Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

бакалавр
Освітній рівень

Система керування віддаленим сервером з веб-орієнтованим інтерфейсом
Назва теми

КвРКІ 210251.21.02.47 ПЗ
Шифр

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія та програмування»*
Назва

Виконав: студент IV курсу, група КІ2-21-2 _____ Захар ФЕСІК
Підпис Ініціали, прізвище

Керівник _____ Ольга ПАВЛОВА
Підпис, дата Ініціали, прізвище

Нормоконтролер _____ Тетяна КИСІЛЬ
Підпис, дата Ініціали, прізвище

До захисту допускаю:
зав. кафедри комп'ютерної інженерії та інформаційних систем _____ Ольга ПАВЛОВА
Підпис Ініціали, прізвище

«9» червня 2025 р.

Хмельницький 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

Освітній рівень БАКАЛАВР

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ ТА ПРОГРАМУВАННЯ»

ЗАТВЕРДЖУЮ

Зав. кафедри Ольга ПАВЛОВА

" 10 " 01 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

Захару ФЕСІКУ

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Система керування віддаленим сервером з веб-орієнтованим інтерфейсом

Керівник проекту (роботи), Ольга ПАВЛОВА д.ф., доцент.

Прізвище, ім`я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 07.02.2025 р. № 23

2. Строк подання студентом проекту (роботи) на кафедру 01.06.2025 р.

3. Вихідні дані до проекту (роботи) Завдання на кваліфікаційну роботу

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

Огляд існуючих рішень для вирішення завдання

Обґрунтування вибору компонентів та середовища реалізації

Програмно-технічний засіб для системи керування віддаленим сервером з веб-орієнтованим інтерфейсом

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Блок-схеми алгоритмів роботи системи

Загальна схема архітектури системи

Вигляд створеної системи керування віддаленим сервером

6. Консультанти розділів дипломного проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Тетяна КИСІЛЬ, доцент кафедри КІІС		
Антиплагіат	Андрій НІЧЕПОРУК, доцент кафедри КІІС		

7. Дата видачі завдання «_10_»_____01_____2025 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2025	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.02.2025	виконано
3	Робота над розділом 1 – огляд існуючих рішень для вирішення завдання	01.03.2025	виконано
4	Робота над розділом 2 – обґрунтування вибору компонентів та середовища реалізації	01.04.2025	виконано
5	Робота над розділом 3 – програмно-технічний засіб для системи керування віддаленим сервером з веб-орієнтованим інтерфейсом	29.04.2025	виконано
6	Оформлення пояснювальної записки згідно вимог	25.05.2025	виконано
7	Попередній захист ВКР	26.05.2025	виконано
8	Захист ВКР на засіданні ЕК	Червень 2025 року	

Студент _____ Захар ФЕСІК
Підпис Ініціали, прізвище

Керівник роботи _____ Ольга ПАВЛОВА
Підпис Ініціали, прізвище

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 210251.21.02.47 ПЗ	Пояснювальна записка	73		
			Графічні матеріали			
2		КвРКІ 210251.21.02.47 Е1	Блок-схеми алгоритмів роботи системи	1		
3		КвРКІ 210251.21.02.47 Е1	Загальна схема архітектури системи	1		
4		КвРКІ 210251.21.02.47 Е1	Вигляд створеної системи керування віддаленим сервером	1		

Зм	Арк	№ докум	Підпис	Дата	КвРКІ 210251.21.02.47 ПЗ		
Розробив		Фесік		9.06.25	Відомість проекту	Літера	Аркуш	Аркушів
Перевір.		Павлова		9.06.25		У	1	1
Н. контр.		Кисіль		9.06.25		ХНУ, КІ2-21-2		
Затв.		Павлова		9.06.25				

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Система керування віддаленим сервером з веб-орієнтованим інтерфейсом».

Автор роботи: Захар ФЕСІК.

Керівник роботи: Павлова Ольга Олександрівна.

Пояснювальна записка: 73 с., 34 рис., 6 табл., 5 дод., 59 джерел.

Графічна частина: 3 креслення.

ВІДДАЛЕНИЙ СЕРВЕР, ВЕБ-ІНТЕРФЕЙС, АРХІТЕКТУРА, МОНІТОРИНГ, БАЗА ДАНИХ.

Метою дипломної роботи є розробити систему керування віддаленим сервером з веб-орієнтованим інтерфейсом

Об'єктом дослідження є системи керування віддаленими серверами.

Предметом дослідження є системи керування віддаленим сервером з веб-орієнтованим інтерфейсом.

Під час проведення даного дослідження був використаний метод систематичного огляду літератури для вивчення і аналізу предметної області даного дослідження з текстових джерел інформації.

Підпис студента

30.05.2025

Дата

ЗМІСТ

ВСТУП

У сучасному цифровому середовищі віддалені сервери відіграють ключову роль у забезпеченні функціонування вебсайтів, застосунків та зберігання даних. Традиційне адміністрування серверів через інтерфейс командного рядка (CLI) вимагає значних технічних знань, що створює бар'єр для багатьох користувачів. Сучасні технології сприяють розвитку систем керування з веб-інтерфейсом, які спрощують ці процеси.

Все більше ентузіастів та представників малого бізнесу прагнуть розгортати власні сервери, але наявні рішення не завжди відповідають їхнім потребам. Комерційні панелі, такі як cPanel або Plesk, є дорогими та надлишковими за функціоналом для індивідуального використання. Безкоштовні альтернативи, як-от Cockpit для моніторингу чи Portainer для управління контейнерами, часто вузькоспеціалізовані, що змушує встановлювати кілька інструментів, ускладнюючи адміністрування. Це підкреслює потребу в єдиному, легковаговому та інтуїтивному програмному рішенні.

Об'єктом дослідження є процеси віддаленого адміністрування серверів, моніторингу їх ресурсів та управління сервісами. Предметом дослідження виступає програмно-технічний засіб із веб-інтерфейсом, що інтегрує моніторинг системних показників і керування Docker-контейнерами. Методи дослідження охоплюють аналіз наявних рішень, принципи проєктування клієнт-серверних архітектур, технології веб-розробки (FastAPI, React) та системного програмування (psutil, docker-py).

Метою роботи є розробка полегшеної системи керування віддаленим сервером із веб-інтерфейсом для моніторингу системних параметрів і управління Docker-контейнерами. Для цього буде проведено аналіз існуючих панелей керування, розроблено архітектуру системи, серверну частину для збору даних і клієнтську частину для їх візуалізації та взаємодії з користувачем.

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ДЛЯ ВИРІШЕННЯ ЗАВДАННЯ

1.1 Розвиток систем керування віддаленим сервером з веб-орієнтованим інтерфейсом.

Наприкінці 1990-х років розпочалися перші спроби спрощення адміністрування серверів. Традиційно адміністрування UNIX/Linux-серверів здійснювалося через інтерфейс командного рядка (CLI) [1], вигляд якого зображено на рисунку 1.1. Ця парадигма, хоч і надавала максимальну гнучкість та контроль, вимагала від адміністратора глибоких знань специфічних команд, синтаксису конфігураційних файлів (що зберігаються зазвичай у директорії /etc/), а також розуміння принципів роботи системних служб (демонів). Процес налаштування веб-сервера, бази даних чи поштового клієнта міг включати редагування десятків текстових файлів, що було трудомістким процесом із високим ризиком людської помилки. Для полегшення цих процесів розробники-ентузіасти створили веб-інтерфейси, які дозволяють виконувати типові адміністративні завдання через веб-браузер.

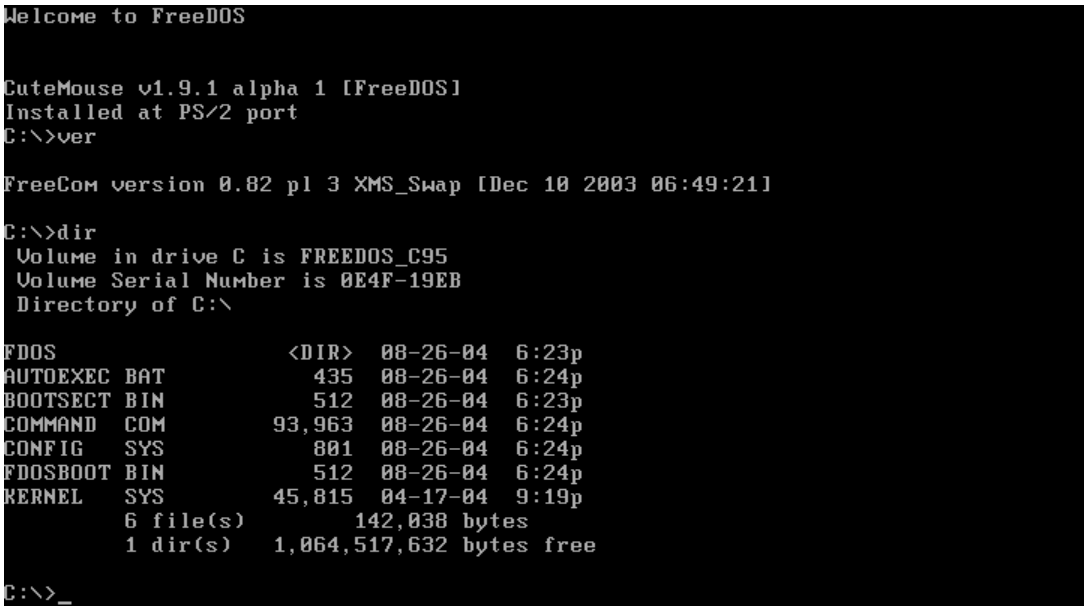


Рисунок 1.1 – Вигляд CLI (Command Line Interface) [55]

Одним із перших таких інструментів став Webmin, випущений у 1997 році. Ця універсальна панель керування, реалізована на основі модулів Perl, забезпечувала доступ до налаштувань операційної системи, керування користувачами, сервісами та іншими компонентами через веб-інтерфейс. Webmin підтримував численні плагіни для адміністрування різноманітних сервісів, що сприяло його швидкому поширенню як інструменту для віддаленого керування Linux/UNIX-серверами. Незважаючи на утилітарний і дещо застарілий інтерфейс, ключовою перевагою Webmin була висока функціональність, забезпечена широким набором модулів для виконання різноманітних адміністративних завдань.

Це означало, що для більшості програмного забезпечення можна було знайти відповідний модуль Webmin або розробити власний, що було значною перевагою в період, коли альтернативні рішення для спрощеного адміністрування серверів були відсутні [2]. На рисунку 1.2 зображено вигляд інтерфейсу користувача програми Webmin.

Рисунок 1.2 – Вигляд Webmin [56]

У сфері комерційного хостингу в той же період з’явилися панелі керування веб-хостингом, такі як cPanel (1996 рік) і Plesk (2001 рік), які надавали можливість

керування веб-сайтами, доменами та поштовими службами через веб-інтерфейс. Ці інструменти були орієнтовані переважно на хостинг-провайдерів, але загалом сприяли популяризації концепції віддаленого керування веб-серверами для користувачів із обмеженими технічними знаннями, зокрема власників веб-сайтів [3-4]. На рисунку 1.3 зображено вигляд інтерфейсу користувача програми cPanel.

Рисунок 1.3 – Вигляд cPanel [57]

На початку 2000-х років, із поширенням широкосмугового інтернету та розвитком домашніх мереж, виникла потреба у вбудованих системах із веб-інтерфейсами для спрощеного керування. Прикладом такого рішення є проєкт OpenWrt, започаткований приблизно у 2004 році як відкрите програмне забезпечення для маршрутизаторів. OpenWrt дозволяв ентузіастам встановлювати повноцінну операційну систему Linux із підтримкою пакетів і веб-інтерфейсом LuCI, замінюючи обмежені заводські прошивки.

Цей проект став важливим кроком, продемонструвавши можливість реалізації зручного веб-керування на пристроях із обмеженими обчислювальними

ресурсами. Підхід OpenWrt здобув широку популярність і вплинув на інші проекти для мережевих пристроїв. Зокрема, веб-інтерфейси з'явилися в спеціалізованих дистрибутивах, таких як pfSense (брандмауер на базі FreeBSD), а також у різноманітних шлюзах для Інтернету речей (IoT) [5]. На рисунку 1.4 зображено вигляд інтерфейсу користувача програми OpenWrt.

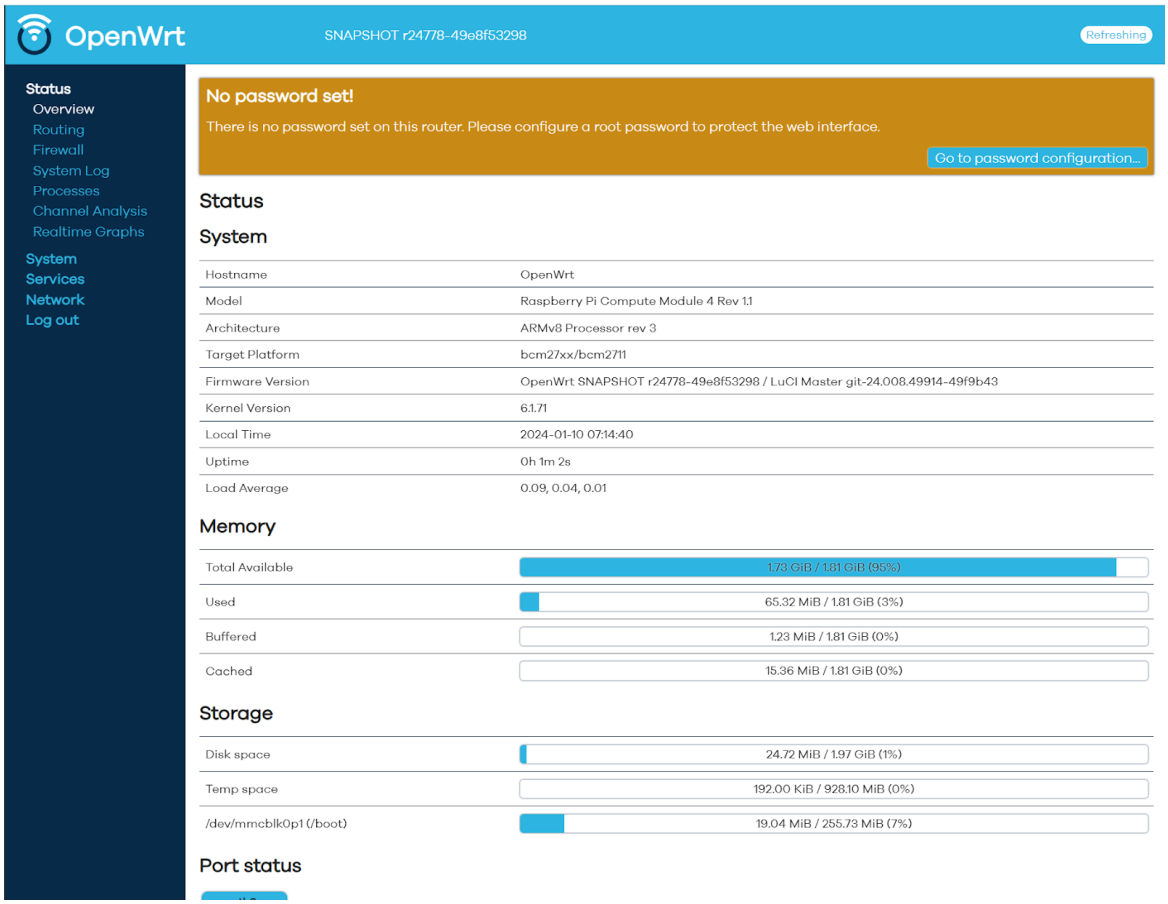


Рисунок 1.4 – Вигляд OpenWrt [58]

Історично веб-інтерфейси для адміністрування еволюціонували від простих панелей керування окремими серверами до комплексних платформ, здатних одночасно керувати кількома вузлами та різноманітними типами навантажень. Цей розвиток супроводжувався значною еволюцією в підходах до безпеки. Якщо ранні веб-панелі часто працювали по незахищеному протоколу HTTP і мали вразливості до атак типу міжсайтового скриптингу (XSS) та підробки міжсайтових запитів (CSRF), то сучасні рішення стандартно використовують шифрування (HTTPS),

двофакторну автентифікацію (2FA), рольову модель доступу (RBAC) та інші передові механізми захисту [6-7].

Важливим є зміна ставлення спільноти Linux до цих інструментів, якщо раніше деякі адміністратори вважали їх надмірними або потенційно вразливими то згодом веб-інтерфейси довели свою ефективність і практичну цінність. У середовищах із високими вимогами до безпеки тривалий час перевага віддавалася ручному керуванню через CLI. Проте вдосконалення механізмів безпеки та поява гігантських хмарних платформ, таких як Amazon Web Services (AWS), Microsoft Azure та Google Cloud Platform (GCP), остаточно легітимізували веб-інтерфейс як основний засіб управління інфраструктурою.

Ці корпоративні рішення, разом із такими інструментами як Red Hat Cockpit та VMware vCenter, сприяли широкому визнанню веб-орієнтованих систем адміністрування в екосистемі Linux. Починаючи з 2010-х років, дистрибутиви, такі як Fedora Server і Red Hat Enterprise Linux (RHEL), почали інтегрувати Cockpit як стандартний інструмент для первинного налаштування. Це вказує на зрілість технології: за умови дотримання належних заходів безпеки веб-інтерфейси значно спрощують адміністрування, не знижуючи надійності системи. [8-13].

1.2 Порівняння існуючих програмно-технічних засобів

У сучасних умовах все більше користувачів виявляють зацікавленість у створенні власного міні-сервера як вдома, так і у віддаленому середовищі. Такий сервер може бути використаний для розміщення персональних проектів, розміщення ігрових серверів або реалізації інших сервісів з обмеженим колом доступу. Однак процес налаштування сервера часто передбачає наявність спеціалізованих знань, зокрема у сфері конфігурації систем Linux, що ускладнює його реалізацію для пересічного користувача.

У зв’язку з цим зростає актуальність застосування веб-орієнтованих систем керування серверами, які надають графічний інтерфейс для адміністрування та

можливість автоматизованого розгортання необхідного функціоналу через доступні команди. Проте постає питання вибору відповідного програмного рішення серед численних панелей керування, що представлені на ринку, з огляду на їх функціональні можливості, цінову політику та вимоги до досвіду користувача.

Більшість комерційних рішень, таких як cPanel, орієнтовані на широкий спектр задач. Зокрема, cPanel передбачає адміністрування файлової системи, керування базами даних (MySQL, PostgreSQL), налаштування поштових служб, безпекових політик (SSL/TLS, SSH), інсталяцію CMS через Softaculous, збір статистики (AWStats), а також інструменти конфігурації веб-сервера. Архітектурно cPanel складається з двох частин: інтерфейсу для кінцевого користувача та WebHost Manager (WHM), що використовується адміністраторами для управління всім сервером та реселлерськими акаунтами. Проте базова вартість ліцензії на дане ПЗ становить від $17 до $60 на місяць, що є фінансово невигідним для непрофесійного користувача, який не потребує більшості з перелічених функцій [3- 5].

З урахуванням цього, доцільним є створення полегшеної веб-панелі з мінімально необхідним функціоналом – зокрема, моніторингом системних ресурсів, можливістю запуску та керування контейнерами Docker та переглядом інформації про стан системи. Такий підхід забезпечує зменшення навантаження на інтерфейс, спрощує експлуатацію та усуває потребу у використанні платного програмного забезпечення з надлишковим функціоналом. Отже, розробка власного програмного засобу у цій ніші є обґрунтованою як з технічної, так і з економічної точки зору.

Іншою провідною комерційною панеллю є Plesk. Вартість її використання починається орієнтовно від $12 на місяць. Plesk підтримує як Linux, так і Windows Server, що є її важливою перевагою. Панель відома своїм сучасним інтерфейсом та потужними розширеннями, такими як WordPress Toolkit для комплексного управління сайтами на WordPress, інтеграцією з Git та підтримкою контейнеризації Docker.

Веб-інтерфейс користувача Plesk розроблений з урахуванням потреб користувачів з різним рівнем технічної підготовки, що полегшує освоєння функцій управління доменами, веб-сайтами та базами даних. Важливою характеристикою панелі є наявність засобів автоматизації стандартних адміністративних завдань, таких як створення резервних копій та оновлення програмного забезпечення системи, що сприяє оптимізації процесів обслуговування сервера.

До переваг Plesk також належать: можливість інтеграції з поширеними системами управління контентом (CMS), зокрема WordPress, Joomla та Drupal; підтримка технології контейнеризації Docker для розгортання програмних додатків; наявність комплексних інструментів для забезпечення безпеки, включаючи веб-фільтр ModSecurity, систему запобігання вторгнень Fail2Ban та інтеграцію з сервісом Let's Encrypt для управління SSL-сертифікатами. Архітектура системи забезпечує масштабованість, дозволяючи адмініструвати від одного до значної кількості веб-сайтів на одному сервері.

Незважаючи на зазначені переваги, Plesk має певні недоліки. Комерційний характер ліцензування продукту зумовлює додаткові фінансові витрати на експлуатацію хостингу, особливо у порівнянні з безкоштовними аналогами. Окрім того, функціонування панелі характеризується відносно високим споживанням системних ресурсів, що може негативно впливати на загальну продуктивність серверів з обмеженою конфігурацією апаратного забезпечення.

У порівнянні з методом прямої конфігурації сервера через інтерфейс командного рядка (CLI), використання Plesk може накладати певні обмеження на гнучкість детального налаштування серверних параметрів. Залежність від єдиного розробника програмного забезпечення також може розглядатися як потенційний ризик, пов'язаний з політикою оновлень, довгостроковою технічною підтримкою продукту та можливими змінами умов ліцензування. Процес міграції даних та конфігурацій з Plesk на альтернативні панелі управління часто характеризується підвищеною технічною складністю.

Узагальнюючи, Plesk позиціонується як одне з провідних рішень на ринку систем управління хостингом, особливо для проектів середнього та великого масштабу, де переваги, пов'язані з автоматизацією та централізованим адмініструванням, можуть компенсувати витрати на ліцензування. Проте, аналогічно до іншої поширеної панелі, cPanel, основний функціонал Plesk переважно орієнтований на управління веб-сайтами, базами даних (БД), доменними іменами та супутніми сервісами. Значна частина цих спеціалізованих можливостей може бути надлишковою для користувачів, чиї потреби обмежуються загальним адмініструванням сервера без глибокої спеціалізації на веб-хостингу. На рисунку 1.5 зображено вигляд інтерфейсу користувача програми Plesk. [3-5, 16-17]

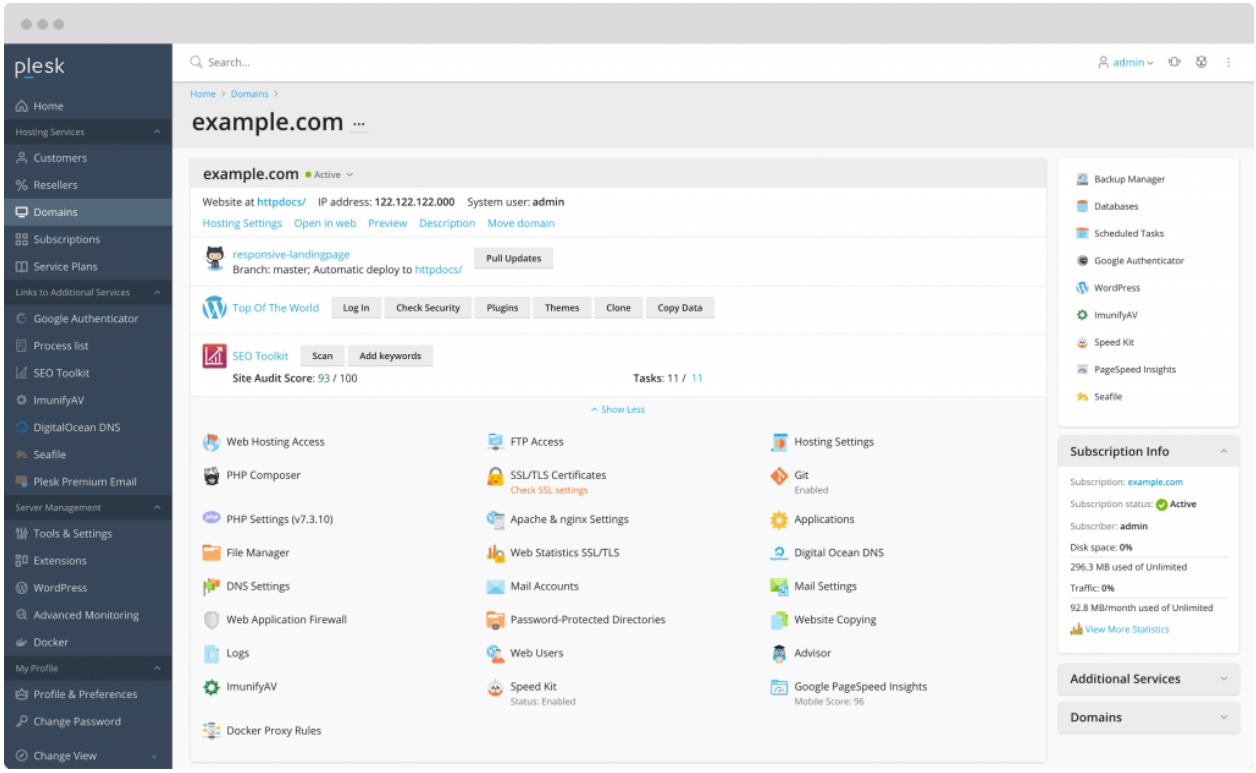


Рисунок 1.5 – Вигляд Plesk [59]

Значна частина існуючих панелей управління серверами орієнтована на комерційне використання, що передбачає платну модель розповсюдження, нерідко з багаторівневою системою ліцензування, де розширений функціонал доступний за додаткову плату. Характерною особливістю таких систем є наявність значної

кількості програмних залежностей, що призводить до підвищеного споживання ресурсів сервера. Деактивація цих залежностей часто є ускладненою або неможливою без порушення працездатності основних функцій панелі. Внаслідок цього, подібні рішення є менш придатними для серверів з обмеженими апаратними ресурсами, оскільки їх функціонування може споживати суттєву частку обчислювальної потужності.

Водночас, на ринку існують альтернативні рішення для управління серверами, які не передбачають ліцензійних відрахувань та характеризуються меншим впливом на системні ресурси. До таких рішень належать продукти з відкритим вихідним кодом (OpenSource), що розповсюджуються на безоплатній основі та надають можливість модифікації їх функціоналу за наявності у користувача відповідних технічних компетенцій. Типовими прикладами такої системи є проєкти Cockpit Project, Hestia Control Panel (HestiaCP), Portainer.

Cockpit Project – це сучасний веб-інтерфейс для адміністрування Linux-серверів. Його ключова перевага – низькі системні вимоги. Cockpit працює як системний сервіс systemd, що активується лише за запитом (on-demand), тобто він не споживає ресурси у фоновому режимі. Він не є прошарком абстракції, а безпосередньо взаємодіє з системними API (наприклад, D-Bus), що гарантує відображення реального стану системи. Панель надає доступ до управління службами, мережею, користувачами, дисковим простором та вбудований термінал. Однак, можливості Cockpit у сфері управління веб-хостингом є обмеженими. У стандартній конфігурації відсутні інструменти для адміністрування баз даних та доменних імен на тому рівні, що пропонують хостинг-панелі.

Інтерфейс Cockpit мінімалістичний і при цьому цьому зберігає функціональність, необхідну для ефективного управління серверами. Це робить його доступним як для досвідчених адміністраторів, так і для новачків, які ще не повністю опанували командний рядок Linux. Панель надає зручний доступ до основних системних функцій, таких як управління службами, мережевими з'єднаннями, користувачами та дисковим простором.

Однією з ключових переваг Cockpit є його низькі системні вимоги. На відміну від багатьох інших панелей управління, він споживає мінімальну кількість ресурсів сервера, що дозволяє використовувати його навіть на невеликих віртуальних машинах без помітного впливу на продуктивність. Крім того, система не вимагає встановлення додаткових залежностей або специфічної конфігурації сервера.

Водночас, система Cockpit має низку функціональних обмежень. При порівняльному аналізі з комерційними платформами, такими як Plesk або cPanel, можливості Cockpit у сфері спеціалізованого управління веб-хостингом є менш розширеними. Зокрема, у стандартній конфігурації відсутні інтегровані інструменти для комплексного управління віртуальними хостами, адміністрування баз даних та доменних імен на тому рівні абстракції, що пропонується типовими хостинг-панелями. Зазначені аспекти роблять Cockpit менш придатним для використання у ролі основної панелі управління для надання послуг комерційного веб-хостингу [18-21]. На рисунку 1.6 зображено вигляд інтерфейсу користувача програми Cockpit Dashboard.

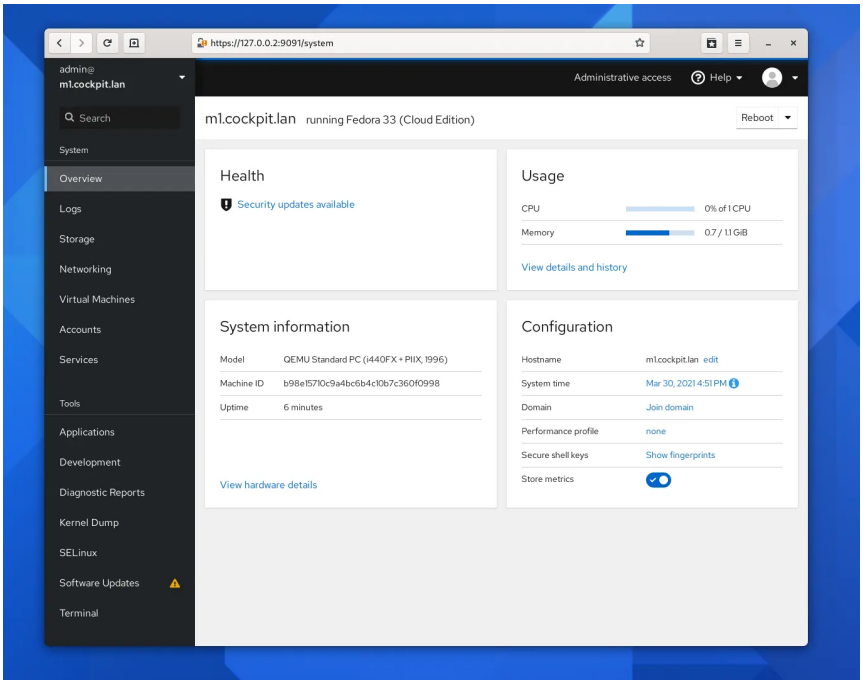


Рисунок 1.6 – Вигляд Cockpit Dashboard [19]

Hestia Control Panel (HestiaCP) – це форк іншого популярного проекту VestaCP, що є безкоштовною панеллю управління з відкритим кодом. HestiaCP позиціонується як прямий конкурент cPanel/Plesk для користувачів, які потребують повноцінного функціоналу для веб-хостингу, але не готові платити за ліцензію. Вона підтримує веб-сервери Nginx та Apache2, керування DNS, поштовими серверами, базами даних MySQL та PostgreSQL. HestiaCP активно розвивається спільнотою, має вбудований фаєрвол та підтримує Let's Encrypt для автоматичного отримання SSL-сертифікатів. Однак, як і комерційні аналоги, вона встановлює велику кількість залежностей і орієнтована саме на хостинг сайтів, що може бути надлишковим для простого управління сервером [22-24]. На рисунку 1.7 зображено вигляд інтерфейсу користувача програми Hestia Control Panel (HestiaCP).

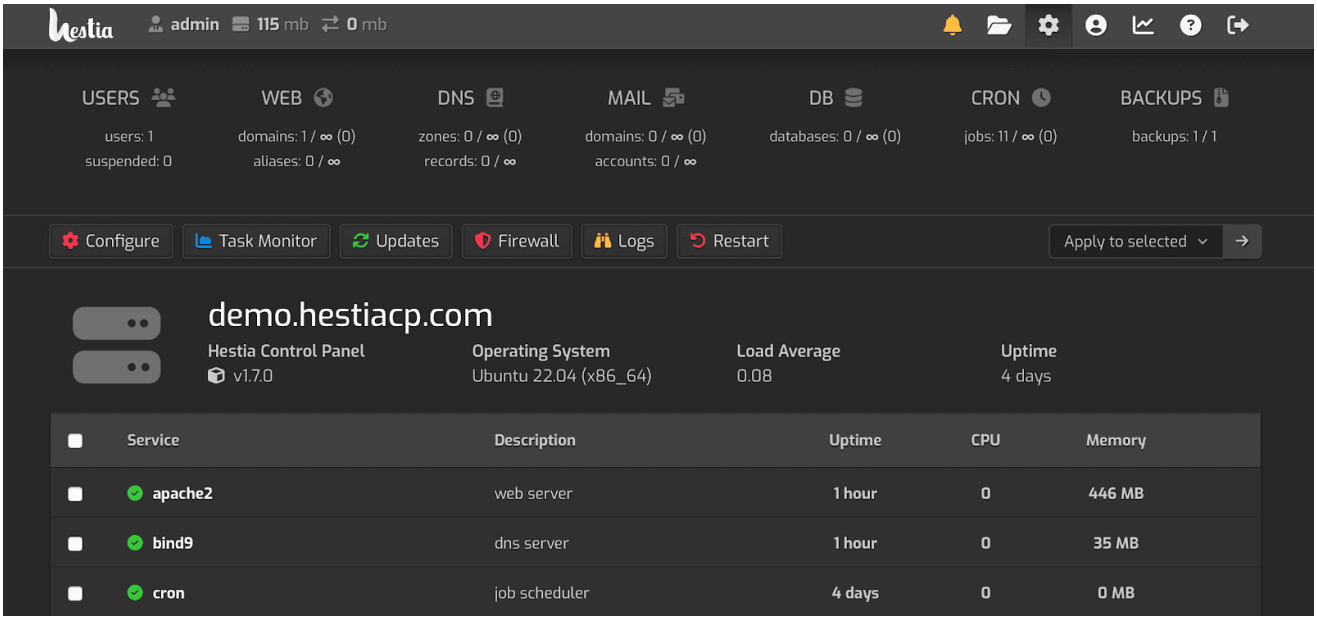


Рисунок 1.7 – Вигляд інтерфейсу користувача Hestia Control Panel (HestiaCP) [24]

Portainer – це ще одне популярне OpenSource-рішення, але з вузькою спеціалізацією. Воно призначене виключно для управління середовищами контейнеризації: Docker, Docker Swarm, Kubernetes та Nomad. Portainer надає потужний та інтуїтивно зрозумілий графічний інтерфейс для розгортання контейнерів, управління образами, томами (volumes), мережами та стеками (docker-

compose). Для користувача, основною задачею якого є запуск та адміністрування Docker-додатків, Portainer є майже ідеальним інструментом. Проте він зовсім не надає функцій для управління самою операційною системою: моніторингу навантаження на CPU, управління системними службами, користувачами чи мережевими інтерфейсами хост-машини [25-26]. На рисунку 1.8 зображено вигляд інтерфейсу користувача програми Portainer.

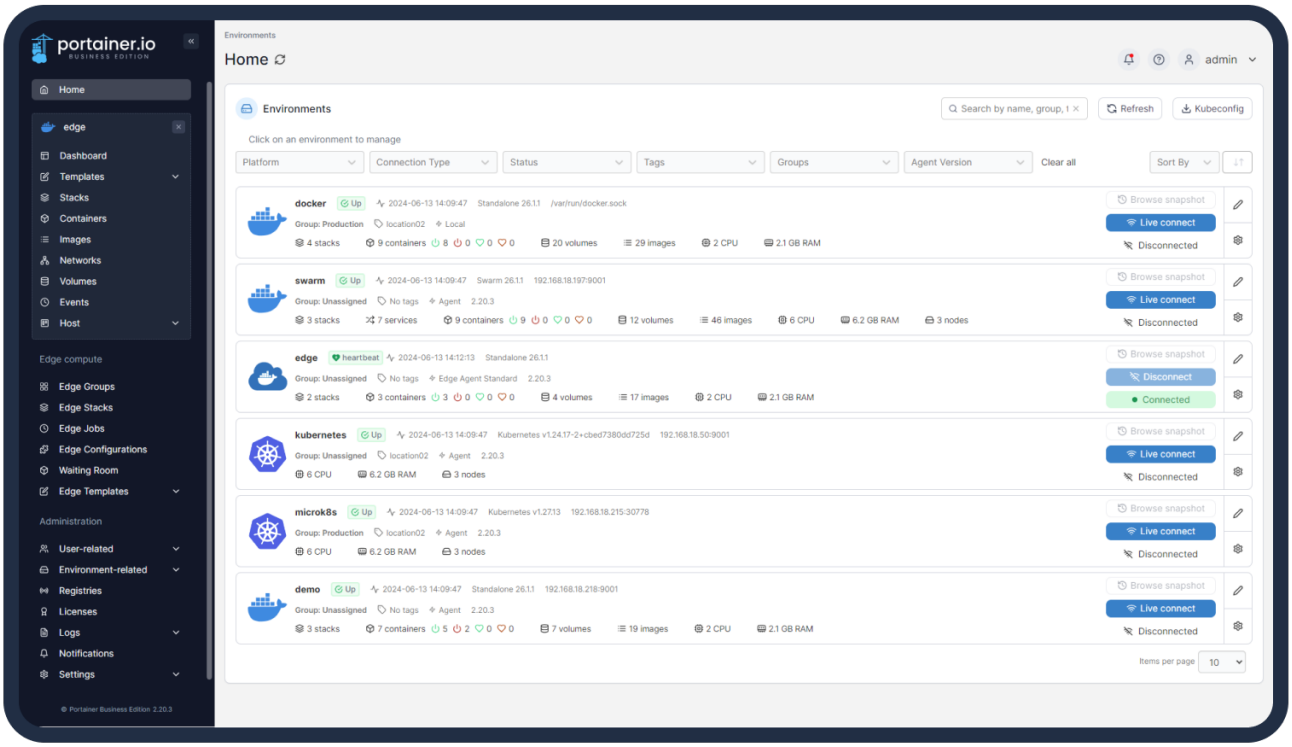


Рисунок 1.8 – Вигляд інтерфейсу користувача Portainer [25]

На основі проведеного аналізу сформовано таблицю 1.1, яка узагальнює порівняльні характеристики систем управління серверами (cPanel, Plesk, Cockpit Project, HestiaCP, Portainer), включаючи функціональність, операційні системи, автоматизацію та цінову політику.

Таблиця 1.1 – Порівняльна характеристика панелей керування веб-хостингом: cPanel, Plesk, Cockpit Project, HestiaCP, Portainer.

Характеристика	cPanel	Plesk	Cockpit Project	HestiaCP	Portainer
Ліцензія	Платна	Платна	Безкоштовна	Безкоштовна	Безкоштовна
Підтримувані ОС	Linux	Linux, Windows	Linux	Linux (Debian/Ubuntu)	Будь-яка ОС з Docker
Основне призначення	Комерційний веб-хостинг	Веб-хостинг і сервер	Системне адміністрування	Безкоштовний веб-хостинг	Управління контейнерами
Інтерфейс	Графічний, дещо застарілий	Сучасний, графічний	Мінімалістичний, функціональний	Класичний, графічний	Сучасний, орієнтований на Docker
Ресурсоємність	Висока	Середня/Висока	Дуже низька	Середня	Низька
Управління Docker	Відсутнє (або через плагіни)	Інтегровано	Є модуль	Немає	Основна функція
Управління ОС	Обмежене (через WHM)	Розширене	Основна функція	Базове	Відсутнє
Цільова аудиторія	Хостинг-провайдери, початківці	IT-фахівці, бізнес, розробники	Системні адміністратори	Ентузіасти, малий бізнес	Розробники, DevOps

Кінець таблиці 1.1

Сильні сторони	Надійніст, екосистема	Мульти-ОС, WordPress Toolkit	Легкість, інтеграція з ОС	Безкоштовність, повний функціонал	Потужне управління Docker
Обмеження	Ціна, високі вимоги	Ціна, складність	Не для веб-хостингу	Вузька підтримка ОС	Не керує хост-системою

З урахуванням проведеного аналізу, стає очевидним, що на ринку існує певна ніша. З одного боку, маємо повнофункціональні, але дорогі та ресурсоємні панелі для веб-хостингу (cPanel, Plesk, HestiaCP). З іншого – мінімалістичні системні інструменти (Cockpit) та вузькоспеціалізовані рішення (Portainer).

Для користувача, якому потрібен простий інструмент для моніторингу системних ресурсів та зручного запуску Docker-контейнерів без зайвого функціоналу для управління сайтами, поштою та DNS, жодне з існуючих рішень не є ідеальним. Необхідно або встановлювати два окремих інструменти (наприклад, Cockpit + Portainer), або миритися з надлишковим функціоналом та ресурсоємністю хостинг-панелей.

Отже, розробка власної полегшеної веб-панелі з мінімально необхідним функціоналом – моніторинг системних ресурсів, запуск та керування контейнерами Docker, перегляд інформації про стан системи – є обґрунтованою як з технічної, так і з економічної точки зору.

1.3 Висновки до першого розділу

У першому розділі проведено аналіз еволюції програмно-технічних засобів для автоматизації адміністрування серверів. Було встановлено, що ключовим трендом став перехід від інтерфейсу командного рядка до веб-інтерфейсів, таких як Webmin, cPanel, Plesk і Cockpit, які забезпечують зручне керування системами.

Також досліджено особливості ключових інструментів на ринку, їх функціональні можливості, технічні вимоги та цінову політику. Проведено розширене порівняння комерційних рішень (cPanel, Plesk) з OpenSource-альтернативами (Cockpit, HestiaCP, Portainer), що дозволило визначити їх переваги та недоліки для різних сценаріїв використання.

Аналіз показав, що на ринку існує незаповнена ніша для інтегрованого рішення, яке б поєднувало простоту системного адміністрування, як у Cockpit, із потужними можливостями управління контейнерами Docker, як у Portainer, уникаючи при цьому надлишковості та високих вимог до ресурсів, властивих повноцінним хостинг-панелям. Це обґрунтовує доцільність створення полегшеної веб-панелі з базовим функціоналом (моніторинг ресурсів, керування контейнерами, відображення стану системи) для спрощення адміністрування та зменшення залежності від платного або надлишкового програмного забезпечення.

На основі проведеного аналізу було сформовано вимоги до власної системи керування віддаленим сервером. У наступному розділі буде проведено додатковий аналіз технологій та архітектурних підходів, які можуть бути використані для її реалізації.

2 ОБҐРУНТУВАННЯ ВИБОРУ КОМПОНЕНТІВ ТА СЕРЕДОВИЩА РЕАЛІЗАЦІЇ

2.1 Апаратне середовище реалізації

Функціонування розроблюваної системи базується на апаратній платформі, вимоги до якої включають забезпечення безперервності виконання обчислювальних процесів та стабільності роботи під час взаємодії з користувачем. У даному контексті, раціональним є вибір віртуалізованого серверного середовища (Virtual Private Server, VPS), що надає можливість гнучкого виділення необхідних системних ресурсів, зокрема обсягу оперативної пам'яті (RAM), продуктивності дискової підсистеми, лімітів навантаження на центральний процесор (CPU) та забезпечення стабільного мережевого з'єднання.

Альтернативно, за наявності невикористовуваного персонального комп'ютера або ноутбука, можливе його застосування як апаратної платформи. Однак, такий підхід потребуватиме значних часових витрат та відповідних технічних компетенцій для належного налаштування серверного середовища. В таблиці 2.1 представлено коротке порівняння між VPS (Virtual Private Server) та домашнім сервером.

Таблиця 2.1 – Порівняння VPS та домашнього сервера.

Критерій	VPS	Домашній сервер
Надійність	Висока (професійні дата-центри)	Залежить від домашньої інфраструктури
Пропускна здатність мережі	Висока (1 Гбіт/с і більше)	Обмежена (залежить від провайдера)

Кінець таблиці 2.1

Статична IP-адреса	Надається	Потрібно додатково замовляти
Вартість	Від $4/міс	Одноразові витрати на обладнання, далі на підтримку
Обслуговування	Включено у вартість	Потребує технічних знань та часу

Для цілей тестування та розгортання демонстраційної версії системи достатньо використання серверного середовища з такими мінімальними технічними характеристиками: 2 vCPU або 2 фізичних CPU, 2 ГБ оперативної пам’яті, а також твердотільного накопичувача (SSD) обсягом не менше 20 ГБ. Такої конфігурації вистачає для запуску базових сервісів, перевірки працездатності інтерфейсів та етапу попереднього відлагодження.

У продуктивному середовищі, де можливе зберігання логів, телеметричних даних або обробка одночасних запитів, рекомендовано збільшити обсяг оперативної пам’яті до 4 ГБ, а дисковий простір – до 40 ГБ. Збільшення ресурсів забезпечує стабільність роботи системи, особливо за умов зростання навантаження.

Критично важливою вимогою до інфраструктури є наявність SSH-доступу до сервера. Протокол SSH (Secure Shell) забезпечує захищений канал для віддаленого керування, що дозволяє адміністратору здійснювати основні дії – запуск/зупинку сервісів, моніторинг системного стану, оновлення програмного забезпечення тощо – з дотриманням вимог інформаційної безпеки.

У якості хостингового середовища може бути використаний як віртуальний сервер (VPS/VDS), так і фізичний сервер, розташований локально, наприклад, звичайний настільний комп’ютер або ноутбук. У разі наявності фінансових можливостей доцільним є придбання серверного обладнання промислового класу,

яке застосовується в дата-центрах (Рисунок 2.1), що забезпечує розширену надійність та тривалість експлуатації.

Рисунок 2.1 – 1U Cisco UCS C220 M4S [27]

Для зберігання та обробки великих обсягів інформації зазвичай використовуються високопродуктивні серверні системи, які забезпечують необхідну обчислювальну потужність, стабільність та масштабованість. Проте для потреб пересічного користувача, який не виконує ресурсомісткі обчислення, достатнім буде використання персонального комп’ютера офісного класу.

У якості прикладу можна розглянути міні-ПК лінійки HP ProDesk Mini, зокрема модель 400 G5, яка поєднує компактні габарити з достатньою продуктивністю для виконання типових задач, пов’язаних з адмініструванням локального або віддаленого сервера (Рисунок 2.2). Такий пристрій є енергоефективним, легко інтегрується в існуючу інфраструктуру та забезпечує належний рівень надійності при мінімальних апаратних витратах.

Рисунок 2.2 – Вигляд HP ProDesk 400 G5 [28]

Представлена апаратна платформа є компактним персональним комп'ютером, укомплектованим 6-ядерним, 12-потоковим центральним процесором Intel® Core™ i5-9500T, 32 ГБ оперативної пам'яті (RAM) типу DDR4 з тактовою частотою 2666 МГц, твердотільним накопичувачем (SSD) об'ємом 1 ТБ та мережевим адаптером Ethernet із пропускною здатністю до 1 Гбіт/с, що забезпечує високошвидкісне підключення до мережі. Враховуючи співвідношення вартості та продуктивності, дана конфігурація може розглядатися як збалансоване рішення для розгортання домашнього сервера.

У якості операційної системи для хостингу серверної (BackEnd) та клієнтської (FrontEnd) частин системи доцільно використовувати Ubuntu Server у редакції LTS (Long Term Support). Основними аргументами на користь такого вибору є стабільність роботи ядра, регулярне отримання оновлень безпеки, широка сумісність з бібліотеками й пакетами, а також наявність офіційної документації та підтримки великої спільноти розробників.

Хоча для розгортання подібних систем можуть бути використані й інші дистрибутиви Linux (наприклад, Debian, CentOS, Arch Linux, Kali Linux тощо), усі вони мають схожу архітектурну основу, однак різняться підходами до конфігурації, менеджерами пакетів та складністю первинного налаштування. Частина з них, зокрема Arch Linux, потребує глибшого розуміння структури операційної системи та роботи в середовищі командного рядка, оскільки встановлення здійснюється повністю вручну, без графічного інтерфейсу.

З огляду на вищезазначене, для користувачів з базовим або середнім рівнем технічної підготовки найбільш оптимальним варіантом є Ubuntu Server, оскільки дистрибутив містить інтуїтивно зрозумілий графічний інсталятор, автоматично визначає та інсталює драйвери, а також забезпечує налаштування основних залежностей для стабільної роботи системи (Рисунок 2.3, Рисунок 2.4).

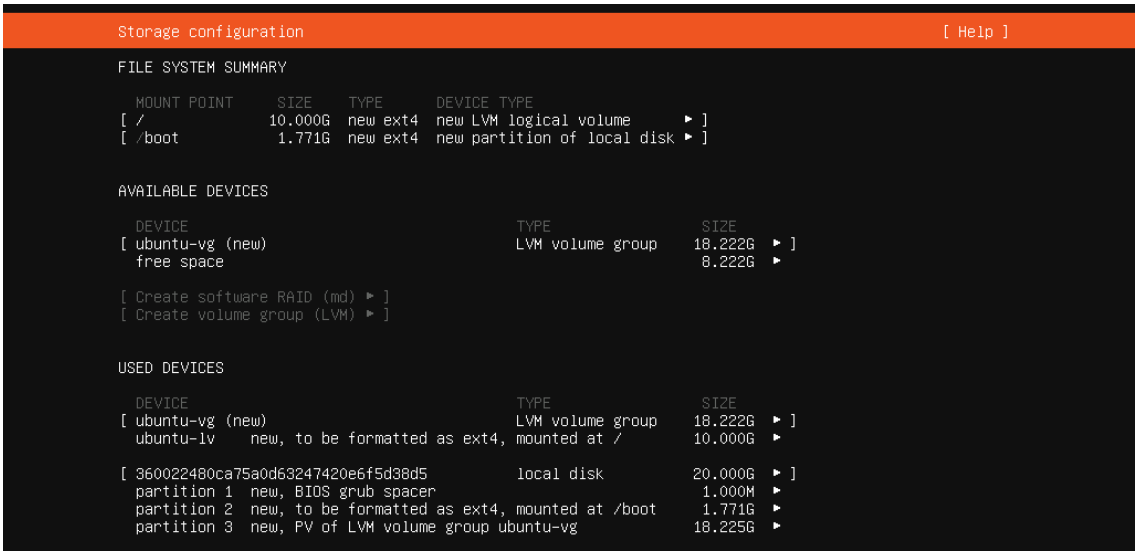


Рисунок 2.3 – Вигляд GUI Ubuntu Server Installer для конфігурації сховища

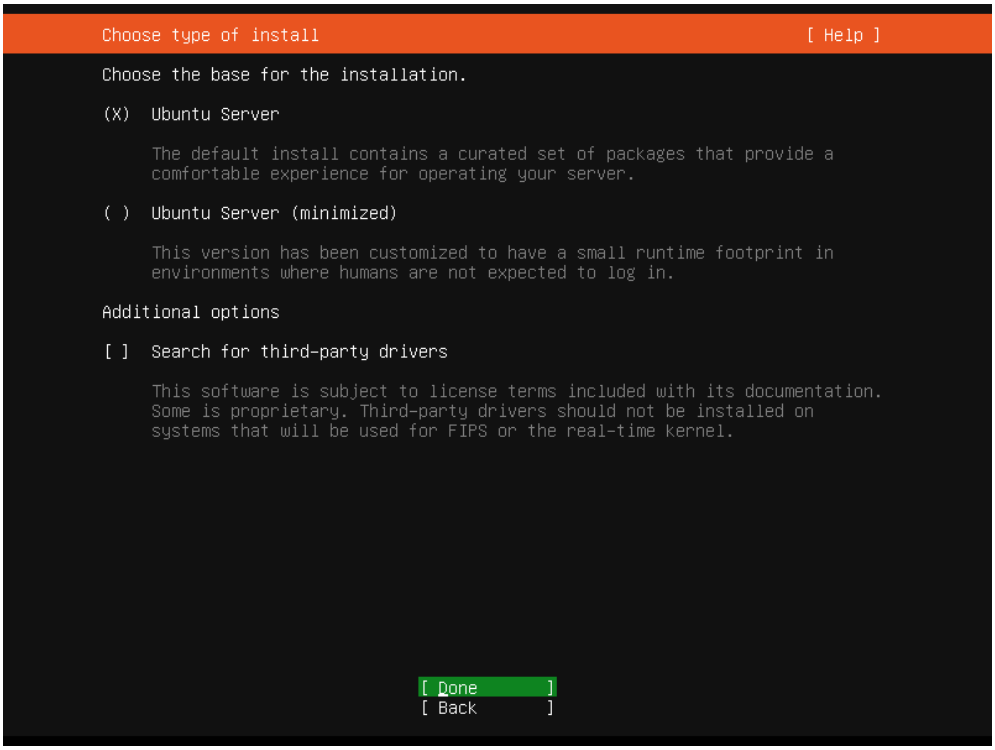


Рисунок 2.4 – Вигляд GUI Ubuntu Server Installer для вибору типа установки

Крім того, версія Ubuntu Server 24.04 LTS має гарантовану офіційну підтримку до 2027 року, що надає змогу уникнути критичних змін та ризиків несумісності під час оновлення системного середовища протягом щонайменше трьох років.

Якщо система розгортається в межах внутрішньої мережі організації, можливо застосування вже налаштованого локального сервера з Hyper-V або Proxmox, що забезпечує запуск віртуалізованих середовищ без зовнішнього підключення. Але для публічного сервісу з доступом через веб-інтерфейс оптимальнішим є варіант хостингу на платформах на зразок Hetzner, DigitalOcean чи AWS.

2.2 Функційні вимоги

Функціональні вимоги визначають набір конкретних дій, які система повинна виконувати для забезпечення належного користувацького досвіду та досягнення поставлених завдань. Вони описують функціональність програмного забезпечення з точки зору як кінцевого користувача, так і розробника. Іншими словами, функціональні вимоги окреслюють очікувану поведінку системи у відповідь на певні вхідні дані або дії користувача. Чітке визначення цих вимог забезпечує не тільки належний рівень взаємодії користувача з програмним продуктом, але й слугує основою для проектування архітектури, розробки логіки роботи програмного інтерфейсу додатків (API) та валідації коректності реалізації на етапі тестування.

В межах розробки системи керування віддаленим сервером функційні вимоги охоплюють операції, пов'язані з моніторингом стану системи, управлінням службами та контейнерами, а також організацією відповідного користувацького інтерфейсу. Система передбачає один тип користувача, що спрощує логіку авторизації та дозволяє сконцентрувати зусилля на реалізації інтерфейсної функціональності без необхідності впровадження складної рольової моделі.

Доступ до системи здійснюється через веб-інтерфейс, після чого користувач отримує доступ до ключових інформаційно-керуючих блоків.

Центральним функціональним компонентом є модуль моніторингу, призначений для відображення таких системних показників, як рівень завантаження центрального процесора (CPU), обсяг використаної оперативної пам'яті (RAM), загальний та доступний дисковий простір, а також інтенсивність вхідного/вихідного мережевого трафіку. Зазначені показники повинні візуалізуватися у графічному форматі з періодичним оновленням даних. Це надає користувачеві можливість в режимі, наближеному до реального часу, відстежувати навантаження на апаратні ресурси серверної системи, що є критично важливим для оперативного реагування при виявленні аномалій або збоїв у функціонуванні.

Окремий функціональний блок забезпечує взаємодію із системою віртуалізації на рівні операційної системи Docker [29-32]. Користувацький інтерфейс повинен надавати перелік активних контейнерів, доступ до деталізованої інформації по кожному з них (включаючи назву, поточний статус, споживані ресурси, команду запуску), а також інструменти для виконання базових операцій: запуск, зупинка, видалення та перегляд журналів (логів). Така інтеграція дозволяє здійснювати не тільки моніторинг активних сервісів, а й управління повним життєвим циклом кожного контейнера через єдиний графічний інтерфейс, без необхідності використання командного рядка.

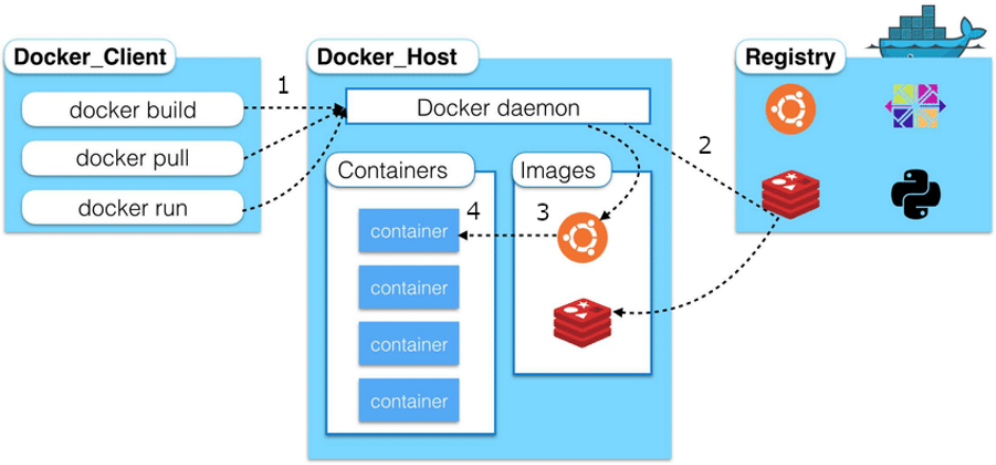


Рисунок 2.5 – Принцип роботи Docker [32]

Розглядається можливість реалізації базового механізму внутрішнього журналювання дій, наприклад, фіксації операцій запуску/зупинки контейнерів або значущих змін, ініційованих через користувацький інтерфейс. Хоча на початковому етапі розробки дана функціональність не є пріоритетною, її подальша імплементація дозволяє відстежувати історію системних змін, реалізувати систему сповіщень або проводити аудит дій користувача.

Комунікація між клієнтською частиною (frontend) та серверною логікою (backend) реалізована за допомогою REST API [33], який забезпечує обмін даними між компонентами системи. У поточній версії API призначений виключно для внутрішнього використання, тобто веб-інтерфейс та серверні компоненти функціонують в межах одного хоста без зовнішніх мережевих запитів. Проте, завдяки застосованому модульному підходу, існує можливість у майбутньому надати авторизований (наприклад, за допомогою токенів) доступ до визначеного підмножини функцій API для зовнішніх користувачів або сторонніх сервісів. Це може бути застосовано, наприклад, для розробки мобільного клієнта, що відображатиме критичні показники системи, або для інтеграції з іншими платформами моніторингу (наприклад, Prometheus, Zabbix).

Функційні вимоги до системи охоплюють моніторинг системних ресурсів, базове адміністрування контейнерів, забезпечення інтерактивної панелі керування для користувача та внутрішню комунікацію через API без залучення сторонніх протоколів передачі даних. Усі зазначені функції мають бути реалізовані таким чином, щоб взаємодія користувача із системою характеризувалася максимальним рівнем зручності, швидкодії та інтуїтивної зрозумілості.

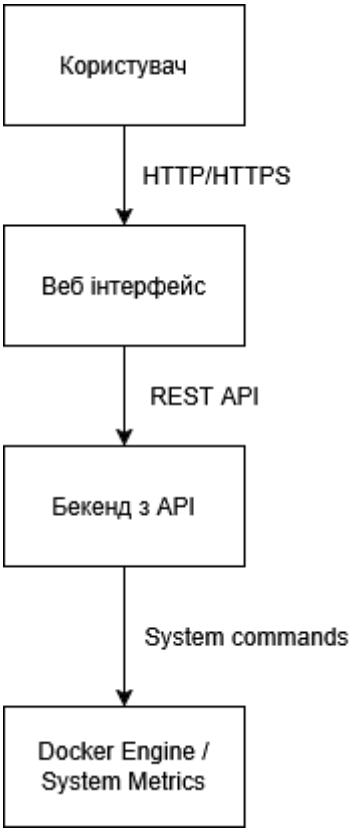


Рисунок 2.6 – Логічна модель взаємодії компонентів

В таблиці 2.2 коротко представлено функціональні вимоги до системи керування сервером з веб-орієнтованим інтерфейсом.

Таблиця 2.2 – Функціональні вимоги до системи

Категорія	Вимога
Моніторинг системних ресурсів	Відображення завантаження CPU, використання RAM, дискового простору (загальний/доступний), мережевого трафіку; графічне представлення з періодичним оновленням даних.
Управління контейнерами Docker	Виведення списку контейнерів та їх детальної інформації (назва, статус, ресурси, команда запуску); виконання операцій: запуск, зупинка, видалення; перегляд логів.

Кінець таблиці 2.2

Інтерфейс користувача та доступ	Авторизований доступ через веб-інтерфейс (підтримка одного типу користувача); надання доступу до ключових інформаційно-керуючих блоків системи.
API та внутрішня взаємодія	Внутрішній REST API для комунікації між клієнтською та серверною частинами; архітектурна можливість для майбутнього журналювання дій та розширення API для зовнішнього доступу.

2.3 Нефункційні вимоги

Нефункційні вимоги являють собою набір атрибутів, що характеризують якісні аспекти функціонування системи, на відміну від функційних вимог, які описують її поведінку та можливості. Нефункційні вимоги охоплюють такі параметри, як продуктивність, безпека, сумісність, доступність, масштабованість та надійність. Їх належна реалізація суттєво впливає на стабільність роботи програмного забезпечення, його довгострокову супроводжуваність, якість користувацької взаємодії та загальну стійкість до експлуатаційних навантажень і нештатних ситуацій. Дотримання нефункційних вимог є ключовим фактором для формування довіри користувачів, оптимізації витрат на підтримку системи та підвищення її технічної досконалості.

При розробці системи керування віддаленим сервером особлива увага була зосереджена на забезпеченні високої продуктивності. Цільовим показником є середній час відповіді програмного інтерфейсу (API), який не повинен перевищувати 500 мс, навіть за умов пікових навантажень. Досягнення зазначеної швидкодії забезпечується шляхом застосування асинхронного підходу до програмування, що уможливлює паралельну обробку множинних запитів без блокування основного потоку виконання. Це має критичне значення при моніторингу значних обсягів метрик, що генеруються в режимі реального часу.

Застосоване рішення також сприяє забезпеченню плавності взаємодії користувацького інтерфейсу з системою під час виконання нею ресурсомістких внутрішніх операцій.

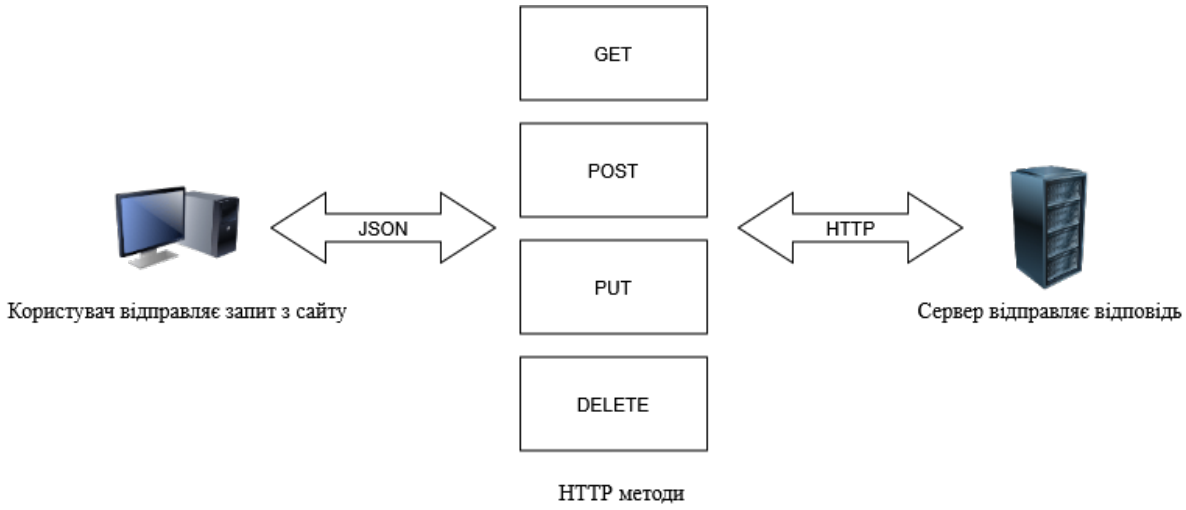


Рисунок 2.7 – Схема клієнт-серверної взаємодії на основі REST архітектури, що ілюструє використання стандартних HTTP-методів (GET, POST, PUT, DELETE) та формату JSON для обміну даними.

Надійність системи визначається як її спроможність до стабільного функціонування протягом тривалого періоду без збоїв та втрати критично важливих даних. У цьому аспекті значну роль відіграє використання технології контейнеризації (Docker), яка створює ізольовані середовища для кожного системного компонента та дозволяє оперативно відновлювати їх працездатність. Додатково імплементовано механізм автоматичного перезапуску сервісів у випадку виникнення критичних помилок, що мінімізує час простою системи. Перспективним напрямком є розробка додаткової системи сповіщень адміністраторів про зафіксовані неполадки.

У контексті безпеки було інтегровано сучасні та широко підтримувані технологічні рішення. Передача даних між клієнтською та серверною частинами здійснюється за допомогою захищеного протоколу HTTPS, що запобігає перехопленню або модифікації інформації. Аутентифікація користувачів базується

на механізмі JWT (JSON Web Tokens), який забезпечує безпечне управління токенами доступу та контроль сесій. Незважаючи на те, що на поточному етапі розробки система підтримує лише один тип користувача, її архітектура спроектована з урахуванням можливості подальшої інтеграції розширених механізмів розмежування прав доступу, таких як ролі, дозволи та групи користувачів. [34-35]

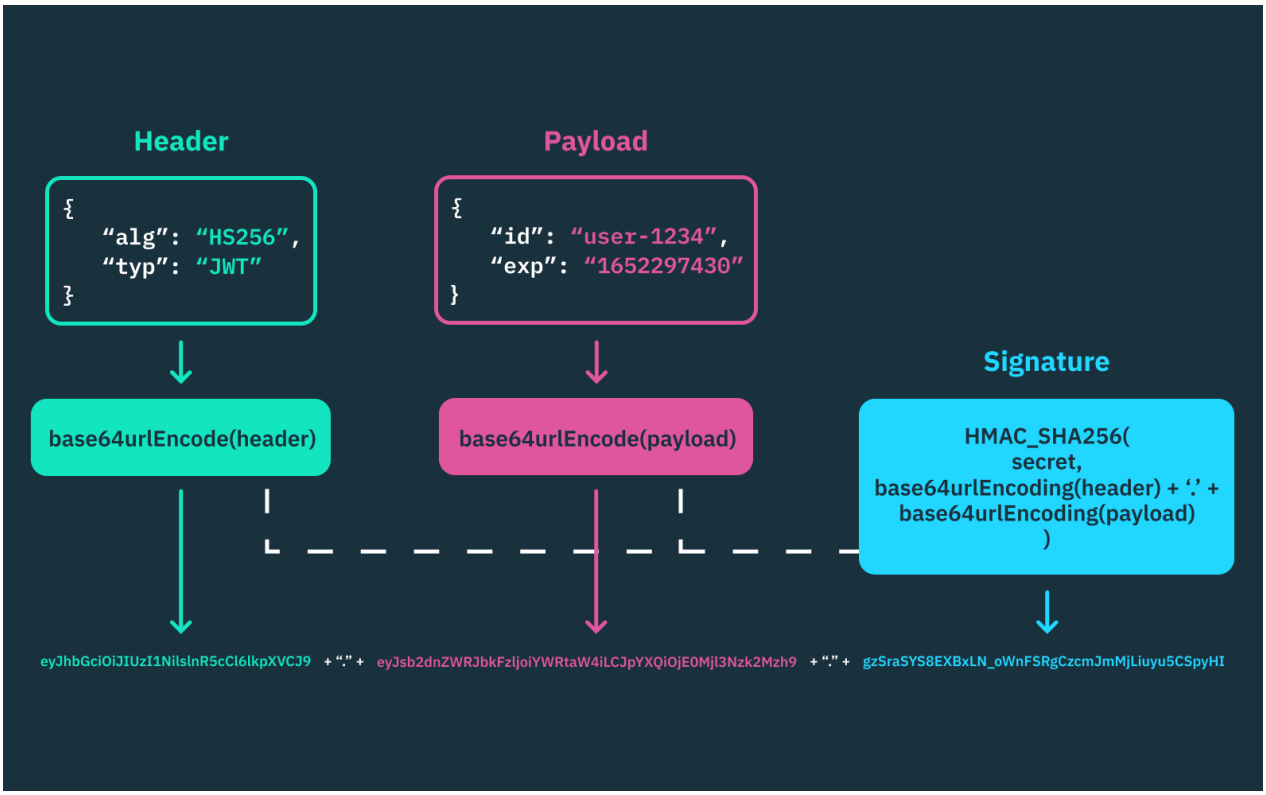


Рисунок 2.8 – Структура JSON Web Token (JWT) та процес його формування [35]

Стосовно масштабованості, хоча система первинно орієнтована на експлуатацію в межах одного фізичного або віртуального сервера, вона спроектована для ефективної обробки значних навантажень, що виникають при одночасній активності багатьох користувачів. З цією метою архітектурно передбачена можливість горизонтального масштабування окремих компонентів, зокрема, винесення бази даних на виділений серверний ресурс або створення окремого рівня для зберігання журналів подій (логів). Така архітектурна гнучкість

забезпечить адаптивність системи до зростання обсягів оброблюваної інформації або розширення її функціональних можливостей у майбутньому.

Система повинна забезпечувати коректне та стабільне функціонування в поширених веб-браузерах, таких як Google Chrome, Mozilla Firefox та Microsoft Edge. Крім того, враховано вимоги щодо адаптації інтерфейсу для мобільних пристроїв. Хоча на поточному етапі мобільна адаптація реалізована частково, подальші ітерації розробки передбачають покращення доступності та зручності використання інтерфейсу на смартфонах і планшетах. Це розширює можливості адміністратора, дозволяючи здійснювати моніторинг та управління сервером не лише зі стаціонарних робочих місць.

На момент підготовки даної кваліфікаційної роботи функціональні можливості офлайн-доступу та системи резервного копіювання даних не були імплементовані. Проте, впровадження зазначених функцій розглядається як один із перспективних напрямів подальшого розвитку програмного продукту. Зокрема, вважається доцільним проектування механізмів для локального кешування критично важливої інформації або реалізація процедур періодичного експорту конфігураційних даних з метою мінімізації ризиків, пов'язаних із потенційними апаратними відмовами серверної інфраструктури.

В таблиці 2.3 коротко представлено нефункціональні вимоги до системи керування сервером з веб-орієнтованим інтерфейсом.

Таблиця 2.3 – Нефункціональні вимоги до системи

Категорія	Вимога
Продуктивність	Затримка API < 500 мс, асинхронна обробка запитів
Надійність	Підтримка Docker restart policy, обробка збоїв без втрати даних
Безпека	JWT-авторизація, HTTPS, контроль доступу по ролях

Кінець таблиці 2.3

Масштабованість	Docker Compose, горизонтальне масштабування
Сумісність	Сучасні браузери, мобільні пристрої

2.4 Структурна схема програмно-технічного засобу та вибір методів та середовища для реалізації програмного забезпечення

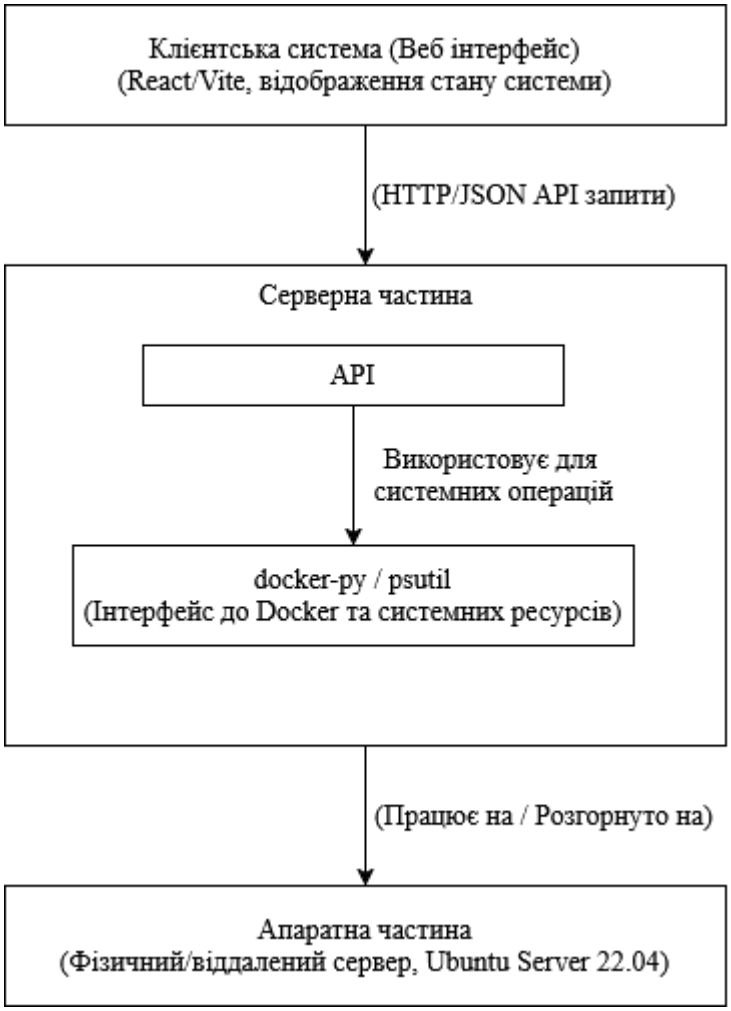


Рисунок 2.6 – Схема програмно-технічного засобу для системи керування віддаленим сервером з веб-орієнтованим інтерфейсом

Програмно-технічний комплекс складається з трьох основних компонентів: апаратного забезпечення, серверної підсистеми та клієнтської підсистеми.

Апаратна частина реалізується у вигляді фізичного або віддаленого сервера, який забезпечує моніторинг системних параметрів та передачу даних на веб-інтерфейс для відображення користувачу.

Розробка системи управління віддаленим сервером базується на чіткому визначенні архітектури, принципів взаємодії компонентів та виборі оптимального програмного середовища, що забезпечує швидку розробку, масштабованість і стабільність роботи. Система структурно поділена на три основні блоки: клієнтську частину, серверний API та рівень операційної системи, з яким взаємодіє серверна частина. Комунікація між компонентами здійснюється через HTTP-запити у внутрішній мережі без використання зовнішніх протоколів.

На етапі проектування архітектури клієнтська частина реалізована як односторінковий веб-застосунок (SPA) з використанням бібліотеки React та інструменту Vite. Vite забезпечує швидку збірку модулів і підтримку гарячого оновлення під час розробки, а React сприяє створенню гнучкої компонентної архітектури з підтримкою сучасного JavaScript і TypeScript. Для стилізації інтерфейсу застосовано Tailwind CSS, що забезпечує швидке створення адаптивних компонентів, оптимізованих для десктопних і мобільних пристроїв. Інтерфейс структуровано у вигляді модульних блоків для відображення метрик, керування Docker-контейнерами, перегляду журналів і моніторингу стану системи [36-38].

Серверна частина розроблена з використанням фреймворку FastAPI, мови Python 3.12 та менеджера залежностей Poetry, який забезпечує ізольовані середовища та сумісність пакетів через конфігурацію у файлі pyproject.toml. Розробка велася в інтегрованому середовищі Cursor IDE, побудованому на основі Visual Studio Code, яке підтримує автодоповнення коду на основі штучного інтелекту, інтеграцію з Git і зручну навігацію. Cursor IDE є ефективним інструментом для розробки на Python, придатним для користувачів різного рівня підготовки [39-41].

Для реалізації серверної логіки використано FastAPI. Це асинхронний фреймворк на базі ASGI, що забезпечує підтримку OpenAPI/Swagger для

автоматичної генерації документації, валідацію даних і асинхронну обробку запитів. Запуск FastAPI здійснюється через сервер Uvicorn, що забезпечує високу продуктивність і можливість масштабування API без зміни архітектури. [42-44]

Взаємодія з Docker API реалізована через бібліотеку docker-py, яка забезпечує програмне керування контейнерами: отримання списку, запуск, зупинку та видалення. Моніторинг системних ресурсів здійснюється за допомогою модуля psutil, який збирає дані про завантаження процесора, використання оперативної пам’яті, мережеві та дискові ресурси [45-47].

Моніторинг системних ресурсів реалізовано з використанням бібліотеки psutil, яка забезпечує ефективне збирання даних про продуктивність системи. Ця бібліотека дозволяє отримувати інформацію про завантаження центрального процесора, використання оперативної пам’яті, дискові операції та мережеву активність у реальному часі. Завдяки кросплатформенній підтримці та низькому споживанню ресурсів, psutil забезпечує надійне відстеження стану системи, що є критично важливим для динамічного управління контейнерами та оптимізації роботи серверної інфраструктури [48-50].

Для зберігання даних використовується реляційна база даних PostgreSQL, розгорнута як окремий контейнер у складі docker-compose. База даних зберігає інформацію про авторизованих користувачів, забезпечуючи структуроване управління даними та можливість подальшого масштабування для аналітики чи історичного моніторингу [51-53].

Архітектура системи спроектована з чітким розподілом зон відповідальності компонентів, що забезпечує розширюваність і незалежність від середовища виконання (локального чи хмарного).

В таблиці 2.4 представлено порівняльну характеристику використаних інструментів розробки.

Таблиця 2.4 – Порівняльна характеристика використаних інструментів розробки

Компонент	Переваги	Недоліки
Cursor IDE	AI-підказки, інтеграція з Git, висока швидкодія, підтримка розширень	Обмежена кастомізація, нестабільність деяких плагінів, відсутність розширеного дебагу
FastAPI	Асинхронна обробка, OpenAPI-документація, Pydantic-інтеграція	Потреба в строгій типізації, ускладнення при stateful-логіці, потреба у додаткових абстракціях
React + Vite	Швидке HMR, компонентна структура, підтримка TypeScript, велика екосистема	Складне керування станом, клієнтське рендеринг (SEO), можливе навантаження на DOM
PostgreSQL	Потужність запитів, ACID-транзакції, розширюваність, стабільність	Більш складне налаштування, ресурсоємність, потреба в окремому сервісі

2.5 Висновки до другого розділу

У ході роботи, а саме у другому розділі, було розглянуто усі позитивні та негативні сторони вибраних компонентів та середовищ для реалізації. Отже для створення серверної частини необхідно використати – FastAPI, Docker, psutil, docker-py, Poetry.

Для реалізації клієнтської частини нам потрібні наступні технології – React, Vite, EsLint. Для реалізації апаратної частини ми можемо використати міні-ПК лінійки HP ProDesk Mini модель 400 G5. Також стануть в нагоді наступні програмні забезпечення – Cursor IDE, Docker Desktop.

3 ПРОГРАМНО-ТЕХНІЧНИЙ ЗАСІБ ДЛЯ СИСТЕМИ КЕРУВАННЯ ВІДДАЛЕНИМ СЕРВЕРОМ З ВЕБ-ОРІЄНТОВАНИМ ІНТЕРФЕЙСОМ

3.1 Цільова аудиторія та принцип роботи програмно-технічного засобу для керування віддаленим сервером з веб-орієнтованим інтерфейсом.

Розроблена система призначена для забезпечення віддаленого адміністрування сервера за допомогою веб-інтерфейсу. Ключовою метою є надання інструментарію для моніторингу та управління серверними ресурсами, що уможливлює виконання основних адміністративних операцій без вимоги до глибоких знань користувача у сфері інформаційних технологій.

На відміну від існуючих рішень, що базуються на ручному конфігуруванні серверів через термінальний доступ, інтерфейс командного рядка або стандартні панелі управління, представлена система використовує графічні інструменти для спрощення процесів взаємодії та прийняття рішень. Користувач отримує можливість оперативно оцінити ступінь завантаженості центрального процесора, визначити активні контейнери, визначити обсяг доступної оперативної пам'яті та швидко перезапустити необхідні процеси.

Цільовою аудиторією системи є користувачі, які мають намір експлуатувати власний сервер (наприклад, для розміщення веб-сайтів, програмних додатків або мультимедійних сервісів), проте не володіють достатнім досвідом роботи з інструментами командного рядка (CLI), протоколом SSH або адмініструванням Unix-подібних операційних систем. Водночас, наявний функціонал системи дозволяє її ефективне застосування технічними фахівцями для виконання рутинних завдань адміністрування.

Таблиця 3.1 – Основні функціональні можливості системи

Категорія функціоналу	Опис можливості
Моніторинг ресурсів	Контроль рівня навантаження на центральний процесор (CPU) та оперативну пам'ять (RAM).
Управління сховищем	Відображення інформації про підключені дискові накопичувачі, їх поточний стан та доступний дисковий простір.
Адміністрування контейнерів	Управління Docker-контейнерами, що включає перегляд стану, ініціалізацію запуску, зупинку, видалення та доступ до файлів журналів.
Система безпеки	Реалізація системи авторизації користувачів на основі базової моделі безпеки для забезпечення контрольованого доступу до системи.
Візуалізація даних	Представлення всіх оперативних даних у графічному форматі з використанням інтерактивних елементів управління.

Взаємодія з серверною частиною системи здійснюється через REST API [33], розроблене з використанням фреймворку FastAPI. Такий підхід забезпечує високу продуктивність, масштабованість та відповідність сучасним стандартам розробки програмних інтерфейсів додатків. Застосування FastAPI також дозволяє формалізувати структуру запитів та відповідей, що позитивно впливає на подальший супровід та розвиток програмного продукту.

Клієнтська частина інтерфейсу користувача розроблена з використанням бібліотеки React та інструментарію Vite. Це забезпечує швидке оновлення відображуваного контенту, мінімальний час завантаження сторінок, а також

створює передумови для подальшої адаптації інтерфейсу для мобільних пристроїв. Завдяки застосуванню компонентного підходу та сучасних методів організації архітектури користувацького інтерфейсу (UI), клієнтська частина проекту характеризується масштабованістю та зручністю для подальшої розробки.

Для реалізації запропонованої в роботі системи керування віддаленим сервером за допомогою веб-інтерфейсу представлено алгоритм роботи цієї системи, який складається з восьми кроків:

1) Запуск інтерфейсу.

Користувач у браузері відкриває головну сторінку системи. Завантажується клієнтська частина, розгорнута за допомогою Vite, ініціалізується React-додаток, а також конфігуруються базові запити до API.

2) Аутентифікація.

Під час першого запуску система не має жодного створеного користувача, тому пропонується форма реєстрації адміністратора. Після цього надалі використовується механізм логіну з перевіркою логіну та пароля, з отриманням JWT-токена для захищеного доступу до API.

3) Запит даних.

Після успішної авторизації система надсилає низку асинхронних HTTP-запитів до бекенду. Зокрема, йдеться про запити на отримання інформації про поточне використання CPU, оперативної пам’яті, список дисків, обсяг доступного місця, а також інформацію про активні Docker-контейнери.

4) Обробка на сервері.

Обробка запитів на сервері здійснюється за допомогою фреймворку FastAPI, який викликає відповідні кінцеві точки (Endpoint), реалізовані в модулях system_monitor.py, docker_service.py та інших. Для взаємодії з системними API сервера застосовуються бібліотеки psutil і docker-py. Усі запити забезпечені обробниками винятків, що гарантує стабільність і надійність відповідей сервера.

5) Передача відповіді.

Отримані дані (наприклад, відсоток завантаження CPU або список контейнерів) формуються у вигляді структурованого JSON-об’єкта і повертаються клієнту з відповідними HTTP-статусами. FastAPI автоматично додає до відповіді валідаційні схеми, що запобігає неконсистентності структури даних.

6) Відображення в інтерфейсі.

React отримує відповіді від API, оновлює локальний стан застосунку (через useState/useEffect або зовнішні менеджери стану) та рендерить відповідні компоненти. Наприклад, оновлюється компонент SystemStatus, який виводить кругові графіки завантаження або таблиці процесів у відповідних модулях.

7) Інтерактивні дії.

Якщо користувач бажає взаємодіяти з системою (наприклад, запустити контейнер), він натискає відповідну кнопку в UI. Формується та надсилається запит POST або DELETE до API з передачею відповідного параметра (наприклад, ID контейнера). API обробляє дію, виконує команду через docker SDK, і надсилає підтвердження назад.

8) Оновлення стану.

Після кожної дії інтерфейс ініціює повторний запит для оновлення списку процесів, контейнерів або ресурсів. Це дозволяє завжди відображати актуальні дані без необхідності оновлення всієї сторінки вручну. У разі помилки користувач бачить відповідне повідомлення.

На рисунку 3.1 представлено блок-схему алгоритму роботи системи, описаного раніше.

Рисунок 3.1 – Блок-схема алгоритму роботи системи керування віддаленим сервером з веб-орієнтованим інтерфейсом

Ця послідовність дій забезпечує злагоджену, безперебійну та прозору роботу всієї системи незалежно від рівня підготовки користувача.

В кінцевому результаті, реалізація системи дозволяє значно зменшити технічний бар’єр між користувачем та інфраструктурою сервера, що, у свою чергу, сприяє поширенню приватного серверного хостингу серед широкого загалу користувачів та підвищує рівень цифрової незалежності.

3.2 Архітектура та структура системи

Архітектура розробленої системи базується на принципах модульності, розподілу відповідальностей і масштабованості. Кожен компонент виконує чітко визначену функцію, що забезпечує логічну ізоляцію підсистем, спрощує їх

реалізацію та подальше розширення. Модульна структура сприяє локалізації помилок, внесенню змін без порушення цілісності системи та додаванню нових функціональних можливостей без модифікації наявної логіки. У додатках Г та Д представлено структуру папок і файлів, організованих відповідно до модульного принципу.

Основною метою при розробці було створення гнучкої архітектури, яка забезпечує адаптивність програмного комплексу до різних сценаріїв використання. У сучасних умовах важливо не лише розробити стабільну систему, а й забезпечити її здатність до масштабування та інтеграції в складніші програмні екосистеми. Для цього архітектура системи побудована на основі чітко визначених програмних інтерфейсів, що дозволяє використовувати її як автономний модуль у складі комплексних інфраструктурних рішень.

Система складається з трьох основних рівнів:

1) Клієнтський рівень (Frontend) забезпечує взаємодію з користувачем і відображення даних, отриманих від серверної частини. Реалізовано з використанням сучасного технологічного стеку: бібліотеки React і інструменту Vite. Модульна структура клієнтської частини забезпечує адаптивність інтерфейсу та його розширюваність. Компоненти інтерфейсу є повторно використовуваними, що дозволяє їх застосовувати в інших проєктах або масштабувати відповідно до нових вимог.

2) Серверний рівень (Backend) відповідає за обробку запитів, взаємодію з операційною системою та базами даних. Реалізовано на основі фреймворку FastAPI, який забезпечує створення продуктивних і типобезпечних REST-сервісів. Використання ORM-моделей спрощує роботу з базою даних, забезпечуючи ефективне управління даними користувачів і параметрами стану системи.

3) Системний рівень взаємодіє з інструментами операційної системи та Docker-середовищем. Застосування бібліотек psutil і docker-py забезпечує ефективний доступ до даних про системні ресурси (процесор, оперативна пам’ять,

дискові накопичувачі) та керування контейнерами, що дозволяє гнучко моніторити та адмініструвати контейнеризовані застосунки.

Архітектуру проєкту, описану раніше, представлено на рисунку 3.2.

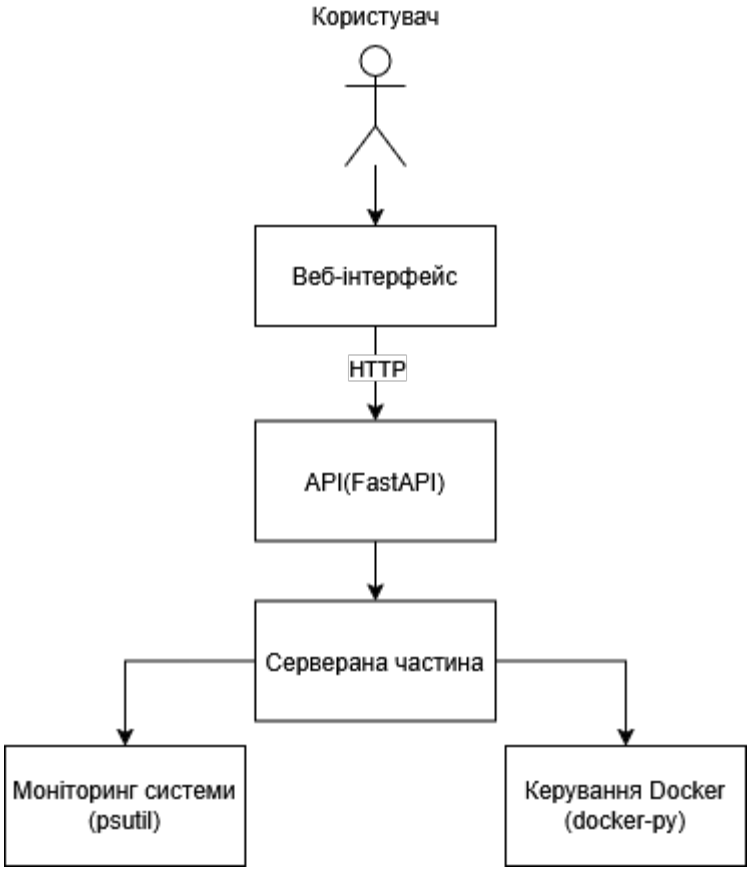


Рисунок 3.2 – Архітектура проекту

Поточна структура проектів демонструє високий рівень модульності, що забезпечує гнучкість при впровадженні нового функціоналу та проведенні рефакторингу існуючих компонентів. Це досягається шляхом інкапсуляції кожного логічного елемента в окремі директорії та файли. Зазначений підхід відповідає принципам об'єктно-орієнтованого програмування (ООП) та SOLID-принципам, що сприяє підвищенню керованості, повторному використанню коду та зменшенню залежностей між модулями.

3.3 Реалізація серверної частини

Серверна частина програмного забезпечення розроблена з використанням фреймворку FastAPI, який є сучасним інструментом для створення високопродуктивних і надійних веб-сервісів на мові програмування Python. Основною перевагою FastAPI є висока швидкість обробки запитів, а також підтримка повної типізації даних завдяки бібліотеці Pydantic, що сприяє підвищенню безпеки та зрозумілості коду.

Архітектура проекту базується на модульному підході. Основна директорія app/ містить усі ключові компоненти, необхідні для реалізації серверної логіки. Кожен підкаталог і файл у цій структурі відповідає за виконання окремої функціональної задачі.

Розглянемо детальніше структуру директорії app/. Файл main.py слугує точкою входу до застосунку. У цьому файлі ініціалізується екземпляр FastAPI та підключаються маршрути (роути) для обробки запитів. Запуск сервера здійснюється саме через цей файл. На початку файлу main.py виконується імпорт необхідних бібліотек і модулів, які забезпечують функціонування сервера. Також визначено асинхронну функцію lifespan, яка описує життєвий цикл програми. Код, розміщений до конструкції yield, виконується перед запуском сервера, дозволяючи, наприклад, встановити з’єднання з базою даних або зовнішніми сервісами. Код після yield виконується під час завершення роботи програми, забезпечуючи, наприклад, коректне закриття з’єднань із базою даних чи зовнішніми сервісами.

Налаштування серверної частини зберігаються у файлі settings.py. У цьому файлі визначено клас Settings, який успадковується від класу BaseSettings бібліотеки pydantic_settings. Такий підхід забезпечує зручне створення та автоматичну валідацію налаштувань проєкту на основі типізації, а також підтримує зчитування змінних із віртуального середовища. Структура файлу settings.py дозволяє легко розширювати конфігурацію в разі потреби, забезпечуючи гнучкість для подальшого розвитку проєкту.

У даному проекті застосовується технологія ORM (Object-Relational Mapping), принцип роботи якої зображено на рисунку 3.3, яка забезпечує спрощену взаємодію з базою даних через визначення класів у мові програмування. Використовується бібліотека Tortoise ORM для Python, відома своєю високою продуктивністю та популярністю. За результатами тестування продуктивності, Tortoise ORM перевищує бібліотеки PonyORM і SQLAlchemy за швидкодією, а також характеризується простотою конфігурації та використання.

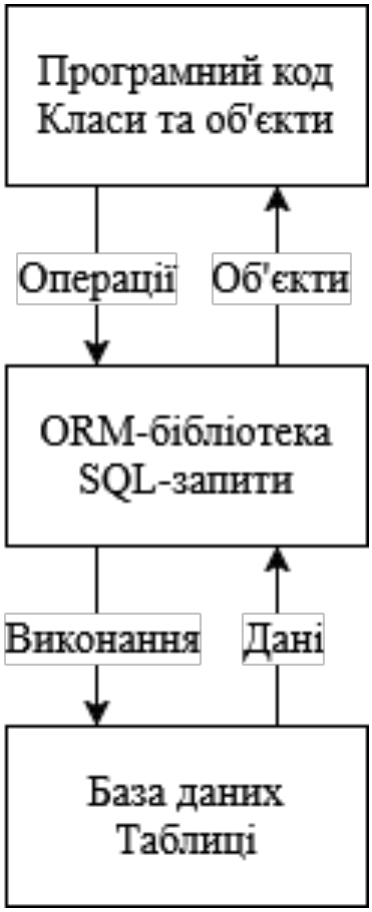


Рисунок 3.3 – Принцип роботи ORM (Object-Relational Mapping)

Для реалізації підключення до бази даних створено файл connections.py, який містить функції для встановлення з’єднань із базою даних та зовнішніми сервісами. Одна з таких функцій, init_external_clients(app), використовується у файлі main.py у межах функції lifespan. Ця функція викликає інші функції, що відповідають за ініціалізацію з’єднань.

Зокрема, у файлі connections.py реалізовано функцію init_tortoise, яка приймає екземпляр класу FastAPI. У цій функції за допомогою методу init класу Tortoise встановлюється з’єднання з базою даних. Параметр modules містить інформацію про розташування моделей, які визначають структуру таблиць бази даних.

Для опису таблиць бази даних за допомогою ORM створено папку app/models із двома файлами: base.py та user.py. У файлі base.py визначено базову модель, від якої успадковуються інші моделі. Ця модель включає загальні поля, такі як id, created_at та updated_at. Важливо зазначити, що в класі Meta встановлено атрибут abstract = True, що позначає базову модель як абстрактну, запобігаючи створенню відповідної таблиці в базі даних Tortoise ORM.

У файлі user.py описано моделі для таблиць користувачів і сесій, які забезпечують авторизацію користувачів та надання доступу до функціональних можливостей системи.

На рисунку 3.4 представлено зв’язки між таблицями бази даних, описані раніше.

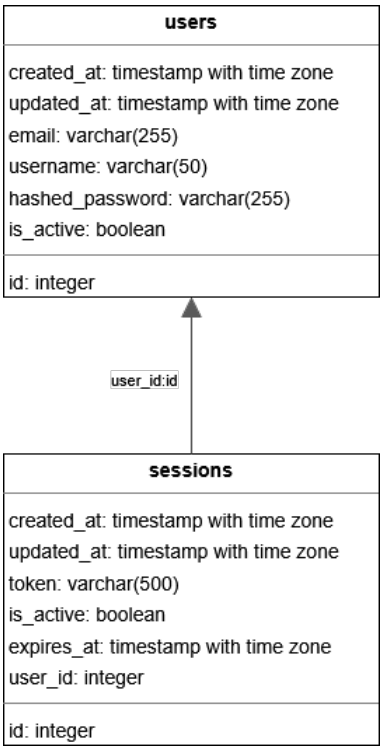


Рисунок 3.4 – Вигляд зв’язків в базі даних.

Класи моделі успадковуються від базового класу BaseModel, що усуває необхідність повторного визначення полів, уже описаних у батьківському класі. У класі Meta можна визначити атрибут table, який вказує назву таблиці в базі даних, що відповідає даній моделі.

На цьому етапі конфігурацію об’єктно-реляційного відображення (ORM) та підключення до бази даних завершено. Для отримання даних, наприклад списку всіх користувачів, у будь-якому маршруті (роуті) проекту достатньо використати асинхронний виклик await User.all().

Система авторизації реалізована в модулях, розташованих у директорії app/auth/. Файл router.py містить маршрути (роути) для обробки операцій авторизації, зокрема register, login та logout, кожна з яких відповідає за виконання певної дії: реєстрацію, вхід та вихід користувача з системи.

Маршрут POST /register реалізує реєстрацію нового користувача, приймаючи об’єкт UserCreate із полями email, username і password. Перед створенням перевіряється унікальність електронної пошти та імені користувача через метод exists() моделі User. Якщо користувач уже існує, генерується виняток HTTPException із кодом 400 і повідомленням про дублювання. У разі успіху пароль хешується функцією get_password_hash, а новий запис створюється в базі через асинхронний метод create(), повертаючи об’єкт UserResponse із кодом стану 201.

Маршрут POST /login забезпечує автентифікацію користувача, приймаючи об’єкт UserLogin із email і password. Функція authenticate_user перевіряє відповідність введених даних із записами в базі; при невдачі генерується HTTPException із кодом 401 і заголовком WWW-Authenticate для авторизації через токен. У разі успіху формується токен доступу з використанням create_access_token, де дані включають ідентифікатор користувача (sub), а термін дії визначається через timedelta на основі конфігурації ACCESS_TOKEN_EXPIRE_MINUTES. Створюється сесія в базі з токеном і часом закінчення дії, після чого повертається об’єкт Token.

Маршрут POST /logout завершує сесію користувача, приймаючи токен через HTTPBearer. Метод filter() моделі Session шукає активну сесію за токеном і оновлює її статус на is_active=False, що деактивує доступ. Успішна операція повертає JSON-відповідь із повідомленням про вихід.

Маршрут GET /me повертає інформацію про поточного користувача, використовуючи залежність get_current_user, яка забезпечує авторизацію, і видає об'єкт UserResponse із деталями профілю.

Маршрут GET /init забезпечує отримання інформації про те, чи є поточний запуск системи першим. На основі цієї інформації на клієнтській стороні визначається, чи відображати форму реєстрації.

У файлі utils.py, розташованому в тій самій директорії, реалізовано допоміжні функції для створення сесії та генерації токена авторизації. Оскільки авторизація базується на використанні JSON Web Token (JWT), застосовується бібліотека jose для генерації токенів. Функція create_access_token призначена для створення токена доступу. У цю функцію передаються дані, які кодуються в токен, а також необов'язковий параметр expires_delta, що визначає термін дії токена. Якщо цей параметр не вказано, використовується значення за замовчуванням, визначене в налаштуваннях проекту. Під час кодування токена застосовується алгоритм, указаний у конфігурації проекту, за замовчуванням – HS256.

Для перевірки наявності користувача в системі та статусу його активної сесії розроблено функцію get_current_user. Ця функція приймає обов'язковий параметр credentials, який містить заголовки (Headers), передані разом із тілом запиту до API. Із заголовків витягується JWT-токен, який декодується за допомогою секретного ключа, використаного під час кодування токена. Цей етап є критично важливим, оскільки декодування без верифікації може дозволити зловмиснику, який знає про відсутність перевірки ключа, згенерувати токен із власними даними для несанкціонованого доступу до API. Тому верифікація секретного ключа є обов'язковою. У разі невалідного токена функція повертає HTTP-статус 401, що вказує на некоректні дані авторизації.

Після успішної валідації токена функція перевіряє наявність відповідної сесії в базі даних. Якщо сесія існує, виконання продовжується; якщо ні – повертається HTTP-статус 401. На завершальному етапі функція отримує об’єкт користувача з бази даних. Логіка залишається аналогічною: якщо користувач знайдений, функція повертає його об’єкт; у протилежному випадку повертається HTTP-статус 401. На рисунку 3.5 представлено блок-схему процесу авторизації користувача, описаного раніше.

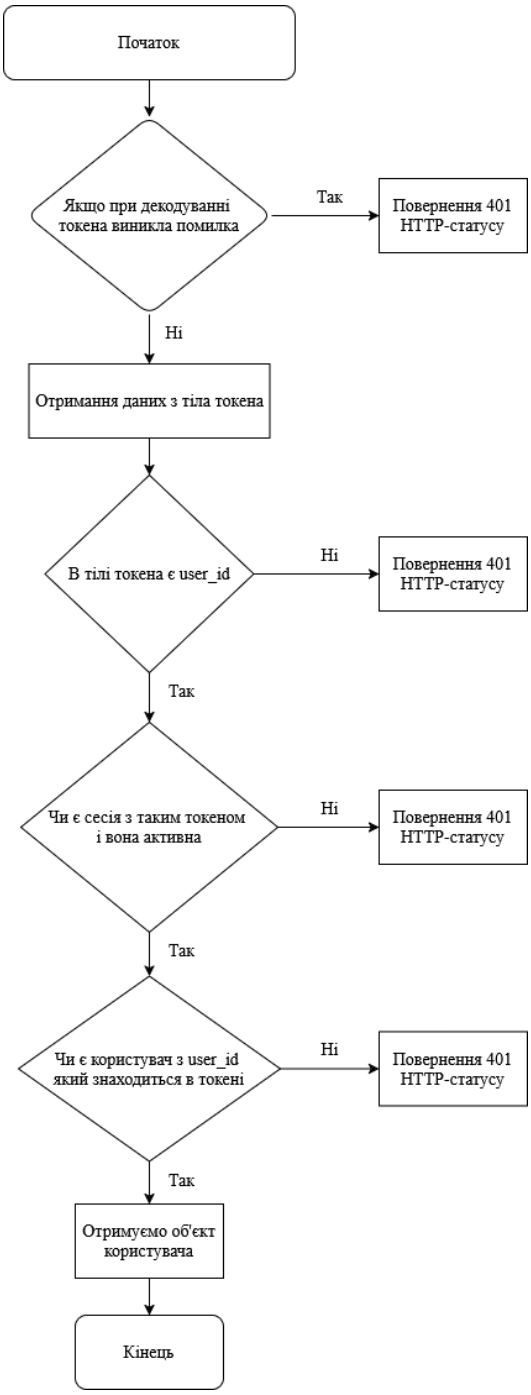


Рисунок 3.5 – Блок-схема авторизації користувача

За допомогою бібліотеки psutil ми можемо отримувати дані про систему, такі як – інформацію про систему, версію системи, процесор, кількість оперативної пам’яті, статус мережі і так далі. Для отримання цієї інформації було створений окремий модуль system який знаходиться по шляху app/system, в цій папці знаходиться файл router.py і папка utils з файлом system_monitor.py.

Файл router.py модуля system визначає маршрути FastAPI для отримання інформації про систему. Об’єкт APIRouter ініціалізується з префіксом /system і тегом System для групування ендпоінтів, пов’язаних із моніторингом системи. Для забезпечення авторизації кожен маршрут використовує залежність get_current_user, яка повертає об’єкт типу User після перевірки токена користувача. Логіка збору даних реалізована в модулі utils.system_monitor, звідки викликаються відповідні функції.

Маршрут GET / викликає функцію system_monitor.get_system_info(), яка повертає повну інформацію про систему у форматі моделі schemas.SystemInfo. Дані включають характеристики платформи, час завантаження системи, відомості про процесор, оперативну пам’ять, дискові розділи та мережеві інтерфейси, забезпечуючи комплексний огляд стану системи.

Маршрут GET /cpu звертається до функції system_monitor.get_cpu_info(), яка повертає детальну інформацію про процесор, зокрема відсоток використання кожного ядра та частоту роботи, у форматі моделі schemas.CpuInfo.

Маршрут GET /memory використовує функцію system_monitor.get_memory_info() для збору даних про оперативну пам’ять і своп-простір. Відповідь містить загальний, доступний і використаний обсяг пам’яті в гігабайтах, а також відсоток використання, і форматується відповідно до моделі schemas.MemoryInfo.

Маршрут GET /disk викликає system_monitor.get_disk_info(), повертаючи список об’єктів типу schemas.DiskPartition, що містять інформацію про всі доступні

дискові розділи, зокрема назву пристрою, точку монтування, тип файлової системи, загальний, використаний і вільний обсяг у гігабайтах, а також відсоток заповнення.

Маршрут GET /network звертається до функції system_monitor.get_network_info(), яка повертає дані про мережеві інтерфейси та статистику вводу-виводу у форматі моделі schemas.NetworkInfo. Інформація включає перелік інтерфейсів із їхніми IP-адресами, масками підмережі, а також статистику передачі даних, зокрема обсяг відправлених і отриманих байтів, кількість пакетів, помилки та втрати пакетів.

Функція get_system_info призначена для збору загальної інформації про систему. Вона повертає словник (Dict), що містить дані про платформу, час завантаження системи, характеристики процесора, стан оперативної пам’яті, дискові розділи та мережеві інтерфейси. Час завантаження системи отримується з psutil.boot_time() і форматується у вигляді рядка "РРРР-ММ-ДД ГГ:ХХ:СС".

Функція get_platform_info повертає нам інформацію про систему на якій було запущено нашу систему. З неї ми можемо отримати системну інформацію про операційне середовище виконання програми, таку як операційної системи, версію ядра або основну версію системи, детальну версію операційної системи, архітектуру апаратного забезпечення, відомості про процесор, назву дистрибутиву Linux у зрозумілому форматі (за наявності), версію дистрибутиву та його ідентифікатор.

Функція get_cpu_info розроблена для збору та повернення даних про характеристики та поточний стан центрального процесора системи у вигляді словника (Dict). Функція забезпечує доступ до таких параметрів: кількість фізичних ядер процесора, загальна кількість логічних ядер, поточна частота роботи процесора (округлена до двох знаків після коми), мінімальна та максимальна частоти процесора (за наявності даних), та загальний відсоток завантаження процесора (обидва значення округлені до двох знаків після коми).

Для отримання даних про частоту процесора використовується метод psutil.cpu_freq(), а для збору інформації про завантаження методи

psutil.cpu_percent() з параметром interval=1, що забезпечує вимірювання протягом однієї секунди. У разі відсутності даних про частоту процесора відповідні значення встановлюються як None.

Функція get_memory_info призначена для отримання інформації про стан оперативної пам’яті та своп-пам’яті системи. Вона повертає словник (Dict), що містить дані про загальний, доступний та використаний обсяг оперативної пам’яті в гігабайтах, а також відсоток її використання. Додатково надається інформація про своп-пам’ять, включаючи загальний, використаний і вільний обсяг у гігабайтах та відсоток використання.

Функція get_disk_info призначена для збору інформації про дискові розділи системи. Вона повертає список словників (List[Dict]), кожен із яких містить дані про пристрій, точку монтування, тип файлової системи, загальний, використаний і вільний обсяг пам’яті в гігабайтах, а також відсоток використання. Параметр all=False у виклику psutil.disk_partitions дозволяє виключити спеціальні файлові системи. У разі відсутності доступу до розділу або інших помилок обробка цього розділу пропускається.

Функція get_network_info призначена для збору інформації про мережеві інтерфейси та статистику вводу-виводу мережі. Вона повертає словник (Dict), який містить дані про мережеві інтерфейси (назва, IP-адреса, маска підмережі, тип адреси) та статистику вводу-виводу для кожного інтерфейсу (обсяг відправлених і отриманих байтів, пакетів, помилки та втрати пакетів). У разі помилки під час отримання статистики вводу-виводу записується повідомлення про помилку.

Усі розглянуті функції (get_memory_info, get_disk_info, get_network_info, get_system_info) включають механізм обробки винятків, що можуть виникнути під час їх виконання. У разі будь-якої помилки генерується виняток HTTPException із кодом стану 500 та повідомленням, що містить опис помилки.

Бібліотека docker-py забезпечує програмний доступ до API контейнерної платформи Docker через Python, дозволяючи отримувати інформацію про контейнери, їхній стан, журнали виконання, а також виконувати операції з

управління (запуск, зупинка, видалення). Для реалізації цієї функціональності розроблено модуль docker, розташований за шляхом app/docker. У цій директорії містяться файл router.py, що визначає маршрути (роути) для обробки запитів, пов'язаних із контейнерами, а також директорія clients, яка включає файл docker.py в якому описаний клас DockerClient який взаємодіє з бібліотекою docker-py. В ньому реалізовані методи для операцій з контейнерами (запуск, зупинка, видалення, список).

Файл router.py модуля docker визначає маршрути FastAPI для управління контейнерами Docker. Для цього створюється об'єкт APIRouter із префіксом /docker і тегом docker для групування ендпоінтів. Функція get_docker_client повертає екземпляр класу DockerClient для використання в якості залежності (Depends) у всіх маршрутах. Кожен маршрут також використовує залежність get_current_user для перевірки авторизації користувача, повертаючи об'єкт типу User.

Маршрут GET /containers викликає асинхронний метод list_containers із параметром all_containers (за замовчуванням False), повертаючи список об'єктів ContainerListResponse із інформацією про всі контейнери (або лише активні, якщо all_containers=False).

Маршрут POST /containers створює новий контейнер, приймаючи об'єкт ContainerCreate, який містить параметри, такі як ім'я образу, тег, мережа, порти, томи, змінні середовища, пристрої, команда, привілеї, розподіл CPU і політика перезапуску, і передає їх у метод create_container, повертаючи об'єкт ContainerDetailResponse.

Маршрут GET /containers/{container_id} отримує деталі контейнера за його ідентифікатором через метод get_container, повертаючи об'єкт ContainerDetailResponse.

Маршрут PUT /containers/{container_id} оновлює конфігурацію контейнера, приймаючи об'єкт ContainerUpdate, конвертуючи його у словник без невизначених

значень (exclude_unset=True), і передає у метод update_container, повертаючи оновлені дані.

Маршрут DELETE /containers/{container_id} видаляє контейнер із параметром force (за замовчуванням False), викликаючи метод delete_container і повертаючи об’єкт ContainerOperationResponse із повідомленням про результат.

Маршрути POST /containers/{container_id}/start, POST /containers/{container_id}/stop і POST /containers/{container_id}/restart відповідно запускають, зупиняють і перезапускають контейнер за його ідентифікатором, викликаючи методи start_container, stop_container і restart_container класу DockerClient. Кожен із цих маршрутів повертає оновлені дані контейнера у форматі ContainerDetailResponse.

ДляКлас DockerClient реалізує інтерфейс для управління контейнерами Docker через бібліотеку docker-py, яка надає доступ до API Docker-демона на сервері. Ініціалізація класу (__init__) встановлює з’єднання з Docker-демоном за допомогою методу from_env(), який автоматично визначає конфігурацію з системних змінних середовища, таких як DOCKER_HOST чи DOCKER_TLS_VERIFY. У разі виникнення помилок підключення, наприклад, через відсутність доступу до демона чи неправильну конфігурацію, метод обробляє винятки DockerException і генерує HTTPException із кодом стану 500, додаючи детальний опис помилки для подальшого аналізу.

Допоміжні методи класу призначені для форматування даних, отриманих від Docker API, у зручний для програми формат. Метод _format_ports перетворює структуру портів, отриману від API, у словник, де ключі – порти контейнера, а значення – відповідні прив’язки хоста, враховуючи можливу множинність зв’язків. Метод _format_volumes обробляє інформацію про томи, формуючи список із полями джерела, місця призначення, типу монтування та режиму доступу (тільки для читання), використовуючи значення за замовчуванням, якщо вони відсутні. Метод _format_environment зберігає або ініціалізує порожній список змінних середовища, тоді як _format_command об’єднує список аргументів команди в рядок

або повертає сам рядок, якщо він переданий як єдиний параметр. Метод _get_cpu_allocation визначає рівень розподілу обчислювальних ресурсів CPU на основі часток (CpuShares), де значення до 512 відповідає "low", до 1024 — "medium", а вище 1024 — "high", із початковим значенням 1024 як стандартним.

Асинхронний метод list_containers повертає список контейнерів типу ContainerListResponse, викликаючи containers.list() із параметром all_containers, який за замовчуванням False, але може бути встановлений у True для включення зупинених контейнерів. Кожен об’єкт у списку містить ідентифікатор, назву, статус, образ (тег або ідентифікатор образу), час створення, відформатовані порти, томи, пристрої, змінні середовища, привілеї, політику перезапуску, розподіл CPU, мережеві налаштування, команду запуску та тег, якщо він визначений. У разі виникнення помилок, таких як збої підключення чи доступу до API, метод обробляє DockerException і генерує HTTPException із кодом 500.

Метод create_container реалізує створення нового контейнера, формуючи ім’я образу з тегом (наприклад, image:tag), якщо тег вказано, і конвертуючи рівень cpu_allocation у відповідну кількість часток (cpu_shares) за попередньо визначеним словником (512 для "low", 1024 для "medium", 2048 для "high"). Змінні середовища перетворюються у список рядків формату "ключ=значення", а конфігурація хоста створюється через create_host_config із параметрами привілеїв, часток CPU, політики перезапуску (за замовчуванням "unless-stopped"), пристроїв, прив’язок портів, томів і мережі. Створення контейнера виконується через containers.create() із параметром detach=True для асинхронного режиму, після чого повертаються деталі через виклик get_container. Помилки, наприклад, через неіснуючий образ чи недостатні права, обробляються як DockerException із генерацією HTTPException кодом 500.

Метод get_container отримує детальну інформацію про контейнер за його ідентифікатором через containers.get(), повертаючи об’єкт ContainerDetailResponse із усіма атрибутами, включаючи відформатовані дані з атрибутів контейнера (attrs).

Метод update_container оновлює конфігурацію контейнера, передаючи параметри через kwargs до методу update(), і повертає оновлені дані.

Метод delete_container видаляє контейнер через remove() із опціональним параметром force, який дозволяє видаляти працюючі контейнери, повертаючи повідомлення про успіх у ContainerOperationResponse.

Методи start_container, stop_container і restart_container відповідно запускають, зупиняють і перезапускають контейнер, використовуючи методи API (start(), stop(), restart()), і повертають оновлені дані через get_container.

Усі методи асинхронно обробляють винятки DockerException, генеруючи HTTPException із кодом 404 для не знайденого контейнера (get_container) або 500 для інших помилок із детальним описом.

3.4 Реалізація клієнтської частини

Клієнтська частина розроблена з використанням бібліотеки React у поєднанні з Vite як інструментом для швидкого збирання проекту, а також Tailwind CSS для ефективного застосування CSS-стилів.

Структура клієнтської частини представлена в додатку X і базується на компонентному підході. Цей підхід передбачає розподіл структури та логіки на окремі компоненти, що сприяє їх повторному використанню. Наприклад, для елемента кнопки (тег <button>) створено компонент Button, який приймає параметри для налаштування, дозволяючи використовувати його в коді у вигляді <Button param1={} param2={}/>. Деякі параметри можуть бути статичними, що зменшує дублювання коду. Для реалізації цього підходу створено директорію components, яка містить усі компоненти клієнтської частини. На рисунку 3.6 зображено компонент моніторингу CPU та RAM, який складається з декількох вкладених компонентів.

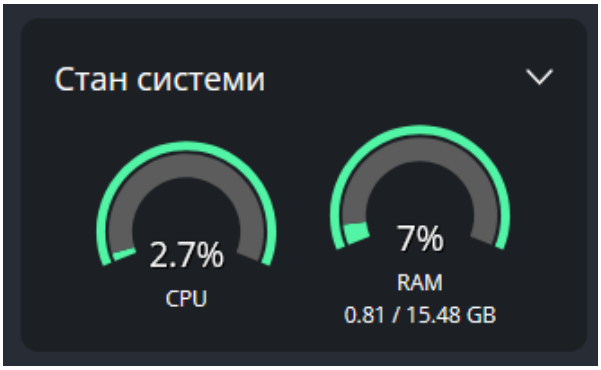


Рисунок 3.5 – Компонент який показує стан системи

Для забезпечення взаємодії з серверною частиною створено директорію api, яка містить файли, призначені для роботи з API. У файлі config.js визначено конфігураційні параметри, зокрема таймаути, інтервали оновлення даних і базовий URL API. У файлі apiClient.js реалізовано клас ApiClient, який містить методи для взаємодії з API. Метод request використовується як базова функція для надсилання запитів, приймаючи як аргументи кінцеву точку (endpoint) та необов'язковий об'єкт options, що може включати додаткові параметри, наприклад, заголовки (headers). Запити до API виконуються за допомогою функції fetch. У класі також реалізовано утилітні методи для обробки різних типів HTTP-запитів: get для GET-запитів, post для POST-запитів, put для PUT-запитів і delete для DELETE-запитів. Наприкінці файлу створюється екземпляр класу ApiClient, який зберігається у змінній apiClient і експортується для подальшого використання.

Для спрощення логіки взаємодії з API в компонентах створено кілька файлів, які описують роботу з окремими частинами API. Ці файли розташовані в директорії api/services/.

У файлі authService.js реалізовано об'єкт, що містить функції для обробки запитів до API, пов'язаних із авторизацією.

Функція register приймає аргумент userData і за допомогою методу apiClient.post() надсилає запит на кінцеву точку /auth/register із даними користувача для реєстрації.

Функція login приймає аргумент userData і через apiClient.post() надсилає запит на кінцеву точку /auth/login із даними користувача для авторизації та отримання токена.

Функція logout відповідає за завершення сеансу користувача. Вона отримує токен із localStorage, формує заголовок запиту з ключем Authorization і значенням Bearer ${token}, після чого виконує запит через apiClient.post() на кінцеву точку /auth/logout. Після виконання запиту токен видаляється з localStorage, а сторінка перезавантажується.

Функція init надсилає запит на кінцеву точку /auth/init за допомогою apiClient.get(). Цей запит повертає інформацію про те, чи є запуск системи першим. На основі цієї інформації визначається, чи відображати користувачу форму реєстрації.

В файлі dockerService.js знаходяться об’єкт dockerService який експортується для забезпечення взаємодії з API для управління контейнерами Docker. Він містить набір функцій, які виконують запити до відповідних кінцевих точок API з використанням токена авторизації, отриманого з localStorage.

Функція listContainers(allContainers = false) – виконує GET-запит до кінцевої точки /docker/containers?all_containers=${allContainers} для отримання списку контейнерів. Параметр allContainers визначає, чи включати в результат усі контейнери, включно із зупиненими.

Функція listContainersStats() – надсилає GET-запит до кінцевої точки /docker/containers/stats для отримання статистики використання ресурсів контейнерами.

Функція createContainer(containerData) – надсилає POST-запит до кінцевої точки /docker/containers із даними containerData для створення нового контейнера.

Функція getContainer(containerId) – виконує GET-запит до кінцевої точки /docker/containers/${containerId} для отримання інформації про конкретний контейнер за його ідентифікатором.

Функція updateContainer(containerId, updateData) – надсилає PUT-запит до кінцевої точки /docker/containers/${containerId} із даними updateData для оновлення параметрів контейнера.

Функція deleteContainer(containerId, force = false) – виконує DELETE-запит до кінцевої точки /docker/containers/${containerId}?force=${force} для видалення контейнера. Параметр force визначає, чи примусово видаляти контейнер.

Функція startContainer(containerId) – надсилає POST-запит до кінцевої точки /docker/containers/${containerId}/start для запуску контейнера.

Функція stopContainer(containerId) – виконує POST-запит до кінцевої точки /docker/containers/${containerId}/stop для зупинки контейнера.

Функція restartContainer(containerId) – надсилає POST-запит до кінцевої точки /docker/containers/${containerId}/restart для перезапуску контейнера.

Усі запити включають заголовок Authorization із значенням Bearer ${token}, отриманим із localStorage, для забезпечення автентифікації.

В файлі systemService.js знаходяться об’єкт systemService який експортується для забезпечення взаємодії з API для отримання інформації про систему. Він містить набір функцій, які виконують запити до відповідних кінцевих точок API. Усі запити використовують токен автентифікації, отриманий із localStorage, для формування заголовка Authorization із значенням Bearer ${token}

Функція getSystemInfo() – виконує GET-запит до кінцевої точки /system/ для отримання загальної інформації про систему.

Функція getCpuInfo() – надсилає GET-запит до кінцевої точки /system/cpu для отримання даних про центральний процесор.

Функція getMemoryInfo(): виконує GET-запит до кінцевої точки /system/memory для отримання інформації про оперативну пам’ять.

Функція getDiskInfo() – надсилає GET-запит до кінцевої точки /system/disk для отримання даних про дисковий простір.

Функція getNetworkInfo() – виконує GET-запит до кінцевої точки /system/network для отримання інформації про мережеві параметри.

Після опису всіх функцій для взаємодії з API можна перейти до детального опису логіки роботи клієнтської частини системи.

На рисунку 3.6 зображено загальний вигляд клієнтської системи, тобто те що буде бачити користувач

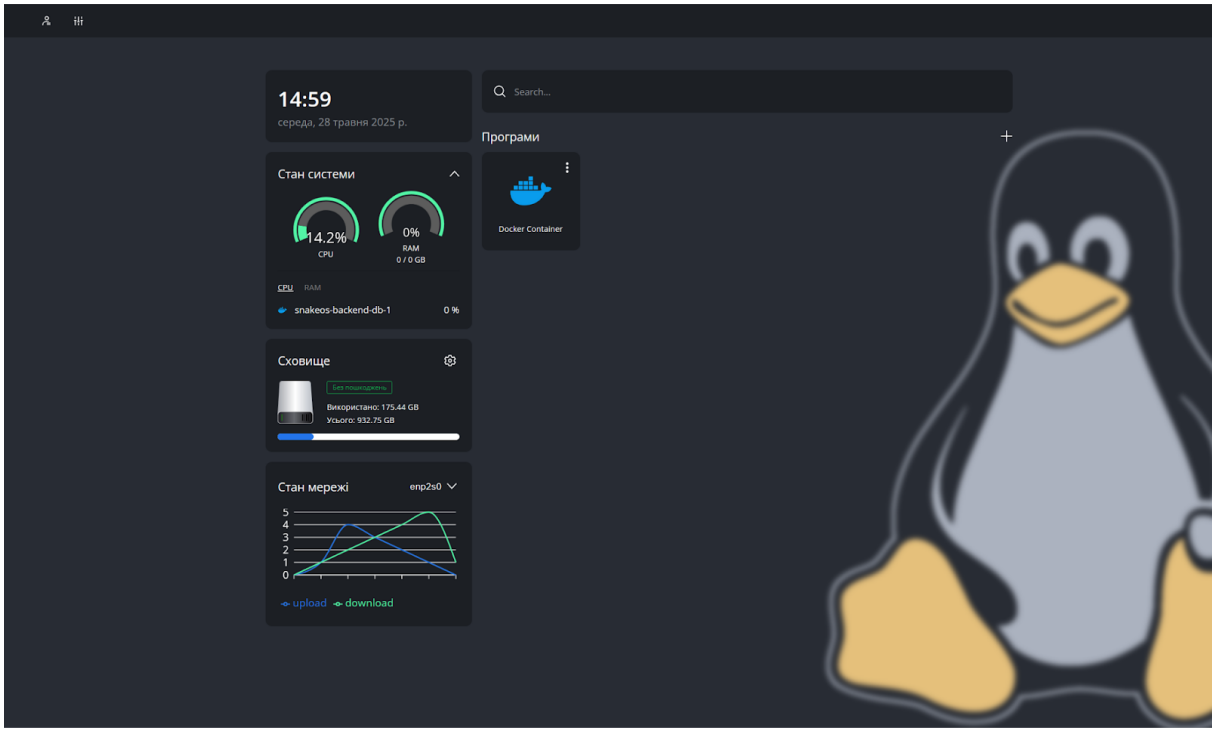


Рисунок 3.6 – Вигляд розробленої системи керування віддаленим сервером з веб-орієнтованим інтерфейсом

Компонент App є основним компонентом клієнтської частини програми, який керує відображенням сторінок залежно від стану авторизації користувача. У компоненті визначено два стани: login для відстеження статусу авторизації та register для визначення режиму відображення форми реєстрації. За допомогою хука useEffect при ініціалізації компонента викликається метод authService.init(), який перевіряє, чи є поточний запуск системи першим. Якщо система ініціалізована (res.initialized є true), перевіряється наявність токена в localStorage: за наявності токена встановлюється стан login у true, інакше – у false. Якщо система не ініціалізована, стани login встановлюється в false (на рисунку 3.7 представлено форму авторизації, відображену за умови, що стан login має значення false.), а

register – у true, що забезпечує відображення форми реєстрації (Рисунок 3.8). У розмітці компонент повертає елемент div із класом app, який містить фоновий елемент із умовним класом background blurred за наявності авторизації та контейнер content для відображення сторінок. Залежно від стану login, відображається компонент Home (для авторизованих користувачів) або Login (з передачею пропсів register і setLogin).

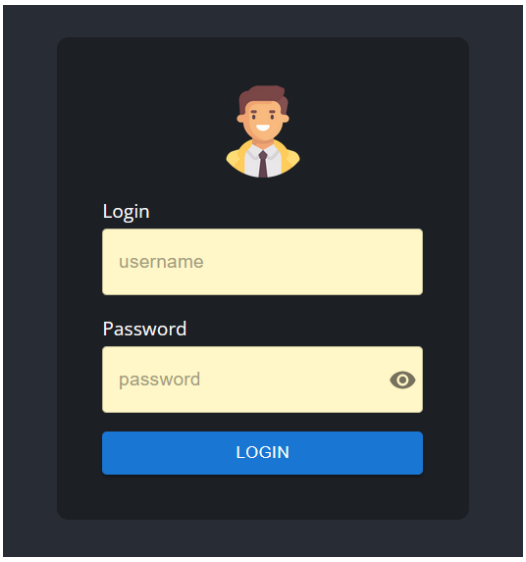


Рисунок 3.7 – Вигляд форми авторизації користувача

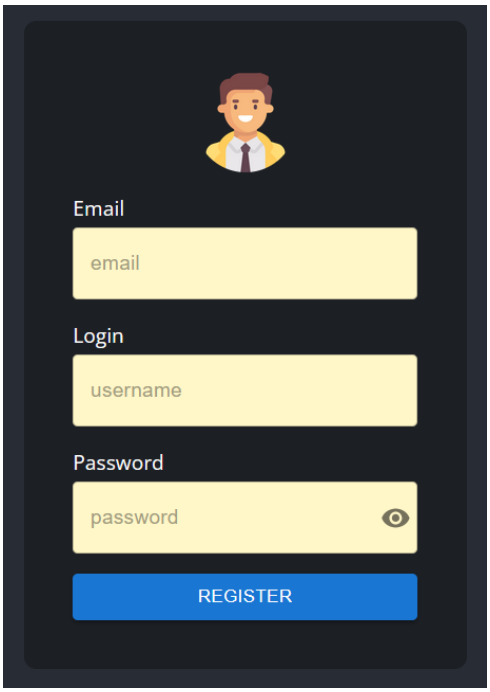


Рисунок 3.8 – Вигляд форми реєстрації користувача

Компонент Login забезпечує створення інтерфейсу для авторизації та реєстрації користувачів. Модуль authService використовується для взаємодії з API. Компонент містить стани showPassword для керування відображенням пароля, loading для індикації виконання запиту, errors для зберігання помилок валідації та serverError для повідомлень про помилки сервера. Функція validateForm перевіряє обов'язковість поля username, наявність і мінімальну довжину пароля (6 символів), а в режимі реєстрації також коректність формату email за допомогою регулярного виразу, оновлюючи стан errors і повертаючи результат валідації. Функція handleForm збирає дані форми (username, password, а для режиму реєстрації – email) з відповідних полів, виконує валідацію через validateForm і припиняє виконання у разі невідповідності.

У режимі реєстрації викликається authService.register із даними форми, а в режимі входу – authService.login, після чого отриманий токен зберігається в localStorage, а стан setLogin встановлюється в true. Помилки сервера обробляються з відображенням у змінній serverError, а стан loading керує індикатором завантаження. Інтерфейс включає компонент Container, аватар користувача через зображення з зовнішнього джерела, відображення помилок сервера через Alert, поле email для режиму реєстрації з валідацією, поля username і password через FormControl і OutlinedInput із підтримкою відображення помилок валідації. Поле password містить кнопку для перемикання видимості пароля (Visibility/VisibilityOff). Кнопка відправки форми (Button) відображає текст "Register" або "Login" залежно від режиму, а під час виконання запиту показує індикатор CircularProgress.

Компонент Home є основною сторінкою клієнтської частини програми, що відображається після авторизації користувача. У компоненті визначено стан storage із початковими значеннями used: 0, all: 0 і used_percentage: 0 для зберігання даних про використаний і загальний обсяг дискового простору та відсоток використання. За допомогою хука useEffect при ініціалізації компонента викликається асинхронна

функція fetchData, яка використовує метод systemService.getDiskInfo() для отримання інформації про диски. З відповіді сервера береться інформація про перший диск (res[0]), і стан storage оновлюється значеннями used, all і used_percentage, отриманими з API. Розмітка компонента включає контейнер із класом app, який містить компонент Header і контейнер mainContainer, поділений на секції left і right. У секції left розміщено компоненти DateTime для відображення дати та часу, SystemStatus для моніторингу стану системи, Storage із параметрами used, all, used_percentage зі стану storage і статичним параметром status="Без пошкоджень" для відображення інформації про сховище, а також NetworkStatus для показу мережевих даних. У секції right розташовано компоненти SearchBar для пошуку та Programs для відображення списку Docker контейнерів.

Компонент SystemStatus (кінцевий вигляд зображено на рисунку 3.9) призначений для відображення інформації про стан системи, зокрема використання процесора (CPU) і оперативної пам’яті (RAM). У компоненті визначено стани isDropDownOpen для керування відображенням випадаючого меню, loadCpuData з початковими значеннями total_cpu_usage: 0, physical_cores: 0, total_cores: 0 для зберігання даних про використання CPU, і loadMemoryData з початковими значеннями percentage: 0, used: 0, total: 0 для даних про RAM. За допомогою хука useEffect викликається асинхронна функція fetchData, яка отримує дані про CPU через systemService.getCpuInfo() і про RAM через systemService.getMemoryInfo(), оновлюючи відповідні стани. У разі помилки виводиться повідомлення в консоль. Дані оновлюються кожні 2 секунди за допомогою setInterval, а при знищенні компонента інтервал очищається. Розмітка компонента включає контейнер Container, який містить заголовок із текстом "Стан системи" і елементом ArrowDown для перемикання стану випадаючого меню, блок із двома компонентами GaugeBar для відображення використання CPU і RAM, а також блок із компонентом DropDown, видимість якого залежить від стану isDropDownOpen. Цей DropDown призначений для відображення випадаючого меню з інформацією про використання ресурсів контейнерами Docker, зокрема CPU і RAM. У

компоненті визначено стани containers для зберігання списку контейнерів і activeTab для керування активною вкладкою (0 для CPU, 1 для RAM). За допомогою хука useEffect викликається метод dockerService.listContainersStats() для отримання статистики контейнерів, оновлюючи стан containers отриманими даними. Цей запит виконується при ініціалізації компонента і повторюється кожні 2 секунди за допомогою setInterval, а при знищенні компонента інтервал очищається. Функція handleTabClick змінює активну вкладку через оновлення стану activeTab.

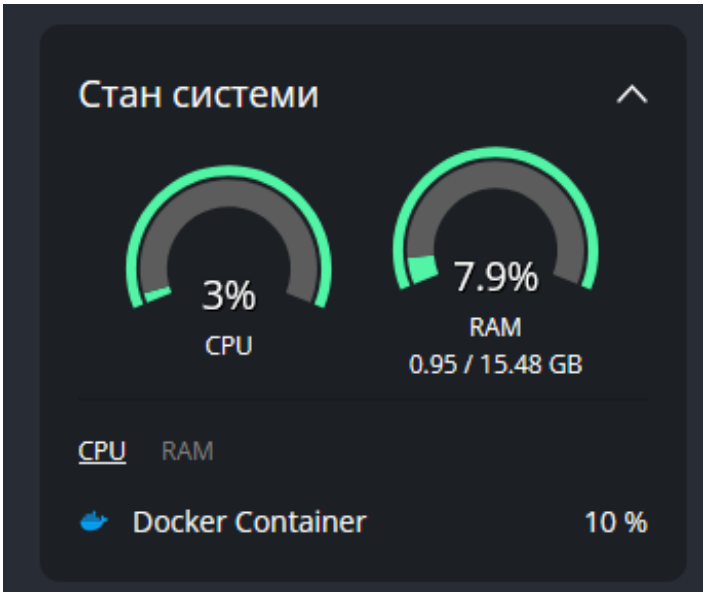


Рисунок 3.9 – Вигляд компонента SystemStatus

Компонент Storage (вигляд компонента зображено на рисунку 3.10) призначений для відображення інформації про стан дискового сховища. Компонент приймає пропси used, all, status і used_percentage для відображення даних про використаний і загальний обсяг сховища, статус і відсоток використання. У компоненті визначено стан isModalOpen для керування відображенням модального вікна. Функція openModal встановлює isModalOpen у true, а closeModal – у false. Розмітка компонента включає контейнер Container, який містить заголовок із текстом "Сховище" і кнопкою IconButton із компонентом SettingsIcon для відкриття модального вікна, блок контенту з іконкою DiscIcon і текстовим блоком, що

відображає статус, використаний обсяг (used) і загальний обсяг (all) у гігабайтах, а також компонент ProgressBar для візуалізації відсотка використання (used_percentage) з фоновим кольором білого. Модальне вікно DiscModal відображається за умови isModalOpen=true і закривається через onClose.

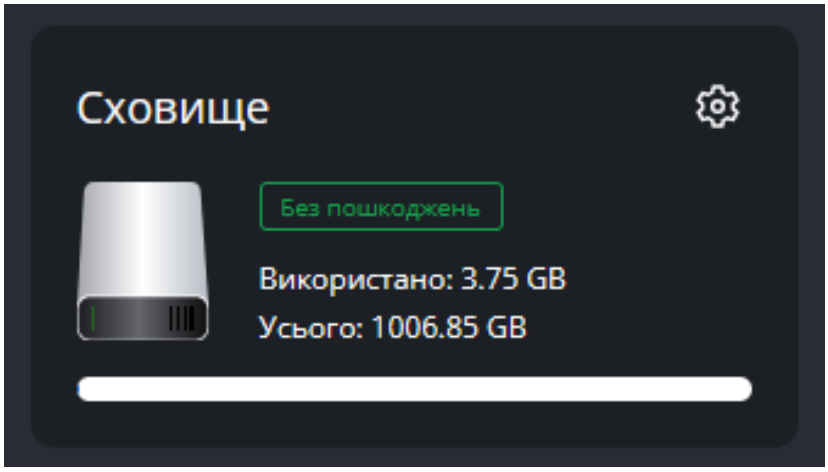


Рисунок 3.10 – Вигляд компонента Storage

Компонент DiscModal (вигляд компонента зображено на рисунку 3.11) призначений для відображення модального вікна з інформацією про дискові сховища системи. Компонент приймає пропси open для керування відображенням модального вікна та onClose для його закриття. У компоненті визначено стани value для керування активною вкладкою і storage для зберігання списку даних про диски. За допомогою хука useEffect викликається метод systemService.getDiskInfo(), який отримує інформацію про диски та оновлює стан storage. Функція handleChange змінює активну вкладку через оновлення стану value. Розмітка компонента включає Modal із затемненим фоном і розмиттям, який містить контейнер Box, розташований у центрі екрана. У контейнері відображається заголовок "Управління сховищами" з кнопкою закриття (CloseIcon), роздільна лінія Divider, вкладка Tabs із єдиним елементом "Сховище" і блок контенту, що відображає список дисків. Для кожного диску в стані storage створюється блок із іконкою StorageIcon, інформацією про шлях монтування (mountpoint, відсоток використання), тип файлової системи (filesystem_type), пристрій (device),

доступний і загальний обсяг (free і total), а також компонент ProgressBar для візуалізації відсотка використання (percentage).

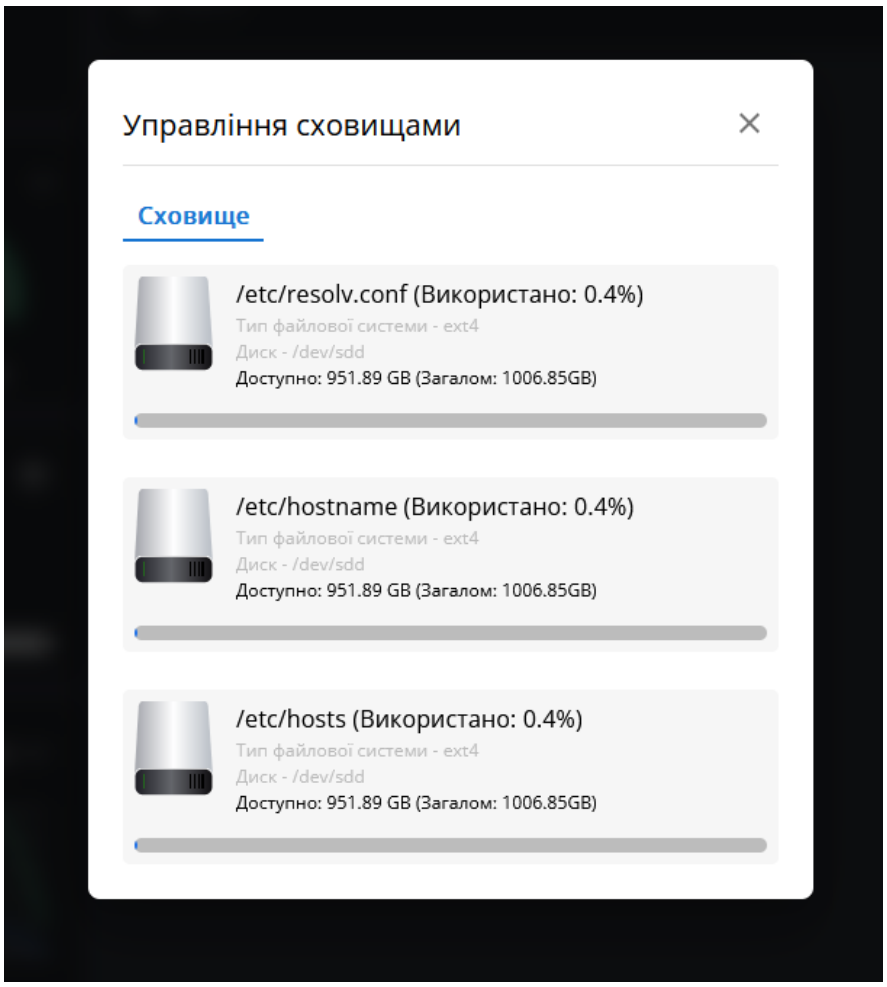


Рисунок 3.11 – Вигляд компонента DiscModal

Компонент NetworkStatus (вигляд компонента зображено на рисунку 3.12) призначений для відображення стану мережевої активності системи. Він ньому використовуються компоненти CartesianGrid, Legend, Line, LineChart, ResponsiveContainer і YAxis з бібліотеки Recharts [54] для створення графіка. У компоненті визначено стани selectedInterface для зберігання вибраного мережевого інтерфейсу, interfaces для списку доступних мережевих інтерфейсів і data для зберігання даних про мережевий трафік. Перший хук useEffect викликає метод systemService.getNetworkInfo(), який отримує список мережевих інтерфейсів і встановлює перший доступний інтерфейс як вибраний, якщо selectedInterface не

визначено. Другий хук useEffect, залежний від selectedInterface, викликає асинхронну функцію fetchData для отримання даних про мережевий трафік вибраного інтерфейсу через systemService.getNetworkInfo(). Дані про вхідний (bytes_recv) і вихідний (bytes_sent) трафік конвертуються в мегабайти, форматуються до двох знаків після коми та додаються до стану data, зберігаючи лише останні 10 точок даних. Запити виконуються кожні 5 секунд за допомогою setInterval, а при знищенні компонента інтервал очищається. У разі помилки виводиться повідомлення в консоль. Розмітка компонента включає контейнер Container із заголовком "Стан мережі" і компонентом NetworkSelect для вибору інтерфейсу, а також ResponsiveContainer із графіком LineChart, який відображає дані про вхідний (download) і вихідний (upload) трафік у мегабайтах. Графік містить сітку CartesianGrid, вісь YAxis із форматуванням значень у мегабайтах, легенду Legend і дві лінії Line для вхідного і вихідного трафіку з відповідними кольорами та стилем.

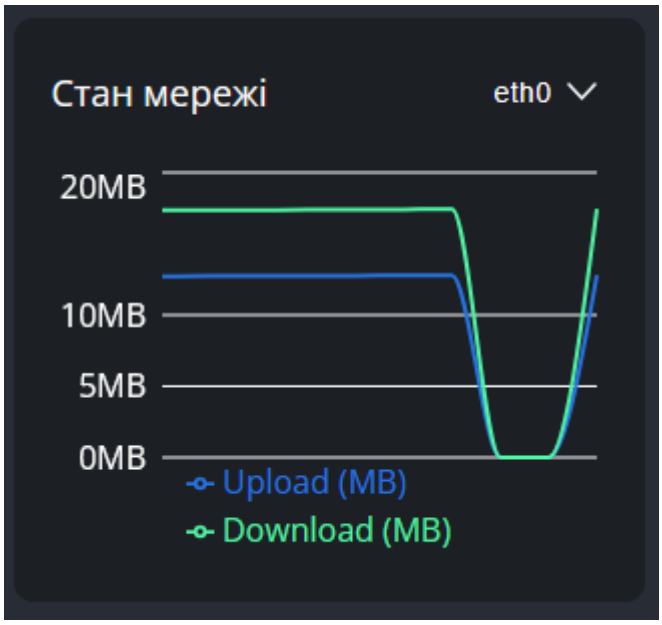


Рисунок 3.12 – Вигляд компонента NetworkStatus

Компонент DateTime (вигляд компонента зображено на рисунку 3.13) призначений для відображення поточної дати та часу. У компоненті визначено стан

currentDate, який ініціалізується викликом getDateTime() для отримання початкових значень дати і часу. За допомогою хука useEffect встановлюється інтервал, який кожну секунду оновлює стан currentDate шляхом повторного виклику getDateTime(). При знищенні компонента інтервал очищається для запобігання витоку пам’яті. Розмітка компонента включає контейнер Container, який містить елемент h1 із класом time для відображення часу та елемент h2 із класом date для відображення дати, отриманих зі стану currentDate.

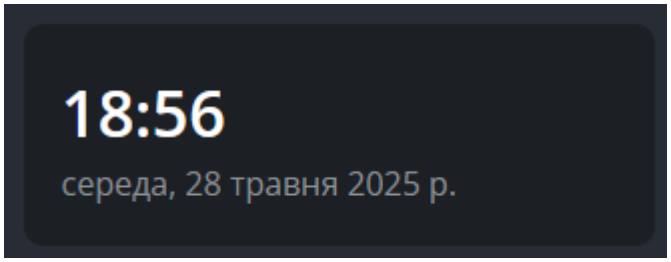


Рисунок 3.13 – Вигляд компонента DateTime

На рисунку 3.14 зображено інтерфейсну частину, яка відповідає за відображення системних метрик і забезпечує користувачу можливість моніторингу стану системи.

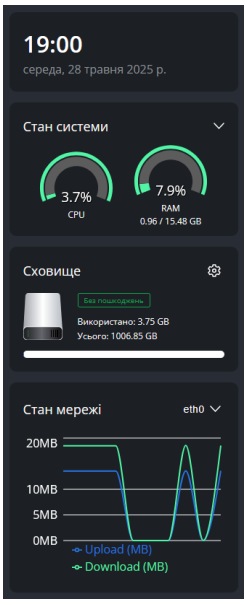


Рисунок 3.14 – Вигляд інтерфейсної частини, яка відповідає за відображення системних метрик

У правій частині інтерфейсу розташовано компонент SearchBar і секцію Programs, яка містить список контейнерів Docker.

Компонент SearchBar призначений для створення пошукового рядка з можливістю виконання пошуку через Google. У компоненті визначено стан searchValue для зберігання введеного користувачем пошукового запиту. Функція handleKeyPress реагує на натискання клавіші Enter, кодуючи значення searchValue за допомогою encodeURIComponent і відкриваючи нову вкладку браузера з URL для пошуку в Google. Розмітка компонента включає контейнер Container із шириною 810 пікселів, який містить елемент із класом container, що об'єднує іконку SearchIcon і поле введення input. Поле введення має стилі input, атрибут placeholder із текстом "Search...", значення, прив'язане до стану searchValue, обробник події onChange для оновлення стану при введенні тексту, а також обробник onKeyDown для виклику функції handleKeyPress.

Компонент Programs призначений для відображення списку Docker-контейнерів із можливістю додавання нових через модальне вікно. У компоненті визначено стани: programs для зберігання списку контейнерів, ініціалізованого масивом із одним об'єктом, що має ідентифікатор, зображення і текст; isModalOpen для керування відображенням модального вікна; newProgram для зберігання даних нового контейнера з початковими значеннями image і text. За допомогою хука useEffect викликається асинхронна функція fetchPrograms, яка отримує список контейнерів через dockerService.listContainers() і оновлює стан programs. Функція handleOpenModal скидає значення newProgram і відкриває модальне вікно, handleCloseModal закриває його, а handleSaveProgram додає новий контейнер до списку programs із автоматично згенерованим ідентифікатором і закриває модальне вікно. Розмітка компонента включає контейнер із класом container, який містить меню з текстом "Програми" і кнопкою з іконкою PlusIcon для відкриття модального вікна, блок із класом programs для відображення списку контейнерів через компонент Program із передачею параметрів container_id, image (із запасним значенням у разі відсутності зображення) і text, а також модальне вікно

ProgramSettingsModal, яке відображається за умови isModalOpen=true і дозволяє зберегти новий контейнер через onSave.

Компонент ProgramSettingsModal призначений для створення модального вікна, яке забезпечує налаштування параметрів або створення Docker контейнера. Компонент приймає пропси open для керування відображенням модального вікна, onClose для його закриття, programName для назви програми та onSave для збереження даних. У компоненті визначено стан programData, що містить параметри контейнера: dockerImage, dockerTag, title, iconUrl, вкладений об’єкт webUI із полями protocol, host, port, path, а також network, ports, volumes, envVars, devices, command, privileges, cpuAllocation, restartPolicy. Функція handleChange оновлює стан programData, враховуючи як звичайні поля, так і вкладені об’єкти, наприклад webUI. Функція handleSave формує об’єкт formData, додаючи повне URL для webUI, викликає onSave і закриває модальне вікно. У контейнері відображається заголовок із назвою програми, поля введення TextField для dockerImage, dockerTag, title, iconUrl, блок для налаштування webUI з вибором протоколу через CustomSelect і полями для хоста, порту та шляху, вибір мережі через CustomSelect, секції AddSection для портів, томів, змінних середовища, пристроїв і команди контейнера, перемикач Switch для привілеїв, слайдер StyledSlider, а також вибір розподілу процесора та політики перезапуску через CustomSelect. У нижній частині розташована кнопка "Save" для збереження налаштувань. При збереженні форми визначається, чи відбувається створення нового контейнера, чи оновлення існуючого. У разі створення нового контейнера викликається метод API dockerService.createContainer(containerData), інакше застосовується dockerService.updateContainer(containerId, updateData). На рисунку 3.15 представлено вигляд компонента ProgramSettingsModal відповідно при редагуванні вже існуючих значень і при створенні нового Docker контейнера.

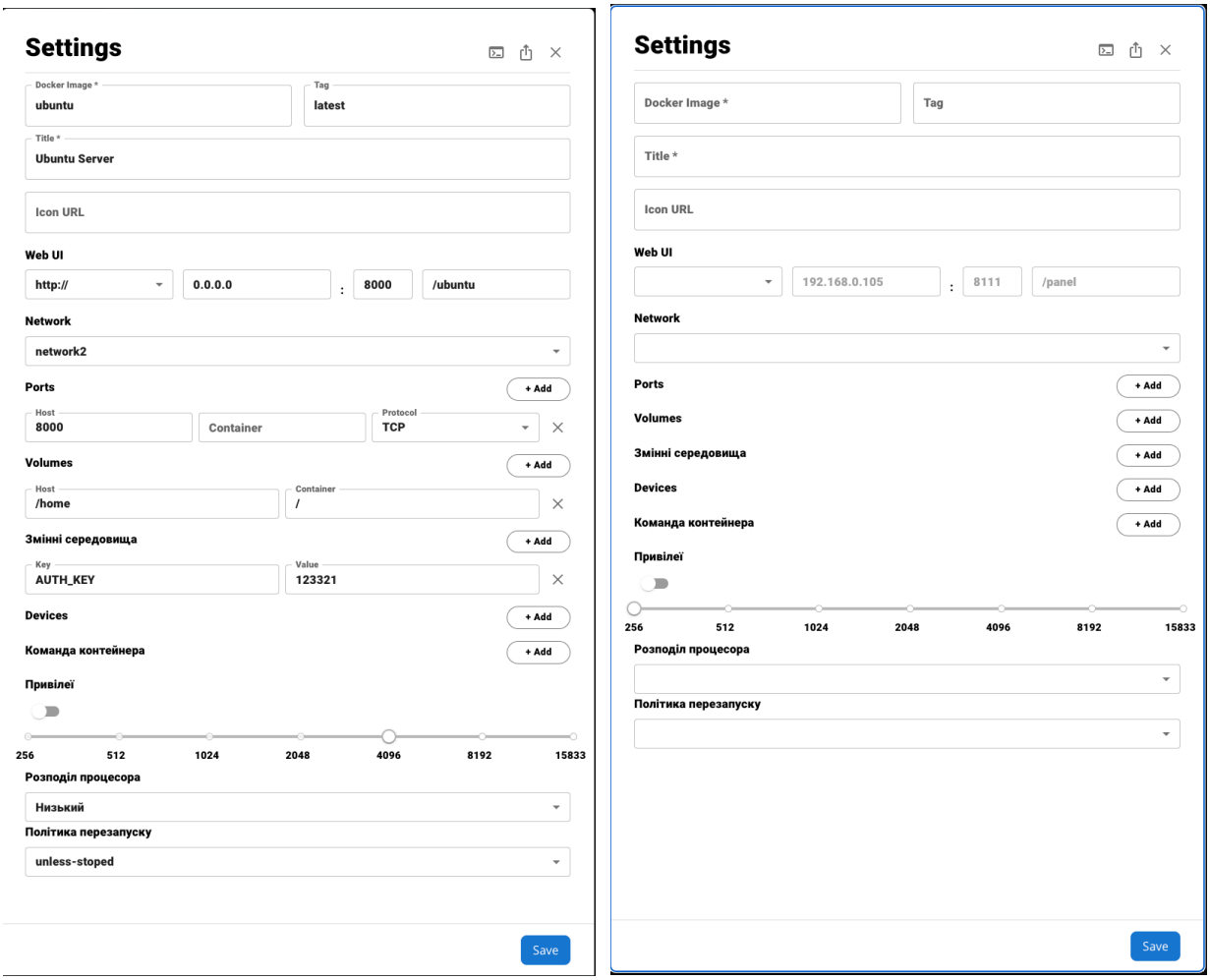


Рисунок 3.15 – Вигляд компонента ProgramSettingsModal, де зліва відображено інтерфейс для редагування існуючих значень, а справа – для створення нового Docker-контейнера.

На цьому етапі розробка клієнтської частини завершується, було реалізовано основний функціонал для моніторингу системи, а також забезпечено інтеграцію API з фронтенд-частиною для отримання актуальних даних у реальному часі.

3.5 Реалізація інсталяційного bash-скрипта

Bash-скрипт SnakeOS Installer Script v1.0 призначений для автоматизованого розгортання програмної системи SnakeOS, яка складається з серверної (backend) та клієнтської (frontend) частин, на операційних системах Linux, зокрема Debian-подібних (Ubuntu, Debian, Mint), RedHat-подібних (RHEL, CentOS, Fedora) та Arch

Linux. Скрипт забезпечує повний цикл встановлення, включаючи налаштування залежностей, бази даних, веб-сервера, системного сервісу та брандмауера, а також розгортання компонентів із репозиторіїв GitHub. Виконання скрипта передбачає запуск із правами суперкористувача через команду завантаження та виконання з віддаленого джерела.

На початку скрипт визначає конфігураційні параметри, такі як URL-адреси репозиторіїв для клонування backend та frontend, шляхи встановлення (/opt/snakeos для компонентів та /var/www/snakeos для статичних файлів), а також параметри бази даних PostgreSQL, включаючи автоматично згенерований пароль. Для ізоляції процесів створюється системний користувач та визначається назва сервісу для backend. Виконання припиняється у разі відсутності прав root, що забезпечує безпечне виконання критичних операцій.

Для інформативного виведення повідомлень використовується кольорове форматування з підтримкою ASCII-арту для позначення початку роботи. Спеціальна функція відповідає за виведення повідомлень про успіх, помилки або інформаційні стани, що полегшує відстеження процесу встановлення. Скрипт припиняє роботу при виникненні будь-якої помилки завдяки використанню директиви set -e, забезпечуючи надійність виконання.

Перший етап роботи скрипта передбачає визначення типу операційної системи та встановлення необхідних залежностей. Для Debian-подібних систем застосовується пакетний менеджер apt-get для встановлення Git, Curl, Python 3, PostgreSQL, Nginx та Node.js (через NodeSource). У RedHat-подібних системах використовується dnf або yum, а також ініціалізується PostgreSQL. Для Arch Linux застосовується pacman із відповідною ініціалізацією бази даних. Якщо Docker відсутній, він встановлюється через офіційний скрипт. У разі невпізнаного дистрибутива скрипт завершує роботу з помилкою.

Наступним кроком є налаштування бази даних PostgreSQL. Скрипт запускає та активує відповідну службу, генерує випадковий пароль за допомогою утиліти openssl, створює базу даних та користувача з необхідними привілеями. Якщо база

або користувач уже існують, пароль оновлюється, що забезпечує гнучкість при повторному виконанні.

Для встановлення backend-компонента створюється системний користувач, клонується репозиторій у визначену директорію, створюється віртуальне середовище Python, встановлюються залежності (FastAPI, Uvicorn, psycopg2-binary тощо), а також формується файл конфігурації .env із параметрами підключення до бази даних. Права доступу до файлів обмежуються для підвищення безпеки.

Далі скрипт створює systemd-сервіс для запуску backend через Uvicorn на порту 8000 із автоматичним перезапуском та залежністю від служб мережі, PostgreSQL і Docker. Після створення сервісу перевіряється його статус, і в разі невдачі виводиться повідомлення про необхідність перевірки логів.

Frontend-компонент встановлюється шляхом клонування відповідного репозиторію, встановлення залежностей через npm, виконання збірки та копіювання статичних файлів у директорію для обслуговування Nginx. Права доступу до файлів налаштовуються для користувача www-data.

Для обслуговування frontend та маршрутизації запитів до backend створюється конфігурація Nginx, яка обробляє запити до статичних файлів та перенаправляє API-запити на порт 8000. Стандартна конфігурація Nginx видаляється, нова активується, перевіряється на коректність, після чого служба перезапускається.

Налаштування брандмауера передбачає відкриття портів 80 (HTTP) та 443 (HTTPS) через ufw або firewalld залежно від системи. Якщо брандмауер не виявлено, цей етап пропускається.

На завершення скрипт виводить інформаційне повідомлення з IP-адресою сервера, розташуванням файлів, інструкціями з керування сервісом та вказівкою на файл із паролем до бази даних. Усі етапи об’єднані в основну функцію, яка забезпечує послідовне виконання.

Скрипт відзначається модульною структурою, що полегшує його модифікацію, крос-дистрибутивною сумісністю, обробкою помилок та

безпековими заходами, такими як ізоляція користувачів та обмеження прав доступу. Скрипт може бути збережений на віддаленому сервері, після чого його використання на цільовому сервері здійснюється за допомогою команди curl -fsSL https://your-domain.com/install.sh | sudo bash. Альтернативно, можливо завантажити скрипт, зберегти його у файл та виконати локально. На рисунку 3.17 зображено успішне повідомлення після встановлення системи на віддалений сервер.

Рисунок 3.17 – Вигляд успішного повідомлення після встановлення системи

3.6. Висновки до третього розділу

У третьому розділі розроблено програмно-технічний засіб для керування віддаленим сервером, що включає серверну частину на базі FastAPI для збору системних метрик (psutil), управління контейнерами (Docker API) та авторизації (JWT-токени), а також клієнтську частину на React для інтерактивної візуалізації даних і адміністрування в реальному часі. Для спрощення розгортання створено універсальний інсталяційний bash-скрипт, що забезпечує відповідність продукту функціональним вимогам.

ВИСНОВКИ

У роботі за результатами виконаних теоретичних та практичних досліджень було розроблено систему керування віддаленим сервером з веб-орієнтованим інтерфейсом, яка вирішує поставлені завдання та відповідає визначеним вимогам. Дана система надає користувачам інтуїтивно зрозумілий інструмент для моніторингу системних ресурсів та управління Docker-контейнерами, спрощуючи процес адміністрування сервера.

У першому розділі проведено аналіз еволюції та сучасного стану програмних засобів для адміністрування серверів. Було досліджено комерційні (cPanel, Plesk) та відкриті (Cockpit, HestiaCP, Portainer) панелі керування, виявлено їхні переваги, недоліки та цільову аудиторію. Цей аналіз дозволив обґрунтувати актуальність розробки та визначити нішу для нового програмного продукту – інтегрованої, легкої системи, що поєднує функції моніторингу ресурсів та управління Docker-контейнерами, уникаючи надлишковості повнофункціональних панелей та фрагментації спеціалізованих інструментів.

У другому розділі було обґрунтовано вибір апаратного середовища, зокрема використання Ubuntu Server LTS як операційної системи, та стеку технологій для програмної реалізації. Для серверної частини (backend) обрано фреймворк FastAPI на мові Python, бібліотеки psutil для моніторингу системних ресурсів, docker-py для взаємодії з Docker API та PostgreSQL як систему управління базами даних. Для клієнтської частини (frontend) було обрано бібліотеку React з інструментарієм Vite та Tailwind CSS для стилізації. Також було сформульовано детальні функційні вимоги, що охоплюють моніторинг CPU, RAM, дискового простору, мережевої активності та управління Docker-контейнерами (перегляд, запуск, зупинка, видалення, доступ до логів), і нефункційні вимоги, такі як продуктивність (відгук API < 500 мс), надійність, безпека (JWT-авторизація, HTTPS) та сумісність.

У третьому розділі детально описано процес проєктування та програмної реалізації розробленої системи. Було представлено архітектуру програмного

засобу, що складається з клієнтського, серверного та системного рівнів. Реалізовано серверну частину на FastAPI, яка забезпечує REST API для авторизації користувачів за допомогою JWT-токенів, збору системних метрик через psutil та управління Docker-контейнерами за допомогою docker-py. Розроблено клієнтську частину на React, яка надає користувачу інтерактивний веб-інтерфейс для візуалізації даних моніторингу (стан системи, сховище, мережа) та адміністрування Docker-контейнерів (створення, перегляд, управління) у реальному часі. Для спрощення розгортання та налаштування всіх компонентів системи на сервері створено універсальний інсталяційний bash-скрипт, що підтримує різні дистрибутиви Linux.

В результаті виконання кваліфікаційної роботи отримано готовий до використання програмний продукт, що відповідає поставленим функційним та нефункційним вимогам. Розроблена система успішно реалізує ключовий функціонал, є легкою у розгортанні та адмініструванні, що робить її доступною для широкого кола користувачів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. GeeksforGeeks. What is Command Line Interface (CLI)?. GeeksforGeeks.org. 18.03.2024. URL: https://www.geeksforgeeks.org/what-is-command-line-interface-cli/ (дата звернення: 03.06.2025).
2. Webmin Community. Documentation | Webmin. Webmin.com. URL: https://webmin.com/docs/ (дата звернення: 03.06.2025).
3. Softlist.io Team. 8 Key Plesk vs. cPanel Comparisons on Finding a Hosting Platform. Softlist.io. 13.02.2025. URL: https://www.softlist.io/plesk-vs-cpanel-comparisons-on-hosting-platform/ (дата звернення: 03.06.2025).
4. ServerGuy Team. Plesk vs cPanel: Which Control Panel is better? (2023 Update). ServerGuy.com. URL: https://serverguy.com/plesk-vs-cpanel/ (дата звернення: 03.06.2025).
5. cPanel: The Hosting Platform for Website Owners. URL: https://cpanel.net/ (дата звернення: 25.05.2025).
6. Cross Site Request Forgery (CSRF). URL: https://owasp.org/www-community/attacks/csrf (дата звернення: 02.06.2025).
7. What Is Role-Based Access Control (RBAC)? A Complete Guide. URL: https://frontegg.com/guides/rbac (дата звернення: 02.06.2025).
8. OpenWrt Project. LuCI - OpenWrt Configuration Interface. GitHub. URL: https://github.com/openwrt/luci (дата звернення: 03.06.2025).
9. Mozilla Developer Network. Introduction to the server side. MDN Web Docs. 11.04.2025. URL: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Introduction (дата звернення: 03.06.2025).
10. Plesk Team. Web Server Security - Beginner's Guide. Plesk.com. 03.01.2024. URL: https://www.plesk.com/blog/various/web-server-security-beginners-guide/ (дата звернення: 03.06.2025).
11. Trio Developers. 10 Server Security Best Practices Every Admin Should Follow. Trio.so. 23.12.2024. URL: https://www.trio.so/blog/server-security-best-practices (дата звернення: 03.06.2025).

12. CloudPanel Team. Choosing the Right Server Administration Tools for 2025. CloudPanel.io. URL: https://www.cloudpanel.io/blog/server-administration-tools/ (дата звернення: 03.06.2025).

13. Storware Team. What Is Server Management? Tools And Best Practices. Storware.eu. URL: https://storware.eu/blog/what-is-server-management-tools-and-best-practices/ (дата звернення: 03.06.2025).

14. Plesk Team. Hosting Control Panels of 2024 – The Definitive Guide. Plesk.com. 01.12.2023. URL: https://www.plesk.com/blog/business-industry/hosting-control-panels-definitive-guide/ (дата звернення: 03.06.2025).

15. CloudPanel Team. Comparison of Server Control Panels: 7 Key Features & Pricing Plans. CloudPanel.io. URL: https://www.cloudpanel.io/blog/comparison-of-server-control-panels/ (дата звернення: 03.06.2025).

16. Plesk Obsidian : Documentation. URL: https://docs.plesk.com/en-US/obsidian/ (дата звернення: 26.05.2025).

17. Plesk: The Leading WebOps and Hosting Platform. URL: https://www.plesk.com/ (дата звернення: 25.05.2025).

18. Usmani H. I tried Cockpit to manage my Raspberry Pi and regret not using it sooner. XDA Developers. 28.05.2025. URL: https://www.xda-developers.com/cockpit-to-manage-raspberry-pi/ (дата звернення: 03.06.2025).

19. Cockpit Project. URL: https://cockpit-project.org/ (дата звернення: 26.05.2025).

20. Lerner D. An introduction to Cockpit, a browser-based administration tool for Linux. Red Hat Enable Sysadmin. 14.04.2020. URL: https://www.redhat.com/en/blog/intro-cockpit (дата звернення: 03.06.2025).

21. Lerner D. An introduction to Cockpit, a browser-based administration tool for Linux. Red Hat Enable Sysadmin. 14.04.2020. URL: https://www.redhat.com/en/blog/intro-cockpit (дата звернення: 03.06.2025).

22. Ahwan A. A. Meet Hestia Control Panel - Open Source cPanel Alternative. Abdulazizahwan.com. 12.02.2024. URL: https://www.abdulazizahwan.com/2024/02/

meet-hestia-control-panel-open-source-cpanel-alternative.html (дата звернення: 03.06.2025).

23. WPJohnny. HestiaCP – server control panel review. WPJohnny.com. 08.09.2020. URL: https://wpjohnny.com/hestiacp-control-panel-review/ (дата звернення: 03.06.2025).

24. Hestia Control Panel : репозиторій програмного забезпечення. GitHub. URL: https://github.com/hestiacp/hestiacp (дата звернення: 02.06.2025).

25. Portainer.io. Portainer Documentation: Welcome. Portainer.io. URL: https://docs.portainer.io/ (дата звернення: 03.06.2025).

26. Portainer Team. portainer/portainer: Making Docker and Kubernetes management easy. GitHub. URL: https://github.com/portainer/portainer (дата звернення: 03.06.2025).

27. Express Computer Systems. Cisco UCS C220 M4 8x 2.5" SFF Chassis : сторінка товару. URL: https://www.expresscomputersystems.com/products/ucsc-c220-m4s (дата звернення: 26.05.2025).

28. Настільний міні-ПК HP ProDesk 400 G5 : сторінка підтримки товару. URL: https://support.hp.com/ua-uk/product/details/hp-prodesk-400-g5-desktop-mini-pc/27780465 (дата звернення: 26.05.2025).

29. Docker, Inc. What is a Container?. Docker.com. URL: https://www.docker.com/what-docker (дата звернення: 23.05.2025).

30. Xiang J., Chen L. A method of docker container forensics based on api. Proceedings of the 2nd International Conference on Cryptography, Security and Privacy, Auckland, New Zealand, Mar. 2018. New York, 2018. P. 159–164.

31. Wu Y. et al. Network user traffic injection based on docker container technology. IEEE 6th International Conference on Computer and Communication Systems (*ICCCS*), Chengdu, China, Apr. 2021. Chengdu, 2021. P. 759–763.

32. Docker In a DevOps Environment. Techso. URL: https://techso.ca/en/docker-introduction/ (дата звернення: 26.05.2025).

33. Вівчарик В. Повний огляд REST: нюанси, поради, приклади. DOU.ua. URL: https://dou.ua/forums/topic/50364/ (дата звернення: 23.05.2025).

34. JSON Web Tokens : офіційний веб-сайт. URL: https://jwt.io/ (дата звернення: 26.05.2025).

35. Rich A. JWT claims. Stytch Blog. 04.10.2023. URL: https://stytch.com/blog/jwt-claims/ (дата звернення: 26.05.2025).

36. Tailwind Labs Inc. Tailwind CSS - Rapidly build modern websites without ever leaving ... : офіційний веб-сайт. URL: https://tailwindcss.com/ (дата звернення: 26.05.2025).

37. Vite Team. Why Vite. Vitejs.dev. URL: https://vitejs.dev/guide/why.html (дата звернення: 03.06.2025).

38. LogRocket Blog. Vite vs. Create React App: Which is better for React?. LogRocket Blog. 15.05.2024. URL: https://blog.logrocket.com/vite-vs-create-react-app/ (дата звернення: 03.06.2025).

39. Python Software Foundation. Welcome to Python.org : офіційний веб-сайт. URL: https://www.python.org/ (дата звернення: 26.05.2025).

40. Poetry - Python dependency management and packaging made easy : офіційний веб-сайт проекту. URL: https://python-poetry.org/ (дата звернення: 26.05.2025).

41. Cursor - The AI Code Editor : офіційний веб-сайт. URL: https://www.cursor.com/ (дата звернення: 26.05.2025).

42. Мартиненко Я. Чому я обираю FastAPI: основні можливості та переваги фреймворку : обговорення на форумі. DOU.ua. URL: https://dou.ua/forums/topic/37547/ (дата звернення: 23.05.2025).

43. Ramírez S. FastAPI : офіційний веб-сайт. URL: https://fastapi.tiangolo.com/ (дата звернення: 23.05.2025).

44. Real Python. Python FastAPI Tutorial – How to Build Lightning Fast APIs. Realpython.com. URL: https://realpython.com/fastapi-python-web-apis/ (дата звернення: 03.06.2025).

45. Docker Inc. Docker Engine API client for Python. Readthedocs.io. URL: https://docker-py.readthedocs.io/en/stable/ (дата звернення: 03.06.2025).

46. Baeldung. Managing Docker Containers With Python. Baeldung.com. 20.05.2024. URL: https://www.baeldung.com/python-docker-container-management (дата звернення: 03.06.2025).

47. GeeksforGeeks. How to Use Docker SDK for Python?. GeeksforGeeks.org. 14.12.2023. URL: https://www.geeksforgeeks.org/how-to-use-docker-sdk-for-python/ (дата звернення: 03.06.2025).

48. Rodola G. Psutil documentation. Psutil. URL: https://psutil.readthedocs.io/en/latest (дата звернення: 23.05.2025).

49. GeeksforGeeks. psutil module in Python. GeeksforGeeks.org. URL: https://www.geeksforgeeks.org/psutil-module-in-python/ (дата звернення: 03.06.2025).

50. Python Awesome. psutil - Cross-platform lib for process and system monitoring in Python. Awesomeopensource.com. URL: https://awesomeopensource.com/project/giampaolo/psutil (дата звернення: 03.06.2025).

51. The PostgreSQL Global Development Group. PostgreSQL: Documentation. Postgresql.org. URL: https://www.postgresql.org/docs/ (дата звернення: 03.06.2025).

52. GeeksforGeeks. Introduction to PostgreSQL. GeeksforGeeks.org. 08.05.2024. URL: https://www.geeksforgeeks.org/introduction-to-postgresql/ (дата звернення: 03.06.2025).

53. PostgreSQL Tutorial. PostgreSQL Tutorial. Postgresqltutorial.com. URL: https://www.postgresqltutorial.com/ (дата звернення: 03.06.2025).

54. Recharts : офіційний веб-сайт. URL: https://recharts.org/en-US (дата звернення: 28.05.2025).

55. Ellis J. Command Line Interface (CLI). URL: https://www.comms-express.com/infozone/article/command-line-interface/ (дата звернення: 03.06.2025).

56. Webmin. URL: https://www.wikiwand.com/ru/articles/Webmin (дата звернення: 03.06.2025).

57. What Is CPanel Hosting. URL: https://www.domainregistrationdns.com.au/article/What-is-cPanel-hosting (дата звернення: 03.06.2025).

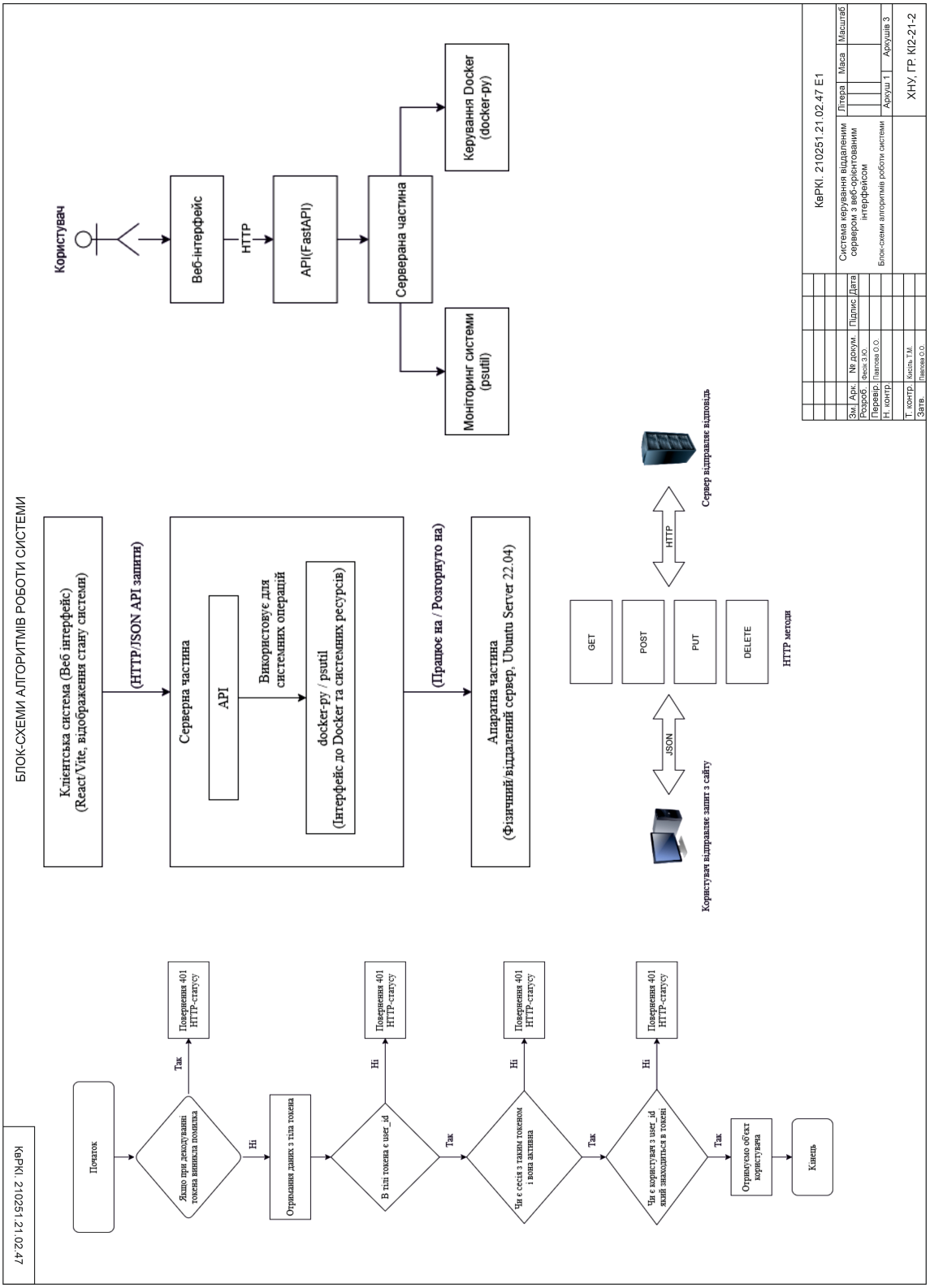
58. Getting Started with OpenWRT. Seeed Studio Wiki. URL: https://wiki.seeedstudio.com/OpenWrt-Getting-Started/ (дата звернення: 26.05.2025).

59. Plesk for IT Admins and WebOps. URL: https://www.plesk.com/it-admins/ (дата звернення: 26.05.2025).

Додаток А

(обов’язковий)

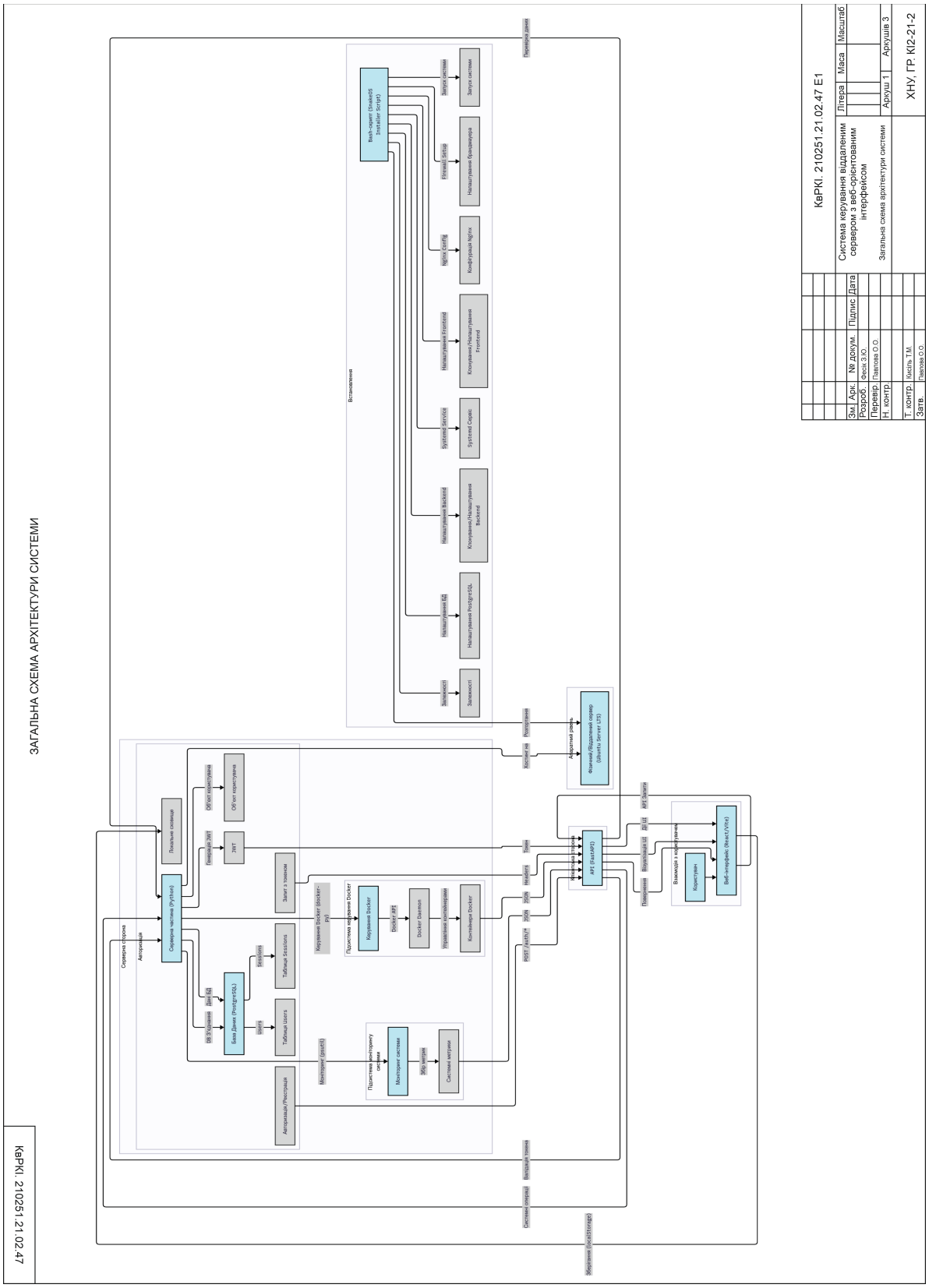
КОПІЯ КРЕСЛЕННЯ «БЛОК-СХЕМИ АЛГОРИТМІВ РОБОТИ СИСТЕМИ»



Додаток Б

(обов’язковий)

КОПІЯ КРЕСЛЕННЯ «ЗАГАЛЬНА СХЕМА АРХІТЕКТУРИ СИСТЕМИ»



Додаток В

(обов'язковий)

КОПІЯ КРЕСЛЕННЯ «ВИГЛЯД РОЗРОБЛЕНОЇ СИСТЕМИ КЕРУВАННЯ ВІДДАЛЕНИМ СЕРВЕРОМ»

Додаток Г

(не обов’язковий)

Структура серверної частини проекту

```
└── SnakeOS-backend/
    ├── .dockerignore
    ├── .env
    ├── .gitignore
    ├── .python-version
    ├── compose.yaml
    ├── Dockerfile
    ├── LICENSE
    ├── pyproject.toml
    ├── README.Docker.md
    ├── README.md
    ├── uv.lock
    └── app/
        ├── connections.py
        ├── main.py
        ├── settings.py
        ├── __init__.py
        ├── auth/
        │   ├── router.py
        │   ├── schemas.py
        │   ├── utils.py
        │   └── __init__.py
        ├── docker/
        │   ├── router.py
        │   ├── schemas.py
```

```
│   ├── __init__.py
│   └── clients/
│       ├── docker.py
│       └── __init__.py
├── models/
│   ├── base.py
│   ├── user.py
│   └── __init__.py
└── system/
    ├── router.py
    ├── schemas.py
    ├── __init__.py
    └── utils/
        ├── system_monitor.py
└── __init__.py
```

Додаток Д

(не обов’язковий)

Структура клієнтської частини проекту

```
└── os-project/
    ├── .env
    ├── .gitignore
    ├── eslint.config.js
    ├── index.html
    ├── package-lock.json
    ├── package.json
    ├── README.md
    ├── vite.config.js
    └── src/
        ├── App.css
        ├── App.jsx
        ├── index.css
        ├── main.jsx
        ├── api/
        │   ├── apiClient.js
        │   ├── config.js
        │   ├── index.js
        │   └── services/
        │       ├── authService.js
        │       ├── dockerService.js
        │       └── systemService.js
        ├── styles/
        │   └── fonts.css
        ├── utils/
```

```
│   └── utils.js
├── assets/
│   ├── fonts/
│   │   ├── OpenSans-Regular.ttf
│   │   └── OpenSans-SemiBold.ttf
│   └── images/
│       ├── App_Store.png
│       ├── background.png
│       ├── crafty.png
│       ├── disk.svg
│       ├── docker-logo.webp
│       └── icon login.svg
├── components/
│   ├── AddSection/
│   │   └── AddSection.jsx
│   ├── Container/
│   │   ├── Container.jsx
│   │   └── container.module.css
│   ├── CustomSelect/
│   │   └── CustomSelect.jsx
│   ├── DateTime/
│   │   ├── datatime.module.css
│   │   └── DateTime.jsx
│   ├── Header/
│   │   ├── Header.jsx
│   │   └── header.module.css
│   ├── NetworkStatus/
│   │   ├── NetworkStatus.jsx
│   │   ├── networkStatus.module.css
```

```
│   │   └── components/
│   │       └── NetworkSelect.jsx
│   ├── Programs/
│   │   ├── Programs.jsx
│   │   ├── programs.module.css
│   │   └── Program/
│   │       ├── Program.jsx
│   │       └── program.module.css
│   ├── SearchBar/
│   │   ├── SearchBar.jsx
│   │   └── searchBar.module.css
│   ├── Slider/
│   │   └── CustomSlider.jsx
│   ├── Storage/
│   │   ├── Storage.jsx
│   │   ├── storage.module.css
│   │   └── components/
│   │       └── ProgressBar/
│   │           └── ProgressBar.jsx
│   ├── SystemStatus/
│   │   ├── SystemStatus.jsx
│   │   ├── systemStatus.module.css
│   │   └── components/
│   │       ├── DropDown/
│   │       │   ├── DropDown.jsx
│   │       │   ├── dropDown.module.css
│   │       │   └── components/
│   │       │       └── Process/
│   │       │           ├── Process.jsx
```

```
│   │   │       └── process.module.css
│   │   └── GaugeBar/
│   │       ├── GaugeBar.jsx
│   │       └── gaugeBar.module.css
│   ├── icons/
│   │   ├── Account/
│   │   │   └── AccountIcon.jsx
│   │   ├── ArrowDown/
│   │   │   └── ArrowDown.jsx
│   │   ├── Disc/
│   │   │   └── DiscIcon.jsx
│   │   ├── Disk/
│   │   │   └── DiscIcon.jsx
│   │   ├── Plus/
│   │   │   └── PlusIcon.jsx
│   │   ├── Search/
│   │   │   └── SearchIcon.jsx
│   │   ├── Settings/
│   │   │   └── SettingsIcon.jsx
│   │   ├── SettingsHeader/
│   │   │   └── SettingsHeaderIcon.jsx
│   │   └── Storage/
│   │       └── StorageIcon.jsx
│   ├── Menus/
│   │   ├── AccountMenu/
│   │   │   └── AccountMenu.jsx
│   │   ├── ProgramMenu/
│   │   │   └── ProgramMenu.jsx
│   │   └── SettingsMenu/
```

```
│   │       └── SettingsMenu.jsx
│   └── Modals/
│       ├── DiscModal/
│       │   ├── DiscModal.jsx
│       │   └── discModal.module.css
│       └── ProgramSettingsModal/
│           └── ProgramSettingsModal.jsx
└── pages/
    ├── Home/
    │   ├── Home.jsx
    │   └── home.module.css
    └── Login/
        ├── Login.jsx
└── login.module.css
```

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Захар ФЕСІК

Співавтор:

Назва: Фесік_Система керування віддаленим сервером з веб-орієнтованим інтерфейсом

Експерт:

Підрозділ: Кафедра комп'ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:1.4%

Коефіцієнт подібності 2:0%

Мікропробіли: 23

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2025-06-06 04:53:28.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2025-06-06 Доцент Андрій Нічепорук

Дата експерт

Anti-Plagiarism (UA) v-15.281 Educational

The maximum coincidence with one document 1.0%

Dictionaries check: en_US, ru_RU, ua_UA. **Errors in the documents: 15%**

	Document		Sum coincidence on the DB	
ID: 243767 Title: БКР Система керування віддаленим сервером з веб-орієнтованим інтерфейсом Added in a DB: 2025-06-05 Authors: Захар ФЕСІК Heads: Ольга ПАВЛОВА Consultants: Opponents:	Symbols	Lexemes	Symbols	Lexemes
	109019	783	2110 (2%)	28 (4%)

Plagiarism sources

ID	Description	Plagiarism presence in the document	
		Symbols	Lexemes

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Фесік Захар Юрійович

Тема: Система керування віддаленим сервером з веб-орієнтованим інтерфейсом

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3 Кількість сторінок записки 73

1. Короткий зміст роботи та прийнятих рішень: Метою кваліфікаційної роботи є розробка системи керування віддаленим сервером з веб-орієнтованим інтерфейсом.

2. Висновок про відповідність роботи дипломному завданню: Робота повністю відповідає поставленому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: В першому розділі кваліфікаційної роботи проведено дослідження предметної області (проаналізовано системи керування віддаленим сервером) та виконано постановку задачі дослідження. В другому розділі кваліфікаційної роботи проведено вибір компонентів та апаратно-програмне середовище для виконання завдання. В третьому розділі кваліфікаційної роботи виконано апаратну та програмну реалізацію системи керування віддаленим сервером з веб-орієнтованим інтерфейсом.

4. Позитивні сторони роботи: висока практична цінність роботи.

5. Негативні сторони роботи:

6.Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно діючих стандартів оформлення документації.

7. Відгук про роботу в цілому: Робота виконана на високому технічному рівні.

8. Інші зауваження: _____

9. Оцінка дипломної роботи: відмінно

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) Бердник Леонід Петрович, д.ф-м.наук, професор, зав. каф. ІПЗ

"9" 06 2025 р. _____(підпис)

Завідувачу кафедри КІІС
д-р. філософії, доц. Ользі ПАВЛОВІЙ

Захара ФЕСІКА
ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ2-21-2

ЗАЯВА

З правилами чинного Положення «Про систему забезпечення академічної доброчесності у Хмельницькому національному університеті» від 01.07.2022, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений(а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Strike-Plagiarism та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

9.06. 2025 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ
КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Система керування віддаленим сервером з веб-орієнтованим інтерфейсом

Автор: Захар ФЕСІК

Спеціальність: 123 – Комп'ютерна інженерія

Освітня програма: освітньо-професійна

Науковий керівник: Ольга ПАВЛОВА, д.ф., доцент

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) запозичення розміщені в розділах аналізу існуючих аналогів та прототипів, які не описують безпосередньо авторське дослідження і не стосуються результатів роботи;
2) усі запозичення фрагментарні, або мають належним чином оформленні посилання;

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 1.4% і адресується до 12 першоджерела; та системою Anti-Plagiarism складає 0%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Керівник роботи _____ Ольга ПАВЛОВА

Гарант ОП _____ Андрій Нічепорук

Завідувач кафедри КІІС _____ Ольга ПАВЛОВА