МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ

ім. Ю.М. Потебні

ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)

(рівень вищої освіти)

на тему **Розробка конструктора сайтів з використанням контейнеризації та підтримкою мультитенантної архітектури**

Виконав: студент 4 курсу, групи 6.1212-пзс

спеціальності 121 Інженерія програмного забезпечення

(код і назва спеціальності)

освітньої програми Програмне забезпечення систем

(код і назва освітньої програми)

М.В. Вруцький

(ініціали та прізвище)

Керівник асистент, д-р філос.(PhD) О.О. Жуков

(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Рецензент директор ТОВ «Дісітел»

П. О. Лютий

(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
(код та назва)
Освітня програма Програмне забезпечення систем
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри Тетяна КРИТСЬКА

“ 01 ” березня 2026 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Вруцькому Максиму В’ячеславовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Розробка конструктора сайтів з використанням контейнеризації та підтримкою мультитенантної архітектури

керівник роботи Жуков Олександр Олексійович, д-р філос. (PhD)
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджена наказом ЗНУ від 23.12.2025 р. № 1830-с

2. Строк подання студентом кваліфікаційної роботи 02.06.2026
3. Вихідні дані бакалаврської роботи
   - комплект нормативних документів ;
   - технічне завдання до роботи.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   - огляд та збір літератури стосовно теми кваліфікаційної роботи;
   - огляд та аналіз існуючих рішень та аналогів;
   - дослідження технологій контейнеризації та мультитенантних архітектур;
   - створення програмного продукту та його опис;
   - перелік вимог для роботи програми;
   - розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) слайдів презентації

6. Консультанти розділів бакалаврської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата
		Завдання прийняв

7. Дата видачі завдання 01.03.2026

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів магістерської роботи	Примітка
1.	Аналіз предметної області.	01.03-15.03	виконано
2.	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником.	09.03	виконано
3.	Аналіз програм-аналогів.	16.03-08.04	виконано
4.	Вибір технологій для реалізації.	01.04-02.04	виконано
5.	Аналіз існуючих рішень та технологій контейнеризації.	02.04-25.04	виконано
6.	Проектування архітектури системи.	02.04-05.04	виконано
7.	Побудова архітектури застосунку.	06.04-08.04	виконано
8.	Розробка модуля управління Docker-контейнерами.	08.04-16.05	в процессі
9.	Реалізація системи динамічного призначення піддоменів.	10.04-11.04	виконано
10.	Розробка користувацького інтерфейсу.	12.04-13.04	виконано
11.	Налаштування Traefik та DNS-маршрутизації.	08.04-16.05	виконано
12.	Розробка веб-інтерфейсу управління розгортаннями.	14.05-16.05	виконано
13.	Рефакторинг коду.	16.05-22.05	виконано
14.	Системне тестування мультитенантної ізоляції та маршрутизації	23.05-28.05	виконано
16.	Оформлення презентації і отримання рецензії.	12.05-31.05	виконано

Студент _____ _____М.В. Вруцький

Керівник роботи _____ _____О. О. Жуков

Нормоконтроль пройдено

Нормоконтролер _____ _____І.А. Скрипник

АНОТАЦІЯ

Сторінок — 61.

Рисунків — 12.

Джерел — 24.

Вруцький М.В. Розробка конструктора сайтів з використанням контейнеризації та підтримкою мультитенантної архітектури : кваліфікаційна робота бакалавра спеціальності 121 Інженерія програмного забезпечення / наук. керівник О.О. Жуков. Запоріжжя : ЗНУ, 2026, 61 с.

Сучасний ринок хостингу та розгортання веб-застосунків стрімко розвивається: зростає попит на платформи типу PaaS, що надають можливість швидко публікувати сайти без глибоких знань серверної інфраструктури. Існуючі рішення або є надто складними для налаштування, або не підтримують ізоляцію між різними клієнтами (тенантами), або не забезпечують автоматизованого призначення доменних імен.

Метою роботи є розробка програмно-інфраструктурного рішення — конструктора сайтів, що поєднує три компоненти: PaaS-платформу для автоматизованого розгортання довільних Docker-контейнерів з динамічним призначенням HTTPS-піддоменів, вбудований візуальний конструктор SPA-сайтів на основі бібліотеки GrapesJS та інтеграцію з OpenAI API для AI-генерації HTML/CSS-контенту за текстовим описом. Система реалізує мультитенантну ізоляцію через окремі Docker-контейнери з лімітами ресурсів та надає механізм відстеження статусу розгортань у реальному часі через WebSocket..

Ключові слова: *Docker, контейнеризація, мультитенантність, DNS, піддомен, конструктор сайтів, PaaS, автоматизоване розгортання, Traefik, веб-інфраструктура, GrapesJS, drag-and-drop редактор, WebSocket, OpenAI, AI-генерація, FastAPI, CQRS, GrapesJS, drag-and-drop редактор, WebSocket, OpenAI, AI-генерація, FastAPI, CQRS*

ABSTRACT

Pages — 61.

Drawings — 12.

Sources — 24.

Vrutskyi M.V. Development of a website constructor using containerization and multi-tenant architecture support: qualification thesis of the bachelor of specialty 121 Software Engineering / Science Manager O.O. Zhukov Zaporizhzhia: ZNU, 2026, 61 p.

The modern web hosting and deployment market is rapidly evolving, with growing demand for PaaS platforms that enable fast website publication without deep knowledge of server infrastructure. Existing solutions are either too complex to configure, do not support isolation between different clients (tenants), or do not provide automated domain name assignment.

The aim of this work is to develop a software-infrastructure solution — a website constructor that combines three components: a PaaS platform for automated deployment of arbitrary Docker containers with dynamic HTTPS subdomain assignment, a built-in visual SPA website builder based on the GrapesJS library, and integration with the OpenAI API for AI-driven HTML/CSS content generation from plain-text descriptions. The system implements multi-tenant isolation through separate Docker containers with resource limits and provides real-time deployment status tracking via WebSocket.

Keywords: *Docker, containerization, multi-tenancy, DNS, subdomain, website constructor, PaaS, automated deployment, Traefik, web infrastructure, GrapesJS, drag-and-drop editor, WebSocket, OpenAI, AI generation, FastAPI, CQRS.*

ЗМІСТ

ВСТУП 8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ТЕХНОЛОГІЙ КОНТЕЙНЕРИЗАЦІЇ 14
1.1 Сучасний стан ринку хостингу та автоматизованого розгортання 14
1.2 Технологія контейнеризації та Docker 15
1.3 Мультитенантні архітектури 16
1.4 Динамічна DNS-маршрутизація та зворотні проксі 17
1.4.1 Принципи DNS та піддомени 17
1.4.2 Зворотні проксі-сервери 18
1.4.3 Traefik як динамічний маршрутизатор 18
1.4.4 Wildcard SSL-сертифікати 19
1.5 Drag-and-drop веб-редактори та бібліотека GrapesJS 19
1.6 Протокол WebSocket та комунікація в реальному часі 20
1.7 Штучний інтелект у генерації веб-контенту та OpenAI API 21
1.8 Висновки з розділу 1 22
2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ВИБІР ТЕХНОЛОГІЙ 23
2.1 Огляд існуючих платформ розгортання 23
2.2 Порівняльний аналіз технологій контейнеризації 25
2.3 Особливості FastAPI та асинхронного програмування в Python 26
2.4 Особливості роботи з Docker SDK для Python та Traefik v3 28
2.5 Вибір технологічного стеку 29
2.6 Висновки з розділу 2 31
3 РОЗРОБКА КОНСТРУКТОРА САЙТІВ З ПІДТРИМКОЮ МУЛЬТИТЕНАНТНОСТІ 32
3.1 Постановка задачі 32
3.2 Вимоги до програмної системи 33
3.3 Архітектура програмної системи 34
3.4 Реалізація серверної частини 36
3.4.1 Доменні моделі та бази даних 36

3.4.2 Репозиторії та Unit of Work 38
3.4.3 Команди та запити (CQRS) 39
3.4.4 Автентифікація та безпека 39
3.5 Реалізація конструктора SPA-сайтів 40
3.5.1 Редактор GrapesJS та збереження контенту 40
3.5.2 Публікація сайту — складання Docker-образу 43
3.5.3 WebSocket — відстеження статусу розгортання в реальному часі 45
3.5.4 AI-генерація контенту через OpenAI API 47
3.6 Реалізація модуля управління Docker-контейнерами 48
3.7 Налаштування маршрутизації та DNS 51
3.8 Розробка веб-інтерфейсу 52
3.9 Тестування системи 54
3.10 Розгортання системи на хмарній платформі DigitalOcean 56
3.10.1 Підготовка Droplet та встановлення Docker 57
3.10.2 Налаштування DNS та wildcard-сертифікатів 57
3.11 Висновки з розділу 3 58
ВИСНОВКИ 59
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 60

ВСТУП

Актуальність теми

Стрімкий розвиток хмарних технологій та контейнеризації кардинально змінює підходи до розгортання та управління веб-застосунками. Платформи як сервіс (PaaS) дозволяють розробникам і не технічним користувачам публікувати веб-сайти та додатки без необхідності вручну налаштовувати серверну інфраструктуру. Завдяки цьому попит на зручні інструменти автоматизованого розгортання постійно зростає.

Однак існуючі рішення мають суттєві обмеження. Комерційні платформи типу Heroku або Render пропонують готові середовища, але обмежують вибір технологій і коштують значних коштів при масштабуванні. Самостійне налаштування Kubernetes або Docker Swarm вимагає глибоких знань DevOps. При цьому жодне з популярних рішень не забезпечує одночасно: повну ізоляцію між різними клієнтами (мультитенантність), автоматичне призначення піддоменів та підтримку довільного програмного забезпечення в контейнерах. Окремою проблемою залишається доступність публікації веб-сайтів для нетехнічної аудиторії. Інструменти на зразок Wix або Squarespace надають зручні конструктори, але не забезпечують ізоляції та обмежують функціональність. Поява великих мовних моделей (LLM) відкрила нові можливості для автоматизації: AI здатний генерувати якісний HTML/CSS-контент за текстовим описом, що суттєво знижує поріг входження для створення веб-сайтів.

Концепція мультитенантності передбачає, що єдина інфраструктура обслуговує кількох незалежних клієнтів (тенантів), забезпечуючи при цьому ізоляцію їхніх даних і ресурсів. У контексті конструктора сайтів це означає, що кожен клієнт отримує власний Docker-контейнер з відповідним піддоменом, при цьому всі контейнери працюють на спільній фізичній інфраструктурі без взаємного впливу.

Таким чином, актуальним є розробка програмно-інфраструктурного рішення, яке автоматизує процес розгортання контейнеризованих веб-застосунків з динамічним призначенням DNS-імен та підтримкою мультитенантної архітектури. Таке рішення дозволить суттєво спростити публікацію сайтів для широкого кола користувачів.

Мета дослідження

Розробка програмно-інфраструктурного рішення — конструктора сайтів, що поєднує: PaaS-платформу з автоматизованим розгортанням Docker-контейнерів та динамічним призначенням HTTPS-піддоменів; вбудований drag-and-drop конструктор SPA-сайтів з публікацією через Docker build; WebSocket-відстеження статусу розгортань у реальному часі; та AI-генерацію HTML/CSS-контенту на основі OpenAI API — з підтримкою мультитенантної ізоляції для всіх компонентів.

Завдання дослідження

Дослідити технології контейнеризації, DNS-маршрутизації, drag-and-drop веб-редакторів та AI-інструментів генерації контенту. Проаналізувати підходи до побудови мультитенантних систем та обґрунтувати вибір технологічного стеку. Розробити архітектуру системи на основі CQRS, Repository та Unit of Work. Реалізувати асинхронний бекенд на Python 3.12 та FastAPI. Реалізувати модуль управління Docker-контейнерами через Docker Engine API з підтримкою ресурсних лімітів та побудовою образів. Інтегрувати Traefik v3 та DuckDNS для автоматичної HTTPS-маршрутизації. Розробити вбудований конструктор SPA-сайтів на базі GrapesJS з публікацією через Docker. Реалізувати WebSocket-механізм сповіщень про статус розгортань у реальному часі. Інтегрувати OpenAI API для AI-генерації HTML/CSS-контенту в редакторі. Розробити React-фронтенд та провести функціональне тестування системи.

Об’єкт дослідження

Об’єктом дослідження є процес автоматизованого розгортання та публікації веб-застосунків і веб-сайтів у контейнеризованому середовищі з підтримкою мультитенантності, візуального конструювання та AI-генерації контенту.

Предмет дослідження

Предметом дослідження є методи та інструменти контейнеризації, динамічної маршрутизації, drag-and-drop веб-редагування (GrapesJS), WebSocket-комунікації та AI-генерації контенту (OpenAI API) для побудови мультитенантного конструктора сайтів з PaaS-функціональністю.

Методи дослідження

Аналіз наукової та технічної документації з технологій контейнеризації та оркестрації. Порівняльний аналіз існуючих рішень для автоматизованого розгортання веб-застосунків. Проектування архітектури програмної системи методами структурного та компонентного аналізу. Практична реалізація та тестування системи управління Docker-контейнерами з динамічною DNS-маршрутизацією.

Практичне значення одержаних результатів

Практичне значення одержаних результатів полягає у створенні готового до використання програмно-інфраструктурного рішення з трьома режимами роботи: PaaS-розгортання довільних Docker-образів, візуальний конструктор SPA-сайтів на GrapesJS та AI-асистований режим генерації контенту. Система орієнтована на малий і середній бізнес без штатних програмістів, розробників-одинаків, що хочуть швидко розгорнути прототип, та на будь-яких користувачів, що мають Docker-образ і потребують його швидкої публікації з персональним доменом і HTTPS без DevOps-знань.

Глосарій

Контейнеризація — технологія ізоляції застосунків та їх залежностей у легковагові, переносні середовища виконання, що спільно використовують ядро операційної системи хоста.

Docker — відкрита платформа для розробки, доставки та запуску застосунків у контейнерах, що забезпечує стандартизоване середовище виконання.

Docker-образ (image) — незмінний шаблон, що містить файлову систему, конфігурацію та залежності, необхідні для запуску контейнера.

Мультитенантність — архітектурний підхід, при якому єдина інфраструктура обслуговує кількох незалежних клієнтів (тенантів) з ізоляцією їхніх даних та ресурсів.

DNS (Domain Name System) — ієрархічна розподілена система іменування, що перетворює доменні імена на IP-адреси та навпаки.

DigitalOcean — хмарна платформа (IaaS) для розгортання серверних застосунків. Надає віртуальні машини (Droplets), керований PostgreSQL (Managed Databases), реєстр контейнерних образів (Container Registry) та підтримку DNS API для автоматизації wildcard-сертифікатів через Let’s Encrypt.

Піддомен (subdomain) — частина доменного імені, що розташована ліворуч від основного домену та відокремлена крапкою, наприклад mysite.example.com.

PaaS (Platform as a Service) — модель хмарних обчислень, що надає платформу та середовище для розробки, запуску та управління застосунками без необхідності управляти базовою інфраструктурою.

Зворотний проксі (reverse proxy) — сервер, що приймає запити від клієнтів та перенаправляє їх до відповідних серверів або контейнерів на основі правил маршрутизації.

Traefik — сучасний HTTP-проксі та балансувальник навантаження, що автоматично виявляє конфігурацію контейнерів і налаштовує маршрутизацію.

API (Application Programming Interface) — інтерфейс прикладного програмування, що забезпечує взаємодію між різними програмами чи компонентами.

Docker Engine API — REST API для програмного управління Docker-демоном, що дозволяє створювати, запускати, зупиняти та видаляти контейнери з коду застосунку.

Тенант (tenant) — незалежний клієнт або організація у мультитенантній системі, що має власні ізольовані ресурси та конфігурацію.

Wildcard DNS — DNS-запис виду *.example.com, що відповідає будь-якому піддомену основного домену, дозволяючи динамічно призначати нові піддомени без зміни DNS-конфігурації.

Docker Compose — інструмент для визначення та запуску багатоконтейнерних Docker-застосунків за допомогою файлу конфігурації у форматі YAML.

Namespace (простір імен) — механізм ізоляції ядра Linux, що використовується Docker для розмежування процесів, мереж і файлових систем між контейнерами.

Маршрутизація (routing) — процес визначення шляху передачі мережевого трафіку від клієнта до відповідного сервісу або контейнера на основі адреси, хосту або інших параметрів запиту.

HTTPS/TLS — захищений протокол передачі даних, що використовує шифрування TLS/SSL для захисту трафіку між клієнтом та сервером.

GrapesJS — відкрита JavaScript-бібліотека для побудови drag-and-drop веб-редакторів. Надає компонентну модель сторінки, панель стилів та зберігає стан у форматі JSON (projectData).

SPA (Single Page Application) — веб-застосунок, що завантажується як єдиний HTML-документ та динамічно оновлює вміст без перезавантаження сторінки через JavaScript-маршрутизацію.

WebSocket — протокол двостороннього зв’язку між клієнтом і сервером поверх одного TCP-з’єднання (RFC 6455). Сервер може надсилати події клієнту без HTTP-запиту — основа real-time сповіщень.

OpenAI API — програмний інтерфейс для взаємодії з мовними моделями OpenAI. Endpoint Chat Completions з JSON mode використовується для генерації HTML/CSS за текстовим описом.

LLM (Large Language Model) — велика мовна модель — нейронна мережа, що генерує текст, код та JSON на основі інструкцій. gpt-4o-mini є моделлю OpenAI, що використовується для AI-генерації в системі.

JSONB — тип даних PostgreSQL для зберігання JSON у двійковому форматі з підтримкою індексування. Використовується для збереження projectData стану GrapesJS-редактора.

Drag-and-drop — інтерфейсна техніка переміщення елементів мишкою або дотиком. Дозволяє конструювати HTML-сторінки без написання коду.

asyncio — стандартна бібліотека Python для асинхронного програмування з async/await. Основа для FastAPI, async SQLAlchemy та WebSocket-обробників у системі.

Pydantic — Python-бібліотека валідації даних за анотаціями типів. Використовується у FastAPI для перевірки тіл запитів та автогенерації OpenAPI-схем.

cgroups (control groups) — механізм ядра Linux для обмеження ресурсів CPU та RAM для груп процесів. Docker використовує cgroups для mem_limit та nano_cpus контейнерів.

Polling — техніка регулярного надсилання HTTP-запитів для отримання оновлень стану. Менш ефективна за WebSocket через надлишкові запити та затримку відображення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ТЕХНОЛОГІЙ КОНТЕЙНЕРИЗАЦІЇ

1.1 Сучасний стан ринку хостингу та автоматизованого розгортання

Останнє десятиліття характеризується стрімким зростанням хмарних технологій та переходом від традиційних підходів до розгортання застосунків до моделей PaaS (Platform as a Service) і IaaS (Infrastructure as a Service). За даними аналітичних агентств, ринок хмарних послуг щорічно зростає більш ніж на 20%, а контейнеризація стає стандартом де-факто для розгортання сучасних веб-застосунків.

Поняття «конструктор сайтів» традиційно асоціюється з візуальними редакторами типу WordPress, Wix або Squarespace. Однак зі зростанням складності веб-застосунків виникла потреба в інфраструктурних конструкторах — платформах, які надають не лише шаблони, а повноцінне ізольоване середовище виконання для кожного клієнта.

Ключовою проблемою в цьому контексті є забезпечення одночасно трьох властивостей: ізоляція між тенантами (клієнтами), автоматизація розгортання та інтеграція з системою доменних імен. Перші дві властивості вирішуються технологіями контейнеризації, третя — механізмами динамічної DNS-маршрутизації. Але існуючі рішення рідко поєднують усі три компоненти в єдиній, простій у використанні системі.

Конкурентне середовище в сегменті PaaS-платформ представлено кількома рівнями. На верхньому рівні знаходяться великі хмарні провайдери (AWS Elastic Beanstalk, Google App Engine, Microsoft Azure App Service), що пропонують потужні інструменти, але вимагають значних фінансових вкладень та глибоких знань. Середній рівень представляють Heroku, Render, Railway та Fly.io — зручні платформи з обмеженнями у виборі технологій і вартісними бар’єрами при масштабуванні.

Самостійне розгортання на основі Kubernetes або Docker Swarm дає максимальну гнучкість, але потребує значних DevOps-компетенцій. Водночас зростає окремий ринок no-code та low-code інструментів для побудови сайтів: Wix, Squarespace, Webflow надають візуальні редактори, але не забезпечують ізоляції між клієнтами та не дозволяють розгортати довільний серверний код. Появу великих мовних моделей (LLM) відкрила ще один вимір: AI-асистовані інструменти здатні генерувати HTML/CSS-код за текстовим описом, знижуючи поріг входження у веб-розробку до нуля. Таким чином, існує незаповнена ніша для рішення, яке поєднує простоту PaaS-платформ із гнучкістю контейнерного підходу, вбудованим конструктором сайтів та AI-асистованою генерацією контенту.

1.2 Технологія контейнеризації та Docker

Контейнеризація — це метод ізоляції застосунків та їх залежностей у легковагові, переносні середовища виконання. На відміну від традиційної віртуалізації, де кожна віртуальна машина містить повноцінну операційну систему, контейнери спільно використовують ядро операційної системи хоста, що дозволяє суттєво знизити накладні витрати ресурсів.

Docker — найпоширеніша на сьогодні платформа контейнеризації, випущена компанією Docker Inc. у 2013 році. Вона заснована на механізмах ізоляції ядра Linux: namespace для розмежування процесів, мереж і файлових систем, та cgroups для обмеження споживання ресурсів. Docker стандартизував формат контейнерних образів (OCI — Open Container Initiative), зробивши контейнери переносними між різними середовищами.

Архітектура Docker складається з кількох ключових компонентів. Docker Engine — це демон, що керує контейнерами на хост-машині. Docker Engine API — REST API, що дозволяє програмно взаємодіяти з демоном: створювати, запускати, зупиняти та видаляти контейнери без використання CLI. Docker Hub та інші реєстри образів — репозиторії для зберігання та

розповсюдження Docker-образів. Docker Compose — інструмент для визначення та запуску багатоконтейнерних застосунків.

Ключовою перевагою Docker для побудови мультитенантних систем є ізоляція на рівні мережі: кожен контейнер може мати власний мережевий простір (namespace), що унеможливлює прямий доступ між контейнерами різних тенантів без явного дозволу. При цьому всі контейнери можуть бути підключені до спільної docker-мережі для взаємодії з маршрутизатором.

Docker Engine API надає повний контроль над lifecycle контейнерів через HTTP-запити до Unix-сокета або TCP-порту. Основні операції включають: завантаження образів (pull), створення контейнерів (create) з параметрами мережі та змінних середовища, запуск (start), зупинку (stop), видалення (remove) та отримання метаданих (inspect). Ці можливості є фундаментом для програмованого управління інфраструктурою в розроблюваній системі.

1.3 Мультитенантні архітектури

Мультитенантність (multi-tenancy) — архітектурний принцип, при якому єдина програмна система обслуговує кількох незалежних клієнтів (тенантів), забезпечуючи при цьому ізоляцію їхніх даних, конфігурацій та ресурсів. Цей підхід протиставляється однотенантній (single-tenant) архітектурі, де кожен клієнт отримує власну окрему інсталяцію системи.

Розрізняють кілька рівнів мультитенантності залежно від ступеня ізоляції. Спільна база даних, спільна схема — найменш ізольований підхід, де всі тенанти зберігають дані в одній таблиці з полем ідентифікатора тенанта. Спільна база даних, окремі схеми — кожен тенант має власний простір імен у базі даних. Окремі бази даних — максимальна ізоляція даних, але більші витрати ресурсів. Окремі контейнери або процеси — повна ізоляція на рівні виконання, що забезпечується контейнеризацією.

Для конструктора сайтів найбільш доцільним є підхід окремих контейнерів. Кожен тенант отримує власний Docker-контейнер, що може запускати довільне програмне забезпечення. Це забезпечує повну ізоляцію не лише даних, але й самого застосунку та його залежностей. При цьому зберігається можливість ефективного використання ресурсів хоста за рахунок спільного ядра ОС.

Ключовими аспектами реалізації мультитенантності є управління ресурсами (CPU та RAM ліміти через cgroups), ізоляція мережі (окремі network namespace), ізоляція файлової системи (через union file system Docker), а також система ідентифікації та автентифікації тенантів. Останнє включає призначення унікальних ідентифікаторів, що використовуються як для управління контейнерами, так і для формування піддоменів.

1.4 Динамічна DNS-маршрутизація та зворотні проксі

1.4.1 Принципи DNS та піддомени

DNS (Domain Name System) — ієрархічна розподілена система, що виконує трансляцію доменних імен у IP-адреси. Піддомен — частина доменного імені ліворуч від основного домену, наприклад client1.constructor.example.com є піддоменом constructor.example.com. Для конструктора сайтів кожен тенант отримує унікальний піддомен, пов’язаний з його контейнером.

Wildcard DNS-запис виду *.example.com відповідає будь-якому піддомену домену example.com. Використання такого запису дозволяє динамічно створювати нові піддомени для нових тенантів без необхідності щоразу редагувати DNS-конфігурацію. Запит на tenant1.example.com та tenant2.example.com буде перенаправлено на одну й ту саму IP-адресу сервера, а розподіл між конкретними контейнерами відбувається на рівні зворотного проксі.

1.4.2 Зворотні проксі-сервери

Зворотний проксі (reverse proxy) — сервер, що приймає вхідні HTTP(S) запити і перенаправляє їх до відповідних серверів або контейнерів на основі правил маршрутизації. На відміну від прямого проксі, зворотний проксі діє від імені серверів, а не клієнтів. Він є центральним компонентом маршрутизації в мультитенантній системі.

Традиційним рішенням є nginx та Apache HTTP Server, де правила маршрутизації задаються статично у конфігураційних файлах. Для мультитенантного конструктора сайтів, де контейнери динамічно створюються та видаляються, статична конфігурація є неприйнятною — вона потребує перезапуску проксі при кожній зміні, що порушує доступність системи.

1.4.3 Traefik як динамічний маршрутизатор

Traefik — сучасний HTTP-проксі та балансувальник навантаження, розроблений компанією Traefik Labs і написаний мовою Go. Ключова особливість Traefik — автоматичне виявлення (service discovery) конфігурації з різних джерел: Docker labels, Kubernetes annotations, файли конфігурації, тощо. При появі нового контейнера з відповідними мітками Traefik автоматично оновлює свою конфігурацію без перезапуску.

Інтеграція Traefik з Docker реалізується через прослуховування Docker socket. Коли контейнер запускається з мітками виду traefik.http.routers.myapp.rule=Host(`myapp.example.com`), Traefik автоматично створює правило маршрутизації для цього хоста. Це є ключовим механізмом для реалізації динамічного призначення піддоменів у конструкторі сайтів.

Traefik також підтримує автоматичне отримання та поновлення SSL-сертифікатів через Let’s Encrypt (протокол ACME), включаючи wildcard-сертифікати через DNS-01 challenge. Це дозволяє забезпечити HTTPS для всіх піддоменів тенантів без ручного керування сертифікатами.

1.4.4 Wildcard SSL-сертифікати

Wildcard SSL-сертифікат — це цифровий сертифікат, що захищає основний домен та всі його піддомени першого рівня. Наприклад, сертифікат для *.example.com буде дійсним для tenant1.example.com, tenant2.example.com та будь-якого іншого піддомену. Це усуває необхідність отримувати окремий сертифікат для кожного нового тенанта.

Отримання wildcard-сертифікатів через Let’s Encrypt вимагає проходження DNS-01 challenge — автоматичного додавання TXT-запису до DNS зони домену для підтвердження права власності. Traefik підтримує багато DNS-провайдерів для цього процесу (Cloudflare, DigitalOcean, AWS Route 53 та інші), що дозволяє повністю автоматизувати управління сертифікатами.

1.5 Drag-and-drop веб-редактори та бібліотека GrapesJS

Drag-and-drop веб-редактори — це клієнтські застосунки, що дозволяють конструювати HTML-документи без написання коду. Вони надають бібліотеку готових блоків (текст, зображення, колонки, кнопки), які користувач перетягує на полотно редактора. Під капотом кожен блок відповідає певній HTML-структурі, а редактор відстежує дерево компонентів і дозволяє налаштовувати стилі через графічний інтерфейс.

GrapesJS — відкрита JavaScript-бібліотека (MIT-ліцензія), розроблена спеціально для побудови таких редакторів. Ключовою особливістю є компонентна модель: кожен елемент сторінки є об’єктом з власними властивостями, дочірніми компонентами та CSS-правилами. Внутрішній стан проєкту GrapesJS серіалізується у формат JSON (так зване projectData), що дозволяє зберегти його в базі даних і повністю відновити стан редактора при наступному відкритті. Додатково бібліотека може рендерувати фінальні HTML та CSS для використання поза редактором — наприклад, для компіляції в Docker-образ при публікації сайту.

Архітектура GrapesJS складається з кількох підсистем: Components Manager (управління деревом компонентів), Styles Manager (CSS-правила), Asset Manager (медіафайли), Panels та Commands (інструменти UI). Бібліотека надає JavaScript API для програматичного управління: editor.setComponents(html) завантажує HTML у редактор, editor.getHtml() та editor.getCss() повертають відрендерований вивід, а editor.loadProjectData(json) відновлює збережений стан. Ці методи використовуються в системі для інтеграції з AI-генерацією та збереженням у базі даних.

1.6 Протокол WebSocket та комунікація в реальному часі

WebSocket (RFC 6455) — протокол для організації двостороннього повнодуплексного зв'язку між клієнтом і сервером поверх одного TCP-з'єднання. На початку клієнт надсилає HTTP-запит з заголовком Upgrade: websocket — це так зване рукостискання (handshake). Після підтвердження сервером HTTP-з'єднання оновлюється до WebSocket-протоколу і залишається відкритим. Обидві сторони можуть в будь-який момент надсилати повідомлення одна одній без додаткових HTTP-запитів.

Порівняно з технікою polling (регулярне відправлення HTTP-запитів для отримання оновлень), WebSocket має кілька переваг. По-перше, усуваються надлишкові запити: замість N HTTP-запитів за N секунд — одне відкрите з'єднання. По-друге, знижується затримка: повідомлення доставляється клієнту відразу після виникнення події на сервері. По-третє, зменшується навантаження на сервер через відсутність overhead від HTTP-заголовків. Це особливо важливо для сповіщень про зміну статусу розгортання: процес може тривати кілька секунд і включати кілька проміжних станів (QUEUED, BUILDING, DEPLOYING, RUNNING), кожен з яких потрібно показати користувачу без затримки.

FastAPI підтримує WebSocket через ASGI-протокол. Обробник WebSocket — це асинхронна функція, що приймає об'єкт WebSocket, викликає

ws.accept() для підтвердження з'єднання і потім знаходиться в циклі очікування повідомлень. Для розсилки повідомлень кільком клієнтам одночасно використовується клас ConnectionManager, що зберігає множину з'єднань і викликає ws.send_json() для кожного з них.

1.7 Штучний інтелект у генерації веб-контенту та OpenAI API

Великі мовні моделі (LLM, Large Language Models) — це нейронні мережі, навчені на великих масивах текстових та кодових даних. Вони демонструють здатність генерувати зв'язний текст, HTML-код, CSS-стилі та JSON-структури на основі текстових інструкцій (промптів). Ця властивість дозволяє використовувати LLM як інструмент для автоматичного створення веб-контенту за описом від користувача.

OpenAI API надає REST-інтерфейс для взаємодії з моделями GPT через Chat Completions endpoint. Запит складається з масиву повідомлень: системного промпту (role: system), що встановлює поведінку моделі, та повідомлення користувача (role: user) з конкретним завданням. Параметр response_format: {type: "json_object"} активує JSON mode — режим, в якому модель гарантовано повертає валідний JSON. Це усуває необхідність парсингу вільного тексту і виключає можливість повернення некоректного формату відповіді. У системі модель gpt-4o-mini використовується для генерації пар {html, css} за текстовим описом секції сайту.

Якість генерованого контенту значною мірою залежить від системного промпту. У розроблюваній системі промпт вимагає від моделі дотримання конкретних правил: використовувати семантичний HTML5, не включати зовнішніх залежностей (Bootstrap, Tailwind, зображення), створювати адаптивний дизайн через Flexbox та CSS Grid, і генерувати контент рівня преміум-шаблонів (Stripe, Linear, Vercel). Такий деталізований промпт забезпечує стабільно якісний результат для різних типів секцій: hero-блок, прайсинг, переваги, форма контакту тощо.

1.8 Висновки з розділу 1

У першому розділі проведено аналіз предметної області розроблюваної системи. Досліджено сучасний стан ринку хостингу та автоматизованого розгортання веб-застосунків, виявлено незаповнену нішу для рішення, що поєднує простоту PaaS-платформ із гнучкістю контейнерного підходу. Розглянуто технологію контейнеризації Docker, її архітектуру та можливості Docker Engine API для програмного управління контейнерами. Проаналізовано підходи до реалізації мультитенантних архітектур та обґрунтовано вибір підходу на основі окремих Docker-контейнерів для кожного тенанта. Вивчено механізми динамічної DNS-маршрутизації, зворотних проксі-серверів та роль Traefik як інструменту автоматичного виявлення конфігурації контейнерів. Розглянуто підхід до автоматизації управління SSL-сертифікатами з використанням wildcard-сертифікатів та Let’s Encrypt.

2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ВИБІР ТЕХНОЛОГІЙ

2.1 Огляд існуючих платформ розгортання

Аналіз конкурентного середовища є важливим етапом проектування будь-якої системи. Для розуміння вимог до розроблюваного конструктора сайтів було проаналізовано основні категорії існуючих рішень: хмарні PaaS-платформи, self-hosted рішення та спеціалізовані конструктори сайтів.

Heroku — одна з піонерських PaaS-платформ, що вивела на ринок концепцію розгортання застосунків через репозиторій Git без ручного налаштування серверів. Платформа підтримує численні мови програмування через систему buildpack та надає велику екосистему сторонніх надбудов (add-ons): бази даних, черги повідомлень, моніторинг тощо. Heroku автоматично масштабує кількість dynos (одиниць виконання) залежно від навантаження і забезпечує HTTPS для всіх застосунків. Однак для завдань розроблюваної системи Heroku має принципові обмеження: платформа не підтримує розгортання довільних Docker-образів (лише офіційні buildpack), відсутня нативна мультитенантна ізоляція на рівні контейнерів, а після скасування безкоштовного тарифу у 2022 році вартість масштабування значно зросла. Відсутність зовнішнього API для програмного управління контейнерами унеможливлює інтеграцію з оркестратором власної розробки.

Render — сучасна PaaS-платформа, що підтримує розгортання довільних Docker-образів, статичних сайтів, баз даних і фонових задач. Платформа надає автоматичне TLS, глобальний CDN, автоматичне масштабування та підтримку приватних мереж між сервісами. Для задач розроблюваної системи Render має суттєві обмеження: ізоляція між кінцевими клієнтами (тенантами) реалізується на рівні платформи, а не на рівні окремих Docker-контейнерів — кожен тенант платить окремо і не отримує власного ізольованого середовища виконання. Налаштування кастомних піддоменів вимагає ручних дій в адміністративній панелі, що унеможливлює автоматичне

призначення піддоменів через API. Відсутність зовнішнього API для програмного управління розгортаннями є критичним обмеженням для сценарію PaaS-конструктора.

Coolify — open-source self-hosted альтернатива Heroku та Netlify, яка дозволяє розгортати власну PaaS-платформу на будь-якому VPS або bare-metal сервері. Платформа підтримує розгортання через Docker, Docker Compose і Nixpacks, має вбудовану інтеграцію з Traefik для автоматичної маршрутизації та управління TLS-сертифікатами, відстежує статус усіх сервісів і підтримує автоматичне резервне копіювання. Для задач розроблюваної системи Coolify має обмеження: платформа орієнтована на технічних розробників, що самостійно налаштовують власну інфраструктуру. Coolify не надає публічного API для програмного управління контейнерами зовнішніми системами, а реалізація мультитенантності з ізоляцією прав доступу між незалежними клієнтами вимагає нетривіального налаштування. Монолітна архітектура Coolify з власною базою даних ускладнює інтеграцію з зовнішнім оркестратором.

Caprover — open-source PaaS-платформа на основі Docker Swarm. Надає веб-інтерфейс та CLI для управління застосунками, підтримує автоматичне TLS через Let’s Encrypt. Проте Caprover не має вбудованої мультитенантної ізоляції та не надає зовнішнього API для інтеграції з іншими системами.

Railway — відносно новий хмарний PaaS, орієнтований на простоту розгортання через Docker-образи або автодетекцію типу проєкту. Railway надає REST API для управління проєктами та змінними середовища. Проте платформа є однотенантною з точки зору кінцевого користувача: вона не надає можливості керувати ізольованими контейнерами від імені різних кінцевих клієнтів. Призначення піддоменів автоматизоване лише для власних сервісів акаунту, але не для незалежних замовників. Проведений аналіз показує, що жодне з розглянутих рішень не забезпечує необхідної комбінації можливостей: довільні Docker-образи, зовнішнє програмне API управління контейнерами, автоматичне призначення піддоменів для різних тенантів та

повна ізоляція між клієнтами. Це підтверджує доцільність розробки власного рішення на основі Docker Engine API.

2.2 Порівняльний аналіз технологій контейнеризації

Для реалізації конструктора сайтів було розглянуто кілька підходів до ізоляції середовищ виконання: Docker-контейнери, LXC/LXD-контейнери та віртуальні машини.

LXC (Linux Containers) — більш низькорівнева технологія контейнеризації, що надає більший контроль над конфігурацією. LXD розширює LXC зручним API та підтримкою образів. Проте екосистема LXC значно менша за Docker: реєстр образів обмежений, а інтеграція з інструментами оркестрації (зокрема Traefik) складніша. Docker має значно більшу спільноту, обширний Hub-реєстр образів і вбудовану підтримку compose-конфігурацій.

Podman — альтернатива Docker від Red Hat, що не потребує привілейованого демона (rootless). Podman сумісний з Docker CLI та образами. Для серверних систем, де потрібне централізоване управління, перевага Podman менш очевидна, оскільки Docker забезпечує стабільніше та документованіше REST API для програмної взаємодії.

Віртуальні машини (KVM, VMware) забезпечують максимальну ізоляцію, але мають значно більші накладні витрати по часу запуску (десятки секунд проти мілісекунд для Docker) та споживанню ресурсів. Для конструктора сайтів, де потрібно швидко створювати середовища для нових тенантів, ВМ є неоптимальним вибором.

В результаті аналізу Docker обрано як основну технологію контейнеризації завдяки: розвиненому REST API (Docker Engine API), широкій екосистемі готових образів, нативній інтеграції з Traefik, підтримці docker-compose для визначення складних конфігурацій та великій спільноті. Важливо розуміти, що ізоляція Docker-контейнерів базується на двох

механізмах ядра Linux. Перший — namespace (простори імен): для кожного контейнера ядро створює окремі простори PID (процеси), NET (мережевий стек), MNT (файлова система), UTS (ім’я хосту) та IPC (міжпроцесна комунікація). Завдяки цьому контейнер «бачить» лише власні процеси та мережу, не маючи доступу до ресурсів інших контейнерів. Другий механізм — cgroups (control groups): дозволяє встановлювати жорсткі ліміти на споживання CPU та RAM для кожного контейнера, що є критичним для мультитенантного середовища, де ресурси сервера ділять кілька незалежних клієнтів. Docker Engine API — це REST API, що прослуховується на Unix-сокеті /var/run/docker.sock. Він надає повний контроль над lifecycle контейнерів: завантаження образів, створення контейнерів з передачею параметрів мережі, змінних середовища, ресурсних лімітів та Docker-міток, запуск, зупинку та видалення. Офіційний Docker SDK для Python є об’єктно-орієнтованою обгорткою над HTTP-викликами до цього сокету.

2.3 Особливості FastAPI та асинхронного програмування в Python

Вибір мови програмування та фреймворку для бекенд-сервісу безпосередньо визначає архітектуру всієї системи. Python 3.12 у поєднанні з FastAPI надає потужну основу для побудови асинхронних веб-сервісів, і розуміння особливостей цього стеку є важливим для обґрунтування архітектурних рішень розроблюваної системи.

Асинхронна модель виконання Python базується на концепції event loop — циклу подій, реалізованого бібліотекою asyncio. На відміну від багатопоточної моделі, де кожен запит обробляється в окремому потоці операційної системи, asyncio дозволяє одному потоку обслуговувати тисячі одночасних з’єднань. Ключовим механізмом є корутини (async def / await): при очікуванні завершення I/O-операції — мережевого запиту, звернення до БД або файлової операції — корутина призупиняється, а event loop передає

виконання іншій корутині. Це усуває марне блокування потоку і підвищує пропускну здатність сервісу без збільшення кількості процесів.

FastAPI побудований поверх бібліотек Starlette (ASGI-сервер) та Pydantic (валідація даних). ASGI (Asynchronous Server Gateway Interface) є сучасним стандартом Python для асинхронних веб-застосунків, що прийшов на зміну синхронному WSGI. FastAPI автоматично генерує OpenAPI-специфікацію та інтерактивну документацію Swagger UI на основі анотацій типів Python, що суттєво спрощує розробку та тестування REST API. Механізм ін'єкції залежностей через Depends дозволяє декларативно підключати спільні ресурси — сесію бази даних, поточного тенанта, UnitOfWork — до обробників маршрутів, уникаючи дублювання коду.

Інтеграція з асинхронним SQLAlchemy 2.x є важливою складовою стеку. На відміну від класичної синхронної ORM, async-версія використовує драйвер asyncpg — нативну асинхронну реалізацію протоколу PostgreSQL — що дозволяє виконувати запити до бази даних без блокування event loop. Це критично для бекенду конструктора сайтів, де один HTTP-запит може ініціювати кілька операцій паралельно: запис в БД, виклик Docker API та відповідь клієнту.

Окремої уваги потребує обробка синхронних операцій в асинхронному контексті. Docker SDK для Python реалізований у синхронному стилі — його функції блокують виконання до завершення HTTP-запиту до Docker socket. Виклик такої функції безпосередньо всередині async def заблокував би весь event loop. Для вирішення цієї проблеми Python надає функцію asyncio.to_thread, яка виконує синхронну функцію у відокремленому потоці з пулу потоків, не блокуючи event loop. Саме цей механізм застосовується в усіх методах DockerClient розроблюваної системи.

2.4 Особливості роботи з Docker SDK для Python та Traefik v3

Для програмного управління контейнерами в проєкті використовується офіційний Docker SDK для Python — бібліотека docker, що надає об'єктно-орієнтований інтерфейс до Docker Engine API. Розуміння внутрішніх механізмів SDK та Traefik є важливим для коректної реалізації модуля управління контейнерами і правильного проєктування взаємодії між компонентами.

Docker SDK ініціалізується через виклик docker.from_env(), що автоматично знаходить точку підключення до Docker-демона: Unix-сокет /var/run/docker.sock на Linux. Для роботи в контейнері API-сервісу цей сокет монтується як volume, що надає контейнеру доступ до Docker-демона хост-машини без запуску вкладеного демона. Основний метод SDK для запуску контейнерів — client.containers.run() — приймає назву образу та набір параметрів: name (ім'я контейнера), environment (словник змінних середовища), labels (словник Docker-міток для Traefik), network (ім'я Docker-мережі), restart_policy (політика перезапуску), mem_limit (обмеження пам'яті у форматі рядка: «512m», «1g») та nano_cpus (обмеження CPU у наносекундах). Параметр detach=True запускає контейнер у фоновому режимі та повертає об'єкт Container з ідентифікатором container.id для подальшого управління.

Traefik v3 виступає центральним маршрутизатором, що автоматично виявляє нові контейнери через Docker provider. При запуску Traefik з параметром --providers.docker=true він підключається до Docker socket та відстежує зміни метаданих контейнерів. Коли новий контейнер запускається з мітками traefik.enable=true та визначеними правилами маршрутизації, Traefik протягом кількох секунд реєструє відповідний маршрут без перезапуску. Параметр --providers.docker.exposedbydefault=false гарантує, що контейнери без явної мітки traefik.enable=true залишаються недоступними ззовні.

Система генерації Traefik-міток є ключовою для автоматичного призначення піддоменів тенантам. Для кожного контейнера генерується два роутери: HTTPS-роутер на entrypoint websecure з правилом Host(`<піддомен>`) і підключеним TLS-резолвером letsencrypt, та HTTP-роутер на entrypoint web з middleware типу redirectscheme для автоматичного перенаправлення HTTP-трафіку на HTTPS. Завдяки цьому контейнери тенантів доступні виключно по захищеному протоколу без жодного ручного налаштування.

TLS-сертифікати управляються Traefik автоматично через протокол ACME від Let’s Encrypt. У розроблюваній системі використовується HTTP-01 challenge: Traefik тимчасово відкриває спеціальний HTTP-ендпоінт на порту 80, за яким сервери Let’s Encrypt перевіряють право власності на домен. Після успішної перевірки Traefik отримує сертифікат та зберігає його у файлі acme.json, змонтованому як Docker volume для збереження між перезапусками сервісу. Сертифікати автоматично оновлюються до закінчення строку дії, що повністю усуває ручне управління TLS у системі.

2.5 Вибір технологічного стеку

На основі аналізу вимог та існуючих рішень було сформовано технологічний стек для розроблюваної системи. Система складається з трьох основних компонентів: бекенд-сервісу управління (оркестратора), системи маршрутизації (Traefik) та веб-інтерфейсу.

Для реалізації бекенд-сервісу обрано Python 3.12 з асинхронним фреймворком FastAPI. Вибір обумовлений кількома чинниками: FastAPI побудований на asyncio та Starlette, що забезпечує нативну асинхронну обробку запитів і дозволяє не блокувати event loop при взаємодії з Docker API. Офіційна бібліотека docker для Python надає повноцінний SDK для управління контейнерами, а asyncio.to_thread дозволяє виконувати синхронні виклики SDK без блокування. FastAPI також автоматично генерує OpenAPI-документацію та надає зручний механізм ін'єкції залежностей через Depends.

Для зберігання стану використовується SQLAlchemy 2.x у асинхронному режимі з бібліотекою asyncpg як драйвером PostgreSQL, а міграції виконуються через Alembic. Альтернативами розглядались Go (gin + Docker SDK) та Node.js (Express + dockerode), проте Python забезпечує оптимальний баланс між продуктивністю розробки, багатством екосистеми та зрілістю інструментів для роботи з базами даних.

Traefik v3 обрано як систему маршрутизації завдяки нативній інтеграції з Docker, підтримці автоматичного TLS через Let's Encrypt, динамічному оновленню конфігурації без перезапуску та підтримці middleware для додаткових налаштувань (заголовки, rate limiting тощо). Nginx або Kong API Gateway розглядались як альтернативи, але потребують складнішого налаштування для динамічної конфігурації.

Для зберігання стану системи (метадані тенантів, конфігурації контейнерів) обрано PostgreSQL. Реляційна модель даних добре підходить для зберігання конфігурацій, а PostgreSQL надає надійність і можливість розширення. SQLite розглядався для простоти, проте обмеження паралельного доступу роблять його неоптимальним для production-середовища.

Веб-інтерфейс реалізується як React SPA (Single Page Application) з використанням Vite як бандлера. REST API бекенду документується за допомогою OpenAPI/Swagger. Такий підхід забезпечує чіткий поділ між фронтендом та бекендом і спрощує потенційну інтеграцію зі сторонніми системами.

Для реалізації конструктора SPA-сайтів обрано бібліотеку GrapesJS — де-факто стандарт серед відкритих drag-and-drop редакторів. GrapesJS надає компонентну модель, панелі стилів, серіалізацію стану у JSONB та JavaScript API для програматичного управління вмістом. Альтернативами розглядались Quill.js (лише текст) та CKEditor (фокус на документах), але вони не підтримують повноцінне візуальне компонування. Для сповіщень про статус розгортання обрано WebSocket замість polling: це усуває надлишкові HTTP-запити і забезпечує миттєве відображення зміни стану. FastAPI підтримує

WebSocket нативно через ASGI-протокол. Для AI-генерації контенту обрано OpenAI API з моделлю gpt-4o-mini. Вибір обумовлений оптимальним балансом якості, швидкості та вартості, а також підтримкою JSON mode, що гарантує коректний формат відповіді. Таким чином, технологічний стек системи охоплює: Python 3.12 та FastAPI для реалізації асинхронного бекенду, SQLAlchemy 2.x та Alembic для роботи з PostgreSQL, Docker SDK для Python для взаємодії з Docker Engine API та побудови образів, Traefik v3 для динамічної маршрутизації та TLS, GrapesJS для drag-and-drop редактора сайтів, WebSocket для real-time сповіщень, OpenAI API (gpt-4o-mini) для AI-генерації, а також React та Vite для фронтенд-інтерфейсу.

2.6 Висновки з розділу 2

У другому розділі проведено аналіз існуючих платформ розгортання та порівняно технологічні альтернативи для реалізації конструктора сайтів. Розглянуто рішення Heroku, Render, Coolify та Caprover, виявлено їхні обмеження в контексті мультитенантної ізоляції та програмного API управління. На основі порівняльного аналізу встановлено, що Docker є оптимальним вибором завдяки розвиненому REST API, широкій екосистемі образів та нативній інтеграції з Traefik.

Сформований технологічний стек системи охоплює: Python 3 та FastAPI (бекенд), SQLAlchemy 2.x та Alembic (ORM та міграції), PostgreSQL з типом JSONB (база даних), Docker SDK для Python (управління контейнерами та побудова образів), Traefik v3 (маршрутизація та TLS), DuckDNS (wildcard DNS), GrapesJS (drag-and-drop редактор сайтів), WebSocket через FastAPI ASGI (real-time сповіщення), OpenAI API — gpt-4o-mini (AI-генерація HTML/CSS), React та Vite (фронтенд).

3 РОЗРОБКА КОНСТРУКТОРА САЙТІВ З ПІДТРИМКОЮ МУЛЬТИТЕНАНТНОСТІ

3.1 Постановка задачі

У попередніх розділах було проведено аналіз предметної області, досліджено технології контейнеризації та порівняно існуючі рішення. На основі цього аналізу сформовано технологічний стек і спроєктовано архітектуру системи. Цей розділ присвячено безпосередній реалізації розроблюваного конструктора сайтів — від налаштування середовища та структури проєкту до розгортання і тестування готової системи.

Метою третього розділу є практична реалізація конструктора сайтів відповідно до спроєктованої архітектури. Розділ охоплює реалізацію доменних моделей та схеми бази даних, побудову репозиторіїв і сервісного шару, розробку модуля управління Docker-контейнерами, реалізацію конструктора SPA-сайтів на основі GrapesJS з підтримкою AI-генерації контенту та WebSocket-відстеженням статусу, налаштування інтеграції з Traefik v3 та DuckDNS, а також розробку фронтенд-застосунку та тестування системи. Розділ охоплює реалізацію доменних моделей та схеми бази даних, побудову репозиторіїв і сервісного шару, розробку модуля управління Docker-контейнерами, налаштування інтеграції з Traefik v3 та DuckDNS, а також розробку фронтенд-застосунку та тестування системи. Кожен тенант, зареєстрований у системі, отримує ізольований Docker-контейнер та унікальний піддомен без будь-якого ручного втручання адміністратора.

Цільова аудиторія системи охоплює кілька категорій користувачів. Основна аудиторія — малий та середній бізнес без штатних програмістів, якому необхідно швидко опублікувати представницький сайт і не перейматись серверною інфраструктурою. Для таких користувачів система надає вбудований візуальний редактор сайтів на основі GrapesJS, де можна сконструювати SPA-сторінку без жодного рядка коду. Для прискорення

роботи доступна інтеграція з OpenAI: достатньо описати потрібний сайт звичайною мовою, і штучний інтелект згенерує готову HTML/CSS-секцію прямо в редакторі, яку потім можна відредагувати. Друга категорія — технічні спеціалісти та розробники, які потребують швидкого розгортання будь-якого Docker-образу з Docker Hub. Завдяки PaaS-режиму системи можна запустити WordPress із передвстановленим SQLite, власний Node.js-застосунок або будь-який інший сервіс: досить вказати назву образу, і через кілька секунд контейнер стане доступним за персональним HTTPS-піддоменом. Третя категорія — студенти та дослідники, які хочуть швидко опублікувати прототип без витрат на хостинг.

Система зосереджена на реалізації трьох основних завдань: управлінні тенантами (реєстрація, автентифікація, збереження конфігурацій), управлінні Docker-контейнерами через Docker Engine API (запуск, зупинка, видалення з підтримкою ресурсних лімітів), а також автоматичному налаштуванні маршрутизації через Traefik v3 з підтримкою TLS. Усі операції виконуються через REST API бекенду, а взаємодія з системою здійснюється через веб-інтерфейс.

3.2 Вимоги до програмної системи

Вимоги до системи повинні відображати її основну мету — надати розробникам та адміністраторам зручний інструмент для розгортання веб-застосунків у ізольованих Docker-контейнерах з автоматичним призначенням піддоменів.

До функціональних вимог системи належать: підтримка реєстрації та автентифікації тенантів на основі JWT-токенів; можливість запуску Docker-контейнера з довільним образом з передачею змінних середовища, обмеженням пам'яті та CPU; автоматичне призначення піддомену на основі унікального ідентифікатора (slug) тенанта; перегляд списку активних розгортань та їх статусу; зупинка та видалення розгортань; синхронізація

статусу контейнера з реальним станом Docker-демона; можливість створення SPA-сайтів у візуальному drag-and-drop редакторі GrapesJS; збереження вмісту сайту (HTML, CSS, GrapesJS JSON) у базі даних; публікація сайту в один клік із автоматичним складанням Docker-образу на основі збережених HTML та CSS; відстеження статусу розгортання в реальному часі через WebSocket-з'єднання без перезавантаження сторінки; AI-генерація секцій сайту через OpenAI API прямо у редакторі.

До нефункціональних вимог належать: асинхронна обробка запитів для забезпечення паралельної роботи з Docker API без блокування event loop; ізоляція тенантів на рівні контейнерів та мережі; підтримка автоматичного отримання TLS-сертифікатів через Let's Encrypt; зберігання стану системи в реляційній базі даних PostgreSQL; дотримання принципів CQRS та чистої архітектури у структурі бекенду; підтримка slug-ідентифікаторів до 63 символів відповідно до DNS-специфікації; можливість масштабування системи шляхом додавання нових сервісів без зміни існуючої архітектури; збереження стану GrapesJS-редактора у форматі JSONB для відновлення сесії редагування; підтримка WebSocket як постійного двостороннього каналу для надсилання подій зміни статусу; коректна обробка повторних публікацій сайту (re-deploy без простою).

3.3 Архітектура програмної системи

Важливою частиною задоволення вимог є вибір правильної архітектури системи. Розроблена система має клієнт-серверну архітектуру, де бекенд виступає оркестратором інфраструктури, а фронтенд забезпечує інтерфейс для управління розгортаннями.

Бекенд-сервіс реалізовано на Python 3 з використанням FastAPI як основного веб-фреймворку. Вибір FastAPI обумовлений нативною підтримкою асинхронної обробки запитів (asyncio), автоматичною генерацією OpenAPI-документації, а також зручним механізмом ін'єкції залежностей

через Depends. Для взаємодії з базою даних використовується SQLAlchemy 2.x в асинхронному режимі з бібліотекою asyncpg як драйвером PostgreSQL.

Архітектура бекенду побудована за принципом CQRS (Command Query Responsibility Segregation), що передбачає чіткий поділ операцій на команди (мутація стану) та запити (читання даних). Кожна команда або запит є окремим модулем з єдиною функцією-обробником handle, яка приймає об'єкт команди або запиту та UnitOfWork, і виконує відповідну операцію. Репозиторії та сесії бази даних управляються через патерн Unit of Work, який інкапсулює відкриття, закриття та відкат транзакцій.

Зовнішня маршрутизація забезпечується Traefik v3, який виступає зворотнім проксі та автоматично виявляє конфігурацію контейнерів через Docker labels. При запуску нового контейнера Traefik миттєво реєструє нове правило маршрутизації без перезапуску. TLS-сертифікати отримуються автоматично через протокол ACME від Let's Encrypt з використанням HTTP-01 challenge. DNS вирішується за допомогою DuckDNS — безкоштовного сервісу динамічного DNS, що підтримує wildcard-записи, дозволяючи динамічно призначати піддомени без ручного редагування DNS-зони. Окрім PaaS-режиму розгортання довільних Docker-образів, система реалізує вбудований конструктор SPA-сайтів на основі бібліотеки GrapesJS. Користувач редагує сайт у візуальному редакторі, після чого вміст (HTML, CSS та внутрішній стан GrapesJS у форматі JSON) зберігається в базі даних. Публікація сайту ініціює фоновий процес складання Docker-образу з nginx:alpine та згенерованим index.html, а потім запуску контейнера. Статус кожного етапу (QUEUED, BUILDING, DEPLOYING, RUNNING, FAILED) передається клієнту в реальному часі через WebSocket-з'єднання без перезавантаження сторінки. Додатково система інтегрована з OpenAI API: за текстовим описом від користувача генерується готовий HTML/CSS-блок, який вставляється прямо в редактор GrapesJS.

Загальна схема компонентів системи зображена на рисунку 1.

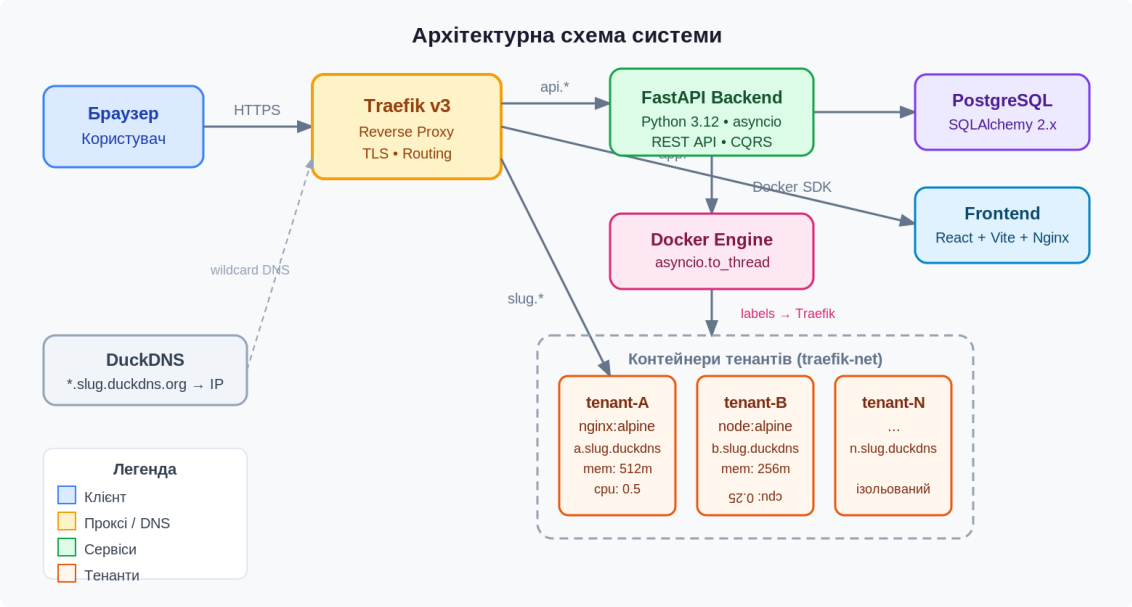


Рисунок *1 — Архітектурна схема системи*

Взаємодія компонентів відбувається наступним чином: браузер користувача надсилає HTTPS-запит на відповідний піддомен; Traefik перехоплює запит і маршрутизує його до відповідного контейнера тенанта на основі правила Host; контейнер тенанта обробляє запит і повертає відповідь. Управління цим процесом здійснюється через REST API бекенду, доступний за адресою api.домен, де фронтенд-застосунок виконує операції реєстрації, входу та управління розгортаннями.

3.4 Реалізація серверної частини

3.4.1 Доменні моделі та бази даних

Для представлення сутностей предметної області використовуються доменні моделі на основі SQLAlchemy 2.x з декларативним підходом. Базовий клас BaseModel успадковується від DeclarativeBase і є спільним для всіх моделей системи. Для додавання повторюваних полів використовуються міксини: TimestampMixin додає поля created_at та updated_at з автоматичним заповненням на рівні СУБД, а SoftDeleteMixin додає поле deleted_at, яке

дозволяє реалізувати м'яке видалення записів без їх фізичного видалення з бази даних.

Модель Tenant описує тенанта системи та містить поля: slug — унікальний URL-безпечний ідентифікатор до 63 символів, що використовується для формування піддомену; name та email — назва та контактна адреса тенанта; hashed_password — хешований пароль, що зберігається з використанням алгоритму bcrypt; is_active — прапорець активності облікового запису. Модель Deployment описує розгортання і містить посилання на тенанта (tenant_id), назву Docker-образу (image), ідентифікатор запущеного контейнера (container_id), піддомен (subdomain), поточний статус (status), а також необов'язкові обмеження ресурсів memory_limit та cpu_limit.

Міграції схеми бази даних виконуються за допомогою Alembic. Перша міграція (0001_initial.py) створює таблиці tenants та deployments з усіма необхідними полями, індексами та зовнішніми ключами. Підключення до бази даних реалізовано через асинхронний рушій create_async_engine з драйвером asyncpg, а фабрика сесій async_sessionmaker забезпечує ефективне перевикористання з'єднань.

З додаванням конструктора сайтів у схему бази даних було введено третю таблицю — sites. Модель Site містить такі поля: id (первинний ключ), tenant_id (зовнішній ключ до tenants), name (назва сайту), slug (унікальний DNS-ідентифікатор), grapes_data (JSONB — внутрішній стан GrapesJS-редактора для відновлення сесії редагування), html та css (відрендерений вміст для складання Docker-образу при публікації), а також deployment_id — зовнішній ключ до deployments, який заповнюється при першій публікації. Поля grapes_data, html, css та deployment_id є nullable: до першого збереження в редакторі вони порожні. Структуру моделі Site наведено у лістингу 1.

Лістинг *1 — Модель Site (app/domain/models/site.py)*

```
class Site(BaseModel, TimestampMixin, SoftDeleteMixin):
    __tablename__ = "sites"

    id: Mapped[int] = mapped_column(primary_key=True)
    tenant_id: Mapped[int] = mapped_column(
        ForeignKey("tenants.id"), nullable=False, index=True
    )
    name: Mapped[str] = mapped_column(String(255), nullable=False)
    slug: Mapped[str] = mapped_column(String(63), unique=True,
nullable=False)

    # GrapesJS project state — reloaded into editor on reopen
    grapes_data: Mapped[dict | None] = mapped_column(JSONB, nullable=True)
    html: Mapped[str | None] = mapped_column(Text, nullable=True)
    css: Mapped[str | None] = mapped_column(Text, nullable=True)

    # FK to last published Deployment; null until first publish
    deployment_id: Mapped[int | None] = mapped_column(
        ForeignKey("deployments.id"), nullable=True
    )
```

3.4.2 Репозиторії та Unit of Work

Для абстрагування доступу до бази даних застосовується патерн Repository. Базовий клас BaseRepository є узагальненим (generic) та параметризується типом моделі. Він надає базові методи: get_by_id для отримання запису за первинним ключем, add для додавання нової сутності до сесії, delete для позначення сутності на видалення, list_all для отримання всього переліку записів. Конкретні репозиторії TenantRepository та DeploymentRepository розширюють базовий клас, додаючи специфічні методи запиту: наприклад, get_by_email та get_by_slug для тенантів, а також get_by_id_and_tenant та list_by_tenant для розгортань.

Управління сесіями бази даних реалізовано через патерн Unit of Work. Клас UnitOfWork використовується як асинхронний контекстний менеджер: при вході в контекст (async with uow) відкривається нова сесія бази даних та ініціалізуються екземпляри репозиторіїв; при виходах з контексту — закривається. При виникненні виключення сесія автоматично відкочується. Явне збереження змін виконується через метод commit, що дозволяє точно контролювати межі транзакцій.

3.4.3 Команди та запити (CQRS)

Відповідно до принципу CQRS, всі операції поділені на дві категорії. Команди знаходяться у директорії app/commands та виконують операції, що змінюють стан системи. Кожна команда являє собою dataclass з полями-параметрами операції та функцію-обробник handle, що приймає екземпляр команди та UnitOfWork. Наприклад, CreateDeploymentCommand містить поля tenant_id, image, env_vars, memory_limit, cpu_limit. Обробник команди відкриває UoW, ініціалізує відповідний сервіс, виконує операцію та підтверджує транзакцію.

Запити знаходяться у директорії app/queries та виконують виключно читання даних без будь-якої зміни стану. Запити звертаються до репозиторіїв безпосередньо через UnitOfWork, не використовуючи сервісний шар. Наприклад, ListDeploymentsQuery приймає tenant_id та повертає перелік розгортань тенанта, а GetDeploymentQuery додатково фільтрує за ідентифікатором конкретного розгортання.

Такий підхід забезпечує чіткий поділ відповідальності між операціями читання та запису. Роутери викликають лише обробники команд або запитів, не містять бізнес-логіки та не звертаються безпосередньо до сервісів чи репозиторіїв.

3.4.4 Автентифікація та безпека

Автентифікація тенантів реалізована на основі JWT (JSON Web Tokens). При реєстрації пароль хешується бібліотекою bcrypt з автоматичним генеруванням salt. При вході система перевіряє пароль, а у разі успіху видає підписаний токен доступу. Токен містить ідентифікатор тенанта у полі sub та час закінчення дії (за замовчуванням 7 днів). Для підпису використовується алгоритм HS256 з секретним ключем з конфігурації.

FastAPI-залежність get_current_tenant автоматично валідує Bearer-токен з заголовка Authorization, декодує його, завантажує тенанта з бази даних та перевіряє його активність. Якщо токен відсутній, недійсний або тенант

деактивований — повертається відповідь 401 Unauthorized. Ця залежність підключається до всіх захищених маршрутів через механізм Depends. Декодування виконується бібліотекою python-jose: перевіряється підпис (HS256), термін дії (exp) та наявність поля sub. Зберігання токена на стороні клієнта виконується в localStorage браузера; при кожному API-запиті фронтенд підставляє його у заголовок Authorization: Bearer. Оскільки токени не зберігаються на сервері (stateless), відкликання токена реалізується через деактивацію тенанта (поле is_active = False) — після цього залежність get_current_tenant повертає 401 навіть для токенів із дійсним підписом.

3.5 Реалізація конструктора SPA-сайтів

Поряд з PaaS-режимом розгортання довільних Docker-образів система реалізує вбудований конструктор SPA-сайтів. Цей модуль дозволяє користувачеві без знань програмування сконструювати повноцінну односторінкову веб-сторінку у візуальному редакторі, опублікувати її в один клік та отримати живе HTTPS-посилання. Технічно модуль охоплює інтеграцію GrapesJS, фонову побудову Docker-образу, WebSocket-повідомлення про статус та AI-генерацію контенту.

3.5.1 Редактор GrapesJS та збереження контенту

GrapesJS — відкрита JavaScript-бібліотека для створення повноцінних drag-and-drop редакторів веб-контенту. Вона надає компонентну модель документа: кожен блок (секція, текст, зображення, кнопка) є незалежним компонентом з власними властивостями, які редагуються через панель стилів. GrapesJS зберігає внутрішній стан проєкту у форматі JSON (так зване projectData), що дозволяє повністю відновити стан редактора при повторному відкритті сайту. Крім projectData, бібліотека може рендерувати фінальні HTML та CSS для подальшого використання.

У фронтенд-застосунку GrapesJS інтегровано через власний React-компонент GrapesEditor. При монтуванні компонента ініціалізується екземпляр редактора, до якого завантажується збережений projectData із бази даних (або порожня сторінка для нового сайту). Методи getProjectData(), getHtml() та getCss() компонента відкривають батьківській сторінці BuilderPage доступ до поточного стану редактора через React ref. Інтерфейс конструктора сайтів показано на рисунку 11.

Компонент GrapesEditor є обгорткою над GrapesJS для React. Він ініціалізує редактор при монтуванні, завантажує збережений стан та відкриває батьківській сторінці методи через useImperativeHandle. Ключові частини компонента наведено у лістингу 8.

Лістинг *8 — GrapesEditor компонент (src/components/GrapesEditor.tsx)*

```
const GrapesEditor = forwardRef<GrapesEditorHandle, Props>(({ site }, ref)
=> {
  const containerRef = useRef<HTMLDivElement>(null);
  const editorRef = useRef<grapesjs.Editor | null>(null);

  useEffect(() => {
    const editor = grapesjs.init({
      container: containerRef.current!,
      storageManager: false,  // manual save
      height: "100%",
    });
    if (site.grapes_data) {
      editor.loadProjectData(site.grapes_data);
    }
    editorRef.current = editor;
    return () => editor.destroy();
  }, []);

  useImperativeHandle(ref, () => ({
    getProjectData: () => editorRef.current!.getProjectData(),
    getHtml: () => editorRef.current!.getHtml(),
    getCss:  () => editorRef.current!.getCss() ?? "",
    insertContent: (html: string, css: string) => {
      editorRef.current!.setComponents(html);
      editorRef.current!.setStyle(css);
    },
  }));
  return <div ref={containerRef} style={{ flex: 1 }} />;
});
```

Зовнішній вигляд сторінки "My Sites" зі списком створених сайтів показано на рисунку 10, а форма створення нового сайту на рисунку 11.

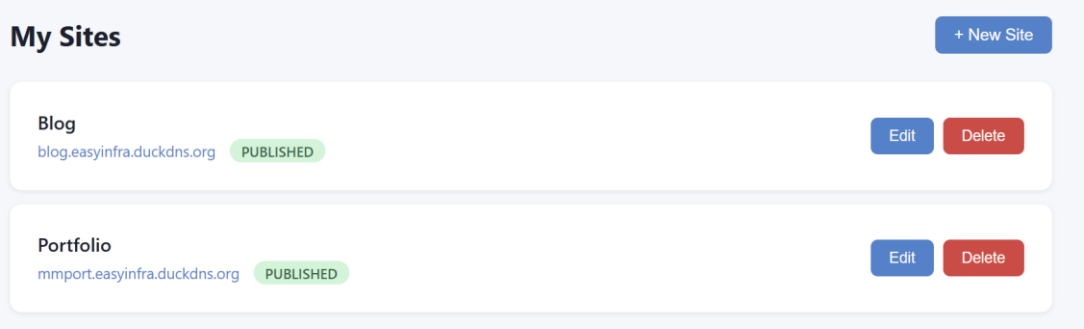


Рисунок *8 — Сторінка My Sites з переліком сайтів та статусами публікації*

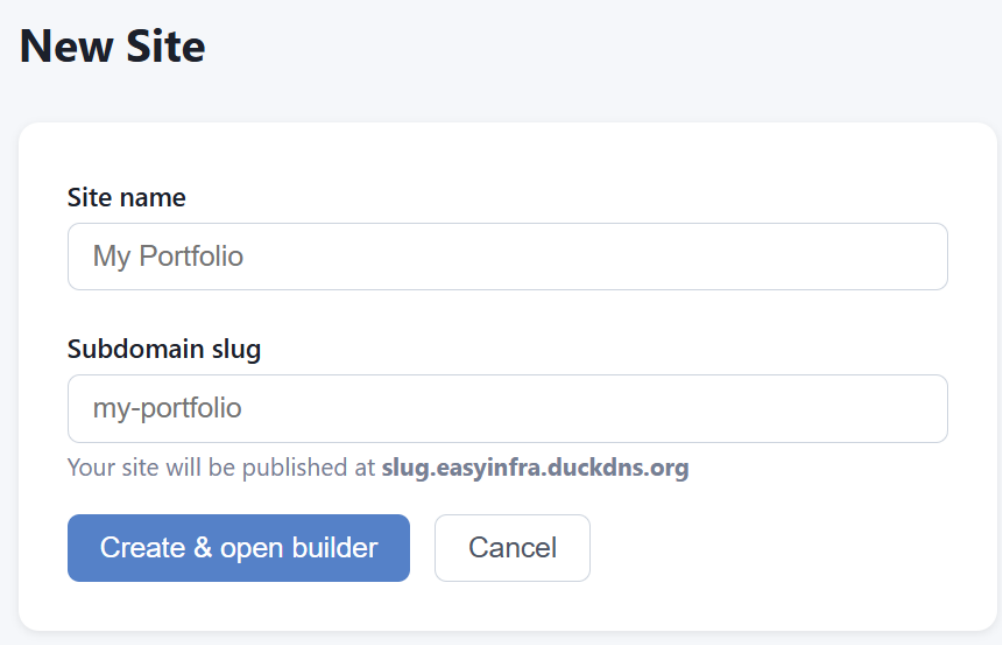


Рисунок *10 — Форма створення нового сайту — введення назви та slug*

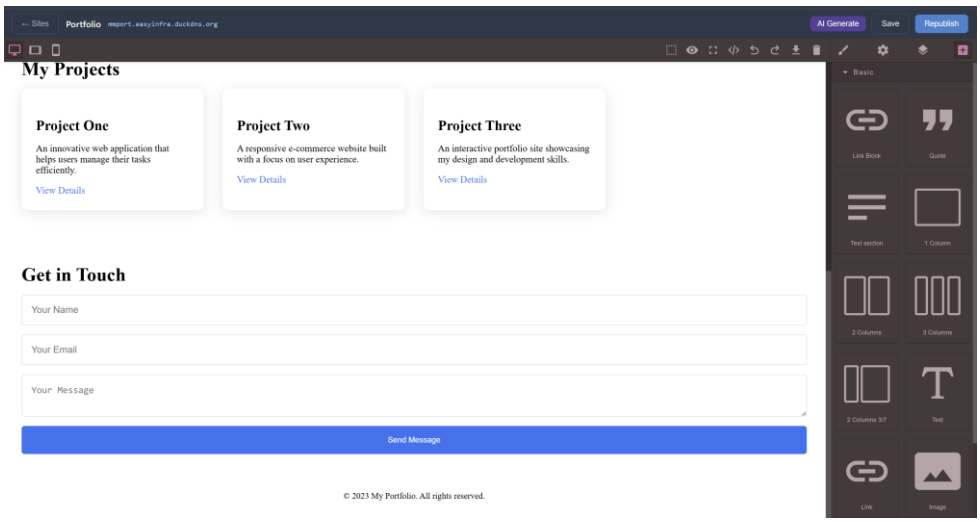


Рисунок *11 — Сторінка конструктора сайтів з GrapesJS редактором*

При натисканні кнопки "Save" фронтенд збирає projectData, html та css через ref-методи компонента і надсилає PUT-запит до ендпоінту PUT /api/v1/sites/{site_id}. На бекенді SaveSiteCommand оновлює відповідні поля моделі Site в базі даних. Структуру запиту визначає схема SiteSave (Pydantic BaseModel): поля grapes_data (dict), html (str) та css (str). Збережений стан редактора наведено у лістингу 2.

Лістинг *2 — Схема збереження сайту (app/services/site_service.py)*

```
# PUT /api/v1/sites/{site_id}
class SiteSave(BaseModel):
    grapes_data: dict   # GrapesJS project JSON
    html: str           # rendered HTML body
    css: str            # rendered CSS

# SiteService.save_content()
async def save_content(self, site_id, tenant_id, grapes_data, html, css):
    site = await self._uow.sites.get_by_id_and_tenant(site_id, tenant_id)
    if not site:
        raise SiteNotFoundError(f"Site {site_id} not found")
    site.grapes_data = grapes_data
    site.html = html
    site.css = css
```

3.5.2 Публікація сайту — складання Docker-образу

Публікація сайту відбувається через POST /api/v1/sites/{site_id}/publish. PublishSiteCommand передає запит до SiteService.prepare_publish(), який перевіряє наявність збереженого HTML і або створює новий запис Deployment

зі статусом PENDING, або позначає вже існуючий як RESTART_REQUESTED для повторного розгортання. Після підтвердження транзакції роутер запускає фонову корутину build_and_deploy_site() через asyncio.create_task().

Фонова задача build_and_deploy_site() виконує наступну послідовність кроків, оновлюючи статус розгортання на кожному з них. Спочатку із бази даних завантажуються html та css збереженого сайту. Потім викликається метод DockerClient.build_image(), який динамічно формує повний HTML-документ (вставляючи CSS у тег <style>), пакує Dockerfile та index.html у tar-архів в пам'яті та передає потік у Docker Engine API для складання образу із тегом site-{slug}:latest. Базовим образом є nginx:alpine, до якого копіюється згенерований index.html. Лістинг 3 демонструє реалізацію build_image.

Лістинг *3 — Метод DockerClient.build_image (app/infrastructure/docker/client.py)*

```
async def build_image(self, tag: str, html: str, css: str) -> None:
    full_html = (
        "<!DOCTYPE html>\n<html>\n<head>\n"
        f"  <style>\n{css}\n  </style>\n</head>\n"
        f"<body>\n{html}\n</body>\n</html>"
    )
    dockerfile = b"FROM nginx:alpine\n"
                 b"COPY index.html /usr/share/nginx/html/index.html\n"
    # Pack into in-memory tar archive
    buf = io.BytesIO()
    with tarfile.open(fileobj=buf, mode="w") as tar:
        _add_bytes(tar, "Dockerfile", dockerfile)
        _add_bytes(tar, "index.html", full_html.encode("utf-8"))
    buf.seek(0)
    # Send to Docker Engine API (in thread pool)
    await asyncio.to_thread(
        self._client.images.build, fileobj=buf, custom_context=True,
tag=tag
    )
```

Після успішного складання образу метод run_container() запускає контейнер у мережі traefik-net з відповідними Traefik-мітками. Traefik автоматично реєструє HTTPS-роутер для піддомену slug.duckdns.org. У разі повторної публікації (RESTART_REQUESTED) метод rebuild_and_redeploy_site() спочатку зупиняє та видаляє старий контейнер,

після чого повторно виконує build_and_deploy_site(). Це забезпечує оновлення сайту без тривалого простою.

Фоновий процес публікації реалізовано у функції build_and_deploy_site(), яка послідовно оновлює статус розгортання через WebSocket на кожному кроці. Спрощений фрагмент функції наведено у лістингу 6.

Лістинг *6 — Фоновий процес публікації (app/services/site_background.py)*

```
async def build_and_deploy_site(site_id: int, deployment_id: int) -> None:
    docker = DockerClient()
    async with UnitOfWork(session_factory) as uow:
        site = await uow.sites.get_by_id(site_id)
        html, css, slug = site.html, site.css, site.slug

    image_tag = f"site-{slug}:latest"
    await _set_status(deployment_id, DeploymentStatus.QUEUED)

    await _set_status(deployment_id, DeploymentStatus.BUILDING)
    try:
        await docker.build_image(tag=image_tag, html=html, css=css)
    except Exception as e:
        await _set_status(deployment_id, DeploymentStatus.FAILED,
error=str(e))
        return

    await _set_status(deployment_id, DeploymentStatus.DEPLOYING)
    labels =
build_traefik_labels(subdomain=f"{slug}.{settings.base_domain}",
                                container_name=f"tenant-{slug}")
    cid = await docker.run_container(image=image_tag, name=f"tenant-
{slug}",
                                     env={}, labels=labels, ...)
    await _set_status(deployment_id, DeploymentStatus.RUNNING,
container_id=cid)
```

3.5.3 WebSocket — відстеження статусу розгортання в реальному часі

Традиційний підхід до відстеження прогресу довготривалих операцій — опитування (polling) сервера — потребує постійних HTTP-запитів і спричиняє затримку у відображенні змін. Протокол WebSocket забезпечує постійне двостороннє з'єднання між клієнтом і сервером, через яке сервер може надсилати події клієнту без запиту з його боку.

На бекенді реалізовано клас ConnectionManager, який зберігає множину відкритих WebSocket-з'єднань для кожного deployment_id. При виклику ws_manager.broadcast(deployment_id, event) повідомлення надсилається всім підключеним клієнтам для цього розгортання. Клас ConnectionManager наведено у лістингу 4.

Лістинг *4 — ConnectionManager (app/infrastructure/websocket/manager.py)*

```
class ConnectionManager:
    def __init__(self) -> None:
        self._connections: dict[int, set[WebSocket]] = defaultdict(set)

    async def connect(self, deployment_id: int, ws: WebSocket) -> None:
        await ws.accept()
        self._connections[deployment_id].add(ws)

    async def broadcast(self, deployment_id: int, data: dict) -> None:
        dead: set[WebSocket] = set()
        for ws in list(self._connections.get(deployment_id, [])):
            try:
                await ws.send_json(data)
            except Exception:
                dead.add(ws)
        for ws in dead:
            self.disconnect(deployment_id, ws)

ws_manager = ConnectionManager()
```

Ендпоінт WebSocket зареєстровано в FastAPI через app.add_api_websocket_route("/api/v1/deployments/{deployment_id}/ws", deployment_ws). При підключенні клієнт передає JWT-токен як query-параметр token (заголовок Authorization недоступний у WebSocket-рукостисканні). Сервер валідує токен через функцію decode_token(), і лише після успішної перевірки приймає з'єднання. На фронтенді React-хук useDeploymentWs() встановлює WebSocket-з'єднання одразу після запуску публікації і слухає події {deploymentId, status, error}. При зміні статусу компонент оновлює стан без перезавантаження сторінки — користувач бачить послідовність QUEUED → BUILDING → DEPLOYING → RUNNING у реальному часі.

Ендпоінт WebSocket виконує автентифікацію через JWT-токен, переданий як query-параметр, а потім підписується на події через ConnectionManager. Реалізацію ендпоінту наведено у лістингу 7.

Лістинг *7 — WebSocket-ендпоінт (app/api/v1/deployments.py)*

```
async def deployment_ws(
    websocket: WebSocket, deployment_id: int, token: str = ""
) -> None:
    tenant_id_str = decode_token(token)
    if not tenant_id_str:
        await websocket.close(code=4001)  # Unauthorized
        return

    await ws_manager.connect(deployment_id, websocket)
    try:
        # Send current status immediately on connect
        async with UnitOfWork(session_factory) as uow:
            dep = await uow.deployments.get_by_id(deployment_id)
            if dep:
                await websocket.send_json({
                    "deploymentId": deployment_id,
                    "status": dep.status,
                })
        while True:
            await websocket.receive_text()  # keep alive
    except WebSocketDisconnect:
        ws_manager.disconnect(deployment_id, websocket)
```

3.5.4 AI-генерація контенту через OpenAI API

Для зниження порогу входження для нетехнічних користувачів система інтегрована з OpenAI API. Користувач натискає кнопку "AI Generate" у редакторі GrapesJS, вводить опис потрібної секції звичайною мовою (наприклад, "Секція hero для ресторану з назвою та кнопкою") і через кілька секунд у редакторі з'являється готова HTML/CSS-секція преміум-якості, яку він може далі відредагувати.

На бекенді запит обробляє AIService, який надсилає два повідомлення до OpenAI Chat Completions API: системний промпт із детальними вимогами до якості та формату відповіді, і повідомлення користувача з його описом. Системний промпт вимагає від моделі повернути виключно валідний JSON із ключами html та css, використовуючи response_format: {type: json_object} (JSON mode), що гарантує коректну структуру відповіді. Вимоги до дизайну в

системному промпті орієнтовані на якість преміум-шаблонів, використання Flexbox та CSS Grid, семантичного HTML5 та адаптивних брейкпоінтів. Реалізацію AIService показано у лістингу 5.

Лістинг *5 — AIService.generate_site (app/services/ai_service.py)*

```
class AIService:
    async def generate_site(self, prompt: str) -> tuple[str, str]:
        response = await self._client.chat.completions.create(
            model=settings.openai_model,    # gpt-4o-mini
            response_format={"type": "json_object"},
            temperature=0.7,
            messages=[
                {"role": "system", "content": _SYSTEM_PROMPT},
                {"role": "user", "content": f"Create: {prompt}"},
            ],
        )
        raw = response.choices[0].message.content or "{}"
        data = json.loads(raw)
        return data.get("html", ""), data.get("css", "")
```

На фронтенді результат генерації вставляється в редактор GrapesJS через метод editor.setComponents(html), після чого css-стилі додаються через editor.setStyle(css). Це дозволяє одразу редагувати згенерований контент у звичному drag-and-drop інтерфейсі. Інтерфейс AI-генерації у редакторі показано на рисунку 9.



Рисунок *9 — AI-генерація секцій сайту в редакторі GrapesJS*

3.6 Реалізація модуля управління Docker-контейнерами

Взаємодія з Docker-демоном реалізована через клас DockerClient, що обгортає офіційний Docker SDK для Python (бібліотека docker). При ініціалізації клієнт підключається до локального Docker-демона через Unix-сокет /var/run/docker.sock, змонтований у контейнер API-сервісу. Якщо підключення неможливе, одразу генерується виключення DockerOperationError.

Ключовою особливістю реалізації є повністю асинхронна обробка викликів Docker SDK. Оскільки бібліотека docker є синхронною, а FastAPI побудований на asyncio, блокуючі виклики SDK виконуються через asyncio.to_thread, що дозволяє не блокувати event loop під час тривалих операцій — наприклад, завантаження образу або запуску контейнера.

Метод run_container відповідає за запуск нового контейнера. Він приймає назву образу, ім'я контейнера, словник змінних середовища, словник Traefik-міток та мережу Docker. Опційно можна передати обмеження пам'яті у форматі рядка (наприклад, «512m») та кількість CPU у вигляді числа з плаваючою крапкою, які конвертуються у відповідні параметри Docker API (mem_limit та nano_cpus). Контейнер запускається з політикою перезапуску unless-stopped, що забезпечує автоматичний перезапуск при збоях.

Методи stop_container та remove_container зупиняють та видаляють контейнер за його ідентифікатором. Виключення від Docker SDK при їх виконанні перехоплюються і не передаються далі — це свідоме рішення, оскільки контейнер може вже бути зупиненим або видаленим вручну. Метод get_container_status повертає поточний статус контейнера (running, exited, paused тощо) або None, якщо контейнер не знайдено.

Клас DeploymentService є сервісним шаром, що координує роботу репозиторіїв та DockerClient. При створенні розгортання сервіс виконує наступну послідовність: завантажує тенанта з бази даних, валідує slug за регулярним виразом, формує ім'я піддомену та контейнера, створює запис про розгортання у базі даних зі статусом starting, виконує flush для отримання ідентифікатора запису, генерує Traefik-мітки та запускає Docker-контейнер. У разі помилки при запуску контейнера статус розгортання встановлюється як failed, а виключення передається далі для обробки в роутері.

Послідовність операцій при створенні розгортання наведена на рисунку 2.

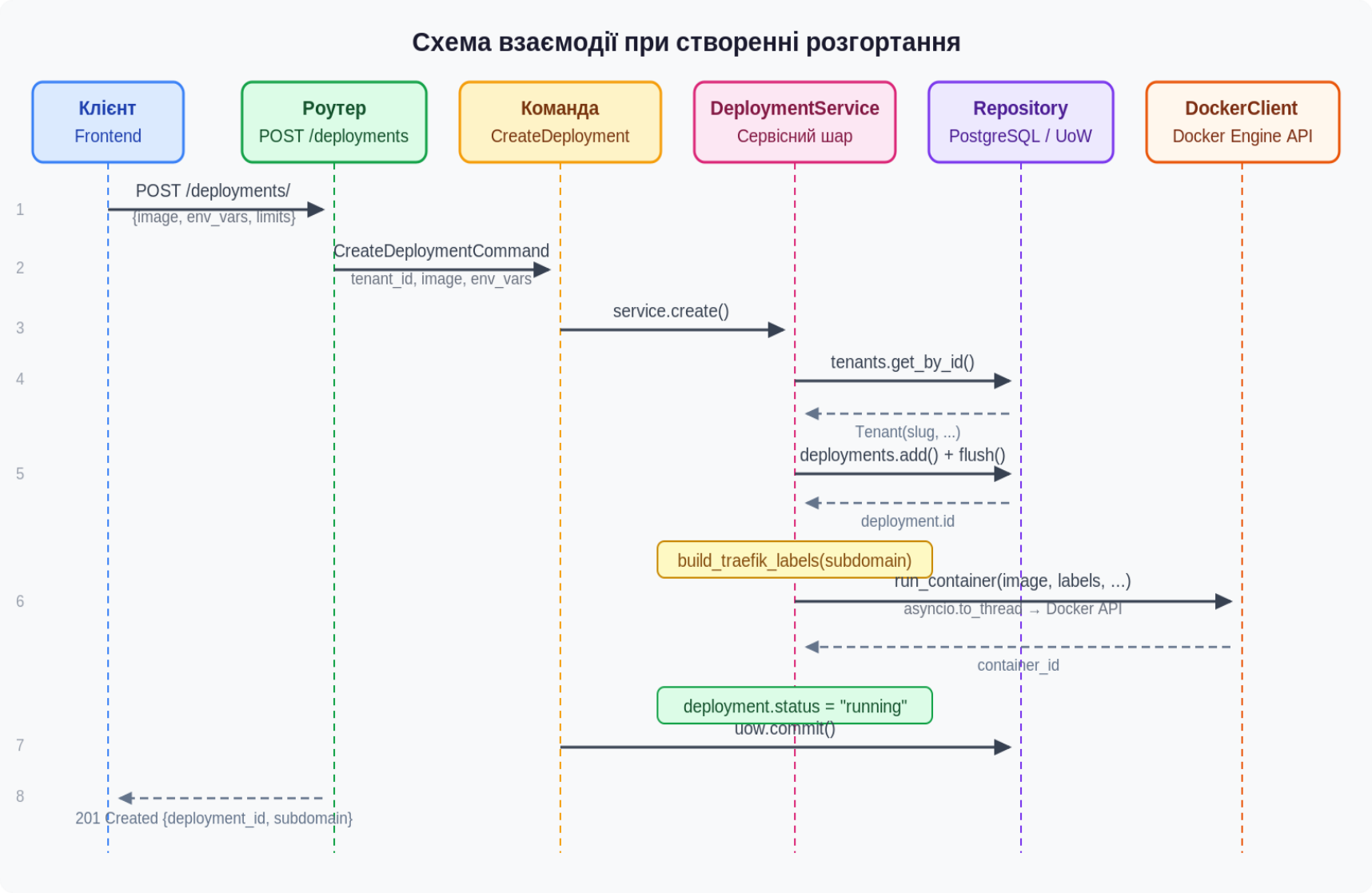


Рисунок *2* — *Схема взаємодії компонентів при створенні розгортання*

3.7 Налаштування маршрутизації та DNS

Функція build_traefik_labels генерує словник Docker-міток, які Traefik зчитує при запуску контейнера. Для кожного тенантського контейнера формуються два роутера: HTTPS-роутер на entrypoint websecure з правилом Host(`<піддомен>`) та підключеним TLS-resolver, і HTTP-роутер на entrypoint web, що автоматично перенаправляє трафік на HTTPS через middleware redirectscheme. Завдяки цьому всі контейнери тенантів доступні виключно по захищеному протоколу.

Traefik налаштований на прослуховування Docker socket у режимі read-only. При появі нового контейнера з міткою traefik.enable=true Traefik автоматично підхоплює нову конфігурацію без перезапуску. Параметр exposedbydefault=false забезпечує, що без явної мітки traefik.enable=true контейнер не буде доступний ззовні. TLS-сертифікати зберігаються у volume traefik-certs/acme.json та автоматично оновлюються до закінчення строку дії.

Для забезпечення маршрутизації DNS використовується сервіс DuckDNS. DuckDNS — це безкоштовний сервіс динамічного DNS, що дозволяє зареєструвати піддомен виду mysite.duckdns.org та прив'язати його до довільної IP-адреси. Wildcard-запис *.mysite.duckdns.org направляє весь трафік з будь-якого піддомену на IP-адресу сервера, де Traefik вже розподіляє запити між конкретними контейнерами відповідно до правил маршрутизації. Такий підхід дозволяє динамічно створювати нові піддомени для тенантів без будь-яких змін у DNS-конфігурації.

Налаштування системи потребує виконання кількох підготовчих кроків: реєстрації домену на DuckDNS та налаштування wildcard-запису; створення зовнішньої Docker-мережі traefik-net командою docker network create traefik-net; заповнення файлу конфігурації .env з параметрами бази даних, домену та секретного ключа; запуску всіх сервісів командою docker compose up -d. Після запуску Traefik автоматично отримує TLS-сертифікат від Let's Encrypt через HTTP-01 challenge.

Зовнішній вигляд Traefik Dashboard із зареєстрованими роутерами тенантів показано на рисунку 3.

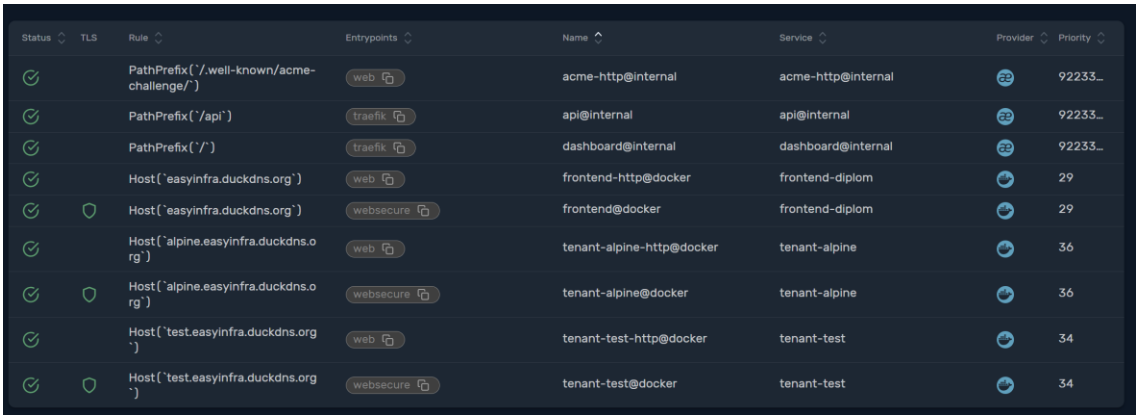


Рисунок *3 — Dashboard Traefik з активними роутерами тенантів*

3.8 Розробка веб-інтерфейсу

Фронтенд-застосунок реалізовано як Single Page Application (SPA) на TypeScript з використанням бібліотеки React та збирача Vite. Застосунок взаємодіє з бекендом через REST API, а маршрутизація між сторінками реалізована за допомогою React Router. Для розгортання фронтенду використовується Docker-контейнер з Nginx, що роздає статичні файли збірки.

Застосунок складається з чотирьох основних сторінок. Сторінка реєстрації (RegisterPage) містить форму для введення slug, імені, email та пароля. Після успішної реєстрації виконується автоматичний вхід в систему. Сторінка входу (LoginPage) дозволяє автентифікуватися за email та паролем, після чого JWT-токен зберігається в localStorage для подальшого використання при API-запитах.

Головна сторінка (DashboardPage) відображає перелік усіх розгортань поточного тенанта. Для кожного розгортання відображається компонент DeploymentCard, що показує назву образу, поточний статус контейнера, посилання на піддомен у форматі slug.домен, а також кнопки для зупинки та видалення. Статус автоматично оновлюється при перезавантаженні сторінки.

Сторінка створення розгортання (NewDeploymentPage) містить форму для введення назви Docker-образу, змінних середовища у форматі KEY=VALUE (одна пара на рядок), а також необов'язкових обмежень пам'яті та CPU. Після відправки форми виконується POST-запит до API, а у разі успіху користувач перенаправляється на головну сторінку.

HTTP-клієнт реалізований через бібліотеку Axios з налаштованим базовим URL та автоматичним підставленням JWT-токена у заголовок Authorization. Адреса API та базовий домен налаштовуються через змінні середовища VITE_API_URL та VITE_BASE_DOMAIN, що дозволяє легко розгортати застосунок у різних середовищах. Окрім сторінок для PaaS-режиму, фронтенд містить дві додаткові сторінки для роботи з конструктором сайтів. Сторінка SitesPage (маршрут /sites) відображає список усіх сайтів поточного тенанта з їх статусом публікації, посиланням на live-URL та кнопками для відкриття редактора і видалення. Сторінка BuilderPage (/sites/{id}/edit) містить компонент GrapesEditor та панель керування з кнопками Save, Publish і AI Generate. При публікації відкривається WebSocket-з'єднання через React-хук useDeploymentWs, і статус у реальному часі оновлюється безпосередньо в інтерфейсі: QUEUED → BUILDING → DEPLOYING → RUNNING, або FAILED із повідомленням про помилку.

Сторінка Dashboard з переліком активних розгортань зображена на рисунку 4.

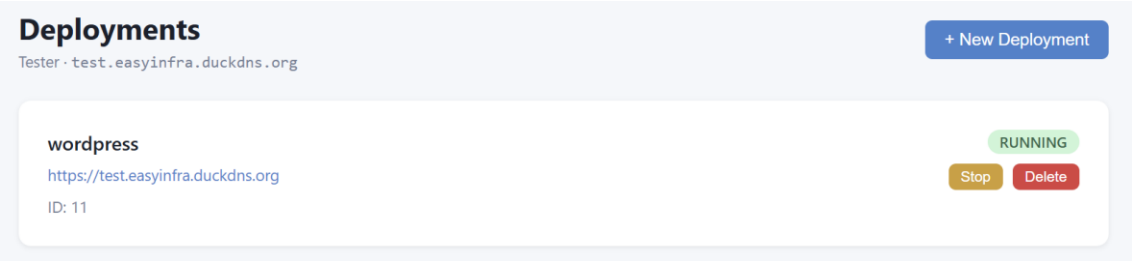


Рисунок *4 — Сторінка Dashboard з переліком розгортань*

Форма введення параметрів нового розгортання зображена на рисунку 5.

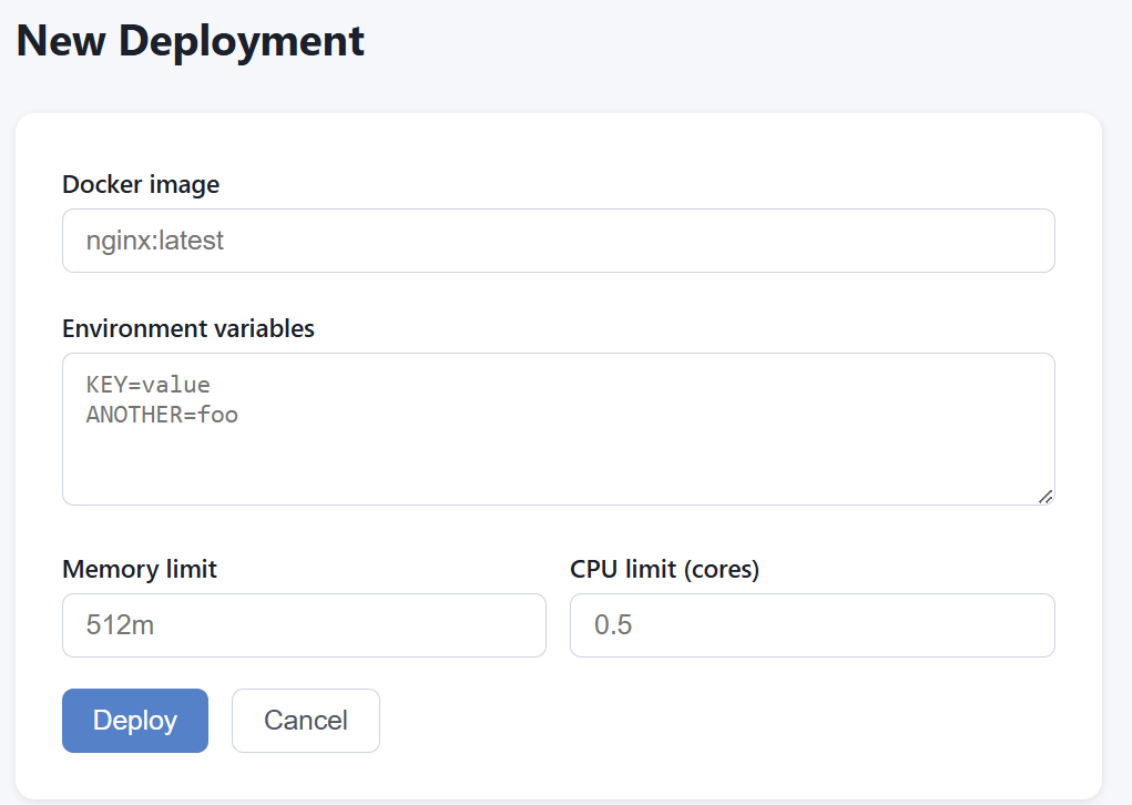


Рисунок *5* — *Форма створення нового розгортання*

3.9 Тестування системи

Тестування системи проводилось вручну у розгорнутому середовищі на базі реального сервера з доступом до мережі інтернет. Для перевірки функціональності API використовувалась вбудована документація OpenAPI/Swagger, автоматично згенерована FastAPI за адресою /docs.

Функціональне тестування охопило такі сценарії: реєстрація тенанта з унікальним slug та email; спроба повторної реєстрації з тим самим slug або email (очікуваний результат — HTTP 409 Conflict); вхід в систему та отримання JWT-токена; спроба доступу до захищених ендпоінтів без токена або з недійсним токеном (очікуваний результат — HTTP 401 Unauthorized); створення розгортання з публічним Docker-образом (nginx:alpine); перевірка доступності контейнера за піддоменом через браузер з HTTPS; зупинка та

повторний запуск розгортання; видалення розгортання та перевірка видалення контейнера через docker ps.

Під час тестування підтверджено коректну роботу всіх реалізованих функцій. Traefik автоматично підхоплював нові контейнери протягом кількох секунд після їх запуску та налаштовував маршрутизацію без перезапуску. TLS-сертифікат від Let's Encrypt успішно отримувався та застосовувався автоматично. Обмеження ресурсів контейнерів коректно передавались у Docker API та відображались у виводі docker inspect.

Тестування безпеки включало перевірку ізоляції між тенантами: тенант A не міг переглядати або видаляти розгортання тенанта B, оскільки всі запити фільтруються за tenant_id поточного автентифікованого тенанта. Також перевірено, що при введенні некоректного slug (з великими літерами, пробілами або спеціальними символами) система повертає відповідну помилку валідації. Окремо перевірено поведінку системи при нестандартних ситуаціях: запит на видалення вже зупиненого контейнера повертає HTTP 204 без помилки, а виклик sync_status для відсутнього контейнера коректно оновлює статус до значення «unknown». Перевірено, що після видалення розгортання через API контейнер справді зникає з виводу docker ps, а Traefik автоматично прибирає відповідний роутер зі своєї конфігурації протягом кількох секунд. Тестування конструктора сайтів охоплювало повний цикл: створення сайту з унікальним slug, редагування у GrapesJS, збереження через PUT-ендпоінт, публікацію через POST /publish, відстеження статусу через WebSocket та перевірку доступності сайту за HTTPS-піддоменом у браузері. Перевірено коректну роботу повторної публікації: після редагування та повторного Save + Publish система зупиняла старий контейнер, перебудовувала образ і запускала новий. Тестування AI-генерації включало надсилання промптів різної складності та перевірку коректності JSON-відповіді від OpenAI; у разі недоступності API повертається HTTP 503 із відповідним повідомленням. WebSocket-тест перевіряв, що при закритті

вкладки браузера з'єднання коректно прибирається з ConnectionManager без витоку пам'яті.

Повний перелік доступних API-ендпоінтів, відображений в інтерактивній OpenAPI-документації, наведено на рисунку 6.

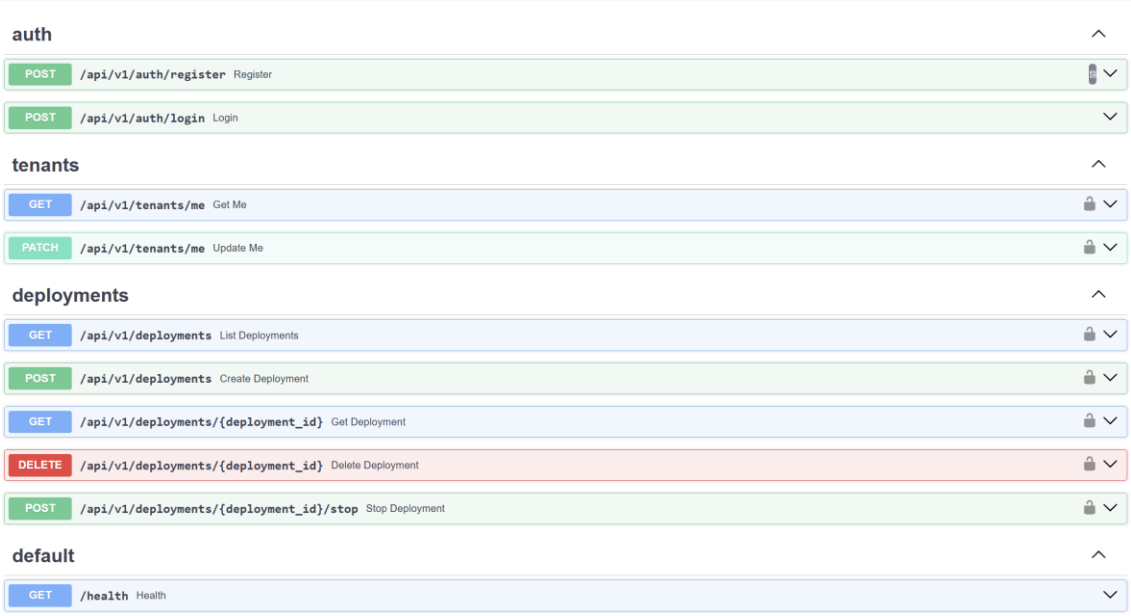


Рисунок *6 — OpenAPI документація (Swagger UI)*

Стан запущених тенантських контейнерів у терміналі показано на рисунку 7.

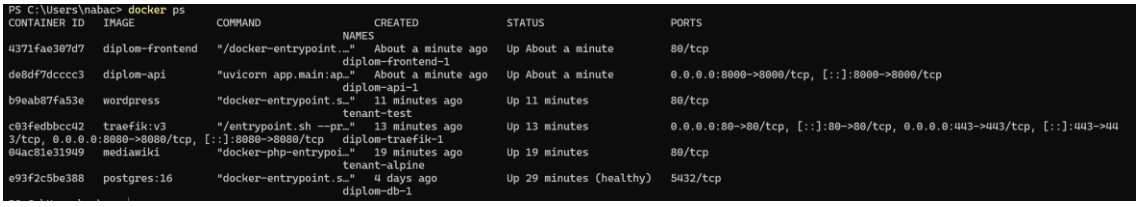


Рисунок *7 — Вивід docker ps під час роботи тенантських контейнерів*

3.10 Розгортання системи на хмарній платформі DigitalOcean

Розроблена система орієнтована на розгортання у хмарному середовищі. DigitalOcean є одним з рекомендованих провайдерів завдяки поєднанню

простоти налаштування, прийнятної вартості та наявності всіх необхідних компонентів інфраструктури: Droplets (VPS).

3.10.1 Підготовка Droplet та встановлення Docker

Для розгортання системи рекомендується Droplet з характеристиками не менше 2 vCPU / 4 GB RAM / 80 GB SSD під управлінням Ubuntu 22.04 LTS. Після створення Droplet необхідно встановити Docker Engine та Docker Compose v2 за офіційними інструкціями. Для доступу до Container Registry виконується автентифікація командою docker login registry.digitalocean.com. Образи бекенду та фронтенду публікуються до реєстру командою docker push, після чого на сервері виконується docker pull та docker compose up -d.

3.10.2 Налаштування DNS та wildcard-сертифікатів

DigitalOcean надає DNS API, яке Traefik може використовувати для DNS-01 challenge — методу автоматичного отримання wildcard TLS-сертифікатів від Let’s Encrypt без необхідності відкривати порт 80. Для цього в конфігурації Traefik вказується провайдер digitalocean та передається токен DO_AUTH_TOKEN. У зоні DNS домену на DigitalOcean реєструється A-запис із IP-адресою Droplet та wildcard CNAME *.yourdomain.com -> yourdomain.com. Після запуску Traefik автоматично отримує сертифікат *.yourdomain.com, що покриває всі піддомени тенантів без додаткового налаштування. Порівняно з DuckDNS (що застосовувався у навчальному середовищі), власний домен із DigitalOcean DNS забезпечує стабільніший і надійніший сервіс, придатний для виробничого застосування.

3.11 Висновки з розділу 3

У третьому розділі виконано повну розробку конструктора сайтів з підтримкою мультитенантності. Сформовано перелік функціональних та нефункціональних вимог до системи. Спроєктовано архітектуру на основі клієнт-серверної моделі з чітким поділом відповідальності між компонентами.

Бекенд-сервіс реалізовано на Python та FastAPI з дотриманням принципу CQRS: команди (CreateDeploymentCommand, StopDeploymentCommand, CreateSiteCommand, PublishSiteCommand тощо) виконують мутацію стану через сервісний шар, тоді як запити звертаються до репозиторіїв безпосередньо. Розроблено модуль конструктора SPA-сайтів на основі GrapesJS: збереження вмісту у базі даних (JSONB для projectData, Text для html/css), фоновий build Docker-образу з nginx:alpine, WebSocket-повідомлення про зміну статусу розгортання без перезавантаження сторінки та інтеграція з OpenAI API для AI-генерації HTML/CSS-секцій за текстовим описом. Схема бази даних версіонована через Alembic-міграції.

Розроблено модуль управління Docker-контейнерами на основі Docker SDK для Python з повністю асинхронним виконанням блокуючих викликів через asyncio.to_thread. Реалізовано підтримку ресурсних лімітів, автоматичного перезапуску та синхронізації статусу з реальним станом Docker-демона.

Налаштовано інтеграцію з Traefik v3 через динамічне генерування Docker-міток, що забезпечує автоматичну маршрутизацію HTTPS-трафіку до контейнерів тенантів без перезапуску проксі. Для DNS-маршрутизації використано сервіс DuckDNS як простий і доступний спосіб налаштування wildcard-записів. Розроблено фронтенд-застосунок на React та Vite з повним покриттям основних сценаріїв використання системи. Проведене тестування підтвердило коректну роботу всіх функціональних вимог.

ВИСНОВКИ

1. У процесі виконання кваліфікаційної роботи розроблено програмно-інфраструктурне рішення — конструктор сайтів, що забезпечує автоматизоване розгортання Docker-контейнерів з динамічним призначенням піддоменів та підтримкою мультитенантної архітектури. За результатами проведеної роботи можна зробити наступні висновки.

2. Перший розділ охоплює аналіз технологій контейнеризації (Docker, namespaces, cgroups), підходів до мультитенантної ізоляції та механізмів динамічної маршрутизації через Traefik v3. Обґрунтовано, чому підхід з окремим контейнером для кожного тенанта є оптимальним з точки зору ізоляції, гнучкості та простоти управління порівняно з рівнями ізоляції на рівні схеми БД або процесу.

3. У другому розділі показано, що жодна з розглянутих платформ (Heroku, Render, Coolify, Caprover) не задовольняє одночасно трьом ключовим вимогам: довільний Docker-образ, програмне API управління та автоматичне призначення піддоменів. Це підтвердило доцільність розробки власного рішення на обраному стеку.

4. Розроблена система вирішує виявлену в ході аналізу проблему — забезпечує одночасно чотири властивості: повну ізоляцію між тенантами на рівні контейнерів, автоматизоване розгортання через програмний API, динамічне призначення піддоменів без ручного редагування DNS-конфігурації та вбудований візуальний конструктор SPA-сайтів.

5. Завдяки інтеграції GrapesJS, OpenAI та WebSocket система покриває потреби широкої цільової аудиторії: від малого бізнесу без штатних програмістів, якому достатньо AI-генерованої посадкової сторінки, до технічних спеціалістів, які розгортають складні веб-застосунки через PaaS-режим з Docker Hub.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Docker Inc. Docker Documentation. URL: https://docs.docker.com (дата звернення: 25.04.2026).

2. Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. // Linux Journal. 2014. № 239. С. 2.

3. Traefik Labs. Traefik v3 Documentation. URL: https://doc.traefik.io/traefik (дата звернення: 25.04.2026).

4. FastAPI Documentation. URL: https://fastapi.tiangolo.com (дата звернення: 25.04.2026).

5. SQLAlchemy Documentation URL: https://docs.sqlalchemy.org (дата звернення: 25.04.2026).

6. Let's Encrypt. ACME Protocol Documentation URL: https://letsencrypt.org/docs (дата звернення: 25.04.2026).

7. DuckDNS Documentation. URL: https://www.duckdns.org/spec.jsp (дата звернення: 25.04.2026).

8. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2002. 533 с.

9. Young G. CQRS Documents URL: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf (дата звернення: 25.04.2026).

10. Percival H. Architecture Patterns with Python. Boston : O'Reilly Media, 2020. 280 с.

11. RFC 1123. Requirements for Internet Hosts URL: https://tools.ietf.org/html/rfc1123 (дата звернення: 30.04.2026).

12. PostgreSQL Documentation URL: https://www.postgresql.org/docs (дата звернення: 30.04.2026).

13. React Documentation URL: https://react.dev (дата звернення: 30.04.2026).

14. Vite Documentation URL: https://vite.dev (дата звернення: 30.04.2026).

15. Docker Python SDK Documentation URL: https://docker-py.readthedocs.io (дата звернення: 09.05.2026).

16. Alembic Documentation URL: https://alembic.sqlalchemy.org (дата звернення: 09.05.2026).

17. Jones M. T. JSON Web Tokens RFC 7519. Internet Engineering Task Force (IETF), 2015.

18. Provos N. A Future-Adaptable Password Scheme. Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, 1999. С. 81–91.

19. Nginx Inc. Nginx Documentation URL: https://nginx.org/en/docs (дата звернення: 09.05.2026).

20. GrapesJS. Documentation URL: https://grapesjs.com/docs (дата звернення: 09.05.2026).

21. OpenAI. Chat Completions API Documentation URL: https://platform.openai.com/docs/guides/chat (дата звернення: 09.05.2026).

22. Fette I. The WebSocket Protocol RFC 6455. Internet Engineering Task Force (IETF), 2011.

23. MDN Web Docs. WebSocket API URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket (дата звернення: 09.05.2026).

24. Вруцький М. Розробка конструктора сайтів з використанням контейнеризації та підтримкою мультитенантної архітектури. *Збірник наукових праць студентів, здобувачів PhD, докторантів і молодих вчених, університетська конференція «МОЛОДА НАУКА-2026» Том III «Секції Інженерного навчально-наукового інституту ім. Ю.М. Потебні Запорізького національного університету»* : м. Запоріжжя, 20–24 квітня 2026 р. Запоріжжя, 2026. С. 86–87.

Декларація

академічної доброчесності

здобувача ступеня вищої освіти ЗНУ

Я, Вруцький Максим Вячеславович студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ, спеціальність 121 Інженерія програмного забезпечення освітньої програми “Програмне забезпечення систем”, адреса електронної пошти se1212-03@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «Розробка конструктора сайтів з використанням контейнеризації та підтримкою мультитенантної архітектури» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;
- згоден на перевірку моєї роботи на відповідність критеріям академічної добро чесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 10.06.2026 _____ Вруцький М. В.

(підпис) (прізвище та ініціали) (Студент)

Дата 10.06.2026 _____ Жуков О. О.

(підпис) (прізвище та ініціали) (Керівник)