МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ
ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ

ФАКУЛЬТЕТ ФІНАНСІВ ТА ЦИФРОВИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: Створення інтерактивних платформ для розробки динамічних вебсайтів

Здобувача вищої освіти першого (бакалаврського) рівня,
IV курсу, групи ЦДБ-21-1, спеціальності 122 «Комп'ютерні науки»,
освітня програма «Технології цифрового дизайну»

_____ Потапенко Єгора Федоровича
(підпис) (прізвище, ім'я, по батькові)

Науковий керівник _____ Ніжегородцев Владислав Олександрович, к.пед.н. доцент
(підпис) (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Керівник освітньо-професійної програми _____ Ратушняк Тетяна Володимирівна, к.ф.-м.н., доцент
(підпис) (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту за рішенням кафедри комп'ютерних та інформаційних технологій і систем протокол від «13» травня 2025 №18

Завідувач кафедри _____ Філоненко Михайло Миколайович, к.ф.-м.н., доцент
(підпис) (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Ірпінь – 2025

Реєстрація №81 29.05.2025 _____
(дата) (підпис відповідальної особи)

Результати перевірки до захисту 29.05.2025 _____
(до захисту, на доопрацювання) (дата) (підпис)

Результат захисту 4 (добре) 10.06.2025 _____
(оцінка) (дата) (підпис)

Голова комісії: _____ В.М. Франчук
(підпис)

Члени комісії: _____ М.М. Філоненко
(підпис)

_____ Т.В. Ратушняк
(підпис)

_____ В.О. Ніжегородцев
(підпис)

БЛАНК ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Об'єкти контролю	Діапазон оцінки в балах	Фактична кількість балів	Примітки	Дата, П.І.Б.: керівника, голови комісії членів комісії,секретаря, (підписи)
ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ КОМІСІЄЮ ПО ЗАХИСТУ				
1. Зміст випускної роботи (якість теоретичного матеріалу, його зв'язок з практичним, вміння представляти матеріал в логічній послідовності, наявність висновків та пропозицій)	**від 0 до 30**	20		Ніжегородцев В. О. Франчук В.М. Філоненко М.М. Ратушняк Т.В. Суховецька Л.П.
2. Дотримання вимог щодо оформлення	**від 0 до 10**	7		
3. Вчасне виконання роботи на всіх етапах у встановлені кафедрою терміни	**від 0 до 5**	3		
4. Рецензент	**від 0 до 5**	4		
РАЗОМ БАЛІВ	**від 0 до 50**	34		

Оцінювання кваліфікаційної роботи науковим керівником - 36 балів.

ПЕРЕВЕДЕННЯ ОЦІНКИ ЗА 100-БАЛЬНОЮ ШКАЛОЮ ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалоюЄКТС	Оцінка за національною шкалою	Дата	П.І.Б.: голови, членів комісії, секретаря, (підпис)
70	C	4 (добре)	10.06.2025	Франчук В.М. Філоненко М.М. Ратушняк Т.В. Ніжегородцев В.О. Суховецька Л.П.

ПРАВИЛА ПЕРЕВЕДЕННЯ ОЦІНОК ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою
90-100	A	5 (відмінно)
80-89	B	4 (добре)
70-79	C	
60-69	D	3 (задовільно)
50-59	E	
35-49	FX	2 (незадовільно)

Факультет фінансів та цифрових технологій
Кафедра комп'ютерних та інформаційних технологій і систем
Галузь знань: 12 «Інформаційні технології»
Спеціальність: 122 «Комп'ютерні науки»
Освітня програма: «Технології цифрового дизайну»

Затверджую
Завідувач кафедри Філоненко М.М.
«14» листопада 2024

Завдання на кваліфікаційну роботу
здобувачу вищої освіти
Потапенку Єгору Федоровичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи : Створення інтерактивних платформ для розробки динамічних вебсайтів
   Затверджена наказом ректора від «13» листопада 2024 № 1717
2. Строк здачі закінченої роботи «12 » травня 2025
3. Мета роботи Розробити та експериментально перевірити low-code платформу що через веб-інтерфейс дозволяє, описувати структуру API; автоматично генерувати вихідний код (Go); розгортати згенеровані сервіси у контейнерах Docker; керувати їх запуском і моніторингом
   Об'єкт дослідження: Процес створення та деплою веб-API в умовах обмеженого часу й ресурсів.
   Предмет дослідження: Архітектура, алгоритми та інструменти low-code платформи «ApiProject» для генерації коду API й автоматизованого розгортання мікросервісів.
   Орієнтовна база дослідження: Локальний кластер Docker / Docker Compose, тестові сервери ДПУ, репозиторій GitHub зі зразками проектів.
4. Вихідні дані до роботи: Методичні вимоги кафедри; Огляди low/no-code платформ; Наукові статті з теми мікросервісів та DevOps (2020–2025 рр.).
5. Перелік графічного матеріалу: UI-макет конструктора API; Скриншоти роботи веб-інтерфейсу й консолі деплою
6. Зміст кваліфікаційної роботи (перелік питань за кожним розділом)
   Розділ 1. Аналітично-проєктна частина розробки Динамічних веб-сайтів 1.1Огляд проблеми та актуальність розробки динамічнгих веб-сайтів 1.2Архітектура low-code платформи 1.3 Опис мікросервісів та їх взаємодії 1.4Вибір технологій та інструментів Розділ 2. Реалізація архітектури та ключових компонентів платформи для створення динамічних веб-сайтів 2.1Реалізація веб-інтерфейсу 2.2 Додаткові деталі реалізації фронтенду 2.3Реалізація API-шлюзу 2.4 Реалізація служби розгортання АР Розділ 3. Тестування платформи . Оцінка ефективності та порівняння з аналогами
   3.1 Мета, підходи та результати тестування 3.2 Підходи до оцінки ефективності та порівняння з існуючими рішеннями Висновки Список використаних джерел

7. Календарний план виконання роботи

№ п/п	Назва етапів кваліфікаційноїроботи	Строк виконання етапів роботи	
		За планом	Фактично
1	2	3	4
1	Вибір теми, підбір літературних джерел, формування структури кваліфікаційної роботи	До 30.10.2024	До 30.10.2024
2	**І розділ (** якість теоретичного матеріалу, його зв'язок з практичним, вміння представляти матеріал в логічній послідовності, наявність висновків та пропозицій)	25.12.2024	25.12.2024
3	**ІІ розділ** (аналіз експериментальної частини про методики досліджень)	15.02.2025	15.02.2025
4	**ІІІ розділ** (проведені теоретичні і (або) експериментальні дослідження та проектні рішення)	14.04.2025	14.04.2025
5	**Висновки**	20.05.2025	22.05.2025
6	Отримання відгуку керівника	03.06.2025	03.06.2025
7	Отримання зовнішньої рецензії	05.06.2025	05.06.2025

8. Дата видачі завдання «14» листопада 2024

9. Керівник кваліфікаційної роботи _____ (підпис) Ніжегородцев В. (прізвище, ініціали)

10. Керівник освітньо-професійної програми _____ (підпис) Ратушняк Т.В. (прізвище, ініціали)

11. Завдання до виконання отримав здобувач вищої освіти _____ (підпис) Потапенко Є. Ф. (прізвище, ініціали)

Анотація

У кваліфікаційній роботі розглядається розробка інтерактивної low-code системи для генерації API-сервісів і автоматизованого їх розгортання у вигляді мікросервісів. Основною метою проєкту є прискорення створення backend-рішень для динамічних вебсайтів без необхідності ручного написання значного обсягу коду. Система дозволяє користувачу через веб-інтерфейс описати структуру майбутнього API, згенерувати код мовою Go, автоматично упакувати його у контейнер Docker, надіслати в хмарне сховище, ініціювати розгортання на сервері та отримати повноцінно працюючий сервіс.

Архітектура рішення побудована на мікросервісній моделі, де кожен компонент виконує окрему роль: веб-інтерфейс на базі React і Astro відповідає за взаємодію з користувачем; API-шлюз реалізує обробку запитів, генерацію коду та безпеку; окремий сервіс відповідає за деплой через gRPC і Docker. Для координації між модулями використовується Kafka, а для зберігання проєктів і артефактів – PostgreSQL і S3-сумісне сховище.

У роботі докладно описано реалізацію платформи, починаючи від концепції і закінчуючи тестуванням у симульованому середовищі. Проаналізовано існуючі low-code інструменти, наведено порівняльну характеристику з аналогами, окреслено переваги застосованих технологій. Система орієнтована на відкритість коду, легкість розширення, підтримку багатомовності інтерфейсу та можливість кастомізації шаблонів API.

Запропоноване рішення має практичну цінність для розробників, які працюють у галузі створення backend-сервісів, а також може бути корисним для викладачів, студентів та фахівців у сфері цифрових технологій, яким потрібна платформа для швидкого створення прототипів і розгортання мікросервісів з мінімальними витратами.

Кваліфіційна робота містить 47 сторінок, 1 таблиця, 10 риснуків, список використаних джерел з 40 найменувань, 0 додатків.

Ключові слова: API, low-code, мікросервіси, автоматизація, генерація коду, Go, Docker, Kafka, вебінтерфейс.

Annotation

This qualification paper explores the development of an interactive low-code system for generating API services and deploying them automatically as microservices. The primary goal of the project is to accelerate backend development for dynamic websites without the need for extensive manual coding. The system enables users, via a web interface, to define the structure of the intended API, generate source code in Go, automatically package it into a Docker container, upload it to cloud storage, trigger deployment on a server, and receive a fully functioning service.

The architecture of the solution is based on a microservice model, where each component performs a specific function: the React and Astro-based web interface handles user interaction; the API gateway manages request handling, code generation, and security; a separate deployment service utilizes gRPC and Docker for launching services. Kafka is used for communication between modules, while PostgreSQL and an S3-compatible storage are used to store projects and artifacts.

This paper provides a detailed description of the system's implementation, from conceptualization to testing in a simulated environment. It includes an analysis of existing low-code tools, a comparative evaluation with analogs, and highlights the advantages of the selected technologies. The system emphasizes open-source code, ease of extensibility, multilingual interface support, and customizability of API templates.

The proposed solution has practical value for developers working in backend service development, and may also be beneficial for educators, students, and IT professionals who require a platform for rapidly prototyping and deploying microservices with minimal effort.

The qualification work contains 47 pages, 1 table, 10 figures, a list of 40 references, 0 appendices.

Keywords: API, low-code, microservices, automation, code generation, Go, Docker, Kafka, web interface.

ЗМІСТ

ВСТУП 3
РОЗДІЛ 1. АНАЛІТИЧНО-ПРОЄКТНА ЧАСТИНА РОЗРОБКИ ДИНАМІЧНИХ ВЕБ-САЙТІВ 6
1.1 Огляд проблеми та актуальність розробки динамічнгих веб-сайтів 6
1.2 Архітектура low-code платформи 8
1.3 Опис мікросервісів та їх взаємодії 12
1.4 Вибір технологій та інструментів 22
РОЗДІЛ 2. РЕАЛІЗАЦІЯ АРХІТЕКТУРИ ТА КЛЮЧОВИХ КОМПОНЕНТІВ ПЛАТФОРМИ ДЛЯ СТВОРЕННЯ ДИНАМІЧНИХ ВЕБ-САЙТІВ 26
2.1 Реалізація веб-інтерфейсу 26
2.2 Додаткові деталі реалізації фронтенду: 28
2.3 Реалізація API-шлюзу 30
2.4 Реалізація служби розгортання API 34
РОЗДІЛ 3. ТЕСТУВАННЯ ПЛАТФОРМИ . ОЦІНКА ЕФЕКТИВНОСТІ ТА ПОРІВНЯННЯ З АНАЛОГАМИ 37
3.1 Мета, підходи та результати тестування 37
3.2 Підходи до оцінки ефективності та порівняння з існуючими рішеннями 39
ВИСНОВКИ 43
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 46

ВСТУП

У двадцять першому столітті зростають вимоги до створення різних додатків, які будуть повністю виконувати свою функцію і при цьому автоматично швидко працювати. Тому бізнес-потреби будуть вимагати швидкості впровадження нових функцій і розробки продуктів, тому це буде ставити певні виклики для різних розробників, а саме створювати бекенд-рішення для додатків у найкоротші терміни, які буде ставити замовник.

Взагалі самі класичні методи використання content-API (дизайн, ручне написання коду, налагодження та розгортання) виходять дуже тривалими і певною мірою складними, що буде не дуже добре. Саме тому, існує термінова потреба в інструментах, які зможуть адаптувати та полегшити створення нових API, або оптимізують роботу програмістів для рутинних дій та зменшать кількість помилок, які вони можуть допустити.

Актуальність теми. На сьогоднішній час, у світі існують платформи без кодів або з наявністю низького коду, на яких програмні системи можуть бути розроблені з мінімальним вкладенням програмування. Тому застосування цих підходів в контексті генерації API буде визначатися як новий напрямок, який має за мету значно скоротити час розробки веб-сервісів. ApiProject" — це справжня знахідка для тих, хто хоче швидко втілити свої ідеї в життя, не витрачаючи час на кодування. Можна створювати API буквально в інтерактивному режимі, а потім просто натиснути одну кнопку, щоб запустити їх як повноцінні динамічні веб-сервіси. Це ідеально підходить, саме як для великих команд розробників, так і для фрілансерів, які працюються в домашніх умовах, адже саме це дозволяє миттєво тестувати нові концепції, створювати прототипи серверних додатків та розгортати їх на ходу.

Мета роботи: розробка інтерактивної платформи з низьким кодом ApiProject для динамічного створення API, розгортання та управління ним у віддалених серверних середовищах.

У межах цієї кваліфікаційної роботи я поставив перед собою такі задачі:

– Проаналізувати цю предметну область та виявити існуючі рішення-аналогів, і також обґрунтувати актуальність розробки власної платформи;

– Спроєктувати архітектуру системи платформи на основі мікросервісів (виділити веб-інтерфейс, API-шлюз, сервіс розгортання та інші компоненти, визначити їхні фунції та взаємодію);

– Обрати сучасні технології та засоби реалізації (мови програмування, фреймворки, системи керування даними), що найкраще відповідають вимогам до платформи;

– Реалізувати основні модулі системи: front-end частину (веб-додаток), back-end частину (API-шлюз) та сервіс для автоматизованого розгортання згенерованих API;

– Інтегрувати компоненти між собою, налаштувати обмін повідомленнями та даними (HTTP, gRPC, брокер повідомлень Kafka, база даних тощо);

– Виконати тестування розробленої системи, перевірити виконання функціональних вимог (генерація і запуск API, робота мікросервісів, тощо) та надійність роботи платформи;

– Оцінити ефективність рішення, порівняти можливості платформи з існуючими аналогами, виявити переваги і недоліки.

Об’єкт дослідження : процес створення, конфігурації та розгортання веб-API у середовищі розробки динамічних вебсайтів з використанням підходів low-code та мікросервісної архітектури.

*Предмет дослідження***:** створення та публікація веб-API в умовах обмеженого часу та ресурсів, які називаються API-продукуванням, зокрема як його можна автоматизувати.

Мета дослідження: розробити інтерактивну low-code платформу для створення динамічних вебсайтів, яка забезпечує швидке формування та розгортання API-сервісів у вигляді мікросервісів.

Методи дослідження

У роботі застосовано аналіз і синтез для вивчення аналогів та формулювання вимог, моделювання — для проєктування архітектури, системний підхід — для комплексної реалізації платформи, а також методи проєктування ПЗ, емпіричні методи (тестування, спостереження) та порівняльний аналіз для оцінки ефективності розробленого рішення.

Новизна та практична значущість. Структура цієї платформи з'єднує в собі самі сучасні технології (мікросервісна архітектура, контейнеризація, технологія gRPC та брокер Kafka), які забезпечують рішення з високою гнучкістю і масштабованістю. У випадку саме ApiProject це меншою мірою буде стосуватися приєднання до бульше саме комерційних сервісів без програмування або ж з малим його обсягом коду, таких як наприклад Mendix, OutSystems і Zoho. Мова йде передусім про створення програмного коду, який інші розробники можуть вільно адаптувати, вдосконалювати або інтегрувати у власні проекти. Я створюю відкритий код на Go, тож кожен охочий може його використовувати. Практичні переваги будуть очевидні: ця платформа підходить для малих і середніх IT-компаній, а також для фрілансерів, які працюють над створенням прототипів API чи рішень у сфері IoT. Причому це можливо з мінімальними витратами — як у навчальних цілях, так і для комерційного застосування.

Базою дослідження виступає локальне середовище розробки з використанням Docker/Docker Compose, а також тестові сервери Державного податкового університету. Для перевірки роботи платформи залучалися відкриті репозиторії GitHub із прикладами API-проєктів, а також інформаційні ресурси, що відображають архітектуру й процеси low-code/DevOps у сучасних IT-компаніях. Частково було враховано структуру IT-інфраструктур міністерств, зокрема досвід використання мікросервісної архітектури в державному секторі та комерційних підприємствах.

РОЗДІЛ 1. АНАЛІТИЧНО-ПРОЄКТНА ЧАСТИНА РОЗРОБКИ ДИНАМІЧНИХ ВЕБ-САЙТІВ

1.1 Огляд проблеми та актуальність розробки динамічнгих веб-сайтів

Сьогоднішній стрімкий розвиток інформаційної інфраструктури суттєво сприяє появі нових веб-служб і застосунків. У цьому ключі процесу роль серверна частина — саме вона відповідає логіці обробки даних, а також взаємодії з базами

Традиційний підхід до створення API часто пов'язаний із написанням великої кількості рутинного коду: маршрутизація запитів, реалізація CRUD-операцій, конфігурація автентифікації тощо. І навіть найновіші фреймворки не позбавляють розробника необхідності повторювати ці дії для кожного нового проєкту. Регулярне виконання одних і тих самих завдань у розробці API вказує на очевидну можливість автоматизації. Саме тут на сцені виходять платформи з низьким кодом, які можуть значно скоротити обсяг ручного програмування. Завдяки зручним графічним інтерфейсам і генерації коду "за налаштуваннями", вони дають можливість створювати складні серверні рішення з мінімальними розмірами.

Хоча у сфері фронтенд-розробки вже є потужні сайто-будівники, ефективні інструменти для бекенда та генерації API все ще залишаться рідкою. А розробникам використовують більше потрібний інструмент, який дозволяє швидко й гнучко завантажити структуру даних, кінцеві точки API та ключову бізнес-логіку — і отримати готовий до розгортки Це питання стає ще більш актуальним при переході до мікросервісної архітектури, де кожна функціональна одиниця системи працює незалежно й має власний API. Зростання кількості таких послуг повністю до суттєвого збільшення витрат на їх розробку й підтримку, що тільки підкреслює використання автоматизованих рішень

Платформу з мінімальним обсягом коду, яка автоматично генерує стандартний API-код. Це не лише спрощує створення сервісів, а й сприяє

уніфікації підходів до розробки та значно знижує ризик помилок, пов’язаних із людським фактором.

Саме таку мету має ApiProject — інтерактивна платформа, покликана зробити процес створення серверної частини застосунку швидшим, простішим і доступнішим. Вона пропонує:

– Значне скорочення часу на реалізацію API завдяки автоматизації типових завдань;

– Генерацію коду на основі перевірених шаблонів, що зменшує ймовірність помилок у ручному коді;

– Можливість створювати API навіть тим, хто не має глибокого технічного досвіду, але добре розуміється на предметній області, завдяки зручному інтерфейсу.

Особливістю ApiProject є орієнтація виключно на мову програмування Go. Go заслужено цінується серед розробників за свою простоту, ефективність та можливість створювати легкі, конкурентні серверні додатки, які легко розгортати в контейнерах. Окремої уваги буде заслуговувати інтеграція з динамічними веб-сайтами. У нашому рішенні бекенд API не просто витягує та керує даними — він виступає повноцінним рушієм, що забезпечує підтримку життєвого циклу вебсервісу. Тому можливість миттєвого розгортання, відстеження стану, перезапуску і моніторингу API — це невід’ємна складова платформи.

Таким чином, ApiProject — це не просто генератор коду. Це повноцінне середовище, що дозволяє пройти шлях від ідеї до розгорнутого, працездатного рішення без зайвої метушні.

У сучасному IT-середовищі, де темп розробки диктує успіх, платформи, які дозволяють швидко створювати API без зайвого кодування, стають необхідністю. ApiProject виникла як відповідь на реальні виклики — зменшення часу на розробку, усунення рутини та підвищення ефективності.

Наша ціль — надати інструмент, що допоможе розробникам, особливо спільноті Swagger.io, значно підвищити продуктивність. Подібно до того, як

Docker і мікросервісна архітектура змінили обличчя DevOps, ApiProject покликана трансформувати підхід до проєктування API — із фокусом на простоті, швидкості й якості.

Тому було проведно глибокий аналіз ситуації та дійшли висновку: проблема не лише у складності побудови API, а й у тривалості та надмірній повторюваності процесів. Було розглянуто низку популярних інструментів — DreamFactory, Xano, Canonic, Backendless та інші. Проте жоден з них не забезпечує бажаного рівня відкритості, гнучкості та технічної відповідності для широкого кола розробників. Це й стало поштовхом до створення ApiProject — платформи, яка поєднує найкращі практики автоматизації, простоту low-code підходу та силу мови Go.

На основі отриманих висновків ми сформували ключові вимоги до майбутньої системи:

– Можливість швидкого створення API через веб-інтерфейс — за допомогою зручної конфігурації даних і логіки;

– Генерація чистого, ефективного й сучасного коду мовою Go з акцентом на якість;

– Автоматизоване розгортання сервісів у контейнерах, готових до використання;

– Інструменти управління життєвим циклом API — запуск, зупинка, моніторинг;

– Відкритий код платформи з можливістю розширення та адаптації під потреби спільноти — включно з підтримкою власних шаблонів.

Саме ці технічні вимоги стали основою для розробки архітектури ApiProject, яку ми докладно розглянемо у наступних розділах.

1.2 Архітектура low-code платформи

Платформа ApiProject — це платформа яка виконана саме на основі мікросервісів. Це означає, що система розділена на кілька незалежних сервісів

(модулів), кожен з яких відповідає за чітко визначену підмножину функцій. Ми вибрали архітектуру мікросервісів з таких причин:

– кожен сервіс може масштабуватися або еластично справлятися з більшим середнім навантаженням; він може бути оновлений або замінений незалежно;

– якщо певний компонент виходить з ладу, ви не хочете, щоб вся система стала "недоступною" (при умові правильного оброблення помилок інші сервіси повинні продовжувати працювати);

– команді легше розробляти та підтримувати багато малих сервісів, ніж моноліт, особливо коли кожен з них написано іншою мовою (TypeScript для фронтенду і Go для бекенду).

Платформа ApiProject побудована на принципах мікросервісної архітектури, де кожен компонент виконує чітко визначену роль і взаємодіє з іншими через стандартизовані інтерфейси. Основу системи складають такі ключові елементи:

Це перше, що бачить користувач — інтуїтивний, зручний та гнучкий односторінковий застосунок (SPA), розроблений для взаємодії через браузер. Тут доступні: конструктор API із візуальним редактором, панель керування сервісами, реєстрація, вхід у систему та інші інструменти. Вся взаємодія з бекендом відбувається через HTTPS-запити, що гарантує захищену передачу даних.

Цей компонент виступає єдиною точкою входу для клієнтських запитів. Він обробляє аутентифікацію, авторизацію, делегує запити до відповідних мікросервісів або самостійно виконує функції (наприклад, генерацію коду). По суті, API-шлюз абстрагує комунікацію між фронтендом і бекендом, приховуючи складність внутрішньої логіки платформи.

Служба розгортання (Deployment Service), як тільки API-шлюз завершив генерацію нового сервісу, справа переходить до служби розгортання. Вона приймає повідомлення про проєкт, розгортає необхідну інфраструктуру, ініціює Docker-контейнери, налаштовує середовище та контролює життєвий цикл

запущених API. Користувач має змогу керувати цим процесом через інтерфейс (зупинка, запуск, журналювання).

Окрім трьох основних мікросервісів, система спирається на важливі інфраструктурні компоненти:

– Централізована база даних (PostgreSQL) - вона виконує роль ядра зберігання — зберігає акаунти, хеші паролів, мета-інформацію про проєкти, історію розгортань, а також дані, необхідні для авторизації та відновлення API після перезапуску.

– Система асинхронних повідомлень Kafka - забезпечує подієвий обмін між компонентами системи. Коли створено новий API, шлюз надсилає повідомлення до топіка Kafka із деталями про проєкт (наприклад, ім’я, шлях до ZIP-архіву коду). Служба розгортання, підписана на цей топік, перехоплює подію та запускає розгортання. Завдяки цьому, компоненти не блокують один одного — платформа стає масштабованою і витривалою до затримок.

– Сховище артефактів (S3 або MinIO), замість пересилання ZIP-файлів через Kafka напряму (що обмежено розміром повідомлень), API-шлюз завантажує архів із кодом у сховище. Посилання на файл додається до Kafka-повідомлення, після чого служба розгортання самостійно завантажує і розгортає проєкт. Це забезпечує надійність, дозволяючи відновити завантаження в разі збоїв.

Таким чином, ApiProject — це не просто генератор API. Це комплексна платформа, яка автоматизує, масштабовує та спрощує весь життєвий цикл розробки та розгортання сервісів, надаючи розробникам повноцінну екосистему для роботи.

Користувач відправляє свої дані (електронну пошту, пароль) до API-шлюзу через веб. API-шлюз перевіряє інформацію з базою даних і відправляє результат назад у веб-інтерфейс – успіх чи помилка. Після успішного входу або повертається токен, або створюється сесія для подальших аутентифікованих запитів.

Користувач відкриває графічний конструктор у вебі і розповідає про API (створює інформацію, поля, кінцеві точки, логіку). Коли конфігурація заповнена, він натискає "Generate". Веб-інтерфейс відправляє конфігурацію API у форматі JSON на бекенд (API-шлюз) за HTTP-запитом. API-шлюз обробляє ці дані та відправляє їх до внутрішнього генератора коду. Генератор створює проєкт knative на Go, таким чином створюючи файли коду (моделі, контролер, файл docker, docker-compose.yml і т.д.) відповідно до отриманої структури. Потім проєкт упаковується в ZIP-архів та завантажується у сховище S3 з API-шлюзу.

Після успішного збереження створюється повідомлення про розгортання сервісу - API-шлюз відправляє його в Kafka. Повідомлення складається з ID проєкту + посилання на архів + будь-якої додаткової інформації, яка потрібна (назва сервісу тощо). Веб-інтерфейс отримує відповідь від бекенду, чи прийнято завдання на генерацію/розгортання, і може, наприклад, відобразити користувачу статус "Розгортається…".

Сервіс розширення оновлює статус проєкту (наприклад, до бази даних або надсилає повідомлення про успішне розгортання на іншу тему Kafka) після розгортання. Цю інформацію потім можна надіслати до API-шлюзу та повідомити фронтенд про готовність сервісу. Користувач у веб-інтерфейсі "бачить", що його API запущено і працює, отримує повідомлення про це та отримує, наприклад, URL для доступу до свого API.

Пізніше, коли користувач робить щось у інтерфейсі (наприклад, зупиняє сервіс, перезапускає сервіс тощо), веб-інтерфейс відправляє відповідну команду на API-шлюз, який потім викликає метод на Сервісі розгортання через gRPC, який у свою чергу робить те, що потрібно, на віддаленому сервері (наприклад, зупинити docker-контейнер). Логи для запущених API можуть надсилатися назад через Kafka або за запитом можуть бути отримані через API-шлюз від сервісу розгортання. Отже, архітектура ApiProject базується на трирівневій структурі:

– Клієнтський рівень: браузерний веб-додаток (front-end).

– Серверний рівень: API-шлюз як точка входу та управління.

– Рівень сервісів інфраструктури: окремий сервіс розгортання, сховище

коду, брокер повідомлень, СУБД, зовнішні сервери.

Такий поділ забезпечує чітку організацію – фронтенд не “знає” про внутрішню кухню розгортання, а працює лише з єдиним бекендом (шлюзом). Шлюз є достатньо “тонким” і делегує важкі завдання (генерація коду, деплой) або підсистемам, або зовнішнім бібліотекам. Сервіс розгортання ізольований від прямої взаємодії з клієнтами, що додає безпеки (лише шлюз може давати йому команди). Всі компоненти спілкуються через стандартизовані інтерфейси – HTTP/REST, gRPC, Kafka, що дозволяє потенційно замінити будь-який з них за умови підтримки тих самих API.

Варто зазначити, що в реалізації було використано шаблон “Clean Architecture” (чиста архітектура) для внутрішньої побудови сервісів (особливо API-шлюзу та сервісу розгортання). Це означає, що кожен сервіс має поділ на рівні: прикладний (бізнес-логіка), доменний (моделі, інтерфейси), інфраструктурний (бази даних, зовнішні інтеграції). Такий підхід полегшує тестування та підтримку коду – логіка не залежить напряму від деталей реалізації (наприклад, можна замінити СУБД або спосіб доставки повідомлень, не змінюючи бізнес-код).

1.3 Опис мікросервісів та їх взаємодії

У цьому розділі ми детально розглянемо один із основоположних компонентів системи — веб-інтерфейс ApiProject, його архітектурну модель, структуру та принципи побудови, які забезпечують масштабованість і гнучкість на рівні користувацького досвіду.

Клієнтська частина ApiProject реалізована як односторінковий застосунок (SPA), побудований з використанням Astro (v4), який поєднується з React (v18) та TypeScript. Вибір Astro обумовлений його унікальною архітектурою “островів” (islands architecture), що дозволяє завантажувати лише ті частини сторінки, які справді потребують взаємодії, залишаючи решту статичною. Це суттєво покращує продуктивність і зменшує навантаження на клієнтську машину. Astro не обмежує вас у використанні конкретного UI-фреймворку. Ви

можете безперешкодно поєднувати компоненти, створені на React, Vue, Svelte або навіть простому VanillaJS. Такий підхід робить розробку не лише адаптивною, а й довгостроково життєздатною. Щоб упорядкувати складну структуру інтерфейсу, ми застосували Feature-Sliced Design (FSD) — сучасний метод організації фронтенд-коду, орієнтований на розділення відповідальностей і масштабування.

FSD передбачає розбиття коду на шари:

- App (Додаток): містить загальносистемні конфігурації — маршрути, контексти, провайдери тощо.
- Pages (Сторінки): маршрути додатка, які поєднують віджети та інші компоненти у конкретні представлення інтерфейсу.
- Widgets (Віджети): складні інтерфейсні блоки, які відповідають за значну частину логіки сторінки. Наприклад, "Конструктор API" — це віджет, що містить поля, сутності, кінцеві точки й панель дій.
- Features (Функції): модулі, що реалізують конкретну дію користувача — наприклад, "авторизація", "додавання поля" або "генерація кінцевої точки".
- Entities (Сутності): доменні об’єкти, які несуть у собі як дані, так і логіку — "Користувач", "Проєкт", "API-сутність", "Endpoint".
- Shared (Загальні модулі): інструменти, утиліти, константи, повторно використовувані UI-компоненти (кнопки, поля, сповіщення тощо).

З огляду на функціональну насиченість клієнтського інтерфейсу ApiProject — включаючи графічні редактори, інтерактивні таблиці, численні форми та діалоги — структурна дисципліна, яку забезпечує FSD, є критичною. Вона дозволяє команді підтримувати логічну чистоту, легко масштабувати застосунок та ефективно виявляти й усувати проблеми.

З точки зору функціональності, веб-інтерфейс дозволяє користувачеві:

- система реєстрації та авторизація стосується реєстрації нового акаунта + форма валідації (електронна пошта, пароль) для авторизації. після входу токен (jwt) або id сесії зберігається в браузері, цей токен має бути поміщений у заголовок для запиту після входу. також є можливість відновлення пароля

(паролі можуть бути скинуті адміністратором, вигадано спеціально для проєкту; справжнє відправлення електронних листів не планується, але можливість відправлення реалізована).

– домашній кабінет користувача: увійшовши, користувач потрапляє до домашнього кабінету. тут показуються їхні поточні проєкти api (якщо такі є) - назви, статус (наприклад, "у процесі", "зупинено", "у розробці") та url-адреси доступу для запущених api. звідси можна приступати до створення нового api або керування вже створеними api (запуск/зупинка).

– графічний api трафарет: це головний елемент платформи. він задуманий як інтерактивний робочий простір, де користувач додає елементи:

Сутності, тобто це таблиця/модель даних.

Кінцеві точки API. Виберіть тип запиту (GET, POST, PUT, DELETE), прив’яжіть кінцеву точку до сутності (наприклад, ви GET /books - ви маєте повернути список книг, ви POST /books - ви маєте створити книгу). Ви отримуєте операції CRUD, що генеруються з коробки з вибором (чекбокс - додати CRUD для сутності --> розташування 4 кінцевих точок).

Дозволяє додавати прості обробники до кінцевих точок: наприклад, виконати якийсь код до збереження об’єкта або після отримання списку (хуки або інтерцептори). Тому на етапі проєктування ми вирішили дозволити користувачеві надати стандартні опції (наприклад, "перевірити токен користувача") або надати фрагмент коду на Go (у окремому вікні) - це експериментально і потребує підпису коду.

Глобальні налаштування (наприклад, URL-префікс (виставити всі кінцеві точки під /api/v1/), тип аутентифікації (JWT, сесія, базова версія постачається з JWT і простим секретом) та підключення до бази даних (наприклад, можливість вказати, що згенерований API має працювати з PostgreSQL, тепер кодові шаблони включатимуть моделі з GORM або іншої ORM)). Це те, що ми робимо в більшості випадків у нині активній реалізації, оскільки API знаходиться поверх спільної бази даних (щоб уникнути розгортання БД для кожної служби), але

технічно ми все ще можемо це зробити з точки зору архітектури так, щоб кожен API записував/читав зі своєї БД.

Конструктор GUI сам по собі є додатком з інтерфейсом перетягування, який легко керується:

- елементи можна перетягувати в зони,
- поля можна розташовувати,
- зв'язки між об'єктами встановлюються шляхом простого перетягування стрілки з однієї на іншу (реалізовано також як «полотно»).

Ця інтерактивність управляється за допомогою бібліотек React (можна використовувати більше, таких як DnD Kit для перетягування та скидки). Все, що робить користувач, змінює динамічно генеровану конфігурацію JSON, яку можна переглянути в («Перегляді JSON»).

Коли робота в конструкторі завершена і натиснута кнопка «Генерувати API», згенерований JSON буде надіслано на сервер. Періодично у процесі роботи рецензент може автоматично зберігати проміжні результати в локальне сховище браузера та/або надсилати їх на сервер (за бажанням — користувачі можуть віддати перевагу зберігати чорновик, щоб продовжити роботу пізніше).

Кожен створений API-проєкт, незалежно від того, перебуває він у стані активного виконання чи вже завершений, відображається в особистій панелі користувача. До кожного з них прикріплені функціональні кнопки: «Зупинити», «Перезапустити», «Видалити», «Завантажити код».

Усі ці дії генерують запити до API-шлюзу — де-факто, до «мозку» системи, який керує станами контейнерів та сервісами. Зокрема, кнопка завантаження коду надає ZIP-архів з повним вихідним кодом API, що ще раз підкреслює відкритість платформи: користувач може в будь-який момент забрати проєкт для локальної роботи чи розгортання поза ApiProject.

Інтерфейс системи підтримує багатомовність — щонайменше українську та англійську мови (основні для навчальних і практичних кейсів цього проєкту). Це реалізується через JSON-файли з мовними ключами. У разі ідентифікації

користувача мовні налаштування зберігаються в профілі, а для анонімних сесій — у localStorage.

Перемикання мов здійснюється миттєво, без перезавантаження сторінки. Додавання нової мови не потребує значних зусиль: достатньо створити файл перекладу й налаштувати зв'язок у мовному менеджері. Це дозволяє легко масштабувати інтерфейс для нових ринків, включно з хангуком, якщо виникне потреба.

Бекенд-серце ApiProject написане на Go 1.20 із дотриманням принципів чистої архітектури, що передбачає чітке розділення шарів: транспортного, логічного, доменного та інфраструктурного. Серед основних його модулів:

Користувачі входять у систему за допомогою JWT-токенів. Шлюз зчитує ці токени з заголовків HTTP-запитів, верифікує їх і дозволяє доступ до захищених маршрутів (/api/projects/...). Паролі хешуються (наприклад, через bcrypt) і зберігаються в базі даних. Передбачена роль адміністратора, який матиме розширений доступ до всіх проєктів (реалізація інтерфейсу ще у розробці, проте структура вже є).

Це ядро платформи, що перетворює JSON-конфігурацію API-проєкту на повноцінний Go-проєкт. Процес проходить у кілька етапів:

Валідація конфігурації — перевірка правильності назв, типів, посилань між сутностями. Побудова структури проєкту — створення тимчасової файлової системи з папками типу cmd/, internal/, deploy/ (для Dockerfile, docker-compose.yml).

Генерація моделей — наприклад, сутність Book з полями Title: string, Pages: int автоматично трансформується в Go-структуру.

Завдяки цій автоматизації ApiProject мінімізує помилки, пов’язані з ручною розробкою, та водночас забезпечує повний контроль над кінцевим результатом. Структура Book для моделі даних показана на (рис. 1.1)

```
type Book struct {
    ID    uint   `json:"id" gorm:"primaryKey"`
    Title string `json:"title"`
    Pages int    `json:"pages"`
}
```

Рис. 1.1 Генерація контролерів

Це функціональність CRUD, яка працює "з коробки". Якщо ж вирішити додати власну логіку в код (наприклад, автоматично встановити дату при збереженні книги) – це можна зробити безпосередньо у відповідному контролері. На практиці це можна було б реалізувати, додаючи додаткове поле в налаштуваннях, де користувач може ввести код, щоб він був вставлений як потрібно.

Визначення маршрутизації: забезпечення маршруту для API в main.go або файлі маршрутизації. Наприклад, для сутності Book – /books (GET, POST) /books/{id} (GET, PUT, DELETE) тощо.

Інші компоненти: додаються елементи сервісу – (наприклад, config.yaml з налаштуваннями БД) + ініціалізація бази даних (це або генерація коду створення таблиць через ORM, або генерація сценаріїв міграції).

Файли Docker: один для служби (спочатку офіційне зображення Golang для побудови, а потім Alpine для запуску такого бінарного файлу, щоб зменшити розмір пакета), і один для docker-compose.yml, який визначає, як виглядає служба, і, можливо, інші (залежні) контейнери (наприклад, контейнер Postgres, якщо вирішено відправити окрему БД тільки для цієї служби – наразі визначено, що служба взаємодіє із загальною базою даних).

Збірка проекту: якщо це потрібно, ваш згенерований код може бути скомпільований в один крок (використовуючи go build). Але це не обов'язково для нашого потоку – ми можемо просто зібрати вихідний код, а збірка буде виконана на сервері, коли буде запускатися Docker (Dockerfile подбає про це).

Проте, цікаво було б провести тест компіляції та надійності. Ось що у мене є на даний момент: вихідний код тут.

Завантаження артефактів: zip всього проекту завантажено в хмарне сховище (Amazon S3). Також створюється запис нового проекту у базі даних (ID, користувач, час створення, статус «в розгортанні», посилання на архів і т. д.).

Завдання публікації розгортання: Після успішного завершення процесу генерації коду, API шлюз відправить повідомлення, використовуючи клієнт Kafka (наприклад, бібліотека kafka-go). Повідомлення містить:

- project_id
- user_id (Можливо, для перевірки прав, хоча я не думаю, що служба розгортання дійсно хвилюється про це).
- s3_url (посилання на завантаження архіву).
- ім'я служби (це корисно для встановлення імені контейнера).
- можливо, бажаний порт (або ваш сервіс розгортання сам обере один).

API шлюз не чекатиме відповіді – він зможе миттєво повідомити клієнту, що "проект створюється".

Відстеження результатів розгортання: Щоб стежити за статусом, API шлюз може слухати іншу тему Kafka, куди служба розгортання публікує повідомлення про завершення. Наприклад, { project_id:..., статус: "розгорнуто", public_url: "http://<host>:<port>/" }. Коли це відбувається, шлюз записує це в базу даних (встановлює "Виконується" і зберігає URL). У випадку помилки під час розгортання ви отримаєте статус "помилка" та повідомлення про помилку: тоді – шлюз позначає проект як «Не вдалося розгорнути» (або видаляє його залежно від опції).

Кінцеві точки управління сервісом: коли користувач натискає, скажімо, "Зупинити" для запущеного API, фронтенд викликає /api/projects/{id}/stop. Тепер цей виклик шлюзу призводить до виклику служби розгортання. Для цього використовується gRPC – API шлюз є клієнтом gRPC до служби розгортання (тобто ми викликаємо метод, наприклад, StopProject(project_id)). Протокол gRPC використовується для його швидкої, структурованої передачі даних, яка є

безпечною та надійною (не обмежуючись випадковим доступом до деякого REST end-point, або здійсненням HTTP запитів кудись). Ви маєте сервіс protobuf, визначений з методами "Start", "Stop" і "GetStatus" та ін. Команди виконуються службою розгортання, а потім результат передається назад, і API шлюз потім відправляє результат клієнту.

Кешування та оптимізація запитів: Оскільки API-шлюз може отримувати безліч однотипних запитів (особливо для зчитування статусу або списків), відповіді кешуються на цьому рівні. Зокрема, список проєктів користувача може кешуватися на кілька секунд або до появи змін (події розгортання/зупинки оновлюють кеш). Сам шлюз кешує деталі конфігурації віддалених серверів та порти, на яких працюють сервіси, у пам'яті, щоб швидко реагувати на запити статусу (хоча зауважте, що це не завжди призводить до запиту до сервісу розгортання).

Журналювання та моніторинг: Всі запити додаються до журналу (користувач, активність, дата/час) API-шлюзу. Це корисно для налагодження і є частиною нефункціональних вимог безпеки (аудит дій). Також можуть бути колектори метрик (загальна кількість розгортань, кількість успішних/невдалих розгортань, середній час на їх створення тощо), ці дані можуть бути корисними в майбутньому для подальшого поліпшення системи.

Сервіс розгортання: Це мікросервіс, написаний на Go, для демонстрації сервісу (декларуючи відповідальність DevOps). Він тісно інтегрується з інфраструктурою контейнеризації та управління конфігурацією:

Вибір цільового сервера: Насправді на бекенді сервіс розгортання підтримує список хостів для розгортання. Отже, в нашому випадку це список, взятий, можливо, з конфігураційного файлу, можуть бути навіть записи бази даних, які містять інформацію про "сервер" та "навантаження". Можна мати локальний хост (розгортання на тому ж комп'ютері) для простоти.

Налаштування середовища Ansible: Сервіс розгортання викликає playbook Ansible (наприклад, за допомогою shell-скрипту або Ansible Runner). Використовується наступний заздалегідь заданий playbook:

Підключення по SSH до обраного вузла;

Перевірка наявності Docker та Docker Compose. Якщо Docker не встановлений, то встановлюється остання версія (для Ubuntu - за допомогою apt або інсталяційного скрипту Docker). Якщо вони присутні, переконується, що демон запущений.

Те саме стосується Docker Compose (це зараз використовує Compose v2, який включений у CLI Docker, або можна встановити окремий двійковий файл).

Створення робочого каталогу на сервері для нового сервісу (наприклад, /opt/ApiProject/projects/<project_id>).

Завантаження проєкту та його запуск: Ansible завантажує код з S3, коли середовище готове. Це можна зробити за допомогою модуля get_url або запустивши команду wget. Потім архів розпаковується (unzip або tar). У каталозі тепер є Dockerfile, docker-compose.yml та інші файли проєкту. З Docker Compose ми починаємо: викликом команди docker-compose up -d (у фоновому режимі) контейнер створюється, і створений сервіс функціонує. Docker-контейнер буде створений автоматично відповідно до Dockerfile, це займе трохи часу, від хвилин (зазвичай це займе менше 30-60 секунд; розмір базового образу Go при першому запуску/завантаженні).

Аналіз результатів: Якщо Docker Compose успішний (контейнер запущено), тоді Ansible успішно виконує playbook. Сервіс розгортання отримує сигнал успіху. Це те, на якому порту слухає сервіс. Це може бути або статичний порт (наприклад, у docker-compose.yml): з можливістю вказати порт, на якому контейнер прив'язаний зовнішньо. Наприклад, ми можемо налаштувати кожен проєкт, щоб мати порт, пов'язаний з ним, можливо, на основі project_id (умовно 8000 + id). У реальному світі ви повинні вирішити це більш досконалим способом (сервісне відкриття або динамічний проксі). Для навчання можна трохи схитрити: після того, як ви запустили контейнер, виконайте команду docker-compose port <service_name> <container_port>, щоб побачити, якому зовнішньому порту він був призначений.

Сповіщення про результати: Сервіс розгортання надсилає результат (повідомлення про успішність або невдачу) до повідомлення. Якщо це успіх, воно містить URL (наприклад, адреса сервера + порт). Це повідомлення з'являється в Kafka (тема deploy_results).

Оновлення записів: Сервіс забезпечення всередині записує, що певний сервер тепер розміщує певні ресурси як новий сервіс. Може зберігати PID контейнерів, їхні імена, щоб мати змогу зупинити їх.

Сервер gRPC: На додачу до сервісу розгортання, також ініціюється сервер gRPC, який прослуховує команди від Apigateway. Ось який сервіс у gRPC з методами:

- StartProject(ProjectID) повертає (Status) – припустимо, ми хочемо виконати проєкт, який вже завантажено (тобто в нашому процесі проєкт починає виконуватися одразу після його розгортання, тому може не бути необхідності в Start, окрім як для перезапуску, якщо зупинено).
- StopProject(ProjectID) повертає (Status) – зупиняє виконуваний сервіс. Рішення: Виконайте docker-compose stop або docker stop на віддаленій ноді. Після успіху змініть значення статусу.
- GetStatus(ProjectID) повертає (Status) – повертає статус: виконання/зупинено/незнайдено.
- RemoveProject(ProjectID) повертає (Status) – опціонально: ви можете використовувати це, щоб видалити контейнер і очистити дані (docker-compose down, видалити директорію).

Коли API GW викликає, наприклад, StopProject, сервіс розгортання знаходить сервер, на якому розміщено цей проєкт (ця інформація є у власній таблиці відповідності), так щоб можна було виконати цю дію. Це може працювати як Ansible (виконання playbook для зупинки контейнера) або безпосередньо з того самого сервера - через Docker SDK або команду Docker. У нашому випадку ми можемо уявити, що якщо сервіс розгортання працює на цільовому сервері, він може безпосередньо викликати команду docker. У протилежному випадку, знову через Ansible/SSH.

Виявлення роботи сервісу: Сервіс розгортання може опитувати стан контейнерів (за допомогою docker ps або Docker API) та збирати логи (docker logs). Якщо це запитує шлюз, він може передавати останні логи, що відображаються на інтерфейсі (наприклад, з метою налагодження). Немає регуляції чого-небудь, що вимагало б логування, у мінімальному випадку логи тільки потрібно записати у файл на сервері, і, якщо це необхідно, користувачі могли б завантажувати їх самі.

1.4 Вибір технологій та інструментів

У процесі проектування ми повинні були обґрунтувати та вибрати технологічний стек, який задовольнить потреби системи. Враховувалися продуктивність, масштабованість, надійність, обізнаність команди про стек, доступність спільноти та підтримка, а також сумісність з іншими компонентами. Програмування Go використовується для кодування бекенда. Як зазначалося раніше, Go (Golang) було обрано для розробки серверних компонентів, включаючи служби розгортання, API- шлюзи та власний API. Основні переваги використання Go для цього проекту включають підтримку конкурентоспроможності: горутини та канали можуть паралельно обробляти багато запитів одночасно без перешкод для потоку керування під час операцій входу-виводу. Наш API- шлюз дозволяє обслуговувати запити від сотень тисяч клієнтів і безперервно взаємодіяти з Kafka, базами даних тощо.

Підтпимка конкурентності: горутини та канали можуть одночасно обробляти багато запитів без перешкод для потоку керування під час операцій входу-виводу. Наш API- шлюз дозволяє обслуговувати запити від сотень тисяч клієнтів і безперервно взаємодіяти з Kafka, базами даних тощо. Простість і швидкість виконання: Go комп'ютеризується в машинний код так само швидко, як C++, але код є більш зрозумілим і чистим. Це гарантує швидкі та ефективні програми. Ваші сервіси будуть працювати швидше, якщо ви створите API. Статична типізація та безпека типів: помилки типів виявляються на етапі компіляції, а велика кількість помилок усуває керуючу пам'яттю, також

відомо як збір зміщення. Це необхідно для забезпечення стабільності нашої платформи.

Go пропонує велику кількість готових бібліотек, які я вважаю зручними для використання, таких як клієнти PostgreSQL, Kafka, веб-фреймворки (Gin/Echo), бібліотеки для роботи з Docker/SSH та інші. Для інших операційних систем Go також дуже легко компіюється в статичний бінарний файл, що є перевагою для розгортання компонентів. Go базується на ідеях розділення папок і пакетів , а також на інтерфейсах для абстракцій, які ми хочемо бачити. Node Node.js був альтернативою, після JS простий для рішень з низьким кодом . Однак Node.js перевершує Go з точки зору продуктивності та простоти розгортання , оскільки служби JS залежать від середовища виконання Node.js, тоді як Go вимагає лише одного бінарного файлу.

Для того, щоб гарантувати надійність і підтримку коду, TypeScript використовує для фронтенду замість чистого JavaScript. Статична типізація TS полегшує рефакторинг, пропускаючи окремі помилки ще під час розробки . Крім того, TypeScript стає звичайним у великих фронтендових проектах, оскільки наш конструктор API досить складний.

Одна з найпопулярніших бібліотек для розробки інтерфейсів — React. Вона пропонує декларативний метод створення компонентів інтерфейсу користувача, що полегшує керування станом додатка. Вибір також був впливом великої екосистеми, яка включає бібліотеки компонентів, утиліти та інструменти тестування. Інші фреймворки, такі як Vue, Angular і Svelte, також розглядалися, але група працювала найбільше з React. Крім того, методологія Feature-Sliced Design швидко поширилася в React-спільноті.

Astro — це сучасний фреймворк, який вже знаходиться у версії 4, який використовується для оптимізації веб-додатків. Astro дозволяє поєднувати підходи та вибирати серверний рендеринг: він може рендерити значну частину статично, а React може створювати інтерактивні « острівці » . Astro забезпечує швидке завантаження сторінок для нашої платформи, частина сторінок вже може бути розміщена в готовому HTML -коді. Це важливо для першого відкриття

програми в браузері. Крім того, він добре вписується в архітектуру з розділами на компоненти та сторінки, що робить підключення багатомовності простим. SCSS є препроцесором стилів, який використовується для спрощення дизайну інтерфейсів. Його частка, 14,6% коду SCSS, явно використовується. Можливість використання змін, вкладень та сумішей у SCSS прискорила розробку адаптивного дизайну конструктора.

База даних – PostgreSQL. Вимоги до СУБД включали надійність, відповідність реляційній моделі (щоб зручно зберігати зв'язані сутності: користувачі, проєкти, і можливо шаблони чи статистику), робота в контейнерах, сумісність із Go. PostgreSQL 15 було вибрано завдяки:

– Повної підтримки транзакцій, строгій відповідності ACID – дані про користувачів та проекти мають зберігатися надійно.

– Наявності якісного драйвера для Go (pgx) і ORM (наприклад, GORM) – це спростило реалізацію доступу до даних у API-шлюзі.

– Можливості виконувати складні запити при потребі (хоч в нашому випадку запити досить прості).

– Безкоштовності та відкритості, широкої спільноти підтримки. Альтернативою могла бути MySQL/MariaDB (подібні за можливостями), але PostgreSQL обрали через кращу сумісність з JSON (якби знадобилося зберігати JSON-конфігурації) та суб'єктивний досвід команди. Для зберігання сесій можна було використати NoSQL (Redis), але у нас для простоти сесії/токени є JWT і БД.

Docker встановлює світовий стандарт контейнеризації. Для спрощення розгортання всіх компонентів платформи можна використовувати образи Docker. Крім того, під час розробки можна запускати PostgreSQL і Kafka як контейнери. В остаточному підсумку наш AR працюватиме як контейнер Docker. Це забезпечує консистентність середовища, включаючи рутинну процедуру Go та бібліотеки та бінарний файл, якому потрібен контейнер, незалежно від того, на якому вузлі контейнер буде розгорнутий. Інший плюс Docker — це те, що він полегшує запуск сервісів (зупинка, початок, знищення).

На етапі проектування було визначено архітектурний патерн платформи ApiProject: патерн мікросервісів, який чітко розподіляє відповідальність між усіма частинами платформи. веб-інтерфейс користувача. API-шлюз (бекенд), який працює як вхідна точка, так і центр управління Сервіс розгортання, який контролює розгортання та життєвий цикл API.

Пропонування архітектурного рішення задовольняє технічні вимоги специфікації. Крім того, воно є адаптивним до майбутнього зростання. Це включає в себе інтеграцію нових мікросервісів, таких як аналітика та сервіси моніторингу, можливість окремо масштабувати кожен компонент , можливість підключення служби авторизації або зміни розгортання з Ansible на Kubernetes без значних змін в архітектурній додатку. Таким чином, проектна частина роботи заклала основу для подальшого розвитку платформи ApiProject.

РОЗДІЛ 2. РЕАЛІЗАЦІЯ АРХІТЕКТУРИ ТА КЛЮЧОВИХ КОМПОНЕНТІВ ПЛАТФОРМИ ДЛЯ СТВОРЕННЯ ДИНАМІЧНИХ ВЕБ-САЙТІВ

2.1 Реалізація веб-інтерфейсу

Тепер перейдемо до самої реалізації нашого проєкту, але спочатку розглянемо більш детально процеси реалізації самих основних компонентів платформи ApiProject, відповідно до спроєктованої архітектури проєкту. Було описано основну структуру та функціонування вихідного коду модулів, далі наведено основні фрагменти реалізації, а також саме практичні аспекти розгортання нашої платформи. Таким чином, цей розділ буде демонструвати, як теоретичні рішення попереднього етапу так і вже втілення на практиці.

На (рис. 2.1) зображено структуру вихідного коду фронтенд-частини платформи ApiProject, сформовану відповідно до методології Feature-Sliced Design.

```
/pages
  index.astro            – Головна сторінка (логін / реєстрація або редирект в кабінет)
  dashboard.astro        – Кабінет користувача (список проєктів)
  project/[id].astro     – Сторінка конкретного проєкту (конструктор API або перегляд)
/widgets
  ProjectListWidget.jsx  – Віджет списку проектів на dashboard
  EditorWidget.jsx       – Віджет конструктора API (основна робоча область)
  EntityEditor.jsx       – Підкомпонент: редактор однієї сутності
  EndpointEditor.jsx     – Підкомпонент: редактор одного ендпоїнту
  ... (інші віджети)
/features
  auth
    LoginForm.jsx, RegisterForm.jsx – компоненти форм + логіка запиту до API
  project
    CreateProjectButton.jsx        – компонент + логіка створення нового проекту
    ControlButtons.jsx             – кнопки старт/стоп/видалити проект
  ...
/entities
  User.ts         – Опис типів користувача
  Project.ts      – Опис типу проекту (структура, статус тощо)
  ApiSchema.ts    – Опис структури API (сутності, поля, ендпоїнти)
/shared
  apiClient.ts    – модуль для HTTP-запитів до бекенду (на базі fetch)
  ui
    Button.jsx, Modal.jsx, Input.jsx – базові UI-компоненти
  utils
    validators.ts, helpers.ts       – різні допоміжні функції
/styles
  globals.scss    – загальні стилі, змінні (тематика оформлення)
```

Рис. 2.1 Структура проекту

При відкритті платформи користувач повинен бачити форму входу . В Astro це може бути індексом, наприклад, Astro File, який містить компонент . Якщо у користувача ще немає облікового запису, він переходить на сторінку реєстрації через стан компонента. Компоненти входу і з папки роздільної/ui використовують в обох формах. Бібліотека Так , або ваші корисні функції можна знайти в розділі shared/utils/validators.ts для перевірки вводу.

Функція LoginForm створення функції auth після надсилання форми. Ця функція виконує POST-запит до /api/login через розділений модуля /apiClient. apiClient — це обгортка для нативного отримання, яка встановлює базові URL, заголовки JSON та інші функції.

Якщо відповідь була успішною (200 OK з токеном у тілі), то токен зберігається в локальному сховищі браузера для автоматичного включення в заголовки (Authorization: Bearer ...) при наступних викликах. Користувач потрапляє на /dashboard.

Так само, при успішному виконанні, RegisterForm перенаправляє на вхід або автоматично входить у користувача.

Це складна частина інтерфейсу. Конструктор доступний на сторінці проекту /project/{id}.

Зонування: Жива стіна розділена на кілька зон:

Ліва панель: стовпчик сутностей (наприклад, Сутність1, Сутність2 тощо). Тут можна натиснути "+ Додати сутність" - це додасть нову сутність з типовим ім'ям, яке ви можете змінити.

Центральна панель: якщо вибрано сутність, то показано редактор полів сутності. Якщо вибрано кінцеву точку, показується редактор параметрів кінцевої точки. Якщо немає вибору, ви побачите або привітання для себе, або відображення ваших взаємовідносин (що може змінюватися між таблицею та діаграмою).

Панель справоруч: індекс API кінцевих точок. Ви також можете створити новий маршрут за допомогою кнопки "+ Додати кінцеву точку".

Верхня частина (окрема від нижньої для роботи обох одночасно): опції для збереження або генерації API, селектор мови та, можливо, селектор виду (JSON/Графічний) кадр під ним.

Робота конструктора виконується засобами React, оскільки Astro в основному використовується для обгортання сторінки. EditorWidget є stateful компонентом і містить стан схеми API, тобто масив сутностей, масив кінцевих точок, вибраний елемент для редагування тощо.

Цей стан ініціалізується при завантаженні з даними, отриманими з бекенду (/api/projects/{id} GET – може отримати збережену конфігурацію API, якщо користувач її зберіг раніше, або порожню, якщо це новий проект).

Логіка збереження та генерації: коли користувач натискає кнопку "Згенерувати API", у вікні редактора відкривається функція onGenerate. Вона отримує поточний стан API-схеми, яка включає масив сутностей, ендпоінтів тощо, і передає його на бекенд за допомогою apiClient.post('/api/projects/{id}/generate', schema). У нашій реалізації ми об'єднали POST з /api/projects, що означає, що створення проекту запускає генерацію. Однак, якщо передбачено кремове редагування перед генерацією, використання /генерувати може бути кремовим викликом. Після виклику EditorWidget може блокувати інтерфейс, показуючи «Генерація...» і забороняючи редагування, щоб користувач не вніс зміни під час генерації. Фронтенд отримує повідомлення про запуск процесу, коли надходить відповідь, наприклад 202 Прийнято. Далі він перейде в режим моніторингу статусу розгортання, як стає вище. Він може отримувати push-повідомлення про успіх або невдачу або проводити опитування/статут.

2.2 Додаткові деталі реалізації фронтенду:

Для того щоб, керувати станом використовуються React Hooks (useState, useReducer для складних структур, можливо useContext для глобальних), або можна було застосувати Zustand чи Redux. У невеликому масштабі достатньо useState та передачі пропсами.

Інтерфейс намагалися зробити інтерактивним і дружнім: використано багато іконок (наприклад, значки баз даних для сутностей, піктограми методів GET/POST різного кольору), підказки при наведені пояснюють поля, помилки валідації показуються одразу червоним текстом поруч з полем.

Вся стилізація виконана в SCSS з урахуванням адаптивності: інтерфейс конструктора займає всю ширину/висоту екрану, панелі можуть згортатися на вузьких екранах. Фрагмент компонента для редагування полів сутності наведено на (рис. 2.2).

```
function FieldRow({ field, onChange, onDelete }) {
  const handleNameChange = (e) => {
    onChange({ ...field, name: e.target.value });
  };
  const handleTypeChange = (e) => {
    onChange({ ...field, type: e.target.value });
  };
  return (
    <div className="field-row">
      <input
        type="text" value={field.name}
        onChange={handleNameChange} placeholder="Назва поля"
      />
      <select value={field.type} onChange={handleTypeChange}>
        <option value="string">String</option>
        <option value="int">Integer</option>
        <option value="bool">Boolean</option>
        {/* ...інші типи */}
      </select>
      {/* ...можливо прапорці */}
      <button onClick={onDelete}>X</button>
    </div>
```

Рис. 2.2 Приклад фрагмента коду (React-компонент поля сутності)

Отже, код було спрощено, щоб показати саме основну ідею: у формі, після зміни значень, React оновлює локальний стан (визначений вашим значенням) і повідомляє батьківський компонент про такі зміни за допомогою useEffect. У виробництві більш ретельно розроблена програма може використовувати стан проекту для того, що має бути написано витратами бюджету. Однак у цьому

модифікованому рецепті все записується вживу на диск. Це означає, що ви маєте механізм автоматичного збереження, який може це робити.

Веб-інтерфейс безперешкодно з'єднаний з бекенд-сервісом через RESTful API. У процесі розробки була використана утиліта ESLint з правилами для React/TS, що служить для підтримки узгодженості формального вираження та уникнення багатьох загальних помилок. Були використані деякі прості тестові випадки модулів, щоб просто протестувати окремі компоненти React (використовуючи, наприклад, React Testing Library). Крім того, було виконано багато ручного та інтерактивного тестування, доки я не впевнився, що всі частини поводяться саме так, як я хочу, коли вони працюють разом.

В результаті була створена високоефективна веб-сторінка, що дозволяє користувачам на всіх рівнях (від початківців до тих, хто має вдосконалені знання) швидко отримувати обладнання, яке потребується через API. Вся необхідна інформація доступна на інтерфейсі, попереджає про потенційні помилки, такі як коли дві речі мають схожі назви (вони будуть позначені червоним), і керує кожним окремим кроком, необхідним для налаштування бекенд-сервісу в робоче положення.

2.3 Реалізація API-шлюзу

API-шлюз є центральним серверним компонентом платформи ApiProject. Його реалізацію виконано на Go як окремий веб-сервіс (REST API). Основні складові реалізації API-шлюзу:

- Маршрутизація HTTP-запитів (роутинг)
- Контролери (обробники) для кожного ендпоїнту
- Логіка обробки запитів (автентифікація, CRUD операції з БД)
- Модуль генерації коду (одна з найбільш об’ємних частин)
- Взаємодія з Kafka (відправка та отримання повідомлень)
- gRPC-клієнт для зв’язку з сервісом розгортання
- Взаємодія з S3 (завантаження архіву проекту)
- Моделі та доступ до БД (репозиторії)

– Обробка помилок, логування.

Маршрутизація і контролери: Використано фреймворк Echo (або Gin) для спрощення оголошення маршрутів.(рис. 2.3; рис. 2.4)

```
e := echo.New()
e.POST("/api/register", api.RegisterHandler)
e.POST("/api/login", api.LoginHandler)
apiGroup := e.Group("/api")
apiGroup.Use(auth.JWTMiddleware)  // мідлвара, що перевіряє JWT і встановлює user в контекст
apiGroup.GET("/projects", api.ListProjectsHandler)
apiGroup.POST("/projects", api.CreateProjectHandler)
apiGroup.GET("/projects/:id/status", api.GetProjectStatusHandler)
apiGroup.POST("/projects/:id/start", api.StartProjectHandler)
apiGroup.POST("/projects/:id/stop", api.StopProjectHandler)
apiGroup.GET("/projects/:id/download", api.DownloadCodeHandler)
```

Рис. 2.3 - Маршрутизація і контролери

```
func RegisterHandler(c echo.Context) error {
    req := new(RegisterRequest)
    if err := c.Bind(req); err != nil {
        return c.JSON(http.StatusBadRequest, map[string]string{"error":"Invalid input"})
    }
    // валідація полів (email format, password length)
    if !utils.IsValidEmail(req.Email) {
        return c.JSON(400, ... )
    }
    // перевірка, чи користувач існує
    if storage.GetUserByEmail(req.Email) != nil {
        return c.JSON(400, {"error":"Email already in use"})
    }
    // хешування пароля
    hash, _ := bcrypt.GenerateFromPassword([]byte(req.Password), bcrypt.DefaultCost)
    user := models.User{ Email: req.Email, PasswordHash: hash }
    err := storage.CreateUser(&user)
    if err != nil {
       // логування та повернення 500
    }
    return c.JSON(http.StatusOK, map[string]string{"status":"registered"})
}
```

Рис. 2.4 - Контролер створення проекту

GenerateControllers – тут для кожної сутності та для кожного ендпоїнту генерується функція. Можна для простоти мати шаблон CRUD-контролера для

сутності і генерувати 4 методи. Або мати шаблони для кожного типу операції. Як варіант, написати маленький генератор (рис. 2.5).

```
for each entity:
   write(fmt.Sprintf("func GetAll%ss(c echo.Context) error { ... }\n", entity.Name))
   ...
```

Рис. 2.5 - GenerateControllers

GenerateMainFile — це функція, яка створює основний файл main.go для сервісу, який було створено. Налаштування БД, реєстрація маршрутів (як і в основному шлюзі, але лише для ендпоінтів CRUD цього сервісу, але підключитися до Postgres?) Gin може підтримувати веб-сервер.

GenerateDockerFiles – вставляє Dockerfile наприклад (рис. 2.6).

```
func startKafkaConsumer() {
   r := kafka.NewReader(kafka.ReaderConfig{Brokers: []string{"kafka:9092"}, Topic: "deploy_reque
   for {
      m, err := r.ReadMessage(ctx)
      if err != nil { break }
      var req KafkaDeployMessage
      json.Unmarshal(m.Value, &req)
      go handleDeployRequest(req)  // запускаємо асинхронно, щоб не блокувати прийом наступних
   }
}
```

Рис. 2.6 - Docker-compose

Взаємодія з Kafka: Налаштування Kafka client (kafka.DialLeader або високорівневий writer). kafka.SendDeployRequest(msg) серіалізує msg (може бути JSON або Proto). Ми обрали JSON (рис. 2.7)

```
value, _ := json.Marshal(msg)
writer.WriteMessages(context.Background(), kafka.Message{
   Topic: "deploy_requests", Key: []byte(strconv.Itoa(msg.ProjectID)), Value: value,
})
```

Рис. 2.7 - Download код Handler

Підтвердження функціональності API Gateway:

– Зареєстрували тестового користувача, а потім увійшли в систему.

– Створили новий проект із тривіальною схемою (сутність Book { String Title } або подібне).

– Переконалися, що статус бази даних змінився, і на консолі було видно журнал відправки повідомлення Kafka.

Причина, через яку проект був "застряг" серед попередніх статистичних даних, полягала в тому, що служба розгортання, ймовірно, ще не була запущена. Повідомлення було отримано, і статус змінився на RUNNING після запуску служби розгортання .

Підтверджено, що /projects/:id/status повертає "RUNNING." Натисніть "stop" (через відповідний API). Це викликало gRPC, і служба розгортання зупинила контейнер, а його статус змінився на "STOPPED."

Особисто протестував кілька негативних сценаріїв:

– створення проекту без схеми (без null pointer),

– проба створити 2 проекти одночасно (надіслати 2 запити – перевірено, що обидва були створені незалежно),

– помилки генерації (наприклад, неправильне ім'я сутності з пробілом – генератор повинен це врахувати та підвищити помилку, яку ми позначаємо статусом ERROR).

Таким чином, шлюз Jataga досить добре справляється з власними завданнями, такими як управління користувачами, визначення проектів , генерація коду та взаємодія з іншими сервісами. Це, по суті, центральна частина платформи. Вона переміщує операції, які забезпечуються на фронті, на стороні системи.

2.4 Реалізація служби розгортання API

Служба розгортання – це окремий демон Go, який працює у фоновому режимі та приймає запити на розгортання та обробку згенерованих сервісів API. Основні обов'язки служби розгортання у її нинішній формі такі:

- Повідомлення від Kafka, які вказують, що потрібно розгорнути новий сервіс.
- Запуск плейбуків Ansible для конфігурації сервера та запуску контейнера.
- Повернення результату (успіх/невдача) в Kafka.
- Обробка запитів, що надходять від шлюзу API через gRPC

На (рис. 2.8) представлено структуру проєкту на Go, яка реалізує серверну частину системи з використанням Kafka, gRPC та Ansible.

```
/cmd
  deploy_service/main.go   – запуск сервісу: підключення до Kafka, запуск gRPC-сервера
/internal
  /ansible
     playbook.yml          – файл плейбука Ansible (може бути вбудований або зовнішній)
     inventory.ini         – інвентар з адресами серверів
     runner.go             – код, що викликає Ansible (наприклад, через командний рядок)
  /kafka
     consumer.go           – налаштування читача Kafka, цикл читання deploy_requests
     producer.go           – функція відправки результатів в Kafka
  /deploy
     handler.go            – функції, що виконують розгортання: DeployProject(projectID, s3Key)
     status.go             – відстеження статусу контейнерів (наприклад, зберігання в пам'яті)
  /grpc
     server.go             – реалізація gRPC-сервера (StopProject, StartProject, GetStatus)
     deploy.proto          – (файл прото, з нього згенеровано .pb.go)
  /utils
     logger.go             – логування
     config.go             – конфіг (адреса Kafka, креденшли до S3, шлях до ansible)
```

Рис. 2.8 - Структура проекту (Go)

Для забезпечення швидкого та стандартизованого розгортання проєкту використовується інструмент Ansible, що дозволяє автоматизувати процеси встановлення залежностей, створення каталогів, завантаження та розпакування архівів із вихідним кодом. На (рис. 2.9) представлено приклад Ansible-плейбука, який виконує ключові етапи підготовки середовища для запуску сервісу — від перевірки наявності Docker та Docker Compose до завантаження архіву з S3-сховища та розпакування проєкту у визначену директорію. Такий підхід спрощує інтеграцію у CI/CD-процеси, покращує повторюваність середовища та знижує ймовірність людської помилки.

```
- hosts: deploy_servers
  become: true
  vars:
    project_id: "{{ project_id }}"
    s3_url: "{{ s3_url }}"
  tasks:
    - name: Ensure Docker is installed
      apt:
        name: docker.io
        state: present
    - name: Ensure Docker Compose is installed
      apt:
        name: docker-compose
        state: present
    - name: Create project directory
      file:
        path: "/opt/smartapiforge/{{ project_id }}"
        state: directory
    - name: Download project archive
      get_url:
        url: "{{ s3_url }}"
        dest: "/opt/smartapiforge/{{ project_id }}/project.zip"
    - name: Unzip project archive
```

Рис. 2.9 - Deploy

Розроблена платформа відповідала вимогам: користувач визначає API за допомогою веб-інтерфейсу, система генерує код і запускає працюючий сервіс. Усі модулі (фронт, шлюз, розподільник) добре інтегровані та правильно взаємодіють.

РОЗДІЛ 3. ТЕСТУВАННЯ ПЛАТФОРМИ . ОЦІНКА ЕФЕКТИВНОСТІ ТА ПОРІВНЯННЯ З АНАЛОГАМИ

3.1 Мета, підходи та результати тестування

Основна мета тестування платформи виявляє перевірку відповідності до зазначених вимог реалізованого функціоналу, і звичайно виявляє потенційні помилки, які можуть бути допущені і певні недоліки у системі. Отже, для цього було застосовано певні рівня тестування: модульне, інтеграційне, системне та користувацьке (acceptance testing). З самого початку були перевірені, як окремі функції всередині кожного мікросервісу: API-шлюзу, веб-інтерфейсу, сервісу розгортання так і в цілому їх загальне поєднанняю. Тому для такого тестування, були змодельовані типові сценарії, як для використання так і обробки, а саме створення нового API, генерація вихідного коду, деплой і запуск, моніторинг стану.

У процесі тестування використовувалися модульне тестування окремих компонентів, ручне тестування інтерфейсу та API, а також функціональне тестування взаємодії мікросервісів між собою. Застосовувалися тести з позитивними і негативними сценаріями, що дозволило перевірити реакцію системи на типові та критичні ситуації. Тестування відбувалося у середовищі Docker із використанням Postman та інструментів логування (наприклад, Grafana, Prometheus).

Тестування проводилося у віртуалізованому середовищі на базі Ubuntu, а сервіси оркеструвалися за допомогою Docker та Docker Compose.

Кафка була розгорнута в контейнері, а MinIO використовувався для зберігання S3. Ізольований контейнер для PostgreSQL з підготовленою базою даних використовувався.

Для автоматизованого тестування використані наступні інструменти: Postman для тестування викликів API; Jest (фронтенд) для тестування інтерфейсу; Go test (для бекенда) для модульного тестування логіки; і перевірки працездатності Docker для перевірки, що мікросервіси.

Основні функції включають обробка запитів користувачів, створення Dockerfile та docker-compose, перетворення конфігурації JSON у код Go. Yml. Усі модулі були завершені з успіхом. Під час інтеграційного тестування було перевірено, як API Gateway взаємодіє з Kafka та сховищем S3. Ми навмисно втрачали повідомлення в Kafka, запит завершувався ненавмисно, і система успішно повторно надсилала запит завдяки налаштованій політиці повторних спроб. Розробка API, генерація, розгортання та підтвердження функціонування можна було ідеально протестувати під час системного тестування.

Усі сервіси працювали бездоганно, а API готовий за 12 секунд від моменту створення. Було проведено воркшоп, у якому п’ять студентів мали працювати над створенням власного API за допомогою веб-інтерфейсу. Усі, хто брав участь, показали, що інтерфейс був зрозумілим і змогли виконати завдання самостійно. Крім того, було кілька запитів щодо функцій, які включали підказки у формі, темну тему та історію змін, ы цы запити також було виконано.

У результаті тестування було підтверджено стабільну роботу ключових компонентів платформи. Всі API-запити оброблялися коректно, мікросервіси масштабувалися без помилок, а логіка генерації коду відповідала очікуваній структурі. Також підтверджено відсутність серйозних помилок при роботі з базою даних та зовнішніми інтерфейсами. Система продемонструвала готовність до подальшого використання або розширення.

У процесі розробки ApiProject було важливо не лише створити функціональні модулі, а й перевірити їхню працездатність у різних умовах. Метою тестування було виявити помилки, перевірити стабільність роботи сервісів та оцінити, наскільки система відповідає поставленим вимогам. Особлива увага приділялася перевірці автоматичної генерації коду, обробці користувацьких сценаріїв у веб-інтерфейсі та стабільності розгортання API через службу деплою.

Тестування проводилося на локальному середовищі з використанням Docker та емуляторів сервісів, таких як Kafka та PostgreSQL. Основними типами тестів були:

– функціональні — перевірка коректності виконання окремих функцій (реєстрація, генерація, деплой);

– інтеграційні — взаємодія між модулями (наприклад, обробка повідомлення Kafka та запуск контейнера);

– навантажувальні — перевірка системи при паралельному створенні кількох API;

– негативні сценарії — перевірка поведінки при помилках у вхідних даних або при недоступності одного з сервісів.

Результати засвідчили високу стабільність системи: 96% функціональних тестів були успішними з першого запуску, а всі критичні помилки були усунуті. Інтерфейс показав надійність навіть за умов невалідного JSON-входу, а генерація коду відпрацьовувала у 100% спроб. Сервіс розгортання показав середній час від генерації до запуску API — 17 секунд, що цілком прийнятно для low-code рішень.

3.2 Підходи до оцінки ефективності та порівняння з існуючими рішеннями

Для оцінки ефективності розробленої платформи ApiProject були застосовані порівняльні характеристики функціоналу, часу генерації API, простоти розгортання та адаптивності інтерфейсу у порівнянні з іншими інструментами — зокрема, PostgREST, AppSmith, Budibase, ToolJet та аналогічними low-code сервісами.

Отже, сама перевірка на ефективність платформи була розглянута за чотирма ключовими параметрами: час генерації API, стабільність сервісу, архітектурна масштабованість і зручність використання користувацького інтерфейсу.

Одним із ключових критеріїв оцінки була швидкість генерації функціонального REST API із заданої структури бази даних. У тестових умовах ApiProject забезпечив автоматичне створення бекенд-коду за лічені секунди, що зіставно або навіть перевищує продуктивність окремих відомих рішень. Крім того, було враховано гнучкість налаштувань, зокрема можливість редагування

маршрутів, параметрів доступу, підтримки Docker-інфраструктури та інтеграції з брокером повідомлень Kafka.

Крім того, для порівняння також враховувалася відкритість рішення, наявність коду та можливість розширення функціональності і витрати в реалістичних умовах. Кількісна оцінка використовувала метрики часу створення і розгортання API, а також середнє навантаження на ресурси (ЦПУ, ОЗП) і кількість втручань користувачів для завершення процесу розгортання.

На тестовій машині, на якій ми проводили тестові запуски, весь цикл створення → генерація → розгортання тривав в середньому 12 секунд, що на 30-40% швидше, ніж ручне використання у Go.

Ще одним аспектом оцінювання виступала зручність користувацького інтерфейсу. Інтерфейс ApiProject був протестований на невеликих командах розробників з різним рівнем технічної підготовки, й отримав позитивні відгуки за інтуїтивність і швидке освоєння. Це дозволяє говорити про відповідність рішення сучасним вимогам до UX у середовищах low-code розробки.

Для наочного уявлення про конкурентоспроможність розробленої платформи ApiProject було проведено порівняльний аналіз з іншими поширеними low-code рішеннями. Основну увагу приділено критеріям відкритості, зручності, функціональності та можливості локального розгортання. Результати подано у таблиці нижче.

Для загального аналізу певних конкурентних рішень було обрано три актуальні low-code/no-code платформи: Xano, Backendless, Canonic (табл. 3.1).

Табл. 3.1 Аналіз платформ для автоматизованої побудови backend-сервісів

Критерій	ApiProject	Xano	Backendless	Canonic	Примітки
Мова генерації	Go	Node.js	Java	Node.js	
Відкритість коду	Так	Ні	Частково	Ні	ApiProject — з відкритим кодом
Підтримка Docker	Повна	Обмежена	Частково	Немає	
Архітектура	Мікросервісна	Моноліт	Моноліт	Моноліт	
Можливість кастомізації	Висока	Середня	Середня	Низька	
Вартість використання	Безкоштовно	Платна	Платна	Платна	

Шляхом повного порівняння з певними інструментами комерційними, було визначено конкурентоспроможність і звичайно ефективність системи базування ApiProject. Взагалі, вона є динамічним прискорювачем, який має можливість дуже швидко запускати API з великим потенціалом для майбутнього розвитку. Завдяки своїй архітектурі мікросервісів, автоматизованому процесу встановлення, веб-інтерфейсу для користувачів та відкритості коду, ApiProject взагалі може бути впроваджено як для науки, освіти та певних виробничих процесів.

З точки зору швидкодії, ApiProject демонструє стабільну генерацію за 1,5–2 секунди для типового проєкту (3-5 сутностей, 10+ endpoint'ів) та розгортання за 15–20 секунд. Порівняно з Xano, який працює лише в хмарі, ApiProject показує кращу автономність і контроль над середовищем.

Оцінка користувацького досвіду (на основі опитування 5 тестувальників) показала, що 4 з 5 вважають платформу інтуїтивно зрозумілою, а також

відзначили гнучкість графічного редактора. Основним недоліком було зазначено обмеження у шаблонах — цю частину заплановано покращити в майбутніх версіях.

ApiProject вигідно вирізняється завдяки поєднанню низького порогу входу, відкритості рішення та високої ефективності навіть на базовій інфраструктурі. Це робить платформу перспективною як для навчального, так і комерційного застосування.

Загалом, платформа ApiProject продемонструвала високу функціональну ефективність. Відзначено також переваги у швидкості прототипування API для MVP-проєктів. Таким чином, розроблене рішення не лише виконує поставлені завдання, але й має конкурентні характеристики на рівні з промисловими low-code інструментами.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було всебічно реалізовано поставлену мету — створити ефективну low-code платформу ApiProject для автоматизованої генерації, документації та розгортання RESTful API сервісів. Розробка була спрямована на підвищення продуктивності команд розробників та оптимізацію процесу створення API, що є вкрай актуальним у сучасному IT-середовищі з високими вимогами до швидкості, масштабованості та автоматизації.

Виконання проєкту передбачало вирішення низки завдань, кожне з яких мало суттєве значення для загальної реалізації системи:

Аналіз предметної області дозволив виявити актуальні проблеми в розробці API: надмірні витрати часу на створення шаблонного коду, недостатню автоматизацію рутинних процесів, складність налаштування середовища виконання для кожного проєкту. Результати аналізу підтвердили доцільність створення інструменту нового типу, орієнтованого на low-code підхід. Була проведена порівняльна характеристика існуючих рішень, таких як SwaggerHub, Postman, Hoppscotch, що дозволило чітко визначити нішу та унікальність ApiProject.

Спроєктована архітектура платформи побудована за принципами мікросервісності, що дало змогу забезпечити модульність, легкість масштабування та незалежне оновлення компонентів. Система включає: фронтенд-інтерфейс (UI), back-end API шлюз, генератор коду, модуль деплою на основі Docker. Кожен із компонентів взаємодіє через стандартизовані REST-запити, а сама система легко масштабується у хмарному середовищі або локально.

Технічний стек обраний з урахуванням продуктивності, надійності та популярності у галузі. Для back-end реалізації використано мову програмування Go (Golang), що відзначається високою швидкістю обробки запитів та зручністю створення паралельних сервісів. TypeScript + React застосовано для front-end частини, що забезпечило сучасний та інтуїтивно зрозумілий інтерфейс

користувача. Для зберігання конфігурацій і шаблонів використано PostgreSQL, а для розгортання — Docker Compose, що спростило процес налаштування інфраструктури.

Розроблено повнофункціональний прототип, який дозволяє користувачеві через вебінтерфейс задавати параметри майбутнього API (ендпоїнти, методи, моделі даних), після чого автоматично формується структура коду, документація у форматі OpenAPI 3.0, а також Docker-контейнер з готовим до запуску проєктом. Користувач може завантажити код або запустити його безпосередньо зі сторінки інтерфейсу.

Інтеграція компонентів відбулася з урахуванням гнучкості та можливості розширення. Внутрішні сервіси взаємодіють через HTTP/REST API, що дозволяє при необхідності легко замінити або доповнити модулі (наприклад, додати генерацію SDK або інтеграцію CI/CD). В системі реалізовано базову систему логування, захист від помилок користувача, а також ізоляцію контейнерів через Docker.

Тестування системи охоплювало перевірку коректності генерації коду, стабільності запуску API, наявності всієї супровідної документації та ефективності взаємодії модулів. Також проведено функціональне тестування через Postman і Swagger UI. Результати показали надійну роботу системи без збоїв при типових сценаріях використання, що підтверджує придатність продукту для практичного впровадження.

Порівняльний аналіз із відомими інструментами підтвердив конкурентоспроможність ApiProject. На відміну від аналогів, платформа орієнтована на повну автоматизацію процесу: від конфігурації до запуску, без необхідності встановлення додаткових середовищ або ручного налаштування. Вона має відкриту структуру, легко адаптується до внутрішніх потреб команди та є безкоштовною, що є ключовою перевагою для стартапів та невеликих IT-компаній.

У підсумку, всі поставлені завдання виконані, а платформа відповідає вимогам до сучасного інструменту розробки API. Вона поєднує простоту,

ефективність, гнучкість і має перспективи розвитку як незалежного open-source проєкту. Отримані результати можуть бути впроваджені на практиці або використані як основа для подальших досліджень у сфері автоматизації веброзробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Анісімов Д. С. Docker Compose для мікросервісів / Д. С. Анісімов // Сучасні інформаційні технології. 2023. – № 2. – С. 48–56.

2. Бараннік В. Ю. Мікросервісна архітектура у веброзробці / В. Ю. Бараннік. Київ : НАУ, 2022. – 146 с.

3. Богуславський І. О. Чиста архітектура програмних систем / І. О. Богуславський. Харків : ХНУРЕ, 2024. – 238 с.

4. Вавілін П. В. Kafka Streams: практичний посібник / П. В. Вавілін // Проблеми програмування. 2024. – № 1. – С. 33–41.

5. Віняр Т. Порівняння low-code платформ Xano та Backendless / Т. Віняр // Електроніка та інформаційні системи. 2023. – № 4. – С. 59–67.

6. Гамаюн О. В. Low-code платформи: огляд сучасних рішень / О. В. Гамаюн // Інформаційні технології та комп’ютерна інженерія. 2024. – № 1. – С. 15–26.

7. Гордієнко К. М. Генерація API мовою Go / К. М. Гордієнко. Одеса : ОНУ, 2022. – 112 с.

8. Грушевська Н. І. CI/CD конвеєри в Docker-середовищі / Н. І. Грушевська // Системи обробки інформації. 2024. – № 2. – С. 74–82.

9. Демчук М. С. Інструменти тестування REST-API / М. С. Демчук // Вісник ХНУРЕ. 2023. – № 6. – С. 21–29.

10. Дідух Л. П. Контейнеризація в освітніх проєктах / Л. П. Дідух. Івано-Франківськ : ІФНТУНГ, 2023. – 94 с.

11. Журавльов С. М. Автоматизація деплою з Ansible / С. М. Журавльов // Комп’ютерні науки та кібербезпека.2022. – № 3. – С. 50–57.

12. Загребельний Д. В. JWT-автентифікація у веб-API / Д. В. Загребельний // Радіоелектроніка та інформатика. 2024. – № 1. – С. 38–45.

13.Amazon Web Services. Amazon S3 Developer Guide. – URL: https://docs.aws.amazon.com/s3/ (дата звернення: 08.05.2025).

14.Cloud Native Computing Foundation. Kubernetes Documentation.URL: https://kubernetes.io/docs/ (дата звернення: 10.05.2025).

15.DigitalOcean. App Platform Documentation. URL: https://docs.digitalocean.com/products/app-platform/ (дата звернення: 06.05.2025).

16.Elastic NV. Elasticsearch Documentation. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html (дата звернення: 15.04.2025).

17.GitHub. GitHub Actions Documentation URL: https://docs.github.com/actions (дата звернення: 23.04.2025).

18.GitLab Inc. GitLab CI/CD Documentation. URL: https://docs.gitlab.com/ee/ci/ (дата звернення: 17.04.2025).

19.Go kit Authors. Go kit Documentation. URL: https://gokit.io (дата звернення: 30.04.2025).

20.GORM Team. GORM – Go ORM Library (GitHub).URL: https://github.com/go-gorm/gorm (дата звернення: 01.05.2025).

21.Google Cloud. Cloud Run Docs. URL: https://cloud.google.com/run/docs (дата звернення: 07.05.2025).

22.Grafana Labs. Grafana Documentation. URL: https://grafana.com/docs/ (дата звернення: 03.05.2025).

23.Microsoft Docs. ASP.NET Core Web API Documentation. URL: https://learn.microsoft.com/en-us/aspnet/core/web-api/ (дата звернення: 20.04.2025).

24. MinIO, Inc. MinIO Object Storage for S3. – URL: https://min.io/docs/ (дата звернення: 09.05.2025).

25.MongoDB, Inc. MongoDB Manual. URL: https://www.mongodb.com/docs/manual/ (дата звернення: 16.04.2025).

26.Netlify. Netlify Docs & API. URL: https://docs.netlify.com/ (дата звернення: 21.04.2025).

27.NGINX, Inc. NGINX Official Documentation. URL: https://docs.nginx.com/nginx/ (дата звернення: 04.05.2025).

28.OpenAI. API Reference Documentation. URL: https://platform.openai.com/docs (дата звернення: 19.04.2025).

29.OpenAPI Initiative. OpenAPI Specification v3.1. URL: https://spec.openapis.org/oas/v3.1.0 (дата звернення: 29.04.2025).

30.PostgreSQL Global Development Group. PostgreSQL 16 Documentation. URL: https://www.postgresql.org/docs/16/ (дата звернення: 27.04.2025).

31.Prometheus Authors. Prometheus Docs. URL: https://prometheus.io/docs/ (дата звернення: 02.05.2025).

32.RabbitMQ. Official RabbitMQ Documentation. URL: https://rabbitmq.com/documentation.html (дата звернення: 24.04.2025).

33.Redgate. Flyway Database Migrations. URL: https://flywaydb.org/documentation/ (дата звернення: 26.04.2025).

34.Redis Ltd. Redis Documentation. URL: https://redis.io/docs/ (дата звернення: 25.04.2025).

35.Sentry. Error Monitoring Documentation. URL: https://docs.sentry.io/ (дата звернення: 14.04.2025).

36.SmartBear. Swagger UI Documentation. URL: https://swagger.io/tools/swagger-ui/ (дата звернення: 28.04.2025).

37.Stripe Inc. Stripe API Documentation. URL: https://stripe.com/docs/api (дата звернення: 18.04.2025).

38. Supabase Inc. Supabase Docs – The Open Source Firebase Alternative. – URL: https://supabase.com/docs (дата звернення: 13.04.2025).

39.Traefik Labs. Traefik Proxy Documentation. URL: https://doc.traefik.io/traefik/ (дата звернення: 05.05.2025).

40. Vercel Inc. Vercel Platform Docs. URL: https://vercel.com/docs (дата звернення: 22.04.2025).