

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«Панель керування серверами»

Виконав студент групи ІІЗ-22-3
ЧЕРНОШТАН Тимур Юрійович

Керівник роботи:
МОРОЗОВ Андрій Васильович

Рецензент:
ЧИЖМОТРЯ Олексій Володимирович

Житомир – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення

Тетяна Вакалюк

«28» лютого 2026 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ЧЕРНОШТАН Тимур Юрійович**

Керівник роботи: **МОРОЗОВ Андрій Васильович**

Тема роботи: **«Панель керування серверами»,**

затверджена Наказом закладу вищої освіти від **«27» лютого 2026 р., № 92/с**

Вихідні дані для роботи: **процеси розгортання та моніторингу програмного забезпечення у контейнеризованих середовищах на базі технології Docker, методи та інструментальні засоби створення веб-систем керування Docker-контейнерами на платформі .NET.**

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Морозов А.В.	01.03.2026р.	02.03.2026р
2	Морозов А.В.	03.04.2026р	04.04.2026р
3	Морозов А.В.	02.05.2026р	03.05.2026р

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з програмного комплексу та пояснювальної записки. Пояснювальна записка до випускної роботи містить текстову частину, викладену на 89 сторінках друкованого тексту. Список використаних джерел містить 30 найменувань і займає 3 сторінки. У роботі наведено 25 рисунків та 5 таблиць. Також у роботі є 11 сторінок додатків. Загальний обсяг роботи — 100 сторінок.

Метою роботи є проєктування та розробка веб-орієнтованої мультитенантної системи керування інфраструктурою Docker у реальному часі, що забезпечує ізольоване управління контейнерами, мережами, томами та стеками для кількох користувачів через графічний інтерфейс з підтримкою моніторингу системних ресурсів.

У роботі використано методи об'єктно-орієнтованого проєктування, структурного аналізу предметної області, UML-моделювання та автоматизованого тестування програмного забезпечення.

Розроблений програмний комплекс реалізує повний життєвий цикл керування Docker-контейнерами, розгортання стеків на основі Docker Compose, керування мережами та томами, моніторинг CPU та оперативної пам'яті у реальному часі, інтерактивний термінал у браузері, файловий менеджер контейнера, систему тікетів підтримки та журнал дій користувачів. Мультитенантна архітектура забезпечує ізоляцію ресурсів між користувачами з індивідуальними квотами та виділеними діапазонами портів. Результати роботи можуть бути використані в організаціях, що застосовують контейнеризацію для розгортання та адміністрування програмного забезпечення.

КЛЮЧОВІ СЛОВА: DOCKER, КОНТЕЙНЕРИЗАЦІЯ, МОНІТОРИНГ, REST API, .NET, SIGNALR, ОРКЕСТРАЦІЯ, МУЛЬТИТЕНАНТНІСТЬ.

					ІПЗ.КР.Б-121 – 26 – ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата				
Розроб.		Черноштан Т.Ю.			Панель керування серверами Пояснювальна записка	Літ.	Арк.	Аркушів
Керівник		Морозов А.В.					3	100
						ФІКТ, гр.ІПЗ-22-3		
Рецензент.		Чижмотря О.В.						
Зав. каф.		Вакалюк Т.А.						

ABSTRACT

The bachelor's qualification work consists of a software package and an explanatory note. The explanatory note contains a text section comprising 89 pages of printed text. The list of references contains 30 entries and occupies 3 pages. The work includes 25 figures and 5 tables. It also contains 11 pages of appendices. The total length of the work is 100 pages.

The aim of the work is to design and develop a web-oriented multi-tenant system for real-time management of Docker infrastructure that provides isolated management of containers, networks, volumes, and stacks for multiple users through a graphical interface with system resource monitoring.

The work employs methods of object-oriented design, structural analysis of the subject domain, UML modeling, and automated software testing.

The developed software package implements the full lifecycle of Docker container management, deployment of stacks based on Docker Compose, management of networks and volumes, real-time monitoring of CPU and RAM usage, an interactive in-browser terminal, a container file manager, a support ticket system, and a user action audit log. The multi-tenant architecture ensures reliable resource isolation between users with individual quotas and allocated port ranges. The results of this work can be used by organizations that apply containerization for software deployment and administration.

KEYWORDS: DOCKER, CONTAINERIZATION, MONITORING, REST API, .NET, SIGNALR, ORCHESTRATION, MULTI-TENANCY.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		4

ЗМІСТ

ВСТУП.....	7
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ЗАСОБІВ КЕРУВАННЯ СЕРВЕРНОЮ ІНФРАСТРУКТУРОЮ	10
1.1. Постановка задачі	10
1.2. Аналіз наявних аналогів програмного комплексу	13
1.3. Визначення функціоналу системи	20
Висновки до розділу 1	24
РОЗДІЛ 2. ТЕХНОЛОГІЧНИЙ СТЕК ТА ЗАСОБИ РЕАЛІЗАЦІЇ ПРОГРАМНОГО КОМПЛЕКСУ	26
2.1. Технологічний стек.....	26
2.2. Архітектура та патерни проєктування.....	34
2.3. Структура бази даних	42
2.4. Реалізація механізмів безпеки та мультитенантної ізоляції.....	48
2.5. Реалізація ключових модулів системи.....	58
Висновки до розділу 2	62
РОЗДІЛ 3. РОЗГОРТАННЯ ТА ЕКСПЛУАТАЦІЯ СИСТЕМИ.....	64
3.1. Розгортання та автоматизація доставки	64
3.2. Опис інтерфейсу користувача	68
3.3. Тестування програмного комплексу	80
Висновки до розділу 3	83
ВИСНОВКИ	85
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	87
ДОДАТКИ.....	90

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API — програмний інтерфейс застосунку (Application Programming Interface)

BCrypt — адаптивний алгоритм хешування паролів

CI/CD — безперервна інтеграція та безперервна доставка (Continuous Integration / Continuous Delivery)

CORS — спільне використання ресурсів між джерелами (Cross-Origin Resource Sharing)

CPU — центральний процесор (Central Processing Unit)

DI — впровадження залежностей (Dependency Injection)

DTO — об'єкт передачі даних (Data Transfer Object)

IDOR — небезпечне пряме посилання на об'єкт (Insecure Direct Object Reference)

JWT — веб-токен JSON (JSON Web Token)

ORM — об'єктно-реляційне відображення (Object-Relational Mapping)

REST — передача репрезентативного стану (Representational State Transfer)

SPA — односторінковий застосунок (Single Page Application)

SQL — мова структурованих запитів (Structured Query Language)

ТОСТОУ — стан гонки між перевіркою та використанням (Time-of-Check to Time-of-Use)

UI — інтерфейс користувача (User Interface)

UML — уніфікована мова моделювання (Unified Modeling Language)

YAML — формат серіалізації даних (YAML Ain't Markup Language)

БД — база даних

ОС — операційна система

ПЗ — програмне забезпечення

СКБД — система керування базами даних

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		6

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується масовим переходом від монолітної архітектури програмного забезпечення до мікросервісної, де ключову роль відіграє технологія контейнеризації. Контейнеризація дозволяє ізолювати програмні середовища, забезпечувати відтворюваність розгортання та ефективно використовувати обчислювальні ресурси серверної інфраструктури. Платформа Docker набула статусу галузевого стандарту у сфері контейнеризації та широко застосовується як у комерційних організаціях, так і в наукових установах для розгортання та супроводу програмних систем [1].

Зі зростанням кількості контейнерів складність їх адміністрування зростає суттєво. Управління контейнерною інфраструктурою засобами інтерфейсу командного рядка створює високий поріг входу для фахівців, підвищує ризик виникнення суб'єктивних помилок адміністрування та унеможливляє оперативне реагування на відхилення у роботі системи. Відсутність наочної візуалізації стану контейнерів та показників використання ресурсів у реальному часі ускладнює своєчасне виявлення проблем і може призводити до відмов у роботі сервісів.

Наявні графічні рішення для керування Docker, зокрема Portainer та CasaOS, або мають надлишковий функціонал для типових сценаріїв використання, або вимагають значних ресурсів для розгортання, або надають розширені можливості лише в платних версіях. Це обумовлює потребу у розробці оптимізованої веб-орієнтованої системи, що фокусується на моніторингу ресурсів у реальному часі та забезпечує оперативне керування контейнерами через інтуїтивно зрозумілий інтерфейс.

Окремою проблемою є організація спільного використання єдиного Docker-хоста кількома користувачами. У типових сценаріях навчального процесу або невеликих команд розробки на одному сервері працює кілька фахівців, кожен з яких має керувати власними контейнерами без доступу до ресурсів інших користувачів. Наявні безкоштовні рішення не забезпечують

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				7
Змн.	Арк.	№ докум.	Підпис	Дата		

повноцінної мультитенантної ізоляції з розмежуванням ресурсів, квотуванням та виділенням окремих діапазонів портів для кожного користувача, що створює ризики несанкціонованого доступу та конфліктів за ресурси.

Важливість моніторингу ресурсів саме у реальному часі пояснюється тим, що показники CPU та оперативної пам'яті змінюються динамічно, і затримка в отриманні даних може унеможливити своєчасне реагування на критичні ситуації. Застосування технології WebSocket дозволяє організувати двосторонній канал зв'язку між сервером і клієнтом без необхідності постійного опитування сервера, що забезпечує мінімальну затримку при передачі даних моніторингу [14].

Метою роботи є проектування та розробка веб-орієнтованої мультитенантної системи керування інфраструктурою Docker у реальному часі, що забезпечує ізольоване управління контейнерами, стеками, мережами та томами для кількох користувачів через графічний інтерфейс з підтримкою моніторингу системних ресурсів та засобів безпеки.

Для досягнення поставленої мети визначено такі завдання: провести аналіз предметної області та наявних засобів керування Docker-інфраструктурою; визначити функціональні та нефункціональні вимоги до програмного комплексу та обґрунтувати вибір технологічного стеку; спроектувати архітектуру системи, структуру бази даних та розробити UML-моделі; реалізувати серверну частину на основі ASP.NET Core з використанням Docker.DotNet та SignalR; забезпечити мультитенантну ізоляцію ресурсів з квотуванням та механізмами безпеки; реалізувати розгортання стеків Docker Compose, керування мережами та томами; розробити клієнтську частину у вигляді односторінкового застосунку на React; провести автоматизоване тестування розробленого програмного комплексу.

Об'єктом дослідження є процес керування контейнеризованою серверною інфраструктурою на базі Docker.

Предметом дослідження є методи та засоби розробки веб-орієнтованих систем керування Docker-контейнерами з підтримкою моніторингу ресурсів у реальному часі.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		8

У роботі використано такі методи дослідження: системний аналіз предметної області, об'єктно-орієнтоване проєктування, UML-моделювання для формалізації вимог та структури системи, а також функціональне тестування для перевірки коректності реалізованих алгоритмів.

Практичне значення отриманих результатів полягає у тому, що розроблений програмний комплекс може бути використаний в організаціях, що застосовують контейнеризацію для розгортання та адміністрування програмного забезпечення, а також у навчальному процесі для демонстрації принципів роботи з Docker API.

Структура та обсяг роботи. Кваліфікаційна робота складається з вступу, трьох розділів, висновків, списку використаних джерел із 30 найменувань, 5 додатків (11 стор.), 5 таблиць та 25 рисунків по тексту. Повний обсяг роботи становить 100 сторінок, з них 80 сторінок основного тексту. Список використаних джерел оформлено відповідно до вимог ДСТУ 8302:2015 [30].

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		9

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ЗАСОБІВ КЕРУВАННЯ СЕРВЕРНОЮ ІНФРАСТРУКТУРОЮ

1.1. Постановка задачі

Сучасна серверна інфраструктура підприємств, організацій та закладів вищої освіти дедалі частіше будується на основі контейнеризованих середовищ. Технологія Docker забезпечує надійну ізоляцію програмних компонентів, спрощує процес розгортання застосунків та дозволяє ефективно використовувати обчислювальні ресурси сервера [7]. Проте зі зростанням кількості контейнерів управління такою інфраструктурою стає складнішим завданням, яке суттєво ускладнюється за умов необхідності спільного використання одного фізичного або віртуального Docker-хоста кількома незалежними користувачами. Це зумовлює потребу в автоматизації адміністрування та наданні зручного безпечного графічного інструменту з чітким розмежуванням прав та ізоляцією ресурсів.

Архітектурно Docker складається з клієнтської частини та системного демона Docker Daemon, який виконує всі низькорівневі операції з контейнерами, образами, мережами та томами. Взаємодія між клієнтом і демоном відбувається через REST API, доступне через локальний Unix-сокет «/var/run/docker.sock» або через захищений мережевий інтерфейс [12]. Інтерфейс командного рядка Docker CLI є лише високорівневою обгорткою над цим API і не є єдиним можливим засобом керування [11]. Використання Docker API безпосередньо через веб-застосунок дозволяє усунути зайві ланки у процесі адміністрування, зокрема необхідність встановлення SSH-з'єднання з сервером для виконання кожної операції, проте вимагає створення надійного проміжного шару, що здійснює валідацію та фільтрацію запитів для запобігання деструктивним діям у мультитенантному середовищі.

На сьогодні основним інструментом керування Docker-інфраструктурою залишається інтерфейс командного рядка. Адміністратори та розробники змушені вручну вводити команди для запуску, зупинки, видалення або інспектування об'єктів. Такий підхід вимагає від фахівців глибоких знань

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		10

синтаксису, є трудомістким і підвищує ризик виникнення суб'єктивних помилок. Будь-яка помилка у введенні команди може призвести до зупинки критично важливих сервісів або незворотної втрати даних. У контексті колективного використання сервера ситуація ускладнюється повною відсутністю в межах Docker CLI механізмів штатної ізоляції користувачів, що робить неможливим безпечне надання доступу до командного рядка студентам або різними командами розробників одночасно.

Окремою проблемою є моніторинг стану обчислювальних ресурсів. Стандартна команда «docker stats» виводить текстовий потік з поточними показниками використання центрального процесора та оперативної пам'яті, однак має суттєві обмеження з практичної точки зору. По-перше, текстовий формат виведення є незручним для аналізу динаміки змін показників у часі. По-друге, команда не зберігає історію метрик, що унеможливорює побудову графіків та виявлення тенденцій у споживанні ресурсів. По-третє, при одночасному моніторингу великої кількості контейнерів виведення займає значну частину екрану термінала і ускладнює паралельну роботу фахівця. У мультитенантних системах відсутність оперативної візуалізації метрик та автоматичного контролю лімітів підвищує ризик виникнення ситуацій, коли один користувач може повністю вичерпати ресурси всього хоста, спричинивши відмову в обслуговуванні для інших.

Найбільш критичним аспектом є забезпечення інформаційної безпеки при наданні доступу до спільної інфраструктури. Надання прямого доступу до сокета Docker «/var/run/docker.sock» фактично рівнозначне отриманню прав суперкористувача на хост-системі, оскільки дозволяє монтувати будь-які директорії сервера та виконувати довільний код [9]. Наявні інструменти адміністрування часто не приділяють достатньої уваги загрозам несанкціонованого доступу до чужих ресурсів через маніпуляцію ідентифікаторами, ризикам станів гонки при динамічній перевірці квот, а також небезпеці розгортання неперевіраних конфігурацій Docker Compose. Існує потреба в автоматичному блокуванні привілейованих режимів, небезпечних

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		11

мережевих налаштувань та несанкціонованих монтувань директорій хоста. Окрім цього, операції з файлами всередині контейнерів через графічний інтерфейс потребують суворих обмежень для запобігання виходу за межі дозволених директорій.

З урахуванням наведених проблем у межах кваліфікаційної роботи визначено комплекс задач, що підлягають автоматизації та реалізації у проєктованій системі.

Першою задачею є побудова мультитенантної архітектури з надійною автентифікацією на основі JWT-токенів з підтримкою механізму інвалідації сесій через версіонування ролей, а також впровадження суворої ізоляції даних та ресурсів, що включає контроль лімітів оперативної пам'яті й процесора та розподіл персональних діапазонів портів для запобігання мережевим конфліктам між користувачами [25].

Другою задачею є реалізація повного життєвого циклу керування контейнерами з валідацією логіки переходів між станами.

Третьою задачею є автоматизація парсингу, посилення безпеки конфігурацій та розгортання багатокomпонентних стеків Docker Compose з організацією потокового передавання логів розгортання.

Четвертою задачею є управління базовими примітивами Docker-ізольованими мережами та томами, а також створення конфігураційних шаблонів.

П'ятою задачею є забезпечення двостороннього обміну даними у реальному часі за допомогою протоколу WebSocket для моніторингу метрик ресурсів, функціонування інтерактивного веб-терміналу та захищеного файлового менеджера, що працює через Docker Archive TAR API.

Шостою задачею є розробка адміністративного контуру, що включає наскрізний аудит дій користувачів, приховування системних ресурсів самої панелі від загального перегляду та інтегровану систему сервісних тікетів технічної підтримки для взаємодії між користувачами та адміністратором.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		12

Розв'язання зазначених задач дозволить створити безпечне та ізольоване середовище для колективного використання обчислювальних потужностей одного сервера, суттєво знизити поріг входу для фахівців, мінімізувати вплив людського фактора та підвищити оперативність реагування на критичні зміни стану системи. Результатом виконання роботи є програмний комплекс — веб-орієнтована мультитенантна панель керування серверами ServerPanel, що реалізує наведений перелік функцій засобами сучасного технологічного стеку на базі платформи .NET 8 та бібліотеки React 19.

1.2. Аналіз наявних аналогів програмного комплексу

Перед проектуванням власного програмного комплексу проведено детальний порівняльний аналіз наявних на ринку рішень для керування серверною інфраструктурою. Метою цього аналізу є визначення стандартного функціоналу систем такого класу, виявлення їхніх архітектурних і безпекових недоліків у контексті колективного використання ресурсів, а також обґрунтування доцільності розробки власного рішення. Для аналізу обрано три найбільш репрезентативні програмні продукти, що використовуються у сучасній практиці адміністрування: Portainer, Pterodactyl та CasaOS.

Portainer є одним з найпоширеніших графічних інструментів керування Docker-інфраструктурою з відкритим вихідним кодом [1]. Серверна частина системи реалізована мовою Go, що забезпечує високу продуктивність та мінімальне споживання ресурсів самою панеллю керування. Клієнтська частина побудована на основі фреймворку Angular і взаємодіє з бекендом через REST API. Система розповсюджується у двох версіях: безкоштовній Community Edition та комерційній Business Edition з розширеним функціоналом.

З архітектурної точки зору Portainer підтримує два режими підключення до Docker. У першому режимі панель підключається безпосередньо до Unix-сокета «/var/run/docker.sock» на локальному вузлі, що є найпростішим способом розгортання. У другому режимі для керування віддаленими серверами використовується окремий компонент Portainer Agent, який встановлюється на

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		13

кожному вузлі та забезпечує захищену комунікацію з центральною панеллю. Такий підхід дозволяє керувати кількома серверами з єдиного інтерфейсу, однак вимагає додаткового налаштування мережевого доступу та встановлення агентів на кожному обчислювальному вузлі.

Інтерфейс Portainer побудований на основі класичних табличних списків з ієрархічною навігацією. Користувач послідовно переходить між рівнями: середовище — стек — контейнер — деталі. Такий підхід є інформативним для досвідчених адміністраторів, однак є надлишково складним для звичайних користувачів. Найбільшим недоліком безкоштовної версії Portainer Community Edition є відсутність повноцінного моніторингу ресурсів у реальному часі та механізмів мультитенантного квотування. Хоча в системі наявне розмежування користувачів на рівні доступу до середовищ, версія Community Edition не дозволяє встановлювати жорсткі обмеження на обсяг оперативної пам'яті чи процесорного часу для окремих облікових записів. Моніторинг у Portainer Community Edition надає лише статичне відображення поточних показників, базуючись на HTTP-опитуванні з фіксованим інтервалом, що створює постійне навантаження на сервер. Головне вікно Community Edition побудовано на основі табличної структури для відображення списку контейнерів з їх поточним статусом, що наочно демонструє (рис. 1.1).

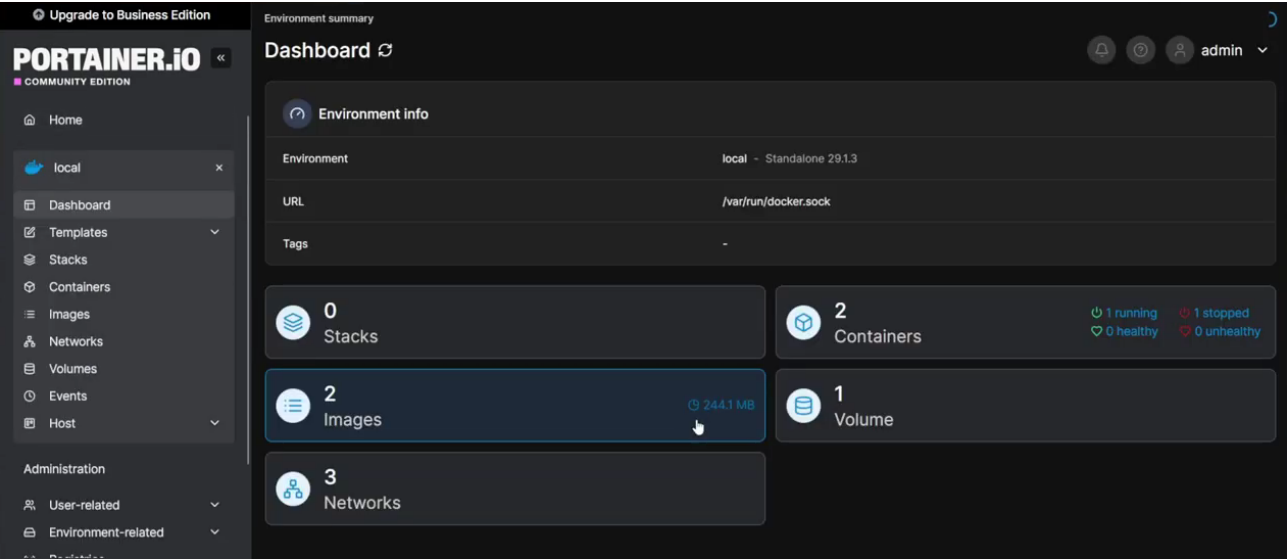


Рис. 1.1. Головний інтерфейс Portainer Community Edition

Pterodactyl є відкритою панеллю керування серверами, що використовує Docker-контейнери як ізольоване середовище для кожного серверного процесу. Архітектура системи є двокomпонентною: основна панель керування реалізована на основі фреймворку Laravel (PHP) і відповідає за веб-інтерфейс, автентифікацію та API, тоді як окремий компонент Wings, реалізований мовою Go, встановлюється безпосередньо на кожному вузлі та відповідає за безпосередню взаємодію з Docker Engine. Така архітектура забезпечує гнучкість у масштабуванні, однак суттєво ускладнює процес встановлення та налаштування системи.

Встановлення Pterodactyl є трудомістким процесом, що вимагає налаштування значної кількості компонентів: веб-сервера Nginx або Apache, бази даних MySQL, менеджера черг Redis, а також окремого встановлення та налаштування компонента Wings на кожному вузлі. Система орієнтована на специфічний сценарій використання або розгортання та керування ігровими серверами — і не є універсальним інструментом керування Docker-інфраструктурою. Мультитенантність у Pterodactyl реалізована на високому рівні, проте вона є жорстко прив'язаною до концепції ігрових серверів через шаблони конфігурацій. Користувач позбавлений можливості керувати загальними примітивами Docker: він не може самостійно створити довільну Docker-мережу, змонтувати сторонній Docker-том, розгорнути чистий багатоконтейнерний стек Docker Compose або завантажити довільний образ з Docker Hub поза межами наявних жорстких шаблонів. Pterodactyl надає зручний інтерфейс для перегляду логів та керування файлами сервера у реальному часі через WebSocket-з'єднання, однак не забезпечує повноцінного графічного моніторингу динаміки зміни метрик процесора та оперативної пам'яті у вигляді часових графіків. Інтерфейс керування серверами у Pterodactyl фокусується на виведенні консольних логів та базових статистичних даних, як показано (рис. 1.2).

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		15

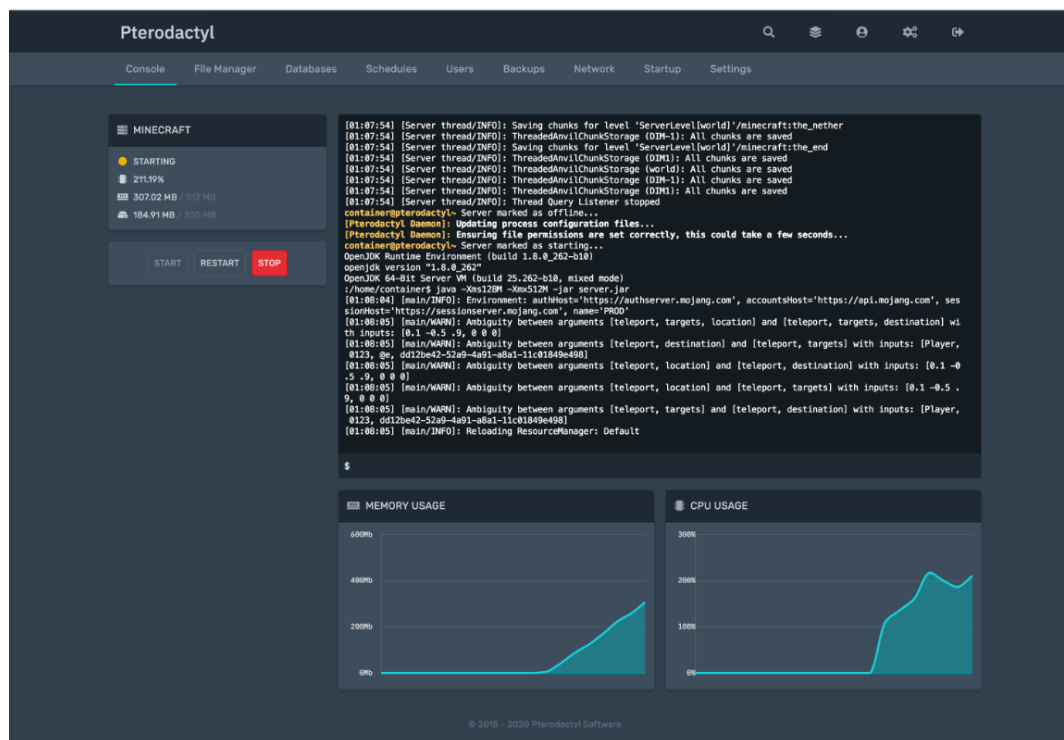


Рис. 1.2. Інтерфейс панелі керування Pterodactyl

CasaOS позиціонується як операційна система для домашнього сервера з підтримкою розгортання застосунків через Docker. Серверна частина реалізована мовою Go, клієнтська — на основі фреймворку Vue.js. Система орієнтована на кінцевого споживача і забезпечує зручний інтерфейс у стилі мобільної операційної системи з віджетами для кожного встановленого застосунку. Встановлення CasaOS є найпростішим серед розглянутих аналогів і здійснюється за допомогою одного сценарію командного рядка.

З архітектурної точки зору CasaOS побудована за принципом єдиного вузла і розрахована виключно на персональне локальне використання. Система повністю позбавлена інструментів мультитенантності: у ній відсутнє поняття керування користувачами, розподілу прав доступу або ізоляції даних - керування обмежується єдиним глобальним обліковим записом адміністратора. Моніторинг стану системи реалізований через періодичне HTTP-опитування з інтервалом у кілька секунд, що не забезпечує відображення метрик у режимі реального часу в розумінні миттєвого оновлення даних. Розширені інструменти безпеки, зокрема обмеження прав доступу до Docker-сокета чи контроль монтувань, у системі не передбачені, оскільки вона розрахована на роботу у

		Черноштан Т.Ю.			Арк.
		Морозов А.В.			
Змн.	Арк.	№ докум.	Підпис	Дата	16

довіреній домашній мережі. Дизайн CasaOS базується на використанні віджетів для швидкого доступу до встановлених застосунків та візуалізації загального стану системи, що продемонстровано (рис. 1.3).

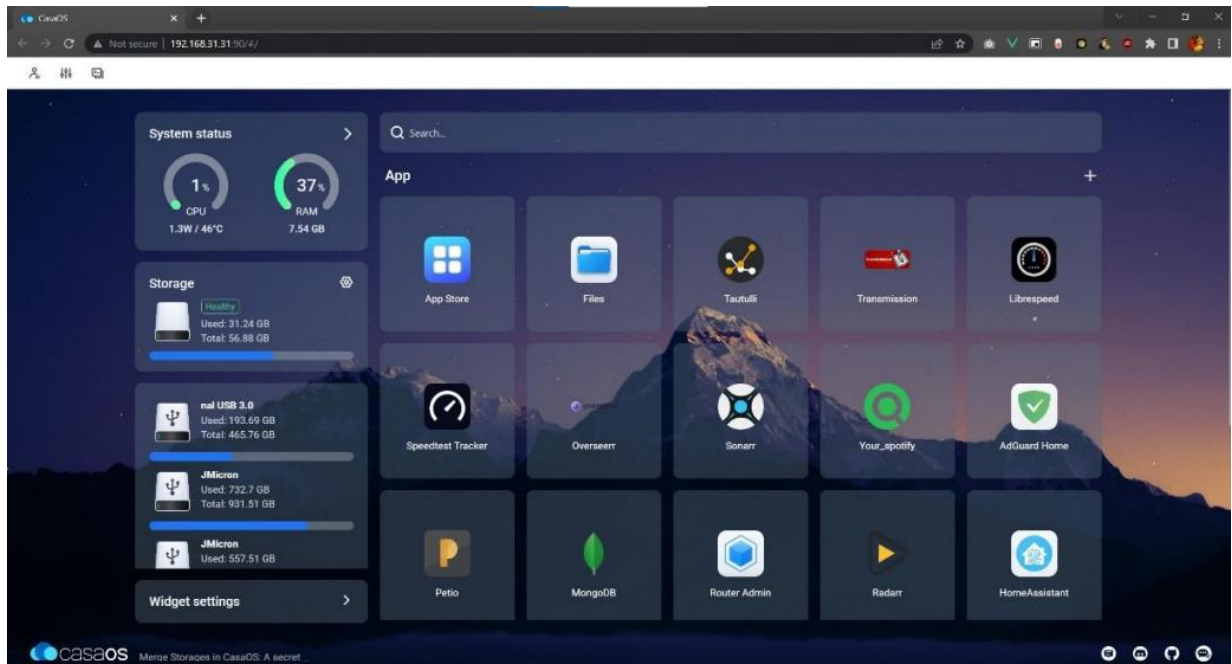


Рис. 1.3. Головне вікно CasaOS

Окремої уваги заслуговує порівняння методів отримання даних моніторингу та логів між розглянутими системами. Більшість наявних рішень, зокрема Portainer Community Edition та CasaOS, використовують підхід HTTP-опитування, за якого клієнт з певним інтервалом надсилає запити до сервера для отримання актуальних даних. Такий підхід створює надмірне навантаження на веб-сервер при збільшенні кількості контейнерів та клієнтських з'єднань, а також не забезпечує миттєвого оновлення показників. Натомість використання дуплексного протоколу WebSocket дозволяє організувати постійне двостороннє з'єднання між сервером і клієнтом, за якого сервер самостійно надсилає оновлені дані в момент їх генерації демоном Docker без необхідності постійних запитів з боку веб-браузера [14]. Бібліотека SignalR, що використовується у власній розробці, бере на себе керування транспортними протоколами і забезпечує мінімальну затримку при передачі динамічних даних.

Для комплексного обґрунтування розробки виконано порівняльний аналіз розглянутих систем за розширеним набором критеріїв, що враховує вимоги до

безпеки, мультитенантності та повноти підтримки Docker-інфраструктури. Результати аналізу наведено в табл. 1.1.

Таблиця 1.1

Порівняльний аналіз аналогів програмного комплексу

Критерій	Portainer CE	Pterodactyl	CasaOS	Власна розробка
Відкритий вихідний код	Так	Так	Так	Так
Мультитенантна ізоляція даних користувачів	Обмежено	Частково (ігрова)	Не передбачено	Так
Сумарне per-user квотування ресурсів	Лише per-container	Так	Не передбачено	Так
Автоматичне виділення діапазонів портів	Не передбачено	Не передбачено	Не передбачено	Так
Керування стеками Docker Compose	Так	Не передбачено	Обмежено	Так
Керування Docker-мережами та томами	Так	Не передбачено	Обмежено	Так
Посилення безпеки конфігурацій Compose	Не передбачено штатно	Не передбачено	Не передбачено	Так
Моніторинг CPU/RAM у реальному часі	Лише у платній версії	Лише текстовий	Обмежено	Так

Продовження таблиці 1.1

Критерій	Portainer CE	Pterodactyl	CasaOS	Власна розробка
Файловий менеджер та веб-термінал	Так	Лише ігровий контур	Обмежено	Так
Метод отримання динамічних даних	HTTP-опитування	WebSocket	HTTP-опитування	WebSocket (SignalR)
Складність розгортання	Середня	Висока	Низька	Низька
Система тікетів підтримки	Не передбачено	Не передбачено	Не передбачено	Так

Проведений аналіз засвідчує, що наявні рішення мають суттєві обмеження для сценаріїв колективного використання єдиного Docker-хоста, де найважливішими пріоритетами є сувора ізоляція даних, автоматичний розподіл мережевих портів, безпекове обмеження конфігурацій Docker Compose та легкий моніторинг обчислювальних ресурсів. Portainer надає базові засоби розмежування користувачів та команд у безкоштовній версії, однак повноцінна рольова модель з ієрархією прав та інструменти примусового квотування ресурсів доступні виключно у комерційній версії Business Edition. Pterodactyl є вузькоспеціалізованим рішенням і непридатним для повноцінного розгортання стандартної інфраструктури Docker, зокрема довільних мереж, томів та стеків поза межами ігрових шаблонів. CasaOS не передбачає засобів багатокористувацької роботи та розмежування доступу, оскільки орієнтований на персональне використання.

Власна розробка ServerPanel спрямована на заповнення цієї технологічної ніші шляхом створення оптимізованої веб-орієнтованої мультитенантної системи. Вона поєднує повний спектр управління інфраструктурними

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		19

примітивами Docker з безпековим контуром, що включає захист квот від станів гонки, автоматичне обчислення непересічних діапазонів портів та сувору валідацію конфігурацій, а також дуплексний моніторинг через WebSocket, забезпечуючи легке розгортання в межах єдиного безкоштовного рішення на базі сучасного стеку .NET 8 та React 19.

1.3. Визначення функціоналу системи

На основі проведеного аналізу предметної галузі та наявних аналогів визначено розширений перелік функціональних можливостей, що мають бути реалізовані у програмному комплексі. Система орієнтована на розробників та DevOps-інженерів, а також на середовища колективного використання, у яких єдиний Docker-хост спільно використовується кількома незалежними користувачами.

Першою і базовою функціональною вимогою є забезпечення багатокористувацької автентифікації на основі стандарту JWT [25]. Система підтримує реєстрацію нових облікових записів та автентифікацію наявних користувачів з видачею токенів доступу з обмеженим терміном дії. Токен передається у заголовок кожного запиту і використовується для підтвердження особи користувача без необхідності зберігання сесій на сервері, що відповідає принципам безстанової архітектури REST [12]. Для забезпечення можливості миттєвого відкликання прав доступу кожен токен містить версію ролі користувача, що дозволяє інвалідувати раніше видані токени при зміні ролі або видаленні облікового запису.

Нерозривно пов'язаною з автентифікацією є функція розмежування прав доступу на основі рольової моделі. Система передбачає дві ролі: адміністратор та звичайний користувач. Адміністратор має повний доступ до всіх функцій системи, включаючи керування користувачами, шаблонами конфігурацій, образами та перегляд журналу дій усіх користувачів. Звичайний користувач має доступ виключно до власних ресурсів у межах визначених для нього квот.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		20

Ключовою архітектурною вимогою є забезпечення мультитенантної ізоляції ресурсів. Кожен користувач має доступ лише до власних контейнерів, стеків, мереж і томів, тоді як спроба доступу до чужих ресурсів блокується на рівні сервісів. Доповненням до ізоляції є система квотування, що обмежує сумарну кількість контейнерів, обсяг оперативної пам'яті та процесорного часу для кожного користувача, запобігаючи монополізації ресурсів сервера. Для уникнення мережеских конфліктів між користувачами система автоматично виділяє кожному користувачеві персональний діапазон мережеских портів обсягом двадцять портів, що обчислюється на основі унікального ідентифікатора користувача та гарантує неперетинання діапазонів різних користувачів.

Центральною функцією системи є керування повним життєвим циклом Docker-контейнерів через веб-інтерфейс. Передбачається підтримка операцій створення, запуску, зупинки, перезапуску, призупинення, відновлення та примусового завершення контейнера, а також його перестворення та видалення. Кожен перехід між станами супроводжується перевіркою його коректності, що унеможливорює виконання недопустимих операцій. При створенні та оновленні контейнера користувач налаштовує його конфігурацію — переадресацію мережеских портів, змінні середовища, монтування томів, політику перезапуску та обмеження ресурсів. Для отримання детальної інформації про контейнер передбачено функцію інспектування, що відображає повний стан, параметри запуску та налаштування безпеки.

Для забезпечення оперативного контролю стану інфраструктури система реалізує потоковий моніторинг ресурсів у реальному часі. Показники використання процесора та оперативної пам'яті кожного контейнера передаються від сервера до браузера через WebSocket-з'єднання і відображаються у вигляді динамічних графіків без перезавантаження сторінки. Поряд з моніторингом система забезпечує потоковий перегляд журналів роботи контейнерів та надає інтерактивний термінал, що дозволяє виконувати команди безпосередньо всередині контейнера через браузер. Додатково реалізовано

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		21

файловий менеджер для перегляду, читання та запису файлів усередині контейнера без необхідності підключення до сервера.

Окремим напрямком функціоналу є керування додатковими примітивами Docker. Система підтримує розгортання багатокомпонентних застосунків на основі стеків Docker Compose з автоматичною валідацією безпеки конфігурацій, а також керування ізольованими мережами та іменованими томами Docker. Передбачено взаємодію з Docker-образами — перегляд списку наявних образів, завантаження нових з реєстру Docker Hub та видалення непотрібних. Для спрощення розгортання типових сервісів адміністратор може створювати шаблони конфігурацій, на основі яких користувачі швидко розгортають попередньо налаштовані контейнери.

Для забезпечення підтримки користувачів система містить вбудовану систему сервісних тікетів, що дозволяє організувати листування між користувачем та адміністратором. Завершальним елементом функціоналу є журнал дій користувачів, що фіксує всі значущі операції із зазначенням часу виконання та облікового запису, забезпечуючи можливість аудиту безпеки та діагностики. Журнал зберігається навіть після видалення облікового запису користувача завдяки збереженню денормалізованої інформації про автора дії.

Визначений перелік функціональних можливостей наведено у зведеному вигляді у табл. 1.2.

Таблиця 1.2

Функціональні можливості програмного комплексу

№	Функція	Роль
1	Реєстрація та багатокористувацька JWT-автентифікація	Всі користувачі
2	Розмежування рівнів доступу на основі рольової моделі	Адміністратор / Користувач

Продовження таблиці 1.2

№	Функція	Роль
3	Мультитенантна ізоляція обчислювальних ресурсів та даних	Всі користувачі
4	Квотування ресурсів хоста (кількість контейнерів, RAM, CPU)	Адміністратор
5	Автоматичний розрахунок та виділення персональних діапазонів портів	Всі користувачі
6	Керування повним життєвим циклом Docker-контейнерів	Всі користувачі
7	Налаштування конфігурацій контейнерів (порти, env, томи, ліміти)	Всі користувачі
8	Інспектування внутрішнього стану об'єктів інфраструктури	Всі користувачі
9	Потоковий моніторинг утилізації CPU/RAM у реальному часі	Всі користувачі
10	Перегляд журналів роботи контейнерів у потоковому режимі	Всі користувачі
11	Інтерактивний веб-термінал всередині контейнера	Всі користувачі
12	Вбудований файловий менеджер контейнера	Всі користувачі
13	Розгортання та керування стеками Docker Compose	Всі користувачі
14	Створення та адміністрування мереж та томів Docker	Всі користувачі

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				23
Змн.	Арк.	№ докум.	Підпис	Дата		

№	Функція	Роль
15	Взаємодія з образами та завантаження з реєстру Docker Hub	Адміністратор
16	Створення та використання конфігураційних шаблонів	Адміністратор / Користувач
17	Система сервісних тікетів технічної підтримки	Всі користувачі
18	Наскрізний журнал дій користувачів (audit log)	Адміністратор

Визначений функціонал відрізняє розроблювану систему від наявних аналогів комплексністю підходу до керування контейнерною інфраструктурою у мультитенантному середовищі. На відміну від Portainer, що надає повноцінне квотування ресурсів та рольову модель лише в комерційній версії, та CasaOS, що взагалі не підтримує багатокористувацьку роботу, розроблювана система забезпечує весь перелік визначених функцій у межах єдиного безкоштовного рішення. Поєднання суворої ізоляції ресурсів, потокового моніторингу через WebSocket, повного набору примітивів Docker та інтегрованих засобів адміністрування формує цілісний інструмент для колективного керування контейнерною інфраструктурою.

Висновки до розділу 1

У першому розділі проведено аналіз предметної галузі, досліджено наявні аналоги програмного комплексу та визначено функціональні вимоги до розроблюваної системи.

Аналіз предметної галузі засвідчив, що контейнеризація на базі платформи Docker є одним з ключових інструментів сучасної розробки та експлуатації програмного забезпечення. Встановлено, що управління контейнерною інфраструктурою засобами інтерфейсу командного рядка є трудомістким

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				24
Змн.	Арк.	№ докум.	Підпис	Дата		

процесом, який підвищує ризик виникнення суб'єктивних помилок адміністрування, унеможливорює оперативний моніторинг ресурсів у реальному часі та створює проблеми з розмежуванням прав доступу і безпечною ізоляцією ресурсів у середовищах з кількома користувачами.

Аналіз наявних аналогів — Portainer, Pterodactyl та CasaOS — виявив суттєві обмеження кожного з розглянутих рішень. Portainer надає повноцінне квотування ресурсів та рольову модель лише в комерційній версії. Pterodactyl є вузькоспеціалізованим рішенням для ігрових серверів з надлишково складною двокомпонентною архітектурою, непридатним для керування довільними примітивами Docker. CasaOS орієнтований на персональне використання та не підтримує багатокористувацьку роботу, розмежування прав доступу і моніторинг у реальному часі через WebSocket. Жоден з розглянутих аналогів не забезпечує повного переліку необхідних функцій у межах єдиного безкоштовного рішення, що обґрунтовує доцільність розробки власного програмного комплексу.

На основі проведеного аналізу визначено вісімнадцять функціональних можливостей розроблюваної системи, що охоплюють багатокористувацьку автентифікацію з JWT, розмежування прав доступу на основі рольової моделі, мультитенантну ізоляцію ресурсів з квотуванням та автоматичним виділенням діапазонів портів, повний життєвий цикл керування контейнерами, потоковий моніторинг ресурсів, перегляд журналів та інтерактивний термінал, файловий менеджер, керування стеками Docker Compose, мережами, томами та образами, шаблони конфігурацій, систему тікетів підтримки та журнал дій користувачів. Визначений функціонал формує цілісний інструмент для колективного керування контейнерною інфраструктурою у мультитенантному середовищі.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		25

РОЗДІЛ 2. ТЕХНОЛОГІЧНИЙ СТЕК ТА ЗАСОБИ РЕАЛІЗАЦІЇ ПРОГРАМНОГО КОМПЛЕКСУ

2.1. Технологічний стек

Вибір технологічного стеку є одним з найважливіших рішень при проєктуванні програмного комплексу, оскільки він визначає не лише технічні можливості системи, а й складність розробки, продуктивність у виробничому середовищі та можливості подальшого супроводу. При виборі інструментів для реалізації панелі керування серверами враховувались такі критерії: продуктивність при обробці великої кількості одночасних WebSocket-з'єднань, наявність типізації для зниження кількості помилок під час розробки, зрілість екосистеми та якість документації, а також наявність готових інструментів для взаємодії з Docker Engine API.

Серверна платформа — ASP.NET Core

Для реалізації серверної частини програмного комплексу обрано платформу ASP.NET Core 8 — кросплатформений фреймворк з відкритим вихідним кодом для розробки веб-застосунків на основі мови програмування C# [13]. Платформа розробляється компанією Microsoft і є наступником класичного ASP.NET Framework з суттєво переробленою архітектурою, орієнтованою на продуктивність та модульність.

Основною перевагою ASP.NET Core є сильна статична типізація мови C#, яка дозволяє виявляти значну частину помилок на етапі компіляції, а не під час виконання програми. Це є критично важливим для серверної частини системи керування інфраструктурою, де помилки можуть призвести до некоректних операцій з контейнерами. Вбудована підтримка патерну Dependency Injection спрощує організацію залежностей між компонентами та підвищує тестованість коду. Асинхронна модель виконання на основі `async/await` забезпечує ефективне використання потоків при одночасному обслуговуванні кількох WebSocket-з'єднань без блокування.

При виборі серверної платформи розглядались альтернативи — Node.js та

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		26

Python з фреймворком FastAPI. Node.js забезпечує високу продуктивність при обробці великої кількості паралельних з'єднань завдяки подійно-орієнтованій архітектурі, однак динамічна типізація JavaScript підвищує ризик виникнення помилок під час виконання програми. Відсутність вбудованої підтримки WebSocket-хабів рівня SignalR потребувала б розробки додаткових рішень для організації real-time комунікації. Python з FastAPI є зручним для швидкого прототипування, однак поступається ASP.NET Core за продуктивністю при обробці великої кількості одночасних з'єднань та не має рівноцінного інструменту для роботи з Docker API. ASP.NET Core обрано як платформу з оптимальним поєднанням типізації, продуктивності та готових інструментів для реалізації визначеного функціоналу.

Real-time комунікація — SignalR

Для організації передачі даних моніторингу у реальному часі обрано бібліотеку SignalR, яка є частиною екосистеми ASP.NET Core [14]. SignalR абстрагує роботу з транспортними протоколами і автоматично обирає оптимальний механізм передачі даних залежно від можливостей середовища: WebSocket, Server-Sent Events або Long Polling. У разі підтримки браузером протоколу WebSocket SignalR використовує саме його як найефективніший варіант з мінімальною затримкою.

Порівняно з традиційним підходом HTTP-опитування SignalR забезпечує значно нижче навантаження на сервер, оскільки з'єднання встановлюється одноразово і залишається активним протягом усього сеансу роботи користувача. Сервер самостійно надсилає оновлені дані клієнту в момент їх появи без необхідності повторних запитів. Концепція хабів SignalR дозволяє організувати групову розсилку повідомлень підключеним клієнтам, що забезпечує одночасну передачу метрик CPU та оперативної пам'яті кільком користувачам системи. Для потокового виведення журналів контейнерів SignalR забезпечує передачу кожного рядка логу одразу після його появи без буферизації.

Взаємодія з Docker Engine — Docker.DotNet

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		27

Для взаємодії з Docker Engine з серверної частини застосунку обрано бібліотеку Docker.DotNet — типізований .NET-клієнт для Docker API [12]. Бібліотека надає зручний об'єктно-орієнтований інтерфейс для виконання всіх операцій з контейнерами, образами та мережами через Docker REST API, що усуває необхідність формування HTTP-запитів вручну та забезпечує перевірку типів на етапі компіляції.

Docker.DotNet підтримує підключення як через Unix-сокет на локальному вузлі, так і через мережевий інтерфейс для взаємодії з віддаленим Docker Engine. Бібліотека повністю підтримує асинхронну модель виконання ASP.NET Core, що забезпечує ефективне використання потоків при виконанні тривалих операцій, зокрема потокового читання журналів контейнерів та отримання статистики використання ресурсів у реальному часі. Підтримка скасування операцій через CancellationToken дозволяє коректно завершувати потокові операції при відключенні клієнта.

Клієнтська частина — React та TypeScript

Для реалізації клієнтської частини програмного комплексу обрано бібліотеку React у поєднанні з мовою TypeScript [17, 18]. React є провідним рішенням для розробки односторінкових застосунків, архітектура якого базується на концепції компонентного підходу. Інтерфейс користувача будується з незалежних повторно використовуваних компонентів з власним станом, що спрощує розробку та супровід складних інтерфейсів. Механізм віртуального DOM забезпечує ефективне оновлення інтерфейсу при зміні даних без повного перезавантаження сторінки, що є особливо важливим для відображення метрик моніторингу, які оновлюються у реальному часі.

TypeScript розширює JavaScript статичною типізацією, що дозволяє виявляти помилки на етапі розробки та забезпечує кращу підтримку з боку редакторів коду завдяки автодоповненню та перевірці типів. Використання TypeScript на фронтенді у поєднанні з C# на бекенді забезпечує типізацію на всіх рівнях системи, що знижує ймовірність помилок при взаємодії між клієнтом і сервером.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		28

При виборі фронтенд-технології розглядались альтернативи — Angular та Vue.js. Angular є потужним фреймворком з вбудованою підтримкою TypeScript та чіткою структурою, однак має значно вищий поріг входу та більший розмір фінального пакету. Vue.js є легшою альтернативою з простішим синтаксисом шаблонів, однак має меншу екосистему готових компонентів для побудови інформаційних панелей з динамічними графіками. React обрано як оптимальний варіант завдяки великій кількості готових бібліотек для візуалізації даних та широкій спільноті розробників [3].

Використання TypeScript на клієнтській частині забезпечує статичну типізацію JavaScript-коду, що дозволяє виявляти помилки на етапі розробки до запуску застосунку. Ключовою перевагою TypeScript у контексті даного проєкту є можливість визначення інтерфейсів та типів, що відповідають моделям даних серверної частини. Наприклад, інтерфейс Container на фронтенді точно відповідає структурі об'єкта, що повертається з API бекенду — назва, статус, прив'язані порти та змінні середовища. Така синхронізація типів між бекендом на C# та фронтендом на TypeScript знижує ймовірність помилок при розборі відповідей API та забезпечує автодоповнення у редакторі коду, що прискорює розробку.

Окремою перевагою TypeScript є підтримка узагальнених типів, що дозволяє створювати повторно використовувані утиліти для роботи з API-відповідями. Типізований клієнт SignalR на фронтенді дозволяє визначити точні типи повідомлень, що надходять через WebSocket-з'єднання — метрики CPU та RAM, рядки журналів контейнерів — що виключає необхідність ручної перевірки структури отриманих даних під час виконання програми.

Складання та збірка клієнтської частини — Vite

Для складання клієнтської частини застосунку використано інструмент Vite — сучасний складальник проєктів для фронтенду [19]. Vite забезпечує швидку збірку застосунку завдяки використанню нативних модулів ECMAScript під час розробки та оптимізованої збірки на основі Rollup для виробничого середовища. Порівняно з традиційними складальниками, Vite надає суттєво

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				29
Змн.	Арк.	№ докум.	Підпис	Дата		

швидший запуск середовища розробки та миттєве оновлення модулів при зміні коду, що прискорює процес розробки інтерфейсу. Результатом складання є набір статичних файлів, які обслуговуються веб-сервером і завантажуються у браузер користувача.

Інтерактивний термінал — xterm.js

Для реалізації інтерактивного терміналу, що дозволяє виконувати команди безпосередньо всередині контейнера через браузер, на клієнтській частині застосовано бібліотеку xterm.js [20]. Ця бібліотека є повноцінною реалізацією емулятора терміналу для веб-середовища і використовується у багатьох відомих проєктах, зокрема у редакторі Visual Studio Code.

Бібліотека xterm.js забезпечує коректне відображення керуючих послідовностей терміналу, підтримку кодування UTF-8 та зміну розміру термінального вікна. Взаємодія між терміналом у браузері та процесом усередині контейнера організована через WebSocket-з'єднання на основі SignalR: введені користувачем команди передаються на сервер, який транслює їх у контейнер через механізм Docker exec з виділенням псевдотерміналу, а результати виконання повертаються назад до браузера у реальному часі. Такий підхід забезпечує повноцінну роботу з командним рядком контейнера без необхідності встановлення SSH-з'єднання з сервером.

База даних — PostgreSQL та Entity Framework Core

Для зберігання даних програмного комплексу обрано систему керування базами даних PostgreSQL — об'єктно-реляційну СКБД з відкритим вихідним кодом [21]. PostgreSQL забезпечує повну підтримку стандарту SQL, надійність транзакцій на основі механізму ACID та широкі можливості індексування, що гарантує цілісність даних при одночасному доступі кількох користувачів. Реляційна модель даних є більш доцільною для зберігання структурованих даних про користувачів, контейнери, порти та змінні середовища, між якими існують чіткі зв'язки з обмеженнями цілісності.

При виборі СКБД розглядалися MongoDB та SQLite. MongoDB як документо-орієнтована СКБД є зручною для зберігання слабоструктурованих

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		30

даних, однак відсутність жорстких зв'язків між документами ускладнює забезпечення цілісності даних при каскадному видаленні контейнерів з пов'язаними портами та змінними середовища. SQLite є простим вбудованим рішенням без необхідності окремого серверного процесу, однак має суттєві обмеження щодо паралельного доступу при одночасній роботі кількох користувачів системи. PostgreSQL позбавлений цих обмежень та забезпечує повноцінну підтримку конкурентного доступу.

Взаємодія з PostgreSQL організована через Entity Framework Core — об'єктно-реляційний маппер, що дозволяє працювати з даними у вигляді об'єктів C# без написання SQL-запитів вручну [15]. EF Core забезпечує міграції бази даних, що дозволяє версіонувати структуру бази даних разом з кодом застосунку у системі контролю версій.

Автентифікація — JWT та BCrypt

Для реалізації автентифікації та захисту паролів обрано стандарт JWT у поєднанні з алгоритмом хешування BCrypt [25, 28]. JSON Web Token є відкритим стандартом для безпечної передачі інформації між сторонами у вигляді підписаного JSON-об'єкта. Токен містить інформацію про користувача та його роль, підписується алгоритмом HMAC-SHA256 і передається у заголовок кожного HTTP-запиту. Безстановий характер JWT усуває необхідність зберігання сесій на сервері, що спрощує горизонтальне масштабування системи. BCrypt є адаптивним алгоритмом хешування паролів, що використовує сіль для захисту від атак за допомогою попередньо обчислених таблиць та налаштований параметр складності для збільшення часу обчислення хешу [10]. На відміну від алгоритмів загального призначення, таких як MD5 або SHA-256, BCrypt розроблений спеціально для хешування паролів і стійкий до атак грубої сили завдяки можливості збільшення обчислювальної складності зі зростанням потужності апаратного забезпечення.

Логування та діагностика — Serilog

Для організації структурованого логування у програмному комплексі обрано бібліотеку Serilog [29] — одне з найпоширеніших рішень для ведення

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		31

журналів у застосунках на платформі .NET. На відміну від стандартного механізму логування, Serilog підтримує структуроване логування, за якого записи журналу зберігаються не як простий текст, а як структуровані дані з іменованими параметрами. Це суттєво спрощує подальший пошук, фільтрацію та аналіз журналів.

У програмному комплексі Serilog налаштовано на запис журналів роботи серверної частини у файлову систему, що дозволяє відстежувати перебіг виконання операцій, діагностувати помилки та аналізувати поведінку системи у виробничому середовищі. Ведення журналів є важливим аспектом експлуатації системи керування інфраструктурою, оскільки дозволяє відновити послідовність дій, що передували виникненню проблеми, та забезпечує прозорість роботи застосунку.

Обробка конфігурацій Docker Compose — YamlDotNet

Для підтримки розгортання багатокомпонентних застосунків на основі стеків Docker Compose у програмному комплексі застосовано бібліотеку YamlDotNet — інструмент для розбору та аналізу документів у форматі YAML на платформі .NET [23]. Конфігурації Docker Compose описуються саме у форматі YAML, тому для роботи зі стеками необхідним є надійний механізм розбору таких документів у структуру об'єктів C#.

Застосування YamlDotNet дозволяє не лише зчитувати конфігурацію стеку, а й здійснювати її валідацію перед розгортанням. Зокрема, перед запуском стеку конфігурація аналізується на наявність небезпечних параметрів — привілейованого режиму, небезпечних мережевих налаштувань та несанкціонованих монтувань директорій хостової системи. Можливість програмного розбору YAML є ключовою для реалізації механізму безпеки конфігурацій, який детально описано у підрозділі 2.4 «Реалізація механізмів безпеки та мультитенантної ізоляції».

На завершення підрозділу наведено зведену порівняльну таблицю обраного технологічного стеку з альтернативами у табл. 2.1.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		32

Таблиця 2.1

Порівняльний аналіз технологічного стеку

Компонент	Обрано	Альтернатива 1	Альтернатива 2	Причина вибору
Серверна платформа	ASP.NET Core 8	Node.js	Python FastAPI	Типізація, вбудований SignalR, DI
Мова бекенду	C#	JavaScript	Python	Статична типізація, продуктивність
Real-time	SignalR	Socket.io	ws (Node.js)	Вбудований в .NET, автовибір транспорту
Docker API	Docker.DotNet	HTTP вручну	Dockerode (JS)	Типізований .NET клієнт
Фронтенд	React + TypeScript	Angular	Vue.js	Екосистема, гнучкість, типізація
СКБД	PostgreSQL	MongoDB	SQLite	ACID, реляційні зв'язки, конкурентний доступ
ORM	EF Core	Dapper	raw SQL	Міграції, типізація, продуктивність
Автентифікація	JWT	Sessions	OAuth2	Безстановість, масштабованість
Хешування	BCrypt	SHA-256	Argon2	Адаптивна складність, захист від брутфорсу
Логування	Serilog	вбудоване логування	NLog	Структуроване логування у файли

Продовження таблиці 2.1

Компонент	Обрано	Альтернатива 1	Альтернатива 2	Причина вибору
Парсинг YAML	YamlDotNet	ручний розбір	-	Розбір та валідація Docker Compose
Веб-термінал	xterm.js	-	-	Емулятор терміналу для браузера
Складальник фронту	Vite	Webpack	Create React App	Швидка збірка, миттєве оновлення

Наведений технологічний стек забезпечує узгоджену типізацію на всіх рівнях системи — від бази даних через C# до TypeScript на фронтенді — що мінімізує ймовірність помилок при взаємодії між компонентами.

2.2. Архітектура та патерни проєктування

Проєктування програмного комплексу здійснювалось з дотриманням загальноприйнятих архітектурних принципів та патернів, що забезпечують розділення відповідальності між компонентами системи, спрощують тестування та підвищують гнучкість коду при внесенні змін. У межах реалізації застосовано патерн Repository, механізм Dependency Injection, а також принципи об'єктно-орієнтованого програмування та SOLID [8].

Варіанти використання системи

Перед розглядом внутрішньої структури програмного комплексу доцільно визначити, яким чином користувачі взаємодіють із системою на високому рівні. У системі передбачено двох акторів — звичайного користувача та адміністратора. Звичайний користувач має доступ до функцій керування власними ресурсами: реєстрації та автентифікації, керування контейнерами та їх конфігурацією, моніторингу ресурсів, перегляду журналів і роботи з

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		34

інтерактивним терміналом, керування стеками, мережами та томами, роботи з файлами контейнерів, а також створення звернень до технічної підтримки. Адміністратор успадковує всі можливості звичайного користувача та додатково отримує доступ до адміністративних функцій — керування обліковими записами користувачів, налаштування квот ресурсів, керування образами та шаблонами конфігурацій, перегляду журналу дій усіх користувачів та обробки звернень до підтримки. Діаграму варіантів використання системи наведено (рис. 2.1).

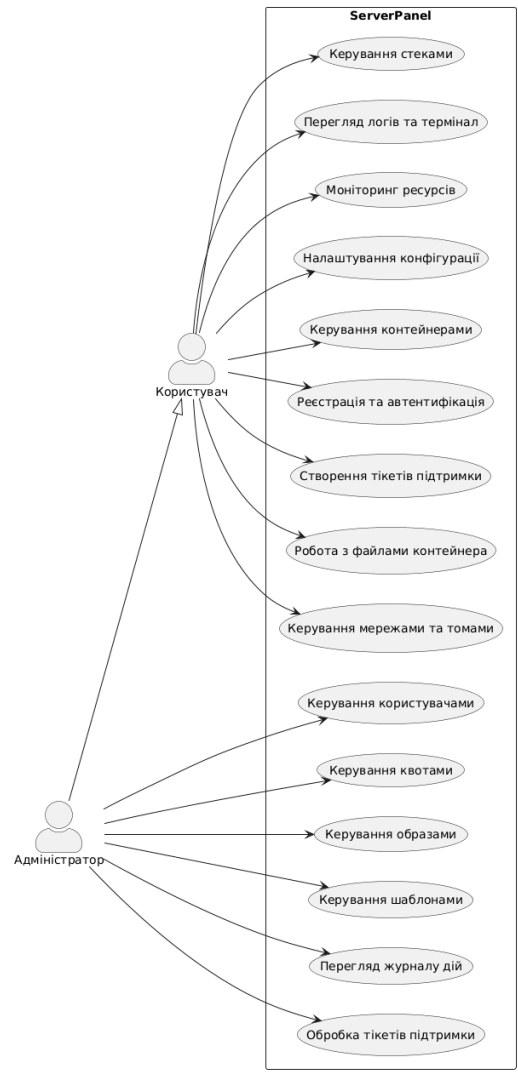


Рис. 2.1. Діаграма варіантів використання системи
Трирівнева архітектура

Архітектура програмного комплексу побудована за принципом трирівневої клієнт-серверної архітектури з чітким поділом відповідальності між шарами. Такий підхід є загальноприйнятою практикою у розробці корпоративних веб-застосунків і забезпечує незалежність кожного шару від

деталей реалізації сусідніх, що суттєво спрощує тестування окремих компонентів та подальший супровід системи. Зміна реалізації одного шару — наприклад, заміна бази даних або переробка інтерфейсу — не впливає на логіку роботи інших шарів за умови збереження контрактів інтерфейсів. Загальну структуру взаємодії шарів програмного комплексу наведено (рис. 2.2).

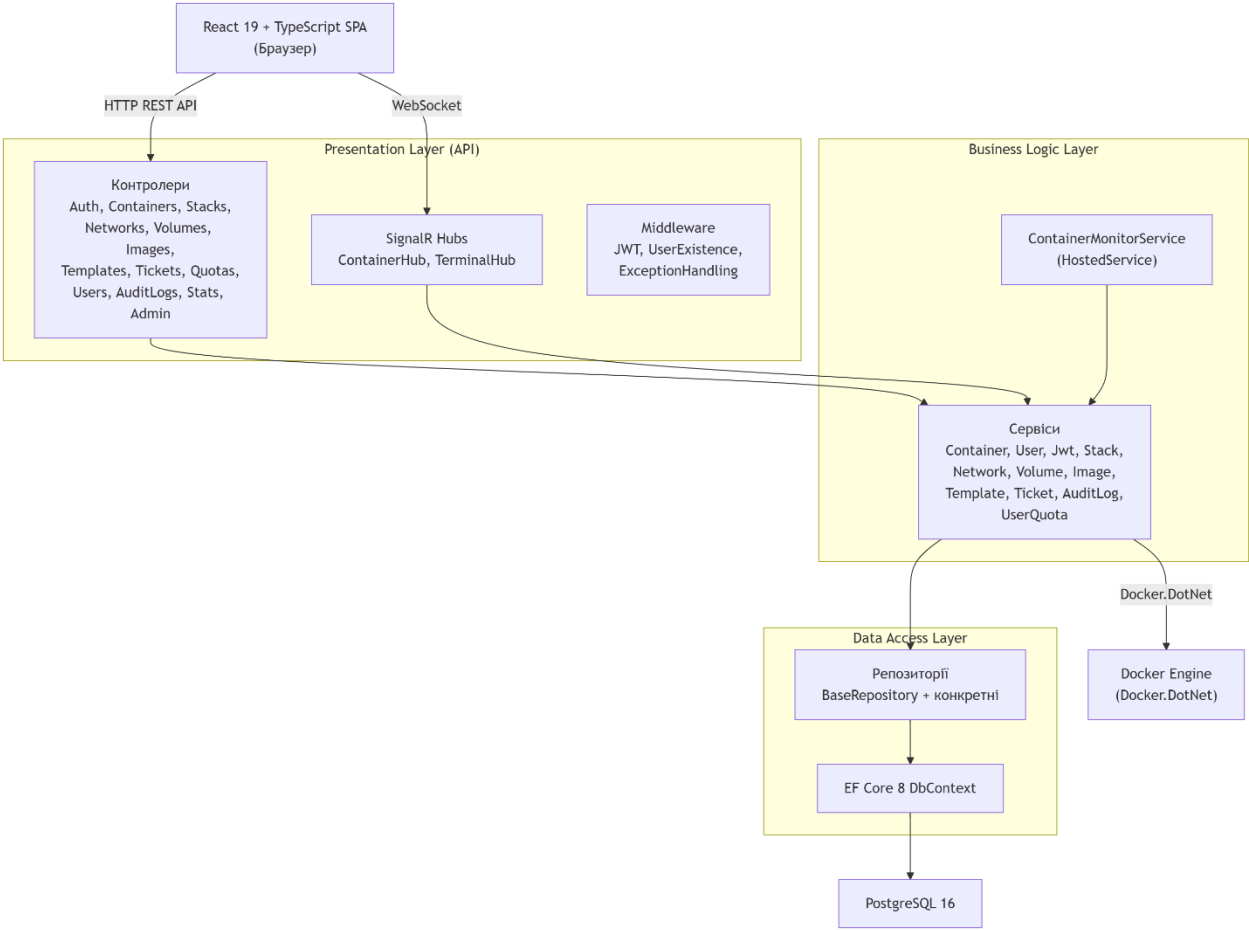


Рис. 2.2. Тривірнева архітектура програмного комплексу

Рівень представлення реалізовано у вигляді контролерів ASP.NET Core Web API та хабів SignalR. Контролери відповідають виключно за приймання HTTP-запитів від клієнта, валідацію вхідних даних на рівні моделі та формування HTTP-відповідей. Вся бізнес-логіка делегується сервісам нижнього шару через інтерфейси — контролер не містить жодної логіки роботи з Docker або базою даних. Хаби SignalR відповідають за встановлення та підтримку WebSocket-з'єднань з клієнтами і організацію передавання повідомлень при оновленні метрик моніторингу та потокового виведення журналів.

Рівень бізнес-логіки містить сервіси, що реалізують основні алгоритми роботи системи. Сервіс ContainerService інкапсулює всю логіку взаємодії з Docker Engine через бібліотеку Docker.DotNet — створення, запуск, зупинку, видалення та інші операції життєвого циклу контейнерів. Саме на цьому рівні виконується перевірка бізнес-правил: чи має користувач право на виконання операції відповідно до своєї ролі, чи не перевищено квоту ресурсів, чи відповідають параметри конфігурації встановленим обмеженням. Сервіс UserService відповідає за логіку автентифікації та керування користувачами, JwtService — за генерацію токенів, AuditLogService — за фіксацію значущих операцій. Окрім зазначених, рівень бізнес-логіки містить низку інших спеціалізованих сервісів — для керування стеками, мережами, томами, образами, шаблонами, тікетами та квотами користувачів, кожен з яких відповідає за власну функціональну область. Такий підхід гарантує, що жодна перевірка безпеки або бізнес-правило не може бути обійдена через прямий доступ до рівня даних.

Рівень доступу до даних реалізовано на основі Entity Framework Core з патерном Repository. Цей шар повністю ізолює логіку роботи з базою даних PostgreSQL від решти системи — сервіси бізнес-логіки взаємодіють виключно з інтерфейсами репозиторіїв, не маючи прямого доступу до контексту бази даних. Використання EF Core міграцій дозволяє версіонувати структуру бази даних разом з кодом застосунку у системі контролю версій GitLab.

Патерн Repository

Патерн Repository є структурним патерном проєктування, що забезпечує абстракцію між рівнем бізнес-логіки та рівнем доступу до даних. Замість того щоб сервіси безпосередньо звертались до контексту бази даних Entity Framework Core, вся логіка роботи з даними інкапсулюється у репозиторіях, які надають сервісам простий інтерфейс для виконання операцій зі збереження та отримання даних.

У програмному комплексі реалізовано базовий узагальнений репозиторій BaseRepository, що містить спільну логіку операцій читання, створення, оновлення та видалення записів. Конкретні репозиторії — ContainerRepository,

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		37

UserRepository, AuditLogRepository — наслідують базовий клас і розширюють його специфічними методами для роботи з відповідними сутностями. Наприклад, ContainerRepository містить метод отримання всіх контейнерів конкретного користувача, що використовується для забезпечення ізоляції даних між користувачами системи.

Застосування патерну Repository надає кілька практичних переваг у контексті даного проєкту. По-перше, сервіси бізнес-логіки залежать від інтерфейсів репозиторіїв, а не від конкретних реалізацій, що дозволяє підміняти реальні репозиторії тестовими заглушками при написанні модульних тестів. По-друге, зміна СКБД або ORM потребує лише переробки репозиторіїв без зміни коду сервісів. По-третє, концентрація логіки роботи з даними в одному місці спрощує оптимізацію запитів та відстеження проблем з продуктивністю.

Патерн Dependency Injection

Dependency Injection є патерном проєктування, що реалізує принцип інверсії залежностей і полягає у передачі залежностей об'єкту ззовні замість їх створення всередині об'єкта. ASP.NET Core має вбудований DI-контейнер, що є центральним механізмом управління залежностями у всьому програмному комплексі.

У проєкті всі сервіси та репозиторії реєструються у DI-контейнері під час запуску застосунку і впроваджуються через конструктори класів [16]. Контролери отримують сервіси бізнес-логіки, сервіси отримують репозиторії, репозиторії отримують контекст бази даних — жоден клас не створює власні залежності самостійно. Такий підхід забезпечує слабе зв'язування між компонентами системи, оскільки кожен клас знає лише про інтерфейс своєї залежності, а не про конкретну реалізацію.

Вбудований DI-контейнер ASP.NET Core підтримує три режими життєвого циклу залежностей. Режим Singleton передбачає створення єдиного екземпляра сервісу на весь час роботи застосунку — у такому режимі зареєстровано сервіси, що не мають стану. Режим Scoped передбачає створення нового екземпляра для кожного HTTP-запиту — у такому режимі зареєстровано репозиторії та контекст

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		38

бази даних, що гарантує коректну роботу транзакцій в межах одного запиту. Режим Transient передбачає створення нового екземпляра при кожному зверненні до DI-контейнера.

Принципи об'єктно-орієнтованого програмування

Програмний комплекс реалізовано з дотриманням чотирьох основних принципів об'єктно-орієнтованого програмування, що забезпечують якість архітектурних рішень та зручність супроводу коду.

Принцип інкапсуляції реалізовано через приховування внутрішнього стану сутностей та сервісів за допомогою приватних полів і публічних властивостей. Сутності бази даних — User, Container, Port, EnvVariable — надають доступ до своїх даних виключно через публічні властивості, тоді як логіка валідації та перетворення даних залишається прихованою всередині класу. Зовнішній код взаємодіє з об'єктами через визначені інтерфейси, що захищає внутрішню реалізацію від небажаних змін ззовні.

Принцип наслідування реалізовано через базовий узагальнений репозиторій BaseRepository, що містить спільну реалізацію операцій з базою даних. Конкретні репозиторії наслідують базовий клас і отримують готову реалізацію базових операцій, розширюючи її лише специфічними методами. Такий підхід усуває дублювання коду та забезпечує однаковий підхід до роботи з різними сутностями системи.

Принцип поліморфізму реалізовано через систему інтерфейсів IRepository, IUserService та IContainerService. Сервіси бізнес-логіки оголошують залежності від інтерфейсів, а не від конкретних класів, що дозволяє DI-контейнеру підставляти різні реалізації залежно від контексту — реальні сервіси у виробничому середовищі та тестові заглушки при виконанні модульних тестів.

Принцип абстракції реалізовано через чіткий поділ системи на шари з визначеними контрактами взаємодії між ними. Рівень бізнес-логіки оперує абстракціями репозиторіїв, не знаючи деталей реалізації рівня доступу до даних. Рівень представлення оперує абстракціями сервісів, не знаючи деталей взаємодії з Docker Engine або базою даних.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		39

Окремої уваги заслуговує підхід до передачі даних між серверною та клієнтською частинами системи. У програмному комплексі застосовано патерн Data Transfer Object — окремі класи, що використовуються виключно для передачі даних між шарами системи та через API. Сутності бази даних — User, Container, Port, EnvVariable — містять всі поля, включаючи хеш паролю, внутрішні ідентифікатори та службові поля, і не передаються безпосередньо клієнту. Натомість для кожної операції визначено окремий DTO з мінімально необхідним набором полів — наприклад, LoginRequestDto містить лише email та пароль, а ContainerResponseDto містить лише ті поля контейнера, які необхідні для відображення в інтерфейсі користувача.

Застосування DTO надає кілька практичних переваг. По-перше, внутрішня структура бази даних залишається прихованою від клієнта, що підвищує безпеку системи — клієнт ніколи не отримує хеш паролю або інші службові поля. По-друге, зміна структури сутностей бази даних не обов'язково впливає на структуру API, оскільки мапінг між сутностями та DTO здійснюється на рівні сервісів. По-третє, TypeScript-інтерфейси на фронтенді визначаються відповідно до структури DTO, що забезпечує точну синхронізацію типів між бекендом та клієнтською частиною без зайвих полів.

Принципи SOLID

Проектування програмного комплексу здійснювалось з дотриманням п'яти принципів SOLID, що є стандартом якісного об'єктно-орієнтованого проектування [8].

Принцип єдиної відповідальності забезпечується чітким розподілом функцій між класами системи. ContainerService відповідає виключно за операції з контейнерами, UserService — за автентифікацію та керування користувачами, AuditLogService — за фіксацію дій користувачів. Жоден сервіс не суміщає кілька непов'язаних відповідальностей, що спрощує внесення змін та локалізацію помилок.

Принцип відкритості та закритості реалізовано через систему інтерфейсів. Додавання нового типу сховища даних або нового способу взаємодії з Docker не

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		40

потребує зміни наявних класів — достатньо створити нову реалізацію відповідного інтерфейсу та зареєструвати її у DI-контейнері. Існуючий код залишається незмінним, що знижує ризик внесення нових помилок.

Принцип підстановки Лісков дотримується через коректну реалізацію наслідування репозиторіїв. Будь-який конкретний репозиторій — UserRepository або ContainerRepository — може бути використаний там, де очікується базовий BaseRepository без порушення коректності роботи системи, оскільки конкретні репозиторії лише розширюють, але не змінюють поведінку базового класу.

Принцип розділення інтерфейсів реалізовано через створення окремих вузькоспеціалізованих інтерфейсів для кожного сервісу та репозиторію замість одного великого інтерфейсу. Клієнтський код залежить лише від тих методів, які він реально використовує, що знижує зв'язність між компонентами системи.

Принцип інверсії залежностей є наскрізним принципом всієї архітектури програмного комплексу. Рівень бізнес-логіки залежить від абстракцій рівня доступу до даних, а не від їх конкретних реалізацій. Рівень представлення залежить від абстракцій рівня бізнес-логіки. Всі залежності впроваджуються через конструктор за допомогою вбудованого DI-контейнера ASP.NET Core, що забезпечує повну відповідність принципу на всіх рівнях системи.

Структуру класів рівня бізнес-логіки та доступу до даних відображено на діаграмі класів, побудованій за стандартом UML (рис. 2.3). Ієрархія репозиторіїв побудована на основі узагальненого інтерфейсу IRepository та базового класу BaseRepository, від якого наслідуються конкретні репозиторії, зокрема ContainerRepository та UserRepository, що розширюють базову функціональність специфічними методами. Рівень сервісів представлено інтерфейсами IContainerService та IUserService з їх конкретними реалізаціями. Сервіс ContainerService залежить від репозиторію контейнерів, клієнта Docker.DotNet та сервісу квот, тоді як UserService взаємодіє з репозиторієм користувачів та сервісом генерації токенів JwtService.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				41
Змн.	Арк.	№ докум.	Підпис	Дата		

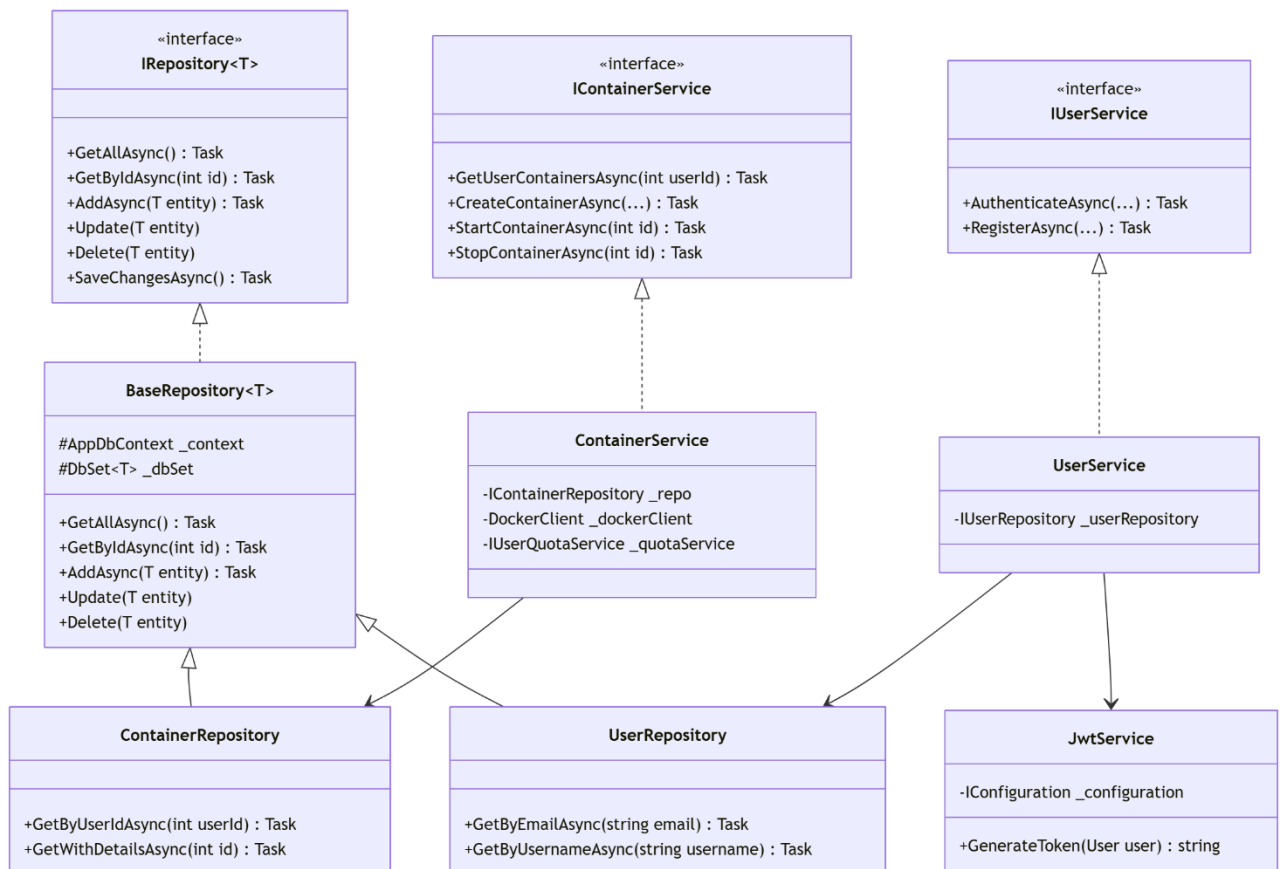


Рис. 2.3. Діаграма класів рівня репозиторіїв та сервісів

2.3. Структура бази даних

Зберігання даних програмного комплексу організовано у реляційній базі даних під керуванням системи PostgreSQL. Робота з базою даних реалізована за допомогою Entity Framework Core із застосуванням підходу Code-First, за якого структура бази даних визначається на основі класів-сутностей мовою C#, а зміни структури фіксуються у вигляді міграцій. Такий підхід дозволяє версіювати схему бази даних разом з кодом застосунку та автоматично застосовувати зміни при розгортанні. Загалом база даних складається з шістнадцяти таблиць, а її схема пройшла поетапний розвиток, зафіксований у двадцяти двох міграціях.

Усі таблиці бази даних можна умовно поділити на чотири логічні групи, що відображають специфіку мультитенантної архітектури системи. Зведену структуру та розподіл таблиць за функціональними групами наведено в табл. 2.2.

Структура таблиць бази даних за функціональними групами

Група	Таблиця	Призначення
Користувачі та квоти	Users	Облікові записи, ролі, автентифікація
	UserQuotas	Квоти ресурсів користувачів
Контейнери та конфігурація	Containers	Контейнери Docker
	Ports	Переадресація портів
	EnvVariables	Змінні середовища
	VolumeMounts	Монтування шляхів хоста
Інфраструктурні примітиви	Networks	Мережі Docker
	Volumes	Іменовані томи Docker
	Stacks	Стеки Docker Compose
	ContainerNetworks	Зв'язок контейнерів і мереж
	ContainerVolumes	Зв'язок контейнерів і томів
Адміністрування та підтримка	Templates	Шаблони конфігурацій
	AuditLogs	Журнал дій користувачів
	Tickets	Звернення до підтримки
	TicketMessages	Повідомлення в межах звернень
	BannedEmails	Заблоковані адреси email

Загальну структуру бази даних та зв'язки між таблицями відображено на діаграмі «сутність-зв'язок» (ER-діаграмі), наведеній у додатку А.

Першу логічну групу складають таблиці, що зберігають інформацію про користувачів системи та встановлені для них квоти ресурсів. Центральною таблицею всієї бази даних є таблиця користувачів «Users». Вона містить унікальний ідентифікатор, ім'я користувача та адресу електронної пошти, для яких визначено обмеження унікальності, хеш паролю, роль користувача та її версію, а також дату створення облікового запису. Поле версії ролі використовується механізмом автентифікації для миттєвого відкликання раніше виданих токенів доступу при зміні прав користувача. Зберігання саме хешу

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				43
Змн.	Арк.	№ докум.	Підпис	Дата		

пароллю, а не його відкритого значення, забезпечує захист облікових даних навіть у разі несанкціонованого доступу до бази даних.

Таблиця квот «UserQuotas» зберігає обмеження ресурсів для кожного користувача та пов'язана з таблицею користувачів відношенням один-до-одного. Вона визначає максимальну кількість контейнерів, які може створити користувач, сумарний обсяг доступної оперативної пам'яті у мегабайтах та сумарну кількість процесорних ядер. Винесення квот в окрему таблицю, а не зберігання їх безпосередньо у таблиці користувачів, дозволяє відокремити дані автентифікації від параметрів обмеження ресурсів та спрощує подальше розширення системи квотування.

Другу логічну групу утворюють таблиці контейнерів та пов'язаних з ними елементів конфігурації. Основною таблицею цієї групи є таблиця контейнерів «Containers», що пов'язана з таблицею користувачів відношенням один-до-багатьох — кожен контейнер належить одному користувачеві, а користувач може володіти багатьма контейнерами. Таблиця зберігає внутрішній ідентифікатор запису, ідентифікатор контейнера у Docker, назву, образ, поточний статус, дату створення, політику перезавпуску, обмеження оперативної пам'яті та процесора, ознаку ручної зупинки, а також зовнішні ключі на користувача-власника та на стек, до якого може належати контейнер. Зовнішній ключ на стек є необов'язковим, оскільки контейнер може бути створений як самостійно, так і у складі стеку Docker Compose. Ознака ручної зупинки використовується сервісом моніторингу для того, щоб відрізнити навмисно зупинені контейнери від тих, що завершилися аварійно, та не надсилати хибних сповіщень про збій.

Таблиця портів «Ports» зберігає правила переадресації мережевих портів між контейнером та хостовою системою і пов'язана з таблицею контейнерів відношенням один-до-багатьох. Кожен запис містить порт хоста, порт контейнера та мережевий протокол. Таблиця змінних середовища «EnvVariables» зберігає пари «ключ — значення», що передаються у контейнер під час запуску, і також пов'язана з контейнером відношенням один-до-багатьох.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		44

Таблиця монтувань «VolumeMounts» зберігає прив'язки шляхів хостової системи до шляхів усередині контейнера. Усі три таблиці конфігурації пов'язані з таблицею контейнерів каскадним видаленням, що гарантує автоматичне видалення всіх пов'язаних налаштувань при видаленні контейнера та запобігає появі осиротілих записів.

Третю логічну групу формують таблиці інфраструктурних примітивів Docker. Таблиця мереж «Networks» зберігає інформацію про створені користувачами мережі Docker — назву, ідентифікатор мережі у Docker, драйвер, область видимості та ознаку ізолюваності. Таблиця томів «Volumes» зберігає дані про іменовані томи — назву, ідентифікатор тому у Docker, драйвер та точку монтування. Обидві таблиці пов'язані з таблицею користувачів і містять необов'язковий зовнішній ключ на власника: якщо власник не визначений, ресурс вважається системним. Для кожної з цих таблиць встановлено обмеження унікальності за парою «власник — назва», що дозволяє різним користувачам створювати ресурси з однаковими назвами без конфліктів.

Таблиця стеків «Stacks» зберігає конфігурації багатокomпонентних застосунків Docker Compose. Кожен запис містить назву стеку, статус, текст конфігурації у форматі YAML, додатковий вміст файлу змінних середовища, а також зовнішній ключ на користувача-власника. Стек пов'язаний з контейнерами, що входять до його складу, відношенням один-до-багатьох, проте видалення стеку не призводить до видалення контейнерів — натомість зовнішній ключ контейнера на стек обнуляється. Таке рішення дозволяє зберегти контейнери навіть після видалення опису стеку.

Оскільки контейнер може бути підключений до кількох мереж, а до однієї мережі може бути підключено кілька контейнерів, між таблицями контейнерів і мереж існує відношення багато-до-багатьох. Аналогічне відношення існує між контейнерами та томами. Для реалізації таких відношень застосовано дві зв'язкові таблиці — «ContainerNetworks» та «ContainerVolumes», кожна з яких містить зовнішні ключі на обидві пов'язані сутності та забезпечує коректне

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		45

зберігання зв'язків. Зв'язкова таблиця томів додатково зберігає шлях монтування тому всередині контейнера та ознаку доступу лише для читання.

Четверту логічну групу складають допоміжні таблиці, що забезпечують функції адміністрування, підтримки та аудиту. Таблиця шаблонів «Templates» зберігає попередньо налаштовані конфігурації для швидкого розгортання типових застосунків. Кожен запис містить назву, опис, образ, піктограму, а також типові порти та змінні середовища, що зберігаються у форматі JSON. Ця таблиця є самостійною і не має зовнішніх зв'язків з іншими таблицями, оскільки шаблони використовуються лише як джерело початкових параметрів при створенні контейнерів.

Таблиця журналу дій «AuditLogs» фіксує всі значущі операції користувачів. Кожен запис містить зовнішній ключ на користувача, ім'я користувача, тип дії, тип та назву об'єкта, над яким виконано дію, додаткові деталі та час виконання. Особливістю цієї таблиці є те, що при видаленні облікового запису користувача зовнішній ключ обнуляється, проте сам запис журналу зберігається, а ім'я користувача залишається у денормалізованому вигляді. Таке рішення забезпечує цілісність історії дій навіть після видалення облікового запису, що є важливим для аудиту безпеки.

Підтримку користувачів реалізовано за допомогою таблиць тікетів «Tickets» та повідомлень тікетів «TicketMessages». Таблиця тікетів зберігає звернення користувачів — тему, текст, статус, категорію та відповідь адміністратора, а таблиця повідомлень забезпечує повноцінне листування в межах звернення і пов'язана з таблицею тікетів відношенням один-до-багатьох. Аналогічно до журналу дій, тікети та повідомлення зберігають денормалізоване ім'я автора, що дозволяє відображати листування коректно навіть після видалення облікового запису користувача.

Таблиця заблокованих адрес «BannedEmails» зберігає адреси електронної пошти, для яких заборонено повторну реєстрацію. Кожен запис містить заблоковану адресу з обмеженням унікальності, дату блокування та зовнішній ключ на адміністратора, який виконав блокування. Ця таблиця використовується

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		46

для запобігання повторній реєстрації користувачів, обліковий запис яких було видалено адміністратором.

Особливу увагу при проектуванні бази даних приділено правилам каскадного видалення, що забезпечують цілісність даних при видаленні пов'язаних записів. У системі застосовано два основні підходи до обробки видалення. Перший підхід — каскадне видалення — використовується для ресурсів, що належать користувачеві та не мають сенсу без нього: при видаленні користувача автоматично видаляються всі його контейнери, стеки, мережі, томи та квоти, а при видаленні контейнера — всі пов'язані порти, змінні середовища та монтування. Другий підхід — обнулення зовнішнього ключа — застосовується для записів, які повинні зберігатися після видалення пов'язаного об'єкта. Зокрема, при видаленні користувача записи журналу дій, тікети та повідомлення не видаляються, а лише втрачають зв'язок з обліковим записом, зберігаючи денормалізоване ім'я користувача для цілісності історії. Аналогічно, видалення стеку не призводить до видалення контейнерів, що входили до його складу.

Структуру бази даних спроектовано відповідно до вимог третьої нормальної форми. Всі атрибути таблиць є атомарними, що відповідає першій нормальної формі. Кожна таблиця має сурогатний первинний ключ, а всі неключові атрибути повністю залежать від первинного ключа, що відповідає другій нормальної формі. Відсутність транзитивних залежностей між неключовими атрибутами забезпечує відповідність третій нормальної формі. Відношення багато-до-багатьох винесено в окремі зв'язкові таблиці, що є стандартним підходом нормалізації реляційних баз даних.

Водночас у структурі бази даних свідомо застосовано контрольоване денормалізаційне дублювання окремих даних. Зокрема, у таблицях журналу дій, тікетів та повідомлень тікетів ім'я користувача зберігається безпосередньо, попри наявність зовнішнього ключа на таблицю користувачів. Таке рішення є обґрунтованим компромісом: воно дозволяє зберегти читабельність історичних записів після видалення облікового запису користувача, коли зовнішній ключ

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		47

обнуляється. Втрата суворої нормалізації у цьому випадку компенсується підвищенням надійності та інформативності системи аудиту.

2.4. Реалізація механізмів безпеки та мультитенантної ізоляції

Оскільки програмний комплекс призначений для колективного використання єдиного Docker-хоста кількома незалежними користувачами, забезпечення безпеки та ізоляції ресурсів є одним з найважливіших аспектів його реалізації. Надання користувачам доступу до керування контейнерами потенційно дозволяє виконувати операції з підвищеними правами на сервері, тому система реалізує комплекс механізмів захисту на всіх рівнях — від ізоляції даних між користувачами до валідації конфігурацій та захисту від типових векторів атак. У цьому підрозділі розглянуто реалізацію ключових модулів системи та застосовані механізми безпеки.

Мультитенантна ізоляція ресурсів

Основою безпеки мультитенантної системи є суворі ізоляція ресурсів між користувачами, за якої кожен користувач має доступ виключно до власних контейнерів, стеків, мереж та томів. Загрозою, що виникає за відсутності такої ізоляції, є вразливість класу IDOR (Insecure Direct Object Reference), за якої зломисник отримує доступ до чужих ресурсів шляхом маніпуляції їх ідентифікаторами у запитах [26]. Наприклад, користувач, змінивши ідентифікатор контейнера у запиті, міг би отримати контроль над контейнером іншого користувача.

Для запобігання таким вразливостям у програмному комплексі реалізовано єдину точку перевірки прав власності для кожного типу ресурсу [2]. Перед виконанням будь-якої операції з ресурсом сервіс перевіряє, чи належить запитаний ресурс користувачеві, що ініціював запит. Ідентифікатор користувача та його роль визначаються виключно на основі токена доступу, а не на основі даних, переданих у тілі запиту, що унеможлиблює їх підміну. Якщо ресурс не належить користувачеві і користувач не має ролі адміністратора, операція відхиляється з поверненням помилки доступу. Адміністратор як єдиний

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		48

привілейований суб'єкт системи має доступ до ресурсів усіх користувачів, що необхідно для виконання функцій адміністрування.

Такий підхід застосовано послідовно до всіх типів ресурсів — контейнерів, стеків, мереж і томів. Перевірка прав власності виконується на рівні сервісів бізнес-логіки, а не на рівні контролерів, що гарантує неможливість обходу перевірки незалежно від того, через яку кінцеву точку надійшов запит. Коректність роботи механізму ізоляції підтверджено набором автоматизованих тестів, які перевіряють, що спроба доступу до чужих ресурсів блокується, а доступ адміністратора — дозволяється.

Квотування ресурсів та захист від станів гонки

Для запобігання монополізації ресурсів сервера окремими користувачами система реалізує механізм квотування. Кожному користувачеві встановлюються обмеження на максимальну кількість контейнерів, сумарний обсяг оперативної пам'яті та сумарну кількість процесорних ядер. Перед створенням нового контейнера або збільшенням його ресурсних лімітів система перевіряє, чи не буде перевищено встановлену квоту з урахуванням уже наявних ресурсів користувача.

Реалізація перевірки квот наражається на загрозу виникнення стану гонки, відомого як проблема TOCTOU (Time-of-Check to Time-of-Use). Суть проблеми полягає у тому, що між моментом перевірки квоти та моментом фактичного створення ресурсу існує проміжок часу, протягом якого стан системи може змінитися. Якщо користувач надсилає кілька запитів на створення контейнерів одночасно, кожен з них може пройти перевірку квоти незалежно, оскільки на момент перевірки жоден з контейнерів ще не створений. У результаті сумарна кількість створених контейнерів може перевищити встановлений ліміт, тобто квота буде обійдена через паралельне виконання запитів.

Для усунення цієї загрози у програмному комплексі застосовано механізм семафорів, що серіалізує операції перевірки та створення ресурсів у межах кожного користувача. Перед виконанням перевірки квоти запит захоплює семафор, унікальний для конкретного користувача, після чого перевірка квоти та

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		49

фактичне створення ресурсу виконуються як єдина неподільна операція в межах захопленої критичної секції. Доки перший запит користувача не завершив перевірку та створення, наступні запити того ж користувача очікують звільнення семафора. Коли наступний запит отримує доступ до критичної секції, він бачить уже оновлений стан системи з урахуванням створеного ресурсу і отримує відмову у разі перевищення квоти.

Послідовність взаємодії компонентів системи при створенні контейнера, що демонструє роботу механізму захисту квот, наведено (рис. 2.4). Запит користувача надходить до контролера, який делегує обробку сервісу контейнерів. Перед перевіркою квоти сервіс захоплює семафор користувача, після чого в межах критичної секції виконується перевірка квоти та діапазону портів, створення контейнера в Docker Engine та збереження інформації про нього в базі даних. Лише після завершення всіх цих операцій семафор звільняється, що унеможливлює паралельне створення контейнерів понад встановлену квоту.

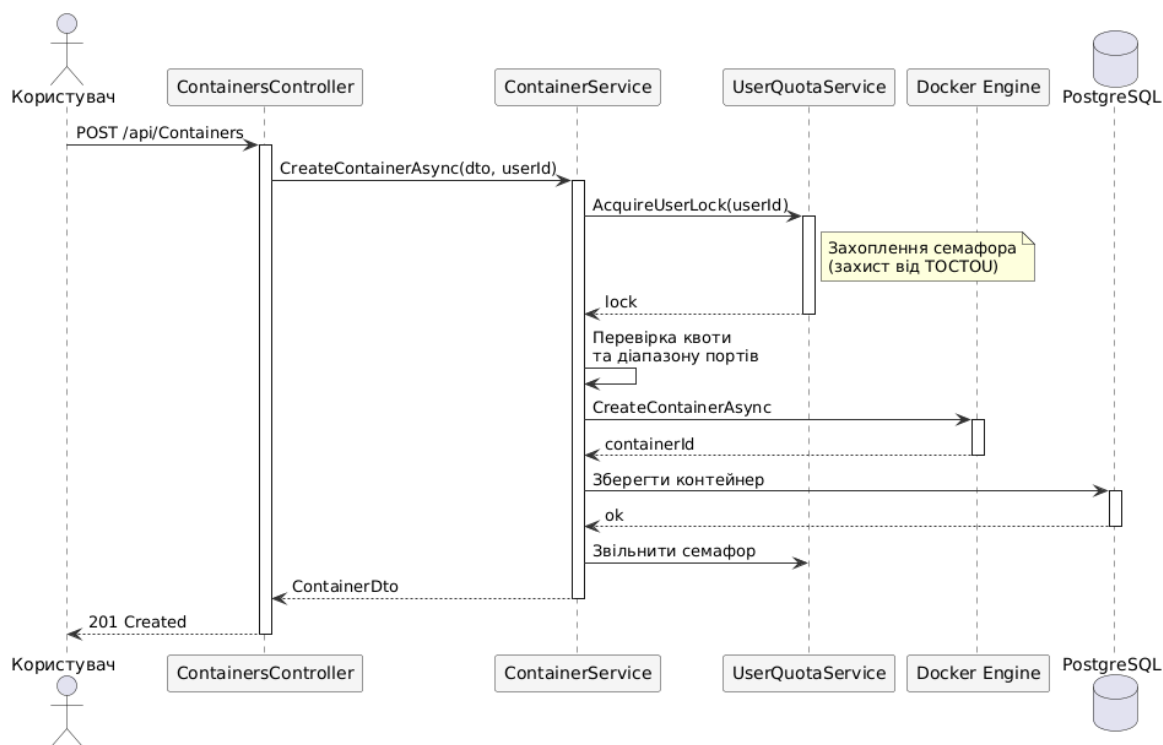


Рис. 2.4. Діаграма послідовності створення контейнера

Важливою особливістю реалізації є застосування окремого семафора для кожного користувача, а не єдиного глобального блокування. Такий підхід

гарантує, що операції одного користувача не блокують операції інших, оскільки стан гонки можливий лише в межах ресурсів окремого користувача. Це дозволяє уникнути виникнення вузького місця у продуктивності системи при одночасній роботі багатьох користувачів. Слід зазначити, що застосований механізм семафорів забезпечує захист у межах одного екземпляра застосунку. У разі горизонтального масштабування системи на кілька екземплярів для коректного квотування знадобився б розподілений механізм блокування, проте для розгортання на єдиному вузлі, передбаченого у межах роботи, наявне рішення є достатнім. Реалізацію механізму захисту квот від станів гонки наведено у додатку Б.

Ізоляція мережевих портів

У мультитенантному середовищі, де кілька користувачів одночасно створюють контейнери на одному хості, виникає проблема конфлікту мережевих портів. Якщо два користувачі спробують прив'язати свої контейнери до одного й того ж порту хостової системи, виникне помилка, оскільки порт може бути зайнятий лише одним процесом. Окрім того, відсутність розмежування портів дозволила б одному користувачеві зайняти порти, потенційно потрібні іншим.

Для вирішення цієї проблеми у програмному комплексі реалізовано механізм автоматичного виділення кожному користувачеві персонального діапазону мережевих портів. Діапазон обчислюється детерміновано на основі унікального ідентифікатора користувача за формулою:

$$startPort = 20000 + userId * 20$$

де *startPort* — початковий порт діапазону користувача; *userId* — унікальний ідентифікатор користувача.

Кінцевий порт діапазону визначається як початковий порт плюс дев'ятнадцять, що забезпечує виділення кожному користувачеві фіксованого пулу з двадцяти портів. Початок діапазону встановлено на рівні порту 20000, що дозволяє уникнути конфліктів із системними та широко вживаними портами. Детермінований характер формули гарантує, що діапазони різних користувачів не перетинаються, а обчислення діапазону не потребує зберігання додаткового

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		51

стану в базі даних — діапазон завжди може бути відновлений на основі ідентифікатора користувача.

Окрім розмежування діапазонів, система підтримує перелік зарезервованих портів, заборонених для використання всіма користувачами, включно з адміністратором. До таких портів належать стандартні порти системних служб та служб самого програмного комплексу, зокрема порти веб-сервера, бази даних та інших критичних компонентів. При спробі прив'язати контейнер до зарезервованого порту або до порту поза межами власного діапазону система відхиляє операцію з відповідним повідомленням. Адміністратор системи не обмежений персональним діапазоном портів, проте заборона на використання зарезервованих портів діє і для нього.

Автентифікація та керування доступом

Автентифікація користувачів реалізована на основі стандарту JSON Web Token [25]. При успішному вході користувача система генерує підписаний токен, що містить ідентифікатор користувача, ім'я, адресу електронної пошти, роль та версію ролі. Токен підписується за алгоритмом HMAC-SHA256 із застосуванням секретного ключа, відомого лише серверу [4]. Під час запуску застосунку система перевіряє довжину секретного ключа і відмовляється запускатися у разі недостатньої його довжини, що запобігає використанню слабкого ключа, вразливого до підбору.

Термін дії токена обмежено однією годиною, що зменшує вікно можливого зловживання у разі компрометації токена. При перевірці токена допуск за часом встановлено рівним нулю, що означає сувору перевірку терміну дії без додаткового запасу. Таке налаштування забезпечує точність роботи механізму відкликання прав доступу. Для встановлення з'єднання SignalR, де передавання токена у заголовок неможливе, токен передається як параметр запиту виключно для шляхів хабів реального часу.

Особливістю реалізації є механізм миттєвого відкликання раніше виданих токенів. Кожен токен містить версію ролі користувача, що зберігається в базі даних. При кожному запиті спеціальний проміжний обробник звіряє версію ролі

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		52

з токена з актуальним значенням у базі даних, а також перевіряє існування облікового запису. Якщо роль користувача було змінено або обліковий запис видалено, версія ролі у базі даних оновлюється, і всі раніше видані токени стають недійсними негайно, незважаючи на те, що термін їх дії ще не вичерпано. Такий підхід вирішує відому проблему безстанових токенів, за якої відкликання токена до завершення терміну його дії є складним завданням.

Захист паролів реалізовано за допомогою алгоритму хешування BCrypt. Паролі ніколи не зберігаються у відкритому вигляді — у базі даних зберігається лише їх хеш, обчислений з використанням солі. До паролів застосовуються вимоги мінімальної складності щодо довжини та складу символів. Адреси електронної пошти нормалізуються до нижнього регістру перед збереженням, що запобігає реєстрації дублікатів облікових записів з різним регістром символів. Повідомлення про помилку автентифікації не розрізняє випадки відсутності користувача та невірного паролю, що ускладнює зловмиснику визначення наявності облікового запису з певною адресою.

Посилення безпеки конфігурацій Docker Compose

Можливість розгортання багатокомпонентних застосунків на основі стеків Docker Compose є однією з найпотужніших функцій системи, проте водночас і однією з найнебезпечніших. Конфігурація Docker Compose у форматі YAML дозволяє задавати широкий спектр параметрів контейнерів, серед яких є потенційно небезпечні, що можуть надати контейнеру привілейований доступ до хостової системи. Якщо дозволити користувачам розгортати довільні конфігурації без обмежень, зловмисник міг би отримати повний контроль над сервером, наприклад, змонтувавши системні директорії хоста або запустивши контейнер у привілейованому режимі.

Для запобігання таким загрозам у програмному комплексі реалізовано механізм посилення безпеки конфігурацій. Перед розгортанням кожна конфігурація стеку аналізується на наявність небезпечних параметрів. Система блокує розгортання, якщо конфігурація містить привілейований режим, який надає контейнеру майже необмежений доступ до хостової системи. Блокуються

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		53

також небезпечні мережеві режими, що дозволяють контейнеру використовувати мережевий стек хоста або іншого контейнера, та режими спільного використання простору процесів, міжпроцесної взаємодії та імені вузла з хостом. Забороняється використання параметрів, що дозволяють пряме звернення до пристроїв хоста, зміну батьківської контрольної групи та послаблення профілів безпеки.

Окрему увагу приділено контролю прав монтування директорій. Система дозволяє монтування у контейнери лише з обмеженого переліку дозволених директорій, призначених спеціально для зберігання даних користувачів. Це запобігає монтуванню критичних системних шляхів, зокрема сокета Docker, монтування якого фактично надало б користувачеві права суперкористувача на сервері. Щодо системних можливостей Linux, система застосовує принцип білого списку — дозволяються лише чітко визначений набір базових можливостей, необхідних для типової роботи контейнерів, тоді як усі інші можливості блокуються.

Окрім блокування небезпечних параметрів, система здійснює автоматичне посилення конфігурації. До кожного сервісу у конфігурації стеку автоматично додається параметр, що забороняє підвищення привілеїв процесами всередині контейнера. Ця операція виконується ідемпотентно, тобто повторне застосування не призводить до дублювання параметра. Додатково система вимагає обов'язкового зазначення обмежень ресурсів для кожного сервісу стеку, що інтегрує розгортання стеків із загальною системою квотування. Коректність роботи механізму посилення безпеки конфігурацій підтверджено найбільшою групою автоматизованих тестів, які перевіряють блокування кожного з небезпечних параметрів. Реалізацію механізму посилення безпеки наведено у додатку В.

Захист файлового менеджера від виходу за межі дозволених директорій

Функція файлового менеджера, що дозволяє переглядати, читати та записувати файли всередині контейнерів через браузер, потребує особливих заходів безпеки. Основною загрозою для подібних функцій є атака класу path

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		54

traversal, за якої зловмисник шляхом використання спеціальних послідовностей у шляху до файлу намагається вийти за межі дозволеної директорії [26] та отримати доступ до критичних системних файлів. Прикладом такої атаки є використання послідовності переходу до батьківської директорії для доступу до файлів конфігурації операційної системи.

Для запобігання таким атакам у програмному комплексі реалізовано багаторівневу валідацію шляхів файлів. Перед виконанням будь-якої файлової операції шлях проходить перевірку на наявність небезпечних елементів — послідовностей переходу до батьківської директорії, нульових символів та інших потенційно небезпечних конструкцій. Додатково всі шляхи нормалізуються, що приводить їх до канонічного вигляду та унеможливорює обхід перевірки за допомогою різних способів запису одного й того ж шляху.

Система застосовує диференційований підхід до операцій читання та запису. Для операцій читання заблоковано доступ до службових директорій, що містять інформацію про процеси та пристрої системи. Для операцій запису, які є потенційно більш небезпечними, застосовується суворіший перелік заборонених директорій, що охоплює системні директорії з виконуваними файлами, конфігураціями та даними суперкористувача. Такий підхід дозволяє запобігти як читанню конфіденційної інформації, так і модифікації критичних системних файлів. Додатково обсяг файлів, доступних для читання та запису, обмежено, що запобігає перевантаженню системи при роботі з надмірно великими файлами. Коректність роботи механізму захисту підтверджено автоматизованими тестами, які перевіряють блокування спроб виходу за межі дозволених директорій.

Реалізація моніторингу ресурсів у реальному часі

Моніторинг використання ресурсів контейнерів у реальному часі є однією з ключових функцій системи. Для його реалізації застосовано поєднання фонових сервісів збору метрик та технології SignalR для передавання даних до браузера користувача. Збір метрик організовано у вигляді окремого фонових сервісу, що працює протягом усього часу роботи застосунку незалежно від запитів користувачів.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		55

Фоновий сервіс із заданою періодичністю звертається до Docker Engine для отримання поточних показників використання процесора та оперативної пам'яті кожного активного контейнера. Отримані метрики передаються підключеним клієнтам через хаб SignalR, який підтримує постійні WebSocket-з'єднання з браузерами користувачів. На відміну від підходу періодичного опитування сервера з боку клієнта, така архітектура забезпечує передавання оновлених даних у момент їх отримання з мінімальною затримкою та без зайвого навантаження на сервер.

Послідовність взаємодії компонентів при моніторингу ресурсів у реальному часі наведено (рис. 2.5). Фоновий сервіс із заданою періодичністю звертається до Docker Engine, отримує метрики використання процесора та оперативної пам'яті, перевіряє стан контейнерів та передає оновлені дані через хаб SignalR. Хаб, у свою чергу, надсилає дані клієнту через WebSocket-з'єднання, після чого клієнтська частина оновлює графіки моніторингу без перезавантаження сторінки.

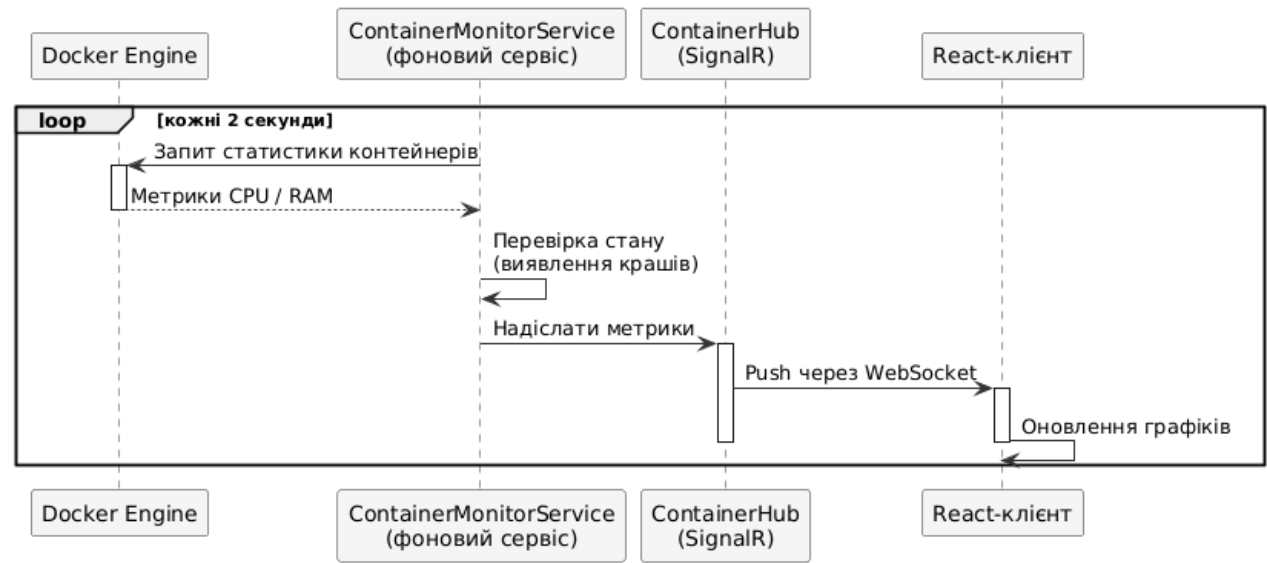


Рис. 2.5. Діаграма послідовності моніторингу ресурсів

Окрім збору метрик, фоновий сервіс виконує функцію виявлення аварійних завершень контейнерів. Сервіс відстежує зміни стану контейнерів і у разі несподіваного завершення роботи контейнера надсилає користувачеві відповідне сповіщення. Для коректної роботи цього механізму система розрізняє навмисно зупинені користувачем контейнери та контейнери, що завершилися

аварійно. Інформація про навмисну зупинку зберігається у спеціальному потокобезпечному кеші, що дозволяє уникнути надсилання хибних сповіщень про збій у випадках, коли користувач самостійно зупинив контейнер. Завдяки цьому сповіщення про аварійне завершення надсилаються лише у дійсно критичних ситуаціях, що підвищує інформативність системи моніторингу.

Додаткові механізми захисту

Окрім описаних вище основних механізмів, у програмному комплексі реалізовано низку додаткових заходів безпеки, що забезпечують комплексний захист системи. Для запобігання атакам підбору паролів та автоматизованим спробам реєстрації застосовано обмеження частоти запитів до кінцевих точок входу та реєстрації. Кількість запитів з однієї адреси протягом визначеного проміжку часу обмежено, а при перевищенні ліміту запити тимчасово відхиляються. Це суттєво ускладнює проведення атак грубої сили проти облікових записів користувачів.

До кожної відповіді сервера додаються заголовки безпеки, що захищають клієнтську частину від поширених атак. Зокрема, застосовано заголовки, що забороняють визначення типу вмісту браузером всупереч зазначеному, забороняють відображення сторінок застосунку у вбудованих фреймах для запобігання атакам класу clickjacking, обмежують передавання інформації про джерело переходу та визначають політику безпеки вмісту. Політика безпеки вмісту налаштована з урахуванням необхідності роботи WebSocket-з'єднань для функціонування компонентів реального часу.

Для забезпечення коректної роботи системи за зворотним проксі-сервером реалізовано обробку заголовків переадресації, що дозволяє визначати реальну адресу клієнта. Це є важливим для коректної роботи механізму обмеження частоти запитів, оскільки без такої обробки всі запити сприймалися б як такі, що надходять з адреси проксі-сервера. Довіра до заголовків переадресації обмежена лише визначеними адресами проксі-сервера, що запобігає підміні адреси клієнта злоумисником.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		57

Взаємодія між клієнтською та серверною частинами захищена за допомогою механізму CORS, що дозволяє доступ до API лише з визначеного переліку дозволених джерел. Усі винятки, що виникають під час обробки запитів, перехоплюються централізованим обробником, який повертає клієнту коректне повідомлення про помилку без розкриття внутрішніх деталей реалізації системи. Такий підхід запобігає витоку конфіденційної технічної інформації, що могла б бути використана зловмисником для аналізу вразливостей системи. Контейнер серверної частини застосунку працює від імені непривілейованого користувача, а доступ до сокета Docker надано лише для читання, що додатково обмежує можливі наслідки потенційної компрометації системи.

2.5. Реалізація ключових модулів системи

Окрім механізмів безпеки, програмний комплекс реалізує низку функціональних модулів, що забезпечують повноцінне керування інфраструктурними примітивами Docker. У цьому підрозділі розглянуто реалізацію найбільш складних з технічної точки зору модулів — життєвий цикл контейнерів, розгортання стеків Docker Compose, керування мережами та томами, інтерактивного термінала та файлового менеджера.

Керування життєвим циклом контейнерів

Базовим модулем системи є керування повним життєвим циклом Docker-контейнерів. Контейнер протягом свого існування проходить через низку станів, а переходи між ними ініціюються діями користувача. Після створення контейнер перебуває у початковому стані, з якого може бути запущений. Запущений контейнер може бути зупинений, перезапущений, призупинений або примусово завершений. Призупинений контейнер може бути відновлений, а зупинений — запущений повторно або видалений. Діаграму станів контейнера з можливими переходами наведено (рис. 2.6).

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		58

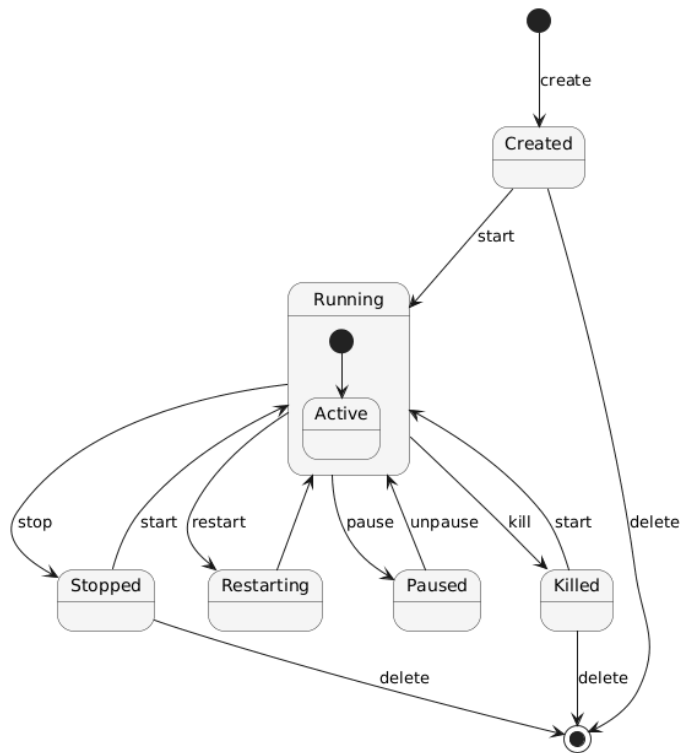


Рис. 2.6. Діаграма станів життєвого циклу контейнера

Особливістю реалізації є те, що кожен перехід між станами супроводжується перевіркою його допустимості. Система не дозволяє виконувати операції, що суперечать поточному стану контейнера, — наприклад, запустити вже запущений контейнер або призупинити зупинений. Такі перевірки реалізовано на рівні сервісу бізнес-логіки, що запобігає виникненню некоректних станів та помилок при взаємодії з Docker Engine. Коректність перевірок допустимості переходів підтверджено окремою групою автоматизованих тестів.

Розгортання стеків Docker Compose

Модуль стеків забезпечує розгортання багатокомпонентних застосунків, що складаються з кількох пов'язаних контейнерів, на основі конфігурації Docker Compose. Користувач задає конфігурацію стеку у форматі YAML, після чого система здійснює її розбір, валідацію та розгортання. Розбір конфігурації виконується за допомогою бібліотеки YamlDotNet, що перетворює текстовий опис у структуру об'єктів для подальшого аналізу.

Перед розгортанням конфігурація проходить перевірку безпеки, описану в попередньому підрозділі, після чого до неї автоматично застосовуються

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		59

параметри посилення. Безпосереднє розгортання стеку виконується через виклик інструменту Docker Compose, для чого у контейнер серверної частини застосунку інтегровано відповідний компонент. Після успішного розгортання система синхронізує інформацію про створені контейнери з власною базою даних, пов'язуючи їх зі стеком, що дозволяє відстежувати належність контейнерів до конкретного стеку та керувати ними як єдиним цілим.

Особливістю реалізації модуля стеків є підтримка потокового передавання журналів процесу розгортання. Оскільки розгортання багатокомпонентного застосунку може тривати значний час, користувачеві важливо бачити перебіг процесу в реальному часі. Для цього журнали розгортання передаються до браузера через хаб SignalR у міру їх появи, що дозволяє користувачеві спостерігати за процесом завантаження образів та запуску контейнерів без очікування завершення всієї операції.

Інтерактивний термінал

Модуль інтерактивного термінала надає користувачеві можливість виконувати команди безпосередньо всередині контейнера через веб-браузер, що є функціональним еквівалентом підключення до контейнера через командний рядок без потреби у прямому доступі до сервера. Реалізація термінала поєднує клієнтську бібліотеку емуляції термінала та серверний механізм взаємодії з контейнером через постійне з'єднання.

На клієнтській частині термінал побудовано на основі бібліотеки xterm.js, що забезпечує повноцінну емуляцію термінального вікна у браузері з підтримкою керуючих послідовностей, кольорового виведення та коректного відображення символів у кодуванні UTF-8, включно з кириличними. На серверній частині при відкритті термінала система створює всередині цільового контейнера сеанс виконання команд з виділенням псевдотермінала. Виділення псевдотермінала є необхідним для коректної роботи інтерактивних програм, що очікують підключення до справжнього термінала, та забезпечує підтримку функцій редагування командного рядка і відображення запрошення введення.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		60

Обмін даними між терміналом у браузері та сеансом усередині контейнера організовано через хаб SignalR на основі постійного WebSocket-з'єднання. Символи, що вводить користувач, передаються на сервер і транслуються у контейнер, а результати виконання команд негайно повертаються до браузера та відображаються у термінальному вікні. Система також підтримує зміну розміру термінального вікна — при зміні розмірів вікна браузера нові розміри передаються на сервер, що забезпечує коректне відображення виведення програм, чутливих до розмірів термінала. Завдяки такій реалізації користувач отримує повноцінний інтерактивний доступ до контейнера, не залишаючи веб-інтерфейсу системи.

Файловий менеджер

Модуль файлового менеджера надає користувачеві можливість переглядати вміст файлової системи контейнера, читати та редагувати файли безпосередньо через веб-інтерфейс без необхідності підключення до контейнера через командний рядок. Ця функція є зручною для перегляду конфігураційних файлів, журналів застосунків та внесення оперативних змін без перезапуску контейнера.

Технічно доступ до файлів усередині контейнера реалізовано через відповідний механізм Docker, що дозволяє передавати файли між контейнером та зовнішньою системою у вигляді архівів. При запиті на перегляд директорії система отримує перелік її вмісту, а при запиті на читання файлу — його вміст, який передається до браузера для відображення. Запис файлу виконується у зворотному напрямку: змінений вміст передається від браузера до системи та записується у відповідний файл усередині контейнера.

Оскільки файловий менеджер надає прямий доступ до файлової системи контейнера, його реалізація приділяє особливу увагу безпеці. Усі шляхи проходять валідацію для запобігання виходу за межі дозволених директорій, як описано у підрозділі 2.4 «Реалізація механізмів безпеки та мультитенантної ізоляції». Додатково обсяг файлів, доступних для читання та запису, обмежено визначеним розміром, що запобігає вичерпанню ресурсів системи при роботі з

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		61

надмірно великими файлами. Завдяки поєднанню зручності використання та суворих заходів безпеки файловий менеджер забезпечує безпечний доступ до вмісту контейнерів у мультитенантному середовищі.

Висновки до розділу 2

У другому розділі розглянуто технологічний стек, архітектуру, структуру бази даних та реалізацію ключових модулів програмного комплексу.

Обґрунтовано вибір технологічного стеку, що включає платформу ASP.NET Core 8 для серверної частини, бібліотеку React 19 з мовою TypeScript для клієнтської частини, систему керування базами даних PostgreSQL, бібліотеку SignalR для організації комунікації в реальному часі та бібліотеку Docker.DotNet для взаємодії з Docker Engine. Вибір кожного інструменту обґрунтовано порівнянням з альтернативними рішеннями за критеріями типізації, продуктивності та зрілості екосистеми.

Описано трирівневу архітектуру програмного комплексу з чітким поділом на рівень представлення, рівень бізнес-логіки та рівень доступу до даних. Розглянуто застосовані патерни проєктування — Repository, Dependency Injection та Data Transfer Object, а також дотримання принципів об'єктно-орієнтованого програмування та SOLID, що забезпечують гнучкість, тестованість та зручність супроводу коду.

Спроектовано структуру бази даних, що складається з шістнадцяти таблиць, об'єднаних у чотири логічні групи. Базу даних нормалізовано відповідно до вимог третьої нормальної форми, а для відношень багато-до-багатьох застосовано зв'язкові таблиці. Обґрунтовано застосування контрольованої денормалізації для забезпечення цілісності історичних записів аудиту після видалення облікових записів користувачів.

Детально розглянуто реалізацію механізмів безпеки та мультитенантної ізоляції, що є найважливішим аспектом системи колективного використання. Описано захист від вразливостей класу IDOR через перевірку прав власності на рівні сервісів, захист квот від станів гонки за допомогою семафорів, автоматичне

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		62

виділення ізолюваних діапазонів портів, механізм посилення безпеки конфігурацій Docker Compose, захист файлового менеджера від виходу за межі дозволених директорій, а також механізми автентифікації з можливістю миттєвого відкликання прав доступу. Розглянуто реалізацію ключових функціональних модулів — розгортання стеків Docker Compose, керування мережами та томами, інтерактивного термінала та файлового менеджера. Застосовані рішення забезпечують безпечне та ізолюване середовище для колективного керування контейнерною інфраструктурою.

		Черноштан Т.Ю			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		63

РОЗДІЛ 3. РОЗГОРТАННЯ ТА ЕКСПЛУАТАЦІЯ СИСТЕМИ

3.1. Розгортання та автоматизація доставки

Розгортання програмного комплексу організовано на основі технології контейнеризації, що забезпечує відтворюваність середовища та незалежність від конфігурації конкретного сервера. Уся система упакована у вигляді набору Docker-контейнерів, які описуються та запускаються за допомогою інструменту Docker Compose. Такий підхід дозволяє розгорнути повноцінну систему на будь-якому сервері з установленим Docker єдиною командою, не вимагаючи ручного встановлення та налаштування окремих компонентів.

Програмний комплекс складається з трьох контейнеризованих компонентів, що працюють спільно. Першим компонентом є контейнер серверної частини, побудований на основі образу середовища виконання .NET. Другим компонентом є контейнер клієнтської частини, що містить зібраний статичний застосунок React, який обслуговується веб-сервером [24]. Третім компонентом є контейнер бази даних PostgreSQL, що зберігає всі дані системи. Взаємодія між компонентами організована через внутрішню мережу Docker, у якій контейнери звертаються один до одного за іменами сервісів.

Серверна частина застосунку збирається з використанням багатоетапного складання Docker. На першому етапі у середовищі з повним набором інструментів розробки виконується відновлення залежностей та компіляція проєкту у режимі випуску [11]. На другому етапі скомпільований застосунок переноситься у легкий образ середовища виконання, що містить лише компоненти, необхідні для запуску. Такий підхід суттєво зменшує підсумковий розмір образу, оскільки інструменти розробки не потрапляють до фінального контейнера. Додатково до образу серверної частини встановлюється клієнт командного рядка Docker та компонент Docker Compose, необхідні для функції розгортання стеків з боку самого застосунку.

З міркувань безпеки контейнер серверної частини налаштовано на роботу від імені непривілейованого користувача, а не від імені суперкористувача, що є

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		64

типовим для контейнерів за замовчуванням. Сокет Docker монтується до контейнера серверної частини у режимі лише для читання, що обмежує можливі наслідки потенційної компрометації застосунку. Конфіденційні параметри конфігурації, зокрема пароль бази даних та секретний ключ для підпису токенів, передаються до контейнерів через змінні середовища і не зберігаються у вихідному коді чи системі контролю версій.

Слід зазначити, що цільовою платформою розгортання програмного комплексу є серверні операційні системи на базі Linux. Це зумовлено особливостями взаємодії з Docker Engine: серверна частина застосунку звертається до демона Docker через Unix-сокет, а функції розгортання стеків, моніторингу та інтерактивного термінала розраховані на нативне Linux-середовище виконання контейнерів. Розгортання на серверах під керуванням Linux відповідає типовій практиці експлуатації контейнеризованих застосунків, оскільки переважна більшість виробничих Docker-хостів працює саме на цій платформі.

Контейнер бази даних налаштовано з перевіркою готовності, що дозволяє системі визначити момент, коли база даних готова приймати з'єднання. Контейнер серверної частини запускається лише після того, як база даних повністю готова до роботи, що запобігає помилкам підключення під час запуску системи. Дані бази зберігаються у іменованому томі Docker, що забезпечує їх збереження незалежно від життєвого циклу контейнера бази даних.

Для автоматизації процесів збирання та перевірки якості коду налаштовано конвеєр безперервної інтеграції на основі платформи GitLab [22]. Репозиторій проєкту розміщено на самостійно розгорнутому сервері GitLab навчального закладу, що забезпечує повний контроль над процесом розробки та виконанням завдань конвеєра. Конвеєр описується конфігураційним файлом у корені репозиторію та автоматично запускається при кожному надходженні змін до відстежуваних гілок.

Конвеєр складається з трьох послідовних стадій — складання, тестування та формування образів. На стадії складання виконується відновлення

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		65

залежностей серверної частини та її компіляція у режимі випуску. Ця стадія підтверджує, що код не містить помилок компіляції і може бути успішно зібраний. У разі виявлення помилок на цьому етапі подальші стадії не виконуються, що дозволяє швидко виявити проблему ще до запуску тестів. Конфігураційні файли контейнеризації та конвеєра безперервної інтеграції наведено у додатку Д.

На стадії тестування запускається набір автоматизованих тестів серверної частини. Тести перевіряють коректність роботи бізнес-логіки, зокрема механізмів безпеки, квотування та валідації, які є критично важливими для системи. Результати виконання тестів зберігаються у вигляді звіту, що дозволяє відстежувати стан тестового покриття. Якщо хоча б один тест завершується невдало, стадія вважається проваленою, а зміни не просуваються далі конвеєром — це запобігає потраплянню некоректного коду до основної гілки.

На стадії формування образів виконується збирання Docker-образів серверної та клієнтської частин застосунку. Ця стадія підтверджує, що застосунок не лише компілюється та проходить тести, а й може бути коректно упакований у контейнери, готові до розгортання. Виконання завдань конвеєра здійснюється на самостійно налаштованому виконавці GitLab, що працює у середовищі з підтримкою Docker, необхідною для збирання образів.

Послідовне виконання стадій реалізує принцип швидкого зворотного зв'язку: помилки компіляції виявляються найпершими, далі перевіряється коректність логіки тестами, і лише після успішного проходження обох етапів виконується ресурсомістка операція збирання образів. Такий порядок мінімізує витрати часу на виявлення проблем та підвищує надійність процесу доставки.

Організація роботи з репозиторієм побудована на основі моделі з двома основними гілками. Гілка основної версії містить стабільний код, що пройшов перевірку та готовий до розгортання, і захищена від прямого внесення змін. Гілка розробки використовується для інтеграції нового функціоналу перед його перенесенням до основної гілки. Розробка окремих функцій ведеться у відповідних гілках, які після завершення роботи об'єднуються з гілкою розробки.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		66

Така модель запобігає потраплянню неперевіреного коду до стабільної версії та забезпечує впорядкований процес розробки.

Для оформлення повідомлень комітів застосовано угоду про структуровані повідомлення, за якою кожне повідомлення містить префікс, що вказує на тип внесених змін [27] — додавання нового функціоналу, виправлення помилки, рефакторинг чи зміну документації. Дотримання єдиного стандарту оформлення комітів забезпечує зрозумілу та інформативну історію змін, що спрощує відстеження розвитку проєкту та пошук причин виникнення проблем.

Результат успішного виконання конвеєра безперервної інтеграції наведено (рис. 3.1). Усі стадії конвеєра — складання, тестування та формування образів — завершилися успішно, що підтверджує коректність коду, проходження автоматизованих тестів та готовність застосунку до розгортання.

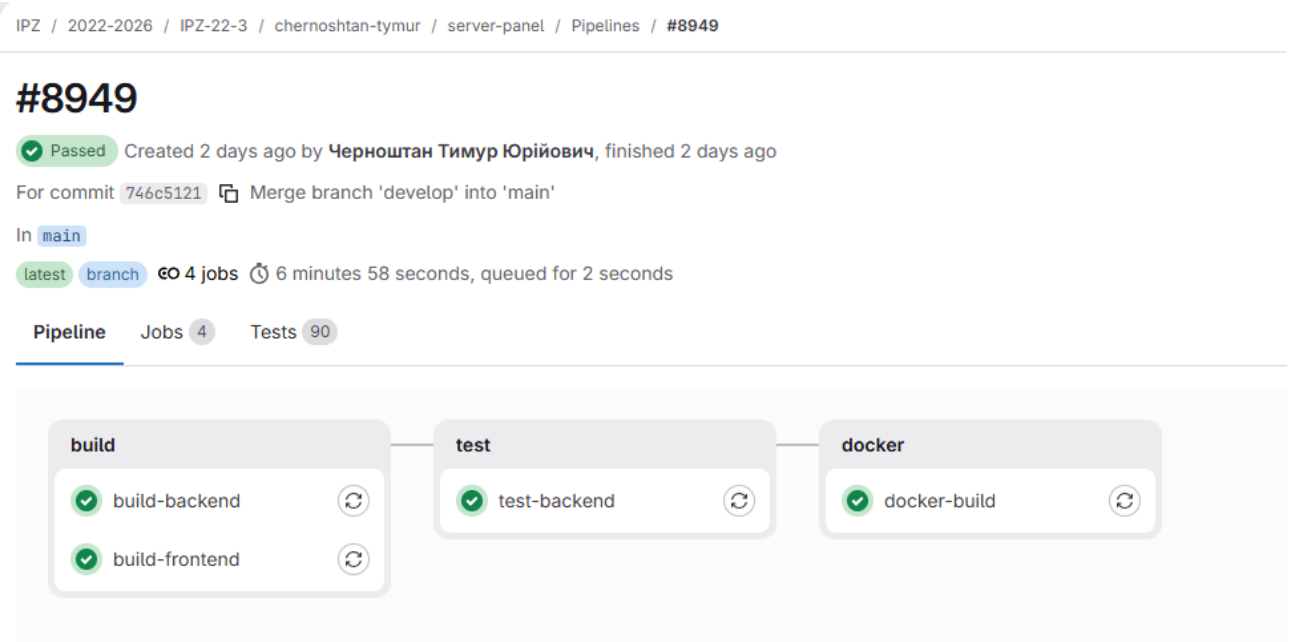


Рис. 3.1. Результат успішного виконання конвеєра CI/CD

Загальну схему розгортання програмного комплексу наведено (рис. 3.2). Усі компоненти системи розгортаються на сервері під керуванням Linux у вигляді ізольованих Docker-контейнерів. Користувач взаємодіє із системою через веб-браузер, який звертається до клієнтської частини за протоколом передавання гіпертексту та до серверної частини через REST API і постійні WebSocket-з'єднання. Серверна частина взаємодіє з базою даних PostgreSQL та з демоном Docker через відповідний сокет, керуючи контейнерами користувачів.

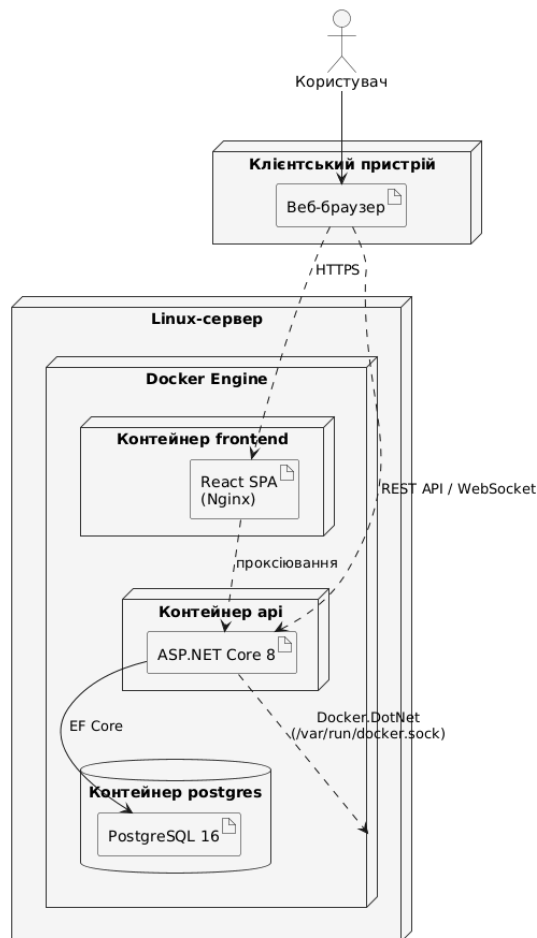


Рис. 3.2. Діаграма розгортання програмного комплексу

3.2. Опис інтерфейсу користувача

Клієнтська частина програмного комплексу реалізована у вигляді односторінкового застосунку, що забезпечує зручну взаємодію користувача з системою через веб-браузер. Інтерфейс побудовано за принципом розмежування доступу: набір доступних користувачеві функцій залежить від його ролі в системі. У цьому підрозділі розглянуто основні екрани інтерфейсу, описано їх призначення та проаналізовано механізми, що активуються на серверній частині під час роботи з кожним екраном.

Вхід та реєстрація

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		68

Першим екраном, з яким взаємодіє користувач, є сторінка автентифікації. Вона містить форму входу з полями для введення адреси електронної пошти та пароллю, а також можливість переходу до форми реєстрації нового облікового запису. Вигляд сторінки входу наведено (рис. 3.3).

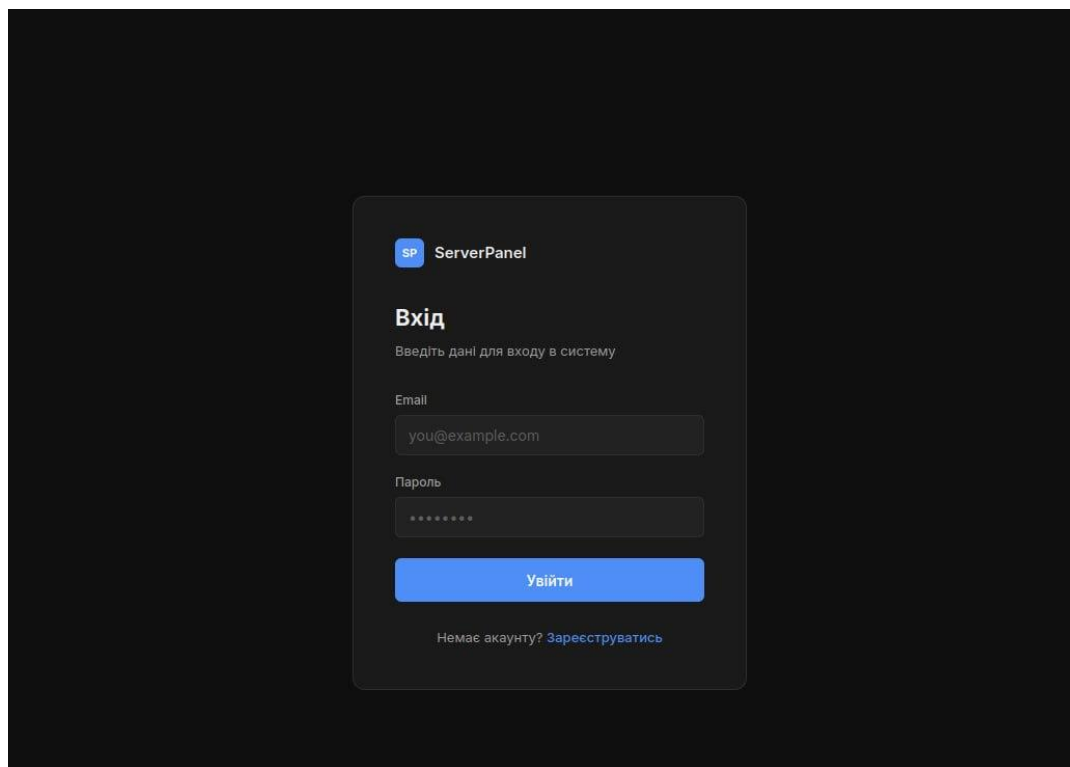


Рис. 3.3. Сторінка входу до системи

При надсиланні форми входу клієнтська частина передає введені дані на сервер, де активується кілька механізмів безпеки. Передусім запит проходить через обмеження частоти звернень, що захищає систему від автоматизованого підбору паролів. Введений пароль порівнюється з його хешем, збереженим у базі даних, за допомогою алгоритму BCrypt. У разі успішної автентифікації сервер генерує токен доступу, що містить ідентифікатор користувача, його роль та версію ролі, після чого токен зберігається на клієнтській частині та додається до всіх подальших запитів. При реєстрації нового облікового запису система додатково перевіряє унікальність адреси електронної пошти та імені користувача, відповідність пароллю вимогам складності, а також відсутність адреси у переліку заблокованих.

Головна панель користувача

		Черноштан Т.Ю			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		69

Після успішної автентифікації користувач потрапляє на головну панель, що надає узагальнену інформацію про стан його ресурсів. Панель містить картки статистики із загальною кількістю контейнерів та кількістю запущених і зупинених, а також картки квот, що відображають поточне використання та встановлені обмеження за кількістю контейнерів, обсягом оперативної пам'яті та процесорними ядрами. Додатково панель відображає перелік останніх контейнерів та останні події з журналу дій користувача. Вигляд головної панелі наведено (рис. 3.4).

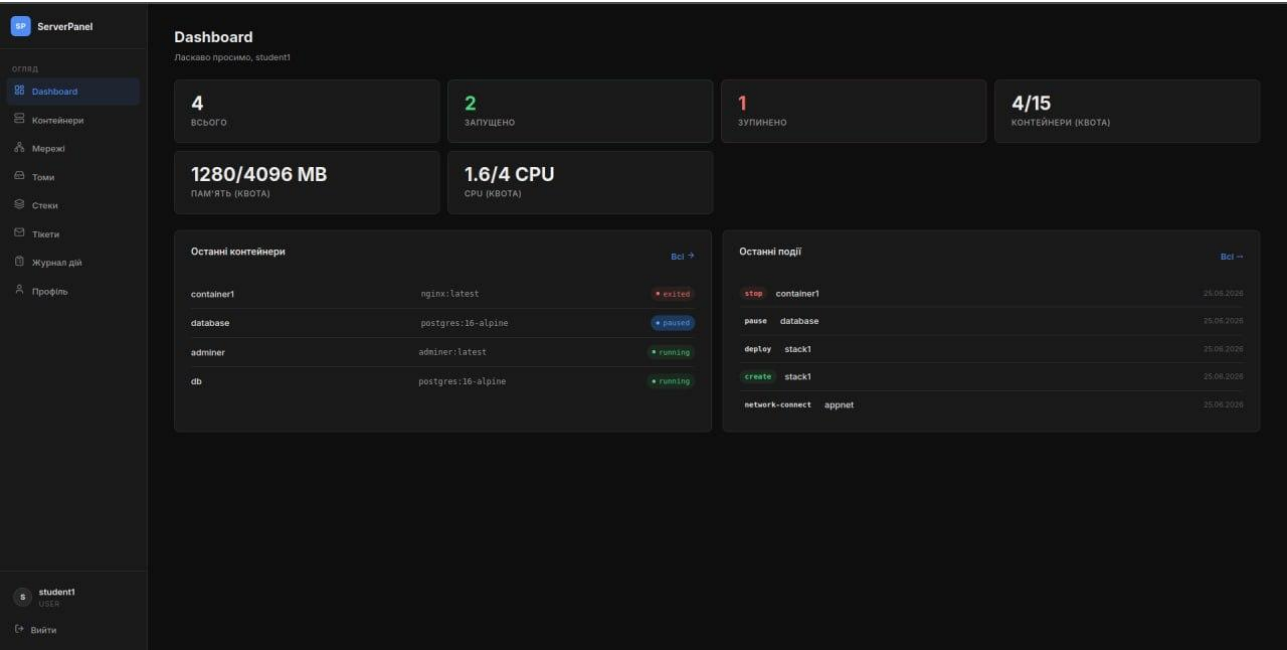


Рис. 3.4. Головна панель користувача

У разі наближення до межі квоти або її перевищення на панелі відображається попередження з пропозицією звернутися до адміністратора через систему тікетів. Дані для панелі завантажуються з кількох джерел — статистики ресурсів, інформації про квоти та журналу дій, що дозволяє користувачеві оцінити стан своїх ресурсів з одного екрана.

Керування контейнерами

Центральним екраном системи є сторінка переліку контейнерів, що відображає всі контейнери користувача у вигляді таблиці. Для кожного контейнера показано назву, образ, поточний статус, показники використання процесора та оперативної пам'яті, прив'язані порти та доступні дії. Особливістю цієї сторінки є відображення показників використання ресурсів у реальному часі.

Значення завантаження процесора та пам'яті для кожного контейнера оновлюються безпосередньо у таблиці завдяки постійному з'єднанню SignalR — індикатор стану з'єднання сигналізує про активність каналу реального часу. Вигляд сторінки керування контейнерами наведено (рис. 3.5).

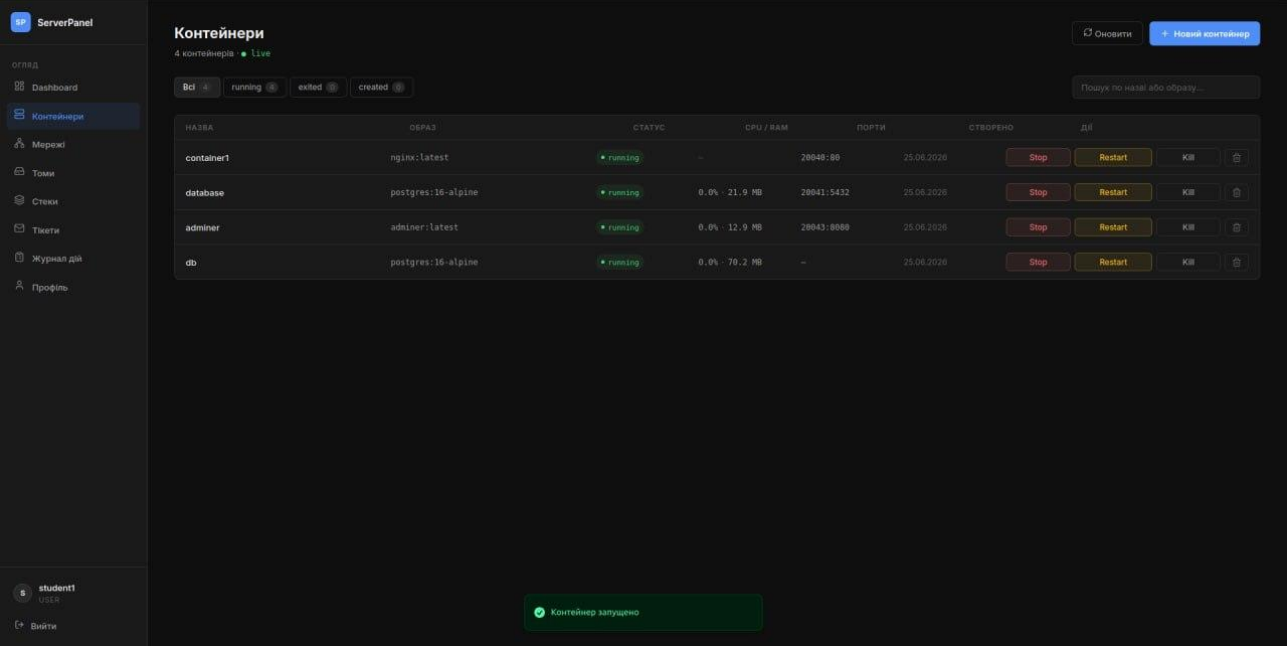


Рис. 3.5. Сторінка керування контейнерами

Сторінка надає засоби фільтрації контейнерів за статусом з відображенням лічильників, а також пошук за назвою та образом. Для кожного контейнера доступні дії керування життєвим циклом — запуск, зупинка, перезапуск, примусове завершення та видалення. Потенційно небезпечні дії, зокрема видалення, супроводжуються діалогом підтвердження, що для видалення вимагає введення назви контейнера. Слід підкреслити, що перелік відображає виключно контейнери поточного користувача завдяки механізму мультитенантної ізоляції — на серверній стороні запит фільтрується за ідентифікатором користувача з токена доступу.

Створення контейнера

Для створення нового контейнера передбачено окрему сторінку з формою, що дозволяє задати всі параметри майбутнього контейнера. Форма містить секцію вибору шаблону, що дозволяє швидко застосувати попередньо налаштовану конфігурацію, поле введення назви та вибору образу з переліком

тегів популярних образів. Вигляд сторінки створення контейнера наведено (рис. 3.6).

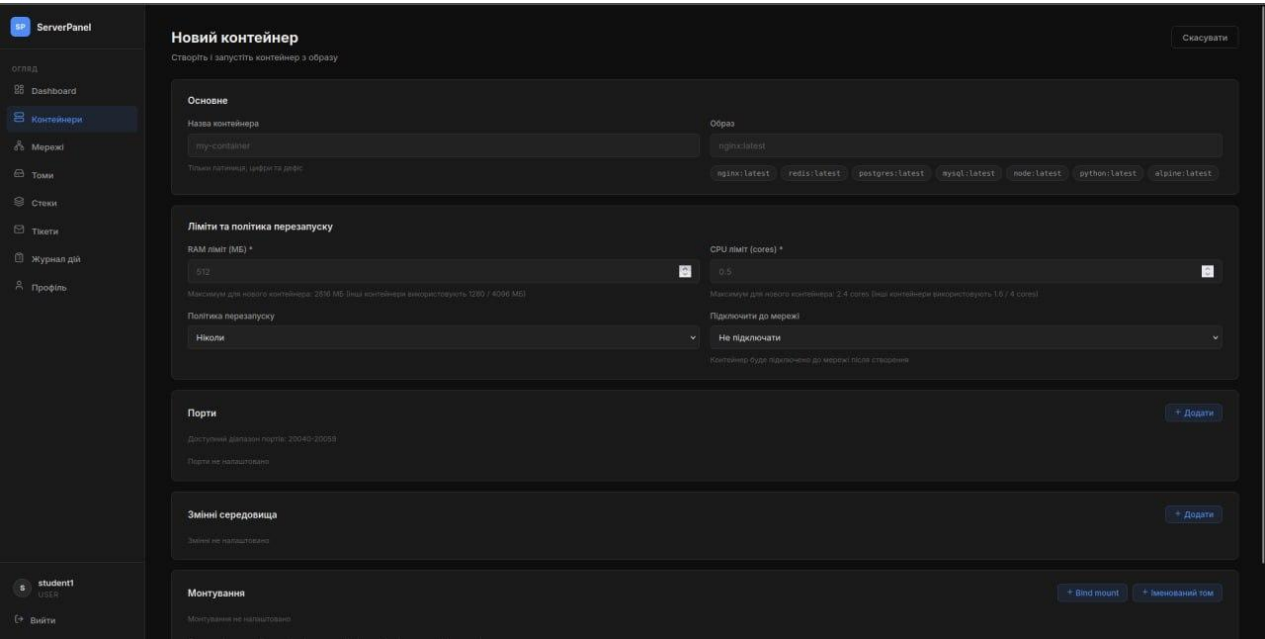


Рис. 3.6. Сторінка створення контейнера

Форма дозволяє задати обмеження ресурсів — обсяг оперативної пам'яті та кількість процесорних ядер, політику перезавантаження, прив'язку портів, змінні середовища та монтування томів. При виборі образу система автоматично пропонує типові для нього порти. Для прив'язки портів передбачено функцію пропонування вільного порту в межах діапазону, виділеного користувачеві. У процесі створення контейнера на серверній стороні активуються одразу кілька механізмів: перевірка квоти користувача з захистом від станів гонки, валідація належності портів до діапазону користувача та перевірка відсутності зарезервованих портів. Обмеження ресурсів є обов'язковими, що гарантує врахування створеного контейнера у системі квотування та запобігає обходу встановлених лімітів.

Перегляд деталей контейнера

Сторінка деталей контейнера надає повну інформацію про окремий контейнер та засоби керування ним. У верхній частині сторінки розташовано кнопки керування життєвим циклом, що відображаються контекстно залежно від поточного стану контейнера. Інформація про контейнер організована у вигляді

кількох вкладок — огляду, журналів, файлів, інспекції та налаштувань. Вигляд вкладки огляду наведено (рис. 3.7).

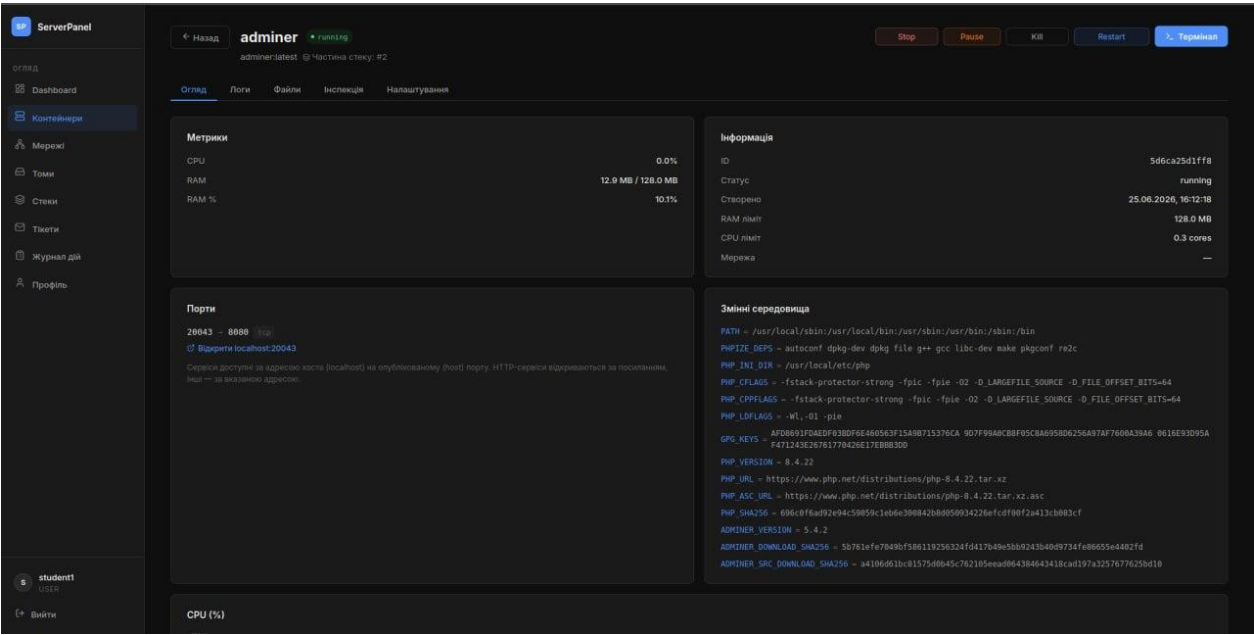


Рис. 3.7. Вкладка огляду контейнера

Вкладка огляду відображає поточні показники використання ресурсів, загальну інформацію про контейнер, прив'язані порти та змінні середовища. Ключовим елементом вкладки є графіки використання процесора та оперативної пам'яті, що оновлюються у реальному часі та відображають динаміку споживання ресурсів за останній період. Графіки реалізовано власним компонентом візуалізації на основі векторної графіки без застосування сторонніх бібліотек, а дані для них надходять через постійне з'єднання SignalR. Така реалізація демонструє можливість побудови інформативних засобів візуалізації власними силами без додаткових залежностей.

Вкладка журналів дозволяє переглядати вивід контейнера із можливістю налаштування кількості відображуваних рядків. На відміну від журналів стеків, журнали окремого контейнера завантажуються за запитом користувача, що є достатнім для діагностики роботи окремого застосунку.

Вкладка файлів надає доступ до файлового менеджера контейнера, що дозволяє переглядати структуру файлової системи, читати та редагувати текстові файли безпосередньо у браузері. Навігація організована за допомогою

послідовності каталогів, що відображає поточне розташування. Вигляд файлового менеджера наведено (рис. 3.8).

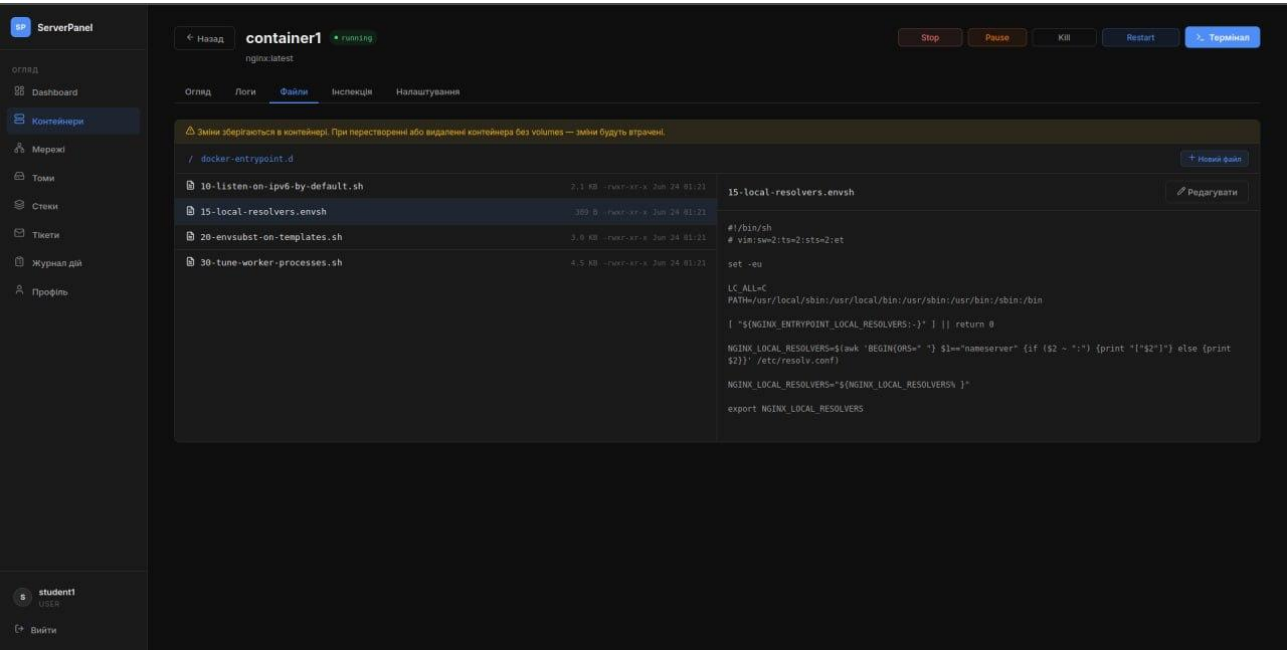


Рис. 3.8. Файловий менеджер контейнера

Під час роботи з файлами на серверній стороні активується механізм захисту від виходу за межі дозволених директорій, описаний у попередньому розділі. Кожен запит на читання чи запис файлу проходить валідацію шляху, що блокує доступ до системних директорій та запобігає виходу за межі файлової системи контейнера. Вкладка інспекції відображає детальну технічну інформацію про контейнер, отриману безпосередньо від Docker Engine, — стан, налаштування безпеки, мережі, точки монтування та параметри запуску. Вкладка налаштувань дозволяє змінювати конфігурацію контейнера з подальшим його перестворенням.

Інтерактивний термінал

Однією з найбільш технічно складних функцій інтерфейсу є інтерактивний термінал, що надає користувачеві повноцінний доступ до командного рядка всередині контейнера безпосередньо у браузері. Термінал відкривається на окремій сторінці та реалізований на основі бібліотеки емуляції термінала. Термінал підтримує повноцінне введення команд, відображення кольорового виводу, коректну роботу з символами у кодуванні UTF-8, включно з кириличними, а також зміну розміру вікна. Вигляд термінала наведено (рис. 3.9).

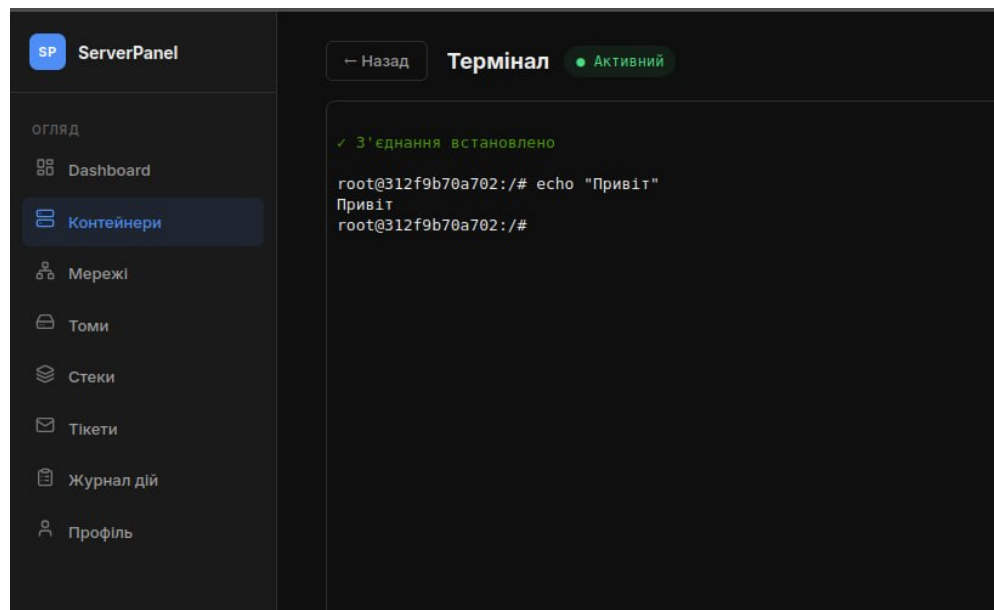


Рис. 3.9. Інтерактивний термінал контейнера

Взаємодія з контейнером організована через окреме з'єднання SignalR, відмінне від каналу моніторингу, що забезпечує двосторонній обмін даними в реальному часі. Введені користувачем команди передаються на сервер та виконуються у сеансі всередині контейнера з виділенням псевдотермінала, а результати негайно повертаються до браузера. Завдяки цьому користувач отримує досвід роботи, еквівалентний прямому підключенню до контейнера, не залишаючи веб-інтерфейсу та без потреби у відкритті доступу до сервера через SSH.

Керування мережами та томами

Для керування додатковими інфраструктурними примітивами Docker передбачено окремі сторінки мереж та томів. Сторінка мереж відображає перелік створених користувачем мереж та надає форму для створення нової мережі із зазначенням назви, драйвера та ознаки ізолюваності. Для кожної мережі відображається перелік підключених контейнерів з можливістю підключення нових та відключення наявних. Сторінка томів організована аналогічно та дозволяє створювати іменовані томи, переглядати перелік контейнерів, що їх використовують, та видаляти непотрібні томи. Вигляд сторінки керування мережами наведено (рис. 3.10).

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				75
Змн.	Арк.	№ докум.	Підпис	Дата		

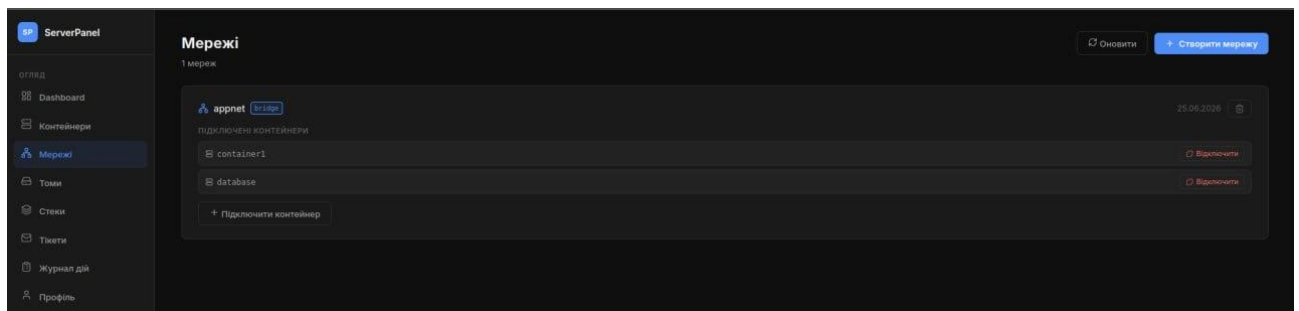


Рис. 3.10. Сторінка керування мережами

Як і всі інші ресурси системи, мережі та топи підпорядковані механізму мультитенантної ізоляції — користувач бачить та може керувати виключно власними ресурсами, тоді як системні мережі та топи самого програмного комплексу приховані.

Керування стеками Docker Compose

Сторінка стеків призначена для розгортання багатокомпонентних застосунків на основі конфігурації Docker Compose. Для створення стеку передбачено редактор коду, що забезпечує зручне редагування конфігурації у форматі YAML з підсвічуванням синтаксису. Вигляд сторінки редагування стеку наведено (рис. 3.11).

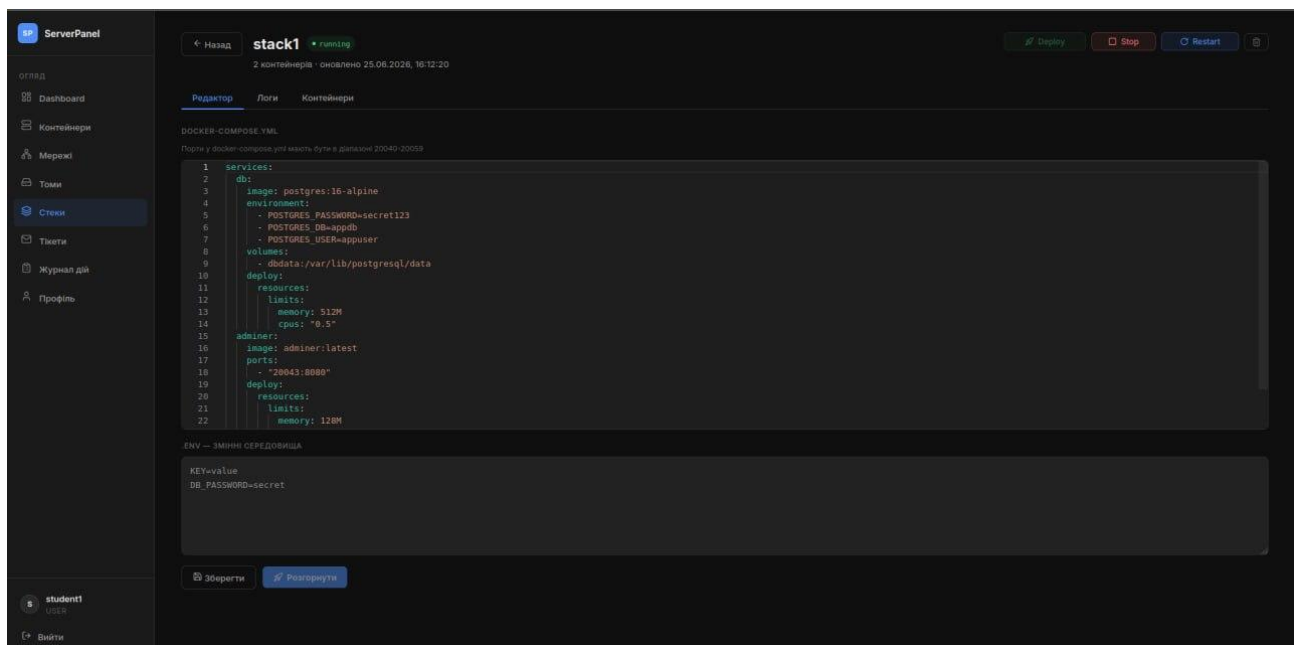


Рис. 3.11. Редактор конфігурації стеку Docker Compose

Сторінка деталей стеку містить вкладки редактора конфігурації, журналів та переліку контейнерів стеку. Журнали процесу розгортання передаються у реальному часі через з'єднання SignalR, що дозволяє спостерігати за перебігом

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				76
Змн.	Арк.	№ докум.	Підпис	Дата		

завантаження образів та запуску контейнерів безпосередньо під час операції. Перед розгортанням кожна конфігурація проходить перевірку безпеки. Дієвість механізму посилення безпеки конфігурацій (рис. 3.12). При спробі розгорнути стек, конфігурація якого містить привілейований режим, система відхиляє операцію з повідомленням про заборону небезпечного параметра, не допускаючи розгортання потенційно небезпечного контейнера.

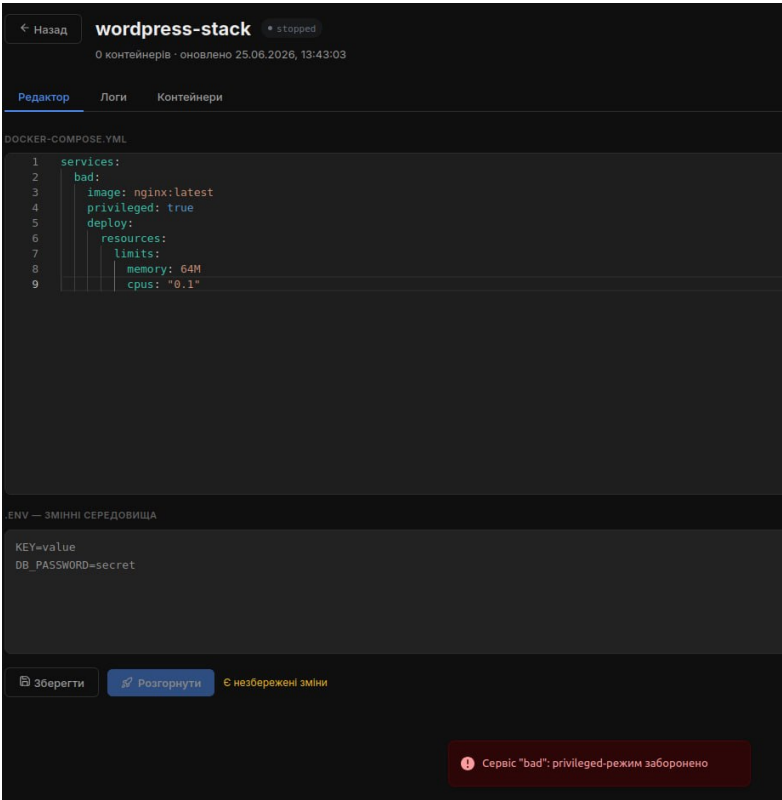


Рис. 3.12. Блокування розгортання небезпечної конфігурації стеку

Система тікетів підтримки

Для організації взаємодії між користувачами та адміністратором передбачено систему сервісних тікетів. Сторінка тікетів відображає перелік звернень користувача з їх статусами та надає форму створення нового звернення із зазначенням теми, категорії та тексту повідомлення. Сторінка окремого тікета організована у вигляді стрічки повідомлень, що реалізує повноцінне листування між користувачем та адміністратором. Користувач може надсилати відповіді у межах звернення після відповіді адміністратора, а адміністратор отримує додаткові засоби керування — зміну статусу звернення та надсилання відповідей. Вигляд сторінки тікета наведено (рис. 3.13).

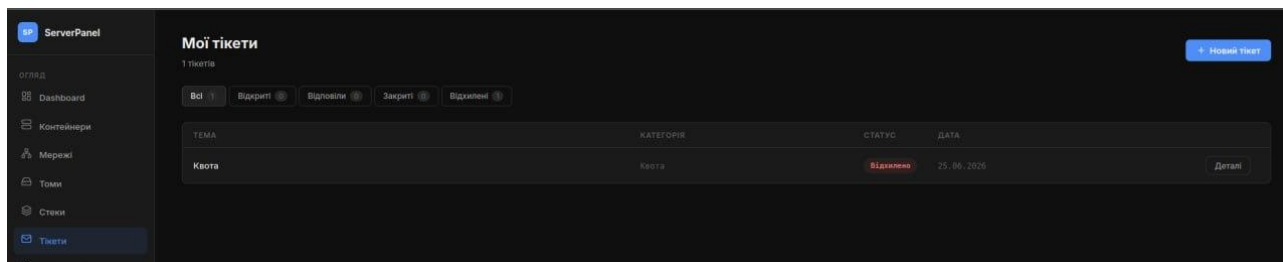


Рис. 3.13. Сторінка звернення до технічної підтримки

Про нові повідомлення та зміни статусу сторони сповіщаються у реальному часі через з'єднання SignalR. Сторінка профілю відображає інформацію про обліковий запис користувача — ім'я, роль, адресу електронної пошти та статистику використання ресурсів із поточними квотами.

Адміністративні екрани

Адміністратор системи має доступ до розширеного набору екранів, недоступних звичайним користувачам. Інтерфейс адміністратора побудовано на окремому компоновальному шаблоні, що відрізняється від інтерфейсу звичайного користувача наявністю додаткових розділів. Головна панель адміністратора відображає статистику всієї системи — загальну кількість користувачів, контейнерів та їх станів, а також останні події з журналу дій усіх користувачів.

Сторінка керування користувачами надає адміністратору засоби перегляду всіх облікових записів системи з можливістю пошуку та посторінкового виведення. Вигляд сторінки керування користувачами наведено (рис. 3.14).

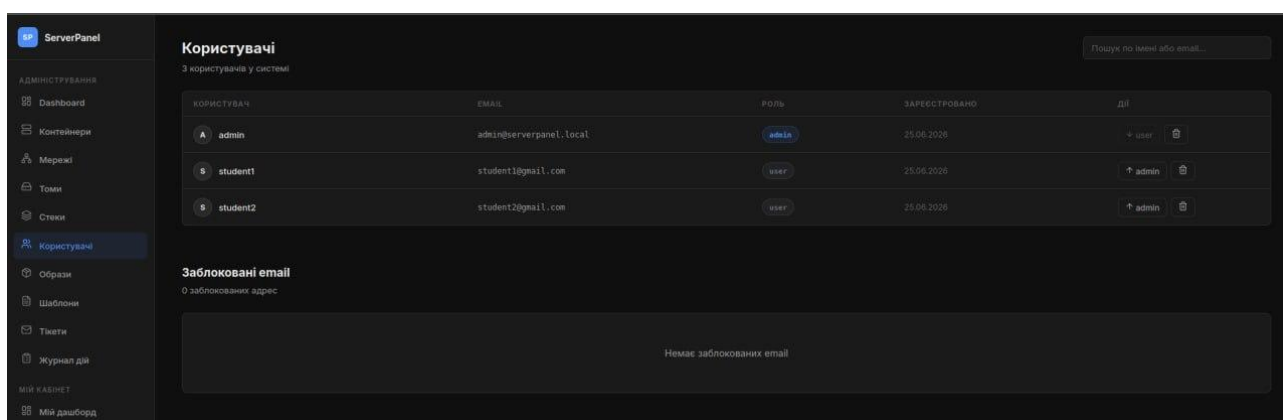
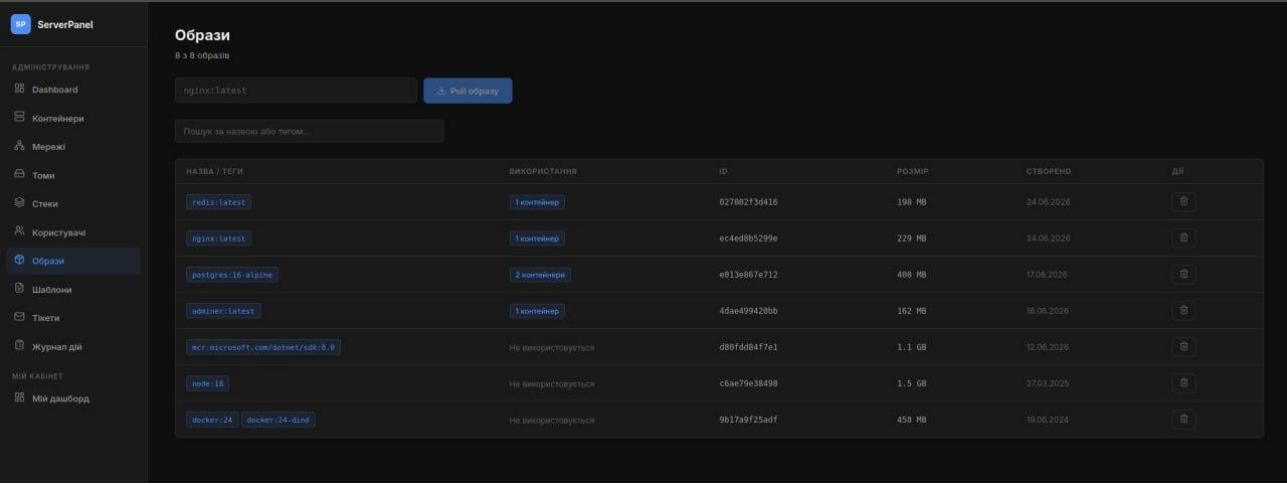


Рис. 3.14. Сторінка керування користувачами

Для кожного користувача доступні зміна ролі та видалення облікового запису, а також редагування квот ресурсів — кількості контейнерів, обсягу оперативної пам'яті та процесорних ядер.

Окремий розділ сторінки містить перелік заблокованих адрес електронної пошти з можливістю їх розблокування. При видаленні облікового запису користувача його адреса автоматично додається до переліку заблокованих, що запобігає повторній реєстрації.

Сторінка керування образами надає адміністратору засоби завантаження нових образів з реєстру, перегляду переліку наявних образів з інформацією про їх розмір та використання, а також видалення непотрібних образів. Вигляд сторінки керування образами наведено (рис. 3.15).



НАЗВА / ТЕГ	ВИКОРИСТАННЯ	ID	РОЗМІР	СТВОРЕНО	ДІЯ
redis:latest	1 контейнер	627f0273d416	198 MB	24.08.2026	🗑️
nginx:latest	1 контейнер	ec4ed8b5299e	229 MB	24.08.2026	🗑️
postgres:16-alpine	2 контейнери	e813e867e712	486 MB	17.08.2026	🗑️
adminer:latest	1 контейнер	4dae499420bb	162 MB	16.08.2026	🗑️
mcr.microsoft.com/dotnet/sdk:8.0	Не використовується	d89fd88477e1	1.1 GB	12.08.2026	🗑️
node:18	Не використовується	c6ae79e38498	1.5 GB	27.03.2025	🗑️
docker:24	Не використовується	9b17a9f15adf	458 MB	19.08.2024	🗑️

Рис. 3.15. Сторінка керування образами

Окрім розглянутих, адміністратору доступні сторінки керування шаблонами конфігурацій, а також адміністративні версії сторінок контейнерів, мереж, томів, стеків, журналу дій та тікетів, що відображають ресурси всіх користувачів системи на відміну від користувацьких версій, обмежених власними ресурсами.

Інтерфейс користувача забезпечує повноцінне керування контейнерною інфраструктурою через веб-браузер без необхідності використання командного рядка. Інтерфейс послідовно дотримується принципу розмежування доступу: звичайний користувач працює виключно з власними ресурсами, тоді як адміністратор отримує доступ до керування всією системою. Важливою

особливістю реалізації є те, що зручність графічного інтерфейсу не послаблює механізмів безпеки — кожна дія користувача на серверній стороні супроводжується відповідними перевірками ізоляції ресурсів, квот та валідації даних. Поєднання інформативної візуалізації стану ресурсів у реальному часі, повноцінного інтерактивного термінала та засобів керування всіма примітивами Docker формує цілісний інструмент, що за зручністю використання не поступається спеціалізованим командним засобам, водночас забезпечуючи безпеку мультитенантного середовища.

3.3. Тестування програмного комплексу

Забезпечення якості програмного комплексу є невід'ємною складовою процесу розробки, особливо для системи, що реалізує механізми безпеки та керування інфраструктурою. Помилки у логіці перевірки прав доступу, квотування чи валідації конфігурацій можуть призвести до серйозних наслідків — від порушення ізоляції між користувачами до компрометації сервера. Тому для перевірки коректності роботи критично важливих компонентів системи розроблено набір автоматизованих тестів.

Підхід до тестування ґрунтується на принципі модульного тестування, за якого окремі компоненти бізнес-логіки перевіряються ізольовано від зовнішніх залежностей [6]. Основну увагу при розробці тестів приділено саме механізмам безпеки та мультитенантної ізоляції, оскільки ці компоненти є найбільш критичними для коректної роботи системи колективного використання [5]. Загалом розроблено дев'яносто модульних тестів, що охоплюють ключові аспекти роботи серверної частини.

Для реалізації тестів застосовано низку спеціалізованих інструментів екосистеми .NET. Основою тестового набору є фреймворк xUnit — один з найпоширеніших інструментів модульного тестування для платформи .NET, що надає засоби організації тестів та перевірки очікуваних результатів. Для ізоляції тестованих компонентів від їх залежностей використано бібліотеку Moq, що дозволяє створювати тестові імітації залежностей — зокрема репозиторіїв та

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		80

зовнішніх сервісів. Завдяки цьому логіка сервісу перевіряється незалежно від реальної бази даних чи Docker Engine, що забезпечує швидкість виконання тестів та відтворюваність результатів.

Для тестів, що потребують роботи з базою даних, застосовано провайдер бази даних у оперативній пам'яті, що дозволяє перевіряти логіку роботи з даними без необхідності розгортання реальної бази даних PostgreSQL. Окремі інтеграційні перевірки використовують спеціальний механізм створення тестового екземпляра застосунку, що дозволяє перевірити взаємодію компонентів у наближених до реальних умовах. Таке поєднання інструментів забезпечує всебічну перевірку логіки системи на різних рівнях.

Розроблені тести згруповано за категоріями відповідно до компонентів системи, які вони перевіряють. Розподіл тестів за категоріями наведено у табл. 3.1.

Таблиця 3.1

Розподіл модульних тестів за категоріями

Категорія тестів	Кількість	Що перевіряється
Безпека стеків Docker Compose	20	Блокування небезпечних параметрів конфігурації
Ізоляція доступу до ресурсів	10	Захист від вразливостей класу IDOR
Виділення діапазонів портів	10	Коректність обчислення та ізоляції портів
Квотування ресурсів	8	Перевірка дотримання квот користувачів
Система тікетів	8	Логіка створення та обробки звернень
Безпека операцій з контейнерами	7	Перевірки прав при операціях з контейнерами
Керування користувачами	7	Реєстрація, автентифікація, зміна ролей
Допустимість переходів станів	6	Валідація операцій життєвого циклу

Продовження таблиці 3.1

Категорія тестів	Кількість	Що перевіряється
Валідація шляхів файлів	5	Захист від виходу за межі директорій
Обробка помилок Docker	4	Коректна обробка винятків Docker Engine
Нормалізація електронної пошти	4	Унікальність та коректність адрес
Інтеграційна перевірка	1	Взаємодія компонентів застосунку

Найбільшу групу тестів становлять перевірки безпеки стеків Docker Compose, що зумовлено критичною важливістю механізму посилення безпеки конфігурацій — кожен з потенційно небезпечних параметрів перевіряється окремим тестом. Значну увагу приділено також тестуванню ізоляції доступу до ресурсів та коректності виділення діапазонів портів, оскільки ці механізми є основою мультитенантної безпеки системи. Приклади реалізації модульних тестів безпеки наведено у додатку Г.

Розроблений набір тестів цілеспрямовано зосереджено на перевірці бізнес-логіки серверної частини, насамперед механізмів безпеки та мультитенантної ізоляції. Тестами повністю охоплено логіку перевірки прав власності на ресурси, обчислення та валідації діапазонів портів, перевірки квот із захистом від станів гонки, аналізу безпеки конфігурацій стеків, валідації шляхів файлових операцій та перевірки допустимості переходів між станами контейнерів. Саме ці компоненти містять найбільш складну логіку та є критичними з погляду безпеки, тому їх ретельна перевірка є пріоритетною.

Водночас свідомо залишено поза межами модульного тестування низку компонентів, перевірка яких потребувала б реального оточення Docker. Безпосередня взаємодія з Docker Engine, робота хабів SignalR у реальному часі та фоновий сервіс збору метрик не охоплені модульними тестами, оскільки їх повноцінна перевірка вимагала б розгортання реального Docker-середовища, що

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		82

виходить за межі модульного підходу. Логіка цих компонентів ізольована таким чином, що безпекові перевірки виконуються до звернення до Docker Engine та покриваються тестами окремо від самої взаємодії з ним. Перевірка коректності взаємодії з Docker та роботи компонентів реального часу здійснювалася шляхом ручного тестування у процесі розробки.

Такий підхід до тестування відповідає принципу раціонального розподілу зусиль: автоматизованими тестами покрито логіку, де ціна помилки найвища та де можлива ізольована перевірка, тоді як компоненти, тісно пов'язані із зовнішнім середовищем, перевірено вручну. Усі розроблені тести успішно проходять і виконуються автоматично на стадії тестування конвеєра безперервної інтеграції при кожній зміні коду, що гарантує своєчасне виявлення регресій та підтримання якості коду протягом усього процесу розробки. Результат виконання повного набору модульних тестів наведено (рис. 3.16). Усі 90 тестів успішно проходять, що підтверджує коректність роботи перевірених компонентів системи.

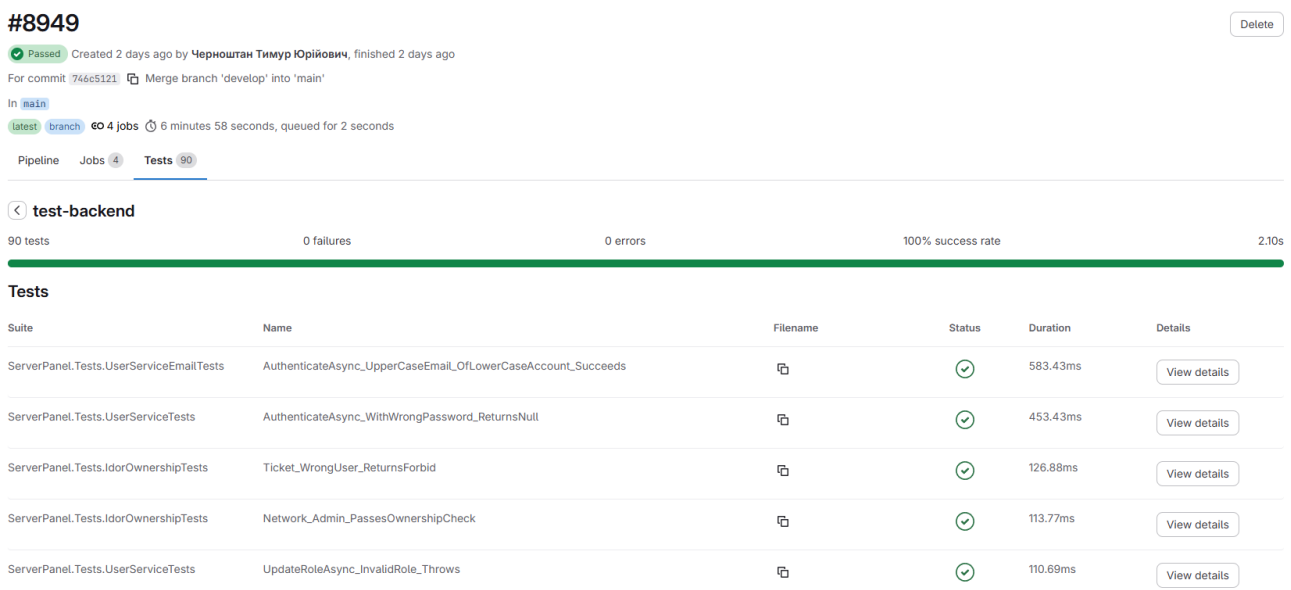


Рис. 3.16. Результат виконання модульних тестів

Висновки до розділу 3

У третьому розділі розглянуто питання розгортання, експлуатації та тестування програмного комплексу.

Описано організацію розгортання системи на основі технології контейнеризації. Програмний комплекс упаковано у вигляді трьох взаємопов'язаних Docker-контейнерів — серверної частини, клієнтської частини та бази даних, що описуються та запускаються за допомогою Docker Compose. Застосовано багатоетапне складання образів, роботу від імені непривілейованого користувача та монтування сокета Docker у режимі лише для читання, що підвищує безпеку розгортання. Налаштовано конвеєр безперервної інтеграції на основі GitLab, що автоматично виконує складання, тестування та формування образів при кожній зміні коду, реалізуючи принцип швидкого зворотного зв'язку.

Детально розглянуто інтерфейс користувача програмного комплексу. Описано основні екрани системи — від автентифікації та головної панелі до керування контейнерами, стеками, мережами та томами, а також адміністративні екрани. Для кожного екрана проаналізовано не лише його призначення та елементи, а й механізми, що активуються на серверній стороні під час роботи з ним. Показано, що зручність графічного інтерфейсу поєднується із послідовним застосуванням механізмів безпеки на всіх рівнях взаємодії.

Розглянуто підхід до тестування програмного комплексу. Розроблено дев'яносто модульних тестів на основі фреймворку xUnit та бібліотеки Moq, що охоплюють критично важливі компоненти системи — механізми ізоляції ресурсів, квотування, виділення портів, безпеки конфігурацій стеків та валідації файлових операцій. Обґрунтовано раціональний розподіл зусиль на тестування, за якого автоматизованими тестами покрито логіку з найвищою ціною помилки, а компоненти, тісно пов'язані із зовнішнім середовищем Docker, перевірено вручну. Усі розроблені тести успішно проходять у складі конвеєра безперервної інтеграції.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		84

ВИСНОВКИ

У кваліфікаційній роботі розв'язано актуальне завдання розроблення веб-орієнтованої мультитенантної системи керування Docker-інфраструктурою, що забезпечує безпечне колективне використання єдиного контейнерного середовища кількома незалежними користувачами.

Проаналізовано предметну галузь керування контейнерною інфраструктурою та розглянуто наявні рішення — Portainer, Pterodactyl та CasaOS. Виявлено, що наявні системи не повною мірою задовольняють потребу в безпечному мультитенантному середовищі з суворою ізоляцією ресурсів та квотуванням, що обґрунтувало доцільність розроблення власного рішення. На основі аналізу сформульовано вимоги до системи та визначено її функціональні можливості.

Обґрунтовано вибір технологічного стеку, що включає платформу ASP.NET Core 8 для серверної частини, бібліотеку React з мовою TypeScript для клієнтської частини, систему керування базами даних PostgreSQL, технологію SignalR для організації взаємодії в реальному часі та бібліотеку Docker.DotNet для взаємодії з Docker Engine. Спроектовано трирівневу архітектуру системи із застосуванням патернів Repository, Dependency Injection та Data Transfer Object і дотриманням принципів об'єктно-орієнтованого програмування та SOLID.

Спроектовано та реалізовано структуру бази даних із шістнадцяти таблиць, нормалізовану відповідно до вимог третьої нормальної форми із застосуванням контрольованої денормалізації для забезпечення цілісності історичних записів аудиту.

Реалізовано комплекс механізмів безпеки та мультитенантної ізоляції, що є ключовою особливістю системи. Розроблено захист від вразливостей класу IDOR через перевірку прав власності на рівні бізнес-логіки, захист квот від станів гонки за допомогою семафорів, автоматичне виділення ізольованих діапазонів портів, механізм посилення безпеки конфігурацій Docker Compose та захист файлових операцій від виходу за межі дозволених директорій. Реалізовано

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		85

автентифікацію на основі токенів JWT з можливістю миттєвого відкликання прав доступу.

Реалізовано повний набір функціональних можливостей системи — керування життєвим циклом контейнерів, розгортання багатокомпонентних застосунків на основі стеків Docker Compose, керування мережами та томами, моніторинг використання ресурсів у реальному часі, інтерактивний термінал та файловий менеджер, що працюють безпосередньо у веб-браузері.

Організовано розгортання системи на основі контейнеризації з налаштованим конвеєром безперервної інтеграції. Розроблено набір з дев'яноста модульних тестів, що підтверджують коректність роботи критично важливих компонентів системи.

Розроблений програмний комплекс є завершеним рішенням, що поєднує зручність графічного інтерфейсу з надійними механізмами безпеки, та може бути використаний у навчальних закладах, невеликих командах розробників та інших середовищах, де потрібне безпечне колективне керування контейнерною інфраструктурою. Перспективними напрямками подальшого розвитку системи є впровадження динамічної рольової моделі доступу, підтримка горизонтального масштабування з розподіленим механізмом блокування квот та інтеграція з системами централізованого моніторингу.

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		86

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Малець О. Б., Андрушко В. М. Аналіз процесів використання Docker для побудови мікросервісів. Науковий вісник НЛТУ України. 2017. Вип. 27, № 2. С. 245–249. URL: <https://nv.nltu.edu.ua/index.php/journal/article/view/1388> (дата звернення: 20.06.2026).
2. Савчук К. Гібридні стратегії кібербезпеки для веб-додатків з використанням штучного інтелекту. Кібербезпека: освіта, наука, техніка. 2025. Т. 3, № 31. С. 323–332. DOI: <https://doi.org/10.28925/2663-4023.2025.31.969>.
3. Трофименко О. Г., Манаков С. Ю., Чикунів П. О., Лобода Ю. Г., Гурін Є. О. Маршрутизація в мікрофронтендних застосунках: архітектурні підходи, виклики та шляхи їх вирішення. Кібербезпека: освіта, наука, техніка. 2025. Т. 3, № 31. С. 635–651. DOI: <https://doi.org/10.28925/2663-4023.2025.31.1057>.
4. Хавікова Ю. Захист інформації в електронних послугах. Кібербезпека: освіта, наука, техніка. 2024. Т. 1, № 25. С. 287–298. URL: <https://csecurity.kubg.edu.ua/index.php/journal/article/view/649> (дата звернення: 20.06.2026).
5. Кравчук Н. В., Коробейнікова Т. І. Аналіз методів та засобів тестування безпеки веб-застосунків. Вісник ЛДУБЖД. 2024. № 30. С. 78–86. URL: <https://journal.ldubgd.edu.ua/index.php/Visnuk/article/view/2770> (дата звернення: 20.06.2026).
6. Кудрявцев О. Актуальність проведення модульного тестування при розробленні програмного забезпечення. Управління розвитком складних систем. 2021. № 45. С. 75–81. URL: <http://mdcs.knuba.edu.ua/article/view/238006> (дата звернення: 21.06.2026).
7. Вплив технології Docker на архітектуру мікросервісів. Наукові записки Державного університету інформаційно-комунікаційних технологій. 2023. URL: <https://journals.dut.edu.ua/index.php/sciencenotes/article/view/2844> (дата звернення: 21.06.2026).

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		87

8. Мартін Р. Чиста архітектура: мистецтво розробки програмного забезпечення / пер. з англ. І. Бондар-Терещенко. Київ: Фабула, 2019. 368 с.
9. Mouat A. Using Docker: Developing and Deploying Software with Containers. Sebastopol: O'Reilly Media, 2016. 362 p.
10. Provos N., Mazières D. A Future-Adaptable Password Scheme. Proceedings of the USENIX Annual Technical Conference. 1999. P. 81–91. URL: https://www.usenix.org/legacy/events/usenix99/full_papers/provos/provos.pdf (дата звернення: 22.06.2026).
11. Docker Documentation. Docker Inc. URL: <https://docs.docker.com> (дата звернення: 22.06.2026).
12. Docker Engine API. Docker Inc. URL: <https://docs.docker.com/engine/api/> (дата звернення: 22.06.2026).
13. ASP.NET Core documentation. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/aspnet/core/> (дата звернення: 23.06.2026).
14. Overview of ASP.NET Core SignalR. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction> (дата звернення: 23.06.2026).
15. Entity Framework Core documentation. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/ef/core/> (дата звернення: 23.06.2026).
16. Dependency injection in ASP.NET Core. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection> (дата звернення: 23.06.2026).
17. React Documentation. Meta Open Source. URL: <https://react.dev> (дата звернення: 23.06.2026).
18. TypeScript Documentation. Microsoft. URL: <https://www.typescriptlang.org/docs/> (дата звернення: 23.06.2026).
19. Vite Guide. Vite. URL: <https://vite.dev/guide/> (дата звернення: 24.06.2026).
20. xterm.js. URL: <https://xtermjs.org> (дата звернення: 24.06.2026).

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				88
Змн.	Арк.	№ докум.	Підпис	Дата		

21. PostgreSQL 16 Documentation. The PostgreSQL Global Development Group. URL: <https://www.postgresql.org/docs/16/> (дата звернення: 24.06.2026).
22. GitLab CI/CD documentation. GitLab. URL: <https://docs.gitlab.com/ee/ci/> (дата звернення: 24.06.2026).
23. YamlDotNet Documentation. URL: <https://github.com/aaubry/YamlDotNet/wiki> (дата звернення: 24.06.2026).
24. Nginx Documentation. URL: <https://nginx.org/en/docs/> (дата звернення: 24.06.2026).
25. RFC 7519: JSON Web Token (JWT). Internet Engineering Task Force. URL: <https://datatracker.ietf.org/doc/html/rfc7519> (дата звернення: 24.06.2026).
26. OWASP Top Ten. OWASP Foundation. URL: <https://owasp.org/www-project-top-ten/> (дата звернення: 24.06.2026).
27. Conventional Commits 1.0.0. URL: <https://www.conventionalcommits.org/en/v1.0.0/> (дата звернення: 24.06.2026).
28. BCrypt.Net-Next. URL: <https://github.com/BcryptNet/bcrypt.net> (дата звернення: 24.06.2026).
29. Serilog Documentation. URL: <https://serilog.net> (дата звернення: 24.06.2026).
30. ДСТУ 8302:2015. Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання. Київ: ДП «УкрНДНЦ», 2016. 17 с.

		Черноштан Т.Ю			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				89
Змн.	Арк.	№ докум.	Підпис	Дата		

ДОДАТКИ

		Черноштан Т.Ю			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		90

Загальна структура бази даних та зв'язки між таблицями



Рис. А.1. ER-діаграма бази даних

Реалізація механізму захисту квот від станів гонки

```

private static readonly ConcurrentDictionary<int, SemaphoreSlim> _userLocks =
new();

private async Task ValidateQuotaCore(int userId, long? memoryLimit, double?
cpu,
    int? excludeContainerId = null)
{
    var quotaEntity = await GetOrCreateCoreAsync(userId);
    var containers = (await
        _containerRepository.GetByUserIdAsync(userId)).ToList();

    var isUpdate = excludeContainerId.HasValue;
    var others = isUpdate
        ? containers.Where(c => c.Id != excludeContainerId!.Value).ToList()
        : containers;

    if (!isUpdate && containers.Count >= quotaEntity.MaxContainers)
        throw new InvalidOperationException(
            $"Перевищено ліміт контейнерів. Максимум:
{quotaEntity.MaxContainers}");

    if (!memoryLimit.HasValue || memoryLimit.Value <= 0)
        throw new InvalidOperationException("Ліміт RAM обов'язковий");
    var currentTotalMemory = others.Sum(c => c.MemoryLimit ?? 0);
    if (currentTotalMemory + memoryLimit.Value > quotaEntity.MaxMemoryMb *
1024L * 1024L)
        throw new InvalidOperationException(
            $"Перевищено загальний ліміт пам'яті. Максимум:
{quotaEntity.MaxMemoryMb} MB");

    if (!cpu.HasValue || cpu.Value <= 0)
        throw new InvalidOperationException("Ліміт CPU обов'язковий");
    var currentTotalCpu = others.Sum(c => c.CpuLimit ?? 0);
    if (currentTotalCpu + cpu.Value > quotaEntity.MaxCpu)
        throw new InvalidOperationException(
            $"Перевищено загальний ліміт CPU. Максимум: {quotaEntity.MaxCpu}
cores");
}

public async Task<IAsyncDisposable> ValidateAndLockCreationAsync(int userId,
    long? memoryLimit, double? cpu)
{
    var semaphore = _userLocks.GetOrAdd(userId, _ => new SemaphoreSlim(1, 1));
    await semaphore.WaitAsync();
    try
    {
        await ValidateQuotaCore(userId, memoryLimit, cpu);
        return new SemaphoreRelease(semaphore);
    }
    catch
    {
        semaphore.Release();
    }
}

```

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				92
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        throw;
    }
}

private sealed class SemaphoreRelease : IAsyncDisposable
{
    private readonly SemaphoreSlim _semaphore;
    private int _released;

    public SemaphoreRelease(SemaphoreSlim semaphore) => _semaphore = semaphore;

    public ValueTask DisposeAsync()
    {
        if (Interlocked.Exchange(ref _released, 1) == 0)
            _semaphore.Release();
        return ValueTask.CompletedTask;
    }
}

```

		Черноштан Т.Ю			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		93

Реалізація механізму посилення безпеки Docker Compose

```

private static readonly HashSet<string> AllowedCapabilities =
new(StringComparer.OrdinalIgnoreCase)
{
    "CHOWN", "DAC_OVERRIDE", "FOWNER", "FSETID", "KILL", "SETGID", "SETUID",
    "SETPCAP",
    "NET_BIND_SERVICE", "NET_RAW", "SYS_CHROOT", "MKNOD", "AUDIT_WRITE",
    "SETFCAP"
};

private static readonly string[] AllowedHostPathPrefixes =
    ["/var/serverpanel/volumes/", "/srv/serverpanel/"];

private static void ValidateComposeSecurity(YamlMappingNode root)
{
    foreach (var (serviceKey, serviceNode) in GetServices(root).Children)
    {
        var name = serviceKey.ToString();
        if (serviceNode is not YamlMappingNode service) continue;

        if (GetScalar(service, "privileged")?.Equals("true",
            StringComparison.OrdinalIgnoreCase) == true)
            throw new InvalidOperationException($"Сервіс \"{name}\":
            privileged-режим заборонено");

        var networkMode = GetScalar(service, "network_mode");
        if (networkMode == "host")
            throw new InvalidOperationException($"Сервіс \"{name}\":
            network_mode: host заборонено");
        if (networkMode is not null &&
            (networkMode.StartsWith("container:",
                StringComparison.OrdinalIgnoreCase) ||
            networkMode.StartsWith("service:",
                StringComparison.OrdinalIgnoreCase)))
            throw new InvalidOperationException($"Сервіс \"{name}\":
            network_mode заборонено");

        if (GetScalar(service, "pid") == "host")
            throw new InvalidOperationException($"Сервіс \"{name}\": pid: host
            заборонено");
        if (GetScalar(service, "ipc") == "host")
            throw new InvalidOperationException($"Сервіс \"{name}\": ipc: host
            заборонено");

        if (TryGetNode(service, "devices", out _))
            throw new InvalidOperationException($"Сервіс \"{name}\": прокидання
            пристроїв заборонено");

        if (TryGetNode(service, "cap_add", out var capNode) && capNode is
            YamlSequenceNode caps)
            foreach (var cap in caps)
    }

```

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				94
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        var capName =
cap.ToString().Trim().ToUpperInvariant().Replace("CAP_", "");
        if (!AllowedCapabilities.Contains(capName))
            throw new InvalidOperationException(
                $"Сервіс \"{name}\": capability {capName} не
дозволено");
    }

    if (TryGetNode(service, "volumes", out var volsNode) && volsNode is
YamlSequenceNode vols)
        foreach (var vol in vols)
        {
            string? source = vol switch
            {
                YamlScalarNode scalar => scalar.Value?.Split(':')[0],
                YamlMappingNode mapping => GetScalar(mapping, "source"),
                _ => null
            };
            if (source is not null) ValidateVolumeSource(name, source);
        }
    }

private static void ValidateVolumeSource(string serviceName, string source)
{
    source = source.Trim();
    if (!source.StartsWith('/')) return;

    if (source.Contains("..") || source.Contains('\\0'))
        throw new InvalidOperationException(
            $"Сервіс \"{serviceName}\": шлях містить недозволені символи");

    var normalized = source.Replace('\\', '/').TrimEnd('/').ToLowerInvariant();
    var allowed = AllowedHostPathPrefixes.Select(p =>
p.TrimEnd('/').ToLowerInvariant());
    if (!allowed.Any(p => normalized == p || normalized.StartsWith(p + "/")))
        throw new InvalidOperationException(
            $"Сервіс \"{serviceName}\": монтування {source} заборонено");
}

private static string HardenComposeYaml(string yaml)
{
    var root = ParseComposeRoot(yaml);
    const string noNewPrivs = "no-new-privileges:true";

    foreach (var (_, serviceNode) in GetServices(root).Children)
    {
        if (serviceNode is not YamlMappingNode service) continue;
        if (TryGetNode(service, "security_opt", out var optNode))
        {
            if (optNode is YamlSequenceNode seq &&
!seq.Children.OfType<YamlScalarNode>()
.Any(s => (s.Value ?? "").Contains("no-new-privileges")))
                seq.Children.Add(new YamlScalarNode(noNewPrivs));
        }
        else
    }
}

```

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				95
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        {
            service.Children.Add(new YamlScalarNode("security_opt"),
                new YamlSequenceNode(new YamlScalarNode(noNewPrivs)));
        }
    }
    return SerializeYaml(root);
}

```

		Черноштан Т.Ю			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		96

Лістинги модульних тестів безпеки

```

public class StackServiceSecurityTests
{
    [Fact]
    public void ValidateComposeSecurity_Privileged_Throws()
    {
        var yaml = """
        services:
          web:
            image: nginx
            privileged: true
        """;
        var ex = Assert.Throws<InvalidOperationException>(() =>
        ValidateSecurity(yaml));
        Assert.Contains("privileged", ex.Message);
    }

    [Fact]
    public void ValidateComposeSecurity_NetworkModeHost_Throws()
    {
        var yaml = """
        services:
          web:
            image: nginx
            network_mode: host
        """;
        var ex = Assert.Throws<InvalidOperationException>(() =>
        ValidateSecurity(yaml));
        Assert.Contains("network_mode", ex.Message);
    }

    [Fact]
    public void ValidateComposeSecurity_CapAddNotInAllowlist_Throws()
    {
        var yaml = """
        services:
          web:
            image: nginx
            cap_add:
              - NET_ADMIN
        """;
        var ex = Assert.Throws<InvalidOperationException>(() =>
        ValidateSecurity(yaml));
        Assert.Contains("NET_ADMIN", ex.Message);
    }

    [Fact]
    public void ValidateComposeSecurity_VolumeMountingDockerSocket_Throws()
    {
        var yaml = """
        services:
          web:
            image: nginx

```

		Черноштан Т.Ю			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				97
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        volumes:
            - "/run/docker.sock:/var/run/docker.sock"
        """;
        var ex = Assert.Throws<InvalidOperationException>(() =>
ValidateSecurity(yaml));
        Assert.Contains("docker.sock", ex.Message);
    }

    [Fact]
    public void
HardenComposeYaml_ServiceWithoutSecurityOpt_InjectsNoNewPrivileges()
    {
        var yaml = ""
        services:
            web:
                image: nginx
        """;
        var result = Harden(yaml);
        Assert.Contains("no-new-privileges:true", result);
    }
}

public class IdorOwnershipTests
{
    private const int OwnerId = 2;    // ресурс належить користувачу #2
    private const int AttackerId = 1; // користувач #1 намагається дістатись
    private const int ResourceId = 5;

    [Fact]
    public async Task Container_WrongUser_ThrowsUnauthorized()
    {
        _containerRepoMock.Setup(r => r.GetByIdAsync(ResourceId))
            .ReturnsAsync(new Container { Id = ResourceId, UserId = OwnerId,
Status = "running" });

        await Assert.ThrowsAsync<UnauthorizedAccessException>(
            () => _containerService.StartContainerAsync(ResourceId, AttackerId,
isAdmin: false));
    }

    [Fact]
    public async Task Container_Admin_PassesOwnershipCheck()
    {
        _containerRepoMock.Setup(r => r.GetByIdAsync(ResourceId))
            .ReturnsAsync(new Container { Id = ResourceId, UserId = OwnerId,
Status = "running" });

        await AssertPassesOwnershipCheck(
            () => _containerService.StartContainerAsync(ResourceId, AttackerId,
isAdmin: true));
    }
}

```

Конфігураційні файли розгортання

```
# Dockerfile (бекенд)
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY ServerPanel.DAL/ServerPanel.DAL.csproj ServerPanel.DAL/
COPY ServerPanel.BLL/ServerPanel.BLL.csproj ServerPanel.BLL/
COPY ServerPanel.API/ServerPanel.API.csproj ServerPanel.API/
RUN dotnet restore ServerPanel.API/ServerPanel.API.csproj
COPY ServerPanel.DAL/ ServerPanel.DAL/
COPY ServerPanel.BLL/ ServerPanel.BLL/
COPY ServerPanel.API/ ServerPanel.API/
RUN dotnet publish ServerPanel.API/ServerPanel.API.csproj -c Release -o
/app/publish --no-restore

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime
WORKDIR /app
RUN addgroup --system --gid 1001 appgroup && \
    adduser --system --uid 1001 --ingroup appgroup appuser && \
    mkdir -p /app/logs /var/serverpanel/stacks && \
    chown -R appuser:appgroup /app/logs /var/serverpanel/stacks
USER appuser
COPY --from=build /app/publish .
EXPOSE 8080
ENV ASPNETCORE_URLS=http://+:8080
ENTRYPOINT ["dotnet", "ServerPanel.API.dll"]

# docker-compose.yml
services:
  postgres:
    image: postgres:16-alpine
    restart: unless-stopped
    environment:
      POSTGRES_DB: serverpanel
      POSTGRES_USER: ${POSTGRES_USER:-postgres}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:?POSTGRES_PASSWORD is required}
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER:-postgres} -d
serverpanel"]
      interval: 10s
      timeout: 5s
      retries: 5

  api:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      postgres:
        condition: service_healthy
    ports:
      - "8080:8080"
```

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				99
Змн.	Арк.	№ докум.	Підпис	Дата		

```

volumes:
  - /var/run/docker.sock:/var/run/docker.sock:ro
  - /var/serverpanel/stacks:/var/serverpanel/stacks
environment:
  ConnectionStrings__DefaultConnection: >-
    Host=postgres;Port=5432;Database=serverpanel;
    Username=${POSTGRES_USER:-postgres};Password=${POSTGRES_PASSWORD:?}
  Jwt__Key: ${JWT_KEY:?JWT_KEY is required}
  Docker__Uri: unix:///var/run/docker.sock

frontend:
  build:
    context: ./client
  depends_on:
    - api
  ports:
    - "80:80"

volumes:
  postgres_data:

# .gitlab-ci.yml
stages:
  - build
  - test
  - deploy

build-backend:
  stage: build
  script:
    - dotnet restore ServerPanel.sln
    - dotnet build ServerPanel.sln --configuration Release --no-restore
  tags: [student]
  only: [main, develop]

test-backend:
  stage: test
  script:
    - dotnet test ServerPanel.Tests --verbosity normal
  tags: [student]
  only: [main, develop]

deploy:
  stage: deploy
  script:
    - echo "Deploy stage"
  tags: [student]
  only: [main]

```

		Черноштан Т.Ю.			ІПЗ.КР.Б-121 – 26 – ПЗ	Арк.
		Морозов А.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		100