НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри
комп'ютерних наук
_____/Голуб Б.Л., доц., к.т.н. /
підпис ПІБ, вчене звання і ступінь

«___»_____2025 р

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

Програмне забезпечення системи моніторингу серверних застосунків

Спеціальність 121 «Інженерія програмного забезпечення»

Гарант освітньої програми
_____к.т.н., доцент_____/ Вайганг Г.О/
Науковий ступень та вчене звання підпис ПІБ

Керівник бакалаврської кваліфікаційної роботи :_____/ Дудник А.О. /
підпис ПІБ

Виконав:_____/ Березовський Д. В./
підпис ПІБ

КИЇВ — 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ

Завідувач кафедри
комп'ютерних наук

_____ / Голуб Б.Л., доцент, к.т.н /
підпис

" " 202 р.

ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи

студенту Березовському Денису Васильовичу

Спеціальність 121 «Інженерія програмного забезпечення»

1. Тема роботи: Програмне забезпечення системи моніторингу серверних застосунків.

Затверджена наказом ректора НУБіП України № 2249 "С" від 16.12.2024

2. Термін подання завершеної роботи на кафедру 2025 . 06 . 02
рік, місяць, число

3. Вихідні дані до роботи: результати аналізу літератури, інтернет джерел

4. Перелік питань що розглядаються:

1. Системний аналіз предметної області в контексті програмного забезпечення системи моніторингу серверних застосунків.
2. Проєктування інформаційного та програмного забезпечення системи моніторингу серверних застосунків.
3. Розробка інформаційного та програмного забезпечення системи моніторингу серверних застосунків.
4. Тестування та експлуатація системи моніторингу серверних застосунків.

Керівник бакалаврської кваліфікаційної роботи _____ / Дудник А.О. /
підпис ініціали та прізвище

Завдання прийняла до виконання _____ / Березовський Д. В. /
підпис ініціали та прізвище

Дата отримання завдання 2024 . 12 . 16
рік, місяць, числ

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API — Application Programming Interface, інтерфейс програмування додатків для взаємод tableії між фронтендом, бекендом і агентом системи.

CPU — Central Processing Unit, центральний процесор, метрики використання якого відстежуються системою (навантаження в %).

RAM — Random Access Memory, оперативна пам’ять, метрики використання якої моніторяться системою (в МБ).

Disk — Disk Storage, дисковий простір, метрики якого включають використаний і загальний обсяг (в ГБ).

Network — Network Metrics, мережеві метрики, включають швидкість прийому (RX) і передачі (TX) даних (в Мбіт/с).

JWT — JSON Web Token, токен для аутентифікації та авторизації користувачів, включає access-токен (15 хвилин) і refresh-токен (7 днів).

REST — Representational State Transfer, архітектурний стиль для API, який використовується для обміну даними між компонентами системи.

Vue.js — Vue JavaScript, фреймворк для створення реактивного веб-інтерфейсу фронтенду системи.

Fastify — Fastify Framework, фреймворк для створення бекенду та агента системи, який забезпечує швидку обробку запитів.

SQLite — SQLite Database, легка реляційна база даних для зберігання метрик, логів, конфігурацій і даних користувачів.

Prisma — Prisma ORM, інструмент для роботи з базою даних SQLite, що забезпечує захист від SQL-ін’єкцій.

PM2 — Process Manager 2, інструмент для керування Node.js-процесами, який використовується для запуску, зупинки та моніторингу застосунків.

Nginx — Nginx Web Server, веб-сервер для маршрутизації запитів і балансування навантаження, керований агентом системи.

WebSocket — WebSocket Protocol, протокол для передачі логів у реальному часі через socket.io.

socket.io — Socket.IO Library, бібліотека для реалізації WebSocket-з'єднань між агентом і фронтендом.

Ant Design Vue — Ant Design Vue Library, бібліотека компонентів для створення інтерфейсу користувача на Vue.js.

Plotly — Plotly.js Library, бібліотека для створення графіків метрик (CPU, RAM) у веб-інтерфейсі.

xterm — XTerm.js Library, бібліотека для реалізації терміналу в веб-інтерфейсі для відображення логів.

bcrypt — Bcrypt Library, бібліотека для шифрування паролів користувачів у системі.

CORS — Cross-Origin Resource Sharing, механізм для обробки міждомених запитів у Fastify.

HTTPS — HyperText Transfer Protocol Secure, протокол для безпечної передачі даних між фронтендом, бекендом і агентом.

systemd — Systemd Service, система керування службами Linux для запуску бекенду та агента.

ВСТУП

Сучасні інформаційні системи дедалі частіше потребують ефективного моніторингу серверів і застосунків, адже від їхньої стабільної роботи залежить безперебійність бізнес-процесів. Згідно з дослідженням Uptime Institute у 2022 році, 80% менеджерів і операторів дата-центрів повідомили про щонайменше один простій за останні три роки, причому 60% таких інцидентів призводять до втрат щонайменше 100 000 доларів США, а 15% — понад 1 мільйон доларів, через недостатній контроль і моніторинг інфраструктури, зокрема мережевих проблем. Система, яка дозволяє в реальному часі відстежувати метрики, управляти процесами та аналізувати логи, стає незамінним інструментом для адміністраторів.

Мета розробки цього програмного додатку — створити зручну та гнучку систему моніторингу, яка допоможе адміністраторам серверів оперативно реагувати на проблеми та підтримувати їхню стабільну роботу. Такий додаток актуальний, оскільки він зменшує час простоїв, підвищує ефективність адміністрування та забезпечує централізований доступ до даних кількох серверів. Розроблена система дає змогу не лише переглядати метрики й логи, а й дистанційно керувати процесами через PM2 і налаштуваннями Nginx, що відповідає потребам сучасних IT-інфраструктур.

Для розробки використано сучасні методи та технології, які забезпечують продуктивність і масштабованість. Фронтенд створено на Vue.js із бібліотеками Ant Design Vue для компонентів, Plotly для графіків і xterm для терміналу, що забезпечує зручний інтерфейс. Бекенд і агент розроблені на Fastify у середовищі Node.js, що гарантує швидку обробку запитів. Для роботи з базою даних SQLite застосовано Prisma ORM, яка спрощує доступ до даних і захищає від SQL-ін’єкцій. Інтеграція з PM2 і Nginx дозволяє управляти процесами та веб-сервером, а socket.io забезпечує передачу логів через WebSocket у реальному часі.

Безпека реалізована через JWT-токени, а для шифрування паролів використано bcrypt.

Пояснювальна записка складається з 94 сторінок, 15 використаних джерел і 4 додатків. Вона структурована на чотири розділи для детального висвітлення процесу розробки. У першому розділі описано предметну область і проблему моніторингу серверів, а також проаналізовано аналоги, такі як Zabbix і Nagios, виявивши їхні недоліки. Другий розділ присвячено проєктуванню системи, зокрема діаграмам (компонентів, розгортання), API, інтеграційним інтерфейсам і безпеці. Третій розділ охоплює розробку: вибір інструментів (Vue.js, Fastify), алгоритмізацію, програмування модулів, інтерфейс користувача, інтеграцію та налаштування агента. Четвертий розділ містить практичні аспекти: вимоги до апаратного й програмного забезпечення, склад інсталяційного пакету та рекомендації щодо експлуатації. У додатках наведено фізичну модель бази даних агента (Додаток А), повний код модулів (Додаток Б), реалізацію інтерфейсу користувача (Додаток В), тестування допоміжних вікон (Додаток Д). У висновках підсумовано досягнення проєкту та перспективи розвитку.

1. СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ В КОНТЕКСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СИСТЕМИ МОНІТОРИНГУ СЕРВЕРНИХ ЗАСТОСУНКІВ

1.1 Опис предметної області

Системи моніторингу серверних застосунків відіграють ключову роль у забезпеченні стабільної роботи сучасних інформаційних систем. Предметна область цієї курсової роботи охоплює розробку програмного забезпечення, яке дозволяє відстежувати продуктивність серверів, керувати застосунками та аналізувати їхню поведінку в реальному часі.

Сервери є основою будь-якої сучасної IT-інфраструктури, підтримуючи роботу веб-застосунків, баз даних і бізнес-логіки. Проте їхня безперебійна робота залежить від своєчасного виявлення проблем, таких як перевантаження процесора, нестача пам’яті чи збої в програмах. Для вирішення цих завдань створено систему, яка об’єднує моніторинг продуктивності, керування процесами через PM2 і конфігурацію веб-сервера Nginx. Це дозволяє адміністраторам швидко реагувати на проблеми, переглядати логи та аналізувати історію навантаження, що критично важливо для підтримки високої доступності серверів[9][10].

Основна мета системи — надати зручний інструмент для адміністрування серверів через веб-інтерфейс. Фронтенд, побудований на Vue.js, забезпечує інтуїтивно зрозуміле відображення метрик, графіків навантаження та логів у реальному часі. Бекенд на Fastify відповідає за обробку запитів, аутентифікацію користувачів і збереження даних про сервери в базі SQLite. Окремий агент, також реалізований на Fastify, взаємодіє з PM2 для керування застосунками та Nginx

для налаштування веб-сервера. SQLite обрано через його легкість і відсутність потреби в складному серверному налаштуванні, що ідеально підходить для компактних систем моніторингу.

PM2 відіграє важливу роль у керуванні Node.js-застосунками, дозволяючи запускати, зупиняти та перезавантажувати процеси, а також аналізувати їхні логи. Інтеграція з Nginx забезпечує гнучке налаштування маршрутизації запитів і балансування навантаження, що підвищує ефективність роботи серверів. У підсумку, система об’єднує ці інструменти в єдину платформу, де адміністратор може не лише стежити за станом сервера, а й оперативно вносити зміни до конфігурації.

Безпека також займає важливе місце в предметній області. Аутентифікація користувачів через бекенд захищає доступ до системи, а використання SQLite дозволяє безпечно зберігати дані про сервери та їхню продуктивність. Взаємодія між компонентами системи — фронтендом, бекендом і агентом — відбувається через RESTful API, що забезпечує швидкий обмін даними та масштабованість. Це дозволяє системі адаптуватися до різних сценаріїв використання, від невеликих серверів до складніших інфраструктур.

Розроблена система спрямована на спрощення роботи адміністраторів, надаючи єдиний інтерфейс для моніторингу та керування. Вона враховує сучасні потреби в швидкому доступі до інформації, аналізі продуктивності та оперативному реагуванні на проблеми. У порівнянні з аналогами, такими як Zabbix чи Prometheus, запропоноване рішення вирізняється легкістю розгортання та інтеграцією з PM2 і Nginx, що робить його зручним для використання в проєктах із обмеженими ресурсами. Усе це формує предметну область курсової, яка зосереджена на створенні ефективного інструменту для адміністрування серверних застосунків.

1.2. Аналіз вимог до програмної системи

Розробка системи моніторингу серверних застосунків потребує чіткого визначення вимог, щоб забезпечити її функціональність, зручність і надійність. У цьому підрозділі проаналізовано ключові аспекти, які система має виконувати, враховуючи особливості веб-застосунку на Vue.js, бекенду й агента на Fastify, бази даних SQLite, а також інтеграцію з PM2 і Nginx. Основна увага приділена тому, щоб система відповідала потребам адміністраторів, які керують серверами, і забезпечувала ефективний моніторинг та управління.

Система має надавати можливість відстежувати продуктивність серверів у реальному часі. Це включає моніторинг використання процесора, оперативної пам’яті, дискового простору та мережевих ресурсів. Дані про навантаження повинні відображатися у вигляді графіків для зручного аналізу. Крім того, передбачено перегляд історії навантаження, що дозволяє виявляти тенденції та прогнозувати потенційні проблеми. Веб-інтерфейс, побудований на Vue.js, забезпечує швидке оновлення даних і зручне представлення інформації, щоб адміністратори могли оперативно реагувати на зміни.

Керування серверними застосунками через PM2 є ще однією важливою вимогою. Система дозволяє запускати, зупиняти, перезавантажувати процеси, а також переглядати їхні логи. Це спрощує адміністрування Node.js-застосунків, забезпечуючи контроль над їхньою роботою без необхідності використання командного рядка. Інтеграція з PM2 через API агента на Fastify забезпечує швидкий доступ до цих функцій через веб-інтерфейс, що економить час і знижує ймовірність помилок.

Конфігурація Nginx також входить до функціоналу системи. Адміністратори можуть переглядати та змінювати налаштування веб-сервера, наприклад, маршрутизацію запитів чи параметри балансування навантаження. Це дозволяє оптимізувати роботу сервера без прямого редагування конфігураційних файлів. Агент на Fastify відповідає за взаємодію з Nginx,

забезпечуючи безпечне виконання команд і збереження змін. У підсумку, система спрощує управління серверною інфраструктурою.

Безпека є критично важливою вимогою. Бекенд на Fastify відповідає за аутентифікацію та авторизацію користувачів, захищаючи доступ до системи. Використання JWT-токенів забезпечує безпечну передачу даних між фронтендом і бекендом. SQLite, як база даних, зберігає інформацію про користувачів, сервери та метрики, причому дані захищені від несанкціонованого доступу. Система також має запобігати типовим вразливостям, таким як SQL-ін'єкції чи атаки XSS.

Нефункціональні вимоги включають високу продуктивність і масштабованість. Система повинна обробляти запити швидко, навіть при великому обсязі даних від кількох серверів. Fastify обрано через його швидкодію порівняно з іншими фреймворками, а SQLite забезпечує легкість розгортання без втрати ефективності. Інтерфейс має бути інтуїтивно зрозумілим, з мінімальним часом відгуку, щоб адміністратори могли швидко знаходити потрібну інформацію.

Взаємодія між компонентами системи — фронтендом, бекендом і агентом — здійснюється через RESTful API, що забезпечує гнучкість і можливість розширення. Система також повинна бути простою в розгортанні, з мінімальними вимогами до апаратного забезпечення, що робить її доступною для використання на різних серверах. Усе це формує набір вимог, які визначають функціонал і ефективність розробленого програмного забезпечення.

1.3. Моделювання предметної області

Для створення системи моніторингу серверних застосунків використано кілька підходів до моделювання, щоб чітко описати процеси та взаємодії між компонентами. У цьому підрозділі представлено аналіз предметної області через діаграму прецедентів і діаграму послідовності, враховуючи специфіку веб-

застосунку, бекенду й агента, , а також інтеграцію з PM2 і Nginx. Моделі допомагають зрозуміти, як система функціонує та взаємодіє з користувачами.

Діаграма прецедентів (рис. 1) стала першим кроком у моделюванні, адже вона чітко показує, які дії доступні користувачам. На ній виділено два основні актори — оператор управління та оператор моніторингу, які взаємодіють із системою. Оператор управління може запускати застосунки, зупиняти їх, керувати конфігурацією, переглядати статус сервера, застосунків та переглядати логи. Оператор моніторингу відповідає за перегляд статусу сервера, застосунків і перегляд історії навантаження. Ця діаграма допомагає зрозуміти основні сценарії використання системи та визначає її ключові функції.

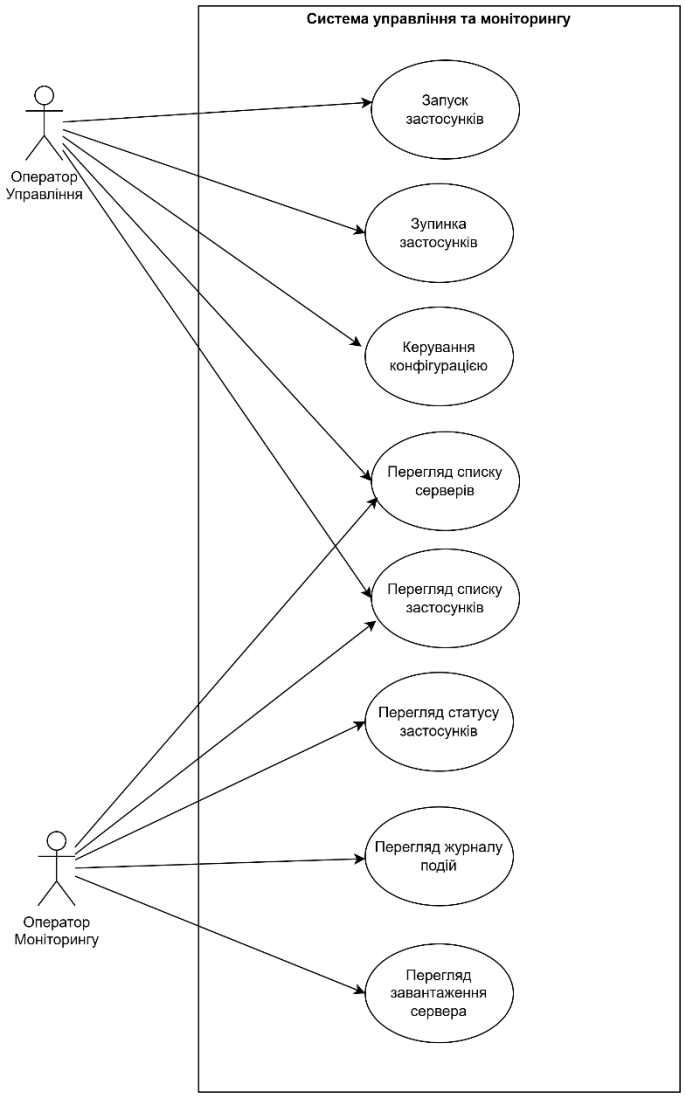


Рис. 1 Діаграма прецедентів

Діаграма послідовності (рис. 2) детально розкриває, як відбувається один із типових сценаріїв — перегляд статусу процесів. Вона показує взаємодію між адміністратором, бекендом і серверним агентом. Спочатку адміністратор через веб-інтерфейс надсилає запит на перегляд статусу процесів. Бекенд обробляє цей запит і передає його серверному агентові. Агент звертається до PM2, щоб отримати актуальні дані про статус процесів, після чого повертає їх серверному агенту. Серверний агент передає отриману інформацію назад у веб-інтерфейс, де вона відображається адміністратору. Якщо потрібно налаштувати нову конфігурацію, агент надсилає команду до Nginx, а потім повертає підтвердження виконання. Ця діаграма наочно ілюструє, як компоненти системи обмінюються даними для виконання запиту.

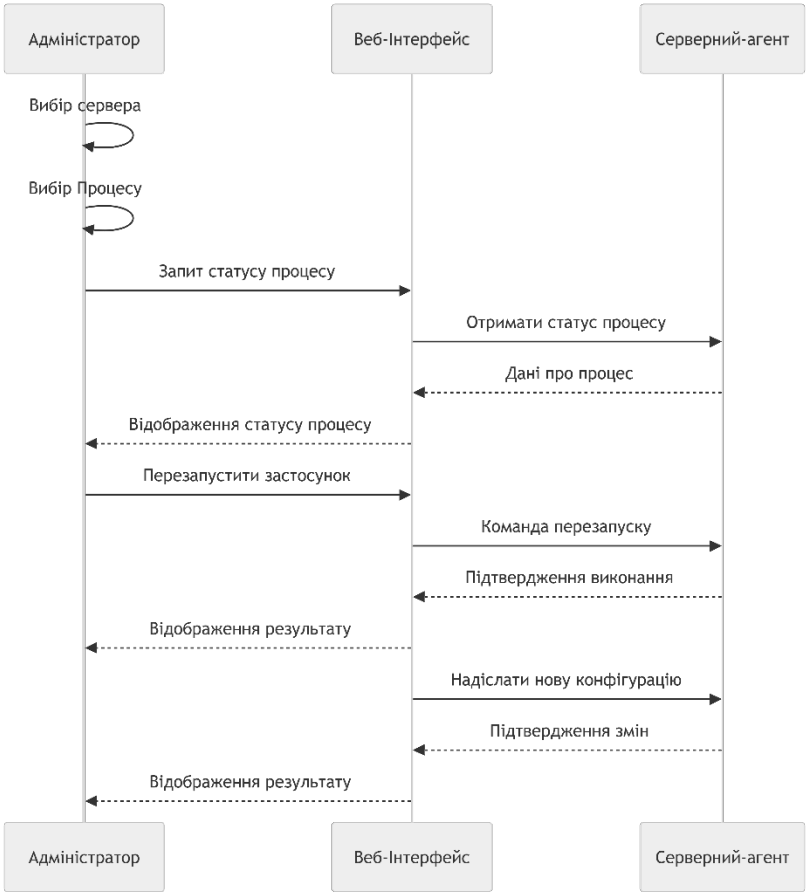


Рис. 2 Діаграма послідовності

У підсумку, моделювання через діаграму прецедентів і діаграму послідовності забезпечує чітке уявлення про функціонал і поведінку системи.

1.4. Огляд інформаційних джерел та існуючих рішень

Для створення системи моніторингу серверних застосунків використано різноманітні інформаційні джерела, які допомогли розібратися в сучасних технологіях і підходах. Основна інформація взята з офіційної документації Vue.js, Fastify, SQLite, PM2 і Nginx.[6] На сайтах розробників цих інструментів знайдено детальні описи їхньої роботи, приклади використання та рекомендації щодо інтеграції. Наприклад, документація Vue.js дала розуміння, як створювати реактивні інтерфейси для відображення графіків у реальному часі, а Fastify надав приклади швидкої обробки API-запитів. SQLite виявився зручним для локального зберігання даних, і його документація допомогла налаштувати базу без зайвих складнощів. PM2 і Nginx також мають детальні посібники, які пояснюють, як керувати процесами та налаштовувати веб-сервер. Крім офіційних джерел, звернення до форумів, таких як Stack Overflow, і статей на Medium дозволило знайти практичні рішення. [14] Там описано, як оптимізувати взаємодію між Fastify і PM2, а також як налаштувати Nginx для балансування навантаження в подібних системах.

Розглянуто кілька популярних рішень для моніторингу серверів, щоб зрозуміти їхні сильні та слабкі сторони. Zabbix — одна з найвідоміших систем, яка підтримує широкий спектр функцій. Вона дозволяє відстежувати продуктивність серверів, аналізувати мережеві ресурси, переглядати логи та навіть створювати сповіщення для адміністраторів. Zabbix має модульну архітектуру, що дає змогу адаптувати його під різні потреби, наприклад, моніторинг великих корпоративних мереж. Але є й недоліки: налаштування Zabbix займає багато часу, адже потрібно створювати окремі шаблони для

кожного типу даних. Інтерфейс здається перевантаженим, що ускладнює швидкий доступ до інформації. Для невеликих проєктів Zabbix виглядає занадто складним, адже його можливості перевищують потреби системи, яка розробляється в цій курсовій.

Prometheus — ще одне потужне рішення, яке спеціалізується на зборі метрик. Воно працює за принципом pull-моделі, коли сервер періодично запитує дані з агента. Prometheus чудово справляється з моніторингом продуктивності, наприклад, використання CPU чи пам’яті, і має гнучкі інструменти для створення запитів через мову PromQL. Проте робота з логами в Prometheus обмежена, адже він більше орієнтований на числові метрики, ніж на текстові дані. Для повноцінного аналізу логів потрібні додаткові інструменти, такі як Loki, що додає складності. До того ж Prometheus потребує інтеграції з Grafana для візуалізації, що збільшує залежність від зовнішніх систем. У порівнянні з ним, розроблена система простіша, адже об’єднує метрики та логи в одному інструменті без додаткових модулів.

Grafana, у свою чергу, славиться можливостями візуалізації. Вона дозволяє створювати інтерактивні дашборди, де можна відображати графіки, таблиці та сповіщення. Grafana підтримує інтеграцію з Prometheus, InfluxDB та іншими джерелами даних, що робить її універсальним інструментом для великих систем. Адміністратори можуть налаштувати дашборди під свої потреби, наприклад, відображати піки навантаження чи аналізувати тренди. Але Grafana не є самостійним рішенням — вона лише візуалізує дані, які збирають інші інструменти. Це означає, що для повноцінної роботи потрібен Prometheus або Zabbix, що ускладнює розгортання. Також Grafana потребує певного часу на освоєння, адже її налаштування вимагає знання джерел даних і створення запитів. Розроблена система виграє за простотою: фронтенд на Vue.js уже готовий до роботи з даними, які надходять від бекенду, без додаткових інструментів.

Nagios — це рішення з довгою історією, яке відоме своєю стабільністю. Воно підтримує базовий моніторинг серверів, перевіряє доступність сервісів і надсилає сповіщення у разі збоїв. Nagios простий у розгортанні та не вимагає складної інфраструктури, що робить його зручним для невеликих проєктів. Проте його можливості обмежені: він не підходить для глибокого аналізу метрик чи роботи з великими обсягами даних. Масштабованість Nagios поступається сучасним системам, адже при збільшенні кількості серверів продуктивність падає. Також інтерфейс виглядає застарілим, що ускладнює роботу з ним у порівнянні з Grafana чи розробленою системою на Vue.js.

Ще одне рішення, яке привернуло увагу, — це Datadog. Воно пропонує комплексний підхід до моніторингу, включаючи метрики, логи та навіть APM (Application Performance Monitoring). Datadog легко інтегрується з хмарними сервісами, такими як AWS чи Azure, і має зручний інтерфейс із готовими дашбордами. Але Datadog — це комерційний продукт, і його використання пов’язане з високими витратами, особливо для великих систем. До того ж для невеликих проєктів його можливості надлишкові, а складність налаштування може відлякувати. Розроблена система, навпаки, безкоштовна у використанні та орієнтована на простоту, що робить її доступнішою для невеликих команд.

Порівняно з цими рішеннями, розроблена система має кілька переваг. Вона легка в розгортанні завдяки SQLite, яке не вимагає складного серверного налаштування, і Fastify, що забезпечує швидку обробку запитів. Інтеграція з PM2 дозволяє зручно керувати Node.js-застосунками, а підтримка Nginx додає гнучкості в налаштуванні веб-сервера. Фронтенд на Vue.js забезпечує інтуїтивно зрозумілий інтерфейс із графіками в реальному часі, що вигідно відрізняє систему від Zabbix чи Nagios. На відміну від Prometheus, розроблена система охоплює як метрики, так і логи, не потребуючи додаткових модулів. Порівняно з Grafana, вона не залежить від зовнішніх джерел даних, адже бекенд на Fastify сам збирає всю інформацію. У порівнянні з Datadog і New Relic, система виграє за доступністю, адже не вимагає платних підписок і складного налаштування.

Проте є й недоліки. Розроблена система менш масштабована, ніж Grafana з Prometheus, і не підходить для великих корпоративних мереж із сотнями серверів. Вона орієнтована на невеликі проєкти, де важливі простота та швидкість розгортання. Також функціонал обмежений порівняно з Datadog чи New Relic, адже немає підтримки хмарних інтеграцій чи аналізу поведінки користувачів. Але для поставлених завдань — моніторингу серверів, керування процесами та аналізу логів — цих можливостей достатньо.

Vue.js обрано за його легкість і реактивність, що ідеально підходить для фронтенду. Fastify перевершив інші фреймворки, як-от Express, за швидкодією, що важливо для бекенду та агента. SQLite забезпечує просте зберігання даних, а PM2 і Nginx додають зручності в адмініструванні. У підсумку, огляд показав, що розроблена система займає нішу між складними корпоративними рішеннями та простими інструментами, пропонуючи оптимальний баланс функціоналу та простоти.

1.5 Постановка завдання

Сучасні IT-інфраструктури потребують оперативного контролю для забезпечення безперебійної роботи, що робить розробку системи моніторингу серверних застосунків актуальним завданням. Завдання полягає в створенні програмного додатку, який дозволить адміністраторам серверів відстежувати метрики, управляти процесами та налаштовувати веб-сервер у реальному часі, щоб мінімізувати простої та підвищити ефективність адміністрування.

Основна мета — розробити систему, що складається з трьох компонентів: фронтенду, бекенду та агента. Фронтенд має забезпечити зручний веб-інтерфейс для відображення метрик CPU, пам’яті, диска й мережі, а також для перегляду логів і керування застосунками через PM2. Бекенд повинен координувати взаємодію між фронтендом і агентами, обробляти автентифікацію користувачів і надавати API для управління серверами та користувачами. Агент, розгорнутий на

кожному сервері, має збирати системні метрики, інтегруватися з PM2 для управління процесами, налаштовувати Nginx і передавати логи в реальному часі через WebSocket.

Функціональні вимоги включають: створення RESTful API для обміну даними між компонентами, зокрема ендпоінти для отримання метрик, управління процесами та створення конфігурацій; забезпечення автентифікації через JWT-токени з access і refresh токенами; збереження даних у SQLite через Prisma ORM; відображення графіків навантаження та логів у фронтенді. Нефункціональні вимоги охоплюють: продуктивність системи для обробки до 100 запитів за секунду, безпеку через захист від SQL-ін’єкцій і XSS-атак, а також сумісність із Linux-серверами (Ubuntu 20.04) та браузерами (Chrome 90+, Firefox 85+).

Для реалізації завдання необхідно спроєктувати архітектуру системи, визначити API та інтеграційні інтерфейси, спроєктувати діаграми, розробити модулі для збору метрик, управління процесами й конфігурації Nginx, а також реалізувати веб-інтерфейс. Додатково потрібно описати вимоги до апаратного й програмного забезпечення, скласти інсталяційний пакет і надати рекомендації щодо експлуатації.

2. ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СИСТЕМИ МОНІТОРИНГУ СЕРВЕРНИХ ЗАСТОСУНКІВ

2.1 Логічна модель даних у вигляді ER-діаграми

Для системи моніторингу серверних застосунків створено логічну модель даних, яка описує структуру бази даних агента та бекенду. Вона представлена у вигляді ER-діаграми, що забезпечує відповідність реляційним принципам і третій нормальній формі. Модель охоплює таблиці для зберігання метрик навантаження, логів, конфігурацій, користувачів, токенів, прав доступу та серверів, а також зв’язки між ними.

База даних агенту (рис. 3) складається з трьох таблиць. Таблиця LoadMetrics призначена для збереження метрик навантаження. Вона містить унікальний ідентифікатор loadMetricId, який є первинним ключем, тип метрики type, значення value і час запису time. Це дозволяє зберігати дані про продуктивність сервера, такі як використання CPU чи пам’яті, у хронологічному порядку. Таблиця Logs зберігає логи застосунків. У ній є ідентифікатор logId як первинний ключ, ідентифікатор процесу processId, тип логу type, повідомлення message і час запису time. Це забезпечує можливість аналізу подій для кожного процесу окремо. Таблиця Configs відповідає за збереження конфігураційних файлів, зокрема для Nginx. Вона включає ідентифікатор configId як первинний ключ, назву файлу fileName, шлях до файлу location і сам вміст content. Така структура дозволяє агенту швидко отримувати та оновлювати конфігурації.

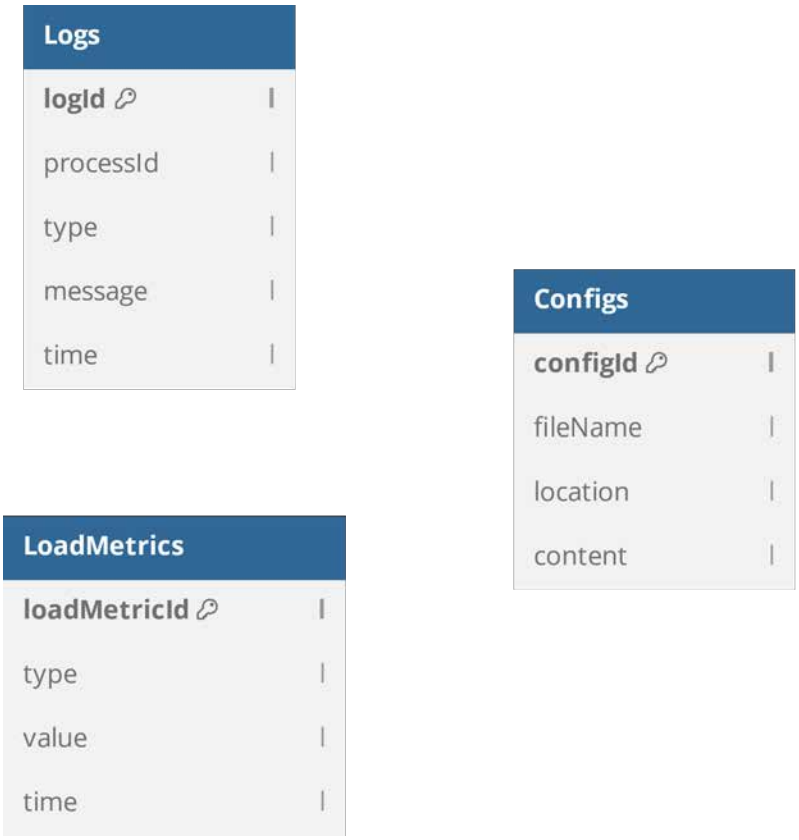


Рис. 3 ER діаграма агенту

База даних бекенду (рис. 4) складається з чотирьох таблиць. Таблиця Users зберігає інформацію про користувачів системи. Вона має ідентифікатор userId як первинний ключ, логін login і пароль password. Це основа для аутентифікації користувачів у системі. Таблиця Tokens відповідає за збереження токенів для безпечного доступу. Вона включає ідентифікатор tokenId як первинний ключ, ідентифікатор користувача userId і сам токен token. Таблиця Permissions визначає права доступу користувачів. У ній є ідентифікатор permissionId як первинний ключ, ідентифікатор користувача userId і набір прав: canView, canEdit, canControl, canDelete, canAdministrate. Це дозволяє гнучко налаштовувати доступ для кожного користувача. Таблиця Servers містить інформацію про сервери, які моніторяться. Вона має ідентифікатор serverId як первинний ключ, назву сервера name і його URL-адресу url.

Зв’язки між таблицями бекенду визначені для забезпечення цілісності даних. Поле userId у таблиці Tokens пов’язане з userId у таблиці Users через відношення "один до багатьох", що означає, що одному користувачу може відповідати кілька токенів. Аналогічно поле userId у таблиці Permissions пов’язане з userId у таблиці Users, дозволяючи одному користувачу мати один набір прав доступу. Ці зв’язки забезпечують реляційність моделі та унеможливлюють появу даних без відповідного користувача.

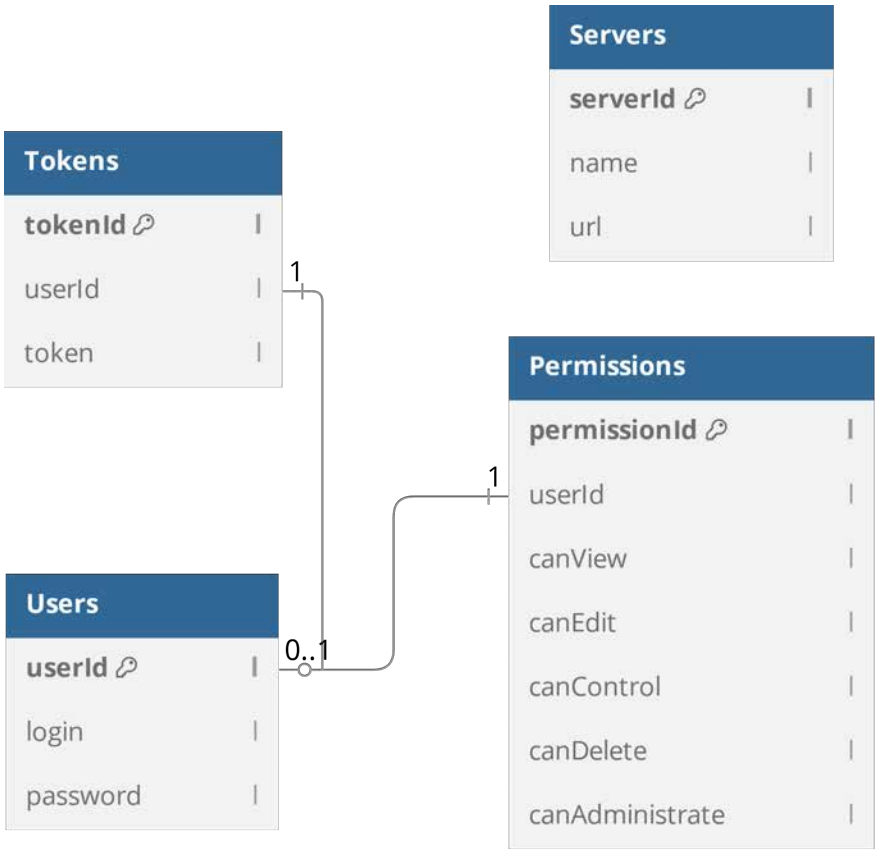


Рис. 4 ER діаграма бекенду

Модель відповідає принципам третьої нормальної форми. Усі таблиці мають первинні ключі, що унеможливлює дублювання записів. Неключові поля, такі як type, value чи message, залежать лише від первинного ключа своєї таблиці, що виключає функціональні залежності між неключовими атрибутами. Наприклад, у таблиці LoadMetrics значення value залежить від loadMetricId, а не від інших полів, таких як time. У таблиці Permissions права доступу canView чи canEdit пов’язані з permissionId і userId, без транзитивних залежностей. Це

забезпечує нормалізацію даних і зменшує ризик аномалій при вставці, оновленні чи видаленні.

Перехід до фізичної моделі буде виконано через SQLite, де кожна таблиця матиме відповідні індекси для полів, що часто використовуються у запитах, наприклад, time у LoadMetrics чи userId у Tokens.[8] У підсумку, ER-діаграма створює основу для ефективного зберігання та обробки даних у системі моніторингу.

2.2 Діаграма пакетів

Для організації архітектури системи моніторингу серверних застосунків створено діаграму пакетів (рис. 5), яка розбиває систему на логічні модулі та визначає їхні взаємодії. Вона відображає структуру веб-застосунку на Vue.js, бекенду та агента на Fastify, а також інтеграцію з базою даних SQLite, PM2 і Nginx. Ця діаграма допомагає зрозуміти, як компоненти розподілені та працюють разом.

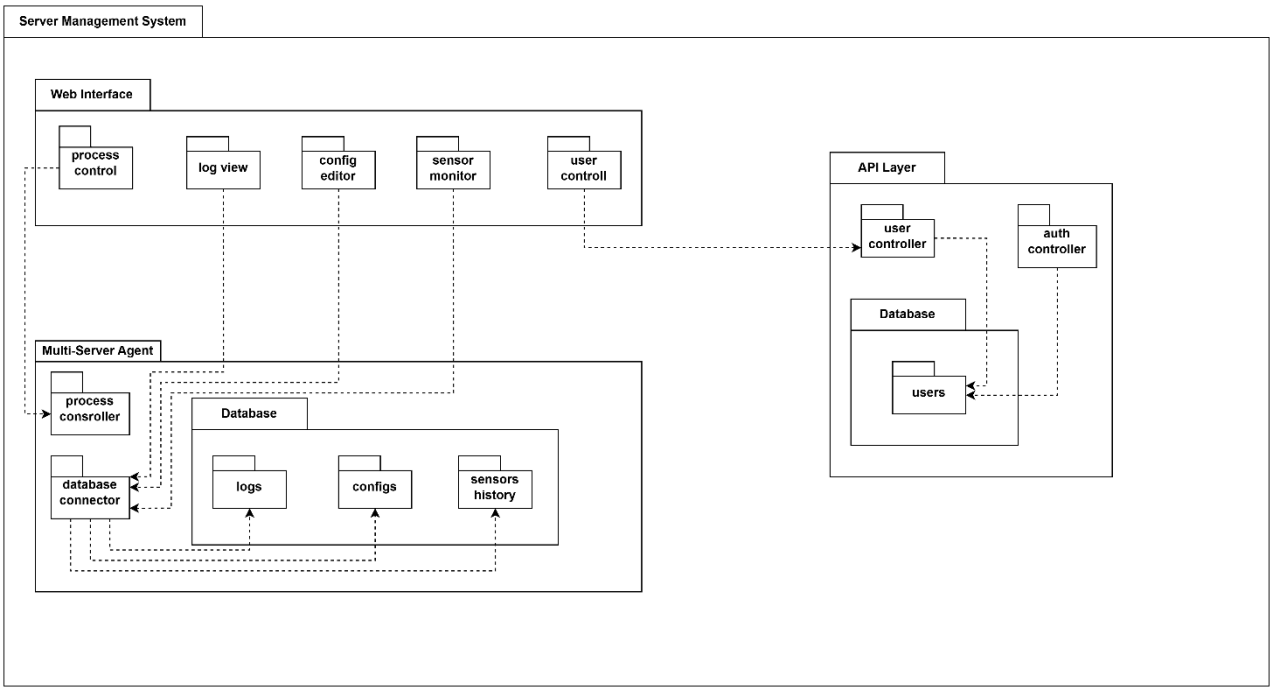


Рис. 5 Діаграма пакетів

Система поділена на кілька основних пакетів. Пакет Web Interface включає модулі, які відповідають за взаємодію з користувачем. До нього входять контролери для управління процесами, перегляду логів, редагування конфігурацій, моніторингу сенсорів і роботи з користувачами. Ці модулі забезпечують інтуїтивно зрозумілий інтерфейс, де адміністратори можуть керувати серверами та аналізувати дані. Пакет Multi-Server Agent об’єднує компоненти агента, який взаємодіє з серверною інфраструктурою. У ньому є контролер процесів, з’єднувач із базою даних і локальні бази даних для логів, конфігурацій і історії сенсорів. Цей пакет відповідає за збір даних із серверів і їхню обробку.

Окремий пакет API Layer відповідає за обробку запитів між фронтендом і бекендом. Він включає контролер користувачів і контролер автентифікації, які забезпечують доступ до системи та захист даних. Цей пакет виступає посередником, передаючи запити від веб-інтерфейсу до бази даних і назад. Пакет Database об’єднує сховище користувачів, яке зберігає інформацію про доступ і автентифікацію. Ця структура дозволяє централізовано управляти даними про користувачів і їхні права.

Взаємодії між пакетами чітко визначені. Web Interface надсилає запити до API Layer, який обробляє їх і передає до відповідних контролерів. Наприклад, запит на перегляд логів іде від модуля log view у Web Interface до API Layer, а звідти до Multi-Server Agent, де контролер процесів звертається до бази даних логів. З’єднувач із базою даних у Multi-Server Agent забезпечує доступ до локальних таблиць, таких як logs і configs. API Layer також взаємодіє з пакетом Database, передаючи дані про користувачів і токени для автентифікації. Ця взаємодія гарантує, що кожен пакет виконує свою роль, не перетинаючись із іншими.

Архітектура відображає розподіл відповідальності. Web Interface зосереджений на поданні інформації, Multi-Server Agent — на зборі й обробці даних із серверів, API Layer — на координації, а Database — на зберіганні

ключової інформації. Такий підхід полегшує підтримку системи та дозволяє додавати нові функції, наприклад, розширення моніторингу чи інтеграцію з іншими серверами. У підсумку, діаграма пакетів створює основу для модульної структури, яка підтримує розробку та подальше вдосконалення системи.

2.3 Діаграма компонентів

Для системи моніторингу серверних застосунків створено діаграму компонентів (рис. 6), яка відображає основні частини системи та їхні взаємодії. Вона показує, як веб-застосунок на Vue.js, бекенд і агент на Fastify, база даних SQLite, а також зовнішні інструменти PM2 і Nginx працюють разом. Діаграма допомагає зрозуміти архітектуру системи та розподіл функціоналу між її складовими.

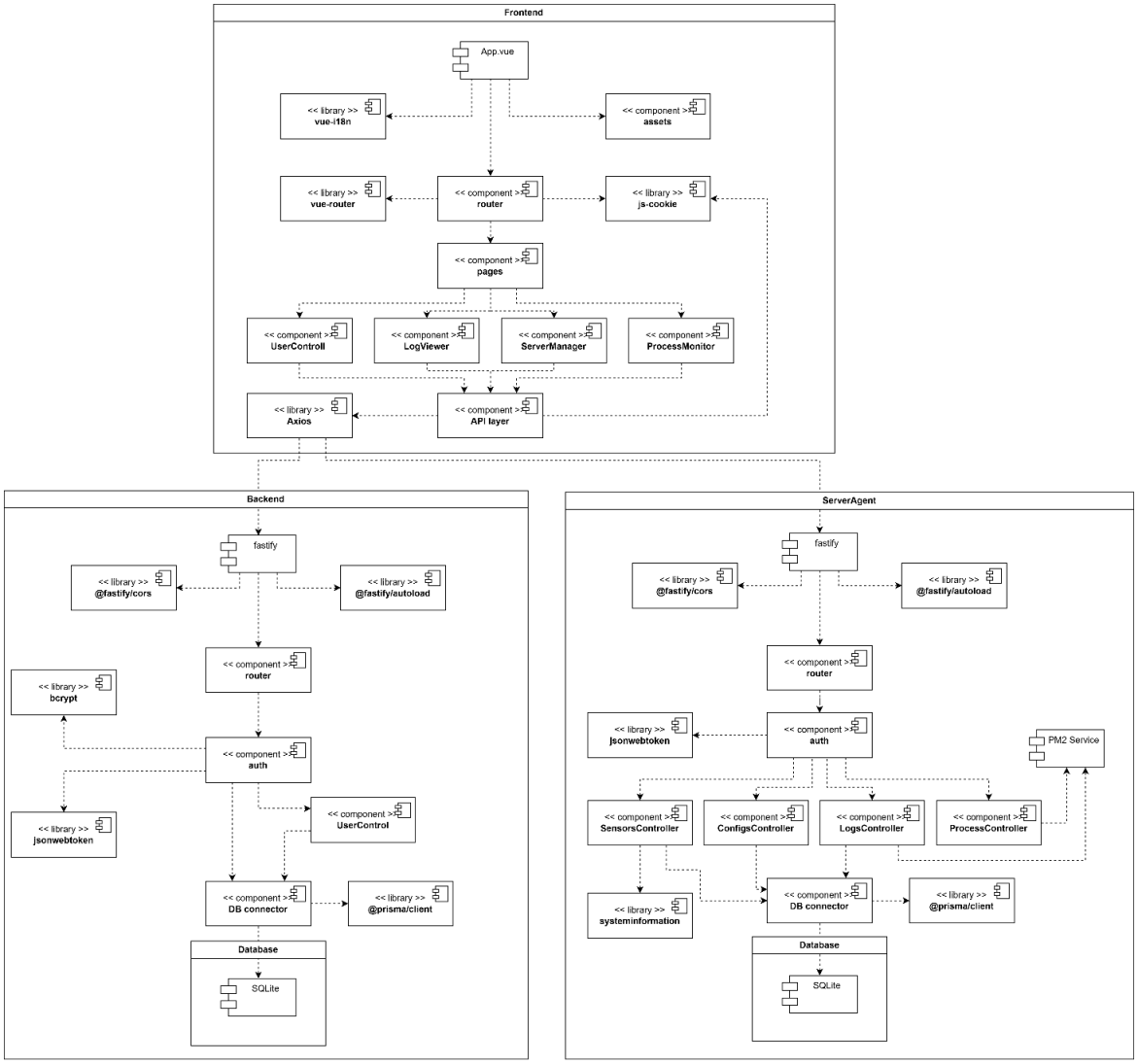


Рис. 6 Діаграма компонентів

Основний компонент — Frontend, який відповідає за інтерфейс користувача. Він реалізований на Vue.js і включає кілька підкомпонентів. Компонент App.vue є точкою входу, яка ініціалізує всю структуру фронтенду. Підкомпоненти vue-i18n і vue-router забезпечують підтримку багатомовності та маршрутизацію між сторінками. Компоненти vue-cookies і js-cookies відповідають за збереження даних у браузері, наприклад, токенів автентифікації. Підкомпонент pages містить сторінки інтерфейсу, які використовуються для відображення даних. Компоненти UserControl, LogViewer, ServerManagement і ProcessMonitor відображають конкретні функції: управління користувачами, перегляд логів, моніторинг серверів і процесів. Компонент Axios забезпечує

відправлення HTTP-запитів до бекенду, дозволяючи фронтенду отримувати дані та надсилати команди.

Другий ключовий компонент — Backend, побудований на Fastify. Він обробляє запити від фронтенду та управляє користувачами. У його складі є підкомпоненти fastify-cors для підтримки CORS, fastify-autoload для автоматичного завантаження плагінів і fastify-jwt для автентифікації через JWT-токени. Компоненти bcrypt і jwt-token відповідають за шифрування паролів і створення токенів. Підкомпоненти UserControl і SensorsController обробляють запити, пов’язані з користувачами та метриками. Компонент prisma-client забезпечує зв’язок із базою даних SQLite через ORM Prisma, а підкомпонент Database містить саме сховище даних SQLite для користувачів і токенів.

Третій компонент — ServerAgent, також побудований на Fastify. Він відповідає за взаємодію з серверною інфраструктурою. У його складі є підкомпоненти fastify-cors і fastify-autoload для аналогічних функцій, як у бекенді. Компонент auth забезпечує автентифікацію агента, а prisma-client і підкомпонент Database зв’язують агента з локальною базою даних SQLite, де зберігаються логи, конфігурації та історія метрик. Компоненти ConfigController, LogsController і ProcessController обробляють запити на налаштування Nginx, перегляд логів і керування процесами через PM2. Зовнішній компонент PM2 Service інтегрується з агентом, дозволяючи запускати, зупиняти та моніторити Node.js-процеси.

Взаємодії між компонентами чітко визначені. Frontend через Axios надсилає запити до Backend, який обробляє їх за допомогою UserControl і SensorsController, звертаючись до бази даних через prisma-client. Backend також взаємодіє з ServerAgent, надсилаючи команди для збору метрик чи керування процесами. ServerAgent, у свою чергу, звертається до PM2 Service для роботи з процесами та до локальної бази даних для збереження логів і конфігурацій. Зміни в конфігураціях передаються до Nginx через ConfigController, що забезпечує оновлення налаштувань веб-сервера.

Ця структура підкреслює модульність системи. Frontend зосереджений на відображенні даних, Backend — на управлінні користувачами та координації, а ServerAgent — на взаємодії з сервером. У підсумку, діаграма компонентів створює основу для ефективної реалізації системи, забезпечуючи чіткий розподіл функцій.

2.4 Діаграма розгортання

Для системи моніторингу серверних застосунків створено діаграму розгортання (рис. 7), яка показує, як компоненти системи розподілені між фізичними пристроями. Вона відображає топологію системи, включаючи клієнтську частину, основний сервер і додаткові сервери, а також взаємодії між ними. Це допомагає зрозуміти, як веб-застосунок на Vue.js, бекенд і агент на Fastify, база даних SQLite, PM2 і Nginx розміщені та працюють разом.

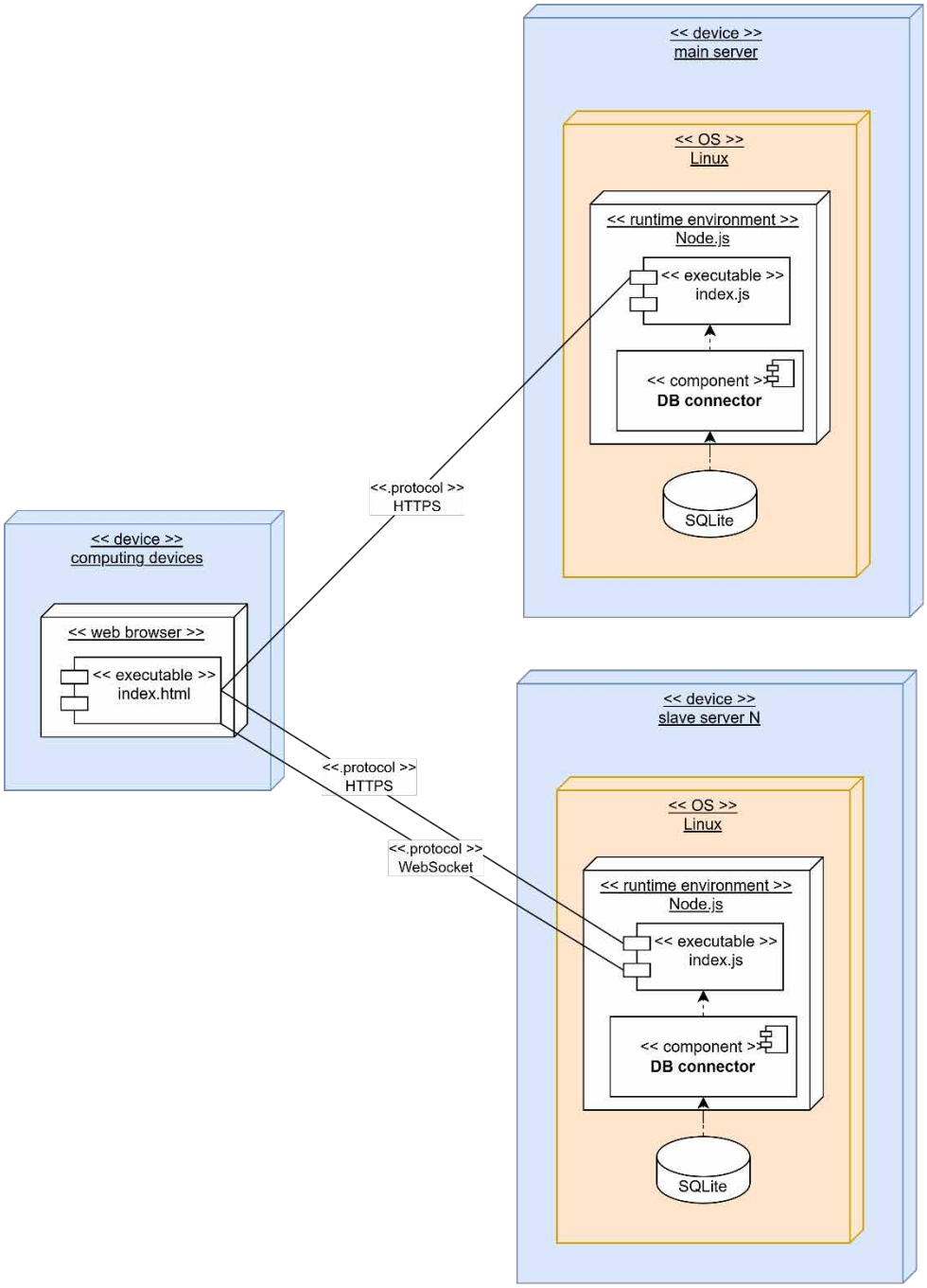


Рис. 7 Діаграма розгортання

Перший вузол — Computing Devices, який представляє клієнтську частину. На ньому розгорнуто веб-браузер із виконавчим файлом index.html. Це точка доступу для користувачів, таких як адміністратори, які взаємодіють із системою через веб-інтерфейс. Браузер надсилає запити до основного сервера через протоколи HTTPS і WebSocket. [12] HTTPS використовується для передачі

стандартних запитів, таких як автентифікація чи перегляд даних, а WebSocket забезпечує зв’язок у реальному часі для оновлення метрик і логів. Цей вузол не потребує додаткових серверних ресурсів, адже вся обробка відбувається на стороні сервера.

Другий вузол — Main Server, який працює під управлінням операційної системи Linux. На ньому розгорнуто основний бекенд системи. Середовище виконання Node.js забезпечує запуск виконавчого файлу index.js, який є точкою входу для бекенду на Fastify. У складі цього вузла є компонент DB Connector, який зв’язує бекенд із базою даних SQLite. SQLite зберігає інформацію про користувачів, токени та права доступу, що дозволяє бекенду обробляти автентифікацію та координувати запити. Main Server взаємодіє з клієнтським вузлом через HTTPS і WebSocket, а також із додатковими серверами через HTTPS для передачі команд і отримання даних.

Третій вузол — Slave Server N, який також працює на Linux. Він представляє додаткові сервери, що моніторяться системою. На цьому вузлі розгорнуто агент, який працює в середовищі Node.js і запускається через виконавчий файл index.js. Агент на Fastify включає свій DB Connector, який зв’язує його з локальною базою даних SQLite. Ця база даних зберігає логи, метрики навантаження та конфігурації для Nginx. Агент відповідає за збір даних із сервера, таких як продуктивність і логи, а також за виконання команд, наприклад, оновлення налаштувань Nginx чи керування процесами через PM2. Slave Server N взаємодіє з Main Server через HTTPS, отримуючи запити на обробку та надсилаючи назад зібрані дані.

Взаємодії між вузлами побудовані так, щоб забезпечити ефективну роботу системи. Computing Devices надсилає запити до Main Server, який обробляє їх і, за потреби, перенаправляє до Slave Server N. Наприклад, запит на перегляд логів іде від браузера до бекенду, а бекенд звертається до агента на Slave Server N, який повертає дані з локальної бази SQLite. Це дозволяє системі гнучко

масштабуватися, додаючи нові сервери за потреби. У підсумку, діаграма розгортання дає чітке уявлення про топологію та розподіл компонентів.

2.5 Проєктування API та інтеграційних інтерфейсів

Для системи моніторингу серверних застосунків спроєктовано API та інтеграційні інтерфейси, які забезпечують зв'язок між фронтендом, бекендом і агентом, а також взаємодію з зовнішніми інструментами, такими як PM2 і Nginx. API реалізовано на Fastify, що дозволяє швидко обробляти запити та забезпечує гнучкість завдяки модульній структурі. Воно підтримує RESTful-підхід, який спрощує взаємодію між компонентами та робить систему масштабованою. Інтеграція з PM2 і Nginx дозволяє агенту керувати процесами та налаштуваннями веб-сервера, а бекенд обробляє автентифікацію та управління користувачами.

API поділено на два основні модулі: Agent і Backend. Agent відповідає за взаємодію агента з серверною інфраструктурою, обробляючи запити на моніторинг, логи та конфігурації. Backend управляє користувачами, серверами та автентифікацією, забезпечуючи доступ до системи через фронтенд. Обидва модулі використовують RESTfull-ендпоінти, які підтримують стандартні методи HTTP, такі як GET, POST, DELETE, а також OPTIONS для обробки CORS-запитів. Fastify обрано за його високу швидкодію порівняно з іншими фреймворками, що критично важливо для реального часу.

Структура ендпоінтів для Agent виглядає так:

- /alive (GET, HEAD) — перевіряє, чи агент доступний і працює.
- /metrics/date (POST) — повертає метрики навантаження за певну дату.
- /metrics/now (GET, HEAD) — повертає поточні метрики в реальному часі.
- /metrics/totalSize (GET, HEAD) — показує загальний розмір збережених метрик.
- /nginx (GET, HEAD) — повертає статус Nginx.
- /nginx/check (GET, HEAD) — перевіряє коректність конфігурації Nginx.

- /nginx/create (POST) — створює нову конфігурацію для Nginx.
- /nginx/config (POST) — оновлює існуючу конфігурацію Nginx.
- /nginx/delete (POST) — видаляє конфігурацію Nginx.
- /nginx/disable (POST) — вимикає певну конфігурацію Nginx.
- /nginx/enable (POST) — вмикає конфігурацію Nginx.
- /nginx/restart (GET, HEAD) — перезапускає Nginx після змін.
- /pm2 (GET, HEAD) — повертає загальний статус процесів у PM2.
- /pm2/date/:name (POST) — повертає логи процесу за назвою та дату.
- /pm2/now/:name (GET, HEAD) — показує поточний статус процесу за назвою.
- /pm2/start/:name (GET, HEAD) — запускає процес за назвою.
- /pm2/stop/:name (GET, HEAD) — зупиняє процес за назвою.
- /pm2/restart/:name (GET, HEAD) — перезапускає процес за назвою.
- /* (OPTIONS) — обробляє CORS-запити для всіх ендпоінтів.

Структура ендпоінтів для Backend виглядає так:

- /auth/login (POST) — автентифікує користувача та видає JWT-токен.
- /auth/refresh (GET, HEAD) — оновлює токен для активної сесії.
- /server (POST, DELETE, GET, HEAD) — створює, видаляє або повертає список серверів.
- /server/getByName (POST) — повертає дані про сервер за назвою.
- /user (PATCH, POST, DELETE, GET, HEAD) — створює, оновлює, видаляє або повертає список користувачів.
- /user/checkLogin (POST) — перевіряє, чи зайнятий логін користувача.
- /* (OPTIONS) — обробляє CORS-запити для всіх ендпоінтів.

Інтеграція з PM2 реалізована через API агента. Ендпоінти /pm2/start/:name, /pm2/stop/:name і /pm2/restart/:name дозволяють керувати Node.js-процесами, викликаючи відповідні команди PM2. Наприклад, коли фронтенд надсилає запит на запуск процесу, бекенд передає його агенту, який через PM2 запускає процес і повертає статус. Ендпоінти /pm2/now/:name і /pm2/date/:name дають змогу

отримувати актуальні дані про процеси та їхні логи, що зберігаються в SQLite. Це дозволяє адміністратору швидко аналізувати стан застосунків.

Інтеграція з Nginx відбувається через ендпоінти /nginx. Агент може перевіряти статус веб-сервера, створювати нові конфігурації, оновлювати існуючі, а також перезапускати Nginx після змін. Наприклад, ендпоінт /nginx/config приймає JSON із новою конфігурацією, яку агент записує у відповідний файл і застосовує через перезапуск. Це забезпечує гнучке керування веб-сервером без прямого доступу до сервера. Усі операції з Nginx захищені автентифікацією, адже агент перевіряє токен, переданий через бекенд. [10]

Фронтенд на Vue.js взаємодіє з бекендом через API, використовуючи бібліотеку Axios для надсилання HTTP-запитів. Наприклад, коли користувач хоче увійти в систему, фронтенд викликає ендпоінт /auth/login, передаючи логін і пароль. Бекенд перевіряє дані в SQLite, генерує JWT-токен і повертає його фронтенду. Цей токен використовується для всіх подальших запитів, таких як /server чи /user, забезпечуючи безпечний доступ. Ендпоінт /auth/refresh дозволяє оновлювати токен, якщо сесія ще активна, що підвищує зручність для користувача.

Такий підхід до API та інтеграції дозволяє системі бути гнучкою та масштабованою. RESTful-ендпоінти забезпечують чіткий і передбачуваний інтерфейс для всіх компонентів, а інтеграція з PM2 і Nginx додає функціональності для адміністрування. У підсумку, спроєктоване API створює міцну основу для ефективної роботи системи моніторингу.

2.6 Забезпечення безпеки системи

Безпека системи моніторингу серверних застосунків спроєктована з урахуванням сучасних підходів, щоб захистити дані та доступ користувачів. Основна увага приділена механізмам аутентифікації та авторизації через JWT-токени, зокрема access і refresh токени, а також безпечній роботі з базою даних

через Prisma ORM. Використані технології, такі як Fastify для бекенду та агента, а також Vue.js для фронтенду, інтегровані з урахуванням захисту від типових вразливостей. Це дозволяє забезпечити надійність системи та захист від несанкціонованого доступу.

Аутентифікація користувачів побудована на основі JWT-токенів, які поділяються на access і refresh токени. Access-токен має короткий термін дії — 15 хвилин, і використовується для доступу до всіх ендпоінтів API, таких як /server чи /metrics/now. Після входу через ендпоінт /auth/login бекенд на Fastify перевіряє логін і пароль користувача, генерує access-токен і передає його фронтенду на Vue.js. Цей токен додається в заголовок Authorization для всіх подальших запитів, що дозволяє серверу перевіряти, чи користувач має право на певну дію. Refresh-токен, навпаки, має довший термін дії — 7 днів, і потрібен для оновлення access-токена без повторного введення логіну та пароля. Ендпоінт /auth/refresh приймає refresh-токен, перевіряє його валідність і видає новий access-токен. Обидва токени зберігаються в базі даних через Prisma ORM, що забезпечує безпечне управління сесіями.

Prisma ORM відіграє ключову роль у безпечній роботі з базою даних SQLite. Вона використовується як на бекенді, так і в агентах для обробки запитів до таблиць, таких як Users, Tokens і Permissions. Prisma автоматично екранує запити, що унеможливлює SQL-ін’єкції. Наприклад, при перевірці логіну та пароля через /auth/login Prisma формує запит до таблиці Users, уникаючи ручного введення SQL-коду, який міг би стати вразливим. У таблиці Tokens зберігаються refresh-токени, пов’язані з userId, що дозволяє швидко перевіряти їх під час оновлення сесії. Prisma також забезпечує типізацію даних, що знижує ризик помилок при роботі з базою, наприклад, при оновленні прав доступу в таблиці Permissions. Це робить взаємодію з SQLite безпечною та ефективною.

Авторизація в системі спирається на права доступу, визначені в таблиці Permissions. Після аутентифікації бекенд перевіряє, які дії користувач може виконувати: переглядати метрики, редагувати конфігурації чи керувати

процесами. Наприклад, якщо користувач із правом canView викликає ендпоінт /metrics/now, бекенд через Prisma перевіряє це право в базі та дозволяє доступ. Якщо ж користувач із правом canEdit намагається змінити конфігурацію через /nginx/config, система також перевіряє відповідне право. Access-токен містить ідентифікатор користувача, що дозволяє швидко зіставити його з правами в базі, не роблячи зайвих запитів. Це прискорює авторизацію та зменшує навантаження на бекенд.

Захист від типових вразливостей також враховано. Для запобігання XSS-атакам фронтенд на Vue.js очищає всі введені користувачем дані перед відображенням, наприклад, у полях логів чи конфігурацій. Fastify на бекенді та агентах використовує заголовки Content-Security-Policy, щоб обмежити джерела скриптів, які можуть виконуватися. Для захисту від CSRF-атак застосовується перевірка походження запитів через CORS, а також використання токенів у всіх POST-запитах, таких як /nginx/create чи /pm2/start/:name. Fastify автоматично обробляє ці заголовки, що спрощує захист. Крім того, усі API-ендпоінти, окрім /auth/login, вимагають наявності валідного access-токена, що унеможливлює доступ для неавторизованих користувачів.

Refresh-токени додають додатковий рівень безпеки. Якщо access-токен викрадено, зловмисник матиме доступ лише протягом 15 хвилин, після чого токен стане невалідним. Refresh-токен, хоча й має довший термін дії, зберігається в базі через Prisma і може бути анульований адміністратором через ендпоінт /user, якщо виявлено підозрілу активність. Наприклад, якщо користувач помічає несанкціонований доступ, він може змінити пароль, що автоматично анулює всі пов’язані токени. Це дозволяє швидко реагувати на загрози без зупинки системи.

У підсумку, безпека системи побудована на комбінації JWT-токенів, Prisma ORM і захисту від вразливостей. Access і refresh токени забезпечують гнучку та безпечну аутентифікацію, Prisma захищає базу даних від ін’єкцій, а додаткові заходи запобігають XSS і CSRF-атакам. Це створює надійний захист для системи моніторингу, дозволяючи адміністраторам працювати без ризиків.

3. РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СИСТЕМИ МОНІТОРИНГУ СЕРВЕРНИХ ЗАСТОСУНКІВ

3.1 Система управління інформаційною базою

Система управління інформаційною базою для моніторингу серверних застосунків побудована з використанням SQLite як основної бази даних, а також Prisma ORM для спрощення роботи з даними [11]. База даних поділена між бекендом і агентом, що дозволяє зберігати різні типи інформації окремо: бекенд відповідає за користувачів, токени, права доступу та сервери, а агент — за метрики. У цьому підрозділі детально розглянуто фізичну модель бекенду, включаючи структуру таблиць і приклади запитів, а модель агента винесено в додатки для кращої структуризації.

Фізична модель бекенду реалізована через Prisma, що забезпечує безпечну та зручну роботу з SQLite. Модель складається з чотирьох таблиць: Users, Tokens, Permissions і Servers. Таблиця Users зберігає дані про користувачів системи, включаючи унікальний ідентифікатор id, логін login, який є унікальним, пароль password, а також час створення createdAt і оновлення updatedAt. Таблиця Tokens пов’язана з Users через поле userId і містить ідентифікатор id, унікальний userId, сам токен token, час створення createdAt і останнього використання lastUsedAt. Таблиця Permissions також пов’язана з Users через userId і визначає права доступу: canView, canEdit, canControl, canDelete і canAdministrate, усі з яких за замовчуванням мають значення true, а також включає час створення та оновлення. Таблиця Servers зберігає інформацію про сервери з полями id, унікальними name і url, а також createdAt і updatedAt. Зв’язки між таблицями налаштовані з каскадним оновленням і видаленням, що забезпечує цілісність

даних: якщо користувач видаляється, його токени та права також видаляються автоматично.

```
generator client {
  provider = "prisma-client-js"
}
datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}
model Users {
  id        Int      @id @default(autoincrement())
  login  String   @unique
  password  String
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  Tokens Tokens?
  Permissions Permissions?
}
model Tokens {
  id        Int      @id @default(autoincrement())
  userId     Int      @unique
  token      String
  createdAt  DateTime @default(now())
  lastUsedAt DateTime @default(now())
  User        Users     @relation(fields: [userId], references: [id], onUpdate: Cascade, onDelete: Cascade)
}
model Permissions {
```

```
  id        Int      @id @default(autoincrement())
  userId    Int      @unique
  canView   Boolean  @default(true)
  canEdit   Boolean  @default(true)
  canControl Boolean  @default(true)
  canDelete Boolean  @default(true)
  canAdministrate Boolean  @default(true)
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  Users     Users     @relation(fields: [userId], references: [id], onUpdate: Cascade, onDelete: Cascade)
}
model Servers {
  id        Int      @id @default(autoincrement())
  name      String   @unique
  url       String   @unique
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}
```

Для роботи з таблицею Servers створено кілька запитів, які дозволяють створювати, отримувати, видаляти та шукати сервери. Функція create додає новий сервер, приймаючи name і url, і повертає створений запис. Функція getAll повертає список усіх серверів у базі. Функція delete видаляє сервер за його id, а findById і findByName дозволяють знайти сервер за id або назвою відповідно. Усі запити написані з використанням Prisma, що забезпечує захист від SQL-ін’єкцій і спрощує роботу з базою завдяки типізації та автодоповненню. Наприклад, виклик findByName для сервера з назвою "server1" поверне його дані, якщо такий сервер існує, або null, якщо ні.

```
const prisma = new PrismaClient();
```

```
module.exports = {
  async create(name, url) {
    return await prisma.servers.create({
      data: {
        name,
        url,
      },
    });
  },
  async getAll() {
    return await prisma.servers.findMany();
  },
  async delete(id) {
    return await prisma.servers.delete({
      where: {
        id,
      },
    });
  },
  async findById(id) {
    return await prisma.servers.findUnique({
      where: {
        id,
      },
    });
  },
  async findByName(name) {
    return await prisma.servers.findUnique({
      where: {
```

```
        name,
      },
    });
  },
};
```

Фізична модель агента, яка включає таблиці для метрик, винесена в ДОДАТОК А.

Використання SQLite як бази даних дозволяє системі бути легкою та простою в розгортанні, адже вона не потребує окремого серверного процесу, на відміну від складніших СУБД, таких як PostgreSQL. Prisma додає зручності в управлінні, дозволяючи писати запити у вигляді об’єктів JavaScript, що зменшує ризик помилок. Наприклад, при створенні сервера через функцію create Prisma автоматично перевіряє унікальність полів name і url, що унеможливлює дублювання. Це робить систему надійною для адміністрування невеликої кількості серверів.

Запити до бази даних оптимізовані для швидкої роботи. Наприклад, функція getAll повертає лише необхідні дані, не витягаючи зайвих полів, що зменшує навантаження на SQLite. Функція delete використовує точне видалення за id, що забезпечує ефективність навіть при великій кількості записів. Prisma також підтримує транзакції, що дозволяє виконувати кілька операцій атомарно, наприклад, створювати сервер і одразу оновлювати права доступу для користувача, який його додав. Це підвищує надійність системи в умовах одночасного доступу кількох адміністраторів.

У підсумку, система управління інформаційною базою забезпечує ефективне зберігання та обробку даних. Фізична модель бекенду, побудована через Prisma і SQLite, дозволяє надійно управляти користувачами, токенами, правами та серверами. Приклади запитів демонструють простоту роботи з даними, а деталі моделі агента доступні в ДОДАТКУ А для глибшого

ознайомлення. Така організація робить систему зручною для використання та підтримки.

3.2 Вибір інструментарію для створення прикладного програмного забезпечення

Для розробки системи моніторингу серверних застосунків підібрано набір інструментів, які забезпечують ефективність, легкість використання та сумісність із сучасними технологіями. Вибір базується на потребах системи, що включає веб-інтерфейс, бекенд, агента, а також інтеграцію з PM2 і Nginx. Використані інструменти дозволяють створити гнучку та масштабовану систему, яка підходить для адміністрування серверів у реальному часі.

Фронтенд розробляється з використанням Vue.js, популярного фреймворку для створення реактивних інтерфейсів. Vue.js обрано за його легкість і швидкість, що ідеально підходить для відображення графіків навантаження та логів. Для маршрутизації сторінок застосовано vue-router, який забезпечує плавну навігацію між секціями, такими як управління користувачами чи перегляд метрик. Багатомовність реалізовано через vue-i18n, що дозволяє адаптувати інтерфейс під різні мови. Для роботи з куками використано vue-cookies і js-cookies, які зберігають токени автентифікації в браузері, забезпечуючи безперервний доступ. Axios інтегровано для надсилання HTTP-запитів до бекенду, що спрощує обмін даними в реальному часі.

Бекенд і агент створені на основі Fastify, легковагового фреймворку для Node.js, який вирізняється високою продуктивністю. [7] Fastify обрано через його здатність швидко обробляти велику кількість запитів, що критично важливо для систем моніторингу. Для автоматичного завантаження плагінів використано fastify-autoload, що полегшує структуру коду. Fastify-cors забезпечує підтримку CORS, дозволяючи фронтенду взаємодіяти з бекендом із різних доменів. Fastify-jwt інтегровано для реалізації JWT-автентифікації, що захищає ендпоінти

від несанкціонованого доступу. Для шифрування паролів застосовано bcrypt, а для створення і перевірки токенів — jwt-token, що додає безпеки системі.

Для роботи з базою даних обрано SQLite через її простоту й відсутність потреби в окремому сервері. SQLite ідеально підходить для невеликих систем, де дані про користувачів, сервери та метрики зберігаються локально. Для спрощення взаємодії з SQLite використано Prisma ORM, яка забезпечує типізовані запити та захист від SQL-ін'єкцій. Prisma-client дозволяє легко створювати, оновлювати та видаляти записи, наприклад, у таблицях Users чи LoadMetrics, що прискорює розробку. Використання Prisma також полегшує інтеграцію бази даних із бекендом і агентом, забезпечуючи єдиний підхід до управління даними.

Інтеграція з PM2 реалізована через його API, яке дозволяє запускати, зупиняти та перезапускати Node.js-процеси. PM2 обрано за його стабільність і зручність у керуванні застосунками, що критично для агента, який моніторить сервери.

Для налаштування веб-сервера використано Nginx, який забезпечує проксіювання, балансування навантаження та маршрутизацію запитів. Nginx інтегровано з агентом через API, що дозволяє оновлювати конфігурації та перезапускати сервер без ручного втручання. Ця комбінація забезпечує гнучкість і надійність у роботі системи.

Для розробки та тестування застосовано Node.js як середовище виконання, яке підтримує як Fastify, так і Vue.js. Node.js обрано через його широку підтримку спільноти та сумісність із сучасними інструментами. Для збирання проєкту використовується Vite, який забезпечує швидку компіляцію фронтенду та гаряче оновлення під час розробки. Для відладки застосовано Visual Studio Code з розширеннями, такими як Prettier для форматування коду та ESLint для перевірки стилю, що підвищує якість коду. Git використовується для контролю версій, дозволяючи відстежувати зміни та співпрацювати в команді.

Вибрані інструменти інтегруються між собою безперебійно. Наприклад, фронтенд на Vue.js через Axios надсилає запити до бекенду на Fastify, який через Prisma працює з SQLite. Агент на Fastify взаємодіє з PM2 для керування процесами та з Nginx для оновлення конфігурацій, використовуючи ті самі принципи. Такий підхід дозволяє швидко реагувати на зміни та розширювати систему, додаючи нові сервери чи функції.

Вибір інструментарію забезпечує баланс між продуктивністю, безпекою та простотою. Vue.js і Fastify створюють швидкий і легкий каркас, SQLite і Prisma спрощують роботу з даними, а PM2 і Nginx додають потужності для серверного адміністрування. Ця комбінація робить систему зручною для розробки та використання в реальних умовах.

3.3 Алгоритмізація та програмування програмних модулів

Розробка системи моніторингу серверних застосунків включає створення програмних модулів для збору системних даних, управління процесами через PM2, аналізу логів і налаштування Nginx. Використовуються Node.js із Fastify для бекенду та агента, а також бібліотеки, такі як systeminformation і pm2, для інтеграції з серверною інфраструктурою. У цьому підрозділі представлено алгоритми та код для ключових модулів, які забезпечують функціональність системи в реальному часі. Повний код усіх модулів доступний у ДОДАТОК Б

Перший модуль відповідає за збір системних метрик із використанням бібліотеки systeminformation. Цей модуль асинхронно отримує дані про навантаження CPU, активну пам’ять, використання диска та мережеву активність. Функція getSystemStats повертає об’єкт із поточним навантаженням CPU, активною пам’яттю в байтах, загальним використаним простором на диску та мережевими показниками rx і tx у біт/с. У разі помилки дані виводяться в консоль для відладки. Цей модуль є основою для моніторингу серверів, надаючи актуальні метрики для подальшого аналізу.

Другий модуль реалізує періодичне оновлення метрик у базі даних. Використовуючи функцію getSystemStats, він порівнює поточні значення з останніми збереженими метриками, які зберігаються в об’єкті lastMetrics. Кожні 5 секунд модуль перевіряє зміни в CPU, пам’яті, диску та мережевих даних. Якщо значення відрізняються, нові дані записуються в базу через методи push і pushNetwork модуля metrics, який працює з Prisma ORM і SQLite. Наприклад, якщо мережевий трафік rx змінився, модуль оновлює його в базі та синхронізує lastMetrics. Це забезпечує безперервний моніторинг із мінімальним навантаженням на систему.

Третій модуль фокусується на зборі метрик застосунків через PM2. Використовуючи бібліотеку pm2, модуль підключається до PM2, отримує список процесів і витягує дані про навантаження CPU та пам’ять для кожного. Кожні 5 секунд він перевіряє зміни в цих метриках порівняно з lastMetrics. Якщо значення CPU чи пам’яті для певного процесу відрізняються, модуль викликає pushAppMetrics для збереження даних у базі з ідентифікатором процесу. Після завершення роботи модуль від’єднується від PM2, уникаючи витоків ресурсів. Це дозволяє ефективно стежити за станом Node.js-застосунків на серверах.

Четвертий модуль відповідає за створення конфігурацій Nginx. Асинхронна функція handler приймає об’єкт із назвою сайту site і конфігурацією config, записує їх у файл у /etc/nginx/sites-available за допомогою fs.promises і повертає відповідь із результатом. У разі помилки, наприклад, якщо доступ до файлу заборонено, функція повертає код 500 із деталями помилки. Цей модуль інтегрується з ендпоінтом /nginx/create і забезпечує гнучке налаштування веб-сервера.

П’ятий модуль активує конфігурацію Nginx через символічне посилання. Функція handler приймає назву сайту site, створює посилання з /etc/nginx/sites-available до /etc/nginx/sites-enabled і повертає успішну відповідь. Якщо виникає помилка, наприклад, через відсутність файлу, повертається код 500. Цей модуль

підтримує ендпоінт /nginx/enable, дозволяючи швидко застосовувати конфігурації.

Шостий модуль видаляє конфігурацію Nginx. Функція handler спочатку видаляє символічне посилання, якщо воно є, а потім видаляє сам файл із /etc/nginx/sites-available. У разі відсутності файлу повертається код 404, а при інших помилках — 500. Цей модуль інтегрується з ендпоінтом /nginx/delete, дозволяючи повністю прибирати непотрібні налаштування.

Сьомий модуль перезапускає Nginx. Функція handler виконує команду sudo systemctl restart nginx через exec і повертає результат із кодом успіху або помилкою. Це підтримує ендпоінт /nginx/restart, забезпечуючи оновлення конфігурацій після змін.

Ці модулі створюють основу для системи моніторингу, дозволяючи збирати дані, управляти процесами та налаштовувати сервер. Їхня асинхронна природа й обробка помилок забезпечують стабільність і надійність у реальному часі.

3.4 Реалізація інтерфейсу користувача

Інтерфейс користувача системи моніторингу серверних застосунків розроблено з використанням Vue.js, що забезпечує реактивність і зручність взаємодії. Компонент представляє сторінку для управління конкретним застосунком через PM2, відображаючи його деталі, метрики ресурсів і логи в реальному часі. Інтерфейс побудовано з використанням бібліотеки Ant Design Vue для компонентів, Plotly для графіків і xterm для терміналу, що додає функціональності та сучасного вигляду. [15] Сторінка інтегрована з бекендом і агентом через API, що дозволяє адміністраторам ефективно керувати серверами. Повний код доступний у ДОДАТОКВ.

Основна структура компонента включає кілька ключових частин. Компонент AppHeader забезпечує навігацію по системі, а a-breadcrumb створює

посилання для орієнтації. Наприклад, шлях від списку серверів до конкретного застосунку відображається як Servers → [назва сервера] → PM2 → [назва застосунку]. Це допомагає користувачу швидко повернутися на попередні сторінки, клікаючи на відповідні посилання. Блок керування застосунком містить три кнопки: "Запустити", "Перезапустити" і "Зупинити". Кнопки активуються залежно від статусу застосунку — якщо він офлайн, доступна лише кнопка запуску, а якщо онлайн — кнопки перезапуску та зупинки.

Блок "Деталі застосунку" відображає інформацію у форматі таблиці через a-descriptions. Тут показано назву застосунку, PID, статус (онлайн чи офлайн із відповідним кольоровим маркером), час роботи, кількість перезапусків, використання пам’яті, CPU, розмір і використання купи, затримку циклу подій, а також кількість активних дескрипторів і запитів. Час роботи форматується у зручний вигляд, наприклад, "2д 5г 30хв", а пам’ять конвертується з байтів у мегабайти для зрозумілості. Ці дані оновлюються кожну секунду через метод fetchProcessData, який викликає API-ендпоінт /pm2/now/:name через функцію getPm2App.

Блок "Використання ресурсів" дозволяє переглядати метрики CPU і пам’яті за певний період. Користувач може обрати діапазон дат через a-range-picker, який підтримує попередньо налаштовані періоди, такі як "Останні 7 днів". Після вибору діапазону метод onRangeChange викликає fetchChartData, який через API-ендпоінт /metrics/date отримує дані та оновлює графіки. Графіки створюються через Plotly: один для CPU у відсотках, інший для пам’яті в мегабайтах. Якщо діапазон не обрано, дані оновлюються кожні 5 секунд через reloadChart, що забезпечує моніторинг у реальному часі.

Блок "PM2 Логи" відображає логи застосунку в терміналі, створеному через xterm. Користувач може підписатися на логи, натиснувши кнопку "Підписатись на логи", що викликає метод subscribeLogs. Термінал ініціалізується через initializeTerminal, який підключається до WebSocket-сервера за допомогою socket.io-client. Логи в реальному часі надходять через подію

pm2Log і відображаються в терміналі, а помилки PM2 виводяться червоним кольором через подію pm2Error. Термінал має темну тему та підтримує прокрутку .

Логіка компонента побудована з урахуванням асинхронності та обробки помилок. При завантаженні компонента через mounted викликаються методи для отримання даних сервера, процесу та метрик. Дані оновлюються через setInterval, а при знищенні компонента (beforeUnmount) усі інтервали очищаються, WebSocket-сокет відключається, а термінал видаляється, щоб уникнути витоків пам’яті. Методи startProcess, restartProcess і stopProcess взаємодіють із API-ендпоінтами /pm2/start/:name, /pm2/restart/:name і /pm2/stop/:name, викликаючи відповідні функції startApp, restartApp і stopApp. Кожен виклик супроводжується повідомленням через message з Ant Design Vue, а для перезапуску та зупинки додається підтвердження через Modal.

Стилі компонента забезпечують адаптивність і зручність. Елементи, такі як a-card, мають тіні та округлені кути для кращого вигляду, а термінал стилізовано з темним фоном і білим текстом. Клас .button-group вирівнює кнопки керування горизонтально з відступами, а .info-panel обмежує ширину блоків для зручного перегляду на великих екранах.

Цей інтерфейс забезпечує адміністраторам повний контроль над застосунками через PM2, дозволяючи переглядати деталі, метрики та логи, а також швидко реагувати на зміни. Інтеграція з API через Axios і WebSocket забезпечує оперативність, а використання Ant Design Vue і Plotly додає зручності та візуальної привабливості.

3.5 Інтеграція та налаштування агента

Агент у системі моніторингу серверних застосунків відповідає за збір даних, управління процесами та налаштування веб-сервера. Його інтеграція з бекендом, PM2, Nginx і базою даних SQLite забезпечує оперативну роботу

системи. Агент побудовано на Fastify, що дозволяє швидко обробляти запити, а налаштування адаптовано для роботи на різних серверах із мінімальними вимогами до ресурсів.

Агент розгортається на кожному сервері, який потрібно моніторити, і запускається як системний сервіс через systemctl. Для цього створюється файл служби, /etc/systemd/system/agent.service, із командою для запуску Node.js-процесу: ExecStart=/usr/bin/node /path/to/agent/index.js. Команда systemctl enable agent.service активує автозапуск, а systemctl start agent.service запускає агента. Логи служби доступні через journalctl -u agent.service, що полегшує діагностику. Агент слухає запити на певному порті, наприклад, 3001, і взаємодіє з бекендом через HTTPS, використовуючи Fastify. Для безпеки всі запити потребують JWT-токена, який перевіряється через fastify-jwt, інтегруючи агента з механізмом автентифікації бекенду.

Інтеграція з PM2 дозволяє агенту керувати Node.js-процесами. Через API-ендпоінти, такі як /pm2/start/:name, /pm2/stop/:name і /pm2/restart/:name, агент викликає методи PM2, наприклад, pm2.start або pm2.stop. Це дає змогу адміністратору віддалено запускати, зупиняти чи перезапускати застосунки через веб-інтерфейс. Агент також збирає метрики процесів, використовуючи pm2.list для отримання даних про CPU і пам’ять, які потім зберігаються в локальній базі SQLite через Prisma ORM. Наприклад, якщо CPU-застосунку змінюється, агент записує це в таблицю LoadMetrics, що дозволяє фронтенду відображати дані через API-ендпоінт /metrics/now.

Для роботи з Nginx агент підтримує ендпоінти, такі як /nginx/create, /nginx/enable і /nginx/restart. Вони дозволяють створювати нові конфігурації, активувати їх через символічні посилання та перезапускати Nginx. Агент використовує fs.promises для роботи з файлами в /etc/nginx/sites-available і /etc/nginx/sites-enabled, а команда systemctl restart nginx викликається через exec для перезапуску. Це забезпечує гнучке управління веб-сервером. Наприклад,

після створення конфігурації через /nginx/create агент може активувати її через /nginx/enable і перезапустити Nginx для застосування змін.

Логи застосунків передаються в реальному часі через WebSocket, використовуючи socket.io. Агент підключається до PM2 для отримання логів, які потім транслюються через подію pm2Log. Фронтенд підписується на ці оновлення, відображаючи логи в терміналі на сторінці застосунку.

База даних агента на SQLite інтегрується через Prisma ORM. Таблиці LoadMetrics зберігають метрики. Prisma захищає від SQL-ін’єкцій, автоматично екрануючи запити, наприклад, при збереженні метрик через pushAppMetrics. Агент ініціалізує базу під час запуску, перевіряючи наявність файлу SQLite і створюючи його за допомогою prisma migrate deploy. Це дозволяє агенту працювати автономно на кожному сервері.

Налаштування агента включає кілька параметрів у файлі .env. Порт визначається через PORT=3001, шлях до бази даних — через DATABASE_URL, а секрет для JWT-токенів — через JWT_SECRET. Для інтеграції з бекендом вказується URL, наприклад, http://main-server:3000, через змінну BACKEND_URL. Права доступу до файлів Nginx налаштовуються так, щоб Node.js-процес мав необхідні дозволи для роботи з /etc/nginx/sites-available і /etc/nginx/sites-enabled. Для WebSocket відкривається окремий порт у брандмауері сервера, наприклад, 3001, щоб забезпечити зв’язок із фронтендом.

Агент розроблено з урахуванням стабільності та автономності. Його запуск через systemctl забезпечує надійність, інтеграція з PM2, Nginx і SQLite через Fastify і Prisma додає функціональності, а WebSocket забезпечує оперативність для логів у реальному часі. Налаштування агента дозволяє швидко розгортати його на нових серверах, що робить систему масштабованою.

4. ТЕСТУВАННЯ ТА ЕКСПЛУАТАЦІЯСИСТЕМИ МОНІТОРИНГУ СЕРВЕРНИХ ЗАСТОСУНКІВ

4.1 Тестування системи

Тестування системи проводилося вручну для перевірки основного функціоналу, включаючи автентифікацію, моніторинг серверів, керування процесами PM2, налаштування Nginx і відображення метрик. Процес включав послідовне виконання сценаріїв, щоб забезпечити коректність роботи інтерфейсу, агента та бекенду. Нижче описано кроки ручного тестування з відповідними скріншотами.

4.1.1 Перевірка створення паролю для автентифікації

Спочатку було запущено скрипт passwordGen.js для генерації логіна та пароля для доступу до веб-панелі. Команда node passwordGen.js виконана в терміналі в директорії /HiveKeeper-backend, що створило логін "admin" і пароль "14cbb5d0-ee7f-4d7b-b5b6-9a83ebd3ed2c". Результат видно на (рис. 8). Потім використано ці дані для входу в систему через форму автентифікації на сторінці входу (рис. 9). Успішний вхід підтвердив коректність механізму генерації паролів і автентифікації.

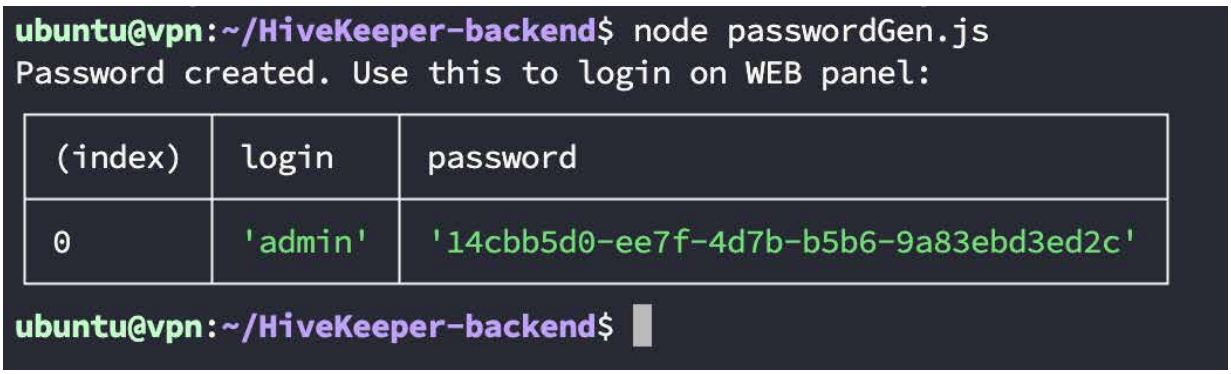


Рис. 8 Створення паролю

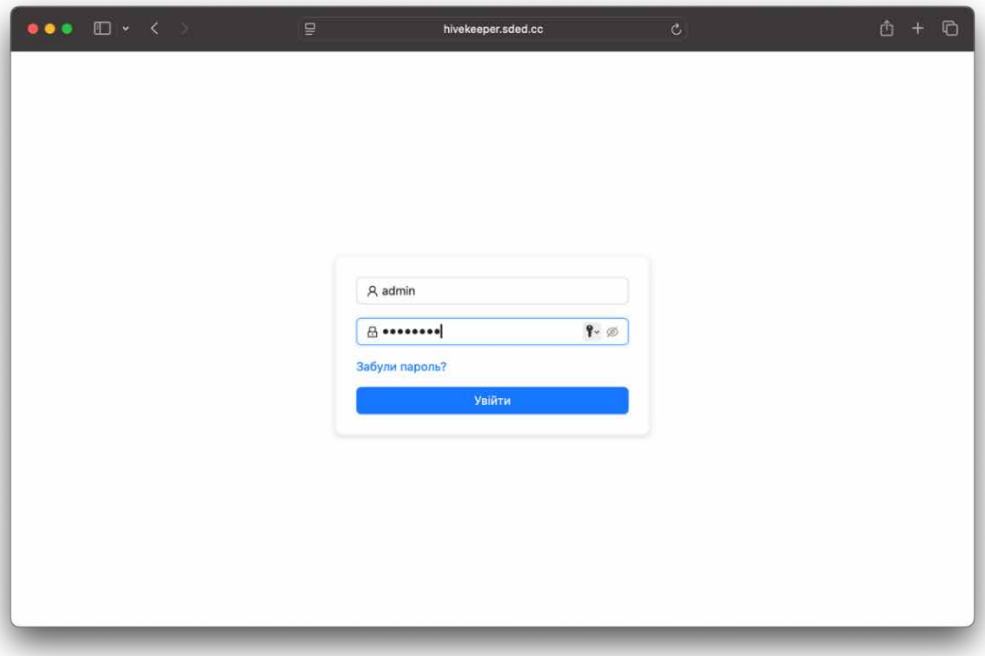


Рис. 9 Форма автентифікації

4.1.2 Перевірка відображення серверів і їхнього стану

Після входу відкрито сторінку "Список серверів" (рис. 10). Перевірено відображення серверних агентів s0 і s1 зі статусами "online".

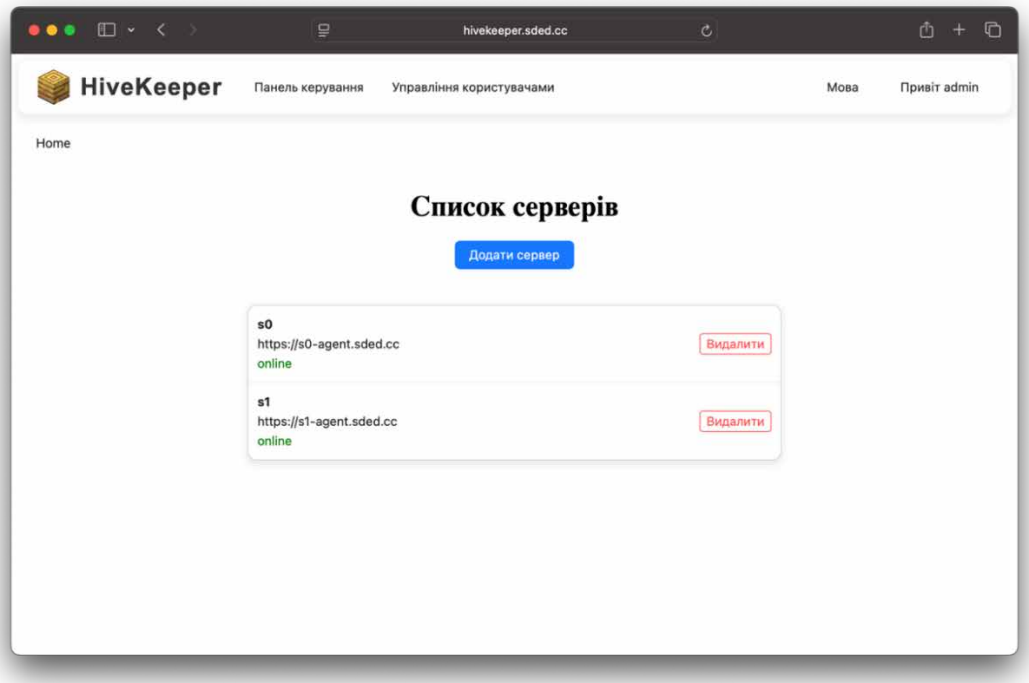


Рис. 10 Сторінка "Список серверів"

Кнопки "Додати сервер" доступні для взаємодії. Натиснення кнопки "Додати сервер" відкрило форму для введення даних (рис. 11). Успішне додавання підтвердило коректність інтеграції з бекендом.

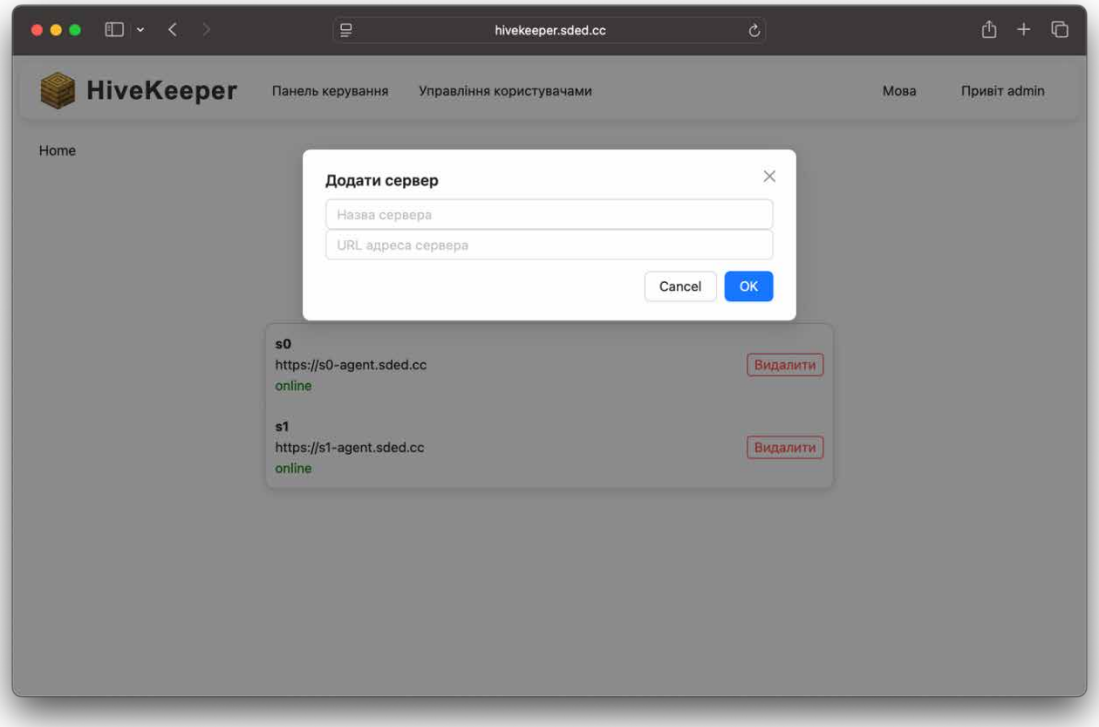


Рис. 11 Вікно "Додати сервер"

4.1.3 Перевірка моніторингу системних ресурсів

На сторінці "Dashboard" для сервера s1 (рис. 12) перевірено відображення поточних метрик: CPU (8%), RAM (1395.57 MB із 3919.92 MB), Disk (29.33 GB із 95.82 GB) і Network Load (RX: 0.02 Mbit/s, TX: 0.01 Mbit/s). Графіки CPU Usage і RAM Usage за проміжок часу з 20:39 до 20:44 (рис. 13) показали коректне відображення даних. Вибір діапазону дат через також оновив графіки, що підтвердило функціональність API-ендпоінта /metrics/date.

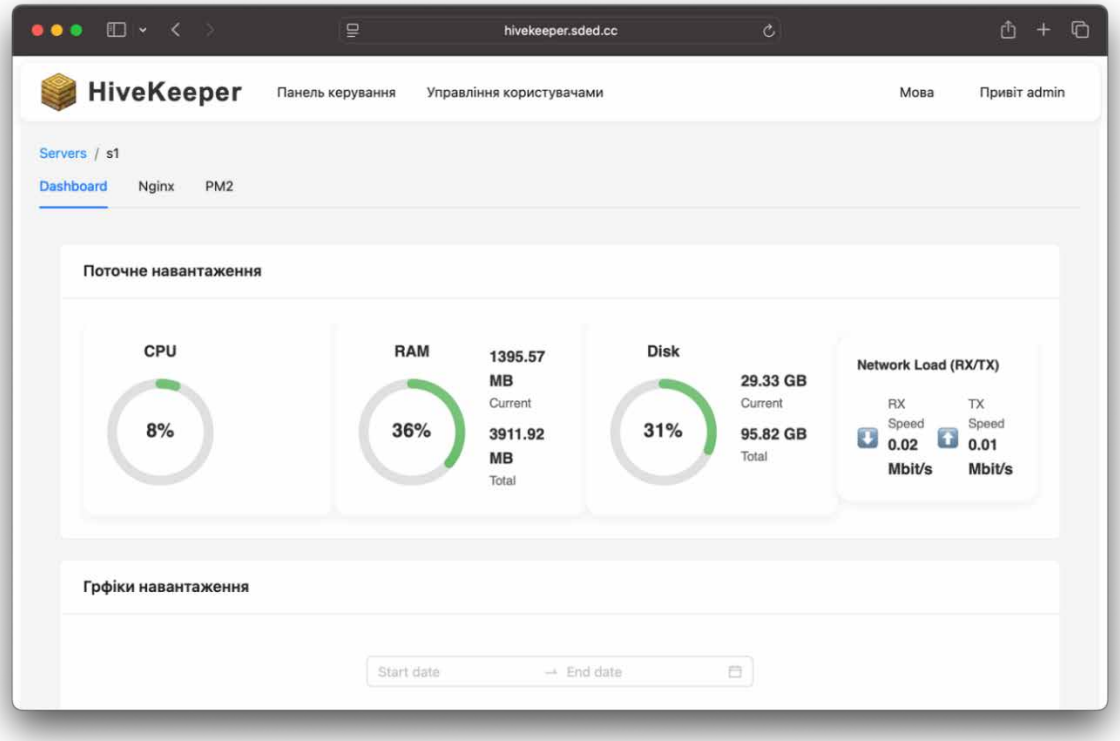


Рис. 12 Поточне навантаження сервера

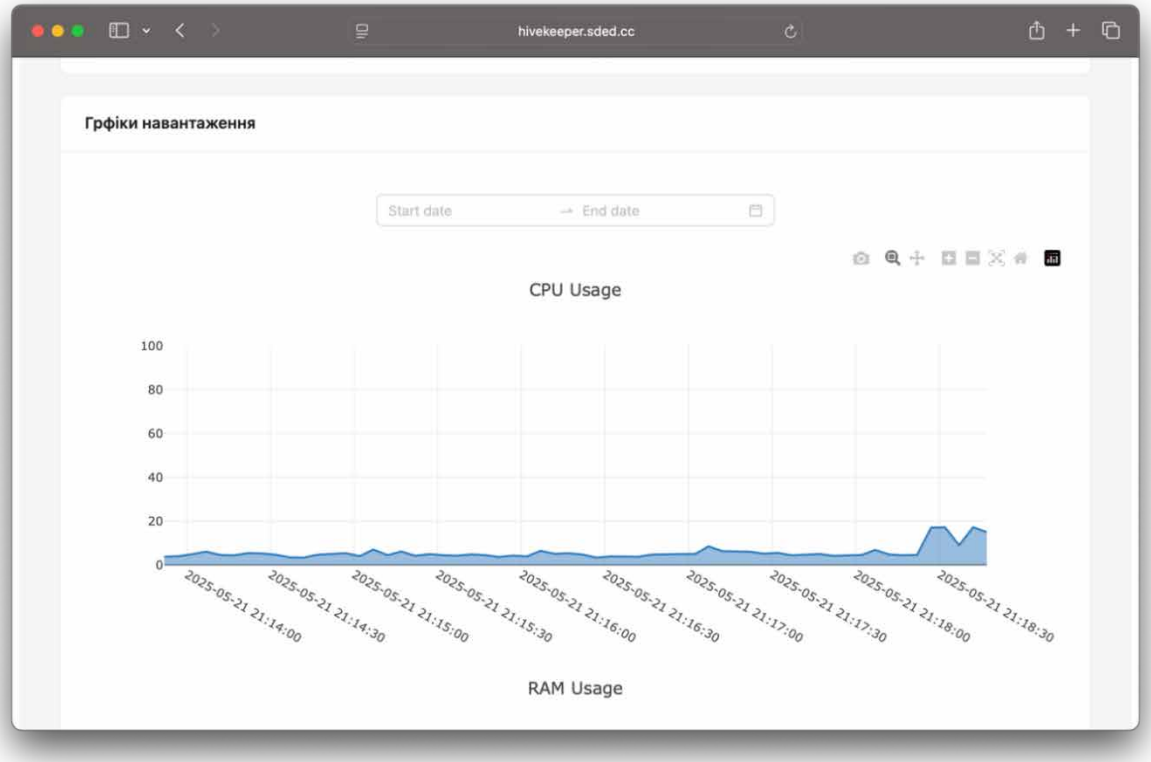


Рис. 13 Графіки навантаження

4.1.4 Перевірка керування PM2-процесами

На сторінці "PM2" для агента HiveKeeper-agent (рис. 14) перевірено деталі процесу: PID (3810250), статус "Онлайн", пам’ять (98.48 MB), CPU (2.9%) та інші метрики. Кнопки "Запустити", "Перезапустити" і "Зупинити" протестовано. Натиснення "Перезапустити" призвело до збільшення кількості перезапусків, а "Зупинити" змінило статус на "Офлайн", що підтвердило коректність ендпоінтів /pm2/restart/:name і /pm2/stop/:name.

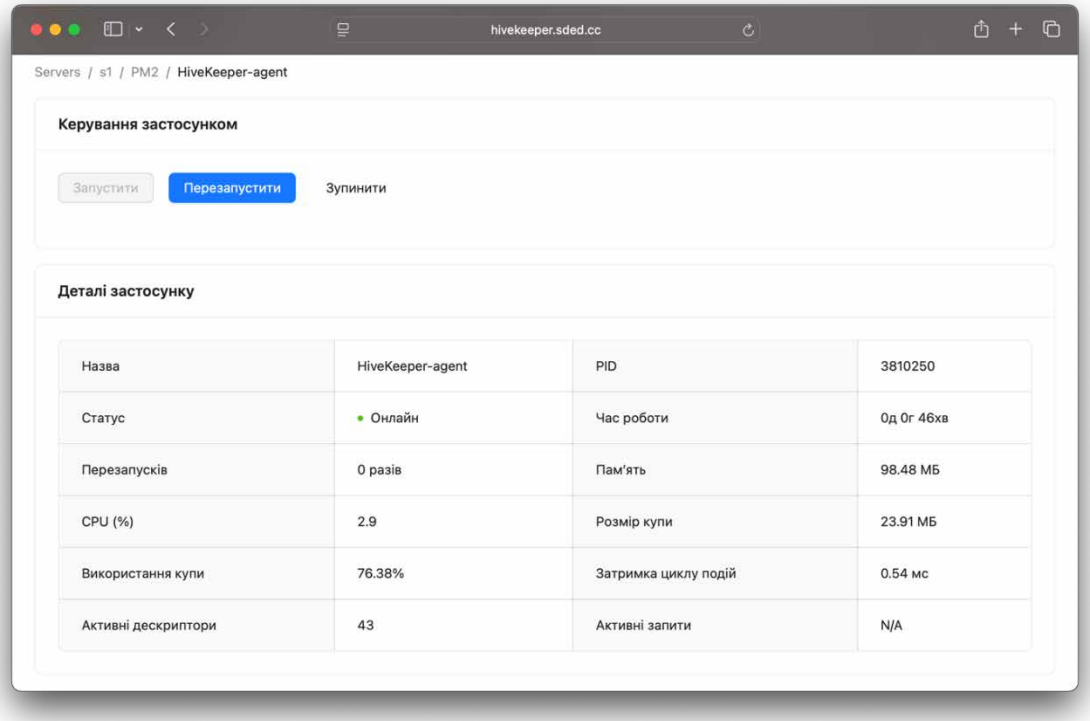


Рис. 14 Сторінка "PM2"

4.1.5 Перевірка логів у реальному часі

У блоці "PM2 Логи" (рис. 15) натиснуто "Підписатися на логи" для HiveKeeper-agent. Термінал відобразив повідомлення, включаючи "ДеньТиждень: вміст", що свідчить про підключення до WebSocket і отримання логів у реальному часі. Відсутність помилок у консолі підтвердила стабільність каналу socket.io.

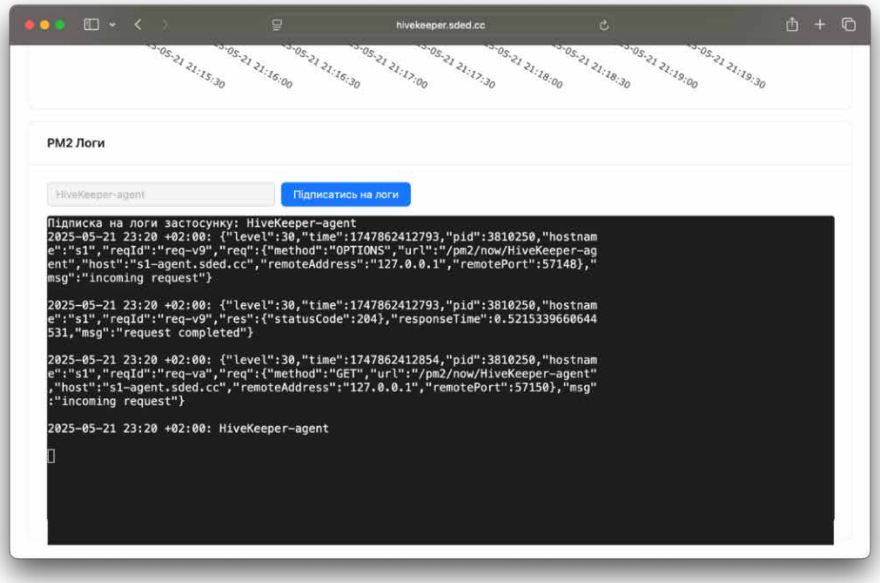


Рис 15 Блок "PM2 Логи"

4.1.6 Перевірка налаштування Nginx

На сторінці "Nginx" (рис. 16) перевірено статус конфігурацій: bitbond.sded.cc, default і s1-agent.sded.cc. Кнопка "Перезапустити Nginx" успішно оновила конфігурацію, а створення нової конфігурації через "Додати конфігурацію" (рис. 10) і її активація підтвердило коректність ендпоінтів /nginx/create і /nginx/enable.

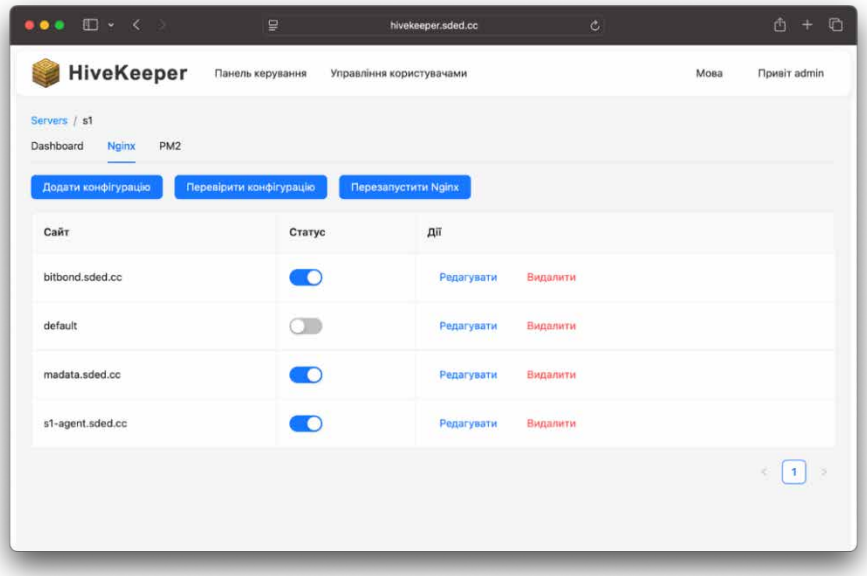


Рис. 16 Сторінка "Nginx"

Тестування показало, що система коректно обробляє автентифікацію, відображає метрики, дозволяє керувати процесами та налаштовувати Nginx. Усі компоненти (фронтенд, бекенд, агент) працюють синхронно, а помилок не виявлено. Результати підтверджують готовність системи до використання.

Тестування допоміжних вікон наведено в ДОДАТОКД

4.2 Вимоги до апаратного та програмного забезпечення

Для коректної роботи системи моніторингу серверних застосунків визначено мінімальні вимоги до апаратного та програмного забезпечення. Вони враховують потреби бекенду, агента та клієнтської частини, забезпечуючи стабільність і ефективність роботи. Система розрахована на використання сучасних технологій, таких як Node.js, SQLite і Nginx, що дозволяє розгортати її на різних серверах із мінімальними ресурсами.

4.2.1 Апаратне забезпечення

Для клієнтської частини, яка працює в браузері, достатньо стандартного комп’ютера з 4 ГБ оперативної пам’яті та двоядерним процесором, наприклад, Intel Core i3 або еквівалент. Операційна система може бути будь-якою сучасною, наприклад, Windows 10/11, macOS 12 чи новіші, або Linux-дистрибутиви, такі як Ubuntu 20.04. Відеокарта не потребує значних ресурсів, адже веб-інтерфейс на Vue.js не використовує складних графічних обчислень. Для комфортного перегляду рекомендується монітор із роздільною здатністю 1920x1080, хоча інтерфейс адаптивний і підтримує менші екрани. Стабільне інтернет-з’єднання зі швидкістю 10 Мбіт/с забезпечує швидке завантаження даних і оновлення логів через WebSocket.

Основний сервер, де розгортається бекенд, потребує мінімум 1 ГБ оперативної пам’яті та двоядерний процесор, наприклад, Intel Core i5 або

аналогічний AMD. Для зберігання бази даних SQLite достатньо 1 ГБ вільного місця на диску, бажано на SSD для швидшого доступу. Сервер працює під управлінням Linux, а саме Ubuntu, що забезпечує стабільність і низьке споживання ресурсів. Мережевий адаптер із пропускною здатністю 100 Мбіт/с гарантує швидку передачу даних між бекендом, агентами та клієнтами. Для масштабування на більшу кількість серверів рекомендується 2 ГБ пам’яті та чотириядерний процесор.

Додаткові сервери, на яких розгортається агент, мають аналогічні мінімальні вимоги: 1 ГБ оперативної пам’яті, двоядерний процесор і 1 ГБ вільного місця на диску. Агент працює на Linux, і потребує доступу до мережевого адаптера для передачі метрик і логів. Оскільки агент збирає дані про CPU, пам’ять і диск, бажано використовувати SSD для швидшого читання системних даних.

4.2.2 Програмне забезпечення

Клієнтська частина потребує сучасного веб-браузера з підтримкою ECMAScript 6, наприклад, Google Chrome 90+, Mozilla Firefox 85+, Microsoft Edge 90+ або Safari 14+. Браузер має підтримувати WebSocket для передачі логів у реальному часі та JavaScript для роботи з Vue.js і Plotly. Для коректного відображення інтерфейсу необхідно увімкнути JavaScript і дозволити спливні вікна, адже система використовує модальні діалоги Ant Design Vue для підтверджень, наприклад, при перезапуску застосунків.

Основний сервер із бекендом потребує встановлення Node.js версії 16 або новіше, адже Fastify і Prisma ORM сумісні з цією версією. SQLite версії 3.30+ використовується як база даних, що не потребує окремого серверного процесу, а працює як файлова база. Для обробки запитів Fastify вимагає модулі fastify-cors, fastify-autoload і fastify-jwt, які встановлюються через npm. Бекенд також залежить від bcrypt для шифрування паролів і socket.io для WebSocket-з’єднань. Операційна система Linux має бути налаштована з відкритими портами 3000 для

HTTPS і 3001 для WebSocket, а також із брандмауером, який дозволяє ці з’єднання.

Додаткові сервери для агента також потребують Node.js 16 або новіше. Агент використовує Fastify із тими ж модулями, що й бекенд, для обробки API-запитів. Для роботи з PM2 необхідно встановити PM2 версії 5.2+ через npm, що забезпечує управління Node.js-процесами. Nginx версії 1.18+ потрібен для проксіювання та управління веб-сервером, а агент має мати права доступу до каталогів /etc/nginx/sites-available і /etc/nginx/sites-enabled для створення конфігурацій. SQLite і Prisma встановлюються для локального зберігання метрик і конфігурацій. Порт 3001 відкривається для WebSocket, щоб передавати логи в реальному часі.

Ці вимоги дозволяють розгорнути систему на типових серверах і клієнтських пристроях, забезпечуючи стабільну роботу з моніторингом, управлінням процесами та налаштуванням веб-сервера.

4.3 Склад інсталяційного пакету

Інсталяційний пакет системи моніторингу серверних застосунків включає набір файлів і ресурсів, необхідних для розгортання бекенду, агента та фронтенду. Структура пакету побудована так, щоб забезпечити легке розгортання на серверах під управлінням Linux, а також доступ до веб-інтерфейсу через браузер. На основі прикріплених скріншотів описано вміст пакету, включаючи ключові файли, залежності та конфігурації для всіх компонентів системи.

Основна структура пакету поділена на три директорії: backend, agent і frontend. Директорія backend містить файли для основного сервера, який координує роботу системи. У ній розташовано файл index.js — точка входу для бекенду, що запускає Fastify-сервер. Папка routes включає модулі для обробки API-ендпоінтів, таких як /auth/login і /server, із файлами auth.js і server.js

відповідно. Папка utils містить допоміжні функції. Файл prisma/schema.prisma визначає схему бази даних. Файл .env.example містить шаблон змінних середовища, таких як PORT=3000 і DATABASE_URL, які потрібно налаштувати перед запуском.

Директорія agent призначена для розгортання агента на додаткових серверах. Вона також містить файл index.js як точку входу для Fastify-сервера агента. Папка routes включає файли для обробки ендпоінтів. Папка db із модулем metrics.js забезпечує взаємодію з локальною базою SQLite через Prisma, зберігаючи метрики в таблицях. Файл prisma/schema.prisma визначає схему бази агента. Файл .env.example включає змінні, такі як PORT=3001, DATABASE_URL і BACKEND_URL, для налаштування зв'язку з бекендом. Директорія utils містить функції, наприклад, getSystemStats.js для збору системних метрик через systeminformation.

Директорія frontend містить файли для веб-інтерфейсу на Vue.js. Файл index.html у корені є точкою входу для браузера, підключаючи основний скрипт main.js із папки src. Папка src/assets включає статичні ресурси, такі як logo.png і styles.css, які використовуються для оформлення інтерфейсу.

Залежності для всіх компонентів описані у відповідних файлах package.json. Для бекенду це Fastify, fastify-cors, fastify-autoload, fastify-jwt, bcrypt, socket.io і @prisma/client, які встановлюються через npm. Агент залежить від тих самих модулів Fastify, а також від systeminformation, pm2 і socket.io для інтеграції з WebSocket.

Конфігураційні файли для сервера включають systemd-служба для запуску бекенду та агента. Файл /etc/systemd/system/backend.service визначає запуск бекенду з командою ExecStart=/usr/bin/node /path/to/backend/index.js, а /etc/systemd/system/agent.service — для агента з аналогічною командою. Ці файли додаються до пакету для спрощення налаштування автозапуску через systemctl. Для Nginx додається приклад конфігураційного файлу nginx.conf у папці config,

який налаштовує проксіювання запитів до бекенду на порту 3000 і до WebSocket на порту 3001.

Пакет також містить документацію у файлі README.md, який описує структуру директорій, залежності та кроки для запуску. Файли ліцензії, такі як LICENSE, додаються в корінь пакету, щоб визначити умови використання.

Інсталяційний пакет забезпечує повний набір файлів для розгортання системи. Він включає код, залежності, конфігурації та документацію, що дозволяє швидко налаштувати бекенд, агент і фронтенд на відповідних серверах і клієнтських пристроях.

4.4 Рекомендації щодо експлуатації

Для стабільної роботи системи моніторингу серверних застосунків адміністраторам необхідно дотримуватися певних рекомендацій, які забезпечать ефективність, безпеку та довговічність використання. Ці поради враховують специфіку бекенду, агента, фронтенду та інтегрованих компонентів, таких як PM2, Nginx і SQLite.

Регулярно перевіряйте стан бекенду та агентів, використовуючи команду systemctl status backend.service чи agent.service. Це допоможе виявити збої або перевантаження, особливо в години пікової активності, наприклад, вранці чи вдень. Якщо система не реагує, перезапустіть сервіси через systemctl restart backend.service або systemctl restart agent.service. Слідкуйте за логами через journalctl -u backend.service або journalctl -u agent.service, щоб швидко реагувати на помилки, пов’язані з API чи базою даних.

Оновлюйте залежності системи регулярно, наприклад, раз на місяць. Використовуйте npm update для оновлення Node.js-модулів, таких як Fastify, Prisma і socket.io, у директоріях backend і agent. Переконайтеся, що версії Node.js (рекомендується 16 або новіше) і SQLite (3.30+) сумісні з оновленими бібліотеками. Для фронтенду оновлюйте Vue.js, Ant Design Vue і Plotly через npm

у директорії frontend, щоб уникнути проблем із відображенням графіків чи інтерфейсу. Перед оновленням створіть резервну копію бази даних SQLite, виконавши команду cp db/db.sqlite3 db/db.sqlite3.bak, щоб убезпечити дані.

Забезпечте безпеку доступу, регулярно оновлюючи JWT ключ у файлі .env. Змінюйте JWT_SECRET кожні три місяці або після підозрілих дій, наприклад, якщо логах з'являються невідомі запити. Використовуйте складні паролі для користувачів. Уникайте відкриття зайвих портів, залишаючи доступними лише 3000 для HTTPS і 3001 для WebSocket, і налаштуйте брандмауер, наприклад, ufw allow 3000/tcp і ufw allow 3001/tcp.

Ці рекомендації допоможуть адміністраторам підтримувати систему в робочому стані, запобігати збоїв і оптимізувати її використання. Регулярне оновлення, моніторинг і резервне копіювання забезпечать надійну експлуатацію системи моніторингу.

ВИСНОВКИ

Розробка системи моніторингу серверних застосунків стала значним кроком у створенні інструменту для ефективного адміністрування серверів у реальному часі. На основі проведеної роботи, можна підвести підсумки досягнень і потенціалу системи. Проєкт реалізовано з використанням сучасних технологій, таких як Vue.js, Fastify, Node.js, SQLite, Prisma, PM2, Nginx і socket.io, що забезпечило високу продуктивність і масштабованість.

Система складається з трьох основних компонентів: фронтенду, бекенду та агента, кожен із яких виконує чітко визначені функції. Фронтенд на Vue.js із бібліотеками Ant Design Vue, Plotly і xterm надає зручний інтерфейс для перегляду метрик, управління процесами та логів, адаптований для різних пристроїв. Бекенд, побудований на Fastify, координує запити, обробляє автентифікацію через JWT і взаємодіє з базою даних SQLite через Prisma, забезпечуючи безпечний доступ до даних користувачів і серверів. Агент, також на Fastify, інтегрований із PM2 і Nginx, збирає системні метрики, керує процесами і налаштовує веб-сервер, передаючи логи в реальному часі через WebSocket.

Архітектура системи довела свою гнучкість і модульність. Діаграми компонентів і розгортання показали чіткий розподіл ролей між фронтендом, бекендом і агентами, що дозволяє легко додавати нові сервери чи розширювати функціонал. API, спроєктоване з RESTful-підходом, забезпечило зручну інтеграцію між компонентами, а безпека, реалізована через JWT-токени і Prisma, захищає від типових загроз, таких як SQL-ін’єкції чи несанкціонований доступ.

Інсталяційний пакет включає всі необхідні файли, від коду до документації, що спрощує розгортання на Linux-серверах. Вимоги до апаратного та програмного забезпечення є мінімальними, що робить систему доступною для широкого кола користувачів, від невеликих серверів із 1 ГБ пам’яті до масштабних інфраструктур. Рекомендації щодо експлуатації, такі як регулярне

оновлення залежностей, моніторинг навантаження та резервне копіювання, забезпечать стабільну роботу системи в довгостроковій перспективі.

Проведена робота виявила, що система ефективно справляється з моніторингом, управлінням процесами та налаштуванням веб-сервера. Проте є простір для вдосконалень, наприклад, оптимізація роботи з великими обсягами метрик чи додавання підтримки інших баз даних, таких як PostgreSQL, для масштабування. Усе ж нинішня реалізація повністю відповідає поставленим цілям і готова до практичного використання адміністраторами серверів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Holmes, L. Node.js Design Patterns / L. Holmes. — 3rd ed. — Birmingham: Packt Publishing, 2020. — 524 p.
2. Amsden, E. Nginx Cookbook: Advanced Recipes for High Performance Load Balancing / E. Amsden, C. Castelnuovo. — Sebastopol: O'Reilly Media, 2021. — 328 p.
3. Cantelon, M. Node.js in Action / M. Cantelon, M. Harter, T. J. Holowaychuk, N. Rajlich. — 2nd ed. — Shelter Island: Manning Publications, 2014. — 392 p.
4. Resig, J. Secrets of the JavaScript Ninja / J. Resig, B. Bibeault. — 2nd ed. — Shelter Island: Manning Publications, 2013. — 416 p.
5. Reese, G. Database Systems: A Practical Approach to Design, Implementation, and Management / G. Reese, T. Connolly, C. Begg. — 6th ed. — Boston: Pearson Education, 2015. — 1440 p.
6. Vue.js Official Documentation [Електронний ресурс]. — Режим доступу: https://vuejs.org/guide/introduction.html.
7. Fastify Official Documentation [Електронний ресурс]. — Режим доступу: https://www.fastify.io/docs/latest/.
8. SQLite Official Documentation [Електронний ресурс]. — Режим доступу: https://www.sqlite.org/docs.html.
9. PM2 Official Documentation [Електронний ресурс]. — Режим доступу: https://pm2.keymetrics.io/docs/usage/pm2-doc-single-page/.
10. Nginx Official Documentation [Електронний ресурс]. — Режим доступу: https://nginx.org/en/docs/.
11. Prisma ORM Official Documentation [Електронний ресурс]. — Режим доступу: https://www.prisma.io/docs/.
12. Socket.IO Official Documentation [Електронний ресурс]. — Режим доступу: https://socket.io/docs/v4/.

13. Uptime Institute’s 2022 Outage Analysis Report [Електронний ресурс]. — Режим доступу: https://uptimeinstitute.com/about-ui/press-releases/2022-outage-analysis-finds-downtime-costs-and-consequences-worsening
14. Medium: JavaScript and Node.js Articles [Електронний ресурс]. — Режим доступу: https://medium.com/tag/javascript.
15. Plotly JavaScript Documentation [Електронний ресурс]. — Режим доступу: https://plotly.com/javascript/.

ДОДАТОКА

Фізична модель агенту:

```
generator client {
  provider = "prisma-client-js"
}
datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}
enum MetricType {
  CPU
  MEMORY
  DISK
}
model LoadMetrics {
  id       Int      @id @default(autoincrement())
  type     MetricType
  value    Float?
  time     DateTime  @default(now())
  @@index([time])
}
model NetworkMetrics {
  id       Int      @id @default(autoincrement())
  rx       Float?
  tx       Float?
  time     DateTime  @default(now())
  @@index([time])
}
model AppMetrics {
  id       Int      @id @default(autoincrement())
  name     String
  type     MetricType
  value    Float?
  time     DateTime  @default(now())
  @@index([time])
}
```

Моделі агенту

```
const prisma = new PrismaClient();


module.exports = {
  async push(type, value, value2) {
    if (type === "NETWORK")
      return await prisma.LoadMetrics.create({
        data: {
          type,
          rxLoad: value,
```

```
        txLoad: value2,
      },
    });
  else
    return await prisma.LoadMetrics.create({
      data: {
        type,
        value,
      },
    });
},
async pushNetwork(rx, tx) {
  return await prisma.NetworkMetrics.create({
    data: {
      rx,
      tx,
    },
  });
},

async getLast(type) {
  return await prisma.LoadMetrics.findMany({
    where: {
      type,
    },
    take: 1,
    orderBy: {
      id: "desc",
    },
  });
},
async getLastNetwork() {
  return await prisma.NetworkMetrics.findMany({
    take: 1,
    orderBy: {
      id: "desc",
```

```
    },
  });
},

async getMetrics(type, startDate, endDate) {
  return await prisma.LoadMetrics.findMany({
    where: {
      type,
      time: {
        gte: startDate,
        lte: endDate,
      },
    },
    select: {
      value: true,
      time: true,
    },
  });
},
async getNetworkMetrics(startDate, endDate) {
  return await prisma.NetworkMetrics.findMany({
    where: {
      time: {
        gte: startDate,
        lte: endDate,
      },
    },
    select: {
      rx: true,
      tx: true,
      time: true,
    },
  });
},

async pushAppMetrics(name, type, value) {
```

```
    return await prisma.AppMetrics.create({
      data: {
        name,
        type,
        value,
      },
    });
  },
  async getAppMetrics(name, type, startDate, endDate) {
    return await prisma.AppMetrics.findMany({
      where: {
        name,
        type,
        time: {
          gte: startDate,
          lte: endDate,
        },
      },
      select: {
        value: true,
        time: true,
      },
    });
  },
  async getAppLast(type) {
    return await prisma.AppMetrics.findMany({
      where: {
        type,
      },
      take: 1,
      orderBy: {
        id: "desc",
      },
    });
  },
};
```

ДОДАТОК Б

Збір системних метрик

```
module.exports = async () => {
  try {
    const si = require("systeminformation");
    const cpuLoad = await si.currentLoad();
    const mem = await si.mem();
    const netStats = await si.networkStats();
    const diskUsage = await si.fsSize();
    const totalUsed = diskUsage.reduce((acc, disk) => acc + disk.used, 0);
    return {
      cpu: cpuLoad.currentLoad.toFixed(2),
      memory: mem.active,
      disk: totalUsed,
      network: {
        rx: (netStats[0].rx_sec * 8).toFixed(2),
        tx: (netStats[0].tx_sec * 8).toFixed(2),
      },
    };
  } catch (error) {
    console.error("Error fetching system stats:", error);
  }
};
```

```
Періодичне оновлення метрик у базі даних
module.exports = async () => {
  const metrics = require("@db/metrics");
  const getSystemStats = require("@utils/getSystemStats");

  let lastMetrics = {
    cpu: 0.0,
    memory: 0,
    disk: 0,
    rx: 0.0,
    tx: 0.0,
  };
  await metrics.getLast("CPU").then((res) => {
    lastMetrics.cpu = res[0]?.value ? res[0].value : 0.0;
  });
  await metrics.getLast("MEMORY").then((res) => {
    lastMetrics.memory = res[0]?.value ? res[0].value : 0;
  });
  await metrics.getLast("DISK").then((res) => {
    lastMetrics.disk = res[0]?.value ? res[0].value : 0;
  });
  await metrics.getLastNetwork().then((res) => {
    lastMetrics.rx = res[0]?.rx ? res[0].rx : 0.0;
    lastMetrics.tx = res[0]?.tx ? res[0].tx : 0.0;
  });
  setInterval(async () => {
    const stats = await getSystemStats();
    if (
      stats.network.rx != 0.0 &&
      stats.network.tx != 0.0 &&
      (stats.network.rx != lastMetrics.rx || stats.network.tx != lastMetrics.tx)
    ) {
      await metrics.pushNetwork(
        parseFloat(stats.network.rx),
        parseFloat(stats.network.tx)
      );
      lastMetrics.rx = stats.network.rx;
      lastMetrics.tx = stats.network.tx;
    }
    if (stats.cpu != lastMetrics.cpu) {
      await metrics.push("CPU", parseFloat(stats.cpu));
      lastMetrics.cpu = stats.cpu;
    }
    if (stats.memory != lastMetrics.memory) {
```

```
      await metrics.push("MEMORY", parseFloat(stats.memory));
      lastMetrics.memory = stats.memory;
    }
    if (stats.disk != lastMetrics.disk) {
      await metrics.push("DISK", parseFloat(stats.disk));
      lastMetrics.disk = stats.disk;
    }
  }, 5000);
};
```

Збір метрик застосунків через PM2

```
module.exports = async () => {
  const metrics = require("@db/metrics");
  let lastMetrics = {
    cpu: 0.0,
    memory: 0,
  };
  await metrics.getAppLast("CPU").then((res) => {
    lastMetrics.cpu = res[0]?.value ? res[0].value : 0.0;
  });
  await metrics.getAppLast("MEMORY").then((res) => {
    lastMetrics.memory = res[0]?.value ? res[0].value : 0;
  });
  setInterval(async () => {
    const pm2 = require('pm2');
    try {
      await new Promise((resolve, reject) => {
        pm2.connect((err) => {
          if (err) {
            console.error(err);
            reject(new Error('ERR_PM2_CANT_CONNECT'));
          } else {
            resolve();
          }
        });
      });
      const processList = await new Promise((resolve, reject) => {
        pm2.list((err, list) => {
          if (err) {
            console.error(err);
            reject(new Error('ERR_PM2_LIST'));
          } else {
            resolve(list);
          }
        });
      });
      const data = processList.map(({ monit: { memory, cpu }, name }) => ({
        name,
        memory,
        cpu,
      }));
      if (data.length > 0) {
        data.forEach(async (process) => {
          if (process.cpu != lastMetrics.cpu) {
            await metrics.pushAppMetrics(process.name, "CPU", parseFloat(process.cpu),);
            lastMetrics.cpu = process.cpu;
          }
          if (process.memory != lastMetrics.memory) {
            await metrics.pushAppMetrics(process.name, "MEMORY", parseFloat(process.memory));
            lastMetrics.memory = process.memory;
          }
        });
      }
    } catch (error) {
      console.log(error);
    } finally {
      pm2.disconnect();
    }
  }, 5000);
};
```

Створення конфігурацій Nginx

```
async function handler(request, reply) {
  const fs = require("fs/promises");
  const path = require("path");
  const availableDir = "/etc/nginx/sites-available";
  const { site, config } = request.body;

  try {
    const configPath = path.join(availableDir, site);
    await fs.writeFile(configPath, config, { encoding: "utf8" });
    return reply.send({
      message: `Конфігурацію створено.`,
    });
  } catch (err) {
    console.error(err);
    return reply
      .status(500)
      .send({ error: "Помилка створення конфігурації." });
  }
}
```

Активація конфігурації Nginx

```
async function handler(request, reply) {
  const fs = require("fs/promises");
  const path = require("path");
  const availableDir = "/etc/nginx/sites-available";
  const enabledDir = "/etc/nginx/sites-enabled";
  const { site } = request.body;
  try {
    const availablePath = path.join(availableDir, site);
    const enabledPath = path.join(enabledDir, site);
    await fs.symlink(availablePath, enabledPath);
    return reply.send({ message: `Сайт активовано.` });
  } catch (err) {
    console.error(err);
    return reply
      .status(500)
      .send({ error: "Помилка активації сайту." });
  }
}
```

Видалення конфігурації Nginx

```
async function handler(request, reply) {
  const fs = require("fs/promises");
  const path = require("path");
  const availableDir = "/etc/nginx/sites-available";
  const enabledDir = "/etc/nginx/sites-enabled";
  const { site } = request.body;
  try {
    const availablePath = path.join(availableDir, site);
    const enabledPath = path.join(enabledDir, site);
    try {
      await fs.unlink(enabledPath);
    } catch (err) {
      if (err.code !== "ENOENT") {
        throw err;
      }
    }
    await fs.unlink(availablePath);
    return reply.send({
      message: `Configuration for site '${site}' has been deleted.`,
    });
  } catch (err) {
    console.error(err);
    if (err.code === "ENOENT") {
      return reply
        .status(404)
```

```
        .send({ error: `Configuration for site '${site}' does not exist.` });
    }
    return reply
      .status(500)
      .send({ error: "An error occurred while deleting the configuration." });
  }
}
```

Перезапуск Nginx

```
function handler(request, reply) {
  exec("sudo systemctl restart nginx", (error, stdout, stderr) => {
    if (error) {
      console.error(`Error restarting Nginx: ${stderr}`);
      return reply.send({
        success: false,
        message: "Failed to restart Nginx.",
        error: stderr.trim(),
      });
    }

    reply.send({
      success: true,
      message: "Nginx restarted successfully.",
      output: stdout.trim(),
    });
  });
}
```

ДОДАТОК В

```
<template>
    <AppHeader />
    <div class="app-viewer">
        <a-breadcrumb class="breadcrumb">
            <a-breadcrumb-item><a @click="this.$router.push('/')">Servers</a></a-breadcrumb-item>
            <a-breadcrumb-item><a @click="this.$router.push(`/server/${this.$route.params.name}`)">{{
                this.$route.params.name }}</a></a-breadcrumb-item>
            <a-breadcrumb-item><a
                    @click="this.$router.push(`/server/${this.$route.params.name}/pm2`)">PM2</a></a-breadcrumb
item>
            <a-breadcrumb-item>{{ this.$route.params.appName }}</a-breadcrumb-item>
        </a-breadcrumb>
        <a-card title="Керування застосунком" class="info-panel">
            <div class="button-group">
                <a-button type="primary" :disabled="data.status === 'online'" @click="startProcess">
                    Запустити
                </a-button>
                <a-button type="primary" :disabled="data.status !== 'online'" @click="confirmRestart">
                    Перезапустити
                </a-button>
                <a-button type="danger" :disabled="data.status !== 'online'" @click="confirmStop">
                    Зупинити
                </a-button>
            </div>
        </a-card>
        <a-card title="Деталі застосунку" class="info-panel">
            <a-descriptions bordered :column="2">
                <a-descriptions-item label="Назва">
                    {{ data.name }}
                </a-descriptions-item>
                <a-descriptions-item label="PID">
                    {{ data.pid }}
                </a-descriptions-item>
                <a-descriptions-item label="Статус">
                    <a-badge :status="data.status === 'online' ? 'success' : 'error'"
                        :text="data.status === 'online' ? 'Онлайн' : 'Офлайн'" />
                </a-descriptions-item>
                <a-descriptions-item label="Час роботи">
                    {{ formatUptime(data.uptime) }}
                </a-descriptions-item>
                <a-descriptions-item label="Перезапусків">
                    {{ data.restarts }} разів
                </a-descriptions-item>
                <a-descriptions-item label="Пам’ять">
                    {{ formatMemory(data.memory) }}
                </a-descriptions-item>
                <a-descriptions-item label="CPU (%)">
                    {{ data.cpu }}
                </a-descriptions-item>
                <a-descriptions-item label="Розмір купи">
                    {{ data.heap_size }} МБ
                </a-descriptions-item>
                <a-descriptions-item label="Використання купи">
                    {{ data.heap_usage }}%
                </a-descriptions-item>
                <a-descriptions-item label="Затримка циклу подій">
                    {{ data.event_loop_latency }} мс
                </a-descriptions-item>
                <a-descriptions-item label="Активні дескриптори">
                    {{ data.active_handles }}
                </a-descriptions-item>
                <a-descriptions-item label="Активні запити">
                    {{ data.active_requests }}
                </a-descriptions-item>
            </a-descriptions>
        </a-card>

        <a-card title="Використання ресурсів" class="info-panel">
            <div class="date-picker-container">
```

```
                <a-range-picker  style="width:  400px"  show-time  format="YYYY/MM/DD  HH:mm:ss"
:presets="rangePresets"
                    @change="onRangeChange" />
            </div>
            <div ref="cpuChart" style="width: 100%; height: 400px;"></div>
            <div ref="ramChart" style="width: 100%; height: 400px;"></div>
        </a-card>

        <a-card title="PM2 Логи" class="info-panel">
            <div class="log-controls">
                <a-input v-model:value="data.appName" placeholder="Назва застосунку"
                    style="width: 300px; margin-right: 8px;" :disabled="true" />
                <a-button type="primary" @click="subscribeLogs">
                    Підписатись на логи
                </a-button>
            </div>
            <div ref="terminalContainer" class="terminal"></div>
        </a-card>
    </div>
</template>

<script>
import AppHeader from '@/components/Header.vue';

import dayjs from 'dayjs';
import { message, Modal } from 'ant-design-vue';
import { getPm2App, startApp, stopApp, restartApp } from '@/api/agent';
import { getByName } from '@/api/server';
import { getAppByDate } from '@/api/metrics';
import { bytesToMB } from '@/utils/byteConverter';
import Plotly from 'plotly.js-dist-min';
import { Terminal } from 'xterm';
import 'xterm/css/xterm.css';
import { io } from 'socket.io-client';

export default {
    components: { AppHeader },
    data() {
        return {
            data: {
                serverUrl: '',
                appName: '',
                name: '',
                pid: 0,
                status: 'offline',
                uptime: 0,
                restarts: 0,
                memory: 0,
                cpu: 0,
                heap_size: '0',
                heap_usage: 0,
                event_loop_latency: '0',
                active_handles: 0,
                active_requests: 'N/A',
            },
            ranged: false,
            rangePresets: [7, 14, 30, 90].map(days => ({
                label: `Останні ${days} днів`,
                value: [dayjs().subtract(days, 'd'), dayjs()],
            })),
            chartData: {
                cpu: { time: [], value: [] },
                memory: { time: [], value: [] },
            },
            intervalIds: [],
            terminal: null,
            socket: null,
        };
    },
    async mounted() {
        this.data.appName = this.$route.params.appName;
        await this.fetchServerName();
        await this.fetchProcessData();
        await this.fetchChartData();
        this.intervalIds.push(setInterval(this.reloadChart, 5000))
```

```
        this.intervalIds.push(setInterval(this.fetchProcessData, 1000));
        this.initializeTerminal();
    },
    beforeUnmount() {
        if (this.intervalIds.length > 0) {
            this.intervalIds.forEach(id => clearInterval(id));
        };
        if (this.socket) this.socket.disconnect();
        if (this.terminal) this.terminal.dispose();
    },
    methods: {
        async fetchServerName() {
            try {
                const data = await getByName(this.$route.params.name);
                this.data.serverUrl = data.url;
            } catch (error) {
                console.error('Error fetching server name:', error);
            }
        },
        async fetchProcessData() {
            try {
                const response = await getPm2App(this.data.serverUrl, this.data.appName);
                this.data = { ...this.data, ...response };
            } catch (error) {
                message.error('Помилка при отриманні даних застосунку');
                console.error('Error fetching process data:', error);
            }
        },
        async startProcess() {
            try {
                message.loading({ content: 'Запуск процесу...', key: 'start' });
                await startApp(this.data.serverUrl, this.data.name);
                this.data.status = 'online';
                message.success({ content: `Процес ${this.data.name} запущено`, key: 'start', duration: 2 });
            } catch (error) {
                void error;
                message.error({ content: `Помилка запуску процесу ${this.data.name}`, key: 'start', duration: 2
});
            }
        },
        confirmRestart() {
            Modal.confirm({
                title: 'Підтвердження перезапуску',
                content: `Ви впевнені, що хочете перезапустити процес ${this.data.name}?`,
                okText: 'Перезапустити',
                okType: 'primary',
                cancelText: 'Скасувати',
                onOk: async () => {
                    await this.restartProcess();
                },
            });
        },
        async restartProcess() {
            try {
                message.loading({ content: 'Перезапуск процесу...', key: 'restart' });
                await restartApp(this.data.serverUrl, this.data.name);
                this.data.restarts += 1;
                this.data.status = 'online';
                message.success({ content: `Процес ${this.data.name} перезапущено`, key: 'restart', duration: 2
});
            } catch (error) {
                void error;
                message.error({ content: `Помилка перезапуску процесу ${this.data.name}`, key: 'restart',
duration: 2 });
            }
        },
        confirmStop() {
            Modal.confirm({
                title: 'Підтвердження зупинки',
                content: `Ви впевнені, що хочете зупинити процес ${this.data.name}?`,
                okText: 'Зупинити',
                okType: 'danger',
                cancelText: 'Скасувати',
                onOk: async () => {
                    await this.stopProcess();
```

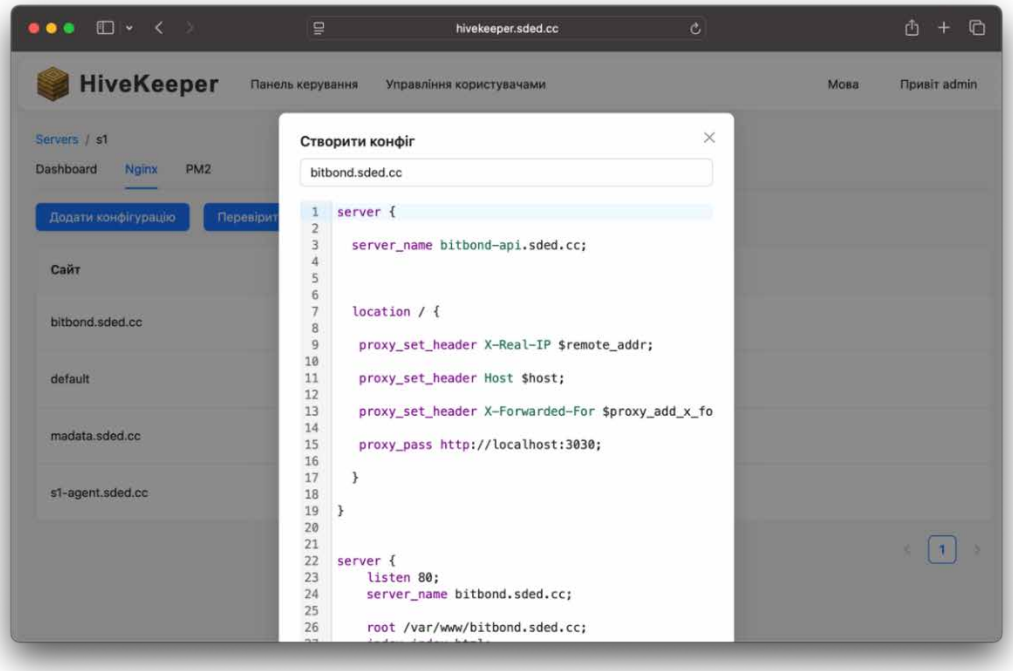
```
                },
            });
        },
        async stopProcess() {
            try {
                message.loading({ content: 'Зупинка процесу...', key: 'stop' });
                await stopApp(this.data.serverUrl, this.data.name);
                this.data.status = 'offline';
                message.success({ content: `Процес ${this.data.name} зупинено`, key: 'stop', duration: 2 });
            } catch (error) {
                void error;
                message.error({ content: `Помилка зупинки процесу ${this.data.name}`, key: 'stop', duration: 2
});
            }
        },
        formatMemory(bytes) {
            const mb = bytesToMB(bytes);
            return `${mb.toFixed(2)} МБ`;
        },
        formatUptime(timestamp) {
            const now = Date.now();
            const diff = now - timestamp;

            if (diff < 0) return "У майбутньому";

            const seconds = Math.floor(diff / 1000);
            const days = Math.floor(seconds / (3600 * 24));
            const hours = Math.floor((seconds % (3600 * 24)) / 3600);
            const minutes = Math.floor((seconds % 3600) / 60);

            return `${days}д ${hours}г ${minutes}хв`;
        },
        async fetchChartData(start = dayjs().subtract(5, 'm'), end = dayjs()) {
            try {
                const data = await getAppByDate(this.data.serverUrl, this.data.appName, start, end);
                this.chartData.cpu.time = data.cpu.map(item => item.time);
                this.chartData.cpu.value = data.cpu.map(item => item.value);
                this.chartData.memory.time = data.memory.map(item => item.time);
                this.chartData.memory.value = data.memory.map(item => bytesToMB(item.value).toFixed(0));
                this.createChart();
            } catch (error) {
                console.error('Error fetching chart data:', error);
            }
        },
        async onRangeChange(dates) {
            this.ranged = dates ? true : false;
            await this.fetchChartData(
                dates ? dates[0].toISOString() : undefined,
                dates ? dates[1].toISOString() : undefined
            );
        },
        reloadChart() {
            if (!this.ranged) {
                this.fetchChartData();
            }
        },
        createChart() {
            Plotly.newPlot(this.$refs.cpuChart, [{
                x: this.chartData.cpu.time,
                y: this.chartData.cpu.value,
                type: 'scatter',
                name: 'CPU Load',
                line: { color: '#2a7abf' },
                fill: 'tozeroy'
            }], {
                title: { text: 'CPU Usage' },
                xaxis: { type: 'date', tickformat: '%Y-%m-%d %H:%M:%S' },
                yaxis: { title: 'CPU %', range: [0, 100] }
            });

            Plotly.newPlot(this.$refs.ramChart, [{
                x: this.chartData.memory.time,
                y: this.chartData.memory.value,
                type: 'scatter',
                name: 'RAM Usage',
```

```
                line: { color: '#48bf2a' },
                fill: 'tozeroy'
            }], {
                title: { text: 'RAM Usage' },
                xaxis: { type: 'date', tickformat: '%Y-%m-%d %H:%M:%S' },
                yaxis: { title: 'RAM (MB)' }
            });
        },
        initializeTerminal() {
            this.terminal = new Terminal({
                cols: 80,
                rows: 24,
                scrollback: 1000,
                fontFamily: "'Fira Mono', monospace",
                theme: {
                    background: '#1e1e1e',
                    foreground: '#ffffff',
                },
            });

            this.terminal.open(this.$refs.terminalContainer);
            this.terminal.writeln('Підключення до сервера...');

            this.socket = io('http://vpn.sded.cc:3001', {
                transports: ['websocket', 'polling'],
                withCredentials: true,
            });

            this.socket.on('connect', () => {
                this.terminal.writeln(`Підключено: ${this.socket.id}`);
            });

            this.socket.on('pm2Log', (log) => {
                this.terminal.writeln(log);
            });

            this.socket.on('pm2Error', (msg) => {
                this.terminal.writeln(`\x1b[31mПомилка PM2: ${msg}\x1b[0m`);
            });

            this.socket.on('disconnect', () => {
                this.terminal.writeln('Відключено від сервера');
            });
        },
        subscribeLogs() {
            if (this.data.appName.trim() === '') {
                this.terminal.writeln('\x1b[33mВведіть назву або ID застосунку\x1b[0m');
                return;
            }
            this.terminal.clear();
            this.terminal.writeln(`Підписка на логи застосунку: ${this.data.appName.trim()}`);
            this.socket.emit('subscribeLogs', this.data.appName.trim());
        },
    },
};
</script>

<style scoped>
.app-viewer {
    padding: 16px;
}

.button-group {
    display: flex;
    justify-content: flex-start;
    gap: 16px;
    margin-bottom: 24px;
}

.info-panel {
    margin-top: 16px;
    max-width: 1900px;
}

a-card {
```

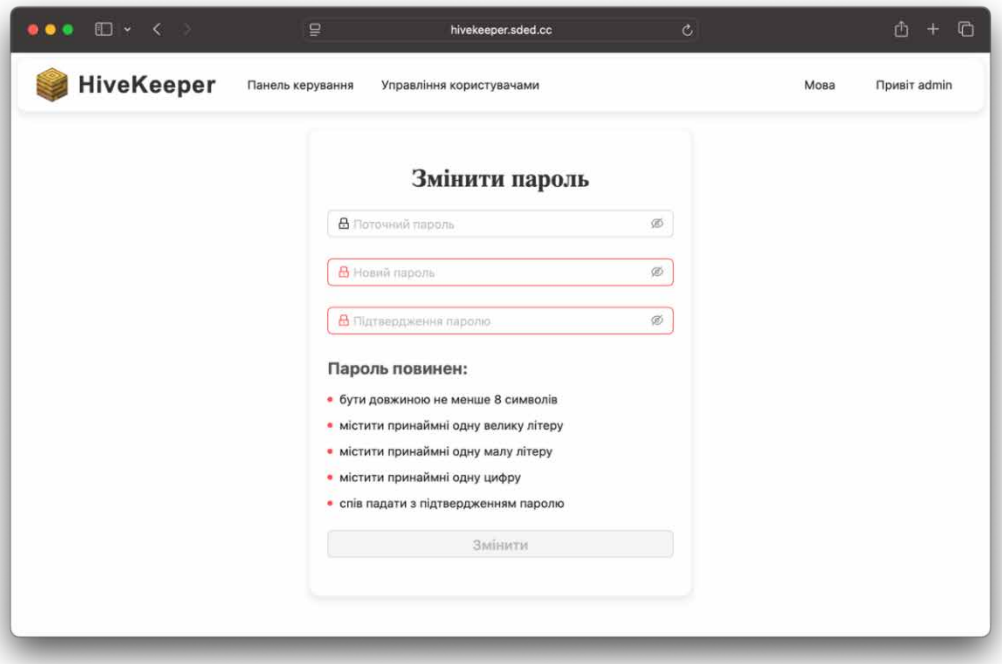
```
    border-radius: 8px;
    box-shadow: 0 2px 8px rgba(0, 0, 0, 0.1);
}

.date-picker-container {
    margin-bottom: 16px;
}

.log-controls {
    display: flex;
    align-items: center;
    margin-bottom: 12px;
}

.terminal {
    width: 100%;
    max-width: 1900px;
    height: 400px;
    border: 1px solid #333;
    border-radius: 4px;
    background: #1e1e1e;
    color: #fff;
    font-family: monospace;
}
</style>
```

ДОДАТОК Д

Створення Nginx конфігурації



Зміна паролю користувачем



Керування користувачами та їх правами