auto_connect,
:save_history,
             :private_key, :address, :dns,
:listen_port, :mtu,
             :public_key, :preshared_key,
:allowed_ips, :endpoint, :persistent_keepalive)
    """
    with get_connection() as conn:
        cur = conn.execute(sql, {k: data.get(k, "")
for k in _PROFILE_FIELDS})
        return cur.lastrowid


def get_all_profiles() -> list[dict]:
    """Return all profiles ordered by name."""
    sql = "SELECT * FROM profiles ORDER BY name
COLLATE NOCASE"
    with get_connection() as conn:
        rows = conn.execute(sql).fetchall()
```

```
    """Update mutable fields of an existing profile.
Returns True if a row was changed."""
    sql = """
        UPDATE profiles SET
            name                = :name,
            description         = :description,
            auto_connect        = :auto_connect,
            save_history        = :save_history,
            private_key         = :private_key,
            address             = :address,
            dns                 = :dns,
            listen_port         = :listen_port,
            mtu                 = :mtu,
            public_key          = :public_key,
            preshared_key       = :preshared_key,
            allowed_ips         = :allowed_ips,
            endpoint            = :endpoint,
            persistent_keepalive =
:persistent_keepalive
        WHERE id = :_id
    """
    params = {k: data.get(k, "") for k in
_PROFILE_FIELDS}
    params["_id"] = profile_id
    with get_connection() as conn:
        cur = conn.execute(sql, params)
        return cur.rowcount > 0


def delete_profile(profile_id: int) -> bool:
    """Delete a profile by id. Returns True if a row
was deleted."""
    sql = "DELETE FROM profiles WHERE id = ?"
    with get_connection() as conn:
        cur = conn.execute(sql, (profile_id,))
        return cur.rowcount > 0
```

```
"""
services/profile_service.py — Business logic for
profile management.

Sits between ProfileDialog (UI) and profile_repository
(DB).
Validates the data dict that
ProfileDialog.get_profile_data() returns,
then delegates persistence to the repository.
"""

from __future__ import annotations

import re
import sqlite3
from typing import Optional

import database.repositories.profile_repository as
repo

# ── Validation helpers
────────────────────────────────────────────────
──
_B64_RE   = re.compile(r"^[A-Za-z0-
9+/]{42,44}={0,2}$")
_CIDR_RE  =
re.compile(r"^\d{1,3}(\.\d{1,3}){3}/\d{1,2}$")
_EP_RE    = re.compile(r"^.+:\d{1,5}$")
```

```
# ── Public API
────────────────────────────────────────────────
─────────


def _guard_integrity(exc: sqlite3.IntegrityError,
name: str) -> None:
    """Convert a SQLite UNIQUE violation into a user-
friendly ProfileValidationError."""
    msg = str(exc).lower()
    if "unique" in msg and "name" in msg:
        raise ProfileValidationError(
            f"A profile named '{name}' already exists.
Choose a different name."
        )
    raise ProfileValidationError(f"Database constraint
error: {exc}") from exc


def create_profile(data: dict) -> int:
    """
    Validate and persist a new profile.

    Returns:
        The new profile's database id.

    Raises:
        ProfileValidationError: on invalid or
duplicate data.
```

```
class ProfileValidationError(ValueError):
    """Raised when profile data fails business-level
validation."""


def _validate(data: dict) -> None:
    """Raise ProfileValidationError if required fields
are missing or malformed."""
    name = (data.get("name") or "").strip()
    if len(name) < 2:
        raise ProfileValidationError("Profile name
must be at least 2 characters.")

    private_key = (data.get("private_key") or
"").strip()
    if not _B64_RE.match(private_key):
        raise ProfileValidationError("Private key is
not valid Base64.")

    address = (data.get("address") or "").strip()
    if not _CIDR_RE.match(address):
        raise ProfileValidationError("Address must be
in CIDR notation (e.g. 10.0.0.2/24).")

    public_key = (data.get("public_key") or
"").strip()
    if not _B64_RE.match(public_key):
        raise ProfileValidationError("Public key is
not valid Base64.")

    allowed_ips = (data.get("allowed_ips") or
"").strip()
    if not allowed_ips:
        raise ProfileValidationError("Allowed IPs must
not be empty.")

    endpoint = (data.get("endpoint") or "").strip()
    if not _EP_RE.match(endpoint):
        raise ProfileValidationError("Endpoint must be
in host:port format.")

    for field in ("listen_port", "mtu",
"persistent_keepalive"):
        val = (data.get(field) or "").strip()
        if val and not val.isdigit():
            raise
ProfileValidationError(f"{field.replace('_', '
').title()} must be a number.")


def _normalise(data: dict) -> dict:
    """Return a clean copy of data ready for the
repository."""
    return {
        "name":                 data.get("name",
"").strip(),
        "description":
data.get("description", "").strip(),
        "auto_connect":
int(bool(data.get("auto_connect", False))),
        "save_history":
int(bool(data.get("save_history", True))),
        "private_key":
data.get("private_key", "").strip(),
        "address":              data.get("address",
"").strip(),
        "dns":                  data.get("dns",
"").strip(),
        "listen_port":
data.get("listen_port", "").strip(),
        "mtu":                  data.get("mtu",
"").strip(),
        "public_key":           data.get("public_key",
"").strip(),
        "preshared_key":
data.get("preshared_key", "").strip(),
    """
    _validate(data)
    clean = _normalise(data)
    try:
        return repo.create_profile(clean)
    except sqlite3.IntegrityError as exc:
        _guard_integrity(exc, clean["name"])


def get_all_profiles() -> list[dict]:
    """Return all profiles from the database, ordered
by name."""
    return repo.get_all_profiles()


def search_profiles(query: str) -> list[dict]:
    """
    Return profiles whose name or description contain
*query* (case-insensitive).
    Filters in memory after a single DB fetch — no
extra repository method needed.
    """
    q = query.strip().lower()
    if not q:
        return repo.get_all_profiles()
    return [
        p for p in repo.get_all_profiles()
        if q in p["name"].lower() or q in
p.get("description", "").lower()
    ]


def update_profile(profile_id: int, data: dict) ->
bool:
    """
    Validate and update an existing profile.

    Returns:
        True if the profile was found and updated.

    Raises:
        ProfileValidationError: on invalid or
duplicate data.
    """
    _validate(data)
    clean = _normalise(data)
    try:
        return repo.update_profile(profile_id, clean)
    except sqlite3.IntegrityError as exc:
        _guard_integrity(exc, clean["name"])


def delete_profile(profile_id: int) -> bool:
    """
    Delete a profile by id.

    Returns:
        True if the profile was found and deleted.
    """
    return repo.delete_profile(profile_id)
```

```
        "allowed_ips":
 data.get("allowed_ips", "").strip(),
        "endpoint":             data.get("endpoint",
 "").strip(),
        "persistent_keepalive":
 data.get("persistent_keepalive", "").strip(),
     }

"""
services/wireguard_service.py — WireGuard tunnel
management (Windows only).

Uses the native WireGuard for Windows CLI
(wireguard.exe):
  /installtunnelservice <config>   — install and start
a tunnel as a Windows service
  /uninstalltunnelservice <name>   — stop and remove
the tunnel service
  sc query WireGuardTunnel$<name>  — check whether the
tunnel service is running
"""

from __future__ import annotations

import os
import subprocess
import time
from dataclasses import dataclass
from datetime import datetime
from pathlib import Path


# ── Constants
─────────────────────────────────────────────────────
──────────

_CONFIGS_DIR = Path(__file__).resolve().parents[1] /
"data" / "configs"

_WIREGUARD_SEARCH_PATHS: list[Path] = [
    Path(os.environ.get("PROGRAMFILES",
r"C:\Program Files"))      / "WireGuard" /
"wireguard.exe",
    Path(os.environ.get("PROGRAMFILES(X86)",
r"C:\Program Files (x86)")) / "WireGuard" /
"wireguard.exe",
    Path(r"C:\Program Files\WireGuard\wireguard.exe"),
]

# wg.exe sits in the same directory as wireguard.exe
_WG_SEARCH_PATHS: list[Path] = [p.parent / "wg.exe"
for p in _WIREGUARD_SEARCH_PATHS]

# Polling parameters for waiting on service state
changes
_POLL_ATTEMPTS = 8
_POLL_DELAY    = 0.75   # seconds between sc query
calls


# ── Result type
─────────────────────────────────────────────────────
─────────

@dataclass
class OperationResult:
    success: bool
    message: str = ""
    details: str = ""      # raw stdout/stderr +
diagnostics

    def __bool__(self) -> bool:
        return self.success


# ── Internal helpers
    except subprocess.TimeoutExpired:
        return False, f"[{ts}] Command timed out:
{cmd_str}"
    except OSError as exc:
        return False, f"[{ts}] OS error running
'{cmd_str}': {exc}"


def _poll_for_running(tunnel_name: str) -> bool:
    """Poll ``sc query`` until the tunnel service
reaches RUNNING state."""
    service_name = f"WireGuardTunnel${tunnel_name}"
    for _ in range(_POLL_ATTEMPTS):
        ok, out = _run("sc", "query", service_name)
        if ok and "RUNNING" in out:
            return True
        time.sleep(_POLL_DELAY)
    return False


def build_runtime_config(profile: dict) -> str:
    """Render a WireGuard .conf string from a profile
dict."""
    lines: list[str] = ["[Interface]"]
    lines.append(f"PrivateKey =
{profile['private_key']}")
    lines.append(f"Address = {profile['address']}")
    if profile.get("dns"):
        lines.append(f"DNS = {profile['dns']}")
    if profile.get("listen_port"):
        lines.append(f"ListenPort =
{profile['listen_port']}")
    if profile.get("mtu"):
        lines.append(f"MTU = {profile['mtu']}")
    lines.append("")
    lines.append("[Peer]")
    lines.append(f"PublicKey =
{profile['public_key']}")
    lines.append(f"AllowedIPs =
{profile['allowed_ips']}")
    lines.append(f"Endpoint = {profile['endpoint']}")
    if profile.get("preshared_key"):
        lines.append(f"PresharedKey =
{profile['preshared_key']}")
    if profile.get("persistent_keepalive"):
        lines.append(f"PersistentKeepalive =
{profile['persistent_keepalive']}")
    return "\n".join(lines) + "\n"


def connect_profile(profile: dict) -> OperationResult:
    """
    Write the tunnel config to a stable path and
install it as a Windows service.
    """
    tunnel_name = get_tunnel_name(profile)
    conf_path   = _config_path(tunnel_name)
    exe         = _find_wireguard_exe()

    if is_profile_connected(profile):
        return OperationResult(
            success=True,
            message=f"Tunnel '{tunnel_name}' is
already connected.",
        )

    try:
```

```
_____

def _find_wireguard_exe() -> str:
    """
    Return the absolute path to wireguard.exe.

    Searches the standard WireGuard for Windows
installation directories.
    Falls back to plain 'wireguard' (assumes it is on
PATH) if not found.
    """
    for candidate in _WIREGUARD_SEARCH_PATHS:
        if candidate.exists():
            return str(candidate)
    return "wireguard"


def _find_wg_exe() -> str:
    """
    Return the absolute path to wg.exe (the WireGuard
control utility).
    """
    for candidate in _WG_SEARCH_PATHS:
        if candidate.exists():
            return str(candidate)
    return "wg"


def _config_path(tunnel_name: str) -> Path:
    """Return the stable .conf path for a tunnel,
creating the directory if needed."""
    _CONFIGS_DIR.mkdir(parents=True, exist_ok=True)
    return _CONFIGS_DIR / f"{tunnel_name}.conf"


def _run(*args: str) -> tuple[bool, str]:
    """Run a subprocess command. Returns (success,
combined_output)."""
    ts = datetime.now().strftime("%H:%M:%S")
    cmd_str = " ".join(str(a) for a in args)
    try:
        result = subprocess.run(
            args,
            capture_output=True,
            text=True,
            timeout=15,
        )
        output = (result.stdout +
result.stderr).strip()
        details = f"[{ts}] $ {cmd_str}\n{output}" if
output else f"[{ts}] $ {cmd_str}"
        return result.returncode == 0, details
    except FileNotFoundError:
        return False, f"[{ts}] Command not found:
{args[0]}"
```

```
    conf_path.write_text(build_runtime_config(profile),
encoding="utf-8")
    except OSError as exc:
        return OperationResult(
            success=False,
            message=f"Failed to write config for
'{tunnel_name}'.",
            details=str(exc),
        )

    ok, details = _run(exe, "/installtunnelservice",
str(conf_path))
    if not ok:
        return OperationResult(
            success=False,
            message=f"wireguard.exe failed to install
tunnel '{tunnel_name}'.",
            details=details,
        )

    if not _poll_for_running(tunnel_name):
        return OperationResult(
            success=False,
            message=f"Tunnel '{tunnel_name}' was
installed but did not reach RUNNING state.",
            details=details,
        )

    return OperationResult(
        success=True,
        message=f"Tunnel '{tunnel_name}' connected.",
        details=details,
    )


def is_profile_connected(profile: dict) -> bool:
    """Return True if the WireGuard tunnel for
*profile* is currently running."""
    tunnel_name  = get_tunnel_name(profile)
    service_name = f"WireGuardTunnel${tunnel_name}"
    ok, out = _run("sc", "query", service_name)
    return ok and "RUNNING" in out


def get_tunnel_name(profile: dict) -> str:
    """Return the stable WireGuard tunnel name for a
profile."""
    return f"wd_{profile['id']}"
```

```
"""
services/session_service.py — Session lifecycle
management.
"""

from __future__ import annotations

import threading
import time
from datetime import datetime, timezone
from typing import Callable

import database.repositories.session_repository as
session_repo
import services.wireguard_service as wg_service
from services.wireguard_service import OperationResult
```

```
                on_health_update(health_status)
        except Exception as exc:
            print(f"[session_service] failed to
schedule UI callback: {exc}", flush=True)


def connect(
    profile: dict,
    on_health_update: Callable[[str], None] | None =
None,
) -> OperationResult:
    """
    Bring the WireGuard tunnel up and record the
session.
    """
    profile_id  = profile["id"]
```

```
_HANDSHAKE_CHECK_DELAY = 5


def _check_handshake_async(
    profile: dict,
    session_id: int,
    on_health_update: Callable[[str], None] | None,
) -> None:
    """
    Background task: wait, sample peer health, persist
the result, then react.
    """
    try:
        time.sleep(_HANDSHAKE_CHECK_DELAY)

        ts = wg_service.get_latest_handshake(profile)
        rx, tx =
wg_service.get_transfer_stats(profile)

        if ts and ts > 0:
            last_handshake_at =
datetime.fromtimestamp(ts, tz=timezone.utc).strftime(
                "%Y-%m-%d %H:%M:%S"
            )
            health_status = "healthy"
        else:
            last_handshake_at = None
            health_status = "no_handshake"

        session_repo.update_health(
            session_id,
            health_status=health_status,
            last_handshake_at=last_handshake_at,
            rx_bytes=rx,
            tx_bytes=tx,
        )

        if health_status == "no_handshake":
            wg_service.disconnect_profile(profile)
            session_repo.close_session(
                session_id,
                status="no_handshake",
                details=(
                    f"Auto-disconnected: no WireGuard
handshake received within "
                    f"{_HANDSHAKE_CHECK_DELAY}
seconds."
                ),
            )

    except Exception as exc:
        print(f"[session_service]
_check_handshake_async error: {exc}", flush=True)
        health_status = "no_handshake"

    if on_health_update is not None:
        try:
            from PySide6.QtCore import QTimer
            from PySide6.QtWidgets import QApplication
            app = QApplication.instance()
            if app is not None:
                QTimer.singleShot(0, app, lambda:
on_health_update(health_status))
            else:
```

```
    tunnel_name = wg_service.get_tunnel_name(profile)

    if wg_service.is_profile_connected(profile):
        return OperationResult(
            success=True,
            message=f"Tunnel '{tunnel_name}' is
already connected.",
        )

    result = wg_service.connect_profile(profile)

    if result.success:
        session_id =
session_repo.open_session(profile_id, tunnel_name)
        threading.Thread(
            target=_check_handshake_async,
            args=(profile, session_id,
on_health_update),
            daemon=True,
        ).start()
    else:
        session_id =
session_repo.open_session(profile_id, tunnel_name)
        session_repo.close_session(
            session_id,
            status="failed",
            details=result.details,
        )

    return result


def disconnect(profile: dict) -> OperationResult:
    """Tear the WireGuard tunnel down and close the
open session row."""
    profile_id  = profile["id"]
    tunnel_name = wg_service.get_tunnel_name(profile)

    if not wg_service.is_profile_connected(profile):
        open_session =
session_repo.get_open_session(profile_id)
        if open_session:
            session_repo.close_session(
                open_session["id"],
                status="disconnected",
                details="Session closed: tunnel was
already stopped.",
            )
        return OperationResult(
            success=True,
            message=f"Tunnel '{tunnel_name}' is
already disconnected.",
        )

    result = wg_service.disconnect_profile(profile)

    if result.success:
        open_session =
session_repo.get_open_session(profile_id)
        if open_session:
            session_repo.close_session(
                open_session["id"],
                status="disconnected",
                details=result.details,
            )

    return result
```

```
"""
database/repositories/session_repository.py — CRUD for
the sessions table.
"""

from __future__ import annotations
```

```
            health_status     = ?,
            last_handshake_at = ?,
            rx_bytes          = ?,
            tx_bytes          = ?
        WHERE id = ?
    """
    with get_connection() as conn:
```

```
from typing import Optional

from database.db import get_connection


def _row_to_dict(row) -> dict:
    return dict(row)


def open_session(profile_id: int, tunnel_name: str) ->
int:
    """Insert a new session row with
status='connected' and started_at=now."""
    sql = """
        INSERT INTO sessions (profile_id, tunnel_name,
status, started_at)
        VALUES (?, ?, 'connected', datetime('now'))
    """
    with get_connection() as conn:
        cur = conn.execute(sql, (profile_id,
tunnel_name))
        return cur.lastrowid


def close_session(
    session_id: int,
    *,
    status: str = "disconnected",
    details: str = "",
) -> bool:
    """Close an open session by setting ended_at,
duration_seconds, status, and details."""
    sql = """
        UPDATE sessions SET
            ended_at         = datetime('now'),
            duration_seconds = CAST(
                (julianday(datetime('now')) -
julianday(started_at)) * 86400 AS INTEGER
            ),
            status  = ?,
            details = ?
        WHERE id = ? AND ended_at IS NULL
    """
    with get_connection() as conn:
        cur = conn.execute(sql, (status, details,
session_id))
        return cur.rowcount > 0


def get_open_session(profile_id: int) ->
Optional[dict]:
    """Return the currently open session for
*profile_id* (ended_at IS NULL), or None."""
    sql = """
        SELECT * FROM sessions
        WHERE profile_id = ? AND ended_at IS NULL
        ORDER BY started_at DESC
        LIMIT 1
    """
    with get_connection() as conn:
        row = conn.execute(sql,
(profile_id,)).fetchone()
    return _row_to_dict(row) if row else None


def update_health(
    session_id: int,
    *,
    health_status: str,
    last_handshake_at: str | None,
    rx_bytes: int,
    tx_bytes: int,
) -> bool:
    """Write peer health data into an open session
row."""
    sql = """
        UPDATE sessions SET
        cur = conn.execute(sql, (health_status,
last_handshake_at, rx_bytes, tx_bytes, session_id))
        return cur.rowcount > 0


def get_history(
    limit: int | None = None,
    profile_id: int | None = None,
    date_from: str | None = None,
    date_to: str | None = None,
) -> list[dict]:
    """
    Return closed sessions (ended_at IS NOT NULL) with
optional filters.
    """
    conditions: list[str] = ["s.ended_at IS NOT NULL"]
    params: list = []

    if profile_id is not None:
        conditions.append("s.profile_id = ?")
        params.append(profile_id)
    if date_from:
        conditions.append("s.started_at >= ?")
        params.append(date_from)
    if date_to:
        conditions.append("s.started_at <= ?")
        params.append(f"{date_to} 23:59:59")

    where = " AND ".join(conditions)
    limit_clause = "LIMIT ?" if limit is not None else
""
    if limit is not None:
        params.append(limit)

    sql = f"""
        SELECT s.*, p.name AS profile_name
        FROM sessions s
        LEFT JOIN profiles p ON p.id = s.profile_id
        WHERE {where}
        ORDER BY s.started_at DESC
        {limit_clause}
    """
    with get_connection() as conn:
        rows = conn.execute(sql, params).fetchall()
    return [_row_to_dict(r) for r in rows]


def count_problem_sessions() -> int:
    """Count sessions whose lifecycle ended in a
failure."""
    with get_connection() as conn:
        return conn.execute(
            "SELECT COUNT(*) FROM sessions WHERE
status IN ('failed', 'no_handshake')"
        ).fetchone()[0]


def get_open_session_with_profile() -> Optional[dict]:
    """Return the most recently opened session that is
still open."""
    sql = """
        SELECT s.*, p.name AS profile_name, p.endpoint
AS profile_endpoint
        FROM sessions s
        LEFT JOIN profiles p ON p.id = s.profile_id
        WHERE s.ended_at IS NULL
        ORDER BY s.started_at DESC
        LIMIT 1
    """
    with get_connection() as conn:
        row = conn.execute(sql).fetchone()
    return _row_to_dict(row) if row else None
```