

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВЕБСИСТЕМИ
КЕРУВАННЯ ФЕРМОЮ FTP-ВУЗЛІВ З ОРКЕСТРАТОРОМ ПРОЦЕСІВ ТА
АДМІНІСТРУВАННЯМ КОРИСТУВАЧІВ
DEVELOPMENT OF A WEB-BASED SYSTEM FOR MANAGING AN FTP
NODE FARM WITH PROCESS ORCHESTRATION AND USER
ADMINISTRATION

Виконав: студент 4 курсу, групи КНТ-113сп
Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ДАНЧУК А.О.

(ПРИЗВИЩЕ та ініціали)

Керівник СТЕПАНЕНКО О.О.

(ПРИЗВИЩЕ та ініціали)

Рецензент ДЬЯЧУК Т.С.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ДАНЧУКА Андрія Олеговича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розроблення програмного забезпечення вебсистеми керування фермою FTP-вузлів з оркестратором процесів та адмініструванням користувачів. Development of a Web-based System for Managing an FTP Node Farm with Process Orchestration and User Administration.

керівник проєкту (роботи) к.т.н., доцент, СТЕПАНЕНКО Олександр Олексійович,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 07 ” квітня 2026 року № 139

2. Строк подання студентом проєкту (роботи) 01 червня 2026 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Матеріали і методи. 3. Розробка програмного забезпечення. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	СТЕПАНЕНКО О.О, доцент		
Нормоконтроль	БЄЛОВА А.В., асистент		

7. Дата видачі завдання «20» квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Андрій ДАНЧУК
(Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Олександр СТЕПАНЕНКО
(Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 102 с., 23 табл., 37 рис., 2 дод., 25 джерел.

FTP-ВУЗОЛ, ДЕСКТОПНИЙ ЗАСТОСУНОК, ЛОКАЛЬНИЙ ВЕБІНТЕРФЕЙС, PYTHON, AIОНТТР, SQLITE, PYFTPLIB, PYWEBVIEW, КЕРУВАННЯ КОРИСТУВАЧАМИ, ПРАВА ДОСТУПУ, ЖУРНАЛ ПОДІЙ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ.

Об'єкт дослідження – процес програмування функцій адміністрування та розгортання FTP-сервісів.

Предмет дослідження – десктопна вебсистема FTPFarmManager для створення, налаштування, запуску, зупинки та моніторингу FTP-вузлів.

Мета роботи – розробити програмне забезпечення, яке дозволить адміністратору централізовано керувати кількома FTP-вузлами, користувачами, правами доступу та журналом подій через зручний інтерфейс, що зменшує кількість ручних дій, спрощує контроль конфігурації та підвищує зручність адміністрування файлового обміну.

Матеріали, методи та технічні засоби: структурне, об'єктно-орієнтоване та асинхронне програмування, технологія організації FTP-сервісів, локальний вебінтерфейс для адміністрування, мова програмування Python, база даних SQLite, бібліотеки aiohttp, aisqlite, pyftplib та pywebview, HTML, CSS, JavaScript, персональний комп'ютер під керуванням операційної системи Microfoft Windows.

Результати. Створено програмний Windows-застосунок для централізованого адміністрування FTP-вузлів підприємства. Система забезпечує створення та налаштування FTP-вузлів, керування FTP-користувачами й правами доступу, запуск, зупинку та перезапуск сервісів,

ведення журналу подій, перевірку параметрів конфігурації, моніторинг основних показників, а також імпорт і експорт налаштувань.

Висновки. Розроблено програмну систему для керування FTP-вузлами. У застосунку реалізовано роботу з вузлами, користувачами, правами доступу, журналом подій, метриками та налаштуваннями системи. Також було перевірено роботу основних функцій застосунку та можливість централізованого керування FTP-сервісами.

Галузь використання – адміністрування FTP-серверів, організація локального обміну файлами, навчальні програмні проєкти з серверного адміністрування.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 102 pages, 23 tables, 37 figures, 2 appendixes, 25 sources.

FTP NODE, DESKTOP APPLICATION, LOCAL WEB INTERFACE, PYTHON, AIOHTTP, SQLITE, PYFTPLIB, PYWEBVIEW, USER MANAGEMENT, ACCESS RIGHTS, EVENT LOG, SOFTWARE.

Object of study is the process of administering FTP services and organizing user access to file resources.

Subject of study is the FTPFarmManager desktop software system for creating, configuring, launching, stopping and monitoring FTP nodes.

The purpose of the work is to develop software that allows an administrator to centrally manage several FTP nodes, users, access rights and the event log through a convenient local interface.

Materials, methods, and tools: structured, object-oriented and asynchronous programming, FTP service technology, local web interface for administration, Python programming language, SQLite database, aiohttp,aiosqlite, pyftplib and pywebview libraries, HTML, CSS and JavaScript, personal computer under the control of the Microsoft Windows operating system.

Results. A Windows desktop application for centralized administration of FTP nodes has been created. The system provides creation and configuration of FTP nodes, user and access rights management, starting, stopping and restarting services, event logging, configuration checking, monitoring of basic system indicators, as well as import and export of settings.

Conclusions. The FTPFarmManager software system has been developed. It simplifies the process of FTP service administration and allows managing several nodes from one local interface. The main functions of the application have

been tested, including node creation, access configuration, user management, service startup and event logging.

The field of use is FTP server administration, organization of local file exchange and educational software projects related to server administration.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	10
Вступ.....	11
1 Аналіз предметної області.....	13
1.1 Призначення FTP та задачі файлового обміну.....	13
1.2 Проблеми ручного адміністрування FTP-серверів.....	14
1.3 Вимоги до системи керування FTP-вузлами.....	15
1.4 Огляд засобів для керування FTP-серверами.....	16
1.5 Висновки за розділом 1	17
2 Матеріали і методи.....	19
2.1 Вибір мови програмування	19
2.2 Вибір бібліотек та засобів для розробки вебсистеми	20
2.3 Порівняльний аналіз готових FTP-рішень і розробленої системи.....	21
2.4 Вибір бібліотек та засобів для розробки вебсистеми.....	22
2.5 Висновки за розділом 2	28
3 Розробка програмного забезпечення.....	30
3.1 Загальна архітектура системи	30
3.2 Схема запуску застосунку	31
3.3 Компонентна структура програмної системи	31
3.4 Реалізація REST API системи	32
3.5 Реалізація бази даних.....	33
3.6 Реалізація підсистеми керування FTP-вузлами	34
3.7 Реалізація FTP-користувачів та прав доступу.....	34
3.8 Реалізація моніторингу та журналу подій	35
3.9 Реалізація захисту керуючих запитів	36
3.10 Реалізація експорту та імпорту конфігурації	36
3.11 Висновки за розділом 3	37
4 Експлуатація, тестування та експериментальне дослідження програми	38

4.1 Інструкції користувача	38
4.2 Інструкції адміністратора	40
4.3 Оптимізація програмного забезпечення	42
4.4 Опис інтерфейсу програми	45
4.5 Висновки за розділом 4	49
Перелік джерел посилання	52
ДОДАТОК А Текст програми	55
ДОДАТОК Б Слайди презентації	87

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	– Application Programming Interface;
CSS	– Cascading Style Sheets;
FTP	– File Transfer Protocol;
FTP-вузол	– окремий FTP-сервіс, який має власну назву, IP-адресу, порт, кореневий каталог, параметри підключення та список користувачів із правами доступу;
HTTP	– HyperText Transfer Protocol;
IP	– Internet Protocol;
JSON	– JavaScript Object Notation;
SQL	– Structured Query Language;
UI	– User Interface;
URL	– Uniform Resource Locator;
БД	– база даних;
ОС	– операційна система;
ПК	– персональний комп'ютер;
ПЗ	– програмне забезпечення;
HTML	– HyperText Markup Language.

ВСТУП

У сучасних умовах підприємства та організації постійно працюють з великою кількістю файлів та даних. Для обміну документами, резервними копіями та іншими файлами не рідко використовуються FTP-сервери. Ручне керування серверами незручне, особливо якщо потрібно працювати з кількома FTP-вузлами, користувачами, правами доступу та журналом подій. Через це виникає потреба у системі, яка дозволяє керувати FTP-сервером зручніше. Така система має спрощувати роботу адміністратора, дозволяти додавати та налаштовувати вузли, керувати користувачами, змінювати права доступу та переглядати події в системі.

Метою роботи є дослідження та розробка вебсистеми для керування фермою FTP-вузлів. Для досягнення цієї мети передбачається виконання таких завдань як: аналіз предметної області, визначення основних вимог до системи, вибір засобів розробки, проектування структури програмного забезпечення, реалізація основних модулів системи та перевірка їх роботи.

У процесі виконання роботи використовуються мова програмування Python, база даних SQLite, бібліотеки aiohttp,aiosqlite, pyftplib та pywebview. Для створення вебінтерфейсу використовуються HTML, CSS та JavaScript.

Практична цінність роботи полягає в тому, що розроблена система вже використовується на підприємстві. Вона допомагає адміністратору керувати FTP-вузлами, користувачами, правами доступу та переглядати журнал подій з одного вебінтерфейсу, що підвищує зручність адміністрування та зменшує кількість ручних дій.

Наукова новизна цієї роботи полягає у створенні вебсистеми, яка поєднує керування FTP-вузлами, користувачами, правами доступу, журналом подій та налаштуваннями в одному вікні програми.

У результаті виконання роботи розроблено серверний застосунок з вебінтерфейсом для централізованого керування FTP-вузлами.

У рамках дослідження було проаналізовано предметну область, визначено вимоги до системи, обрано засоби розробки, реалізовано основні функції програмного забезпечення та проведено перевірку роботи.

Сфера застосування результатів цієї роботи – адміністрування FTP-серверів, організація файлового обміну та керування доступом користувачів до файлових ресурсів як локально, так і за потреби з відкритого інтернету.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Призначення FTP та задачі файлового обміну

FTP (File Transfer Protocol) – це мережевий протокол для передавання файлів між клієнтом і сервером. Сучасна специфікація протоколу описана в документі RFC 959 [1]. Протокол належить до прикладного рівня та призначений для роботи з файлами у комп'ютерних мережах.

Першу специфікацію протоколу було опубліковано у 1971 році. Пізніше FTP адаптували для роботи з TCP/IP, після чого він почав використовуватися у сучасних комп'ютерних мережах [2].

Основна ідея FTP проста. Користувач підключається до сервера, проходить автентифікацію за логіном і паролем, а за потреби ще додаткові перевірки. Після цього він може працювати з дозволеним каталогом. Залежно від виданих прав користувач може переглядати файли, завантажувати їх із сервера або передавати файли на сервер.

FTP використовує два типи з'єднань. Перше потрібне для керування роботою сервера. Через нього передаються команди: увійти в систему, відкрити каталог, переглянути список файлів або почати завантаження. Друге з'єднання використовується для передавання самих даних, тобто файлів або списку файлів у каталозі [1].

У внутрішній мережі підприємств FTP-сервер або FTP-з'єднання може бути корисним для простих та середніх задач файлового обміну. Через нього можна передавати звіти, архіви, резервні копії, технічні матеріали та службові документи.

Перевага FTP полягає у простоті. Для нього існує багато клієнтських програм. Адміністратор може швидко створити окремий файловий вузол для конкретної задачі або групи користувачів, налаштувавши конфігурацію та параметри запуску конкретного вузла.

При цьому FTP має важливе обмеження. У звичайному режимі він не шифрує логін, пароль і дані. Для захисту можуть використовуватися розширення безпеки FTP [3] або FTP через TLS [4].

У цій роботі FTP розглядається як основа для створення керованих файлових вузлів без обмеження в кількості. Головна увага приділяється не самому протоколу, а зручному адмініструванню декількох FTP-вузлів одночасно.

1.2 Проблеми ручного адміністрування FTP-серверів

Ручне адміністрування FTP-серверів може бути зручним лише тоді, коли сервер один. Якщо вузлів кілька, ситуація ускладнюється. Адміністратор має стежити за портами, каталогами, користувачами та правами доступу як до каталогу, так і до самих правил редагування.

Перша проблема пов'язана з конфігурацією. Кожен вузол має власну адресу, порт, кореневий каталог і обмеження. Якщо ці дані зберігаються окремо, легко допустити помилку та втратити доступ до потрібного вузла.

Друга проблема стосується користувачів. Користувача потрібно створити, задати пароль і видати права. Якщо користувач має доступ до різних вузлів, ручне налаштування стає незручним, тому що це збільшує кількість окремих дій для адміністратора.

Третя проблема – це контроль стану. Вузол може бути запущений, зупинений або завершитися з помилкою. Без єдиного журналу адміністратору складніше зрозуміти, що саме сталося. Окремий FTP-сервер у разі помилки може просто завершити роботу, а без нормального відображення подій знайти причину збою складніше.

Ще одна проблема – відсутність єдиного місця керування вузлами. Частина дій на FTP виконується через конфігураційні файли. Зазвичай помилки та журнали перевіряються через консоль. Це забирає багато часу, бо

в журналі може бути багато звичайних дій користувачів, а не тільки помилки. Через це процес пошуку проблем у журналах сервера ускладнюється.

Через це вигідніше використовувати окрему систему керування. Вона дає адміністратору один інтерфейс як з комп'ютера, так і з ноутбука. Через нього можна бачити вузли, змінювати параметри, запускати сервери та переглядати події без вимкнення сервера.

1.3 Вимоги до системи керування FTP-вузлами

Після аналізу предметної області було визначено основні вимоги до системи. Система повинна дозволяти створювати FTP-вузли. Для кожного вузла потрібно задавати назву, адресу, порт, кореневий каталог, обмеження підключень і користувачів, які можуть працювати з цим вузлом.

Система повинна підтримувати запуск, зупинку та перезапуск вузлів. Ці дії мають виконуватися через інтерфейс програми. Адміністратор не повинен вручну запускати кожен сервер через консоль, бо це має виконувати сама система, це спрощує керування серверами.

Окремою вимогою є робота з користувачами. Користувач має створюватися в системі один раз. Після цього для нього за потреби можна призначати доступ до одного або кількох FTP-вузлів.

Права доступу мають бути зрозумілими. Адміністратор повинен бачити, до якого вузла має доступ користувач. Також він має бачити дозволений каталог і набір прав цього користувача.

Також важливо мати журнал подій. У ньому повинні фіксуватися запуск і зупинка вузлів, помилки, зміни налаштувань або інші службові повідомлення.

Система також має бути простою у використанні та запуску. Вона повинна працювати локально, без складної інфраструктури та без окремого сервера з базами даних, щоб спростити встановлення програми.

1.4 Огляд засобів для керування FTP-серверами

Для порівняння було розглянуто кілька готових засобів для роботи з FTP-серверами та файловим обміном. Це FileZilla Server, Wing FTP Server, Cerberus FTP Server, CrushFTP та vsftpd.

Популярний FileZilla Server з FTP/FTPS-сервером. Офіційна документація вказує, що він підтримує FTP і FTPS та може працювати на різних операційних системах [5].

Дуже функціональним є Wing FTP Server підтримує FTP, FTPS, HTTP, HTTPS і SFTP. Також він має вебінтерфейс для адміністрування, моніторингу сесій і перегляду стану сервера [6].

Програмне забезпечення Cerberus FTP Server орієнтований на корпоративне використання. У ньому є вебадміністрування та можливість віддаленого доступу до налаштувань через браузер [7].

Програма CrushFTP є багатофункціональним рішенням для передавання файлів. Воно підтримує різні захищені протоколи, зокрема FTPS, SFTP, SCP і HTTPS [8].

Також vsftpd є простим і поширеним FTP-сервером для UNIX-подібних систем. Його основні переваги – швидкість, стабільність і орієнтація на безпеку [9].

FileZilla Server є зручним варіантом для окремого FTP-сервера. Його плюси включають відомість і підтримку FTPS. Мінус для цієї роботи – менший акцент на керуванні кількома окремими вузлами з одного простого інтерфейсу.

Wing FTP Server має ширші можливості. Він підтримує кілька протоколів і має вебадміністрування. Але для невеликої внутрішньої системи такий продукт може бути надлишковим.

Cerberus FTP Server більше підходить для корпоративного використання. Його перевага – сильніші засоби адміністрування. Мінус – складність і більша кількість налаштувань.

CrushFTP також має багато функцій. Він підтримує захищені протоколи і різні способи автентифікації. Для простої задачі керування FTP-вузлами така система може бути занадто складною.

vsftpd є легким і стабільним FTP-сервером. Його мінус у межах цієї роботи – потреба у ручному налаштуванні. Також він не дає готового інтерфейсу для керування всіма FTP-вузлами.

Порівняння показує, що готові рішення мають більше функцій. Вони часто підтримують захищені протоколи, корпоративну авторизацію, повідомлення та моніторинг.

Розроблена система простіша. Її головна перевага – керування кількома FTP-вузлами з одного інтерфейсу, який має бути зрозумілим для користувача. Також вона може бути швидко змінена під потреби конкретного підприємства або задачі.

Головні недоліки розробленої системи – відсутність SFTP та FTPS, простіша система безпеки і менша кількість корпоративних можливостей. Але для внутрішнього службового інструмента така простота може бути перевагою, оскільки система швидше налаштовується і не потребує зайвої складності.

1.5 Висновки за розділом 1

У першому розділі було розглянуто призначення FTP та задачі файлового обміну. Було визначено, що FTP підходить для простого передавання файлів у внутрішній мережі.

Також було описано проблеми ручного адміністрування. Основні труднощі пов'язані з налаштуваннями вузлів, користувачами, правами доступу та контролем стану.

На основі цього було сформовано вимоги до системи. Система має дозволяти керувати FTP-вузлами, користувачами, правилами доступу та журналом подій з одного зрозумілого інтерфейсу.

Огляд аналогів показав, що готові продукти мають ширші можливості, але не завжди мають зручний інтерфейс для простого користувача. Водночас розроблена система є простішою, зрозумілішою з боку інтерфейсу і краще підходить для конкретної внутрішньої задачі підприємства.

2 МАТЕРІАЛИ І МЕТОДИ

2.1 Вибір мови програмування

Для розробки програмної частини було обрано Python. Ця мова підходить для службових застосунків, у яких потрібно працювати з мережею, файлами, базою даних і окремими процесами одночасно. При роботі з цією мовою програмування ці можливості потрібні для створення панелі керування FTP-вузлами, обробки запитів і запуску серверів [10].

Для системи FTPFarmManager це важливо, оскільки застосунок має не тільки відображати інтерфейс, а й запускати FTP-вузли, перевіряти порти, зберігати конфігурацію та вести журнал подій.

Для вибору мови програмування було розглянуто кілька варіантів. Порівняння та причини вибору мови програмування подано в таблиці 2.1.

Таблиця 2.1 – Вибір мов програмування

Мова	Переваги для задачі	Обмеження для задачі	Висновок
Python	Простий синтаксис, багато бібліотек для мережі, файлів, SQLite, HTTP API та FTP-сервера; підтримка асинхронного програмування.	Нижча швидкість порівняно з C++ або C#, але для локального адміністративного інструмента це не є критичним.	Обрано як основну мову розробки.
C#	Добре підходить для Windows-застосунків, має розвинені засоби створення інтерфейсу.	Для реалізації вбудованого FTP-сервера та локального вебінтерфейсу потрібно більше окремих рішень; складніша зміна логіки для невеликого проекту.	Не обрано.
Java	Кросплатформеність, стабільна екосистема, багато серверних бібліотек.	Більша кількість службового коду та складніше розгортання локального desktop-застосунку для простої адміністративної задачі.	Не обрано.
JavaScript, Node.js	Зручний для веброзробки та API, добре працює з асинхронними запитами.	Менш зручний для вбудованого керування локальними FTP-вузлами та файловою системою Windows без додаткової складності.	Не обрано.
C++	Висока швидкість та повний контроль над системними ресурсами.	Значно складніша розробка, більше ризиків помилок пам'яті, надлишкова складність для задачі адміністрування FTP-вузлів.	Не обрано.

Отже, Python було обрано через поєднання простоти розробки, наявності готових бібліотек, підтримки асинхронної роботи та достатньої продуктивності для локальної системи адміністрування. У межах цієї роботи швидкість створення та підтримки програмного забезпечення важливіша за максимальну обчислювальну продуктивність.

Ще однією причиною вибору Python є підтримка асинхронного програмування. У системі одночасно можуть виконуватися різні операції: обробка запиту з інтерфейсу, звернення до бази даних, запуск або зупинка FTP-вузла. Асинхронність допомагає виконувати такі дії без повної зупинки роботи програми, та без зависання інтерфейсу.

Також Python зручний для поділу коду на окремі модулі. Це спрощує підтримку проєкту, бо роботу з API, базою даних, FTP-серверами та інтерфейсом можна розмістити окремо, забезпечивши їхню взаємодію як єдиної системи.

У результаті написаний код легше змінювати, доповнювати і перевіряти. Ще одним з плюсів вибору Python є можливість переносити застосунок на інші операційні системи в подальшому.

2.2 Вибір бібліотек та засобів для розробки вебсистеми

Для створення вебсистеми було використано кілька основних бібліотек та засобів: aiohttp, SQLite,aiosqlite, pyftplib, pywebview, HTML, CSS і JavaScript.

Бібліотека aiohttp використовується для створення HTTP API. Вона підтримує серверну і клієнтську частини HTTP-з'єднань [11]. У цьому проєкті aiohttp потрібна для приймання запитів від інтерфейсу та передавання їх до внутрішньої логіки програми.

Для збереження даних було обрано SQLite. Вона не потребує встановлення окремого сервера або бази даних, тому підходить для локального адміністративного інструмента. За офіційним описом SQLite є

самодостатнім SQL-рушієм, який часто вбудовується безпосередньо у застосунки [12].

Для асинхронної роботи інтерфесу з SQLite використовується `aiosqlite` [13]. Вона дає змогу виконувати SQL-запити без блокування інших частин програми, в цьому випадку асинхронність потрібна, бо програма запускає в фоні FTP-сервери і оркестратор для них.

FTP-функціонал реалізується за допомогою `pyftplib` [14], [15]. Ця бібліотека дозволяє створювати FTP-сервери всередині Python-застосунку, задавати користувачів, домашні каталоги, права доступу, адресу і порт запуску.

Інтерфейс побудовано з використанням `ruwebview` модуля [16]. З його допомогою вебсторінки відкриваються у вікні як звичайне програмне забезпечення. Візуальна частина створюється за допомогою HTML, CSS і JavaScript. HTML відповідає за структуру сторінок, CSS за оформлення, а JavaScript за взаємодію з елементами інтерфейсу.

2.3 Порівняльний аналіз готових FTP-рішень і розробленої системи

Для порівняння було розглянуто готові FTP-рішення та розроблену систему `FTPFarmManager`. Порівняння виконувалося за призначенням, перевагами, обмеженнями та доцільністю використання для задачі керування кількома FTP-вузлами. Основні відмінності між цими підходами подано в табл. 2.2.

Таблиця 2.2 – Порівняння засобів для керування FTP-серверами

Засіб або підхід	Характеристика засобу		Оцінка для цієї роботи	
	Основне призначення	Переваги	Обмеження	Висновок
1	2	3	4	5
Готові FTP-рішення	Готові сервери для файлового обміну	Багато функцій, підтримка різних протоколів	Можуть бути складними або надлишковими	Підходять як аналоги

Кінець таблиці 2.2

1	2	3	4	5
FTPFarmManager	Керування кількома FTP-вузлами	Простий локальний інтерфейс, журнал подій	Немає SFTP/FTPS	Доцільний для цієї роботи

Як видно з табл. 2.2, готові FTP-рішення мають більше функцій і підтримують різні протоколи, але для невеликої внутрішньої системи частина цих можливостей може бути зайвою і може заважати при використанні. Розроблена система FTPFarmManager простіша, але вона краще підходить для задачі керування кількома FTP-вузлами з одного локального інтерфейсу.

2.4 Вибір бібліотек та засобів для розробки вебсистеми

Для реалізації програмної системи FTPFarmManager потрібно було обрати такі засоби, які дозволяють створити локальний застосунок з вебінтерфейсом, HTTP API, базою даних і вбудованим FTP-сервером. Основними критеріями вибору були простота розробки, можливість локального запуску, підтримка асинхронної роботи, відсутність потреби в складній серверній інфраструктурі та сумісність з мовою Python.

Для серверної частини було розглянуто кілька варіантів реалізації HTTP API. Порівняння наведено в таблиці 2.3.

Таблиця 2.3 – Порівняння засобів для реалізації HTTP API

Засіб	Переваги	Обмеження	Висновок
1	2	3	4
aiohttp	Підтримує асинхронну обробку HTTP-запитів, добре працює з asyncio, підходить для локального API, дозволяє створювати маршрути для взаємодії інтерфейсу з логікою програми.	Потребує розуміння асинхронного програмування.	Обрано, оскільки система має одночасно обробляти запити інтерфейсу, працювати з базою даних і керувати FTP-вузлами.

Кінець таблиці 2.3

1	2	3	4
Flask	Простий і зрозумілий фреймворк для створення вебзастосунків.	Основна модель роботи не є асинхронною, тому для цієї задачі може бути менш зручною при паралельній роботі з вузлами.	Не обрано, бо для FTPFarmManager важлива асинхронна робота.
FastAPI	Сучасний фреймворк для API, підтримує асинхронність і автоматичну документацію.	Має більше можливостей, ніж потрібно для локального адміністративного інструмента.	Не обрано, бо для цієї роботи він є надлишковим.
Власний HTTP-сервер на стандартних бібліотеках Python	Не потребує додаткових залежностей.	Потрібно вручну реалізовувати маршрути, обробку запитів, помилки та структуру API.	Не обрано через зайву складність підтримки.

Отже, для реалізації HTTP API було обрано aiohttp. Ця бібліотека підходить для системи, у якій інтерфейс надсилає запити до локального ядра програми, а саме ядро паралельно працює з базою даних, журналом подій та FTP-вузлами.

Для збереження даних було розглянуто кілька варіантів баз даних і способів зберігання інформації. Порівняння наведено в таблиці 2.4.

Таблиця 2.4 – Порівняння засобів для збереження даних

Засіб	Переваги	Обмеження	Висновок
1	2	3	4
SQLite таaiosqlite	Не потребує окремого сервера, зберігає дані в одному файлі, підходить для локального desktop-застосунку,aiosqlite дозволяє виконувати SQL-запити асинхронно.	Не призначена для великої кількості одночасних користувачів і складних корпоративних навантажень.	Обрано, бо система працює локально і не потребує окремого сервера баз даних.
PostgreSQL	Потужна серверна база даних, підходить для великих систем і багатьох користувачів.	Потрібно встановлювати та налаштовувати окремий сервер баз даних.	Не обрано, бо для локального адміністративного інструмента це надлишково.

Кінець таблиці 2.4

1	2	3	4
MySQL MariaDB	Поширені серверні бази даних, підходять для вебсистем.	Також потребують окремого сервера і додаткового адміністрування.	Не обрано через зайву складність розгортання.
JSON-файли	Простий спосіб зберігання налаштувань.	Незручно працювати зі зв'язками між вузлами, користувачами, правами доступу та журналом подій.	Не обрано, бо система потребує структурованих таблиць і зв'язків між даними.

Для FTPFarmManager було обрано SQLite разом з aiosqlite. SQLite дозволяє зберігати інформацію про вузли, користувачів, правила доступу, події та ревізії в одному локальному файлі. Це спрощує встановлення програми та її перенесення на інший комп'ютер. Використання aiosqlite дозволяє виконувати запити до бази даних без повного блокування роботи програми.

Для реалізації FTP-функціоналу було розглянуто варіанти, наведені в таблиці 2.5.

Таблиця 2.5 – Порівняння засобів для реалізації FTP-функціоналу

Засіб	Переваги	Обмеження	Висновок
pyftplib	Дозволяє створювати FTP-сервер безпосередньо в Python-застосунку, підтримує користувачів, права доступу, обмеження підключень, події та налаштування сервера.	Для захищених режимів потрібно окремо допрацьовувати конфігурацію.	Обрано, бо бібліотека дозволяє запускати FTP-вузли прямо з програми.
ftplib	Входить до стандартних можливостей Python і дозволяє працювати з FTP як клієнт.	Не призначена для створення FTP-сервера.	Не обрано, бо системі потрібен саме FTP-сервер, а не FTP-клієнт.
Зовнішній FTP-сервер, наприклад FileZilla Server	Готове рішення з власними можливостями адміністрування.	Важче інтегрувати в один локальний застосунок, потрібно окремо керувати зовнішньою програмою.	Не обрано через те що мета роботи створити власну систему керування FTP-вузлами.

Кінець таблиці 2.5

Засіб	Переваги	Обмеження	Висновок
vsftpd	Стабільний FTP-сервер для UNIX-подібних систем.	Більше підходить для Linux-серверів, а не для локального Windows-застосунку з власним інтерфейсом.	Не обрано через меншу відповідність формату цієї роботи.

Для реалізації FTP-вузлів було обрано `pyftplib`. Ця бібліотека дозволяє створювати FTP-сервери програмно, задавати адресу, порт, кореневий каталог, користувачів і права доступу. Завдяки цьому кожен FTP-вузол можна запускати, зупиняти та перезапускати безпосередньо з інтерфейсу `FTPFarmManager`.

Для створення desktop-інтерфейсу було розглянуто кілька варіантів, які наведено в таблиці 2.6.

Таблиця 2.6 – Порівняння засобів для створення інтерфейсу

Засіб	Переваги	Обмеження	Висновок
pywebview HTML CSS JavaScript	Дозволяє відкрити вебінтерфейс у нативному вікні desktop-застосунку, не потребує окремого браузера для користувача, добре поєднується з Python.	Інтерфейс залежить від webview-компонента операційної системи.	Обрано, бо програма має виглядати як desktop-застосунок, але використовувати зручний вебінтерфейс.
Electron	Дозволяє створювати desktop-застосунки на HTML, CSS і JavaScript.	Потребує Node.js, має більший розмір застосунку та складнішу інтеграцію з Python-ядром.	Не обрано, бо для цієї роботи він є занадто важким рішенням.
PyQt	Дає можливість створити повноцінний desktop-інтерфейс.	Потрібно окремо створювати всі елементи інтерфейсу.	Не обрано, бо вебінтерфейс зручніше редагувати для цієї системи.
Tkinter	Входить до стандартних засобів Python і підходить для простих віконних програм.	Має простіший вигляд і менш зручний для створення сучасної панелі адміністратора.	Не обрано через обмежені можливості інтерфейсу.
Відкриття інтерфейсу тільки в браузері	Найпростіший варіант для вебінтерфейсу.	Користувачу потрібно окремо відкривати браузер і адресу локального сервера.	Не обрано, бо програма має запускатися як окремий застосунок.

Для інтерфейсу обрано pywebview, HTML, CSS і JavaScript. Це дозволяє створити локальний застосунок із вебінтерфейсом, який відкривається у звичайному вікні та надсилає запити до API. Загальний вибір засобів наведено в таблиці 2.7.

Таблиця 2.7 – Підсумковий вибір бібліотек та засобів розробки

Частина системи	Обраний засіб	Причина вибору
HTTP API	aiohttp	Асинхронна обробка запитів і зручна інтеграція з Python-ядром.
База даних	SQLite	Локальне зберігання даних без окремого сервера.
Асинхронна робота з БД	aiosqlite	Можливість виконувати SQL-запити асинхронно.
FTP-сервер	pyftplib	Створення та запуск FTP-вузлів безпосередньо з Python-коду.
Desktop-вікно	pywebview	Відкриття вебінтерфейсу у вікні локального застосунку.
Візуальна частина	HTML, CSS, JavaScript	Просте створення сторінок, форм, таблиць і кнопок.
Оркестрація вузлів	Python, asyncio, threading	Керування запуском, зупинкою та контролем стану [17], [18].
Середовище розробки	Visual Studio Code	Зручна робота з Python, HTML, CSS, JavaScript, вбудований термінал і просте керування файлами проєкту.

Таким чином, обрані бібліотеки та засоби відповідають поставленій задачі. aiohttp забезпечує локальний HTTP API, SQLite та aiosqlite відповідають за збереження даних, pyftplib дозволяє створювати FTP-вузли, а pywebview дає змогу відкрити вебінтерфейс у форматі desktop-застосунку. Це дозволяє реалізувати систему без складної інфраструктури, окремого сервера баз даних або зовнішнього FTP-сервера.

Програма FTPFarmManager складається з кількох основних частин, які відповідають за інтерфейс, обробку запитів, збереження даних і запуск FTP-вузлів. Основні модулі системи подано в табл. 2.8.

Таблиця 2.8 – Основні модулі системи FTPFarmManager

Модуль системи	Опис модуля		Реалізація модуля	
	Призначення	Основні функції	Використані засоби	Результат
Інтерфейс користувача	Керування системою через вікно застосунку	відображення вузлів, користувачів, журналу подій і налаштувань	HTML, CSS, JavaScript, pywebview	адміністратор працює з програмою без консолі
HTTP API	Зв'язок між інтерфейсом і логікою програми	приймання запитів, передавання команд, повернення даних	aiohttp	Комірка
База даних	Збереження інформації системи	збереження вузлів, користувачів, прав доступу, налаштувань	SQLite,aiosqlite	дані не губляться після перезапуску програми
FTP-модуль	Створення та запуск FTP-вузлів	запуск, зупинка, перезапуск вузлів, робота з портами і каталогами	pyftplib	система може створювати окремі FTP-сервіси
Оркестратор вузлів	Керування станом FTP-вузлів	контроль запуску, зупинки, помилок і активних процесів	Python	кілька вузлів керуються з одного місця
Журнал подій	Фіксація дій і помилок	запис запусків, зупинок, змін налаштувань і збоїв	Python, SQLite	адміністратор може переглядати історію роботи системи

Як видно з табл. 2.8, система поділена на окремі модулі. Такий поділ спрощує розробку і підтримку програми, бо кожна частина відповідає за свою задачу. Інтерфейс використовується для взаємодії з адміністратором, HTTP API передає команди до логіки програми, база даних зберігає інформацію, а FTP-модулі і оркестратор відповідають за роботу FTP-вузлів.

Для написання та редагування програмного коду було використано середовище розробки Visual Studio Code. Воно підходить для цього проєкту, оскільки є легким, зручним для роботи з Python-файлами, HTML, CSS і JavaScript, має вбудований термінал, пошук по проєкту та підтримку розширень. Для FTPFarmManager це важливо, бо код системи поділений на кілька модулів, а інтерфейс складається з окремих файлів HTML, CSS, JS. Порівняння середовищ розробки наведено в таблиці 2.9.

Таблиця 2.9 – Порівняння середовищ розробки

Середовище розробки	Переваги	Обмеження	Висновок
Visual Studio Code	Легке середовище, підтримує Python, HTML, CSS, JavaScript, має вбудований термінал, зручний пошук по файлах і підтримку розширень.	Частина можливостей потребує встановлення розширень.	Обрано як основне середовище розробки, бо воно зручне для проекту з Python-ядром і вебінтерфейсом.
PyCharm	Добре підходить для Python-розробки, має багато інструментів для роботи з кодом.	Більш важке середовище, частина можливостей доступна у професійній версії.	Не обрано, бо для цієї роботи достатньо простішого середовища.
IDLE	Входить до стандартного набору Python і не потребує додаткового встановлення.	Незручне для роботи з великим проектом, багатьма файлами та вебінтерфейсом.	Не обрано через обмежені можливості.
Visual Studio	Потужне середовище для великих проектів, особливо на C# та C++.	Надлишкове для невеликого Python-застосунку.	Не обрано через зайву складність.

Отже, для розробки було використано Visual Studio Code, оскільки воно дозволяє зручно працювати з Python-кодом, файлами інтерфейсу, структурою проекту та запуском програми через термінал. Для цієї роботи таке середовище є достатнім і не створює зайвої складності під час розробки.

2.5 Висновки за розділом 2

У другому розділі обрано мову програмування та основні засоби розробки. Python використовується як головна мова, оскільки він підходить для роботи з мережею, файлами, базами даних і окремими процесами завдяки швидкості написання та редагування коду, реалізації ідеї і його гнучкості.

Для HTTP API обрано aiohttp, для локального збереження даних SQLite, для асинхронних SQL-запитів aiosqlite, для запуску FTP-серверів pyftplib, а для відкриття інтерфейсу у вікні, як застосунок, – бібліотека pywebview. Візуальна частина реалізується за допомогою HTML, CSS і JavaScript.

Середовищем розробки було обрано Visual Studio Code, оскільки воно зручне для роботи з Python-кодом, файлами HTML, CSS і JavaScript, має вбудований термінал та підтримує розширення для розробки.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Загальна архітектура системи

Застосунок FTPFarmManager реалізовано як локальну вебсистему у форматі desktop-застосунку. Адміністратор працює не з консоллю, а з вікном програми, у якому відкривається вебінтерфейс. Саме інтерфейс надсилає HTTP-запити до Core API.

Загальна архітектура складається з трьох основних рівнів. Першого інтерфейсного рівня, серверного рівня та рівня підсистем і даних. Інтерфейсний рівень містить ruwebview та Web UI. Серверний рівень містить Core API з використанням aiohttp. Рівень підсистем і даних містить SQLite, NodeManager, FarmLogger та FTP-вузли на базі бібліотеки pyftplib. Детально показано який рівень за що відповідає у таблиці 3.1.

Таблиця 3.1 – Порівняння засобів для керування FTP-серверами

Рівень системи	Компоненти	Призначення
Інтерфейсний рівень	ruwebview, HTML, CSS, JavaScript	Відображення панелі адміністратора, форм, таблиць, журналу подій і показників стану.
Серверний рівень	Core API, aiohttp	Приймання запитів від інтерфейсу, перевірка токена, передавання команд до внутрішніх модулів.
Рівень даних	SQLite,aiosqlite	Збереження вузлів, FTP-користувачів, правил доступу, подій і ревізій.
Рівень керування вузлами	NodeManager, FtpNode, pyftplib	Запуск, зупинка, перезапуск і контроль FTP-вузлів.
Рівень журналювання	FarmLogger, таблиця events	Фіксація подій, попереджень, помилок і службових повідомлень.

Такий поділ дозволяє відокремити інтерфейс від основної логіки. Якщо потрібно змінити вигляд сторінок, це не потребує повної зміни роботи FTP-вузлів. Якщо потрібно допрацювати логіку запуску вузлів, це можна зробити у NodeManager та FtpNode без переписування всього інтерфейсу.

3.2 Схема запуску застосунку

Запуск програми починається з файлу `run.py`, який викликає головну функцію адміністративної панелі. Далі система перевіряє наявність файлу `settings.ini`. Якщо файл налаштувань відсутній, відкривається майстер першого запуску, у якому адміністратор обирає порт для локального Core API.

Після вибору порту програма зберігає налаштування, створює `token.txt` і запускає ядро системи. Core API стартує у фоновому асинхронному циклі, після чого відкривається основне вікно `pywebview` з адміністративною панеллю. Повні етапи запуску програми показано в таблиці 3.2.

Таблиця 3.2 – Етапи запуску вебзастосунку

Етап запуску	Що виконується	Результат
1	Запуск <code>run.py</code>	Викликається головна функція програми.
2	Перевірка <code>settings.ini</code>	Визначається, чи виконувалося початкове налаштування.
3	Майстер першого запуску	Адміністратор обирає вільний порт для локального API.
4	Створення файлів	<code>settings.ini</code> та <code>token.txt</code> .
5	Старт Core API	Запускається <code>aiohhttp</code> -сервер, база даних, журнал і менеджер вузлів.
6	Відкриття Admin UI	У <code>pywebview</code> відкривається локальна панель адміністратора.

3.3 Компонентна структура програмної системи

Код поділено на окремі модулі. Кожен модуль виконує власну задачу, що спрощує підтримку та подальше просте розширення системи. Детально про модулі та їх ролі в застосунку описано в таблиці 3.3.

Таблиця 3.3 – Модулі та їх ролі в системі

Файл або модуль	Роль у системі
<code>run.py</code>	Головна точка запуску застосунку.
<code>adminpanel.app.py</code>	Запускає майстер першого старту, Core API та вікно <code>pywebview</code> .
<code>core.app.py</code>	Ініціалізує базу даних, журнал, <code>NodeManager</code> і <code>aiohhttp</code> -застосунок.

Кінець таблиці 3.3

Файл або модуль	Роль у системі
api.py	Містить маршрути REST API для вузлів, FTP-користувачів, правил доступу, логів, метрик, імпорту й експорту налаштувань і шляхів.
db.py	Створює таблиці SQLite та виконує SQL-запити.
node_manager.py	Керує життєвим циклом FTP-вузлів і watchdog-перевіркою.
ftp_node.py	Створює окремий FTP-сервер за допомогою ruftplib.
logging.py	Веде буфер подій і записує службові повідомлення до БД.
security.py	Містить функції генерації солі, хешування паролів і безпечного порівняння токенів.
config.py	Працює з settings.ini, startup.ini та token.txt.
utils.py	Містить допоміжні функції перевірки портів, створення каталогів і нормалізації шляхів.
webui.app	Містить HTML, CSS і JavaScript-файли інтерфейсу адміністратора.

Компоненти взаємодіють через Core API. Інтерфейс надсилає запит до API, а API перевіряє токен, читає або змінює дані в SQLite, викликає NodeManager для операцій з FTP-вузлами та передає інформацію про події до FarmLogger.

3.4 Реалізація REST API системи

REST API є центральною точкою взаємодії між інтерфейсом адміністратора та внутрішньою логікою програми. Усі керуючі запити проходять через API. Це дозволяє не змішувати HTML-інтерфейс із логікою запуску вузлів, роботою з базою даних і перевіркою параметрів. Методи та всі endpoint зв'язки показано в таблиці 3.4.

Таблиця 3.4 – Зв'язки методів у API

Група endpoint	Приклади методів	Призначення
Nodes	GET /api/v1/nodes, POST /api/v1/nodes, POST /api/v1/nodes/{id}/start	Створення, редагування, запуск, зупинка та перезапуск FTP-вузлів.
Users	GET /api/v1/users, POST /api/v1/users, PUT /api/v1/users/{id}	Створення та підтримка FTP-облікових записів.
Access Rules	GET /api/v1/access_rules, POST /api/v1/access_rules, DELETE /api/v1/access_rules/{id}	Призначення прав доступу до каталогів FTP-вузлів.
Logs	GET /api/v1/logs	Отримання журналу подій.
Metrics	GET /api/v1/metrics	Отримання показників dashboard: кількість вузлів, активні сесії, помилки.

Кінець таблиці 3.4

Група endpoint	Приклади методів	Призначення
Wizard	GET /api/v1/wizard/port_free	Перевірка вільного порту під час першого налаштування.
Import та Export	POST /api/v1/import_config, POST /api/v1/export_config	Збереження та перенесення конфігурації системи.

Під час створення нового вузла інтерфейс спочатку надсилає параметри для перевірки. API перевіряє адресу, порт і шлях. Якщо порт зайнятий або параметри некоректні, адміністратор отримує повідомлення про помилку. Якщо перевірка успішна, вузол зберігається у таблиці nodes, а NodeManager перечитує конфігурацію.

3.5 Реалізація бази даних

Для збереження даних використовується база даних SQLite. База даних зберігається у файлі data.app.db [19]. Це дозволяє не встановлювати окремий сервер баз даних і спрощує перенесення системи на інший комп'ютер. Повна реалізація бд показана в таблиці 3.5.

Таблиця 3.5 – Детальна реалізація бази даних

Таблиця	Ключові поля	Призначення
nodes	id, name, bind_ip, port, root_path, autostart, max_cons	Зберігання конфігурації FTP-вузлів.
users	id, username, password_hash, salt, ftp_password_plain, is_active	Зберігання FTP-облікових записів та даних для автентифікації.
access_rules	id, user_id, node_id, rel_path, can_write	Зв'язок FTP-користувача з вузлом, каталогом і типом доступу.
events	id, ts, level, source, message, node_id	Журнал подій системи.
revisions	id, ts, entity_type, entity_id, action, payload_json	Аудит змін конфігурації.

Модель даних побудована так, щоб один FTP-вузол міг мати багато правил доступу, а один FTP-користувач міг отримувати доступ до різних вузлів. Правило access_rules визначає, до якого вузла і до якої папки має доступ конкретний обліковий запис.

3.6 Реалізація підсистеми керування FTP-вузлами

Керування FTP-вузлами реалізовано через модулі NodeManager та FtpNode. FtpNode відповідає за створення конкретного FTP-сервера, а NodeManager відповідає за завантаження вузлів із бази, запуск, зупинку, перезапуск і контроль стану. Як виглядає повна структура і організація вузлів можна побачити в таблиці 3.6.

Таблиця 3.6 – Детальна реалізація бази даних

Параметр вузла	Призначення
name	Назва FTP-вузла, яка відображається в інтерфейсі адміністратора.
bind_ip	IP-адреса, до якої прив'язується FTP-сервіс.
Port	Порт, на якому запускається FTP-вузол.
root_path	Кореневий каталог файлового вузла.
max_cons	Загальна максимальна кількість підключень.
max_cons_per_ip	Максимальна кількість підключень з однієї IP-адреси.
idle timeout	Час очікування неактивної сесії.
dl_limit_kbps ul_limit_kbps	Обмеження швидкості завантаження та вивантаження для файлів [20].
Autostart	Ознака автоматичного запуску вузла під час старту програми.

Під час запуску вузла NodeManager формує список FTP-користувачів і правил доступу, після чого передає його до FtpNode. У FtpNode створюється FTPServer, налаштовується Handler, підключається DummyAuthorizer та задаються обмеження підключень [15], [20]. Якщо для вузла встановлено обмеження швидкості, використовується ThrottledDTPHandler.

3.7 Реалізація FTP-користувачів та прав доступу

У системі потрібно розрізняти адміністратора програми та FTP-користувачів. Адміністратор працює з панеллю керування, створює вузли, налаштовує права та переглядає журнал. А FTP-користувач має обліковий запис, який використовується для підключення до конкретного FTP-вузла окремо.

Модель доступу побудована за принципом користувач до вузла та до папки. Це означає, що правило доступу пов'язує конкретний FTP-обліковий запис із конкретним FTP-вузлом і каталогом у межах кореневої папки вузла, щоб не виходити за рамки дозволеного каталогу. Модель доступу користувача до папки FTP-вузла показано в таблиці 3.7.

Таблиця 3.7 – Модель доступу користувача до FTP-вузла

Режим доступу	can_write	Права у pyftplib	Для чого
Read only	0	elr	Перегляд каталогу, читання та завантаження файлів.
Read-write	1	elradfmwMT	Читання, запис, створення, перейменування та видалення файлів.

Перед створенням правила доступу шлях перевіряється функцією `normalize_inside_root()` [21]. Вона обчислює абсолютний шлях і перевіряє, щоб він не виходив за межі `root_path` відповідного вузла. Це потрібно, щоб FTP-користувач не отримав доступ до сторонніх каталогів файлової системи.

3.8 Реалізація моніторингу та журналу подій

Для адміністратора підприємства важливо не тільки запускати FTP-вузли, а й бачити їх стан. Для цього в системі реалізовано головну дошку, метрики та журнал подій. Дані про кількість вузлів, запущені вузли, активні сесії та помилки за останній час отримуються через API та відображаються в інтерфейсі.

Журнал подій реалізовано через `FarmLogger`. Події спочатку додаються до внутрішнього буфера, а потім зберігаються в таблиці подій `events`. У журналі використовуються рівні `INFO`, `WARN` та `ERROR`. Це дозволяє відокремлювати звичайні службові повідомлення від попереджень і помилок. Детально подивитись побудову та структуру журналу можна у таблиці 3.8.

Таблиця 3.8 – Події та їх призначення у журналі

Джерело події	Приклад події	Призначення
Core API	Запуск ядра, обробка запитів, помилки конфігурації.	Контроль роботи серверної частини.
NodeManager	Запуск, зупинка, перезапуск або збій вузла.	Контроль життєвого циклу FTP-вузлів.
FtpNode	Підключення, помилки входу, аварійні ситуації.	Діагностика роботи конкретного FTP-сервісу.
Web UI	Дії адміністратора через інтерфейс.	Зручне відстеження операцій керування.

3.9 Реалізація захисту керуючих запитів

Оскільки система є локальним адміністративним інструментом, у ній реалізовано базові механізми захисту. Під час першого запуску створюється файл з токеном `token.txt`. Значення токена передається у вебінтерфейс і додається до керуючих HTTP-запитів у заголовку `X-Auth-Token` [22].

На стороні API кожен захищений запит проходить перевірку функцією `_require_token()`. Якщо токен відсутній або не збігається зі збереженим значенням, API повертає помилку 401. Порівняння токена виконується через безпечне зіставлення рядків, щоб зменшити ризик некоректної перевірки [23].

Адміністративний пароль не зберігається у відкритому вигляді. Для нього генерується індивідуальна сіль, після чого пароль хешується за допомогою криптографічного стандарту PBKDF2-HMAC-SHA256 [24], [25]. У базі даних зберігаються `password_hash` і `salt`. Окремо варто зазначити, що FTP-пароль у поточній реалізації зберігається локально окремо, оскільки метод `DummyAuthorizer` у бібліотеці `pyftplib` потребує звичайний пароль для авторизації FTP-підключення. Це є технічним обмеженням, цього проекту, тому застосунок працює тільки локально і під наглядом адміністратора.

3.10 Реалізація експорту та імпорту конфігурації

Для перенесення або резервного збереження налаштувань у системі реалізовано експорт та імпорт конфігурації. Під час експорту програма формує JSON-структуру з даними про вузли, FTP-користувачів і правила доступу. Такий файл можна зберегти як резервну копію або використати для перенесення конфігурації на інший комп'ютер.

Під час імпорту цільові таблиці очищуються, після чого дані з JSON завантажуються назад у SQLite. Після імпорту виконується оновлення списку вузлів у NodeManager, щоб система працювала вже з новою конфігурацією. Що робить імпорт та експорт детальніше можна побачити в таблиці 3.9.

Таблиця 3.9 – Операції з експорту та імпорту налаштувань

Операція	Що виконується	Результат
Експорт	Зчитуються таблиці nodes, users та access_rules.	Формується JSON з основними налаштуваннями.
Імпорт	Дані з JSON записуються до БД, після чого виконується load_nodes().	Система перечитує оновлену конфігурацію.

3.11 Висновки за розділом 3

У третьому розділі було описано безпосередню розробку програмного забезпечення FTPFarmManager. Розглянуто загальну архітектуру системи, схему запуску, компонентну структуру, REST API, базу даних, бібліотечні та самописані модулі керування FTP-вузлами, модель прав доступу, журнал подій, захист керуючих запитів та механізм експорту й імпорту конфігурації.

Розроблена структура дозволяє адміністратору підприємства керувати кількома FTP-вузлами з одного локального інтерфейсу, а внутрішній поділ на модулі спрощує підтримку та подальше розширення програми.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Інструкції користувача

При першому запуску програми для роботи з програмою, користувач запускає програму. Якщо застосунок відкривається з вихідного коду, запуск виконується через файл `run.py`, або `FTPFarmManager.exe`. Після цього відкривається вікно `FTPFarmManager`.

Під час першого запуску може з'явитися майстер вибору порту. Він потрібен для того, щоб локальна панель керування відкривалася на вільному порті. Якщо запропонований порт підходить, користувач зберігає його. Якщо порт зайнятий, можна вибрати інший зі списку. Після цього програма створює службові файли налаштувань і відкриває основне вікно.

У лівій частині вікна розміщене меню. Через нього користувач переходить до сторінки огляду, сторінки FTP-вузлів, користувачів, доступів, журналу подій та інформації про програму. Таке меню зроблено для того, щоб не шукати потрібні дії в різних місцях, тобто для зручності.

На сторінці огляду відображається загальний стан системи. Там видно кількість створених FTP-вузлів, кількість запущених вузлів, активні сесії та помилки за останній час. Ця сторінка потрібна для швидкої перевірки, чи працює система нормально.

Для створення нового FTP-вузла користувач переходить на сторінку FTP-вузлів і відкриває форму додавання. У формі вказується назва вузла, IP-адреса, порт, коренева папка та обмеження підключень. Перед збереженням бажано перевірити порт, щоб не створити вузол на вже зайнятому значенні.

Після збереження вузол з'являється у списку. Його можна запустити, зупинити, перезапустити, змінити або видалити. Якщо були змінені порт, коренева папка або доступи, вузол краще перезапустити. Так простіше переконатися, що нові параметри вже застосовані.

Користувачі додаються на відповідній сторінці користувачів. Для нового запису вказується логін, пароль і статус активності. Якщо обліковий запис вимкнений, такий користувач не повинен отримувати доступ до FTP-вузлів.

Права доступу налаштовуються на сторінці доступу. Там потрібно вибрати користувача, FTP-вузол, папку всередині кореневого каталогу та режим доступу. Якщо потрібен тільки перегляд і завантаження файлів, використовується режим читання. Якщо користувачу потрібно ще створювати або змінювати файли, обирається режим читання і запису.

Журнал подій використовується для перевірки роботи програми. У ньому відображається запуск ядра, запуск і зупинка вузлів, попередження та помилки. Якщо FTP-вузол не запускається або користувач не може підключитися, спочатку варто переглянути саме журнал. Детально основні дії користувача описано в таблиці 4.1.

Таблиця 4.1 – Основні дії користувача у програмі

Дія	Де виконується	Що потрібно зробити	Результат
Перевірити стан системи	Огляд	Відкрити сторінку огляду і переглянути показники вузлів.	Видно, скільки вузлів створено і які з них запуснені.
Створити FTP-вузол	FTP-вузли	Заповнити назву, адресу, порт, кореневу папку і зберегти форму.	Новий вузол додається до списку.
Запустити вузол	Огляд	Натиснути кнопку запуску біля потрібного вузла.	FTP-сервіс починає працювати на вказаному порту.
Створити користувача	Користувачі	Ввести логін, пароль і статус активності.	Користувач з'являється у таблиці.
Надати доступ	Доступи	Вибрати користувача, вузол, папку і тип прав.	Створюється правило доступу.
Перевірити події	Журнал	Відкрити журнал або скористатися фільтром рівнів.	Можна побачити останні дії та помилки.
Зберегти конфігурацію	Про програму	Виконати експорт налаштувань.	Налаштування можна перенести або зберегти як копію.

Якщо користувач не може підключитися до FTP-вузла, потрібно перевірити кілька простих речей. Спочатку треба подивитися, чи запущений сам вузол. Далі перевіряється адреса, порт і права доступу для цього

користувача. Якщо доступ був змінений вже після запуску вузла, потрібно перезапустити вузол.

Завершення роботи виконується закриттям головного вікна. Після цього програма зупиняє ядро, закриває активні вузли та завершує роботу з базою даних. Створені вузли, користувачі і правила доступу залишаються в SQLite, тому після наступного запуску їх не потрібно створювати заново.

4.2 Інструкції адміністратора

Адміністратор відповідає за початкове налаштування програми, контроль службових файлів, перевірку портів і резервне копіювання. Програма зроблена як локальний застосунок, тому для її роботи не потрібно встановлювати окремий сервер баз даних.

Для запуску з вихідного коду на комп'ютері має бути встановлений Python і потрібні бібліотеки проекту. Після запуску файлу run.py відкривається адміністративна панель. Якщо програма зібрана як окремий застосунок, користувач запускає готовий файл програми, а внутрішні модулі стартують автоматично, тому і наявність Python на комп'ютері не є обов'язковою.

Під час першого запуску створюються основні службові файли. У settings.ini зберігається порт локального інтерфейсу та шлях до бази даних. У token.txt зберігається локальний токен для запитів до API. У startup.ini може зберігатися список вузлів, які потрібно запускати автоматично. Основні дані системи знаходяться у файлі data.app.db. Для чого саме ці файли призначені, описано у дивитись таблицю 4.2.

Таблиця 4.2 – Службові файли програми

Файл або папка	Призначення	Що потрібно враховувати
settings.ini	Зберігає порт, каталог даних і шлях до БД.	Не варто видаляти бо видаляться основні налаштування.

Кінець таблиці 4.2

Файл або папка	Призначення	Що потрібно враховувати
token.txt	Містить токен для захисту локальних API-запитів.	Файл не можна передавати стороннім користувачам.
startup.ini	Зберігає вузли, які запускаються разом із програмою.	Після зміни автозапуску список може оновлюватися.
Data.app.db	Основна база даних з вузлами, користувачами, правилами і журналом.	Цей файл бажано копіювати при резервному копіюванні.
Webui.app	Файли інтерфейсу HTML, CSS і JavaScript.	Без цієї папки панель керування не відобразиться правильно.

Перед відкриттям FTP-вузла для інших комп'ютерів на підприємстві адміністратор має перевірити адресу прив'язки. Для локальної перевірки можна залишити 127.0.0.1. Якщо вузол має бути доступний у локальній мережі, потрібно вказати потрібну адресу або 0.0.0.0 для доступу зі всюди, але так не рекомендовано робити якщо мережа не захищена, або на сервері зберігаються важливі, або цінні файли.

Панель керування краще залишати локальною. У програмі є перевірка токена, але це не повна рольова система для відкритого інтернету. Тому адміністративний інтерфейс не потрібно відкривати назовні без додаткового захисту.

Пароль користувача для внутрішньої частини системи зберігається не у відкритому вигляді, а як хеш із сіллю. При цьому FTP-пароль у поточній реалізації зберігається локально окремо, бо DummyAuthorizer з pyftplib потребує звичайний пароль для підключення користувача. Через це файл бази даних треба зберігати в захищеному місці.

Для резервного копіювання можна використовувати експорт конфігурації у JSON. Окремо бажано зберігати і файл data.app.db, тому що в ньому знаходиться повний стан системи. При перенесенні на інший комп'ютер потрібно перевірити шляхи до корневих папок FTP-вузлів, бо вони можуть відрізнятись.

4.3 Оптимізація програмного забезпечення

Оптимізація виконувалася з урахуванням майбутнього вдосконалення та можливості зручного переписування коду застосунку. Код було розподілено між окремими файлами та модулями, щоб уникнути надмірного навантаження на один файл і зробити структуру застосунку більш зрозумілою та зручною для подальшої підтримки. Під час розробки основних модулів це потрібно враховувати, щоб не витрачати час на пошук необхідних функцій у різних файлах, які можуть не мати прямого зв'язку з відповідним функціоналом.

Для цієї програми важливо, щоб інтерфейс не зависав під час запуску вузлів, щоб дані зберігалися локально, а помилки було легко знайти без перегляду консолі або файлу з логами.

Серверна частина працює через бібліотеку `aiohhttp`. Інтерфейс надсилає запити до локального `Core API`, а `API` вже працює з базою даних, менеджером вузлів і журналом. Для `SQLite` використовується `aiosqlite`, тому запити до бази не повинні повністю блокувати роботу програми.

FTP-вузли запускаються окремо від інтерфейсу. Кожен вузол має власні параметри: назву, IP-адресу, порт, кореневу папку, обмеження підключень, тайм-аут та обмеження швидкості. Завдяки цьому можна створювати кілька різних вузлів і керувати ними з одного місця.

Щоб зменшити кількість помилок, перед збереженням вузла перевіряється порт. Якщо порт вже зайнятий, програма не повинна створювати вузол з такими параметрами. Також перевіряється шлях доступу, щоб користувач не міг вийти за межі встановленої, або кореневої папки FTP-вузла.

Для стабільності роботи FTP-вузлів, використовується `watchdog`. Він періодично перевіряє стан вузлів. Якщо вузол завершився з помилкою, система може спробувати перезапустити його. Кількість таких спроб

обмежена, щоб програма не повторювала перезапуск без кінця. Усі рішення для оптимізації можна побачити у таблиці 4.3.

Таблиця 4.3 – Основні рішення для оптимізації програми

Рішення	Як реалізовано	Що це дає
Локальний вебінтерфейс	Панель відкривається у вікні <code>ruwebview</code> .	Користувач працює з програмою без браузера і консолі.
Асинхронний API	Запити обробляються через <code>aioshttp</code> .	Команди інтерфейсу виконуються без зайвого блокування.
SQLite	Дані зберігаються у <code>data.app.db</code> .	Не потрібен окремий сервер баз даних.
Перевірка портів	Перед збереженням перевіряється <code>ip</code> і <code>port</code> .	Менше помилок під час запуску вузла.
Перевірка шляху	Використовується <code>normalize inside root()</code> .	Правило доступу не виходить за межі кореневої папки [21].
Обмеження сесій	Для вузла задаються <code>max_cons</code> і <code>max_cons_per_ip</code> .	Зменшується ризик перевантаження.
Журнал подій	Події записуються у буфер і таблицю <code>events</code> .	Помилки можна знайти у вікні програми.
Watchdog	<code>NodeManager</code> перевіряє стан вузлів.	Після аварійної зупинки можлива спроба відновлення [17].

Після реалізації основних функцій було проведено функціональне тестування. Перевірка виконувалась вручну через інтерфейс програми, тому що основні сценарії пов'язані саме з діями адміністратора у вікні застосунку. Для кожного сценарію порівнювався очікуваний результат і фактична поведінка програми. Подивитись на результати перевірки можна звернувшись до таблиці 4.4.

Таблиця 4.4 – Результати функціонального тестування

Сценарій перевірки	Очікуваний результат	Фактичний результат	Статус
Створення FTP-вузла	Після збереження вузол з'являється у списку.	Вузол створився і відобразився на сторінці FTP-вузлів.	Успішно
Перевірка зайнятого порту	Програма не повинна створити вузол на зайнятому порту.	При зайнятому порту з'являється помилка, вузол не створюється.	Успішно
Створення користувача	Новий користувач додається у таблицю користувачів.	Користувач створився, пароль збережено через хешування.	Успішно
Призначення доступу	Правило зберігається для обраного користувача і вузла.	У таблиці доступів з'явився запис з папкою і типом прав.	Успішно

Кінець таблиці 4.4

Сценарій перевірки	Очікуваний результат	Фактичний результат	Статус
Запуск FTP-вузла	Після запуску вузол переходить у робочий стан.	Вузол запустився, у журналі з'явився запис про запуск.	Успішно
Зупинка FTP-вузла	Після зупинки вузол не приймає підключення.	Вузол зупинився, стан оновився в інтерфейсі.	Успішно
Отримання журналу	На сторінці журналу мають бути останні події.	Журнал відкривається, записи можна шукати і фільтрувати.	Успішно
Експорт конфігурації	Програма формує JSON з налаштуваннями.	Конфігурація експортується з вузлами, користувачами та правилами.	Успішно
Імпорт конфігурації	Дані з JSON завантажуються назад у систему.	Після імпорту дані оновлюються, вузли перечитуються менеджером.	Успішно

Окремо було виділено, які підсистеми перевірялись найбільше. Це кількість ручних перевірок за частинами програми. Для цієї роботи такий варіант підходить, бо головна задача була перевірити реальні дії під час використання програми. Покриття тестування програми можна подивитись за таблицею 4.5.

Таблиця 4.5 – Покриття тестування за підсистемами

Підсистема	Кількість перевірок	Що перевірялось
Nodes	4	Створення, перевірка порту, запуск, зупинка або перезапуск вузла.
Users	3	Створення користувача, зміна даних, активність облікового запису.
Rules	5	Створення правила, тип доступу, шлях до папки, видалення правила, перевірка меж каталогу.
Logs	5	Запис подій, фільтр рівня, пошук, оновлення журналу, повідомлення про помилки.
Export	2	Формування JSON і перевірка наявності основних даних.
Import	2	Завантаження JSON і оновлення списку вузлів після імпорту.

За результатами перевірки основні сценарії виконались правильно. Програма створює вузли, не дозволяє використовувати зайнятий порт, додає користувачів, зберігає доступи, запускає FTP-вузол і записує події у журнал. Також було перевірено, що конфігурацію можна експортувати та знову імпортувати.

Під час тестування були помітні і обмеження поточної версії. Система не підтримує FTPS або SFTP, тому її краще використовувати у локальній мережі. Для більш серйозного використання потрібно допрацювати безпеку FTP-паролів і додати повнішу модель прав адміністратора.

4.4 Опис інтерфейсу програми

Програма має бічне меню, через яке користувач може перейти до огляду вузлів, списку FTP-вузлів, керування користувачами, доступами, журналу подій та інформації про програму. Така структура дозволяє швидко знаходити потрібні функції та керувати FTP-сервером без роботи з командним рядком.

На рисунку 4.1 показано сторінку огляду FTP-вузлів. Це головна сторінка при запуску програми. Ця сторінка використовується для швидкої перевірки стану FTP-вузлів. У блоці з вузлами відображаються загальні показники: кількість вузлів, кількість запущених вузлів, активні сесії та помилки за останні 24 години. Також для зручності розташовано пошук і фільтр, які допомагають швидко знайти потрібний вузол.

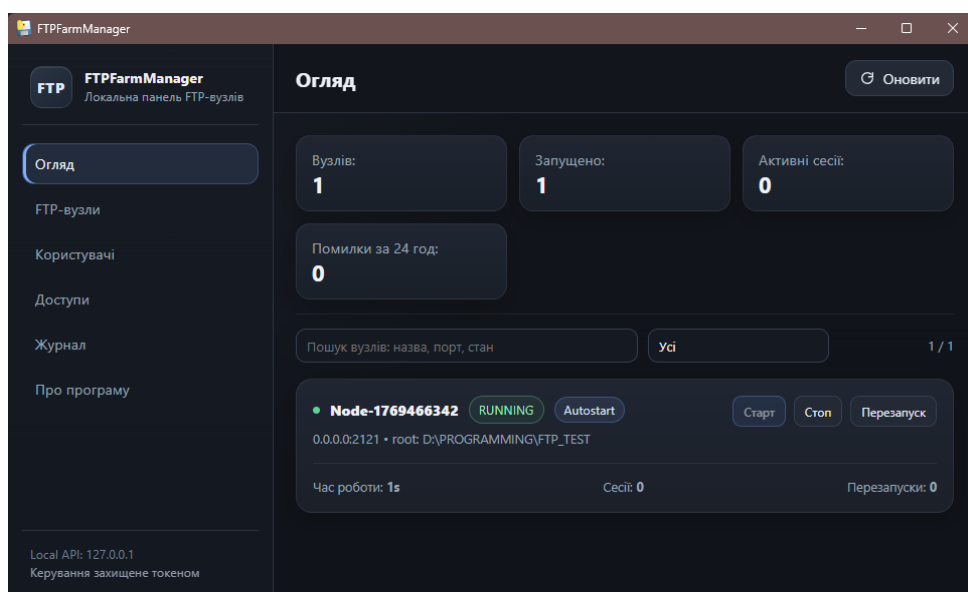


Рисунок 4.1 – Сторінка огляду FTP-вузлів

На рисунку 4.2 показано сторінку керування FTP-вузлами. На цій сторінці користувач може переглядати створені вузли, знаходити потрібний вузол через пошук, додавати нові вузли, а також змінювати або видаляти вже наявні записи.

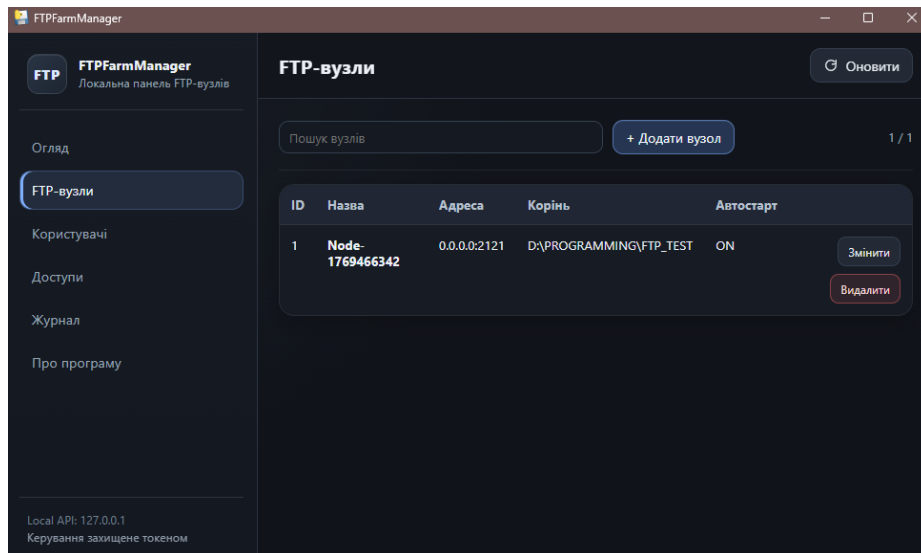


Рисунок 4.2 – Сторінка керування FTP-вузлами

На рисунку 4.3 форма додавання нового FTP-вузла. У ній користувач задає назву вузла, IP-адресу, порт, кореневу папку та інші параметри роботи сервера. Також доступні налаштування автозапуску, обмеження підключень і перевірка порту. Після заповнення полів новий вузол додається до списку FTP-вузлів.

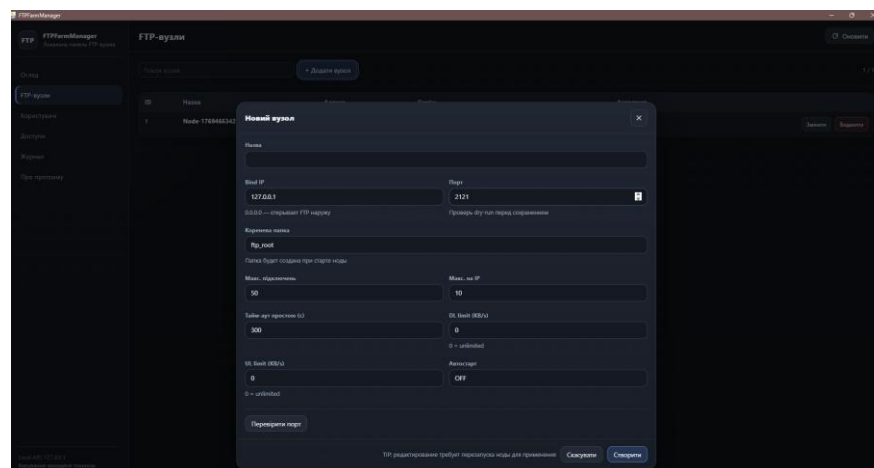


Рисунок 4.3 – Форма додавання нового FTP-вузла

На рисунку 4.4 показано сторінку користувачів, а саме тих людей хто буде отримувати доступи до вузла, тобто бачити файли на сервері. Сторінка призначена для перегляду та керування обліковими записами, які можуть використовувати FTP-доступ. На сторінці є поле пошуку, кнопка додавання нового користувача та таблиця зі створеними записами.

У таблиці відображається ID користувача, його логін і статус активності. Праворуч розміщені кнопки зміни та видалення, які дозволяють редагувати дані користувача або видалити його зі списку.

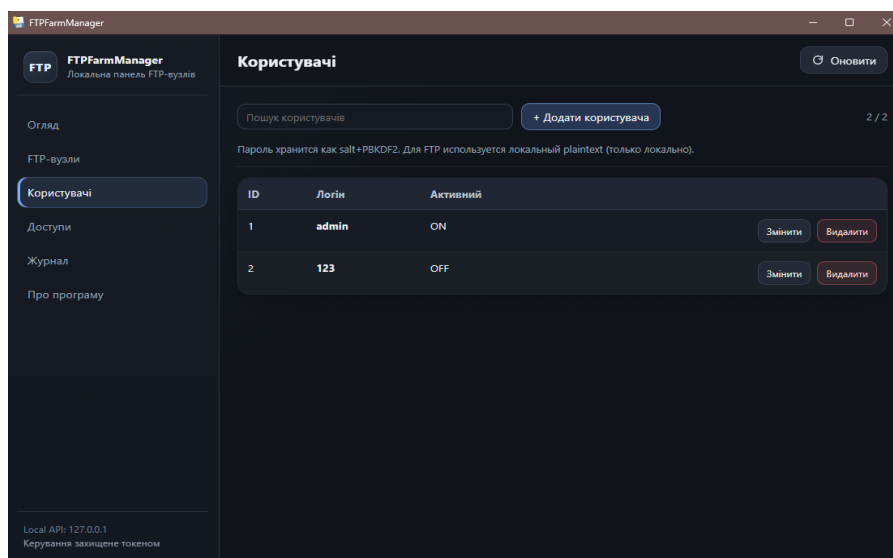


Рисунок 4.4 – Сторінка керування користувачами

На рисунку 4.5 показано сторінку доступу. Вона використовується для налаштування прав доступу користувачів до FTP-вузлів і окремих папок. У верхній частині сторінки користувач обирає обліковий запис, потрібний вузол, вказує відносний шлях до папки та задає тип прав доступу.

Нижче відображається таблиця вже створених правил доступу, де показано користувача, вузол, папку та права. За потреби правило можна видалити відповідною кнопкою.

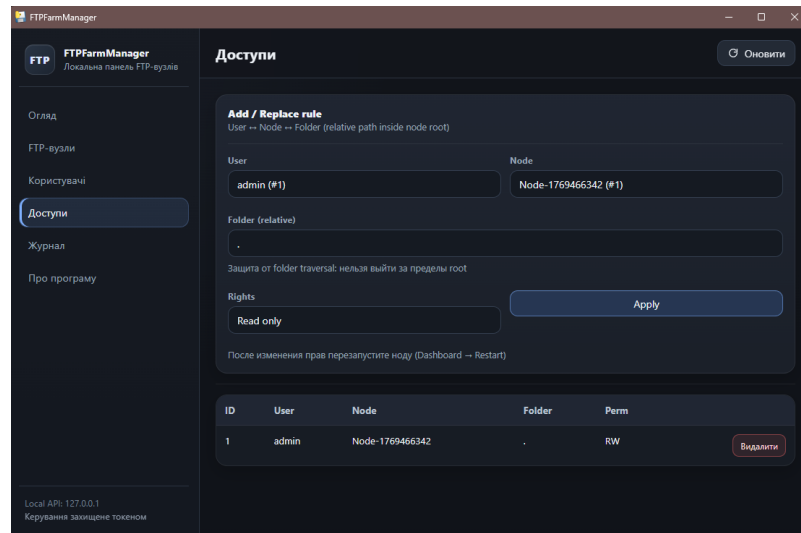


Рисунок 4.5 – Сторінка налаштування доступу користувачів

На рисунку 4.6 зображено сторінку журналу подій. Вона використовується для перегляду службових повідомлень програми. У журналі відображаються події запуску ядра, запуску FTP-вузлів та інша інформація про роботу системи.

У верхній частині сторінки розміщено поле пошуку, фільтр рівнів подій, кнопку оновлення та кнопку копіювання записів. Це дозволяє швидко знайти потрібну подію або зберегти інформацію з журналу для подальшої перевірки. Усього доступно три рівні подій, а саме інформативні, проблемні та попередження з приводу роботи вузлів.

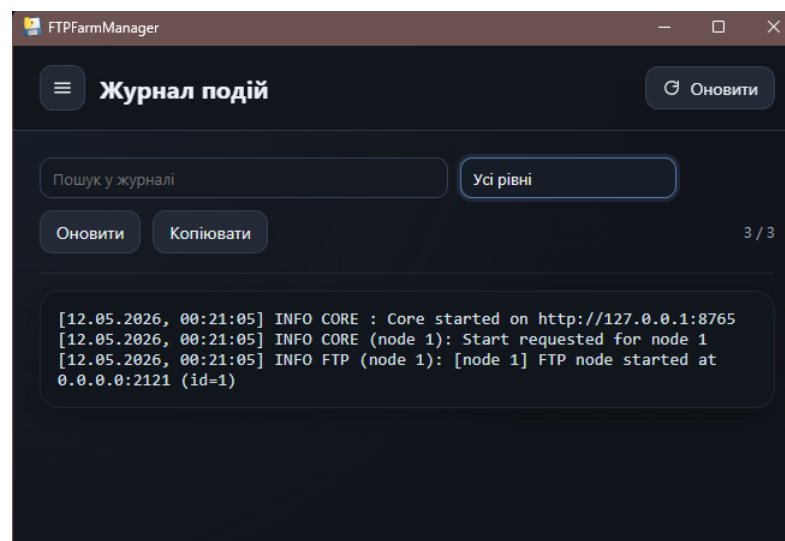


Рисунок 4.6 – Сторінка журналу подій

На рисунку 4.7 сторінка про програму. У цьому розділі відображається назва застосунку, його версія, коротка інформація про програму, а також блоки з контактами та майбутніми можливостями.

Також на сторінці є кнопки для перевірки оновлень, копіювання версії програми та експорту конфігурації у форматі JSON. Це зручно для збереження налаштувань і перевірки актуальності встановленої версії програми, але ця можливість працює тільки з актуальним сервером, тому це тільки демонстрація майбутнього можливого оновлення.

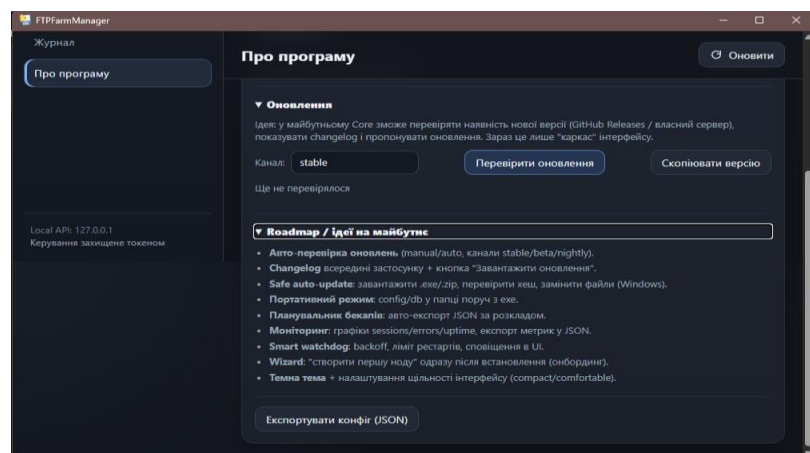


Рисунок 4.7 – Сторінка «Про програму»

4.5 Висновки за розділом 4

У четвертому розділі було описано порядок експлуатації програми FTPFarmManager. Було показано, як адміністратор запускає програму, створює FTP-вузол, додає користувачів, налаштовує доступи, запускає вузли та перевіряє журнал подій.

Також були описані дії адміністратора. До них належать початкове налаштування, перевірка портів, робота зі службовими файлами, резервне копіювання бази даних і контроль безпеки. Окремо зазначено, чому програму краще використовувати як локальний інструмент.

У підрозділі з оптимізації було розглянуто рішення, які роблять систему зручнішою у використанні: локальна база SQLite, асинхронний API, перевірка портів, перевірка шляхів доступу, обмеження сесій, журнал подій і watchdog для контролю вузлів.

Функціональне тестування показало, що основні можливості програми працюють. Успішно перевірено створення вузла, перевірку зайнятого порту, створення користувача, призначення правила доступу, запуск вузла, отримання журналу, експорт та імпорт конфігурації.

Надалі програму можна покращити за рахунок додавання FTPS або SFTP, детальнішої системи ролей, зручнішого перегляду аудиту змін і безпечнішого зберігання FTP-паролів. Але у межах поставленої задачі система виконує потрібні функції та підходить для керування кількома FTP-вузлами з одного локального інтерфейсу.

ВИСНОВКИ

У ході виконання роботи було розроблено програмне забезпечення для керування фермою FTP-вузлів. У застосунку можна створювати вузли, змінювати їх налаштування, запускати, зупиняти та видаляти вже створені записи.

Було проведено аналіз предметної області та розглянуто проблематика ручного адміністрування FTP серверів. Після цього було визначено, що зручніше керувати вузлами, користувачами, доступами та журналом подій з одного інтерфейсу, а не виконувати всі дії окремо використовуючи консоль, або занадто функціональний інструмент, з перенасиченим інтерфейсом.

Також було проведено пошук і аналіз аналогів. Завдяки цьому вдалося побачити сильні і слабкі сторони готових рішень. На основі цього було реалізовано більш вузьконаправлений застосунок, який не перевантажений зайвими функціями і підходить саме для керування FTP-вузлами.

У практичній частині було реалізовано сторінки для огляду вузлів, керування FTP-вузлами, додавання нового вузла, роботи з користувачами, налаштування доступів, перегляду журналу подій та інформації про програму. Це дозволяє користувачу швидко знаходити потрібні дані та виконувати основні дії через вебінтерфейс.

В результаті розробки програмного забезпечення було отримано робочий застосунок, який спрощує адміністрування FTP-вузлів. Користувачу не потрібно вручну працювати з різними налаштуваннями через консоль, бо основні можливості зібрані в одному місці.

У подальшому систему можна доповнити розширеними налаштуваннями доступу, детальнішим журналом дій користувача, додатковою перевіркою введених даних та можливістю безпечного віддаленого доступу до панелі керування.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Postel J., Reynolds J. File Transfer Protocol : RFC 959. RFC Editor, 1985. URL: <https://www.rfc-editor.org/rfc/rfc959.html> (дата звернення: 05.05.2026).
2. File Transfer Protocol. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/File_Transfer_Protocol (дата звернення: 05.05.2026).
3. Horowitz M., Lunt S. FTP Security Extensions : RFC 2228. RFC Editor, 1997. URL: <https://www.rfc-editor.org/rfc/rfc2228.html> (дата звернення: 05.05.2026).
4. Ford-Hutchinson P. Securing FTP with TLS : RFC 4217. RFC Editor, 2005. URL: <https://www.rfc-editor.org/rfc/rfc4217.html> (дата звернення: 05.05.2026).
5. FileZilla Server Features. FileZilla Pro. URL: <https://filezillaapro.com/docs/server/features/filezilla-server-features/> (дата звернення: 05.05.2026).
6. Wing FTP Server Help. Wing FTP Software. URL: <https://www.wftpserver.com/help/ftpserver/uoeoeia.htm> (дата звернення: 05.05.2026).
7. Web Administration. Cerberus FTP Server. URL: <https://www.cerberusftp.com/features/administration-tools/web-administration/> (дата звернення: 05.05.2026).
8. CrushFTP Features. CrushFTP. URL: <https://www.crushftp.com/features.html> (дата звернення: 05.05.2026).
9. vsftpd - Secure, fast FTP server for UNIX-like systems. vsftpd. URL: <https://security.appspot.com/vsftpd.html> (дата звернення: 05.05.2026).
10. Python 3 Documentation. Python Software Foundation. URL: <https://docs.python.org/> (дата звернення: 05.05.2026).
11. aiohttp Documentation. aiohttp. URL: <https://docs.aiohttp.org/> (дата

звернення: 05.05.2026).

12. SQLite Documentation. SQLite. URL: <https://sqlite.org/docs.html> (дата звернення: 05.05.2026).

13. aiosqlite Documentation. Omnilib. URL: <https://aiosqlite.omnilib.dev/> (дата звернення: 05.05.2026).

14. pyftplib Documentation. Read the Docs. URL: <https://pyftplib.readthedocs.io/> (дата звернення: 05.05.2026).

15. Tutorial. pyftplib Documentation. Read the Docs. URL: <https://pyftplib.readthedocs.io/en/latest/tutorial.html> (дата звернення: 16.05.2026).

16. pywebview Documentation. pywebview. URL: <https://pywebview.flowrl.com/> (дата звернення: 05.05.2026).

17. Coroutines and Tasks. Python 3 Documentation. Python Software Foundation. URL: <https://docs.python.org/3/library/asyncio-task.html> (дата звернення: 13.05.2026).

18. Threading Thread-based parallelism. Python 3 Documentation. Python Software Foundation. URL: <https://docs.python.org/3/library/threading.html> (дата звернення: 13.05.2026).

19. Write-Ahead Logging. SQLite. URL: <https://sqlite.org/wal.html> (дата звернення: 13.05.2026).

20. API reference. pyftplib Documentation. Read the Docs. URL: <https://pyftplib.readthedocs.io/en/latest/api.html> (дата звернення: 13.05.2026).

21. CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'). MITRE. URL: <https://cwe.mitre.org/data/definitions/22.html> (дата звернення: 13.05.2026).

22. secrets Generate secure random numbers for managing secrets. Python 3 Documentation. Python Software Foundation. URL: <https://docs.python.org/3/library/secrets.html> (дата звернення: 13.05.2026).

23. hmac Keyed-Hashing for Message Authentication. Python 3 Documentation. Python Software Foundation. URL:

<https://docs.python.org/3/library/hmac.html> (дата звернення: 13.05.2026).

24. hashlib Secure hashes and message digests. Python 3 Documentation. Python Software Foundation. URL: <https://docs.python.org/3/library/hashlib.html> (дата звернення: 13.05.2026).

25. Password Storage Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (дата звернення: 13.05.2026).

ДОДАТОК А
Текст програми

A.1 Текст файлу адмін панелі app.py

```

from __future__ import annotations
import asyncio
import json
import threading
import os
import webview
from src.core.config import (
    load_settings,
    save_settings,
    AppConfig,
    DEFAULT_PORT_CANDIDATES,
    load_or_create_token,
)
from src.core.utils import is_port_free, resource_path
from src.core.app import start_core, stop_core
class PortWizardAPI:
    def __init__(self) -> None:
        self.candidates = list(DEFAULT_PORT_CANDIDATES)
    def get_candidates(self):
        return self.candidates or list(DEFAULT_PORT_CANDIDATES)
    def check_port(self, port) -> bool:
        # JS can send NaN -> null -> None или str, handle all cases
        try:
            p = int(port)
        except (TypeError, ValueError):
            return False
        return is_port_free("127.0.0.1", p)
    def save_port(self, port) -> bool:
        try:
            p = int(port)
        except (TypeError, ValueError):
            return False
        cfg = AppConfig(ui_port=p)
        save_settings(cfg)
        load_or_create_token()
        return True
    def _run_async(loop: asyncio.AbstractEventLoop) -> None:
        asyncio.set_event_loop(loop)
        loop.run_forever()
    def main() -> None:
        cfg = load_settings()
        # --- First run wizard (choose port) ---

```

```

if cfg is None:
    api = PortWizardAPI()
    wizard_html = resource_path(os.path.join("webui", "wizard", "index.html"))
    win = webview.create_window(
        "FTPFarmManager - First Run",
        wizard_html,
        width=520,
        height=380,
        resizable=False,
        js_api=api,
    )
    def wizard_post_init():
        # Refill port candidates after window is ready (fix race with pywebview
api)
        candidates = api.get_candidates()
        candidates_json = json.dumps(candidates)
        js = f"""
(function(){{
    const sel = document.getElementById("port");
    if(!sel) return;
    sel.innerHTML = "";
    const arr = {candidates_json};
    for(const p of arr){{
        const o = document.createElement("option");
        o.value = p;
        o.textContent = p;
        sel.appendChild(o);
    }}
    if(arr.length) sel.value = String(arr[0]);
    try {{
        if (typeof init === "function") {{
            sel.innerHTML = "";
            init();
        }}
    }} catch(e) {{}}
}}})();
"""
        try:
            win.evaluate_js(js)
        except Exception:
            pass
    webview.start(wizard_post_init, gui="edgechromium")
    cfg = load_settings()
    if cfg is None:
        return

```

```

# --- Ensure chosen port is free (fallback to first free candidate) ---
if not is_port_free(cfg.api_host, cfg.ui_port):
    for p in DEFAULT_PORT_CANDIDATES:
        if is_port_free(cfg.api_host, p):
            cfg.ui_port = p
            save_settings(cfg)
            break
# --- Start Core in background asyncio loop ---
loop = asyncio.new_event_loop()
t = threading.Thread(target=_run_async, args=(loop,), daemon=True)
t.start()
core_rt_holder = {"rt": None, "err": None}
start_evt = threading.Event()
async def boot():
    try:
        core_rt_holder["rt"] = await start_core(cfg)
    except Exception as e:
        core_rt_holder["err"] = e
    finally:
        start_evt.set()
asyncio.run_coroutine_threadsafe(boot(), loop)
if not start_evt.wait(timeout=15):
    # Core didn't start, stop loop
    loop.call_soon_threadsafe(loop.stop)
    raise RuntimeError("Core did not start in time")
if core_rt_holder["err"] is not None:
    loop.call_soon_threadsafe(loop.stop)
    raise core_rt_holder["err"]
# --- Open Admin UI ---
url = f"http://{cfg.api_host}:{cfg.ui_port}/"
webview.create_window("FTPFarmManager", url, width=1200, height=750)
# IMPORTANT: old pywebview versions don't support on_closed= in
webview.start().
# We do graceful shutdown after webview.start() returns.
try:
    webview.start(gui="edgechromium")
finally:
    try:
        rt = core_rt_holder.get("rt")
        if rt is not None:
            fut = asyncio.run_coroutine_threadsafe(stop_core(rt), loop)
            try:
                fut.result(timeout=8)
            except Exception:
                pass

```

```

finally:
    try:
        loop.call_soon_threadsafe(loop.stop)
    except Exception:
        pass
    try:
        t.join(timeout=2)
    except Exception:
        pass

```

A.1.1 Текст файла ядра app.py

```

from __future__ import annotations
import asyncio
from dataclasses import dataclass
from pathlib import Path
from aiohttp import web
from .config import AppConfig, read_startup_ids, write_startup_ids
from .db import Database
from .logging import FarmLogger
from .node_manager import NodeManager
from .api import create_app
from .utils import ensure_dir, resource_path
@dataclass
class CoreRuntime:
    cfg: AppConfig
    db: Database
    logger: FarmLogger
    mgr: NodeManager
    web_app: web.Application
    runner: web.AppRunner | None = None
    site: web.TCPSite | None = None
    async def start_core(cfg: AppConfig) -> CoreRuntime:
        ensure_dir(cfg.data_dir)
        db = Database(cfg.db_path)
        await db.open()
        logger = FarmLogger(db)
        mgr = NodeManager(db, logger)
        await mgr.load_nodes()
        await mgr.start_watchdog()
        ui_root = resource_path("webui")
        web_app = await create_app(db, mgr, logger, ui_root=ui_root)
        runner = web.AppRunner(web_app)
        await runner.setup()
        site = web.TCPSite(runner, cfg.api_host, cfg.ui_port)

```

```

    await site.start()
    await logger.log("INFO", "CORE", f'Core started on
http://{cfg.api_host}:{cfg.ui_port}')
    # Autostart
    startup_ids = set(read_startup_ids())
    rows = await db.fetchall("SELECT id, autostart FROM nodes")
    for r in rows:
        if int(r["autostart"]) == 1:
            startup_ids.add(int(r["id"]))
    write_startup_ids(sorted(startup_ids))
    for node_id in sorted(startup_ids):
        try:
            await mgr.start_node(node_id)
        except Exception as e:
            await logger.log("ERROR", "CORE", f'Autostart failed for node
{node_id}: {e!r}', node_id=node_id)
    return CoreRuntime(cfg=cfg, db=db, logger=logger, mgr=mgr,
web_app=web_app, runner=runner, site=site)
async def stop_core(rt: CoreRuntime) -> None:
    try:
        await rt.logger.log("INFO", "CORE", "Stopping Core...")
    except Exception:
        pass
    try:
        await rt.mgr.stop_watchdog()
    except Exception:
        pass
    # stop all nodes
    for node_id in list(rt.mgr.nodes.keys()):
        try:
            await rt.mgr.stop_node(node_id)
        except Exception:
            pass
    if rt.runner:
        try:
            await rt.runner.cleanup()
        except Exception:
            pass
    try:
        await rt.db.close()
    except Exception:
        pass

```

A.1.2 Текст файлу адмін панелі `арі.py`

```

from __future__ import annotations
import json
import time
from pathlib import Path
from typing import Any
from aiohttp import web
from .config import load_or_create_token, write_startup_ids, read_startup_ids
from .db import Database
from .node_manager import NodeManager
from .logging import FarmLogger
from .security import safe_token_eq, new_salt, hash_password
from .utils import is_port_free, normalize_inside_root
def _require_token(request: web.Request, token: str) -> None:
    hdr = request.headers.get("X-Auth-Token", "")
    if not hdr or not safe_token_eq(hdr, token):
        raise web.HTTPUnauthorized(text="Missing or invalid token")
def json_response(data: Any, status: int = 200) -> web.Response:
    return web.json_response(data, status=status, dumps=lambda x: json.dumps(x,
ensure_ascii=False))
def clamp_int(v: Any, lo: int, hi: int) -> int:
    try:
        n = int(v)
    except Exception:
        return lo
    return max(lo, min(hi, n))
async def create_app(db: Database, mgr: NodeManager, logger: FarmLogger,
ui_root: str) -> web.Application:
    token = load_or_create_token()
    app = web.Application()
    # Serve UI (inject token to window.__API_TOKEN__)
    async def index(request: web.Request) -> web.Response:
        html_path = Path(ui_root) / "app" / "index.html"
        html = html_path.read_text(encoding="utf-8")
        injected = html.replace("/* __TOKEN__ */", f'window.__API_TOKEN__ =
"{token}";')
        return web.Response(text=injected, content_type="text/html", charset="utf-
8")
    async def static_handler(request: web.Request) -> web.Response:
        rel = request.match_info["path"]
        # prevent traversal
        if ".." in rel or rel.startswith("/"):
            raise web.HTTPNotFound()
        file_path = Path(ui_root) / "app" / rel

```

```

if not file_path.exists() or not file_path.is_file():
    raise web.HTTPNotFound()
ctype = "text/plain"
if rel.endswith(".js"):
    ctype = "application/javascript"
elif rel.endswith(".css"):
    ctype = "text/css"
elif rel.endswith(".html"):
    ctype = "text/html"
return web.Response(text=file_path.read_text(encoding="utf-8"),
content_type=ctype, charset="utf-8")
# Wizard static
async def wizard_index(request: web.Request) -> web.Response:
    html_path = Path(ui_root) / "wizard" / "index.html"
    html = html_path.read_text(encoding="utf-8")
    return web.Response(text=html, content_type="text/html", charset="utf-8")
async def wizard_static(request: web.Request) -> web.Response:
    rel = request.match_info["path"]
    if ".." in rel or rel.startswith("/"):
        raise web.HTTPNotFound()
    file_path = Path(ui_root) / "wizard" / rel
    if not file_path.exists() or not file_path.is_file():
        raise web.HTTPNotFound()
    ctype = "text/plain"
    if rel.endswith(".js"):
        ctype = "application/javascript"
    elif rel.endswith(".css"):
        ctype = "text/css"
    elif rel.endswith(".html"):
        ctype = "text/html"
    return web.Response(text=file_path.read_text(encoding="utf-8"),
content_type=ctype, charset="utf-8")
# API routes
routes = web.RouteTableDef()
@routes.get("/api/v1/status")
async def status(request: web.Request) -> web.Response:
    _require_token(request, token)
    st = await mgr.get_status()
    return json_response([s.__dict__ for s in st])
@routes.get("/api/v1/nodes")
async def list_nodes(request: web.Request) -> web.Response:
    _require_token(request, token)
    rows = await db.fetchall("SELECT * FROM nodes ORDER BY id")
    return json_response([dict(r) for r in rows])
@routes.post("/api/v1/nodes")

```

```

async def create_node(request: web.Request) -> web.Response:
    _require_token(request, token)
    body = await request.json()
    now = int(time.time())
    name = str(body.get("name", "")).strip() or f"Node-{now}"
    bind_ip = str(body.get("bind_ip", "127.0.0.1")).strip()
    port = int(body.get("port", 2121))
    root_path = str(body.get("root_path", "")).strip() or "ftp_root"
    max_cons = clamp_int(body.get("max_cons", 50), 1, 5000)
    max_cons_per_ip = clamp_int(body.get("max_cons_per_ip", 10), 1, 1000)
    idle_timeout = clamp_int(body.get("idle_timeout", 300), 30, 86400)
    dl_limit_kbps = clamp_int(body.get("dl_limit_kbps", 0), 0, 10_000_000)
    ul_limit_kbps = clamp_int(body.get("ul_limit_kbps", 0), 0, 10_000_000)
    autostart = 1 if bool(body.get("autostart", False)) else 0
    # Validate port not occupied (best-effort)
    if not is_port_free(bind_ip, port):
        raise web.HTTPBadRequest(text="Port is busy on this bind IP")
    cur = await db.execute(
        """INSERT INTO nodes(name, bind_ip, port, root_path, autostart,
max_cons, max_cons_per_ip, idle_timeout, dl_limit_kbps, ul_limit_kbps,
created_at, updated_at)
VALUES(?,?,?,?,?,?,?,?,?,?,?,?)""",
        (name, bind_ip, port, root_path, autostart, max_cons, max_cons_per_ip,
idle_timeout, dl_limit_kbps, ul_limit_kbps, now, now),
    )
    node_id = cur.lastrowid
    await db.add_revision("node", int(node_id), "create", {"node": dict(body)})
    # load into manager
    await mgr.load_nodes()
    return json_response({"id": node_id}, status=201)
@routes.put("/api/v1/nodes/{node_id}")
async def update_node(request: web.Request) -> web.Response:
    _require_token(request, token)
    node_id = int(request.match_info["node_id"])
    body = await request.json()
    row = await db.fetchone("SELECT * FROM nodes WHERE id=?",
(node_id,))
    if not row:
        raise web.HTTPNotFound(text="Node not found")
    now = int(time.time())
    # if changing bind/port, check availability (unless node currently running)
    bind_ip = str(body.get("bind_ip", row["bind_ip"])).strip()
    port = int(body.get("port", row["port"]))
    if (bind_ip != row["bind_ip"] or port != row["port"]) and not
is_port_free(bind_ip, port):

```

```

        raise web.HTTPBadRequest(text="Port is busy on this bind IP")
    fields = {
        "name": str(body.get("name", row["name"])).strip() or row["name"],
        "bind_ip": bind_ip,
        "port": port,
        "root_path": str(body.get("root_path", row["root_path"])).strip() or
row["root_path"],
        "autostart": 1 if bool(body.get("autostart", row["autostart"])) else 0,
        "max_cons": clamp_int(body.get("max_cons", row["max_cons"]), 1,
5000),
        "max_cons_per_ip": clamp_int(body.get("max_cons_per_ip",
row["max_cons_per_ip"]), 1, 1000),
        "idle_timeout": clamp_int(body.get("idle_timeout", row["idle_timeout"]),
30, 86400),
        "dl_limit_kbps": clamp_int(body.get("dl_limit_kbps",
row["dl_limit_kbps"]), 0, 10_000_000),
        "ul_limit_kbps": clamp_int(body.get("ul_limit_kbps",
row["ul_limit_kbps"]), 0, 10_000_000),
        "updated_at": now,
    }
    await db.execute(
        """UPDATE nodes SET name=?, bind_ip=?, port=?, root_path=?,
autostart=?, max_cons=?, max_cons_per_ip=?, idle_timeout=?, dl_limit_kbps=?,
ul_limit_kbps=?, updated_at=?
        WHERE id=?""",
        (fields["name"], fields["bind_ip"], fields["port"], fields["root_path"],
fields["autostart"],
        fields["max_cons"], fields["max_cons_per_ip"], fields["idle_timeout"],
fields["dl_limit_kbps"], fields["ul_limit_kbps"], fields["updated_at"], node_id),
    )
    await db.add_revision("node", node_id, "update", {"before": dict(row),
"after": fields})
    await mgr.load_nodes()
    # update startup.ini
    startup_ids = set(read_startup_ids())
    if fields["autostart"]:
        startup_ids.add(node_id)
    else:
        startup_ids.discard(node_id)
    write_startup_ids(sorted(startup_ids))
    return json_response({"ok": True})
@routes.delete("/api/v1/nodes/{node_id}")
async def delete_node(request: web.Request) -> web.Response:
    _require_token(request, token)
    node_id = int(request.match_info["node_id"])

```

```

await db.execute("DELETE FROM nodes WHERE id=?", (node_id,))
await db.add_revision("node", node_id, "delete", {"id": node_id})
# also stop if running
if node_id in mgr.nodes:
    try:
        await mgr.stop_node(node_id)
    except Exception:
        pass
    mgr.nodes.pop(node_id, None)
# update startup.ini
startup_ids = [x for x in read_startup_ids() if x != node_id]
write_startup_ids(startup_ids)
return json_response({"ok": True})
@routes.post("/api/v1/nodes/{node_id}/start")
async def start_node(request: web.Request) -> web.Response:
    _require_token(request, token)
    node_id = int(request.match_info["node_id"])
    await mgr.start_node(node_id)
    return json_response({"ok": True})
@routes.post("/api/v1/nodes/{node_id}/stop")
async def stop_node(request: web.Request) -> web.Response:
    _require_token(request, token)
    node_id = int(request.match_info["node_id"])
    await mgr.stop_node(node_id)
    return json_response({"ok": True})
@routes.post("/api/v1/nodes/{node_id}/restart")
async def restart_node(request: web.Request) -> web.Response:
    _require_token(request, token)
    node_id = int(request.match_info["node_id"])
    await mgr.restart_node(node_id)
    return json_response({"ok": True})
@routes.post("/api/v1/check")
async def check_config(request: web.Request) -> web.Response:
    _require_token(request, token)
    body = await request.json()
    # dry-run validation
    bind_ip = str(body.get("bind_ip", "127.0.0.1")).strip()
    port = int(body.get("port", 2121))
    root = str(body.get("root_path", "ftp_root")).strip()
    ok_port = is_port_free(bind_ip, port)
    # path traversal checks are applied in access rules
    return json_response({"port_free": ok_port, "root_path": root})
# Users API
@routes.get("/api/v1/users")
async def list_users(request: web.Request) -> web.Response:

```

```

    _require_token(request, token)
    rows = await db.fetchall("SELECT id, username, is_active, created_at,
updated_at FROM users ORDER BY id")
    return json_response([dict(r) for r in rows])
@routes.post("/api/v1/users")
async def create_user(request: web.Request) -> web.Response:
    _require_token(request, token)
    body = await request.json()
    username = str(body.get("username", "")).strip()
    password = str(body.get("password", "")).strip()
    if not username or not password:
        raise web.HTTPBadRequest(text="username and password required")
    ftp_password = str(body.get("ftp_password", password)).strip()
    is_active = 1 if bool(body.get("is_active", True)) else 0
    salt = new_salt()
    pwh = hash_password(password, salt)
    now = int(time.time())
    # ensure ftp_password_plain column
    try:
        await db.execute("ALTER TABLE users ADD COLUMN
ftp_password_plain TEXT NOT NULL DEFAULT ''")
    except Exception:
        pass
    cur = await db.execute(
        "INSERT INTO users(username, password_hash, salt, ftp_password_plain,
is_active, created_at, updated_at) VALUES(?,?,?,?,?,?,?)",
        (username, pwh, salt, ftp_password, is_active, now, now),
    )
    user_id = int(cur.lastrowid)
    await db.add_revision("user", user_id, "create", {"username": username,
"is_active": bool(is_active)})
    return json_response({"id": user_id}, status=201)
@routes.put("/api/v1/users/{user_id}")
async def update_user(request: web.Request) -> web.Response:
    _require_token(request, token)
    user_id = int(request.match_info["user_id"])
    body = await request.json()
    row = await db.fetchone("SELECT * FROM users WHERE id=?", (user_id,))
    if not row:
        raise web.HTTPNotFound(text="User not found")
    username = str(body.get("username", row["username"])).strip() or
row["username"]
    is_active = 1 if bool(body.get("is_active", row["is_active"])) else 0
    password = body.get("password", None)
    ftp_password = body.get("ftp_password", None)

```

```

    salt = row["salt"]
    pwh = row["password_hash"]
    ftp_plain = row["ftp_password_plain"] if "ftp_password_plain" in row.keys()
else ""
    if password:
        salt = new_salt()
        pwh = hash_password(str(password), salt)
    if ftp_password is not None:
        ftp_plain = str(ftp_password)
    now = int(time.time())
    await db.execute(
        "UPDATE users SET username=?, password_hash=?, salt=?,
ftp_password_plain=?, is_active=?, updated_at=? WHERE id=?",
        (username, pwh, salt, ftp_plain, is_active, now, user_id),
    )
    await db.add_revision("user", user_id, "update", {"before": dict(row), "after":
{"username": username, "is_active": bool(is_active)}})
    return json_response({"ok": True})
@routes.delete("/api/v1/users/{user_id}")
async def delete_user(request: web.Request) -> web.Response:
    _require_token(request, token)
    user_id = int(request.match_info["user_id"])
    await db.execute("DELETE FROM users WHERE id=?", (user_id,))
    await db.add_revision("user", user_id, "delete", {"id": user_id})
    return json_response({"ok": True})
# Access rules
@routes.get("/api/v1/access_rules")
async def list_rules(request: web.Request) -> web.Response:
    _require_token(request, token)
    rows = await db.fetchall(
        """SELECT ar.id, ar.user_id, u.username, ar.node_id, n.name as
node_name, ar.rel_path, ar.can_write, ar.created_at
FROM access_rules ar
JOIN users u ON u.id=ar.user_id
JOIN nodes n ON n.id=ar.node_id
ORDER BY ar.id"""
    )
    return json_response([dict(r) for r in rows])
@routes.post("/api/v1/access_rules")
async def create_rule(request: web.Request) -> web.Response:
    _require_token(request, token)
    body = await request.json()
    user_id = int(body.get("user_id"))
    node_id = int(body.get("node_id"))
    rel_path = str(body.get("rel_path", "")).strip() or "."

```

```

    can_write = 1 if bool(body.get("can_write", False)) else 0
    # validate traversal
    node = await db.fetchone("SELECT root_path FROM nodes WHERE id=?",
(node_id,))
    if not node:
        raise web.HTTPBadRequest(text="Node not found")
    try:
        normalize_inside_root(node["root_path"], rel_path)
    except ValueError as e:
        raise web.HTTPBadRequest(text=str(e))
    now = int(time.time())
    await db.execute(
        "INSERT OR REPLACE INTO access_rules(user_id, node_id, rel_path,
can_write, created_at) VALUES(?,?,?,?,?)",
        (user_id, node_id, rel_path, can_write, now),
    )
    await db.add_revision("access_rule", user_id, "upsert", {"user_id": user_id,
"node_id": node_id, "rel_path": rel_path, "can_write": bool(can_write)})
    return json_response({"ok": True}, status=201)
@routes.delete("/api/v1/access_rules/{rule_id}")
async def delete_rule(request: web.Request) -> web.Response:
    _require_token(request, token)
    rule_id = int(request.match_info["rule_id"])
    await db.execute("DELETE FROM access_rules WHERE id=?", (rule_id,))
    await db.add_revision("access_rule", rule_id, "delete", {"id": rule_id})
    return json_response({"ok": True})
# Logs
@routes.get("/api/v1/logs")
async def logs(request: web.Request) -> web.Response:
    _require_token(request, token)
    limit = int(request.query.get("limit", "200"))
    level = request.query.get("level", None)
    node_id_q = request.query.get("node_id", None)
    node_id = int(node_id_q) if node_id_q is not None and node_id_q.isdigit()
else None
    out = await logger.tail(limit=limit, level=level, node_id=node_id)
    return json_response(out)
# Metrics
@routes.get("/api/v1/metrics")
async def metrics(request: web.Request) -> web.Response:
    _require_token(request, token)
    st = await mgr.get_status()
    total_sessions = sum(s.active_sessions for s in st)
    total_errors = sum(s.errors_24h for s in st)
    return json_response({

```

```

        "nodes_total": len(st),
        "nodes_running": sum(1 for s in st if s.running),
        "active_sessions": total_sessions,
        "errors_24h": total_errors,
        "ts": int(time.time()),
    })
# Export/Import
@routes.post("/api/v1/export_config")
async def export_config(request: web.Request) -> web.Response:
    _require_token(request, token)
    nodes = [dict(r) for r in await db.fetchall("SELECT * FROM nodes ORDER
BY id")]
    users = [dict(r) for r in await db.fetchall("SELECT id, username, is_active,
created_at, updated_at FROM users ORDER BY id")]
    rules = [dict(r) for r in await db.fetchall("SELECT * FROM access_rules
ORDER BY id")]
    return json_response({"nodes": nodes, "users": users, "access_rules": rules})
@routes.post("/api/v1/import_config")
async def import_config(request: web.Request) -> web.Response:
    _require_token(request, token)
    body = await request.json()
    nodes = body.get("nodes", [])
    users = body.get("users", [])
    rules = body.get("access_rules", [])
    # naive import: wipe and reinsert
    await db.execute("DELETE FROM access_rules")
    await db.execute("DELETE FROM nodes")
    await db.execute("DELETE FROM users")
    now = int(time.time())
    # ensure ftp_password_plain column
    try:
        await db.execute("ALTER TABLE users ADD COLUMN
ftp_password_plain TEXT NOT NULL DEFAULT ''")
    except Exception:
        pass
    # Users - imported users will be created inactive unless password fields
provided
    for u in users:
        username = str(u.get("username", "")).strip()
        if not username:
            continue
        # can't restore password safely without it -> create disabled
        salt = new_salt()
        pwh = hash_password("changeme", salt)
        is_active = int(u.get("is_active", 0))

```

```

    await db.execute(
        "INSERT INTO users(username, password_hash, salt,
ftp_password_plain, is_active, created_at, updated_at) VALUES(?,?,?,?,?,?,?)",
        (username, pwh, salt, "", is_active, now, now),
    )
    for n in nodes:
        await db.execute(
            """"INSERT INTO nodes(name, bind_ip, port, root_path, enabled,
autostart, max_cons, max_cons_per_ip, idle_timeout, dl_limit_kbps,
ul_limit_kbps, created_at, updated_at)
VALUES(?,?,?,?,?,?,?,?,?,?,?,?,?)""",
            (
                n.get("name", "node"),
                n.get("bind_ip", "127.0.0.1"),
                int(n.get("port", 2121)),
                n.get("root_path", "ftp_root"),
                int(n.get("enabled", 1)),
                int(n.get("autostart", 0)),
                int(n.get("max_cons", 50)),
                int(n.get("max_cons_per_ip", 10)),
                int(n.get("idle_timeout", 300)),
                int(n.get("dl_limit_kbps", 0)),
                int(n.get("ul_limit_kbps", 0)),
                now, now,
            ),
        )
    # map usernames to ids
    user_rows = await db.fetchall("SELECT id, username FROM users")
    user_map = {r["username"]: r["id"] for r in user_rows}
    node_rows = await db.fetchall("SELECT id, name FROM nodes")
    node_map = {r["name"]: r["id"] for r in node_rows}
    for r in rules:
        # allow rules to refer by user_id/node_id in dump; but they are new now
        user_id = r.get("user_id")
        node_id = r.get("node_id")
        # if dump includes username/node_name we can restore
        username = r.get("username")
        node_name = r.get("node_name") or r.get("name")
        if username in user_map:
            user_id = user_map[username]
        if node_name in node_map:
            node_id = node_map[node_name]
        if not user_id or not node_id:
            continue
        rel_path = r.get("rel_path", ".")

```

```

        can_write = int(r.get("can_write",0))
        await db.execute(
            "INSERT INTO access_rules(user_id, node_id, rel_path, can_write,
created_at) VALUES(?,?,?,?,?)",
            (int(user_id), int(node_id), str(rel_path), can_write, now),
        )
        await mgr.load_nodes()
        return json_response({"ok": True})
# Wizard API
@routes.get("/api/v1/wizard/port_free")
async def wizard_port_free(request: web.Request) -> web.Response:
    host = request.query.get("host", "127.0.0.1")
    port = int(request.query.get("port", "8765"))
    return json_response({"free": is_port_free(host, port)})
app.add_routes(routes)
app.router.add_get("/", index)
app.router.add_get("/wizard", wizard_index)
app.router.add_get("/app/{path:.*}", static_handler)
app.router.add_get("/wizard/{path:.*}", wizard_static)
return app

```

A.1.3 Текст файлу конфігурації app.py

```

from __future__ import annotations
import configparser
import secrets
from dataclasses import dataclass
from pathlib import Path
from .utils import ensure_dir
SETTINGS_FILE = Path("settings.ini")
TOKEN_FILE = Path("token.txt")
STARTUP_FILE = Path("startup.ini")
DEFAULT_PORT_CANDIDATES = [8765, 8766, 8767, 8768, 8769]
@dataclass
class AppConfig:
    ui_port: int
    api_host: str = "127.0.0.1"
    data_dir: str = "data"
    db_path: str = "data/app.db"
def load_or_create_token() -> str:
    if TOKEN_FILE.exists():
        return TOKEN_FILE.read_text(encoding="utf-8").strip()
    token = secrets.token_urlsafe(32)
    TOKEN_FILE.write_text(token, encoding="utf-8")

```

```

    return token
def load_settings() -> AppConfig | None:
    if not SETTINGS_FILE.exists():
        return None
    cp = configparser.ConfigParser()
    cp.read(SETTINGS_FILE, encoding="utf-8")
    ui_port = int(cp.get("app", "ui_port",
    fallback=str(DEFAULT_PORT_CANDIDATES[0])))
    data_dir = cp.get("app", "data_dir", fallback="data")
    ensure_dir(data_dir)
    db_path = cp.get("app", "db_path", fallback=f"{data_dir}/app.db")
    return AppConfig(ui_port=ui_port, data_dir=data_dir, db_path=db_path)
def save_settings(cfg: AppConfig) -> None:
    cp = configparser.ConfigParser()
    cp["app"] = {
        "ui_port": str(cfg.ui_port),
        "api_host": cfg.api_host,
        "data_dir": cfg.data_dir,
        "db_path": cfg.db_path,
    }
    SETTINGS_FILE.write_text("", encoding="utf-8")
    with SETTINGS_FILE.open("w", encoding="utf-8") as f:
        cp.write(f)
def read_startup_ids() -> list[int]:
    if not STARTUP_FILE.exists():
        return []
    ids: list[int] = []
    for line in STARTUP_FILE.read_text(encoding="utf-8").splitlines():
        line = line.strip()
        if not line:
            continue
        if line.isdigit():
            ids.append(int(line))
    return ids
def write_startup_ids(ids: list[int]) -> None:
    STARTUP_FILE.write_text("\n".join(str(x) for x in ids) + ("\n" if ids else ""),
    encoding="utf-8")

```

A.1.5 Текст файлу бази даних db.py

```

from __future__ import annotations
import json
import time
from dataclasses import dataclass

```

```

from typing import Any, Optional
import aiosqlite
SCHEMA = r"""
PRAGMA journal_mode=WAL;
CREATE TABLE IF NOT EXISTS nodes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    bind_ip TEXT NOT NULL,
    port INTEGER NOT NULL,
    root_path TEXT NOT NULL,
    enabled INTEGER NOT NULL DEFAULT 1,
    autostart INTEGER NOT NULL DEFAULT 0,
    max_cons INTEGER NOT NULL DEFAULT 50,
    max_cons_per_ip INTEGER NOT NULL DEFAULT 10,
    idle_timeout INTEGER NOT NULL DEFAULT 300,
    dl_limit_kbps INTEGER NOT NULL DEFAULT 0,
    ul_limit_kbps INTEGER NOT NULL DEFAULT 0,
    created_at INTEGER NOT NULL,
    updated_at INTEGER NOT NULL
);
CREATE UNIQUE INDEX IF NOT EXISTS idx_nodes_bind_port ON
nodes(bind_ip, port);
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    password_hash TEXT NOT NULL,
    salt TEXT NOT NULL,
    is_active INTEGER NOT NULL DEFAULT 1,
    created_at INTEGER NOT NULL,
    updated_at INTEGER NOT NULL
);
CREATE TABLE IF NOT EXISTS access_rules (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    node_id INTEGER NOT NULL,
    rel_path TEXT NOT NULL,
    can_write INTEGER NOT NULL DEFAULT 0,
    created_at INTEGER NOT NULL,
    UNIQUE(user_id, node_id),
    FOREIGN KEY(user_id) REFERENCES users(id) ON DELETE CASCADE,
    FOREIGN KEY(node_id) REFERENCES nodes(id) ON DELETE CASCADE
);
CREATE TABLE IF NOT EXISTS events (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    ts INTEGER NOT NULL,

```

```

level TEXT NOT NULL,
source TEXT NOT NULL,
message TEXT NOT NULL,
node_id INTEGER,
FOREIGN KEY(node_id) REFERENCES nodes(id) ON DELETE SET NULL
);
CREATE TABLE IF NOT EXISTS revisions (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    ts INTEGER NOT NULL,
    entity_type TEXT NOT NULL,
    entity_id INTEGER NOT NULL,
    action TEXT NOT NULL,
    payload_json TEXT NOT NULL
);
"""
@dataclass
class Database:
    path: str
    conn: aiosqlite.Connection | None = None
    async def open(self) -> None:
        self.conn = await aiosqlite.connect(self.path)
        await self.conn.executescript(SCHEMA)
        await self.conn.commit()
        # Migration: local-only plaintext FTP password (for pyftplib
        DummyAuthorizer)
        try:
            await self.conn.execute("ALTER TABLE users ADD COLUMN
ftp_password_plain TEXT NOT NULL DEFAULT ''")
        except Exception:
            pass
    async def close(self) -> None:
        if self.conn:
            await self.conn.close()
            self.conn = None
    async def execute(self, sql: str, params: tuple[Any, ...] = ()) -> aiosqlite.Cursor:
        assert self.conn
        cur = await self.conn.execute(sql, params)
        await self.conn.commit()
        return cur
    async def fetchall(self, sql: str, params: tuple[Any, ...] = ()) ->
list[aiosqlite.Row]:
        assert self.conn
        self.conn.row_factory = aiosqlite.Row
        cur = await self.conn.execute(sql, params)
        rows = await cur.fetchall()

```

```

    return rows
    async def fetchone(self, sql: str, params: tuple[Any, ...] = ()) ->
Optional[aiosqlite.Row]:
    assert self.conn
    self.conn.row_factory = aiosqlite.Row
    cur = await self.conn.execute(sql, params)
    row = await cur.fetchone()
    return row
    async def add_event(self, level: str, source: str, message: str, node_id: int | None
= None) -> None:
    ts = int(time.time())
    await self.execute(
        "INSERT INTO events(ts, level, source, message, node_id)
VALUES(?,?,?,?,"),
        (ts, level, source, message, node_id),
    )
    async def add_revision(self, entity_type: str, entity_id: int, action: str, payload:
dict[str, Any]) -> None:
    ts = int(time.time())
    await self.execute(
        "INSERT INTO revisions(ts, entity_type, entity_id, action, payload_json)
VALUES(?,?,?,?,"),
        (ts, entity_type, entity_id, action, json.dumps(payload,
ensure_ascii=False)),
    )

```

A.1.6 Текст файлу управління нодами ftp_node.py

```

from __future__ import annotations
import os
import threading
import time
from dataclasses import dataclass, field
from typing import Callable, Optional
from pyftplib.handlers import FTPHandler, ThrottledDTPHandler
from pyftplib.servers import FTPServer
from pyftplib.authorizers import DummyAuthorizer
@dataclass
class NodeRuntimeStats:
    started_at: int = 0
    last_error: str = ""
    restarts: int = 0
    active_sessions: int = 0
    errors_24h: int = 0 # simple counter; Core can reset periodically

```

```

class InstrumentedHandler(FTPHandler):
    # will be injected by factory
    on_session_open: Callable[[], None] | None = None
    on_session_close: Callable[[], None] | None = None
    on_error: Callable[[str], None] | None = None
    def on_connect(self):
        if self.on_session_open:
            self.on_session_open()
    def on_disconnect(self):
        if self.on_session_close:
            self.on_session_close()
    def on_login_failed(self, username, password):
        if self.on_error:
            self.on_error(f'Login failed for user '{username}''')
    def on_file_sent(self, file):
        pass
    def on_incomplete_file_sent(self, file):
        if self.on_error:
            self.on_error(f'Incomplete send: {file}')
    def on_incomplete_file_received(self, file):
        if self.on_error:
            self.on_error(f'Incomplete recv: {file}')
@dataclass
class FtpNode:
    node_id: int
    name: str
    bind_ip: str
    port: int
    root_path: str
    max_cons: int = 50
    max_cons_per_ip: int = 10
    idle_timeout: int = 300
    dl_limit_kbps: int = 0
    ul_limit_kbps: int = 0
    # internal
    _thread: threading.Thread | None = field(default=None, init=False)
    _server: FTPServer | None = field(default=None, init=False)
    _stop_flag: threading.Event = field(default_factory=threading.Event, init=False)
    stats: NodeRuntimeStats = field(default_factory=NodeRuntimeStats, init=False)
    def is_running(self) -> bool:
        return self._thread is not None and self._thread.is_alive()
    def _make_authorizer(self, users: list[dict]) -> DummyAuthorizer:
        auth = DummyAuthorizer()
        # users list items: {username, password_plain, home, perms}
        for u in users:

```

```

        auth.add_user(u["username"], u["password_plain"], u["home"],
perm=u["perms"])
    return auth
def start(self, users: list[dict], on_log: Callable[[str, str], None]) -> None:
    if self.is_running():
        return
    self._stop_flag.clear()
    self.stats.started_at = int(time.time())
    self.stats.last_error = ""
    # Ensure root exists
    os.makedirs(self.root_path, exist_ok=True)
    def run():
        try:
            authorizer = self._make_authorizer(users)
            class Handler(InstrumentedHandler):
                pass
            Handler.authorizer = authorizer
            Handler.timeout = self.idle_timeout
            # Throttling (0 = unlimited)
            if self.dl_limit_kbps > 0 or self.ul_limit_kbps > 0:
                Handler.dtp_handler = ThrottledDTPHandler
                # bytes/sec
                Handler.dtp_handler.read_limit = self.dl_limit_kbps * 1024 if
self.dl_limit_kbps > 0 else 0
                Handler.dtp_handler.write_limit = self.ul_limit_kbps * 1024 if
self.ul_limit_kbps > 0 else 0
            # hooks
            def sess_open():
                self.stats.active_sessions += 1
            def sess_close():
                self.stats.active_sessions = max(0, self.stats.active_sessions - 1)
            def err(msg: str):
                self.stats.errors_24h += 1
                on_log("WARN", msg)
            Handler.on_session_open = staticmethod(sess_open)
            Handler.on_session_close = staticmethod(sess_close)
            Handler.on_error = staticmethod(err)
            address = (self.bind_ip, self.port)
            self._server = FTPServer(address, Handler)
            self._server.max_cons = self.max_cons
            self._server.max_cons_per_ip = self.max_cons_per_ip
            on_log("INFO", f"FTP node started at {self.bind_ip}:{self.port}
(id={self.node_id})")
            self._server.serve_forever(timeout=0.5, blocking=True,
handle_exit=False)

```

```

except Exception as e:
    self.stats.last_error = str(e)
    on_log("ERROR", f"FTP node crashed: {e!r}")
finally:
    try:
        if self._server:
            self._server.close_all()
    except Exception:
        pass
    self._server = None
    self._thread = threading.Thread(target=run, name=f"ftp-node-{self.node_id}",
daemon=True)
    self._thread.start()
def stop(self, on_log: Callable[[str, str], None]) -> None:
    if not self._server:
        return
    try:
        on_log("INFO", f"Stopping FTP node id={self.node_id}...")
        self._server.close_all()
    except Exception as e:
        on_log("WARN", f"Stop error: {e!r}")
    finally:
        self._server = None
def restart(self, users: list[dict], on_log: Callable[[str, str], None]) -> None:
    self.stop(on_log)
    time.sleep(0.2)
    self.stats.restarts += 1
    self.start(users, on_log)

```

A.1.7 Текст файлу логування logging.py

```

from __future__ import annotations
import asyncio
import logging
import time
from dataclasses import dataclass
from typing import Deque, Optional
from collections import deque
from .db import Database
@dataclass
class InMemoryLog:
    ts: int
    level: str
    source: str
    message: str

```

```

    node_id: int | None = None
class FarmLogger:
    def __init__(self, db: Database, max_items: int = 2000) -> None:
        self.db = db
        self.buffer: Deque[InMemoryLog] = deque(maxlen=max_items)
        self._lock = asyncio.Lock()
        async def log(self, level: str, source: str, message: str, node_id: int | None =
None) -> None:
            item = InMemoryLog(int(time.time()), level, source, message, node_id)
            async with self._lock:
                self.buffer.append(item)
            # Persist best-effort
            try:
                await self.db.add_event(level, source, message, node_id=node_id)
            except Exception:
                # avoid recursive logging
                pass
        async def tail(self, limit: int = 200, level: str | None = None, node_id: int | None
= None) -> list[dict]:
            async with self._lock:
                items = list(self.buffer)
            if level:
                items = [x for x in items if x.level == level]
            if node_id is not None:
                items = [x for x in items if x.node_id == node_id]
            items = items[-limit:]
            return [
                {"ts": x.ts, "level": x.level, "source": x.source, "message": x.message,
"node_id": x.node_id}
                for x in items
            ]

```

A.1.8 Текст файлу тонкого керування вузлами node_manager.py

```

from __future__ import annotations
import asyncio
import time
from dataclasses import dataclass
from typing import Any
from .db import Database
from .ftp_node import FtpNode
from .utils import normalize_inside_root
from .logging import FarmLogger
@dataclass
class NodeStatus:

```

```

id: int
name: str
bind_ip: str
port: int
root_path: str
autostart: bool
running: bool
state: str # RUNNING/STOPPED/ERROR
uptime_s: int
active_sessions: int
errors_24h: int
restarts: int
last_error: str
class NodeManager:
    def __init__(self, db: Database, logger: FarmLogger) -> None:
        self.db = db
        self.logger = logger
        self.nodes: dict[int, FtpNode] = {}
        self._watchdog_task: asyncio.Task | None = None
        self._stop_event = asyncio.Event()
        self.max_restart_attempts = 3
        # node_id -> attempts in current crash loop
        self._restart_attempts: dict[int, int] = {}
    async def load_nodes(self) -> None:
        """Reload nodes from DB.

        Keeps existing runtime stats for already-known nodes, but updates
        configuration fields.
        """
        rows = await self.db.fetchall("SELECT * FROM nodes ORDER BY id")
        new_nodes: dict[int, FtpNode] = {}
        for r in rows:
            node_id = int(r["id"])
            prev = self.nodes.get(node_id)
            node = FtpNode(
                node_id=node_id,
                name=r["name"],
                bind_ip=r["bind_ip"],
                port=r["port"],
                root_path=r["root_path"],
                max_cons=r["max_cons"],
                max_cons_per_ip=r["max_cons_per_ip"],
                idle_timeout=r["idle_timeout"],
                dl_limit_kbps=r["dl_limit_kbps"],
                ul_limit_kbps=r["ul_limit_kbps"],
            )

```

```

# carry over stats & running server references if same id exists
if prev:
    node.stats = prev.stats
    node._thread = prev._thread
    node._server = prev._server
    new_nodes[node_id] = node
self.nodes = new_nodes
async def _build_node_users(self, node_id: int) -> list[dict[str, Any]]:
    """Build per-node auth list. One access rule per (user,node)."""
    rows = await self.db.fetchall(
        """
        SELECT u.username, ar.rel_path, ar.can_write
        FROM access_rules ar
        JOIN users u ON u.id = ar.user_id
        WHERE ar.node_id = ? AND u.is_active = 1
        ORDER BY u.username
        """,
        (node_id,),
    )
    # Practical compromise for MVP: store plaintext FTP password locally in DB
    # (separate from admin hash). If column doesn't exist yet, add it.
    try:
        await self.db.execute(
            "ALTER TABLE users ADD COLUMN ftp_password_plain TEXT
NOT NULL DEFAULT ''"
        )
    except Exception:
        pass
    users_rows = await self.db.fetchall(
        """
        SELECT username, ftp_password_plain
        FROM users
        WHERE is_active = 1
        """
    )
    user_plain = {ur["username"]: ur["ftp_password_plain"] for ur in users_rows}
    built: list[dict[str, Any]] = []
    for row in rows:
        username = row["username"]
        plain = user_plain.get(username, "")
        if not plain:
            # user can exist but FTP password not set
            continue
        rel_path = row["rel_path"]
        can_write = int(row["can_write"]) == 1

```

```

node = self.nodes[node_id]
home = normalize_inside_root(node.root_path, rel_path)
perms = "elr" if not can_write else "elradfmwMT"
built.append(
    {
        "username": username,
        "password_plain": plain,
        "home": home,
        "perms": perms,
    }
)
return built
async def start_node(self, node_id: int) -> None:
    node = self.nodes.get(node_id)
    if not node:
        raise KeyError("Node not found")
    users = await self._build_node_users(node_id)
    loop = asyncio.get_running_loop()
    def on_log(level: str, msg: str) -> None:
        asyncio.run_coroutine_threadsafe(
            self.logger.log(level, "FTP", f"[node {node_id}] {msg}"),
            loop,
        )
    node.start(users, on_log)
    await self.logger.log("INFO", "CORE", f"Start requested for node {node_id}", node_id=node_id)
    self._restart_attempts[node_id] = 0
    async def stop_node(self, node_id: int) -> None:
        node = self.nodes.get(node_id)
        if not node:
            raise KeyError("Node not found")
        loop = asyncio.get_running_loop()
        def on_log(level: str, msg: str) -> None:
            asyncio.run_coroutine_threadsafe(
                self.logger.log(level, "FTP", f"[node {node_id}] {msg}"),
                loop,
            )
        node.stop(on_log)
        await self.logger.log("INFO", "CORE", f"Stop requested for node {node_id}", node_id=node_id)
    async def restart_node(self, node_id: int) -> None:
        node = self.nodes.get(node_id)

```

```

if not node:
    raise KeyError("Node not found")
users = await self._build_node_users(node_id)
loop = asyncio.get_running_loop()
def on_log(level: str, msg: str) -> None:
    asyncio.run_coroutine_threadsafe(
        self.logger.log(level, "FTP", f"[node {node_id}] {msg}"),
node_id=node_id),
        loop,
    )
    node.restart(users, on_log)
    await self.logger.log("INFO", "CORE", f'Restart requested for node
{node_id}', node_id=node_id)
    self._restart_attempts[node_id] = 0
async def get_status(self) -> list[NodeStatus]:
    out: list[NodeStatus] = []
    now = int(time.time())
    rows = await self.db.fetchall("SELECT id, autostart FROM nodes")
    autostart = {int(r["id"]): int(r["autostart"]) == 1 for r in rows}
    for node_id, node in self.nodes.items():
        running = node.is_running()
        state = "RUNNING" if running else ("ERROR" if node.stats.last_error else
"STOPPED")
        uptime = now - node.stats.started_at if running and node.stats.started_at
else 0
    out.append(
        NodeStatus(
            id=node_id,
            name=node.name,
            bind_ip=node.bind_ip,
            port=node.port,
            root_path=node.root_path,
            autostart=autostart.get(node_id, False),
            running=running,
            state=state,
            uptime_s=uptime,
            active_sessions=node.stats.active_sessions,
            errors_24h=node.stats.errors_24h,
            restarts=node.stats.restarts,
            last_error=node.stats.last_error,
        )
    )
    return out
async def start_watchdog(self) -> None:
    if self._watchdog_task:

```

```

        return
    self._stop_event.clear()
    self._watchdog_task = asyncio.create_task(self._watchdog_loop())
    async def stop_watchdog(self) -> None:
        self._stop_event.set()
        if self._watchdog_task:
            await self._watchdog_task
        self._watchdog_task = None
    async def _watchdog_loop(self) -> None:
        # basic watchdog: if node thread died and last_error exists -> auto-restart with
        attempts
        while not self._stop_event.is_set():
            await asyncio.sleep(2.0)
            for node_id, node in list(self.nodes.items()):
                # if previously running but now dead with error
                if node.stats.started_at and not node.is_running() and
node.stats.last_error:
                    attempts = self._restart_attempts.get(node_id, 0)
                    if attempts >= self.max_restart_attempts:
                        continue
                    self._restart_attempts[node_id] = attempts + 1
                    await self.logger.log(
                        "WARN",
                        "WATCHDOG",
                        f'Node {node_id} crashed; restart attempt {attempts + 1}',
                        node_id=node_id,
                    )
                try:
                    await self.start_node(node_id)
                except Exception as e:
                    node.stats.last_error = str(e)
                    await self.logger.log(
                        "ERROR",
                        "WATCHDOG",
                        f'Restart failed: {e!r}',
                        node_id=node_id,
                    )

```

A.1.9 Текст файлу хешування паролів в базі даних security.py

```

from __future__ import annotations
import hmac
import hashlib
import secrets
from dataclasses import dataclass

```

```

PBKDF2_ITER = 200_000
def new_salt() -> str:
    return secrets.token_hex(16)
def hash_password(password: str, salt_hex: str) -> str:
    salt = bytes.fromhex(salt_hex)
    dk = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"), salt,
PBKDF2_ITER)
    return dk.hex()
def verify_password(password: str, salt_hex: str, expected_hash_hex: str) -> bool:
    calc = hash_password(password, salt_hex)
    return hmac.compare_digest(calc, expected_hash_hex)
def safe_token_eq(a: str, b: str) -> bool:
    return hmac.compare_digest(a.encode("utf-8"), b.encode("utf-8"))

```

A.1.10 Текст файлу системних утиліт `utils.py`

```

from __future__ import annotations
import os
import sys
import socket
from pathlib import Path
def resource_path(rel_path: str) -> str:
    """Return absolute path to resources, compatible with PyInstaller."""
    base_path = getattr(sys, "_MEIPASS", None)
    if base_path:
        return os.path.join(base_path, rel_path)
    return os.path.join(os.path.abspath(os.path.dirname(__file__)), "..", "..",
rel_path)
def ensure_dir(path: str) -> None:
    Path(path).mkdir(parents=True, exist_ok=True)
def is_port_free(host: str, port: int) -> bool:
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            s.bind((host, port))
        return True
    except OSError:
        return False
def normalize_inside_root(root: str, user_rel: str) -> str:
    """Return absolute path for user_rel inside root; raise ValueError if traversal
escapes root."""
    root_p = Path(root).resolve()
    candidate = (root_p / user_rel).resolve()
    if root_p == candidate or root_p in candidate.parents:
        return str(candidate)

```

```
raise ValueError("Path is outside node root (folder traversal detected)")
```

A.1.11 Текст головного файла для запуску програми run.py

```
from src.adminpanel.app import main  
if __name__ == "__main__":  
    main()
```

ДОДАТОК Б
Слайди презентації



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
"ЗАПОРІЗЬКА ПОЛІТЕХНІКА"
Кафедра ПРОГРАМНИХ ЗАСОБІВ



**Розроблення програмного забезпечення вебсистеми керування фермою
FTP-вузлів з оркестратором процесів та адмініструванням користувачів**

Виконав:
Студент гр. КНТ-113сп
Андрій ДАНЧУК

Керівник: к.т.н., доцент
Олександр СТЕПАНЕНКО

1

Рисунок Б 1 – Слайд 1

Об'єкт, предмет, мета та завдання роботи



Об'єкт дослідження
процес програмування функцій адміністрування та розгортання FTP-сервісів.

Предмет дослідження
методи і засоби побудови локальної вебсистеми керування вузлами, користувачами, правами доступу та журналом подій

Мета роботи
розробити програмне забезпечення для централізованого керування кількома FTP-вузлами з одного інтерфейсу

Основні завдання
аналіз предметної області і аналогів
вибір засобів розробки
проектування архітектури, БД та API
реалізація модулів і тестування



2

Рисунок Б 2 – Слайд 2

Постановка задачі	
Проблема предметної області	Необхідне рішення в системі
Ручне керування кількома FTP-серверами	Єдина локальна панель адміністрування
Розрізнені конфігурації портів і каталогів	Централізоване збереження параметрів у SQLite
Складність роботи з користувачами і правами	Модель User → Node → Folder з правилами доступу
Немає наочного контролю стану	Dashboard, метрики, журнал подій
Ризик помилок у конфігурації	Dry-run перевірка та валідація параметрів
Складний запуск для користувача	ruwebview + майстер першого запуску

Рисунок Б 3 – Слайд 3

Порівняння з аналогами				
Критерій	FileZilla Server	Wing FTP	Cerberus FTP	FTPFarmManager
Керування кількома вузлами	частково	так	так	так
Простий локальний запуск	так	частково	частково	так
Гнучкі правила User-Node-Folder	обмежено	так	так	так
Локальна БД без окремого сервера	ні	ні	ні	так
Простота адаптації під задачу підприємства	низька	середня	середня	висока
Надлишковість функціоналу для невеликої системи	низька	висока	висока	низька

Рисунок Б 4 – Слайд 4

Обґрунтування архітектурного підходу			
Підхід	Переваги	Недоліки	Висновок
Окремий FTP-сервер без панелі	простий базовий запуск	ручне адміністрування, слабка масштабованість	не обрано
Віддалена вебсистема	гнучкість, централізація	потрібен окремий сервер і складніше розгортання	не обрано
Локальний desktop-застосунок з Core API	просте розгортання, контроль, локальна робота, зручний інтерфейс	обмежена віддаленість	обрано

Рисунок Б 5 – Слайд 5

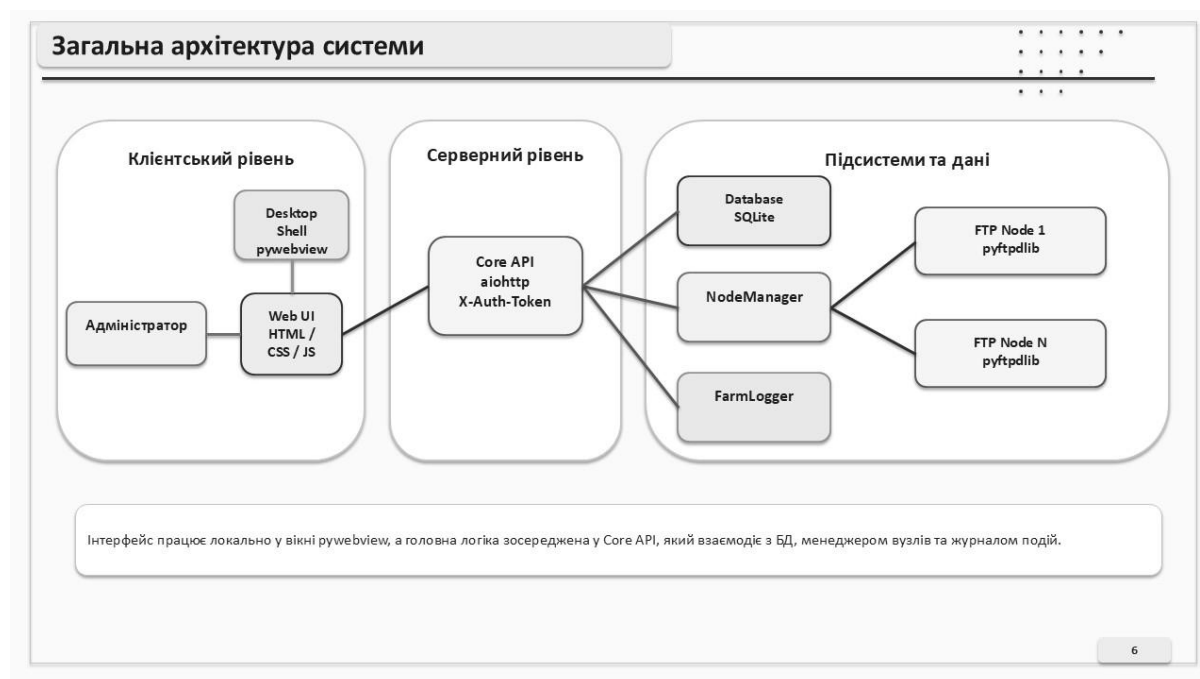


Рисунок Б 6 – Слайд 6

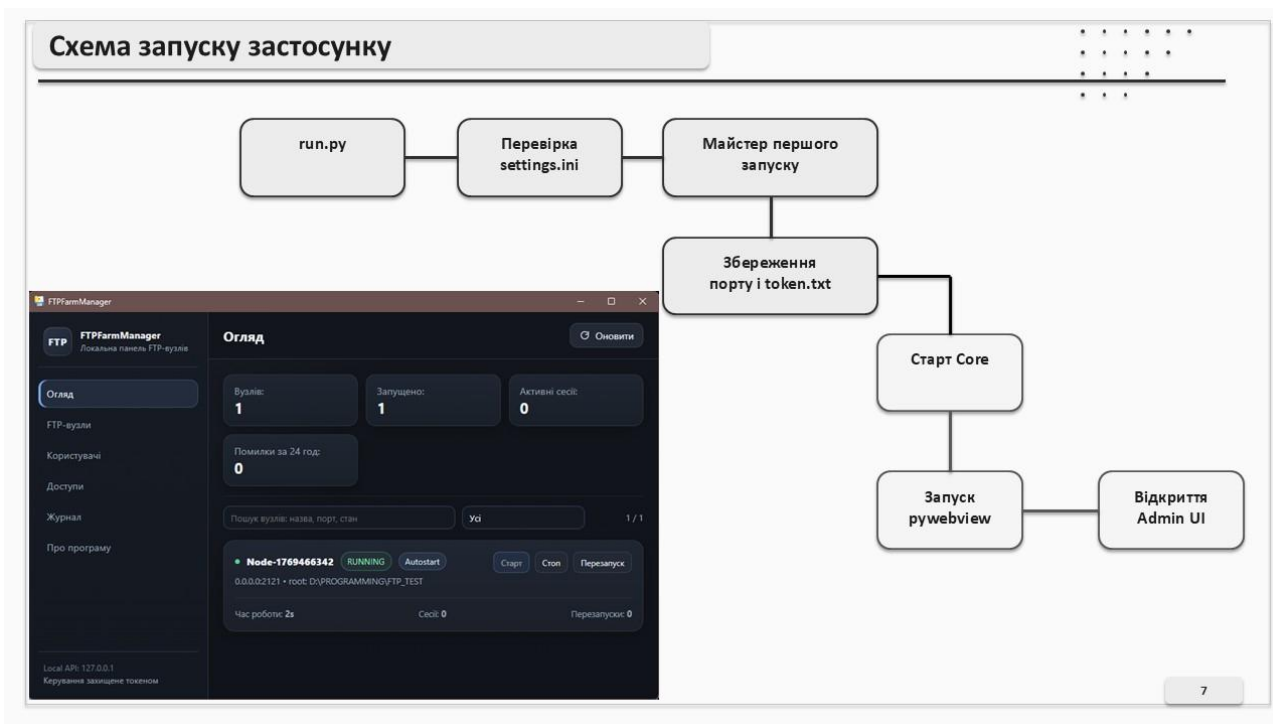


Рисунок Б 7 – Слайд 7



Рисунок Б 8 – Слайд 8

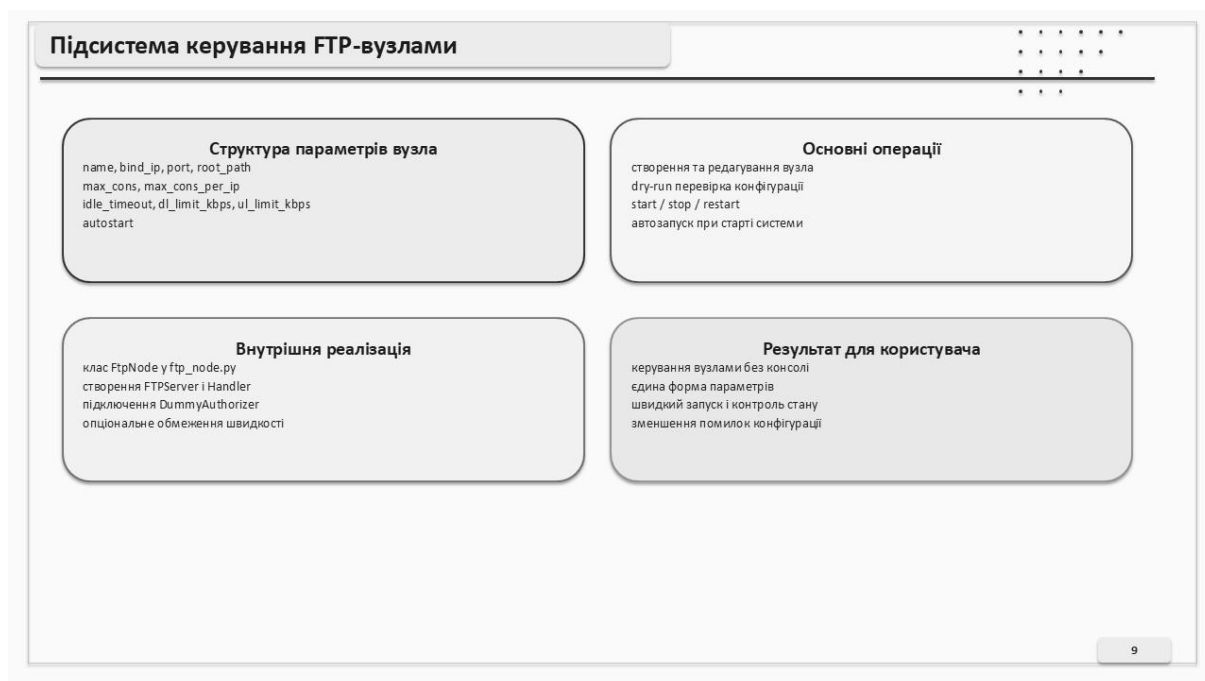


Рисунок Б 9 – Слайд 9

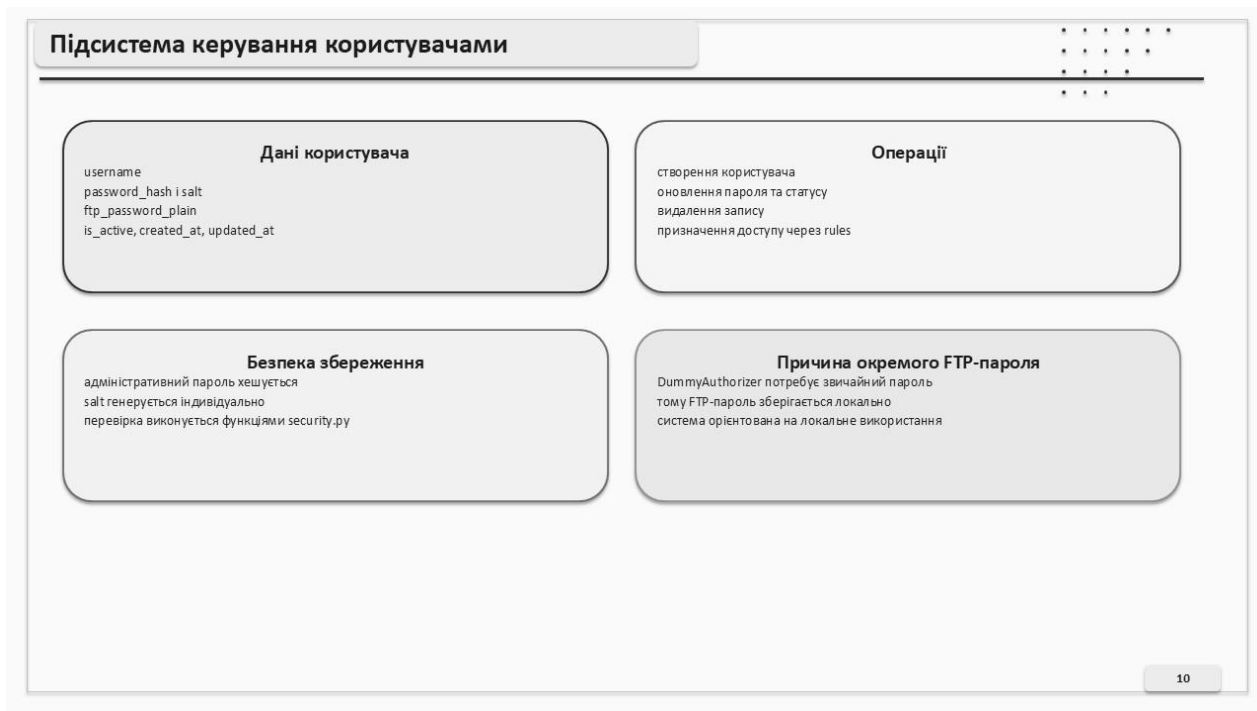


Рисунок Б 10 – Слайд 10

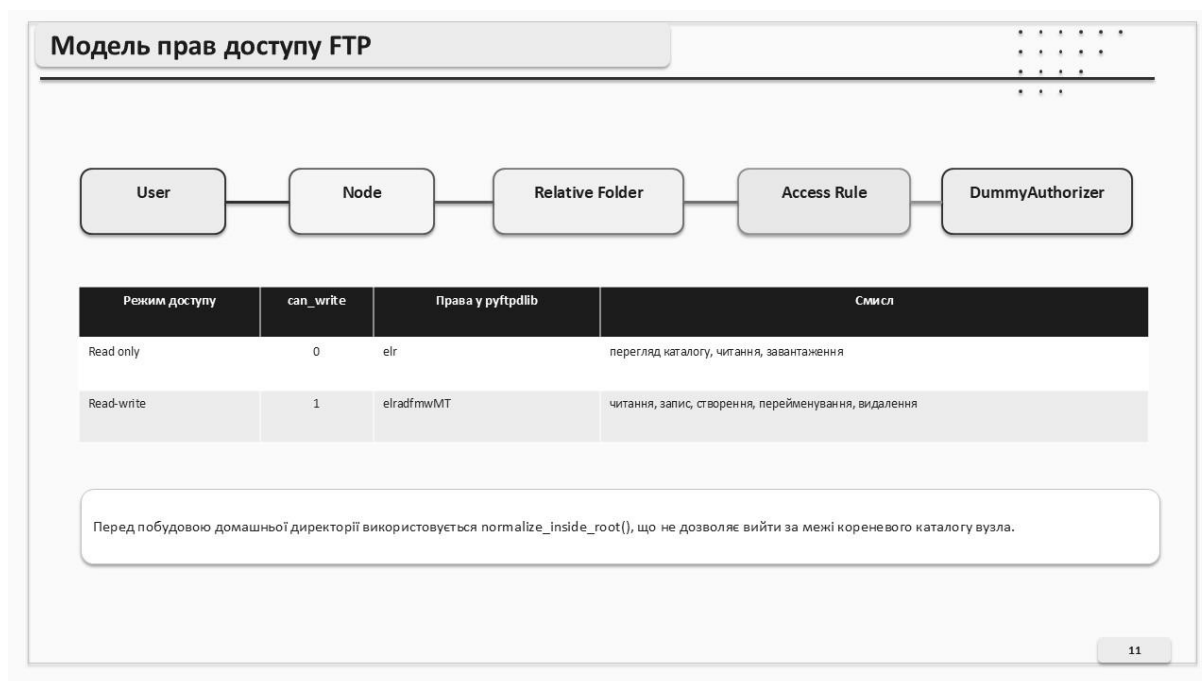


Рисунок Б 11 – Слайд 11

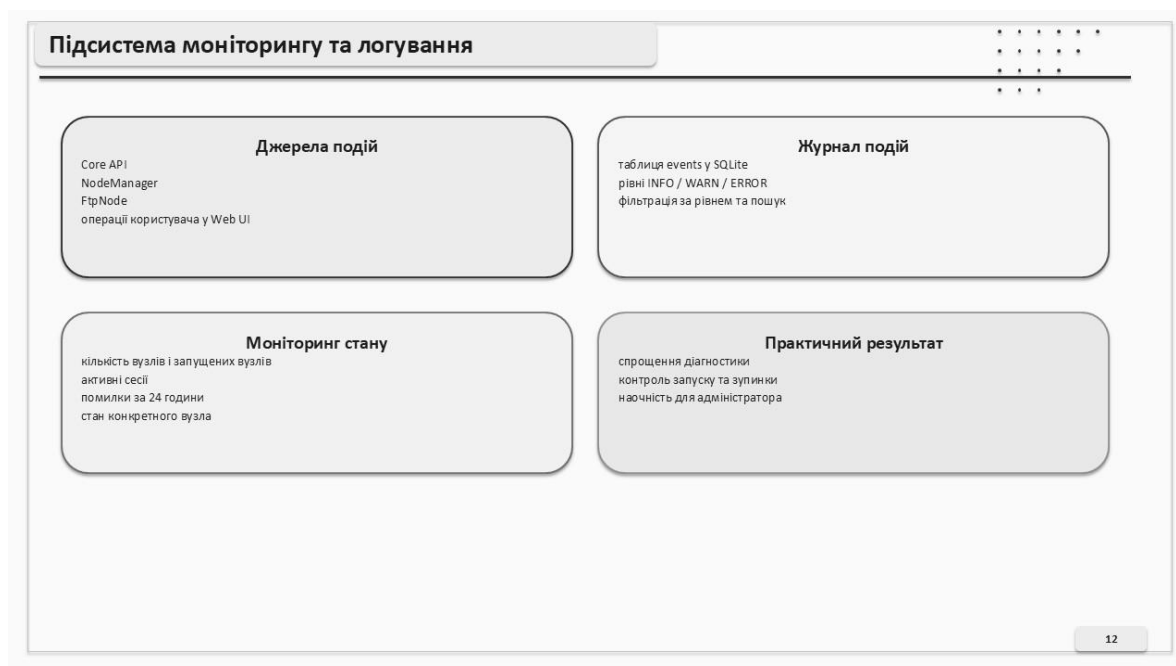


Рисунок Б 12 – Слайд 12

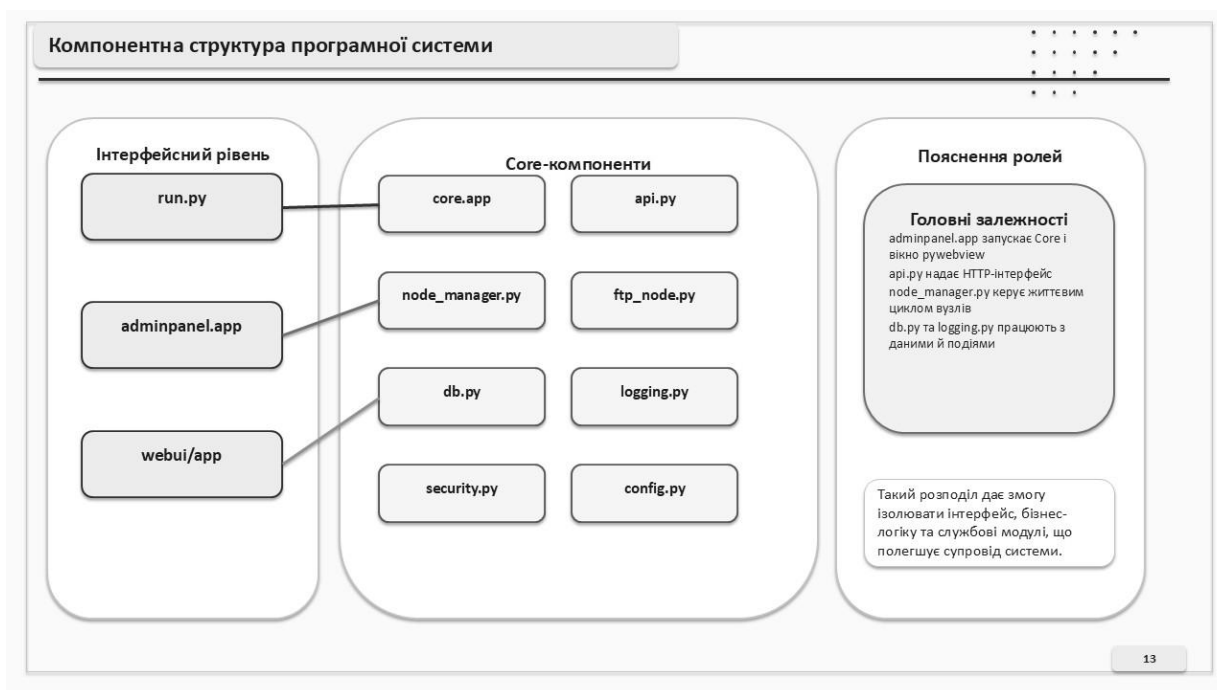


Рисунок Б 13 – Слайд 13

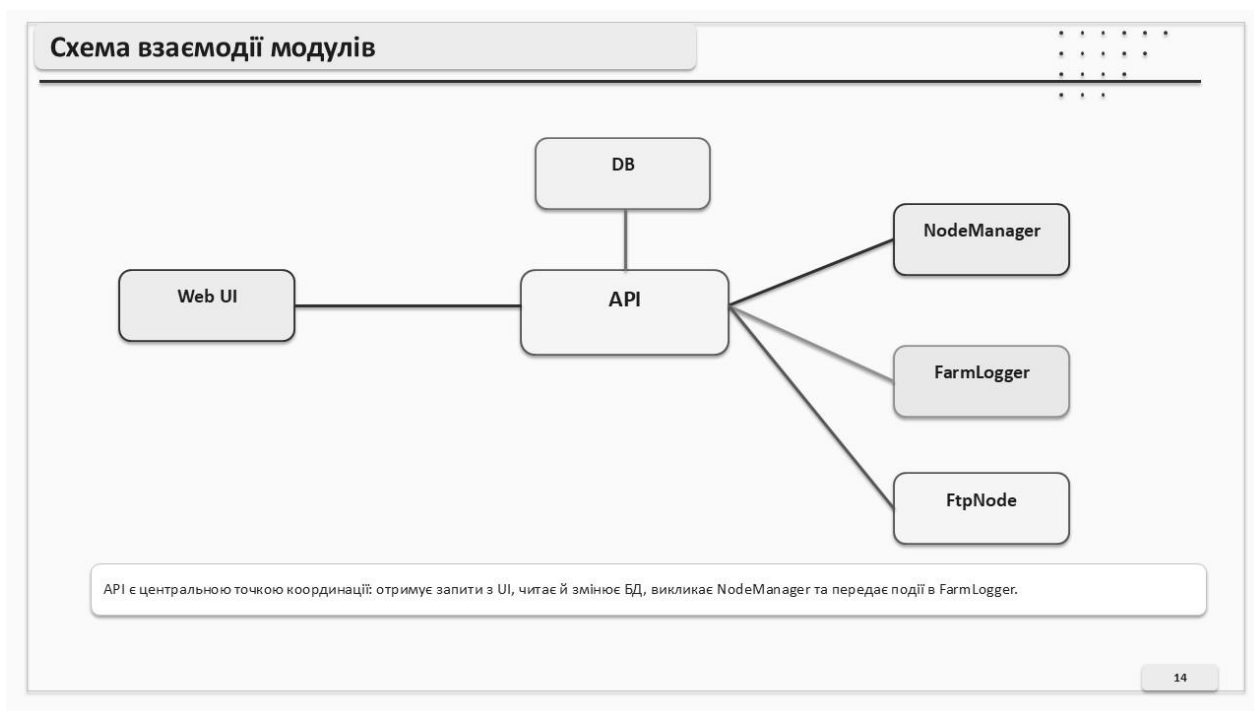


Рисунок Б 14 – Слайд 14

REST API системи

Група endpoint'ів	Приклади методів	Призначення
Nodes	GET /nodes, POST /nodes, POST /nodes/{id}/start	керування конфігурацією та станом FTP-вузлів
Users	GET /users, POST /users, PUT /users/{id}	створення та підтримка облікових записів
Access Rules	GET /rules, POST /rules, DELETE /rules/{id}	призначення прав доступу до каталогів
Logs	GET /logs	перегляд журналу подій
Metrics	GET /metrics	отримання агрегованих показників dashboard
Wizard	GET /wizard/port_free	перевірка доступності порту під час налаштування
Import / Export	POST /import, GET /export	міграція конфігурації системи

15

Рисунок Б 15 – Слайд 15

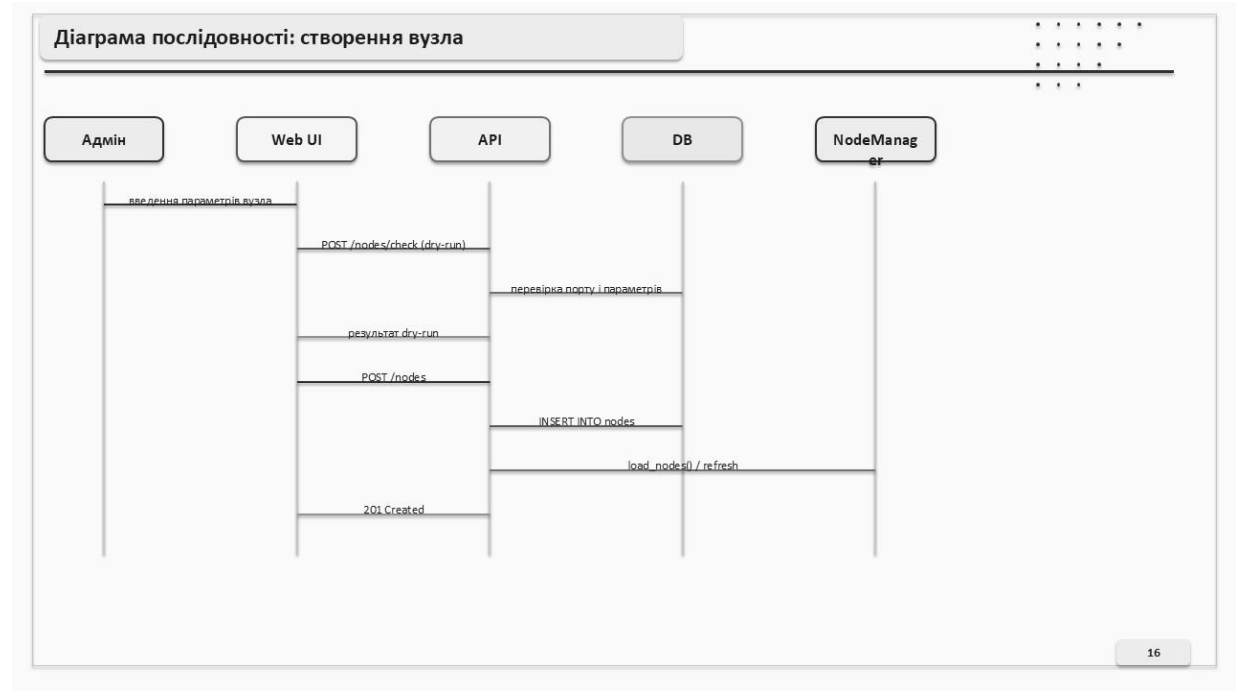


Рисунок Б 16 – Слайд 16



Рисунок Б 17 – Слайд 17

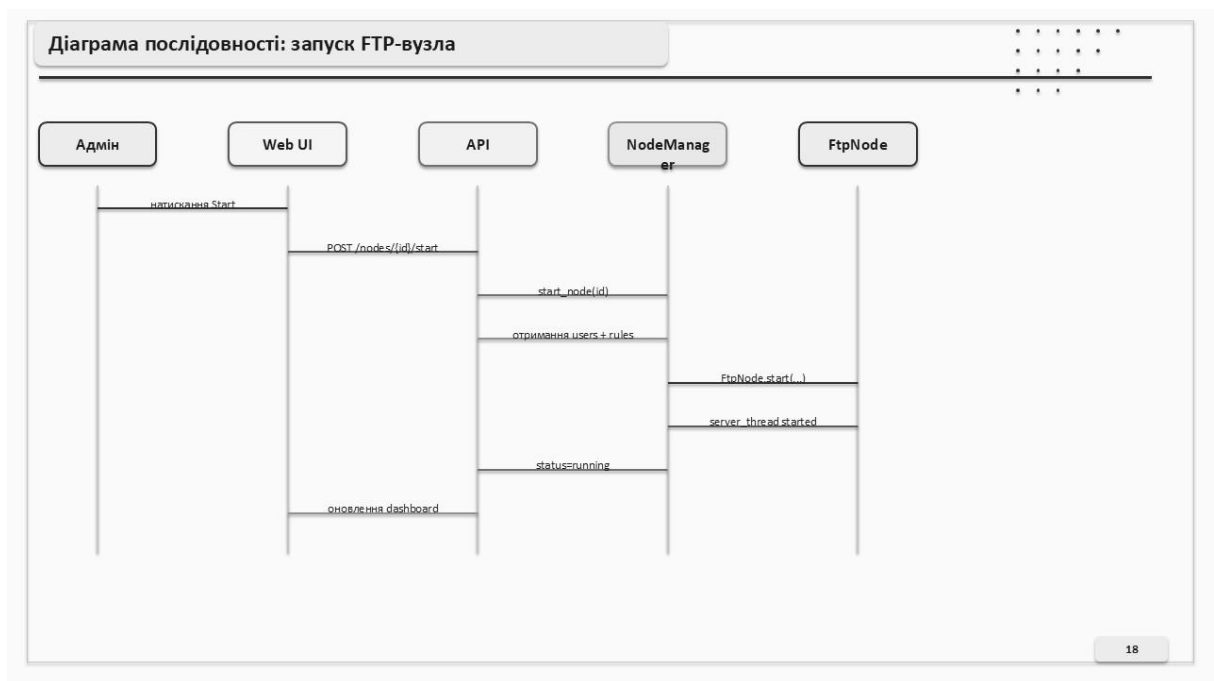


Рисунок Б 18 – Слайд 18

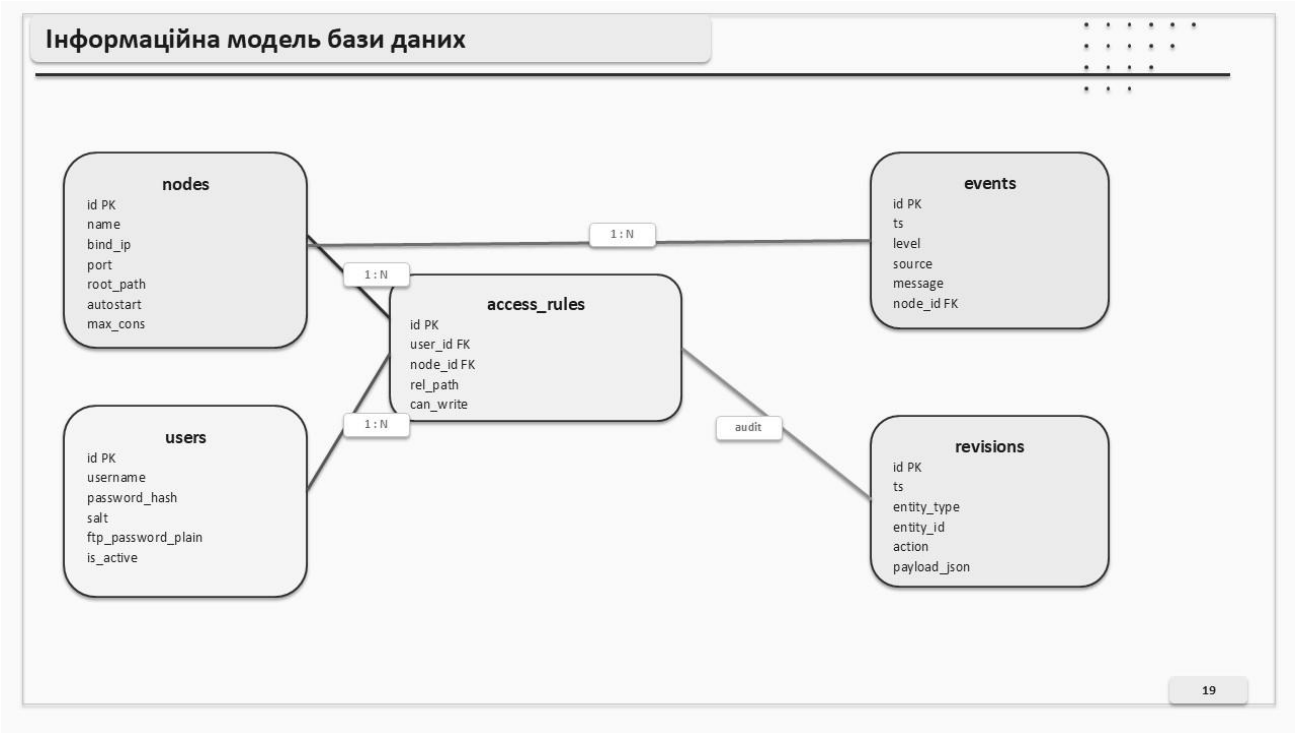


Рисунок Б 19 – Слайд 19

Структура основних таблиць БД

Таблиця	Ключові поля	Призначення
nodes	id, name, bind_ip, port, root_path, autostart	зберігання конфігурації FTP-вузлів
users	id, username, password_hash, salt, ftp_password_plain, is_active	облікові записи користувачів
access_rules	id, user_id, node_id, rel_path, can_write	зв'язок користувача з вузлом і каталогом
events	id, ts, level, source, message, node_id	журнал подій системи
revisions	id, ts, entity_type, entity_id, action, payload_json	аудит змін конфігурації

20

Рисунок Б 20 – Слайд 20

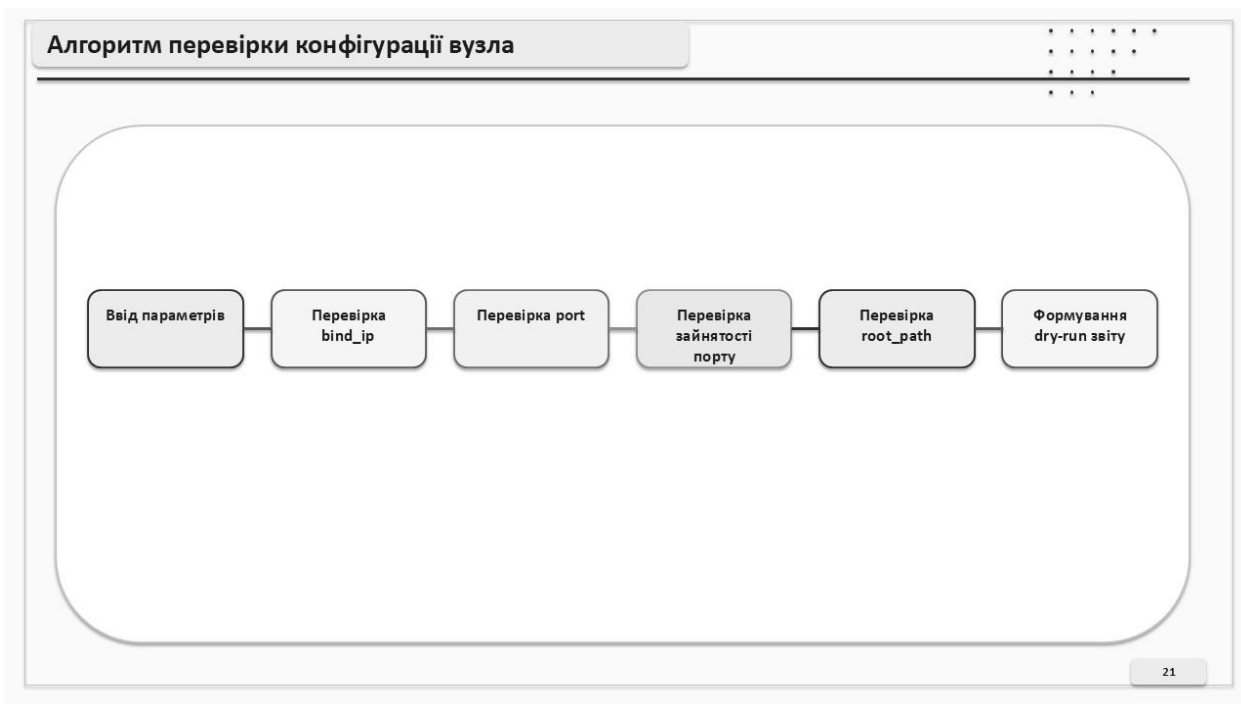


Рисунок Б 21 – Слайд 21

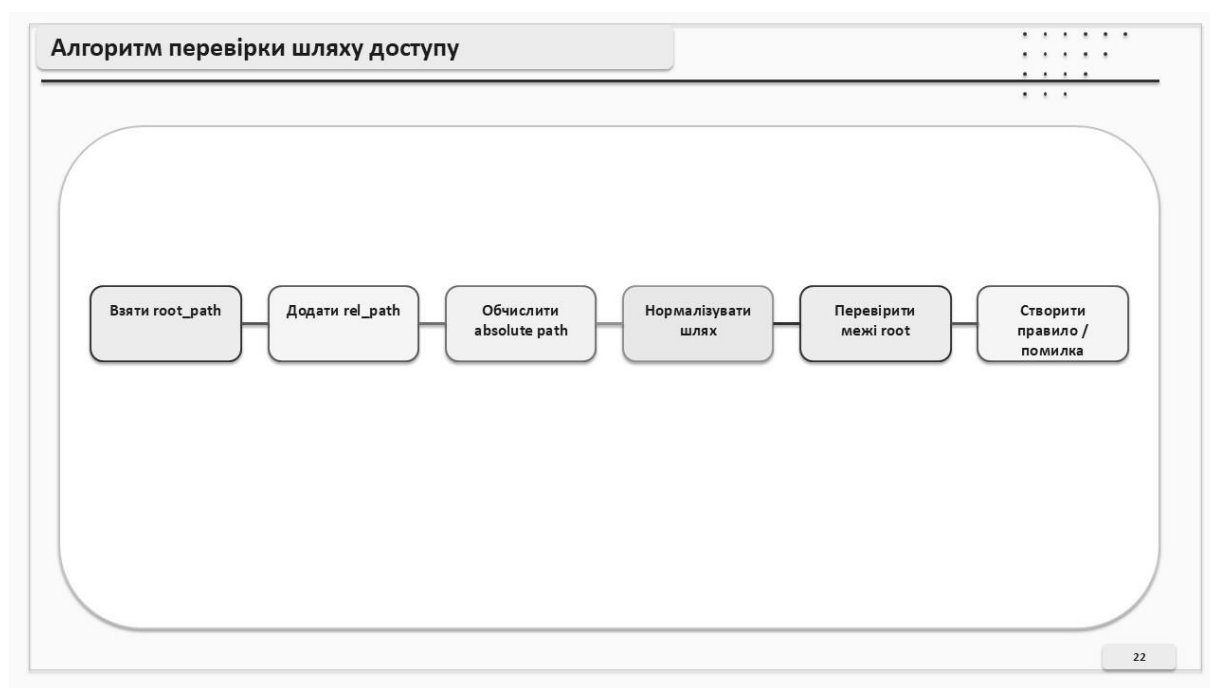


Рисунок Б 22 – Слайд 22

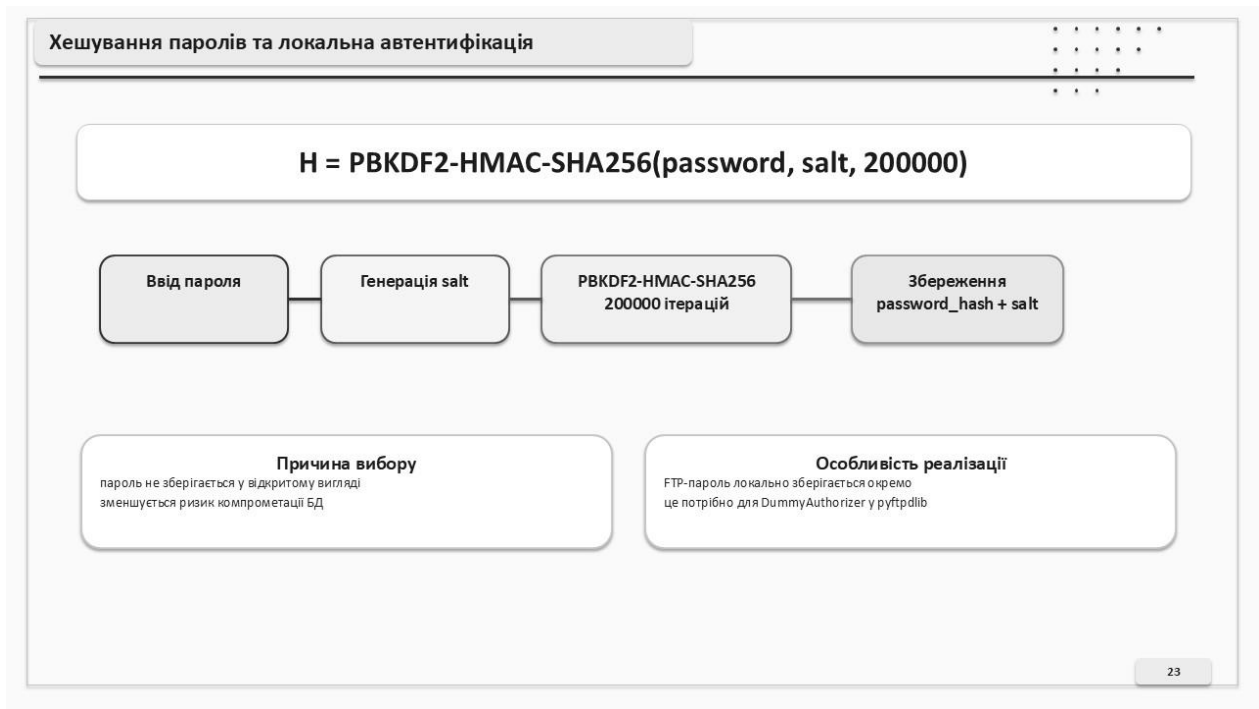


Рисунок Б 23 – Слайд 23

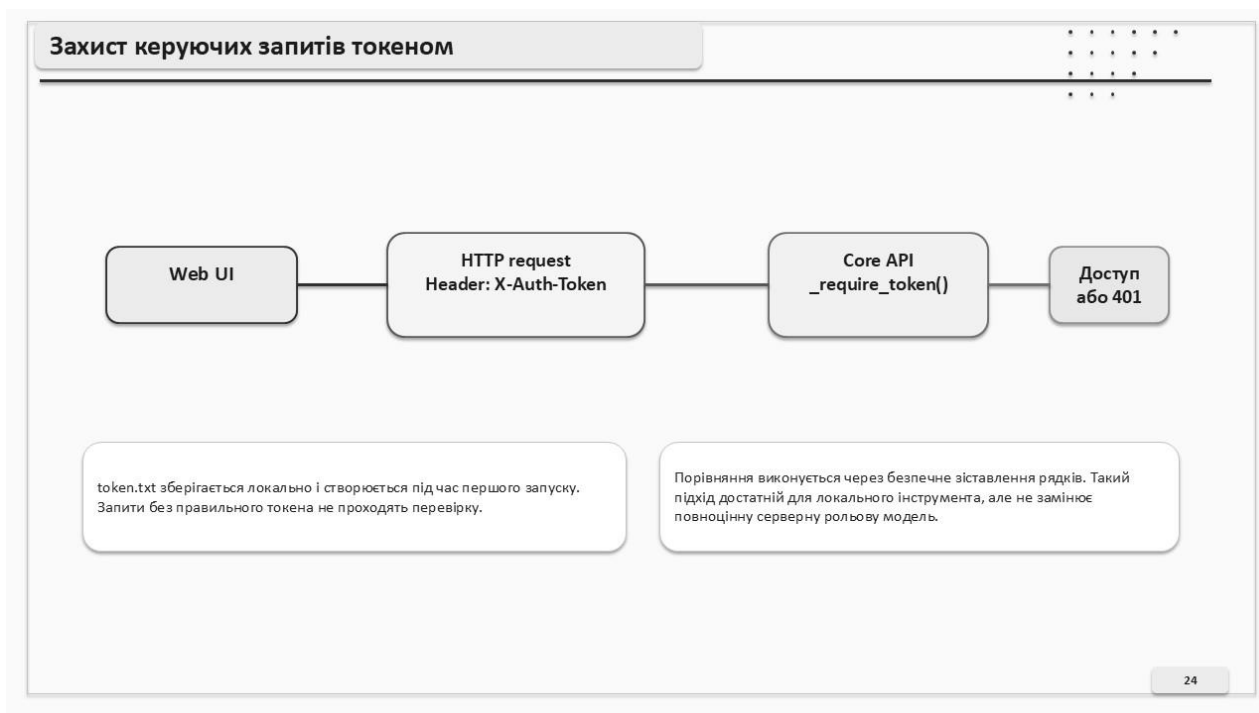


Рисунок Б 24 – Слайд 24

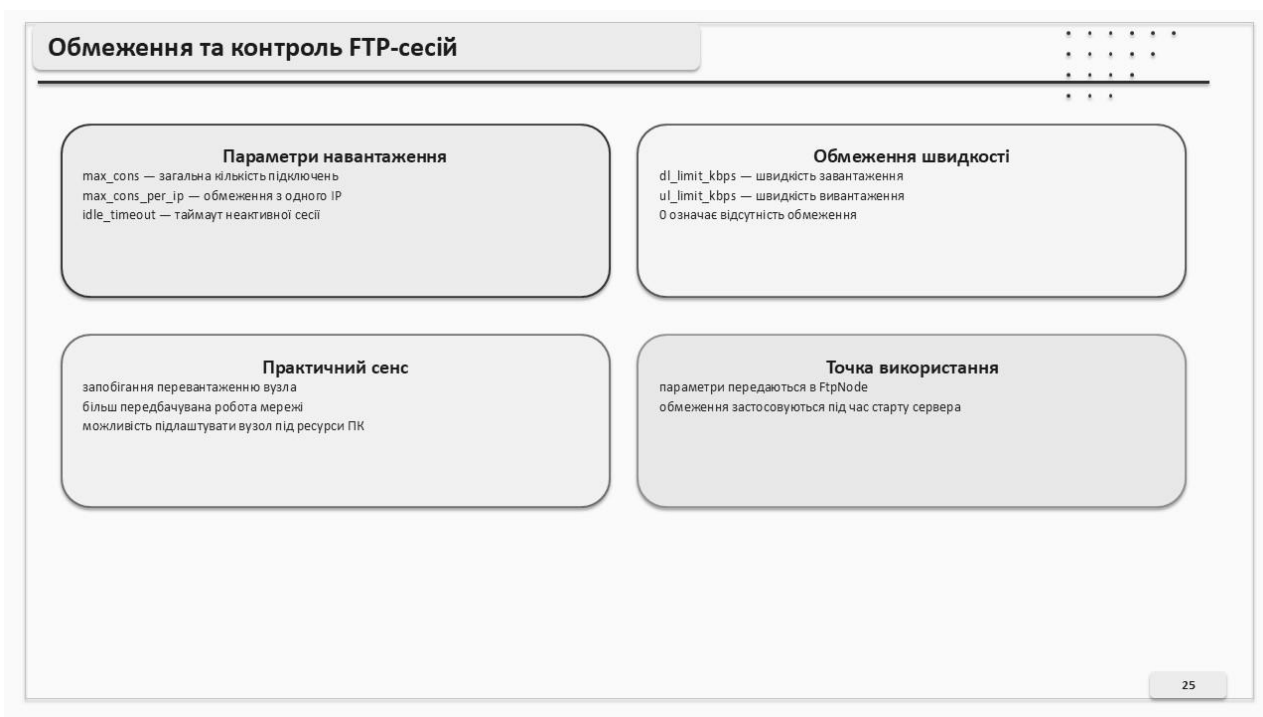


Рисунок Б 25 – Слайд 25

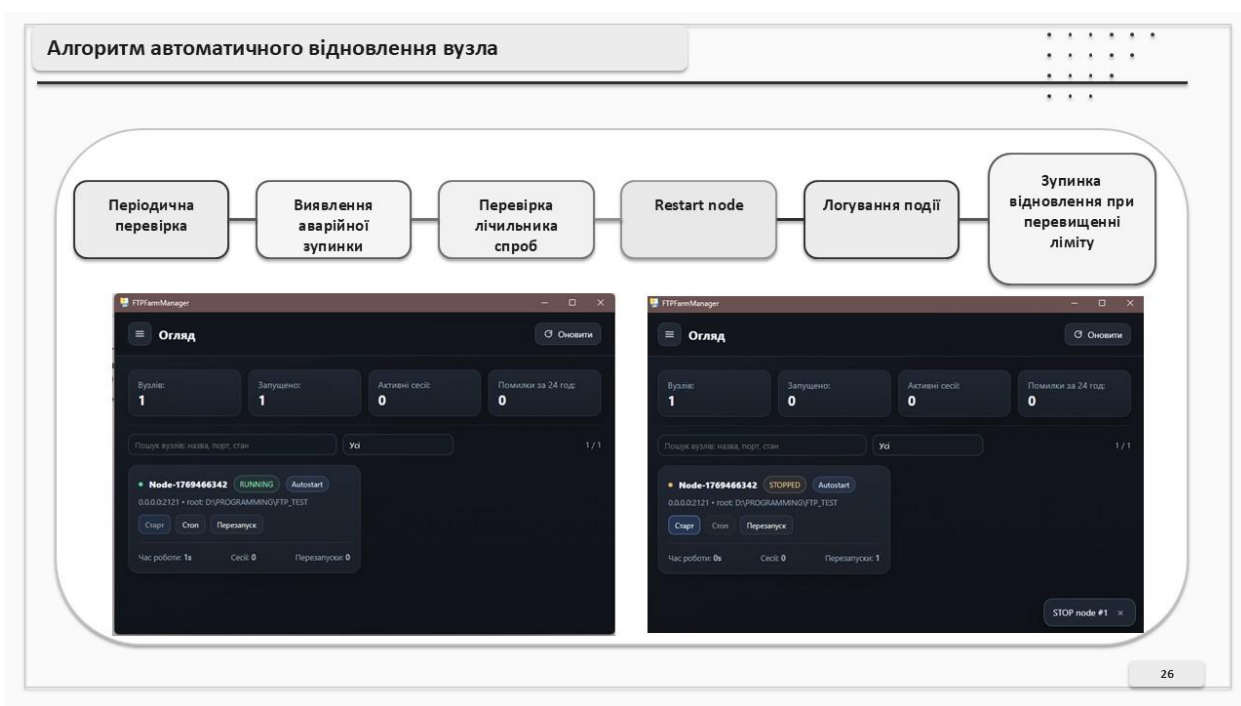


Рисунок Б 26 – Слайд 26

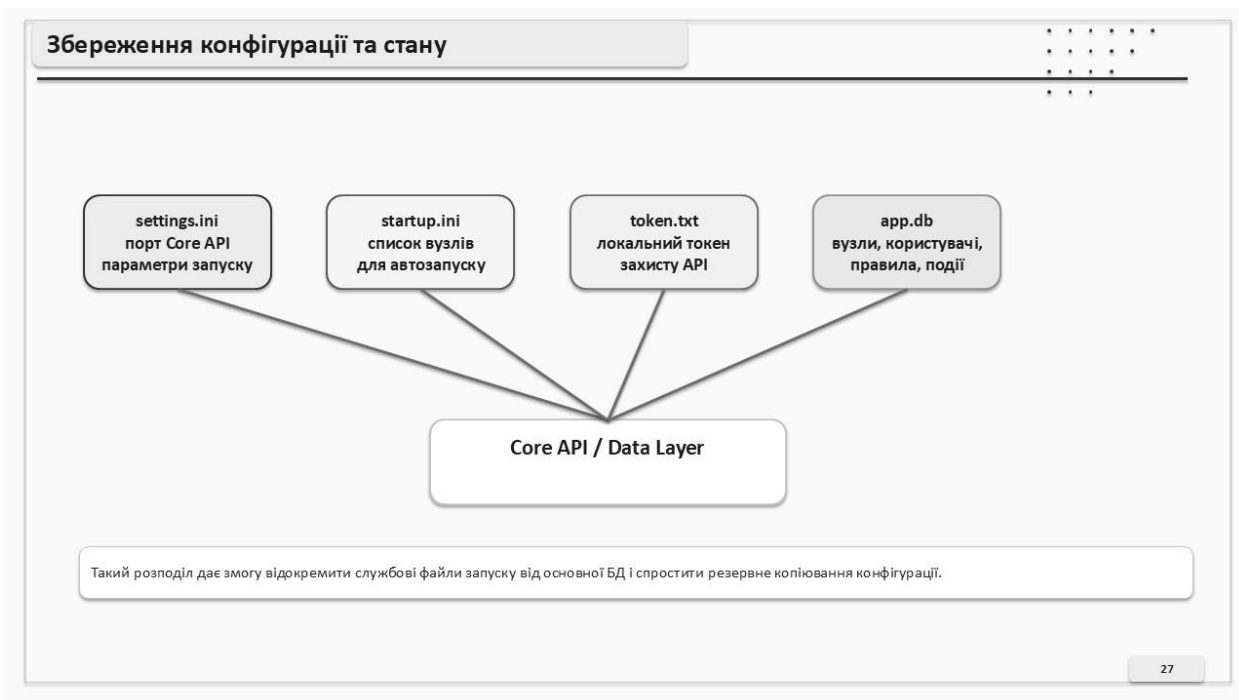


Рисунок Б 27 – Слайд 27

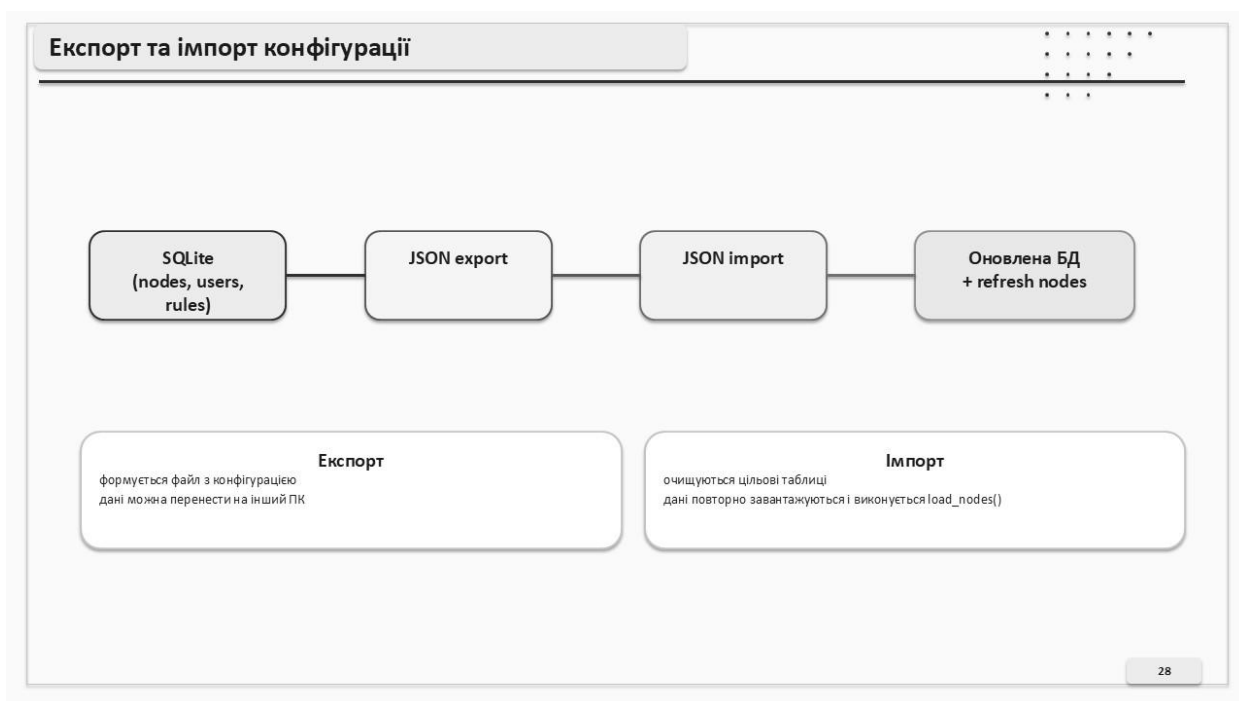


Рисунок Б 28 – Слайд 28

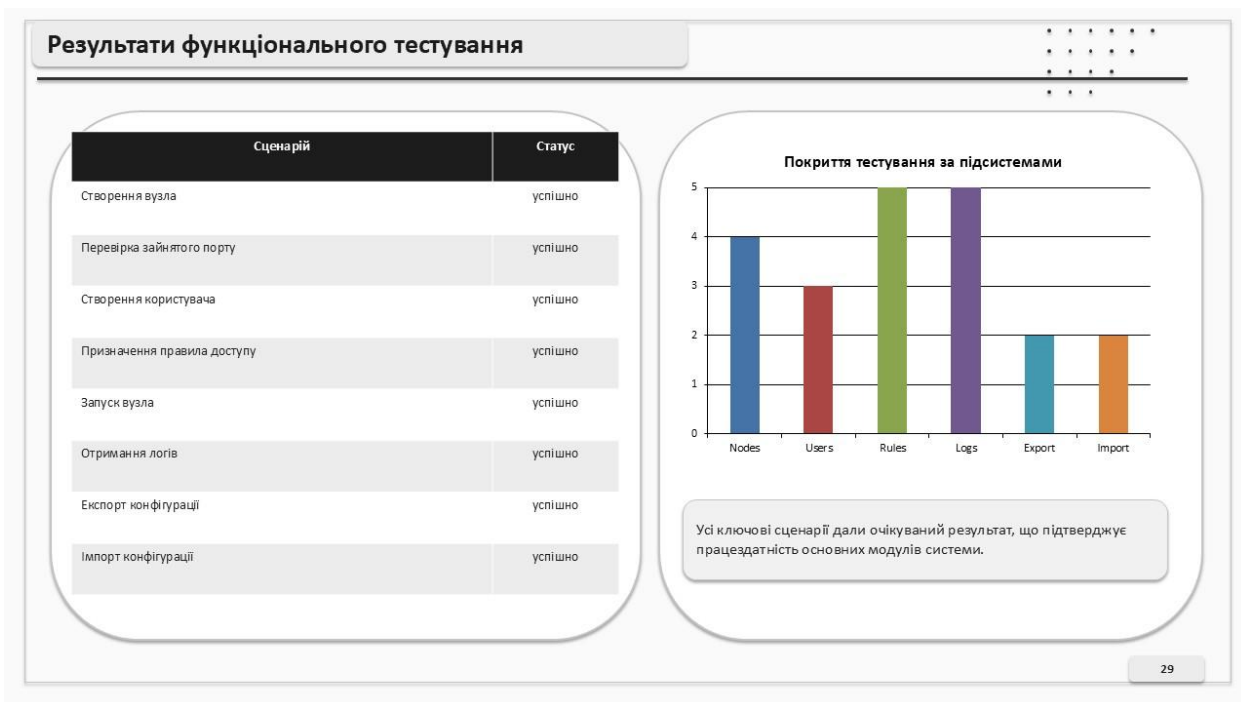


Рисунок Б 29 – Слайд 29

Висновки

Розроблено локальну вебсистему керування FTP-вузлами з інтерфейсом у ruwebview та серверною частиною на aiohttp.

Спроектовано архітектуру, що поєднує Web UI, Core API, SQLite, NodeManager, FarmLogger та набір FTP-вузлів на базі pyftplib.

Реалізовано модулі керування вузлами, користувачами, правами доступу, журналом подій, експортом та імпортом конфігурації.

Забезпечено базові засоби захисту: токен керування, PBKDF2-хешування паролів, перевірку відносних шляхів і контроль параметрів вузла.

Функціональність системи перевірено за ключовими сценаріями, результати тестування підтвердили працездатність розробленого програмного засобу.

30

Рисунок Б 30 – Слайд 30