Д. Д. Узун, А. Г. Тецький, М. Р. Немов

ТЕХНОЛОГІЇ ДЕВОПС.
РЕАЛІЗАЦІЯ CI/CD ПРОЕКТА З ВІДКРИТИМ ВИХІДНИМ КОДОМ

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

Д. Д. Узун, А. Г. Тецький, М. Р. Немов

ТЕХНОЛОГІЇ ДЕВОПС.
РЕАЛІЗАЦІЯ CI/CD ПРОЕКТА З ВІДКРИТИМ ВИХІДНИМ КОДОМ

Навчальний посібник

Харків «ХАІ» 2024

УДК 004.4’24(075.8)
У34

Рецензенти: канд. техн. наук А. А. Акулинічев,
д-р техн. наук В. А. Кроснобаєв

Узун, Д.Д.,

У34 Технології ДЕВОПС. Реалізація CI/CD проекта з відкритим вихідним кодом [Електронний ресурс] : навч. посіб. / Д. Д. Узун, А. Г. Тецький, М. Р. Немов. – Харків : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харків. авіац. ін-т», 2024. – 103 с.

Висвітлено матеріали відповідно до програми навчальної дисципліни «Технології ДевОпс», який викладається студентам зі спеціальності «Комп’ютерна інженерія». Наведено методику реалізації безперервної інтеграції та доставки або розгортання проекта з відкритим вихідним кодом з практичними прикладами для студентів усіх форм навчання. Надано перелік основної й додаткової літератури для самостійного виконання роботи.

Для студентів денної й заочної форм навчання технічних спеціальностей, а також для всіх осіб, що зацікавлені у підвищенні свого рівня технічної освіти.

Іл. 85. Табл. 7. Бібліогр.: 21 назв

© Узун Д.Д., А.Г. Тецький, М.Р. Немов, 2024
© Національний аерокосмічний
університет ім. М.Є. Жуковського
«Харківський авіаційний інститут», 2024

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ. 7
ВСТУП 8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ 11
1.1 Актуальність і аналіз предметної області 11
1.2 Аналіз існуючих рішень 13
1.3 Проблеми та способи їх рішення 13
1.4 Постановка задачі 14
Висновки за розділом 15
2 АНАЛІЗ ТА ВИБІР ІНСТРУМЕНТІВ РОЗРОБЛЕННЯ ІНФРАСТРУКТУРИ 17
2.1 Аналіз вимог до інфраструктури 17
2.2 Основні компоненти інфраструктури 17
2.3 Аналіз та вибір існуючих методологій та фреймворків розроблення програмного забезпечення 18
2.4 Вибір програмного забезпечення для створення інфраструктури 19
2.5 Вибір програмного забезпечення для розгортання ПЗ на інфраструктурі ... 20
2.6 Визначення застосунку для розгортання 20
2.7 Вибір програмного забезпечення для моніторингу та візуалізації 22
2.8 Вибір інструментального засобу безперервної інтеграції коду 23
Висновки за розділом 24
3 РОЗРОБКА ПРОеКТНОГО РІШЕННЯ 25
3.1 Реалізація локальної інфраструктури та розгортання застосунку 26
3.1.1 Налаштування віртуальної машини на Oracle VirtualBox 27
3.1.2 Налаштування PostgreSQL та S3 28
3.1.3 Встановлення Java і збірка бекенду 28
3.1.4 Установка Nginx та збірка фронтенду 29
3.1.5 Перевірка роботи застосунку 29
3.2 Розгортання інфраструктури у хмарі з контейнеризованим додатком 30
3.2.1 Створення сервера у хмарі 31

3.2.2 Встановлення Docker на сервер....32
3.2.3 Опис Dockerfile для бекенду додатка....32
3.2.4 Опис Docker Compose та перевірка роботи застосунку....34
3.3 Автоматичне розгортання інфраструктури через код у хмарі з контейнеризованим додатком....36
3.3.1 Початкова активація API сервісів GCP для Terraform....37
3.3.2 Створення користувача для взаємодії з ресурсами GCP....38
3.3.3 Створення образу бекенду і нового Dockerfile для фронтенду....39
3.3.4 Встановлення Terraform....40
3.3.5 Структура маніфесту Terraform....40
3.3.6 Основний модуль....42
3.3.7 Модуль мережі....43
3.3.8 Модуль екземплярів обчислення....44
3.3.9 Виконувані скрипти при запуску віртуальних машин....44
3.3.10 Запуск інфраструктури....45
3.3.11 Перевірка результату роботи застосунку....47
3.4 Розгортання контейнеризованного застосунку за допомогою Kubernetes....48
3.4.1 Створення образів Docker для фронтенду та бекенду додатка для кластеру Kubernetes....49
3.4.2 Створення кластеру Kubernetes....49
3.4.4 Створення Ingress в кластері Kubernetes та призначення доменного імені для застосунку....51
3.4.5 Перевірка працездатності застосунку у кластері Kubernetes....56
3.4.6 Розгортання застосунку через Helm у кластері Kubernetes....58
3.5 Налаштування системи безперервної інтеграції....63
3.6 Налаштування системи моніторингу....67
3.6.1 Моніторинг стану кластеру Kubernetes....68
3.6.2 Моніторинг стану фронтенду у кластері Kubernetes....71
Висновки за розділом....75
4 ТЕСТУВАННЯ ТА ВЕРИФІКАЦІЯ....77
4.1 Функції інфраструктури, які необхідно протестувати....77

4.2 Верифікація інфраструктури 77
4.3 Тестування інфраструктури та CI/CD 79
Висновки за розділом 85
ВИСНОВКИ 86
ПЕРЕЛІК ПОСИЛАНЬ 87
ДОДАТОК А 89

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ.

ОЗП – оперативний запам’ятовуючий пристрій;
ОС – операційна система;
ПЗ – програмне забезпечення;
ЦП – центральний процесор;
API – (з англ. Application Programming Interface);
AWS – (з англ. Amazon Web Services);
CI/CD – (з англ. Continuous Integration/Continuous Deployment);
CLI – (з англ. Command Line Interface);
DevOps – (з англ. Development Operations);
EC2 – (з англ. Elastic Compute Cloud);
GCE – (з англ. Google Compute Engine);
GCP – (з англ. Google Cloud Platform);
GPG – (з англ. GNU Privacy Guard);
HLC - (з англ. HashiCorp Configuration Language);
HTTP – (з англ. HyperText Transfer Protocol);
IAM – (з англ. Identity and Access Management);
IaC – (з англ. Infrastructure as Code);
IP – (з англ. Internet Protocol);
JSON – (з англ. JavaScript Object Notation);
JRE – (з англ. Java Runtime Environment);
NAT – (з англ. Network Address Translation);
OpenJDK – (з англ. Open Java Development Kit);
S3 – (з англ. Simple Storage Service);
SDK – (з англ. Software Development Kit);
SQL – (з англ. Structured Query Language);
SSH – (з англ. Secure Shell);
TCP – (з англ. Transmission Control Protocol);
UDP – (з англ. User Datagram Protocol);
VM – (з англ. Virtual Machine);
VPC – (з англ. Virtual Private Cloud);
vCPU – (з англ. Virtual Central Processing Unit);
YAML – (з англ. YAML Ain't Markup Language).

ВСТУП

У сучасних умовах, коли технологічний прогрес і цифрова трансформація відіграють вирішальну роль у розвитку бізнесу та суспільства в цілому, питання ефективного управління інформаційними системами набувають особливої актуальності. Одним із найбільш динамічно розвиваючихся підходів до забезпечення ефективності та надійності ІТ-інфраструктури є концепція DevOps. DevOps (скорочено від Development Operations) об'єднує процеси розробки та експлуатації програмного забезпечення, створюючи умови для тісної взаємодії між розробниками, тестувальниками та системними адміністраторами. Такий підхід дозволяє значно скоротити час на розгортання додатків, підвищити якість програмного забезпечення та забезпечити його безперервне оновлення.

Спеціальність 123 "Комп'ютерна інженерія", яка є основою підготовки майбутніх фахівців у галузі ІТ, потребує інтеграції новітніх технологій у навчальний процес. Вивчення технологій DevOps стає невід'ємною частиною підготовки сучасних інженерів, які здатні швидко реагувати на зміни ринку та адаптуватися до нових вимог роботодавців. Це, в свою чергу, підвищує їх конкурентоспроможність та затребуваність на ринку праці.

Значна кількість сучасних додатків, особливо у сфері великих даних, хмарних обчислень та штучного інтелекту, вимагають безперервного циклу розробки, тестування та впровадження. Використання DevOps забезпечує автоматизацію цих процесів, що дозволяє зменшити кількість помилок, прискорити розгортання нових функцій та мінімізувати час простою системи. Це робить DevOps незамінним інструментом для сучасних ІТ-команд.

Окрім того, впровадження принципів DevOps дозволяє краще управляти інфраструктурою, автоматизувати її розгортання, забезпечувати моніторинг і підтримку на високому рівні. Це створює умови для ефективної роботи команд розробників і системних адміністраторів, сприяє підвищенню продуктивності та якості розробки програмного забезпечення.

Навчальний посібник, розроблений в рамках цієї роботи, спрямований на формування у студентів глибокого розуміння принципів DevOps, а також практичних навичок, необхідних для розгортання,

управління та підтримки IT-інфраструктури. Особлива увага приділяється таким аспектам, як безперервна інтеграція, доставка та розгортання програмного забезпечення, автоматизація управління конфігурацією, моніторинг та інші ключові компоненти DevOps.

Впровадження принципів DevOps у навчальний процес дозволяє підготувати студентів до реальних викликів, з якими вони зіткнуться у професійній діяльності. Це забезпечить їм конкурентні переваги на ринку праці та підвищить ефективність їхньої роботи в умовах сучасної економіки, яка стрімко розвивається і потребує нових підходів до управління IT-інфраструктурою.

Таким чином, дана робота є важливим внеском у підготовку кваліфікованих фахівців у галузі комп'ютерної інженерії, які зможуть ефективно використовувати технології DevOps для вирішення складних завдань, що виникають в умовах сучасного цифрового світу. З розвитком технологій та збільшенням вимог до швидкості випуску нових версій програмного забезпечення, важливість ефективної автоматизації процесів розробки, тестування та розгортання додатків зростає. Щоб забезпечити високу якість та безперебійність роботи продуктів, виникла потреба у нових підходах. Ця потреба привела до появи парадигми DevOps, яка об’єднує практики безперервної інтеграції, безперервного постачання та безперервного розгортання CI/C.

Однак, впровадження DevOps у великих компаніях часто стикається з низкою викликів, таких як складність налаштування та управління інфраструктурою, відсутність уніфікованих інструментів для різних етапів процесу розробки, складність моніторингу та усунення збоїв. Для подолання цих проблем потрібне комплексне рішення, яке забезпечить автоматизацію всього циклу розробки, тестування та розгортання додатків.

Метою даної роботи є створення повнофункціональної інфраструктури для реалізації концепції DevOps на прикладі розгортання тестового веб-застосунку. Для цього був проведений аналіз ринку праці в сфері DevOps та визначені найбільш затребувані технології та інструменти.

На основі цього аналізу було спроєктовано інфраструктурне рішення, яке включає: створення хмарної інфраструктури на базі Google Cloud Platform за допомогою Terraform для налаштування віртуальних машин, мереж та інших ресурсів; автоматизоване розгортання та конфігурація Kubernetes; створення тестового веб-застосунку на базі Nginx та його контейнеризація за допомогою Docker; налаштування системи моніторингу

Prometheus, Grafana та Telegraf для збору та візуалізації метрик; інтеграція з GitLab для автоматичної збірки, тестування та розгортання застосунку через Docker-контейнери.

У навчальному посібнику детально розглянуто процес проєктування, реалізації та налаштування кожного компоненту інфраструктури. Також наведені практичні приклади використання створеного рішення на різних етапах безперервної інтеграції, доставки та розгортання.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Актуальність і аналіз предметної області

Дана робота присвячена створенню повноцінної DevOps інфраструктури для автоматизації процесів розробки, тестування та розгортання програмного забезпечення. Ця область є однією з найбільш актуальних та затребуваних у сучасній IT-індустрії.

Швидкий розвиток інформаційних технологій, збільшення вимог до якості та оперативності випуску нових продуктів змушують компанії шукати ефективні рішення для оптимізації процесу розробки програмного забезпечення. DevOps стає невід'ємною частиною великих IT-проєктів, адже дозволяє автоматизувати більшість етапів життєвого циклу створення додатків – від написання коду до його розгортання та моніторингу в продакшн-середовищі [1].

Однак, на шляху впровадження DevOps існує низка викликів: складність налаштування інфраструктури, відсутність уніфікованих інструментів для різних етапів процесу, проблеми з моніторингом та усуненням збоїв. Для подолання цих проблем потрібен комплексний підхід із залученням різноманітних технологій та рішень.

Перед тим, як приступити до проєктування інфраструктури, було проведено ґрунтовний аналіз ринку праці у сфері DevOps. Для цього було розглянуто 23 вакансій з вимогою від 1 до 3 років досвіду роботи на платформі jobs.dou.ua [2]. На основі вимог до навичок та інструментів, зазначених у цих вакансіях, була побудована рейтингова таблиця найбільш затребуваних технологій DevOps на ринку.

З аналізу випливає, що найбільш популярними та необхідними технологіями на сьогодні є: Linux, Bash, AWS, CI/CD та Jenkins, Kubernetes, Docker, Python, Terraform, Ansible, Grafana, Prometheus, MySQL, Nginx, Google Cloud Platform, GitLab та деякі інші. Саме тому для реалізації практичного DevOps-рішення було вирішено використати цей стек провідних технологій та інструментів.

Запропонована інфраструктура буде орієнтована на вирішення типових завдань DevOps, таких як забезпечення безперервної інтеграції, постачання, розгортання та моніторингу контейнеризованого веб-застосунку. Комбінація обраних інструментів дозволить повною мірою продемонструвати можливості автоматизації усього циклу створення та

експлуатації програмного продукту, відповідаючи вимогам роботодавців та сучасним тенденціям IT-індустрії.

Таблиця 1.1 – Рейтинг технологій на ринку праці за дослідженням

Технологія	Збіг																							
Linux and Bash	23	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
AWS	17	1		1		1	1	1	1		1	1	1	1	1	1	1				1	1	1	1
CI/CD and Jenkins	17	1	1		1		1	1	1			1	1		1	1	1	1	1	1		1	1	1
Kubernetes	15		1	1			1			1	1	1		1	1	1	1	1	1		1	1	1	
Docker	14			1	1	1		1	1	1	1			1		1	1			1	1	1	1	
Python	13	1		1	1	1		1		1		1	1	1	1	1	1					1		
Terraform	13	1		1			1				1	1			1	1	1		1	1	1	1	1	
Ansible	12	1		1	1	1						1	1		1	1	1			1		1	1	
Grafana	12			1	1	1			1		1				1	1	1	1		1	1	1		
Prometheus	12			1	1	1				1	1				1	1	1	1		1	1	1		
MySQL	12		1	1	1	1			1	1						1	1	1		1	1	1		
Nginx	10	1			1					1			1			1	1	1		1	1	1		
GCP	8	1				1	1		1							1	1	1	1					
Gitlab	8					1	1			1					1	1	1			1	1			
PostgreSQL	8	1		1	1	1										1	1			1	1			
Azure	6	1				1		1				1	1				1							
Kafka	6			1	1				1						1		1				1			
NoSQL	6		1		1					1					1	1						1		
Java	5										1		1		1			1				1		
MongoDB	5				1	1				1						1	1							
JS and Node.js	5	1			1								1			1		1				1		
AWS CloudFormation	4	1									1						1				1			
DNS	4								1				1	1										1
HELM	4			1								1					1						1	
PHP and PHP-fpm	4				1											1	1					1		
Redis	4															1	1	1			1			
Zabbix	4				1	1				1										1				
Apache	3				1					1											1			
AWS CodeBuild	3						1		1		1													
Chief	3	1				1							1											
Cassandra	3														1	1	1							
DigitalOcean	3						1									1	1							
ElasticSearch	3									1						1	1							
Hyper-V	3				1					1			1											
LXC	3					1			1								1							
OpenVpn	3				1	1								1										
Puppet	3	1										1					1							
Sentry	2			1		1																		
TLS / SSL	2								1															1
Cloudflare	1																1							
GCP CloudBuild	1						1																	
GitHub Actions	1						1																	
Jaeger	1			1																				
Nagios	1									1														
OpenDistro	1			1																				
Windows Server	1									1														

1.2 Аналіз існуючих рішень

На початку розвитку інформаційних технологій інфраструктура для розгортання застосунків зазвичай була локальною, на власних серверах компанії [3]. Цей підхід вимагав значних ресурсів на придбання та обслуговування фізичних серверів, масштабування було складним і дорогим.

З появою хмарних провайдерів, таких як AWS, з’явилась можливість оренди віртуальних машин та інших ресурсів за потребою. Це дозволило компаніям зосередитися на розробці, а не управлінні інфраструктурою.

Паралельно відбувався розвиток контейнеризації програм за допомогою Docker для вирішення проблем переносимості та управління залежностями додатків [4]. Згодом з’явилась мікросервісна архітектура для розподілу навантаження між компонентами системи.

Для оркестрації контейнеризованих мікросервісів було створено Kubernetes від Google [5]. Він дозволяє створювати розподілені кластери, автоматично розподіляти ресурси та керувати інфраструктурою мікросервісів.

Таким чином, сучасні рішення еволюціонували від локальних серверів до хмарних інфраструктур, контейнеризації, мікросервісної архітектури та оркестрації за допомогою Kubernetes, забезпечуючи гнучкість, масштабованість, надійність та швидкість розробки.

1.3 Проблеми та способи їх рішення

Однією з ключових проблем є необхідність ефективного моніторингу величезної кількості компонентів інфраструктури – від окремих контейнерів до кластерів і систем зберігання даних. Потрібно відстежувати метрики продуктивності, використання ресурсів, стан служб та збирати логи з усіх рівнів системи. Без належного моніторингу дуже складно виявляти збої, аналізувати проблеми та вчасно на них реагувати.

Крім того, виникають складнощі з управлінням такою розподіленою та динамічною інфраструктурою. Ручна зміна конфігурацій ресурсів у хмарі чи оркестратора є ризикованою та може призвести до порушення роботи всієї системи. Також проблематично відновлювати інфраструктуру в ідентичному стані після збоїв чи помилок.

Ще одна проблема пов’язана з контейнеризацією додатків та необхідністю швидко випускати нові версії програмного забезпечення. Потрібен надійний спосіб збирати, тестувати, публікувати та розгортати оновлені контейнери без ручного втручання.

Сучасні додатки часто потребують використання кількох середовищ – окремих для розробки, тестування, та випуску, що ускладнює управління конфігураціями та розгортаннями різних версій програмного забезпечення в цих середовищах.

Концепція інфраструктури як коду IaC дає змогу описувати бажану конфігурацію всіх ресурсів за допомогою коду та автоматично розгортати чи відновлювати її за потреби, що підвищує надійність, захищає від людських помилок та спрощує бекапи і версіонування.

Практики безперервної інтеграції та постачання в поєднанні з контейнеризацією додатків допомагають автоматизувати процес збірки, тестування та публікації оновлених версій програмного забезпечення за кожною зміною в коді.

Інструменти для оркестрації контейнерних робочих навантажень, такі як Kubernetes, забезпечують можливість створювати ізольовані простори для різних середовищ та легко розгортати в них необхідні версії додатків з відповідними конфігураціями.

1.4 Постановка задачі

У рамках даного навчального посібника спроектовано та реалізовано повнофункціональна DevOps інфраструктура із застосуванням провідних підходів та інструментів. Ключовими завданнями є:

– створити повністю автоматизовану хмарну інфраструктуру із заданими мережевими налаштуваннями, правилами безпеки і потрібними обчислювальними ресурсами;

– розгорнути кластерне середовище для оркестрації контейнерів додатків, балансування навантаження, масштабування;

– підготувати тестове веб-застосування, упакувати його в контейнер для незалежного розгортання;

– налаштувати систему збору, агрегації та візуалізації метрик з компонентів інфраструктури;

– інтегрувати процеси безперервної інтеграції та безперервного постачання CI/CD для автоматичної збірки, тестування і розгортання нових версій застосунку.

Основні функціональні вимоги:

– можливість автоматичного створення та відновлення всієї інфраструктури з коду;
– гнучке масштабування обчислювальних ресурсів під поточні потреби;
– ізольоване розгортання застосунку у середовищах «dev» та «prod»;
– балансування навантаження між декількома примірниками застосунку;
– безпечне оновлення застосунку без простоїв;
– моніторинг стану всіх компонентів та оперативне виявлення проблем;
– автоматична збірка та тестування нових версій на кожну зміну в коді;
– розгортання успішно зібраних релізів у різні середовища.

Висновки за розділом

У цьому розділі було проведено аналіз автоматизації процесів розробки та розгортання програмного забезпечення в контексті DevOps. Розглянуто актуальність цієї тематики, існуючі підходи та рішення, а також виклики, з якими стикаються компанії у цій сфері.

Через потребу швидкого та безперебійного випуску нових версій програмних продуктів виникла необхідність впровадження DevOps практик для автоматизації життєвого циклу створення ПЗ. Еволюція підходів пройшла шлях від локальних серверів до хмарних інфраструктур, контейнеризації додатків, мікросервісної архітектури та оркестрації за допомогою Kubernetes. Проаналізовано основні проблеми, що виникають при підтримці складних контейнеризованих DevOps інфраструктур: моніторинг ресурсів, управління конфігураціями, безперервна інтеграція, роздільне середовище для різних оточень. Визначено способи вирішення цих проблем за допомогою сучасних інструментів та практик. Для створення практичного рішення, яке продемонструє можливості автоматизації DevOps процесів, було поставлено задачу спроєктувати та

реалізувати комплексну інфраструктуру з використанням хмарного провайдера, контейнеризації, оркестрації, CI/CD і моніторингу.

Реалізоване рішення охоплює повний цикл DevOps – від автоматизованого розгортання ресурсів до безперервної доставки та моніторингу працюючого застосунку в хмарі. Це дозволить продемонструвати практичне застосування актуальних технологій та принципів безперервної інтеграції, розгортання і постачання програмного забезпечення.

2 АНАЛІЗ ТА ВИБІР ІНСТРУМЕНТІВ РОЗРОБЛЕННЯ ІНФРАСТРУКТУРИ

2.1 Аналіз вимог до інфраструктури

Для реалізації практик DevOps та автоматизації життєвого циклу розроблення, тестування та розгортання програмного забезпечення, інфраструктура має відповідати наступним ключовим вимогам:

- інфраструктура повинна легко адаптуватися до мінливих навантажень та потреб застосунку, забезпечуючи можливість горизонтального та вертикального масштабування ресурсів;
- інфраструктура має бути стійкою до збоїв та забезпечувати високу доступність застосунку шляхом резервування критичних компонентів, балансування навантаження та автоматичного відновлення після відмов;
- інфраструктура повинна відповідати вимогам безпеки, забезпечуючи захист даних, додатків та компонентів від несанкціонованого доступу, шкідливих програм та інших загроз, необхідно передбачити механізми шифрування, аутентифікації, авторизації та аудиту;
- інфраструктура має надавати засоби для всебічного моніторингу стану, продуктивності та використання ресурсів додатків та системних компонентів;
- процеси розгортання, конфігурування, оновлення та керування інфраструктурою мають бути максимально автоматизовані з метою підвищення ефективності, повторюваності та зменшення ризику людських помилок.

2.2 Основні компоненти інфраструктури

Для задоволення описаних вимог та реалізації практик DevOps, інфраструктура матиме такі ключові компоненти:

- хмарна платформа для забезпечення гнучкості, масштабованості та відмовостійкості;
- система управління конфігураціями для автоматизованого розгортання та керування інфраструктурними ресурсами на основі коду;
- система контейнеризації для упакування додатків разом з їх залежностями та бібліотеками в ізольовані контейнери;

– система оркестрації контейнерів для автоматичного розгортання, масштабування та керування контейнеризованими додатками;
– система моніторингу для збору, візуалізації та аналізу метрик і логів з усіх компонентів інфраструктури;
– система безперервної інтеграції та доставки для автоматизації процесів збирання, тестування та розгортання додатків.

Така комплексна архітектура із зазначеними компонентами забезпечить гнучку, масштабовану та автоматизовану інфраструктуру для реалізації концепцій DevOps у розробці та експлуатації сучасних додатків.

2.3 Аналіз та вибір існуючих методологій та фреймворків розроблення програмного забезпечення

Методологія розроблення програмного забезпечення – це систематичний підхід до планування, організації та контролю процесу створення програмних продуктів [6]. Її основна мета полягає в забезпеченні ефективної співпраці в команді, чіткого розподілу обов'язків, дотримання термінів та вимог якості.

Існують різні методології, які пропонують альтернативні шляхи організації робочого процесу. Традиційні методології, такі як Waterfall, передбачають лінійний, послідовний підхід із чітким розподілом етапів розроблення. Однак вони не завжди добре пристосовані до мінливих вимог та часто призводять до зайвої бюрократії.

З появою гнучких методологій Agile, акцент зміщується на ітеративний, інкрементний підхід з тісною співпрацею в команді та швидким реагуванням на зміни.

Ключовими принципами Agile є:
– ітеративний підхід з короткими циклами розроблення;
– активна участь замовника на всіх етапах;
– постійне тестування та інтеграція;
– гнучка адаптація до змінних вимог;
– регулярні узгодження та зворотний зв’язок.

Найбільш популярними для Agile фреймворками є Scrum та Kanban. Scrum орієнтований на створення «інкременту» продукту в ітераційних спринтах. Kanban, в свою чергу, фокусується на візуалізації потоку робіт, обмеженні незавершених завдань та безперервному вдосконаленні процесів.

Саме методологія Kanban найкраще підходить для DevOps та автоматизації інфраструктури через її гнучкість та можливість швидко реагувати на зміни пріоритетів і вимог. Вона візуалізує всі етапи життєвого циклу у вигляді дошки із завданнями та забезпечує рівномірне навантаження для команд.

Для реалізації Kanban буде використовуватись інструмент Trello, який надає зручний інтерфейс для візуалізації дошки завдань, співпраці в команді та відстеження прогресу [7].

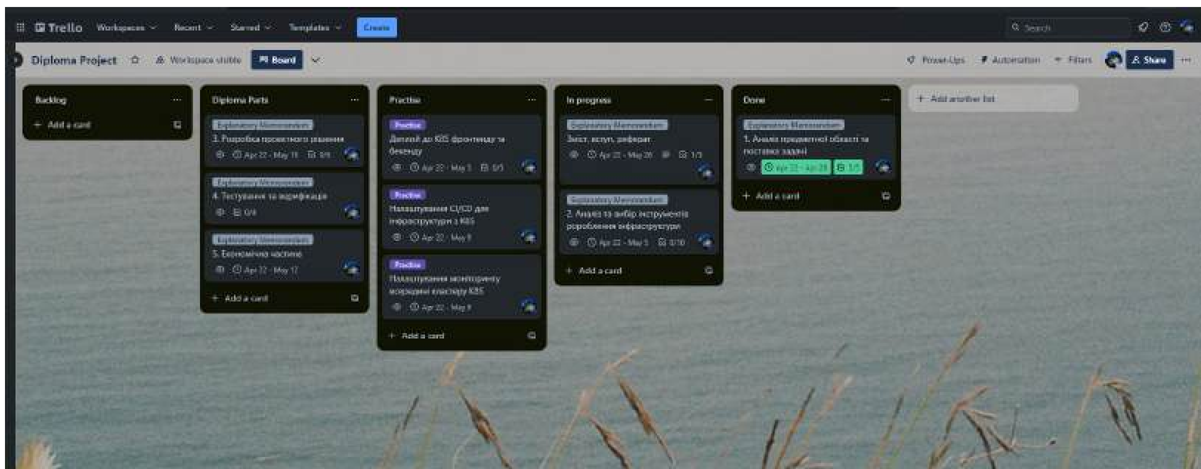

Рисунок 2.1 – Сторінка Trello з дошкою завдань проєкту.

2.4 Вибір програмного забезпечення для створення інфраструктури

В рамках DevOps підходу виникає потреба у гнучкій та масштабованій інфраструктурі, яку можна швидко створювати та оновлювати відповідно до потреб додатків. Для цього використовуються хмарні платформи та інструменти автоматизації інфраструктури як коду – IaC [8].

У даному проєкті було обрано Google Cloud Platform як хмарну платформу завдяки безкоштовному пробному періоду, зручності опису інфраструктури, низькій латентності та інтеграції з DevOps інструментами.

Для автоматизованого керування хмарними ресурсами обрано інструмент IaC Terraform через його універсальність, декларативний підхід, можливість співпраці та активну спільноту [9]. Конфігурація Terraform описується мовою HCL – зручним для читання нотаційним синтаксисом.

Поєднання GCP та Terraform забезпечить необхідну основу для швидкого та автоматизованого створення масштабованої інфраструктури в рамках DevOps.

2.5 Вибір програмного забезпечення для розгортання ПЗ на інфраструктурі

Для ефективного розгортання та управління додатками в складній хмарній інфраструктурі використовують підхід контейнеризації та оркестрації контейнерів, який забезпечує ізоляцію, масштабованість, портативність та стандартизацію робочого середовища додатків.

Контейнеризація дозволяє запакувати застосунок з усіма його залежностями в окремий контейнер. Популярним рішенням є Docker – платформа з відкритим кодом. Конфігурація Docker-образів пишеться у форматі Dockerfile, який використовує власний синтаксис та інструкції для визначення етапів створення образу [10]. Створені Docker образи додатків зберігаються в репозиторії Docker Hub.

Для розгортання, масштабування, балансування навантаження та відновлення після збоїв використовуються системи оркестрації контейнерів, такі як Kubernetes, Docker Swarm, Apache Mesos.

Для цього проєкту обрано Kubernetes як потужне та гнучке рішення для оркестрації. Причини: висока відмовостійкість, автоматичне масштабування, велика спільнота та підтримка провайдерів, можливості моніторингу. Конфігурація Kubernetes пишеться мовою YAML – стислим форматом серіалізації даних, зручним для читання та запису людиною [11]. YAML використовується для визначення ресурсів Kubernetes, таких як deployment, service, ingress та інших.

Поєднання Docker для контейнеризації та Kubernetes для оркестрації забезпечить надійне та гнучке розгортання ПЗ на створеній інфраструктурі.

2.6 Визначення застосунку для розгортання

У рамках цього проєкту необхідно розгорнути веб-застосунок «Proved Code» для демонстрації можливостей створеної інфраструктури та DevOps процесів [12]. «Proved Code» – це тестовий проєкт, який складається з frontend та backend частин, які будуть запаковані в Docker-контейнери та розгорнуті в кластері Kubernetes.

Фронтенд застосунку написаний на JavaScript з використанням бібліотеки React [13]. Для збірки фронтенду необхідне середовище Node.js та пакетний менеджер npm. Зібрані файли фронтенду будуть розміщені на веб-сервері Nginx.

Бекенд частина Proved Code реалізована на мові програмування Java. Для збірки та розгортання бекенду потрібно мати встановлену Java Development Kit [14].

Бекенд-застосунок взаємодіє з базою даних PostgreSQL для зберігання даних користувачів та інформації [15]. Також бекенд потребує сховища об’єктів AWS S3 для зберігання завантажених зображень [16].

Отже, для розгортання застосунку Proved Code необхідно підготувати наступне програмне забезпечення:

– Node.js та npm для збірки фронтенду;
– Java JDK для збірки бекенду;
– веб-сервер Nginx для розміщення фронтенду;
– база даних PostgreSQL;
– сховище AWS S3.

Така архітектура була визначена розробниками застосунку Proved Code. Як DevOps інженер, необхідно забезпечити розгортання та функціонування всіх компонентів у створеній інфраструктурі з використанням контейнеризації та оркестрації Kubernetes.

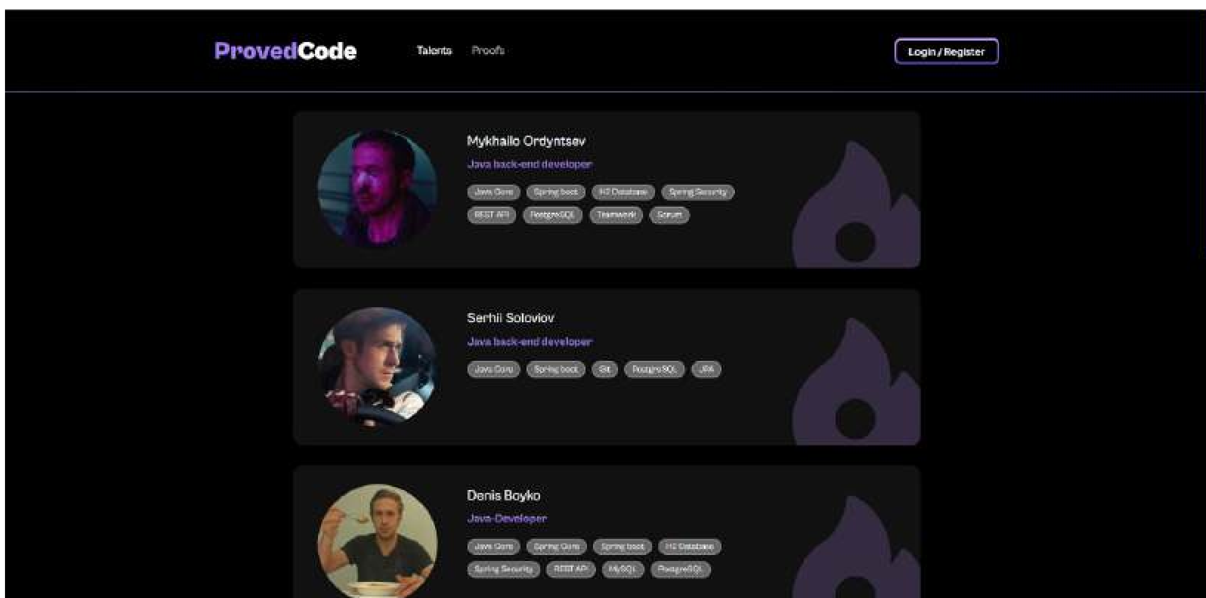

Рисунок 2.2 – Вигляд сторінки застосунку.

2.7 Вибір програмного забезпечення для моніторингу та візуалізації

У контексті DevOps та автоматизованої інфраструктури критично важливим аспектом є моніторинг стану та продуктивності всіх її компонентів: хмарних ресурсів, операційних систем, контейнерів, додатків тощо. Це дозволяє виявляти проблеми, прогнозувати пікові навантаження та вживати превентивних заходів.

Для збору, візуалізації та аналізу метрик від різних джерел використовуються спеціалізовані системи моніторингу.

Найбільш популярними рішеннями є:

– Prometheus - відкрита система моніторингу та оповіщень, орієнтована на збір та обробку числових метрик у реальному часі, вона має потужну мову запитів, візуалізацію графіків та гнучку систему сповіщень;

– Grafana – аналітична веб-платформа для візуалізації метрик та логів з різних джерел даних, включно з Prometheus, надає зручний інтерфейс для створення інформативних дашбордів;

– Elasticsearch, Logstash, Kibana – набір інструментів для централізованого збору, індексації, пошуку та візуалізації логів з різних джерел;

– InfluxDB – спеціалізована система управління часовими рядами метрик для моніторингу та аналітики додатків у реальному часі.

Для забезпечення всебічного моніторингу в цьому проєкті буде використано комбінацію Prometheus, Grafana:

– Prometheus як основна система збору та обробки метрик від хмарних ресурсів, Kubernetes, додатків та експортерів метрик;

– Grafana для створення інформативних дашбордів із візуалізацією зібраних Prometheus метрик.

Такий стек моніторингу є гнучким, масштабованим та забезпечує всеосяжний нагляд за станом і продуктивністю розгорнутої інфраструктури та додатків в режимі реального часу [18].

Для збору метрик від різних джерел (вузлів, контейнерів, додатків) будуть використовуватись відповідні експортери та агенти, такі як Telegraf, Node Exporter, cAdvisor тощо.

Telegraf – універсальний агент збору метрик та даних, сумісний з різними системами моніторингу, включно з Prometheus [17]. Він має модульну структуру та дозволяє гнучко налаштовувати типи даних для збору через вхідні та вихідні плагіни.

У цьому проєкті Telegraf буде використовуватись для:

– збору метрик з вузлів Kubernetes для моніторингу кластера;

– збору метрик з фронтенд-серверу Nginx для моніторингу продуктивності застосунку.

Налаштування Telegraf як універсального агента забезпечить всеохопний збір необхідних метрик та даних для комплексного моніторингу стану інфраструктури та додатків.

Модульна архітектура Telegraf дозволяє легко додавати чи видаляти потрібні плагіни збору даних без змін основного коду агента.

Тому, у поєднанні з Prometheus, Grafana та Telegraf стане ключовим елементом зручної та гнучкої системи моніторингу створеної DevOps інфраструктури.

2.8 Вибір інструментального засобу безперервної інтеграції коду

Для забезпечення ефективної співпраці в команді, автоматизації процесів збирання, тестування та безпечного розгортання нових версій додатків використовуються інструменти безперервної інтеграції та безперервного розгортання CI/CD.

Провідними рішеннями у цій сфері є:

– Jenkins – одна з найпопулярніших відкритих платформ для автоматизації процесів збирання, тестування та розгортання програмного забезпечення.

– GitLab CI/CD – вбудований функціонал безперервної інтеграції/безперервного розгортання в Git-репозиторії GitLab.

– Travis CI – розподілена платформа безперервної інтеграції як сервісу CI-as-a-Service.

– CircleCI – хмарна платформа для автоматизації процесів CI/CD з підтримкою Docker.

Для цього проєкту було обрано GitLab CI/CD як інтегроване середовище для Git-репозиторіїв, налаштувань безперервної інтеграції та розгортання додатків.

Причини вибору:

– наявність вбудованих можливостей CI/CD без необхідності інтеграції із зовнішніми сервісами;

– зручність керування репозиторіями коду та конфігураціями CI/CD в єдиному інтерфейсі;

– підтримка Docker для збирання, тестування та упакування додатків у контейнери;

– безкоштовний тарифний план для невеликих команд та відкритих проєктів;

– активна підтримка та розвиток GitLab спільнотою.

GitLab CI/CD забезпечить автоматизацію процесів розробки, тестування, контейнеризації та безпечного розгортання додатків на створеній інфраструктурі за допомогою визначених конвеєрів.

Висновки за розділом

У цьому розділі було детально розглянуто та обґрунтовано вибір інструментів та технологічних рішень для створення масштабованої та автоматизованої DevOps інфраструктури на основі хмарного провайдера GCP.

Повна картина використаних компонентів має такий вигляд:

– хмарна платформа: Google Cloud Platform (GCP)
– система управління інфраструктурою: Terraform
– контейнеризація: Docker
– оркестрація контейнерів: Kubernetes
– тестовий застосунок: Nginx та OpenJDK
– моніторинг: Prometheus, Grafana
– збір метрик та логів: Telegraf
– безперервна інтеграція: GitLab CI/CD
– методологія розробки: Kanban з Trello

Детальний опис кожного компонента та обґрунтування його вибору дозволить забезпечити розуміння загальної архітектури системи та ролі кожного елемента у процесах розробки, тестування, розгортання та моніторингу додатків на створеній інфраструктурі.

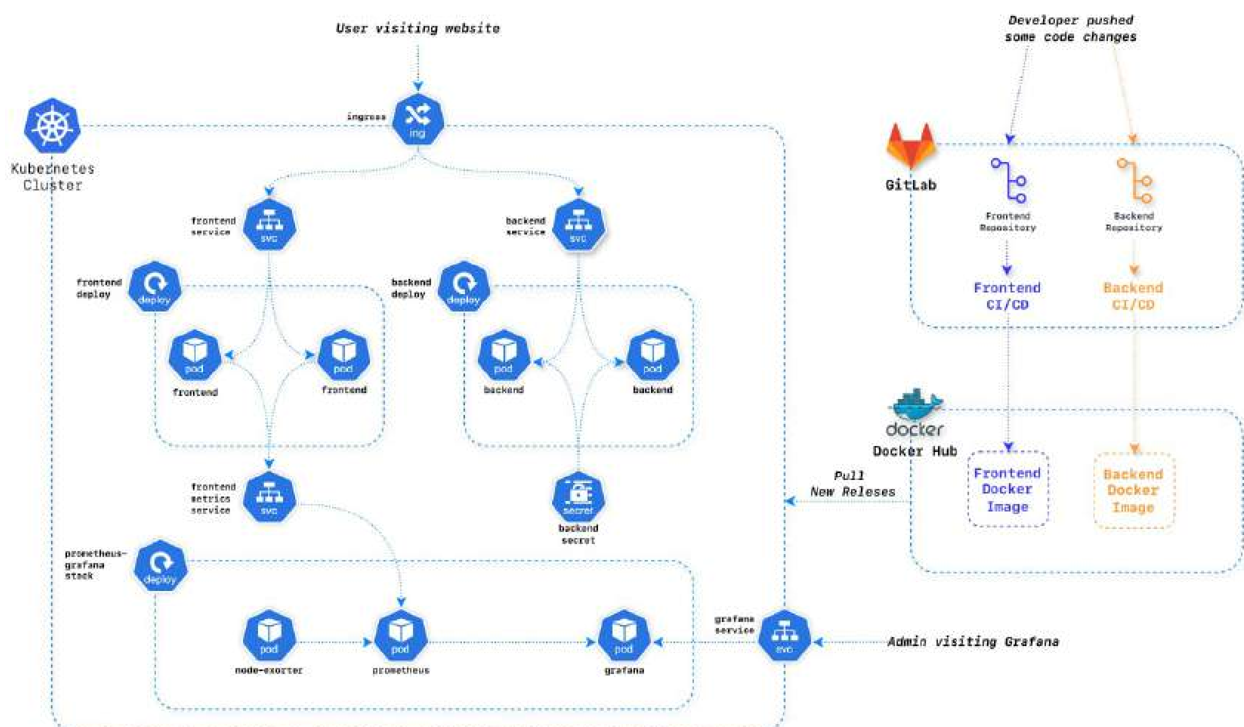

Рисунок 2.3 – Схема інфраструктури проєкту.

3 РОЗРОБКА ПРОЕКТНОГО РІШЕННЯ

Для успішної реалізації складної та масштабованої інфраструктури, яка передбачає використання різноманітних технологій, таких як контейнеризація, оркестрація контейнерів за допомогою Kubernetes, хмарні обчислення, системи безперервної інтеграції та доставки, а також моніторинг, необхідно дотримуватися поступового та ретельного підходу.

Замість того, щоб одразу переходити до розгортання повної інфраструктури у хмарному середовищі з використанням Kubernetes, буде розпочато з простіших локальних розгортань, що забезпечує можливість закласти фундамент у розумінні того, як працюють складові інфраструктури, необхідних для опанування складніших компонентів системи:

1) на першому етапі буде створена локальна інфраструктура з використанням гіпервізора, де будуть розгорнуті окремі екземпляри фронтенду і бекенду додатка, це створює умови отримати основні концепції та практичний досвід роботи з віртуальними машинами та базовим розгортанням веб-додатків;
2) другий етап передбачає контейнеризацію додатків за допомогою Docker і розгортання контейнерів у хмарному провайдері Google Cloud Platform з використанням Terraform для керування інфраструктурними ресурсами. Контейнеризований застосунок буде розгорнуто на хмарних інстансах GCE;
3) далі буде створено кластер Kubernetes у GCP, у кластері буде розгорнуто контейнеризований застосунок для декількох середовищ, що дозволить отримати переваги оркестрації контейнерів і автоматичного масштабування додатків;
4) після розгортання в Kubernetes буде побудована система безперервної інтеграції та розгортання CI/CD за допомогою GitLab, що забезпечує автоматизацію процесів збірки, тестування та розгортання додатків;
5) останнім етапом стане налаштування системи моніторингу за допомогою Prometheus, Grafana та Telegraf для відстеження працездатності додатків, кластера Kubernetes.

Такий поетапний підхід дозволить поступово збільшити складність проєкту та отримати практичні навички роботи з різними інструментами та технологіями, створеними в DevOps.

3.1 Реалізація локальної інфраструктури та розгортання застосунку

Завдання полягало у побудові локальної інфраструктури на віртуальних машинах для розгортання бекенду і фронтенду застосунку. Фронтенд мав бути доступним через веб-браузер на порту 80, робити запит до бекенду на порту 8080, а бекенд - до бази даних PostgreSQL та сховища S3 від AWS.

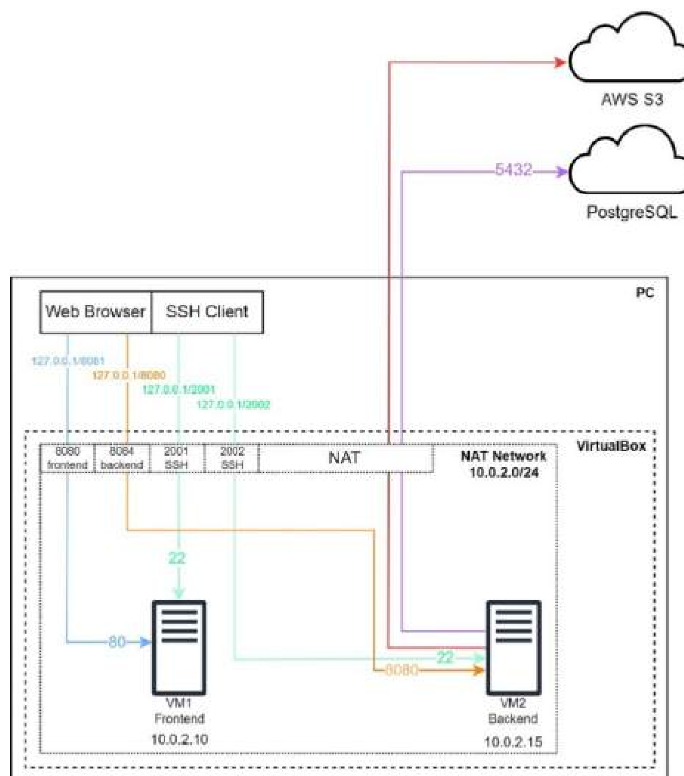

Рисунок 3.1 – Вигляд локальної інфраструктури.

Необхідно було виконати такі кроки:

1. налаштування віртуальної машини на Oracle VirtualBox, створення клонів VM і віртуальної мережі;
2. встановлення PostgreSQL, створення сховища S3 і користувача доступу;
3. встановлення Java і збірка бекенд-застосунку;

4. встановлення Nginx і збірка фронтенд-застосунку;
5. перевірка роботи сайту.

3.1.1 Налаштування віртуальної машини на Oracle VirtualBox

Для створення базової віртуальної машини була обрана Ubuntu 22.04. Створені два повні клони цієї машини для фронтенду та бекенду. Вони були поміщені у спільну віртуальну мережу з різними IP-адресами. Налаштовано перекидання портів для зручного доступу через SSH.

Для зручної роботи з віртуальними машинами через SSH-термінал були відкриті наступні порти:

1. 2001:22 – порт 2001 на хості було перекинуто на 10.0.2.10:22 для SSH доступу до веб-сервера;
2. 2002:22 – порт 2002 на хості було перекинуто на 10.0.2.15:22 для SSH доступу до бекенду.

Також були відкриті порти для доступу до веб-сервера і застосунку бекенду:

1. 8081:80 – порт 8081 на хості перекинуто на 10.0.2.10:80 для доступу до Nginx;
2. 8080:8080 – порт 8080 на хості перекинуто на 10.0.2.15:8080 для доступу до Java застосунку бекенду.

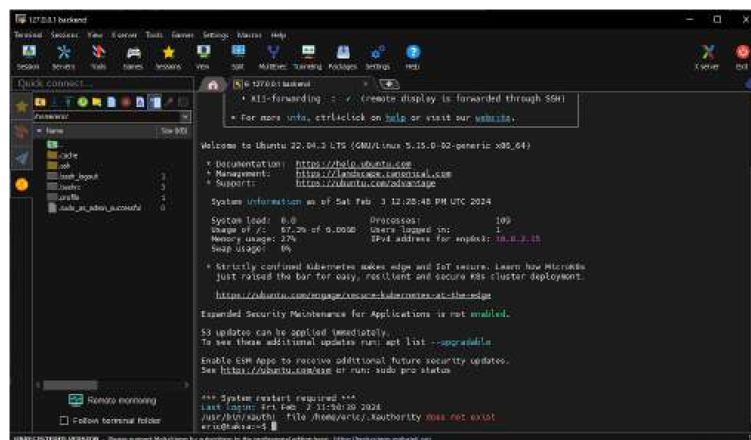

Рисунок 3.2 – Вхід до другої віртуальної машини через SSH.

Далі необхідно було визначити порядок запиту: спочатку робиться запит фронтенду, далі фронтенд до бекенду, після чого робиться запит до

бази даних та сховища S3 з бекенду. Тому наступним етапом стало створення бази даних та S3 сховища на AWS.

3.1.2 Налаштування PostgreSQL та S3

PostgreSQL було розгорнуто як Docker-контейнер на хмарному сервері Oracle Cloud. Створено сховище S3 на AWS, користувача з правами доступу та згенеровано ключі доступу.

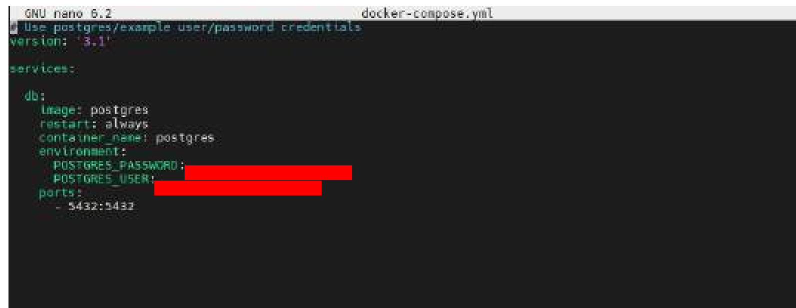

Рисунок 3.3 – docker-compose файл для роботи бази даних PostgreSQL.

Для зберігання зображень сайту використовувалося хмарне сховище S3 на AWS. Для його створення були виконані наступні дії:

1. в AWS створюється сховище S3;
2. створюється користувач з правами читання і запису в сховище;
3. для користувача, яким буде користуватись бекенд-сервер, генеруються і зберігаються ключі доступу до сховища S3 – Access Key та Secret Key.

3.1.3 Встановлення Java і збірка бекенду

На віртуальній машині бекенду встановлено Java, клоновано репозиторій бекенду, встановлено необхідні змінні середовища. Виконано збірку бекенду та налаштовано його як systemd-сервіс для автоматичного запуску.

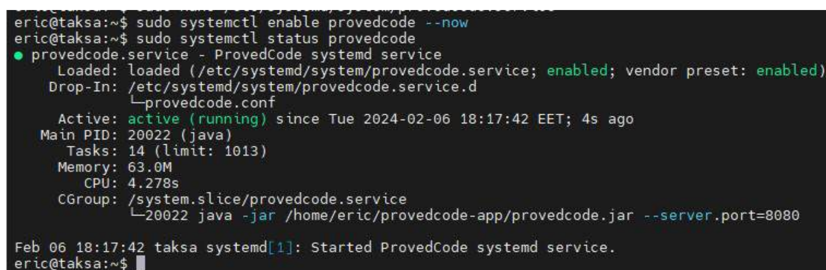

Рисунок 3.4 – Успішний статус виконання сервісу «provedcode.service».

3.1.4 Установка Nginx та збірка фронтенду

На машині фронтенду встановлено Nginx, Node.js, клоновано репозиторій фронтенду. Виконано збірку фронтенду, відредаговано вихідний код для правильного посилання на бекенд. Зібрані файли скопійовано до каталогу Nginx.

```
eric@taksa:~/frontend$ chmod -R 755 ./build/*
eric@taksa:~/frontend$ sudo rm -rf /var/www/html/*
[sudo] password for eric:
eric@taksa:~/frontend$ sudo mv ./build/* /var/www/html/
eric@taksa:~/frontend$ ll /var/www/html/
total 28
drwxr-xr-x 3 root root 4096 Feb  6 17:26 ./
drwxr-xr-x 3 root root 4096 Feb  2 15:49 ../
-rwxr-xr-x 1 eric eric 4181 Feb  6 17:14 asset-manifest.json*
-rwxr-xr-x 1 eric eric 1066 Feb  6 17:11 icon.svg*
-rwxr-xr-x 1 eric eric  405 Feb  6 17:14 index.html*
drwxr-xr-x 5 eric eric 4096 Feb  6 17:14 static/
eric@taksa:~/frontend$
```

Рисунок 3.5 – Файли збірки фронтенду перенесені до каталогу, звідки Nginx їх буде зчитувати.

3.1.5 Перевірка роботи застосунку

Перевірено доступність фронтенду у браузері та коректну роботу бекенду при отриманні даних про учасників проєкту.

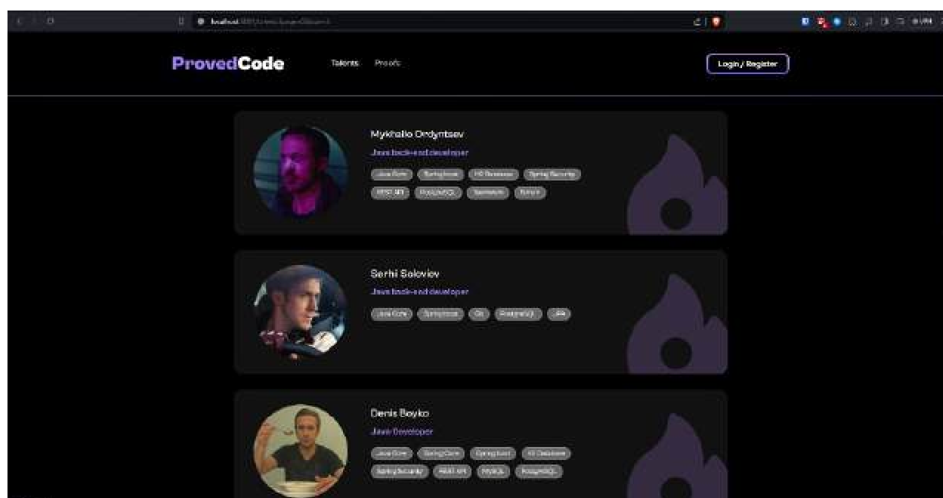

Рисунок 3.6 – Вигляд стартової сторінки Talents застосунку Proved Code.

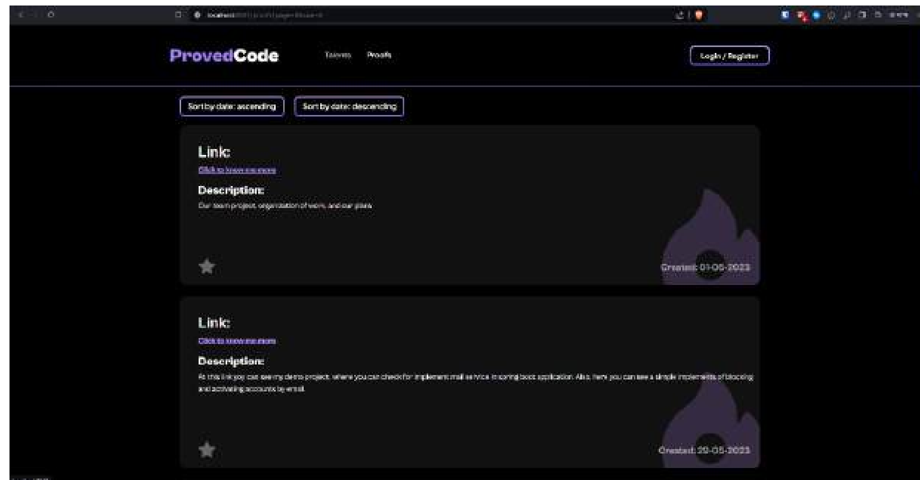

Рисунок 3.7 – Сторінка Proofs застосунку Proved Code.

Підсумовуючи, відображення головної сторінки застосунку за адресою віртуальної машини вказує на успішне розгортання всієї інфраструктури. Тепер відомо як робити розгортання фронтенд і бекенд частини застосунку, які для цього потрібні інструменти. Подальшим покращенням інфраструктури буде її контейнеризація фронтенду та бекенду.

3.2 Розгортання інфраструктури у хмарі з контейнеризованим додатком

Завданням було створити інфраструктуру на сервері за допомогою Docker, розмістивши сервіси фронтенду та бекенду в окремих контейнерах. Для цього необхідно виконати наступні кроки:

– обрати, налаштувати та розгорнути сервер у хмарному провайдері;
– встановити Docker на сервер;
– описати Dockerfile для бекенду додатка;
– описати Dockerfile для фронтенду додатка;
– описати Docker Compose для розгортання бекенду та фронтенду застосунку на сервері;
– перевірити працездатність застосунку.

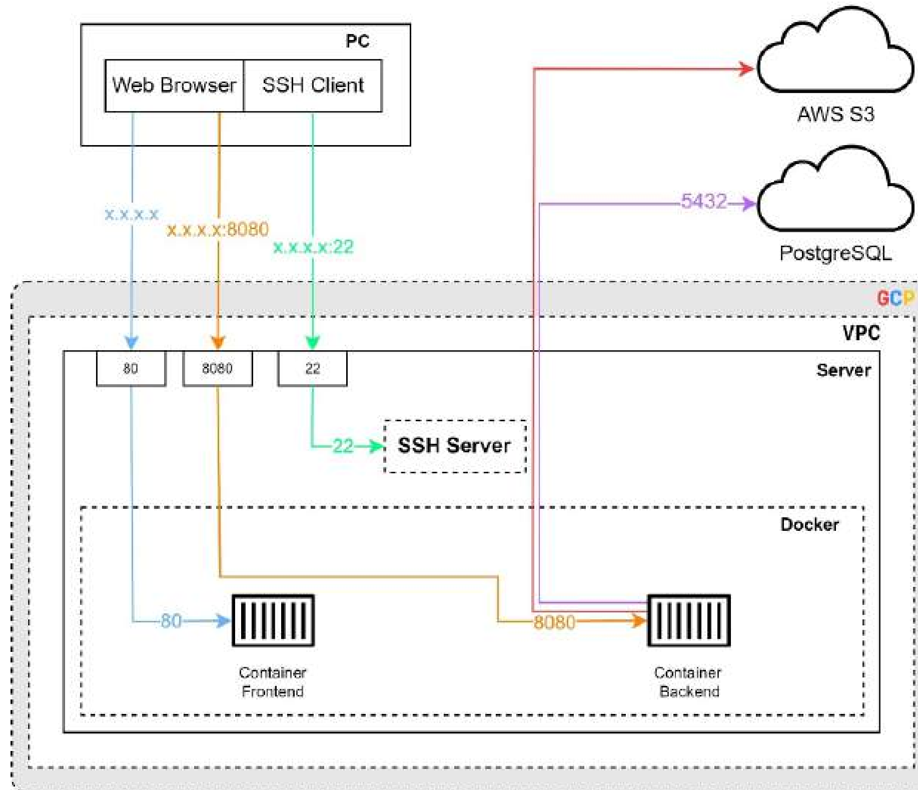

Рисунок 3.8 – Структура інфраструктури з сервером на GCP та контейнерами Docker для фронтенду і бекенду додатка.

Перед встановленням Docker потрібно визначити, які ресурси будуть використовуватися контейнерами додатка. З урахуванням досвіду зі збіркою фронтенду, для цього процесу потрібно приблизно 2 ГБ оперативної пам’яті. Отже, використовуватиметься виділений сервер у хмарного провайдера з щонайменше 1 vCPU, 2 ГБ оперативної пам’яті та 8 ГБ місця на жорсткому диску, що працює під управлінням ОС Debian.

3.2.1 Створення сервера у хмарі

Інстанс – це екземпляр віртуальної машини, створений та керований через хмарні сервіси чи платформи. Для створення віртуальної машини буде використано сервіс Google Compute Engine, у ньому можна створювати інстанси, налаштовувати їхні характеристики, мережеві параметри, стартові скрипти, логи тощо.

Враховуючи вимоги до ресурсів для збирання та запуску програми, тип інстансу e2-small є відповідним вибором. Цей тип інстансу має такі

характеристики: 1 vCPU та 2 ГБ оперативної пам'яті, що задовольнить потреби застосунку.

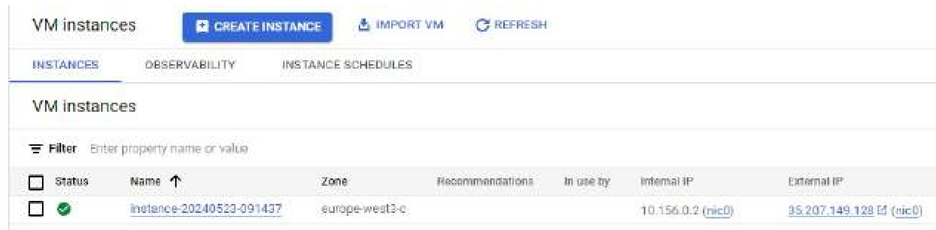

Рисунок 3.9 – Було створено інстанс, тепер він має публічну IP-адресу.

3.2.2 Встановлення Docker на сервер

Команди для встановлення Docker потрібно брати виключно з офіційного сайту, оскільки там міститься ключ для безпечного завантаження інсталяційних файлів. Скрипт «**install_docker.sh»** встановлює Docker у ОС. Перевірити, що Docker працює, достатньо через команду «sudo docker ps», що показує поточні контейнери.

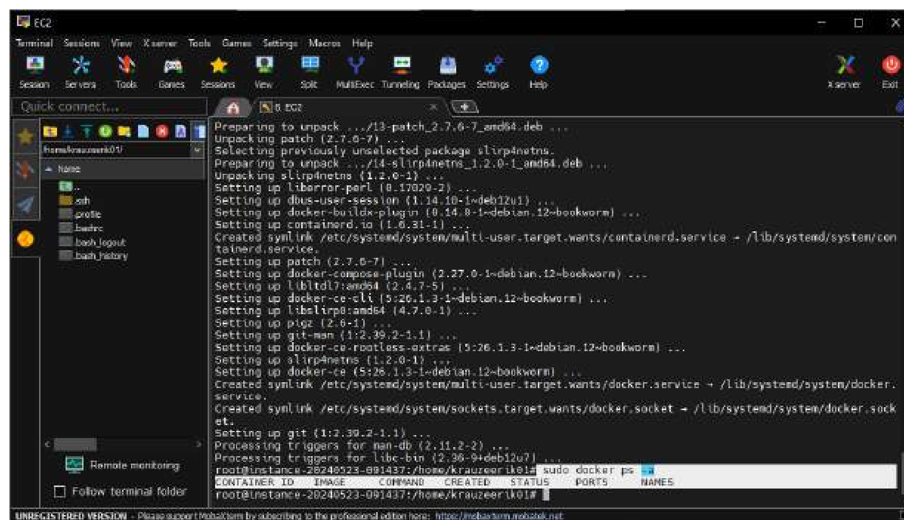

Рисунок 3.10 – Перевірка того, що Docker встановлено і у ньому немає працюючих контейнерів.

3.2.3 Опис Dockerfile для бекенду додатка

Спочатку треба створити Dockerfile для образу, в якому буде здійснюватись збірка вихідного коду разом із вказаними змінними середовища. Важливим питанням було, які використовувати образи для збірки на Java. Eclipse-temurin використовується замість open-jdk через те, що останній застарів, хоча раніше він був популярним рішенням.

Важливо знати тип архітектури процесора сервера, де встановлений Docker. Наприклад, багато образів не працюють на arm, оскільки архітектура на цей менш розповсюджена. На архітектуру amd64 розраховано більшість образів із Docker Hub. Образи на базі дистрибутива Alpine відрізняються досить невеликим розміром, тому зручні для використання.

Під час виконання Dockerfile був встановлений git, оскільки в дистрибутиві Alpine він не встановлений за замовчуванням. Далі створюється клон вихідного коду бекенду, обирається гілка dev, встановлюються змінні оточення і відбувається збірка.

У Dockerfile використано Multi-Stage Builds, де для збірки застосунку на Java використовується тимчасове середовище eclipse-temurin:17-jdk-alpine. Це середовище призначене лише для інструментів збірки і вихідного коду, таких як компілятор Java, стандартні бібліотеки класів Java, приклади, документація, утиліти і виконавча система Java.

Після завершення збірки файли переносяться до іншого середовища - eclipse-temurin:17-jre. Це середовище призначене лише для запуску зібраного додатка на Java і не може здійснювати збірку. Воно займає менше місця в пам'яті, оскільки представляє лише мінімальну реалізацію віртуальної машини, необхідну для виконання Java-додатків. У нього переносяться файли збірки з минулого середовища «backend-build» через «COPY --from=backend-build /app/backend/target/*.jar /app/provedcode.jar». Після завершення збірки середовище «backend-build» або eclipse-temurin:17-jre очищається.

З цього випливає, що якщо образ був би зібраний без перемикання на середовище FROM eclipse-temurin:17-jre, він важив би приблизно 550 МБ, у той час як з використанням цього підходу вага образу становила близько 330 МБ. Створення образу бекенду додатка описано у «**Dokerfile-backend»**.

3.2.3 Опис Dockerfile для фронтенду додатка

Тут також використовувався підхід Multi-Stage Builds, node:21-alpine або frontend-build – для збірки на Node.js, а nginx:latest – для збереження файлів збірки в директорії для відображення застосунку в Nginx. В результаті виходить образ контейнера розміром близько 200 МБ.

Тут, окрім git, встановлюється утиліта curl, оскільки була необхідність отримати адресу сервера для налаштування змінної backend-server та виправлення файлу фронтенду застосунку. Після цього за допомогою регулярного виразу змінюється неправильне значення IP-адреси.

Створення образу фронтенду додатка описано у «**Dokerfile-frontend»**.

3.2.4 Опис Docker Compose та перевірка роботи застосунку

Для зручності управління контейнерами фронтенду і бекенду додатка був використаний Docker Compose.

Docker Compose – це інструмент, що спрощує процес налаштування та керування багатоконтейнерними Docker-додатками. За допомогою YAML-файлу визначається конфігурація сервісів, мереж та томів. Це дозволяє запускати всі сервіси однією командою «docker-compose up», автоматизуючи їх створення та налаштування, що значно полегшує розробку, тестування та розгортання додатків.

У файлі docker-compose.yml були описані обидва сервіси: «frontend» і «backend». Кожен сервіс описано в своїх Dockerfile і кожен розміщений у своєму підкаталозі. Тобто docker-compose.yml повинен знаходитись на рівень вище за каталоги з Dockerfile.

У випадку, якщо застосунок у контейнері стане неактивним, контейнер буде перезапущено завдяки налаштуванню «restart: always» у файлі Docker Compose. У Docker Compose можна задавати змінні оточення у окремий файл через «env_file:», в результаті файл стає більш зручним для читання. Для кожного контейнера сервісу потрібно задати перекидання портів, інакше з контейнерами не можна буде зв’язатися через Інтернет. В кінці налаштовується параметр «depends_on», де вказується, що фронтенд-сервіс залежить від бекенд-сервісу. Це означає, що бекенд-сервіс буде запущено перед фронтенд-сервісом і лише після цього останній буде запущено. Описання Docker Compose знаходиться у файлі «**docker-compose-provedcode»**.

Спочатку потрібно запустити Docker Compose у неінтерактивному режимі, щоб під час збирання і розгортання бекенд-сервісу і фронтенд-сервісу можна було бачити логи у реальному часі. Це дозволяє оперативно виявити можливі помилки.

Якщо Docker Compose працює належним чином, його треба запустити його у фоновому режимі, використовуючи аргумент «-d» у команді.

Рисунок 3.11 – Сервіси бекенду та фронтенду були вдало розгорнуті через Docker Compose.

Рисунок 3.12 – Переключення Docker Compose у фоновий режим.

Після запуску всіх контейнерів необхідно перевірити роботу сайту. Оскільки всі контейнери працюють на одному сервері, запити на отримання списку користувачів та реєстрацію повинні відправлятися на цей сервер через порт 8080. Крім того, встановлення світлини акаунту у своїй обліковці на сайті має працювати, якщо сховище S3 і база даних налаштовані правильно.

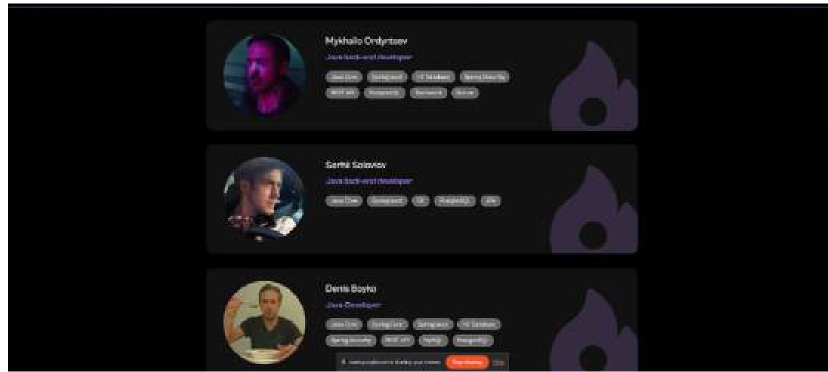

Рисунок 3.13 – За IP-адресою інстансу можна потратили на сторінку сайту Proved Pode.

Отже, було розглянуто створення та налаштування інстансів певного типу з визначеною обсягом пам’яті, SSH-ключем і налаштуванням файрвола в мережі інстанса. Також розглянуто роботу з Dockerfile через Docker Compose для фронтенда та бекенда. Тепер можна покращити інфраструктуру, створивши хмарну інфраструктуру за допомогою Terraform, що автоматично створює інстанси з необхідними програмами.

3.3 Автоматичне розгортання інфраструктури через код у хмарі з контейнеризованим додатком

Після того, як було розглянуто створення інстансів у GCP і розгортання додатків через Docker, наступним кроком є автоматизація цих процесів за допомогою Terraform. Використання Terraform дозволяє створити нову інфраструктуру не вручну, а автоматично, що значно спрощує та пришвидшує цей процес.

Було прийнято рішення, що кожен сервіс буде працювати на окремій віртуальній машині з встановленим Docker. Це дозволяє розподілити навантаження між різними інстансами. Наприклад, якщо потрібно обробити більше запитів до бекенду, вони не повинні впливати на інстанс із фронтендом. Terraform дозволяє створювати або швидко відновлювати хмарну інфраструктуру у випадку збою.

Отже, для розгортання додатка інфраструктура буде створюватись у такому порядку:

1. першим кроком буде створення віртуальної приватної мережі VPC, яка забезпечить середовище для інстансів;
2. до мережі буде додано фаєрвол для захисту та контролю трафіку між бекендом і фронтендом;

3. будуть створені окремі віртуальні машини для кожного сервісу: одна для бекенду і одна для фронтенду;

4. на віртуальних машинах будуть виконані стартові скрипти, які встановлюють Docker;

5. після встановлення Docker на одній машині буде запущено у контейнері бекенд, а на іншій – фронтенд.

Після цього можна буде перейти на сторінку за IP-адресою інстанса з фронтендом і перевірити роботу застосунку. Використання Terraform для автоматизації створення та налаштування інфраструктури забезпечує гнучкість, надійність та швидкість розгортання додатків.

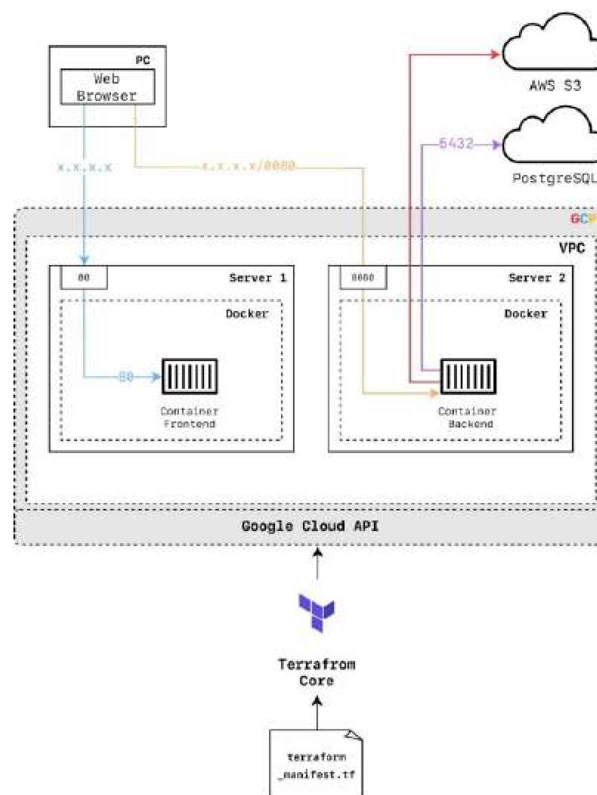

Рисунок 3.14 – Схема розгортання інфраструктури за допомогою Terraform з контейнеризованим бекендом та фронтендом на двох окремих інстансах.

3.3.1 Початкова активація API сервісів GCP для Terraform

Для того, щоб Terraform міг використовувати ресурси провайдера GCP, необхідно активувати API деяких сервісів. Перш за все, потрібно мати Compute Engine API для роботи з віртуальними машинами, який раніше було вже активовано. Наступним кроком є активація IAM API, що

забезпечить доступ Terraform до сервісного аккаунту, який матиме права створювати інфраструктуру.

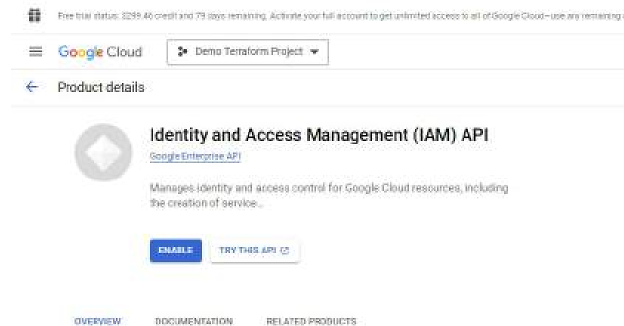

Рисунок 3.15 – Сторінка активації сервісу IAM.

3.3.2 Створення користувача для взаємодії з ресурсами GCP

Для того, щоб Terraform міг використовувати ресурси провайдера GCP, необхідно активувати API деяких сервісів. Перш за все, потрібно мати Compute Engine API для роботи з віртуальними машинами, який раніше було вже активовано. Наступним кроком є активація IAM API, що забезпечить доступ Terraform до сервісного аккаунту, який матиме права створювати інфраструктуру. Далі потрібно створити сервісний аккаунт. Для цього потрібно перейти до сервісу IAM & Admin і обрати вкладку Service Accounts. Там додається новий сервісний аккаунт. Далі надаються необхідні права цьому аккаунту, щоб він мав дозвіл на використання сервісів. У списку сервісних аккаунтів повинен з'явитися аккаунт для Terraform. Щоб Terraform мав доступ через цей аккаунт, йому необхідні дані облікового запису. Для цього використовується спеціальний ключ, який можна згенерувати у вкладці KEYS цього аккаунту. Для створення ключа обирається формат JSON. Ключовий файл облікового запису, який буде потрібний для Terraform-маніфесту буде збережено для подальшого використання.

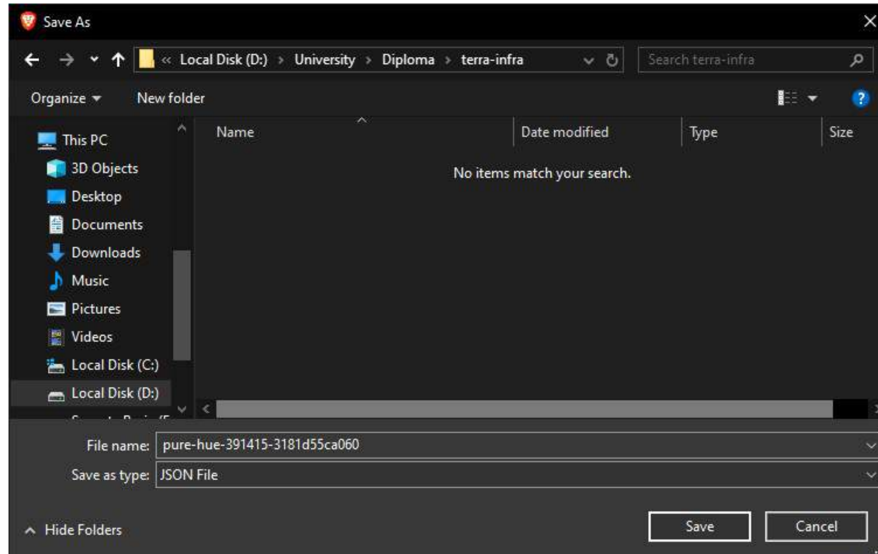

Рисунок 3.16 – Збереження секретного ключа сервісного акаунта.

3.3.3 Створення образу бекенду і нового Dockerfile для фронтенду

Щоб уникнути необхідності перезбирати застосунок кожного разу на новій машині та забезпечити його запуск у будь-якому середовищі, наприклад, у Docker або Kubernetes, потрібно використовувати образ застосунку. Цей образ може бути збережений у Docker Hub. Образ бекенду можна створити та зберегти, оскільки розробники передбачили функцію використання змінних оточення. Щодо фронтенд-застосунку, оскільки розробники не передбачили таку функцію, його доведеться перезбирати щоразу при зміні адреси бекенду.

Раніше було створено Dockerfile для створення Docker-образу бекенду додатка. Цей Dockerfile включає кроки клонування репозиторію, налаштування змінних оточення та збірку додатка. При збиранні образу важливо враховувати архітектуру процесора.

Перед тим як виконати push в сховище образів, потрібно авторизуватися в оболонці командою «sudo docker login». Лише після успішної авторизації можна зберігати образ у сховище. Після збереження необхідно перевірити, чи є він доступним в обліковому записі на Docker Hub.

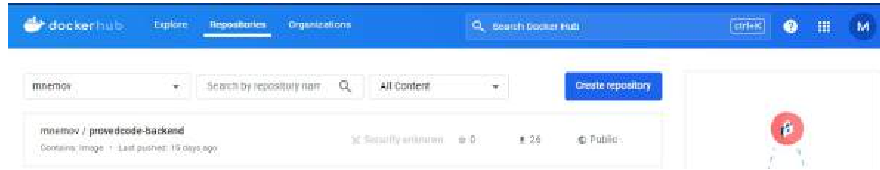

Рисунок 3.17 – Збережений образ бекенду у репозиторії Docker Hub.

Для того, щоб можна було змінювати значення адреси бекенд-серверу, довелось зробити розгалуження репозиторію з фронтендом, і змінити вихідний код призначення константи адресу.

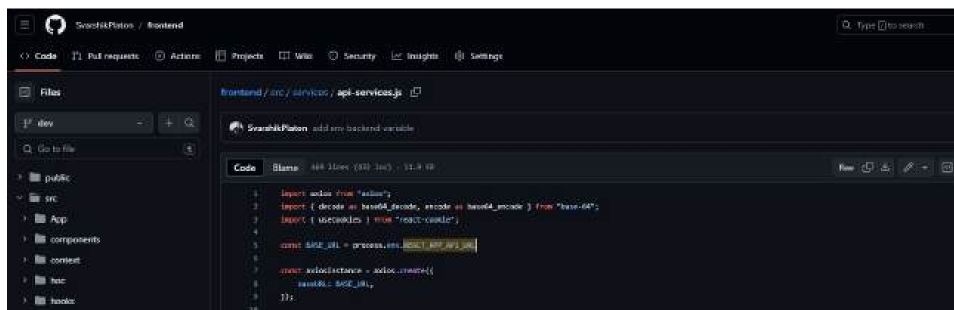

Рисунок 3.18 – Зміна вихідного коду у фронтенді, для завдання посилання на сервер бекенду при збірці.

Для зібрання образу фронтенду додатка потрібно клонувати репозиторій GitHub з його вихідним кодом на інстансі GCE з встановленим Docker. Під час збірки Dockerfile вихідний код буде скопійовано з репозиторію і зібрано. Після цього образ запустить Nginx для відображення зібраних файлів застосунку. Змінений Dockerfile образу фронтенду для Terraform описано у файлі «**Dockerfile-frontend-terraform».**

3.3.4 Встановлення Terraform

Для автоматизації створення та управління хмарною інфраструктурою використовується Terraform. Інсталяція Terraform виконується через скрипт «**install_terraform.sh**».

3.3.5 Структура маніфесту Terraform

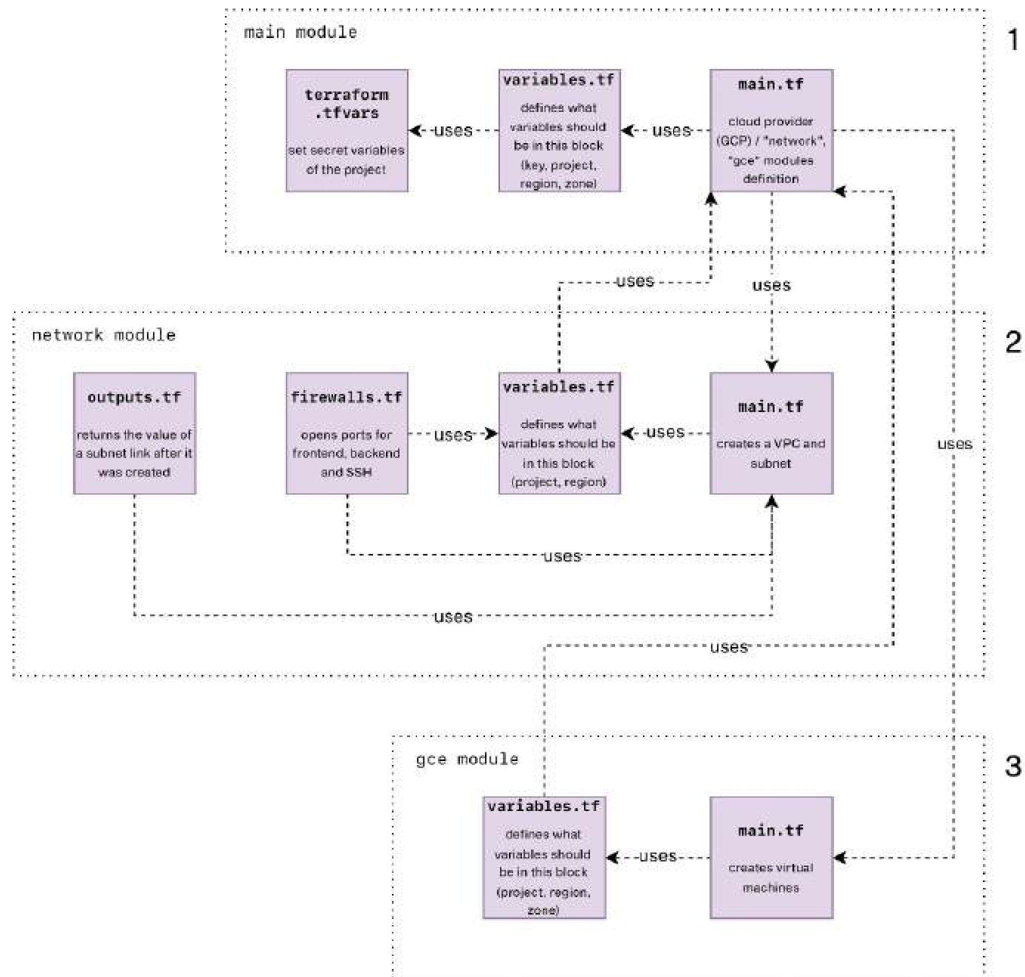

Рисунок 3.19 – Модульна структура маніфесту Terraform.

Маніфест Terraform є основним конфігураційним файлом, який визначає інфраструктуру, яку потрібно розгорнути. Він складається з різних модулів, кожен з яких відповідає за різні частини інфраструктури. Кожен модуль може включати різні файли, такі як **«main.tf»**, **«variables.tf»**, **«terraform.tfvars»**, а також інші файли з налаштуваннями і скриптами, які будуть виконані на створених ресурсах. Для поточного проєкту реалізовано три модулі.

Важливо зауважити, що крім основних конфігураційних файлів **«main.tf»**, **«variables.tf»**, **«terraform.tfvars»**, також присутні файли з різними скриптами, які будуть виконуватися на віртуальних машинах **«env.sh»**, **«install_docker.sh»**, **«run_backend.sh»**, **«run_frontend.sh»**. Кожен файл буде розглянуто з точки зору його використання.

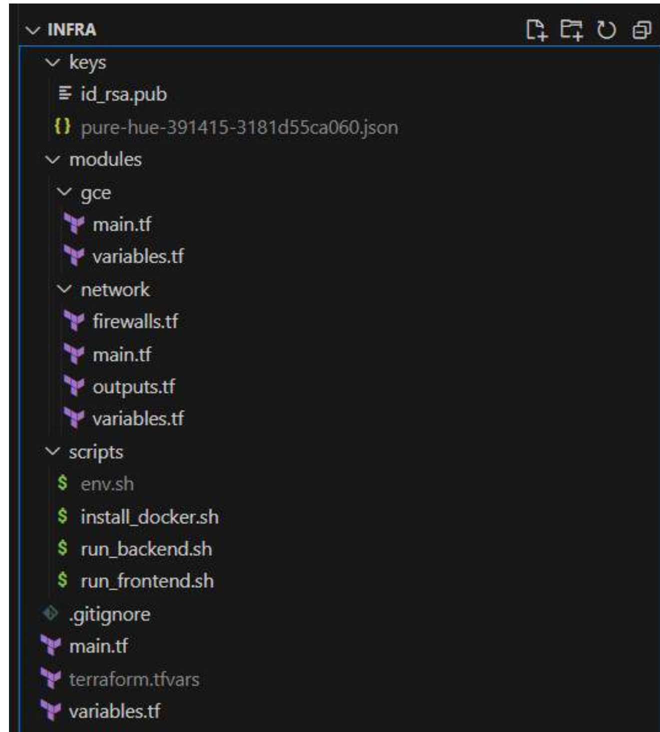

Рисунок 3.20 – Список файлів для розгортання інфраструктури через Terraform.

3.3.6 Основний модуль

Файл **«main.tf»** є основним конфігураційним файлом Terraform для проєкту. Він виконує наступні функції:

– налаштування провайдера Google Cloud Platform (GCP) – цей блок коду вказує Terraform використовувати GCP як провайдера хмарних послуг, тут вказуються облікові дані для аутентифікації, ідентифікатор проєкту та регіон за замовчуванням;

– модуль «network» – цей блок коду посилається на модуль «network», який знаходиться в каталозі ./modules/network, модуль відповідає за створення і налаштування мережевих ресурсів в GCP, таких як VPC та підмережі;

– модуль «gce» – цей блок коду посилається на модуль «gce», який знаходиться в каталозі ./modules/gce, модуль відповідає за створення і налаштування ресурсів Google Compute Engine, таких як віртуальні машини.

Файл «**variables.tf**» визначає змінні, які використовуються в проєкті Terraform:

– gcp_svc_key – ключ служби для Google Cloud Platform;
– gcp_project – ідентифікатор проєкту в Google Cloud;
– gcp_region – регіон в Google Cloud, де будуть створюватися ресурси;
– gcp_zone – зона у вказаному регіоні.

Файл «**terraform.tfvars»** містить значення змінних для цього проєкту Terraform:
– gcp_project – ідентифікатор проєкту в Google Cloud;
– gcp_svc_key – шлях до файлу ключа служби Google Cloud;
– gcp_region – регіон Google Cloud для створення ресурсів;
– gcp_zone – зона у вказаному регіоні.

3.3.7 Модуль мережі

Файл «**main.tf»** визначає мережеві ресурси в Google Cloud:
– google_compute_network.main – створює VPC з ім’ям «main-vpc»;
– google_compute_subnetwork.provedcode-subnet – створює підмережу з ім’ям «provedcode-subnet» в раніше створеному VPC.

Файл «**variables.tf»** визначає наступні змінні для модуля «network»:
– gcp_project – ідентифікатор проєкту в Google Cloud;
– gcp_region – регіон у Google Cloud, де будуть створюватися ресурси.

Файл «**firewalls.tf»** визначає правила брандмауера в Google Cloud:
– google_compute_firewall.frontend_firewall – правило брандмауера, яке дозволяє вхідний трафік на порт 80 для ресурсів з тегом «frontend»;
– google_compute_firewall.backend_firewall – правило брандмауера, яке дозволяє вхідний трафік на порт 8080 для ресурсів з тегом «backend»;
– google_compute_firewall.ssh_firewall – правило брандмауера, яке дозволяє вхідний SSH-трафік на порт 22 для всіх інстансів.

Файл «**outputs.tf»** визначає значення, які Terraform виводить після застосування конфігурації:
– provedcode_subnet – посилання на підмережу, яку використовують екземпляри, значення береться з властивості self_link ресурсу google_compute_subnetwork.provedcode-subnet.

3.3.8 Модуль екземплярів обчислення

Файл **«main.tf»** визначає дві віртуальні машини в Google Cloud:

1. google_compute_instance.backend – створює віртуальну машину з ім'ям «tf-backend», типом «e2-small» і тегом «backend», ця машина має відкритий порт 8080 і 22 через брандмауер:
   - metadata_startup_script – скрипт, який виконується при запуску машини, він об'єднує три скрипти: встановлення Docker, експорт секретних аргументів для запуску образу backend і запуск backend додатка;
   - boot_disk – налаштування завантажувального диска машини, що використовує образ «debian-cloud/debian-11»;
   - network_interface – налаштування мережі машини, що використовує підмережу provedcode_subnet;
   - metadata – налаштування SSH-ключів для машини, тут вказується ім'я користувача і його публічний ключ для входу на сервер;
2. google_compute_instance.frontend – створює віртуальну машину з ім'ям «tf-frontend», типом «e2-small» і тегом «frontend», ця машина має відкритий порт 80 і 22 через брандмауер, залежить від створення машини «backend»:
   - metadata_startup_script – скрипт, який виконується при запуску машини, він об'єднує три скрипти: встановлення Docker, експорт IP-адреси машини backend і запуск frontend додатка;
   - boot_disk – налаштування завантажувального диска машини, що використовує образ «debian-cloud/debian-11»;
   - network_interface – налаштування мережі машини, що використовує підмережу provedcode_subnet;
   - metadata – налаштування SSH-ключів для машини, тут вказується ім'я користувача і його публічний ключ для входу на сервер.

Файл **«variables.tf»** визначає наступні змінні для модуля:

- gcp_project – ідентифікатор проєкту в Google Cloud;
- gcp_zone – зона в Google Cloud, де будуть створюватися ресурси;
- provedcode_subnet – посилання на підмережу, яку використовують екземпляри.

3.3.9 Виконувані скрипти при запуску віртуальних машин

Startup-script – це користувацький скрипт або команда, яку можна автоматично запустити при створенні або перезапуску віртуальної машини в хмарі. Цей скрипт виконує налаштування віртуальної машини та додатків в ній, автоматизуючи такі завдання, як встановлення програмного забезпечення, налаштування середовища та інші операції.

Скрипт **«install_docker.sh»** встановлює Docker на систему на основі Debian. Він використовується як для фронтенду, так і для бекенду.

Скрипт **«env.sh»** експортує змінні середовища, необхідні для роботи бекенду. Цей файл є конфіденційним і не повинен бути опублікований, його потрібно включити в «.gitignore». Використовується тільки на віртуальній машині бекенду.

Скрипт **«run_backend.sh»** запускає Docker контейнер бекенду додатка у такі кроки:

– спочатку завантажує образ mnemov/provedcode-backend з Docker Hub;

– запускає контейнер з цим образом, відкриваючи порт 8080 і встановлюючи змінні середовища, які перезаписують змінні середовища з образу.

Скрипт **«run_frontend.sh»** збирає та запускає Docker контейнер для фронтенду додатка з Dockerfile з клонованого репозиторію.

3.3.10 Запуск інфраструктури

Спочатку треба ініціалізувати робочу директорію Terraform, завантажити провайдера та створити необхідні файли стану. Це виконує команда «terraform init».

Далі потрібно перевірити, які зміни Terraform планує внести до інфраструктури. Це передбачає перегляд запланованих дій, які виконуються командою «terraform plan».

Застосовування змін, необхідних для досягнення бажаного стану інфраструктури, описаного у конфігураційних файлах Terraform виконуються командою «terraform apply».

Команда потребує підтвердження. Після підтвердження будуть створені ресурси і розгорнуті нові інстанси. Контейнер з бекендом має запуститися швидко, але збірка образу та розгортання контейнера з фронтендом може зайняти 5–8 хвилин.

```
ubuntu@craft:~$ cd terraform/infra/
ubuntu@craft:~/terraform/infra$ ll
total 68
drwxrwxr-x 6 ubuntu ubuntu  4096 Apr  7 13:51 ./
drwxrwxr-x 3 ubuntu ubuntu  4096 Apr  4 19:06 ../
drwxr-xr-x 4 ubuntu ubuntu  4096 Mar 24 06:27 .terraform/
-rw-r--r-- 1 ubuntu ubuntu  1155 Mar 24 05:58 .terraform.lock.hcl
drwxrwxr-x 2 ubuntu ubuntu  4096 Apr  4 19:06 keys/
-rw-rw-r-- 1 ubuntu ubuntu   438 Apr  7 11:39 main.tf
drwxrwxr-x 4 ubuntu ubuntu  4096 Mar 24 06:50 modules/
drwxrwxr-x 2 ubuntu ubuntu  4096 Apr  7 11:37 scripts/
-rw-rw-r-- 1 ubuntu ubuntu 22882 Apr  7 11:44 terraform.tfstate
-rw-rw-r-- 1 ubuntu ubuntu   182 Apr  7 11:43 terraform.tfstate.backup
-rw-rw-r-- 1 ubuntu ubuntu   147 Apr  7 11:39 terraform.tfvars
-rw-rw-r-- 1 ubuntu ubuntu   361 Apr  7 11:39 variables.tf
ubuntu@craft:~/terraform/infra$ terraform  init

Initializing the backend...
Initializing modules...

Initializing provider plugins...
- Reusing previous version of hashicorp/google from the dependency lock file
- Using previously-installed hashicorp/google v5.21.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Рисунок 3.21 – Ініціалізація Terraform у каталозі з маніфестом.

```
ubuntu@craft:~/terraform/infra$ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # module.gce.google_compute_instance.backend will be created
  + resource "google_compute_instance" "backend" {
      + can_ip_forward       = false
      + cpu_platform         = (known after apply)
      + current_status       = (known after apply)
      + deletion_protection  = false
      + effective_labels     = (known after apply)
      + guest_accelerator    = (known after apply)
      + id                   = (known after apply)
      + instance_id          = (known after apply)
      + label_fingerprint    = (known after apply)
      + machine_type         = "e2-small"
      + metadata             = {
          + "ssh-keys" = <<-EOT
```

Рисунок 3.22 – Показ того, що буде створено при виконанні поточного Terraform-маніфесту.

```
ubuntu@craft:~/terraform/infra$ terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # module.gce.google_compute_instance.backend will be created
  + resource "google_compute_instance" "backend" {
      + can_ip_forward       = false
      + cpu_platform         = (known after apply)
      + current_status       = (known after apply)
      + deletion_protection  = false
      + effective_labels     = (known after apply)
      + guest_accelerator    = (known after apply)
      + id                   = (known after apply)
      + instance_id          = (known after apply)
      + label_fingerprint    = (known after apply)
      + machine_type         = "e2-small"
      + metadata             = {
          + "ssh-keys" = <<-EOT
```

Рисунок 3.23 – Прийняття Terraform-маніфесту та створення інфраструктури через код.

3.3.11 Перевірка результату роботи застосунку

Після прийняття команди за допомогою Terraform по IP-адресі віртуальної машини фронтенду повинен бути доступний сайт.

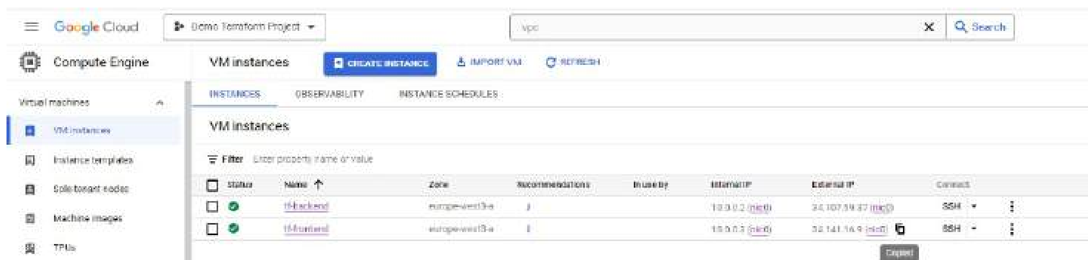

Рисунок 3.24 – Копіювання публічної IP-адреси у інстанса фронтенду, щоб відкрити сайт.

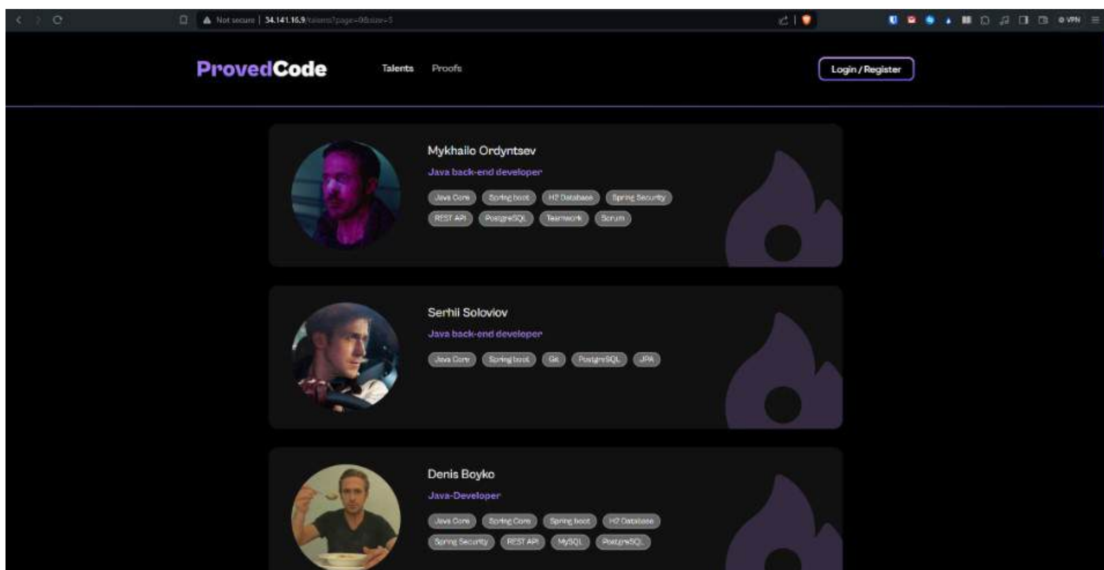

Рисунок 3.25 – Вигляд стартової сторінки Talents застосунку Proved Code, після створеної інфраструктури через Terraform.

3.4 Розгортання контейнеризованнного застосунку за допомогою Kubernetes

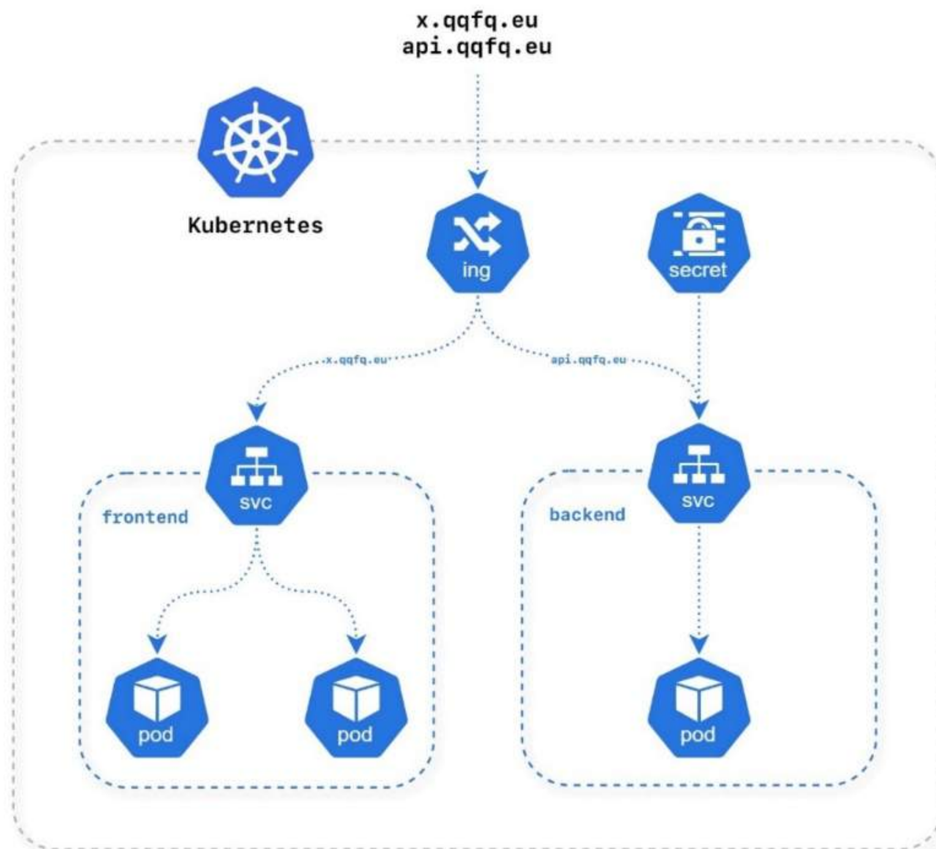

Рисунок 3.26 – Схема інфраструктури роботи застосунку через Kubernetes.

У цій інфраструктурі розгорнуто фронтенд та бекенд веб-застосунку у кластері Kubernetes з використанням Ingress для маршрутизації трафіку. Інфраструктура складається з наступних компонентів:

1. кластер Kubernetes – основна середа, де відбувається оркестрація та управління контейнеризованими додатками;
2. Ingress – компонент Kubernetes, який діє як вхідна точка для трафіку ззовні кластера, він перенаправляє вхідний трафік на відповідні сервіси на основі правил маршрутизації;
3. Secret – сховище конфіденційної інформації, такої як паролі та ключі доступу, необхідної для роботи бекенду;

4. Pod – основні робочі одиниці в Kubernetes, що містять один або кілька контейнерів додатків.
5. Deployment – ресурс Kubernetes, який визначає бажаний стан застосунку, включаючи кількість реплік Pod-ів та стратегію оновлення;
6. Service – абстракції, що представляють логічну групу схожих Pod-ів та забезпечують мережеве з'єднання з ними всередині кластера.

В інфраструктурі є два Deployment: один для фронтенду і один для бекенду. Кожен Deployment містить відповідний набір Pod-ів, об'єднаних у Service. Ingress перенаправляє запити до відповідного Service залежно від доменного імені («x.qqfq.eu» для фронтенду та «api.qqfq.eu» для бекенду). Секрети використовуються для безпечного зберігання конфіденційної інформації бекенду, такої як облікові дані баз даних та ключі доступу. Dockerfile для фронтенду та бекенду будуть зібрані, а їх образи збережені у репозиторій на Docker Hub. Вони мають той самий вміст файлів, що й для інфраструктури Terraform з Docker, але дещо змінені для потреб кластера Kubernetes. Додатково, вони містять виконувані скрипти для збірки. Kubernetes Manifests знаходяться у теці «k8s» і містять файли для Deployment фронтенду та бекенду, а також Ingress та Secret для бекенду.

3.4.1 Створення образів Docker для фронтенду та бекенду додатка для кластеру Kubernetes

Відмінність **«Dockerfile-frontend-k8s»** полягає в тому, що тепер за замовчуванням адресою бекенд-сервера вказано доменне ім'я, яке вирішено використовувати для застосунку. Виконавчий скрипт **«build_frontend.sh»** для збірки фронтенду та збереження до Docker Hub. **«Dockerfile-backend-k8s»** для бекенду тепер може тільки приймати змінні середовища при збірці, оскільки їх не бажано містити у файлі.

Виконавчий скрипт **«build_backend.sh»** для збірки бекенду та збереження до Docker Hub, він потребує перед виконанням скрипту секретів **«secrets.sh»**, що експортує змінні середи.

3.4.2 Створення кластеру Kubernetes

Після створення Docker-образів, потрібно створити Kubernetes-кластер та розгорнути додатки. Кластер складається з вузлів – нод, на яких потім будуть розгортатись Pod-и, всередині яких буде фронтенд та бекенд.

Кластер з трьох нод буде розгорнуто командою: «gcloud container clusters create provedcode --machine-type e2-small --disk-size=10 --num-nodes=3».

Рисунок 3.27 – Статус створення кластеру у терміналі.

Після розгортання нод, В GCE вони будуть відображатися як інстанси віртуальних машин.

Рисунок 3.28 – Оскільки кластер було створено через GCP ноди є інстансами сервісу Compute Cloud.

3.4.3 Підготовка ресурсів для розгортання застосунку в кластері Kubernetes

Спочатку необхідно створити сутність для передачі змінних середовища для роботи бекенду. Для цього можна використовувати ConfigMap, але тільки для тих випадків, коли передаються публічні дані. У випадку з паролями і ключами, які не можна зберігати на GitHub, створюється Secret для зберігання секретної інформації. Файл Secret **«provedcode-backend-secret.yaml»** для Deployment бекенду зберігає приватні змінні. Передавати значення без кодування в base64 не можна, Kubernetes вимагає їх у такому вигляді. Кодування значення до base64 відбувається так:

```
echo -n "your_string" | base64
```

Прийняття Secret має відбуватись пізніше Deployment бекенда, інакше буде помилка під час запуску:

```
kubectl apply -f provedcode-backend-secret.yaml
kubectl get secrets
```

Після того як Secret **«provedcode-backend-secret.yaml»** був застосований, потрібно створити Deployment **«provedcode-backend-deployment.yaml»** для бекенду. Цей Deployment матиме наступні характеристики:

– одна репліка Pod;
– працюватиме через 80 порт;
– перевірка працездатності за запитом до бекенду на цьому ж порту;
– використання змінних середовища з Secret provedcode-backend-secret;
– Pod-и повинні розміщуватись на менш завантажених вузла

Застосування Deployment **«provedcode-backend-deployment.yaml»** до кластера:

```
kubectl apply -f provedcode-backend-deployment.yaml
```

Далі повинен бути запущений **«provedcode-frontend-deployment.yaml»** фронтенду. Тут будуть дві Pod-и з додатком, які працюють через 80 порт.

Застосування Deployment **«provedcode-frontend-deployment.yaml»** до кластера:

```
kubectl apply -f provedcode-frontend-deployment.yaml
```

3.4.4 Створення Ingress в кластері Kubernetes та призначення доменного імені для застосунку

Щоб на певні адреси був перенаправлений трафік, потрібен Ingress та Ingress Controller. Цей контролер поєднує в собі проксі-сервер та балансувальник навантаження, він необхідний, щоб Ingress міг працювати. У цьому випадку буде використовуватися Nginx для контролера:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.10.1/deploy/static/provider/cloud/deploy.yaml
kubectl get services -n ingress-nginx
```

```
namespace/ingress-nginx created
serviceaccount/ingress-nginx created
serviceaccount/ingress-nginx-admission created
role.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
configmap/ingress-nginx-controller created
service/ingress-nginx-controller created
service/ingress-nginx-controller-admission created
deployment.apps/ingress-nginx-controller created
job.batch/ingress-nginx-admission-create created
job.batch/ingress-nginx-admission-patch created
ingressclass.networking.k8s.io/nginx created
validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-nginx-admission created
```

Рисунок 3.29 – Статус творення Ingress у кластері.

Після встановлення Ingress Controller, потрібно зробити Deployment provedcode-frontend-deployment та provedcode-backend-deployment доступними всередині кластера, щоб можна було до них звернутися зсередини мережі Kubernetes у вигляді Service:

```
kubectl expose deployment provedcode-backend-deployment --port=80 --type=ClusterIP
kubectl expose deployment provedcode-frontend-deployment --port=80 --type=ClusterIP

kubectl get services -o wide
```

Щоб трафік з Інтернету міг потрапити до потрібних сервісів всередині кластера, використовуються правила Ingress, що визначені у файлі **«provedcode-ingress-hosts.yaml»**.

Таким чином, всі запити, що будуть надходити до Ingress за адресою «api.qqfq.eu», будуть перенаправлені до Deployment provedcode-backend-deployment. Запити, що будуть надходити за адресою «x.qqfq.eu», будуть перенаправлені до Deployment provedcode-frontend-deployment.

Після того як Ingress Controller готовий, можна застосовувати правила Ingress та переглянути його опис:

```
kubectl apply -f provedcode-ingress-hosts.yaml
kubectl describe ingress provedcode-ingress-hosts
```

```
ubuntu@craft:~/k8s-infra/k8s$ kubectl get pods
NAME                                          READY   STATUS    RESTARTS      AGE
provedcode-backend-deployment-567f669668-2p7h6   1/1     Running   1 (63s ago)   95s
provedcode-frontend-deployment-bdd7d7b9f-p2p85   1/1     Running   0             41m
provedcode-frontend-deployment-bdd7d7b9f-qqgb9   1/1     Running   0             41m
ubuntu@craft:~/k8s-infra/k8s$ kubectl describe ingress
Name:             provedcode-ingress-hosts
Labels:           <none>
Namespace:        default
Address:          34.49.211.99
Ingress Class:    <none>
Default backend:  <default>
Rules:
  Host         Path  Backends
  ----         ----  --------
  api.qqfq.eu
               /     provedcode-backend-deployment:80 (10.120.0.27:80)
  x.qqfq.eu
               /     provedcode-frontend-deployment:80 (10.120.1.14:80,10.120.2.22:80)
Annotations:   ingress.kubernetes.io/backends:
                 {"k8s1-907d2e02-default-provedcode-backend-deployment-80-40692388":"HEALTHY","k8s1-907d2e02-default-provedcode-frontend-deployment-8-cdee7...
               ingress.kubernetes.io/forwarding-rule: k8s2-fr-t6s6ulke-default-provedcode-ingress-hosts-ajad5yvo
               ingress.kubernetes.io/target-proxy: k8s2-tp-t6s6ulke-default-provedcode-ingress-hosts-ajad5yvo
               ingress.kubernetes.io/url-map: k8s2-um-t6s6ulke-default-provedcode-ingress-hosts-ajad5yvo
Events:
  Type    Reason     Age                 From                      Message
  ----    ------     ----                ----                      -------
  Normal  Sync       39m                 loadbalancer-controller   UrlMap "k8s2-um-t6s6ulke-default-provedcode-ingress-hosts-ajad5yvo" created
  Normal  Sync       39m                 loadbalancer-controller   TargetProxy "k8s2-tp-t6s6ulke-default-provedcode-ingress-hosts-ajad5yvo" created
  Normal  Sync       39m                 loadbalancer-controller   ForwardingRule "k8s2-fr-t6s6ulke-default-provedcode-ingress-hosts-ajad5yvo" created
  Normal  IPChanged  39m                 loadbalancer-controller   IP is now 34.49.211.99
  Normal  Sync       8m49s (x10 over 41m)  loadbalancer-controller   Scheduled for sync
```

Рисунок 3.30 – Вивід детального описання Ingress у кластері. Виділено публічну адресу Ingress, на яку мають йти запити до фронтенду та бекенду додатка.

У GCP має з'явитися ще один балансувальник навантаження, і він буде типу Application.

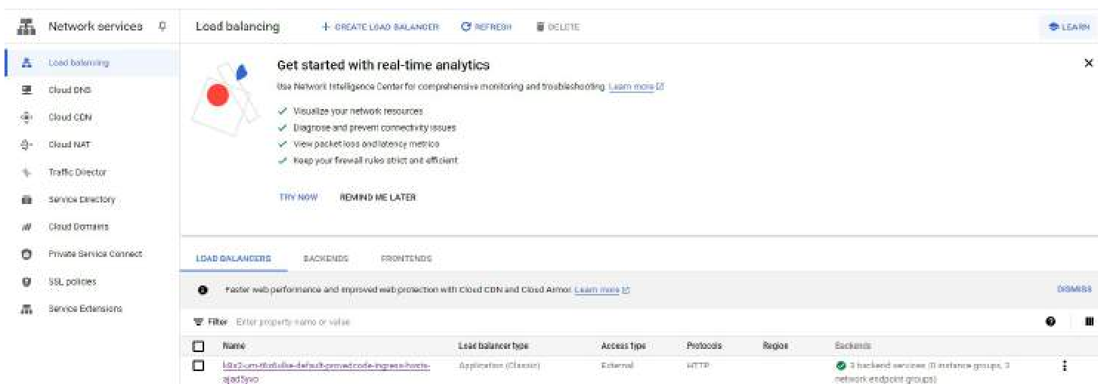

Рисунок 3.31 – Створений балансувальник навантаження, що є частиною Ingress.

У балансувальнику навантаження прописані правила маршрутизації Ingress за доменними іменами та публічна IP-адреса, що була зазначена вище. Її потрібно зберегти.

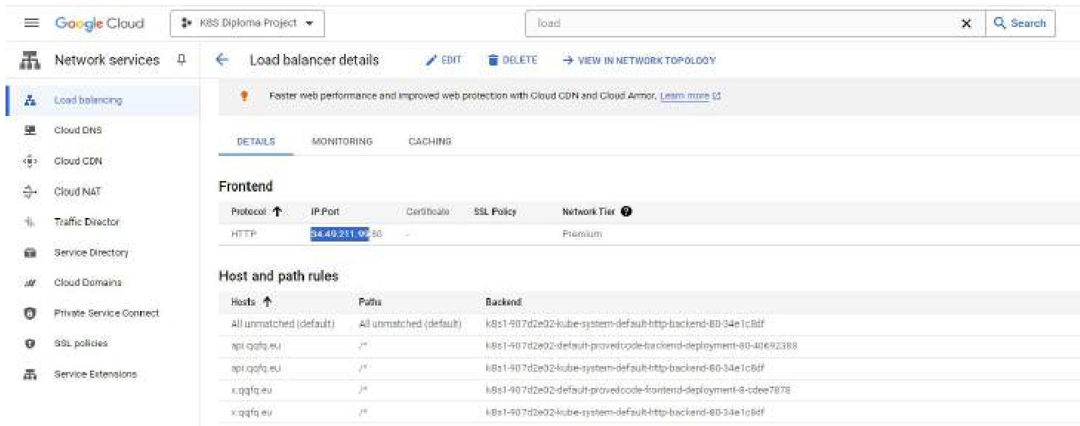

Рисунок 3.32 – У балансувальнику навантаження вказано шляхи до сервісів залежно від доменного імені, з якого надійшов запит.

Нижче в описі балансувальника навантаження вказано, скільки Pod-ів у робочому стані і які до них застосовуються перевірки працездатності (healthcheck).

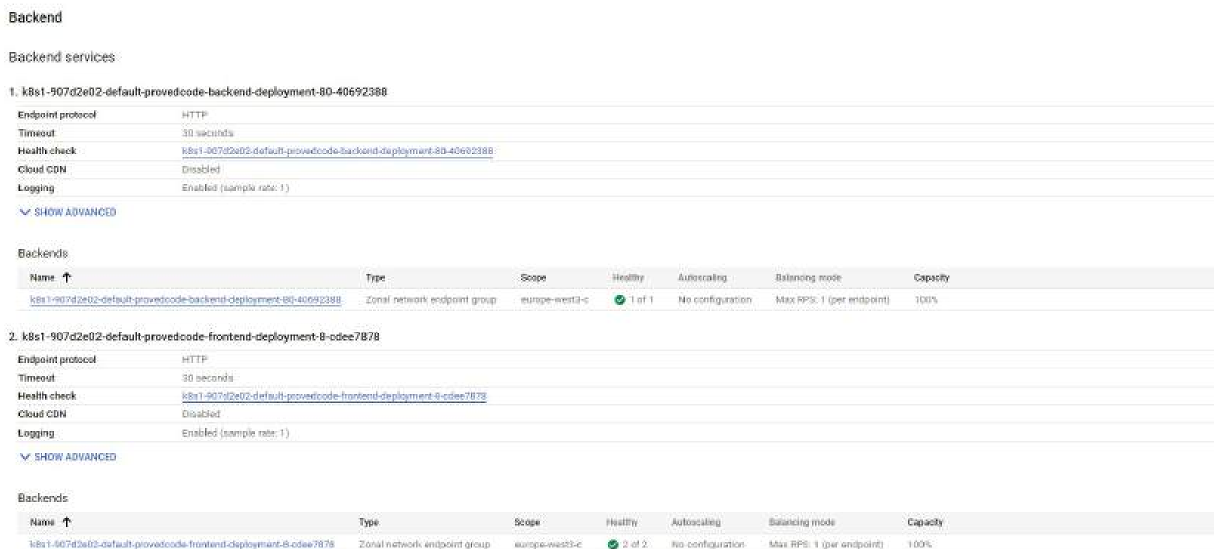

Рисунок 3.33 – Перед віправленням запиту до Pod-и з Ingress відбувається перевірка працездатності фронтенду та бекенду.

При переході за посиланням на одну з перевірок можна дізнатися, як саме вона працює. Зокрема для бекенду використовується HTTP GET-запит /api/v2/talents?page=0&size=5 до сервера у Pod-і через 80 порт.

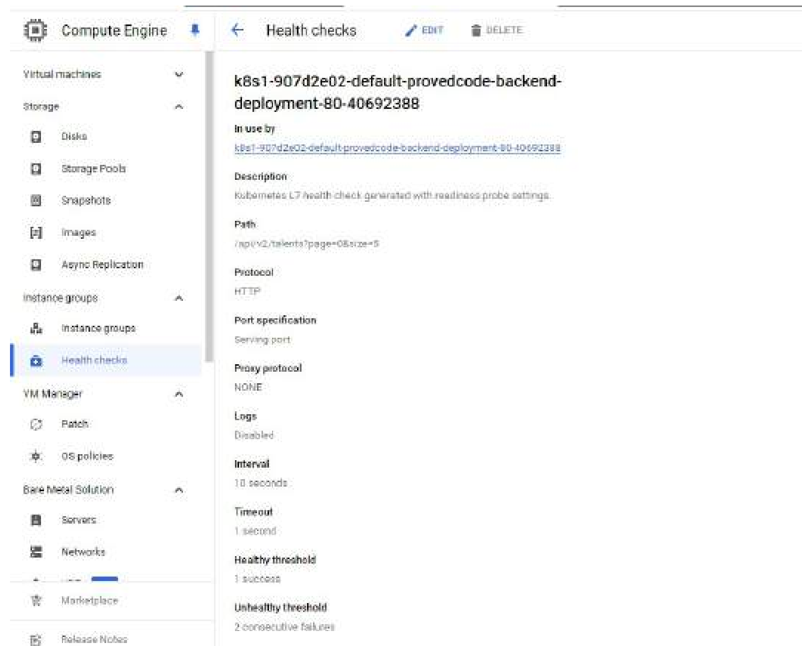

Рисунок 3.89 – Інформація про перевірку працездатності для Pod-ів з бекендом.

Наступним кроком потрібно прив’язати IP-адресу Ingress до доменних імен, створених для фронтенду і бекенду, у наявного доменного реєстратора, такого як CloudFlare.

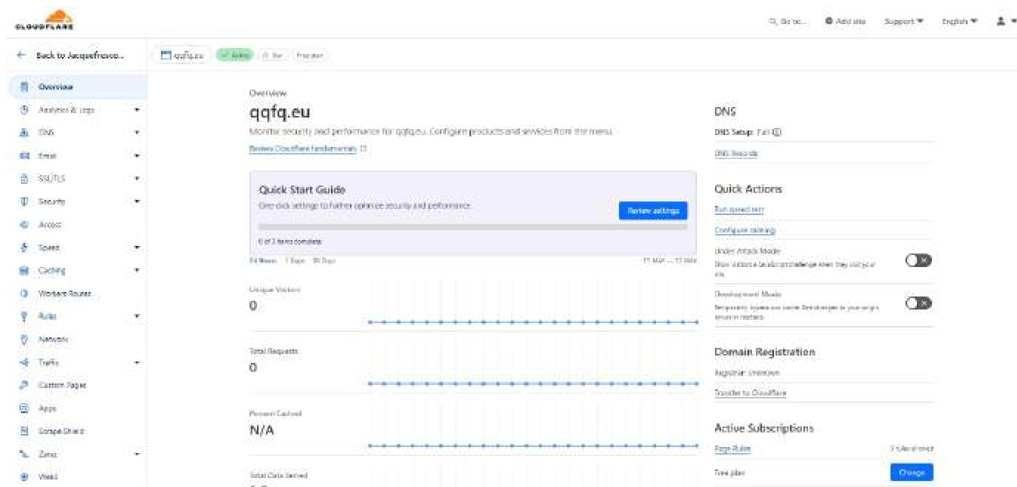

Рисунок 3.34 – Доступне доменне ім’я на CloudFlare для прив’язання до застосунку.

Рисунок 3.35 – Прив’язка IP-адреси до імен піддоменів «api.qqfq.eu» та «x.qqfq.eu».

3.4.5 Перевірка працездатності застосунку у кластері Kubernetes

Залежно від доменного реєстратора час оновлення DNS-імені в Інтернеті може варіюватися. Щоб переконатися, що воно оновилося для пристрою, з якого буде здійснюватися перевірка, достатньо виконати команду ping в терміналі. Якщо DNS-служба оновилася, буде показано IP-адресу, до якої було зроблено прив’язку.

Рисунок 3.36 – Перевірка оновлення IP-адреси Ingress до доменного імені через утиліту ping на пристрої де буде відбуватись перевірка.

Наступним кроком потрібно перевірити роботу сайту в приватній вкладці браузера. Це слід робити в такому режимі, оскільки він очищає кеш для кожної сесії, що дозволяє завантажувати сторінку заново щоразу.

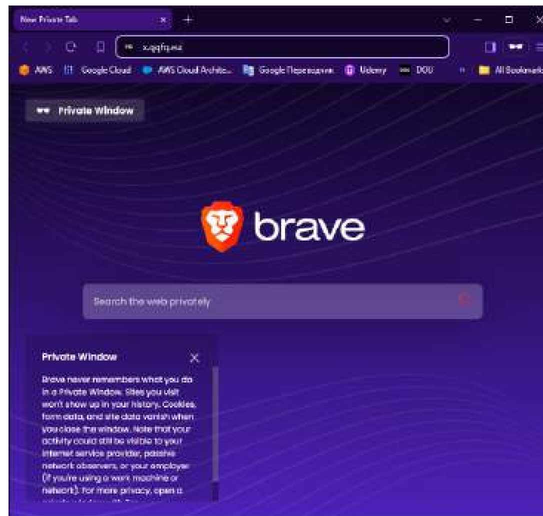

Рисунок 3.37 – Вхід до сайту має відбуватись через приватний режим, для очищення кешу сторінки, яка може працювати некоректно.

Якщо сторінка буде заповнена списком, це означає, що бекенд працює правильно.

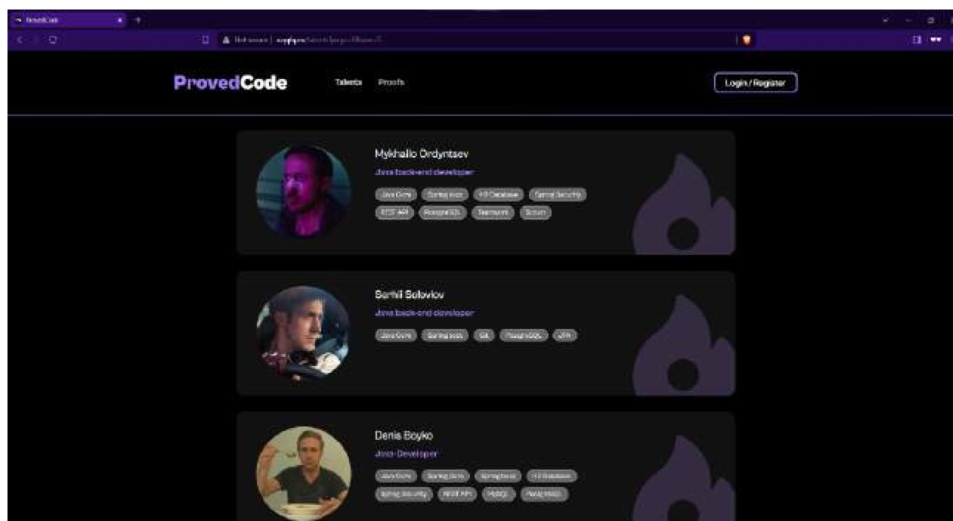

Рисунок 3.38 – Вигляд стартової сторінки Talents застосунку Proved Code, за доменною адресою для фронтенду, після створеної інфраструктури через Kubernetes.

У меню розробника можна побачити, що запит до API бекенду йде на доменне ім’я «api.qqfq.eu».

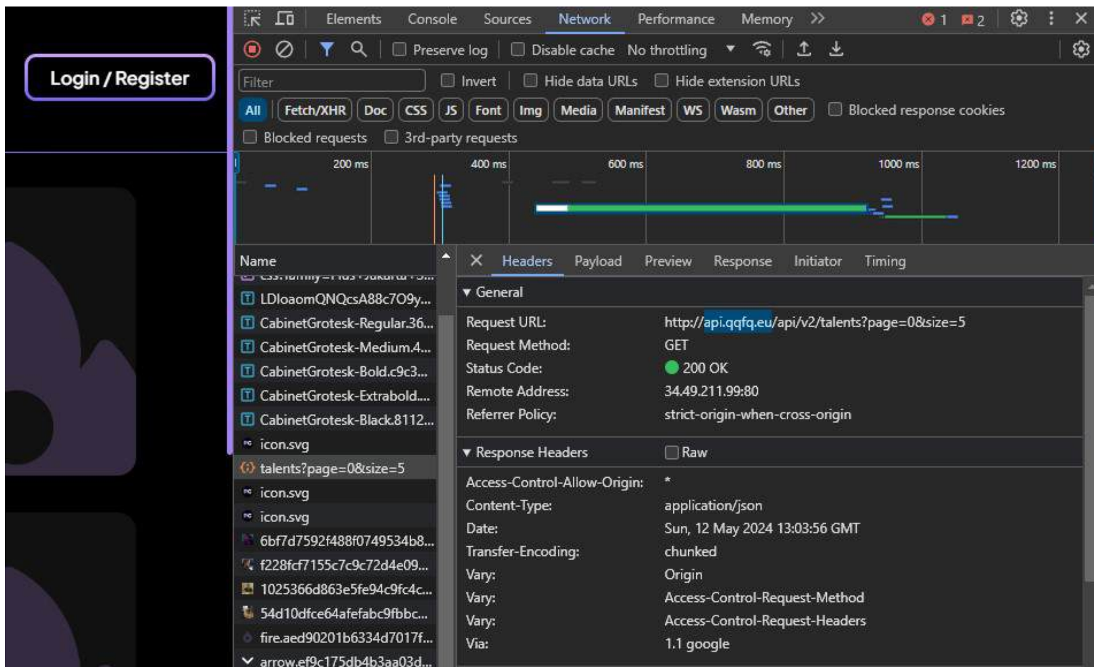

Рисунок 3.39 – Запит до бекенду, через налаштоване доменне ім’я.

3.4.6 Розгортання застосунку через Helm у кластері Kubernetes

Helm – це менеджер пакетів для Kubernetes, який спрощує визначення, встановлення та оновлення складних додатків у кластері Kubernetes. За допомогою Helm можна розгорнути застосунок Proved Code у середовищах розробки dev та продакшн prod, використовуючи один і той же шаблон, але з різними налаштуваннями для кожного середовища. Встановлення Helm на сервер виконується шляхом завантаження бінарного файлу з офіційного репозиторію на GitHub [19] та його правильної конфігурації.

Код встановлення Helm:

```
curl https://get.helm.sh/helm-v3.15.0-rc.2-linux-arm.tar.gz
tar -xf helm-v3.15.0-rc.2-linux-arm.tar.gz
sudo mv linux-arm/helm /usr/bin
rm -rf linux-arm/ helm-v3.15.0-rc.2-linux-arm.tar.gz
helm
```

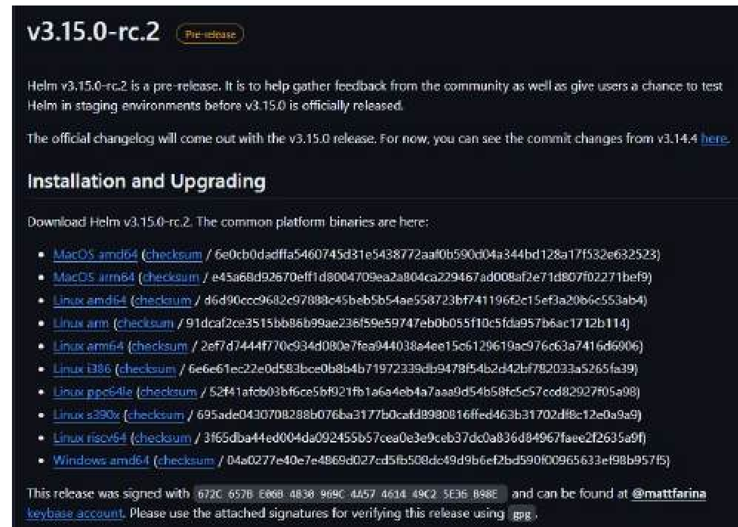

Рисунок 3.40 – Офіційний репозиторій Helm у GitHub, посилання на завантаження утиліти.

При створенні Helm-чарту для застосунку використовуються два ключових файли. **«chart.yaml»** містить метадані чарту, такі як назва, версія та опис застосунку. Файл **«values.yaml»** відіграє важливу роль, дозволяючи визначити змінні, які можна легко змінювати при розгортанні в різних середовищах, без необхідності модифікувати самі шаблони.

У файлах застосунку до назв ресурсів було додано суфікс «-{{ .Values.environment }}», для додання назви «-prod», або «-dev». Це рішення, яке допомагає уникнути конфліктів при створенні копій ресурсів у різних середовищах. Окрім того, до змінних у **«values.yaml»** були додані важливі параметри: кількість реплік для Deployment фронтенду та бекенду, що дозволяє масштабувати застосунок відповідно до навантаження; назви Docker-образів з DockerHub для кожної частини застосунку; параметри healthcheck для перевірки працездатності бекенду; назви доменних імен для Ingress, що направляють трафік до відповідних сервісів.

За замовчуванням у файлі values.yaml встановлені значення для продакшн-середовища. Для розгортання в prod не потрібно вказувати додаткові параметри – достатньо виконати команду «helm install provedcode-production ProvedCode-Deploy/».

```
ubuntu@craft:~/helm-infra$ helm install provedcode-production ProvedCode-Deploy/
NAME: provedcode-production
LAST DEPLOYED: Sun May 26 13:11:24 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Рисунок 3.41 – Повідомлення про розгортання застосунку «Proved Code» у середовищі «production».

Команда для розгортання у середовищі dev потребує завдання специфічних змінних, як environment=dev для ідентифікації ресурсів, ingress.frontendUrl=dev.x.qqfq.eu та ingress.backendUrl=dev.api.qqfq.eu для коректної маршрутизації трафіку:

```
helm install --set
environment=dev,ingress.frontendUrl=dev.x.qqfq.eu,ingress.backendUrl=dev.api.qqfq.eu
provedcode-development ProvedCode-Deploy/
```

```
ubuntu@craft:~/helm-infra$ helm install --set environment=dev,ingress.frontendUrl=dev.x.qqfq.eu,ingress.
NAME: provedcode-development
LAST DEPLOYED: Sun May 26 13:20:06 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Рисунок 3.42 – Повідомлення про розгортання застосунку «Proved Code» у середовищі «development».

Для кожного застосунку dev і prod створюється окремий Ingress Controller, який отримує унікальну публічну IP-адресу:

```
kubectl get pods
kubectl describe ingress
```

```
ubuntu@craft:~$ kubectl describe ingress
Name:             provedcode-ingress-hosts-prod
Labels:           app.kubernetes.io/managed-by=Helm
Namespace:        default
Address:          34.160.48.188
Ingress Class:    <none>
Default backend:  <default>
Rules:
  Host        Path  Backends
  ----        ----  --------
  api.qqfq.eu
              /   provedcode-backend-service-prod:80 (10.20.1.4:80,10.20.2.4:80)
  x.qqfq.eu
              /   provedcode-frontend-service-prod:80 (10.20.1.3:80,10.20.2.5:80)
Annotations:  ingress.kubernetes.io/backends:
                {"k8s1-42b6fe4c-default-provedcode-backend-service-pro-8-4f16e472":"HEALTHY","k8
s1-42b6fe4c-default-provedcode-frontend-service-pr-8-280a5...
              ingress.kubernetes.io/forwarding-rule: k8s2-fr-g8a9bzxt-default-provedcode-ingress
-hosts-prod-dorz82by
              ingress.kubernetes.io/target-proxy: k8s2-tp-g8a9bzxt-default-provedcode-ingress-ho
sts-prod-dorz82by
              ingress.kubernetes.io/url-map: k8s2-um-g8a9bzxt-default-provedcode-ingress-hosts-p
rod-dorz82by
              meta.helm.sh/release-name: provedcode-production
              meta.helm.sh/release-namespace: default
Events:
  Type    Reason  Age                    From                     Message
  ----    ------  ----                   ----                     -------
  Normal  Sync    6m44s (x156 over 25h)  loadbalancer-controller  Scheduled for sync
```

Рисунок 3.43 – Виділення публічної адреси до Ingress для застосунку у середовищі «production».

```
ubuntu@craft:~/helm-infra$ kubectl describe ingress
Name:             provedcode-ingress-hosts-dev
Labels:           app.kubernetes.io/managed-by=Helm
Namespace:        default
Address:          34.111.106.189
Ingress Class:    <none>
Default backend:  <default>
Rules:
  Host            Path  Backends
  ----            ----  --------
  dev.api.qqfq.eu
                  /   provedcode-backend-service-dev:80 (10.20.1.14:80,10.20.2.13:80)
  dev.x.qqfq.eu
                  /   provedcode-frontend-service-dev:80 (10.20.2.14:80,10.20.3.25:80)
Annotations:      ingress.kubernetes.io/backends:
                    {"k8s1-42b6fe4c-default-provedcode-backend-service-dev-8-e35698cb":"HEALTHY"
,"k8s1-42b6fe4c-default-provedcode-frontend-service-de-8-c85ea...
                  ingress.kubernetes.io/forwarding-rule: k8s2-fr-g8a9bzxt-default-provedcode-ing
ress-hosts-dev-fd0yecyg
                  ingress.kubernetes.io/target-proxy: k8s2-tp-g8a9bzxt-default-provedcode-ingres
s-hosts-dev-fd0yecyg
                  ingress.kubernetes.io/url-map: k8s2-um-g8a9bzxt-default-provedcode-ingress-hos
ts-dev-fd0yecyg
                  meta.helm.sh/release-name: provedcode-development
                  meta.helm.sh/release-namespace: default
Events:
  Type    Reason  Age     From                     Message
  ----    ------  ----    ----                     -------
  Normal  Sync    9m24s   loadbalancer-controller  UrlMap "k8s2-um-g8a9bzxt-de
fault-provedcode-ingress-hosts-dev-fd0yecyg" created
  Normal  Sync    9m15s   loadbalancer-controller  TargetProxy "k8s2-tp-g8a9bz
xt-default-provedcode-ingress-hosts-dev-fd0yecyg" created
```

Рисунок 3.44 – Виділення публічної адреси до Ingress для застосунку у середовищі «development».

Для кожного середовища необхідно налаштувати відповідні піддомени: dev.x.qqfq.eu і dev.api.qqfq.eu для середовища розробки, x.qqfq.eu і api.qqfq.eu для продакшну.

Type ▲	Name	Content	Proxy status	TTL	Actions
A	api	34.160.48.188	DNS only	Auto	Edit ▸

A	x	34.160.48.188	DNS only	Auto	Edit
A	dev.api	34.111.106.189	DNS only	Auto	Edit
A	dev.x	34.111.106.189	DNS only	Auto	Edit

Рисунок 3.45 – Призначенням публічним IP-адресам доменних імен.

Після налаштування доменних імен та Ingress, можна перевірити роботу застосунку, відвідавши сторінки фронтенду в браузері.

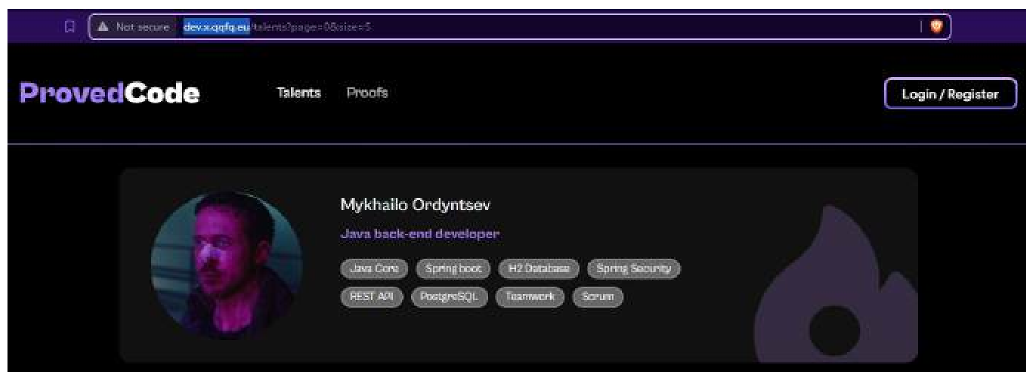

Рисунок 3.46 – Працюючий застосунок з середовища «development» за посиланням «dev.x.qqfq.eu».

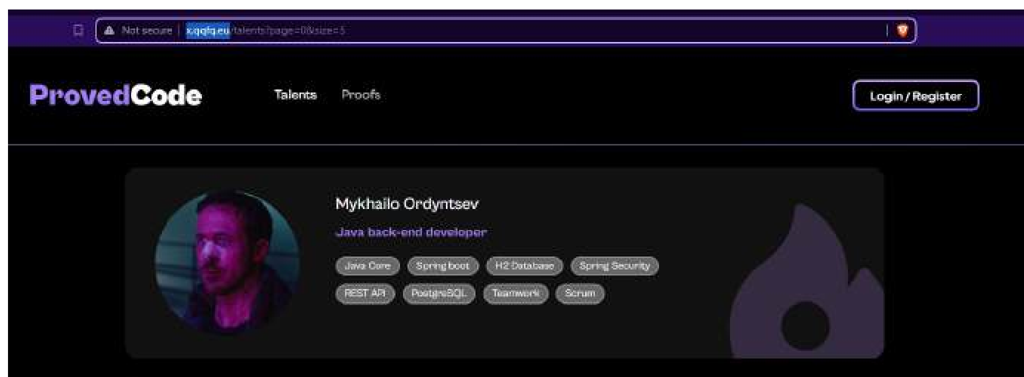

Рисунок 3.47 – Працюючий застосунок з середовища «production» за посиланням «x.qqfq.eu».

Отже, через Helm можна швидко розгорнути копії створеного застосунку для різних середовищ. Після того, як були створені різні оточення для застосунку, дня їх необхідно налаштувати CI/CD.

3.5 Налаштування системи безперервної інтеграції

Далі необхідно розглянути процес налаштування системи безперервної інтеграції та доставки CI/CD для автоматичного збирання, тестування та розгортання оновлень застосунку. За допомогою CI/CD автоматизується шлях програмного коду від розробки до виробничого середовища, що значно прискорює цикл розробки та зменшує ризик помилок при розгортанні.

Інструментом для організації CI/CD обрано GitLab CI, популярну та потужну платформу, що дозволяє повністю автоматизувати процеси збирання, тестування та розгортання додатків.

У персональному записі GitLab створено групу provedcode для централізованого управління кодовою базою. У цій групі засновано два проєкти: Backend та Frontend, які виступають сховищами коду для серверної та клієнтської частин застосунку відповідно. У кожному з цих репозиторіїв є окремі гілки prod та dev.

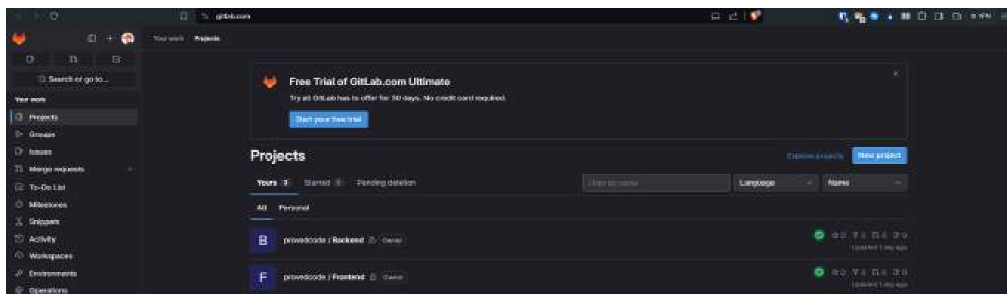

Рисунок 3.48 – Проєкти з репозиторіями вихідного коду фронтенду та бекенду додатка.

Ключовим елементом CI/CD є файл .gitlab-ci.yml, розміщений у кореневому каталозі кожного репозиторію. Цей файл визначає конфігурацію процесу CI/CD і автоматично запускає pipeline з набором дій щоразу, коли відбувається новий пуш у репозиторій за вказаною гілкою.

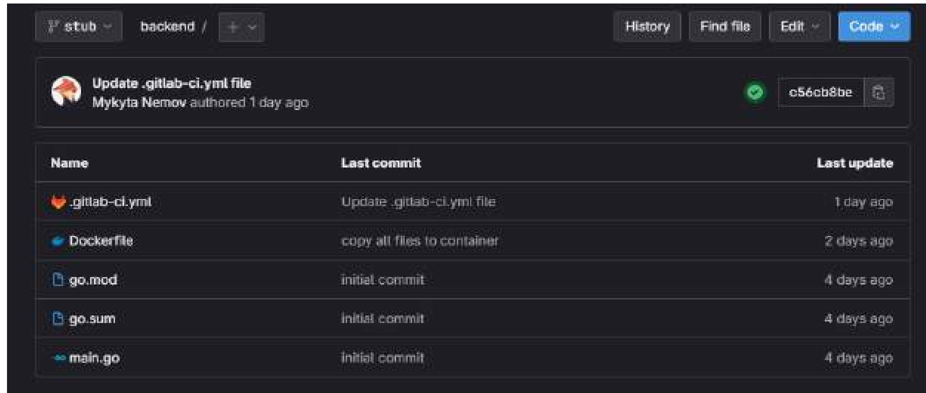

Рисунок 3.49 – Репозиторій вихідного коду бекенду на гілці «dev».

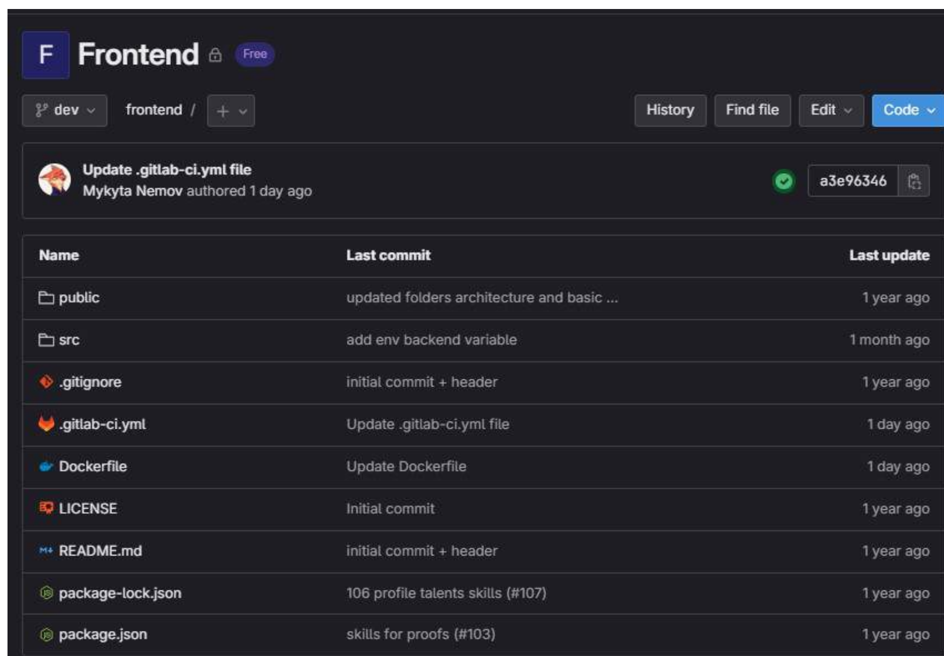

Рисунок 3.50 – Репозиторій вихідного коду фронтенду на гілці dev.

Для кожного застосунку налаштовується pipeline, що складається з трьох основних етапів: збирання, тестування та розгортання.

На етапі збірки використовується Dockerfile, який визначає інструкції для побудови Docker-образу застосунку.

Далі відбувається тестування - запити до розгорнутого контейнеру з новозібраним образом, що перевіряє його працездатність. Після успішного проходження тестів, образ завантажується до Docker Hub - централізованого сховища контейнерів.

Фінальним етапом pipeline є розгортання. За допомогою SSH-з'єднання GitLab зв'язується з сервером, що керує Kubernetes кластером, та виконує команду rollout для Deployment. Ця команда оновлює вказаний Deployment до останньої версії застосунку, завантаженої в Docker Hub.

dev backend / .gitlab-ci.yml Find file Blame History Permalink

Update .gitlab-ci.yml file
Mykyta Nemov authored 4 days ago c56cb8be

✓ This GitLab CI configuration is valid. Learn more

.gitlab-ci.yml 1.06 KiB Blame Edit Replace Delete

```yaml
image: docker:latest

services:
  - docker:dind

stages:
  - build
  - deploy

build-image:
  stage: build

  before_script:
    - echo "$DOCKER_REGISTRY_PASS" | docker login --username $DOCKER_REGISTRY_USER --password-stdin

  script:
    - echo "Building the Docker image..."
    - docker build -t  provedcode-backend . --platform linux/x86_64
    - docker tag provedcode-backend mnemov/provedcode-backend:go
    - echo "Done."

    - echo "Pushing the image to the registry..."
    - docker push mnemov/provedcode-backend:go
    - echo "Done."

deploy-job:
  stage: deploy
  before_script:
    - 'which ssh-agent || ( apt-get update -y && apt-get install openssh-client -y )'
    - eval $(ssh-agent -s)
    - echo "$K8S_CLUSTER_SSH_KEY" | base64 -d > /tmp/ssh_private_key
    - chmod 600 /tmp/ssh_private_key
    - ssh-add /tmp/ssh_private_key
    - mkdir -p ~/.ssh
    - ssh-keyscan h.qqfq.eu >> ~/.ssh/known_hosts

  script:
    - echo "Rolling out..."
    - ssh ubuntu@h.qqfq.eu 'kubectl rollout restart deployment/provedcode-backend-deployment-prod'
    - echo "Rolled out."
```

Рисунок 3.51 – Pipeline для виконання збірки та деплою бекенду на гілці «dev».

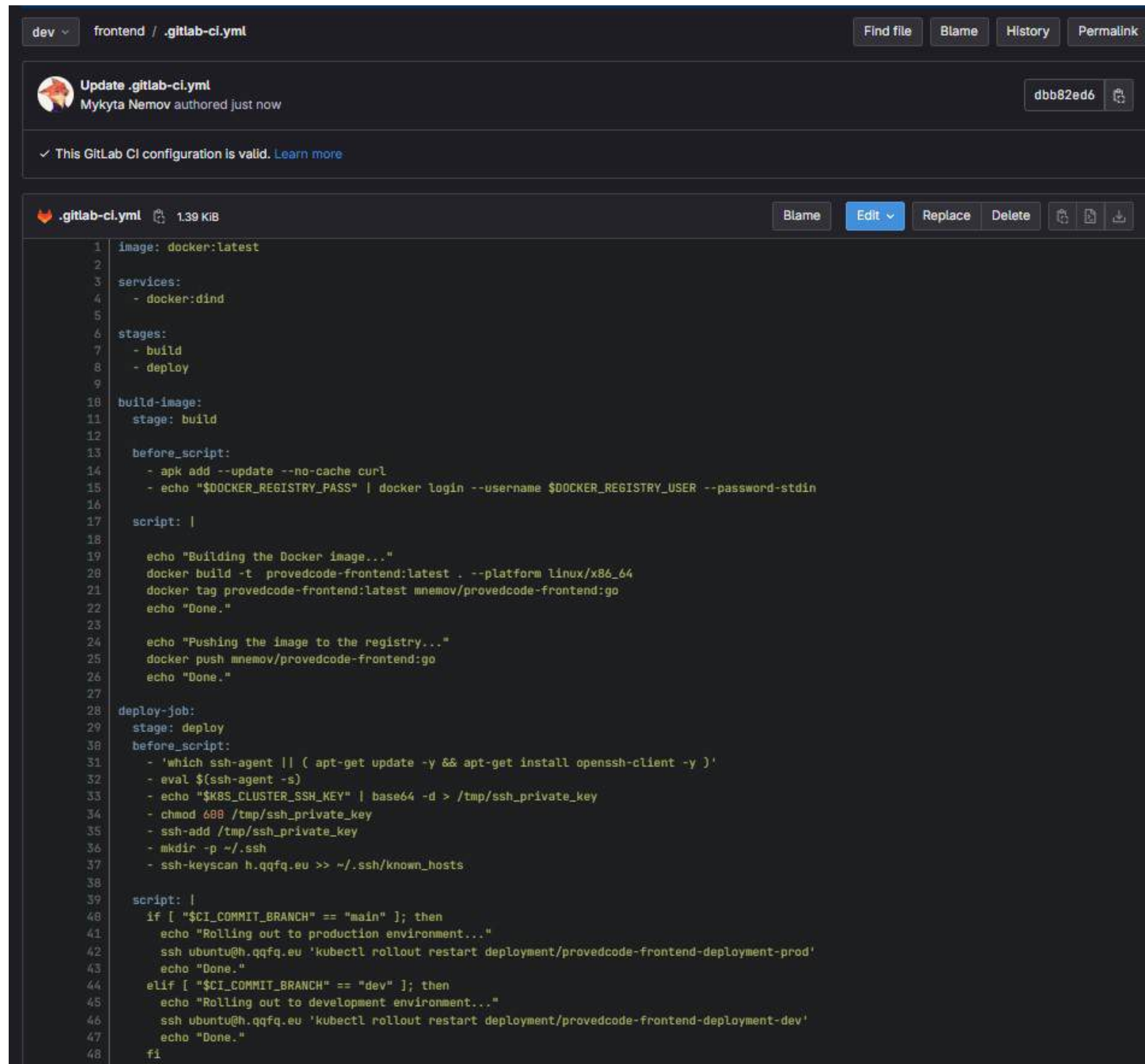

Рисунок 3.52 – Pipeline для виконання збірки та деплою фронтенду на гілці «dev».

Після кожного пушу змін у репозиторій виконується pipeline, який візуально показує статус кожного етапу: його успішне завершення чи можливі помилки. Це надає розробникам миттєвий зворотній зв’язок про їхні зміни.

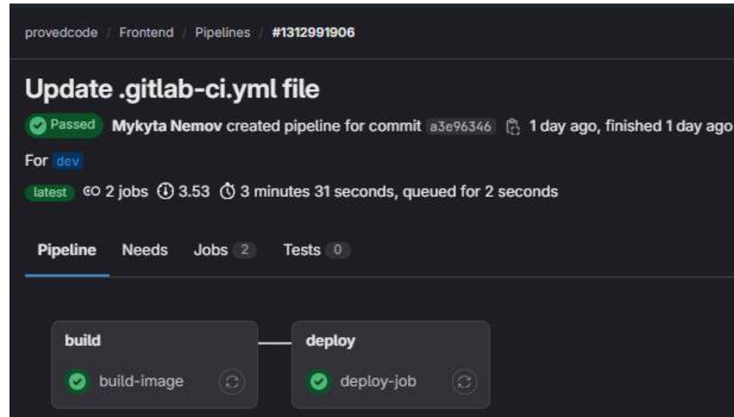

Рисунок 3.53 – Статус виконаного етапу збірки та деплою для pipeline фронтенду.

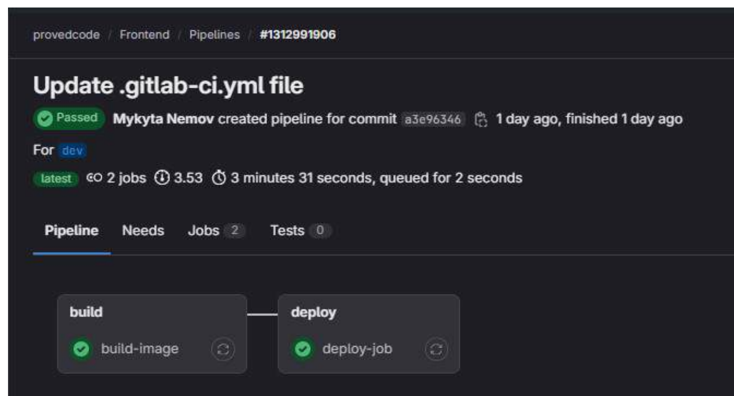

Рисунок 3.54 – Статус виконаного етапу збірки та деплою для pipeline бекенду.

Використання GitLab CI/CD у проєкті дозволяє значно прискорити цикл розробки, підвищити якість коду через автоматичне тестування та зменшити ризик помилок при розгортанні завдяки автоматизації процесів. Розробники можуть зосередитися на створенні функціоналу, залишивши рутинні операції збирання та розгортання системі CI/CD. Тепер необхідно відстежувати стан інфраструктури через моніторинг системи.

3.6 Налаштування системи моніторингу

Із зростанням масштабу та складності розгорнутих систем, стає важливим не лише запускати сервіси, але й стежити за їхньою роботою. Саме тому налаштування моніторингу в кластері Kubernetes є необхідністю.

Першочерговим завданням є моніторинг стану самого кластера. Це включає відстеження стану нод та використання ресурсів. Такий моніторинг дозволяє вчасно виявляти вузькі місця, прогнозувати потреби в ресурсах інфраструктури.

Окрім моніторингу кластеру, необхідно також звернути увагу на моніторинг фронтенду. Саме через фронтенд користувачі взаємодіють із сервісом, і будь-які проблеми на цьому рівні безпосередньо впливають на сприйняття якості сервісу. Моніторинг фронтенду в Kubernetes є важливим і включає аналіз поведінки веб-серверів, зокрема Nginx. Це означає відстеження типів запитів, швидкості відповідей, помилок та навантаження. Такий підхід дозволяє зрозуміти, як користувачі використовують систему, де виникають затримки чи збої, і оперативно реагувати на ці проблеми.

3.6.1 Моніторинг стану кластеру Kubernetes

Для ефективного управління кластером Kubernetes необхідно постійно слідкувати за його станом. Моніторинг дозволяє отримувати актуальну інформацію про використання ресурсів: пам’яті, процесорів, місця на жорстких дисках. Ці дані критично важливі для оптимізації роботи кластеру та запобігання можливим проблемам. Для цього використовується стек «kube-prometheus-stack», який включає попередньо налаштовані інструменти Grafana та Prometheus, що значно спрощує процес налаштування моніторингу.

Щоб Prometheus функціонував в середовищі Kubernetes, необхідно ретельно налаштувати змінні під час його розгортання з файлу **«prometheus-values.yaml»**.

Після створення файлу файлу зі змінними, треба його використовувати для розгортання стеку kube-prometheus-stack:

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
helm install prometheus prometheus-community/kube-prometheus-stack -f prometheus-values.yaml
```

```
prometheus-grafana                        ClusterIP   10.7.231.202   <none>   80/TCP             110s
prometheus-kube-prometheus-alertmanager   ClusterIP   10.7.232.97    <none>   9093/TCP,8080/TCP  110s
prometheus-kube-prometheus-operator       ClusterIP   10.7.234.34    <none>   443/TCP            110s
prometheus-kube-prometheus-prometheus     ClusterIP   10.7.234.239   <none>   9090/TCP,8080/TCP  110s
prometheus-kube-state-metrics             ClusterIP   10.7.234.221   <none>   8080/TCP           110s
prometheus-operated                       ClusterIP   None           <none>   9090/TCP           97s
prometheus-prometheus-node-exporter       ClusterIP   10.7.236.113   <none>   9100/TCP           110s
```

Рисунок 3.55 – Сервіси, що були створені через helm-chart «kube-prometheus-stack».

Доступ до веб-інтерфейсу Grafana можна отримати, відкривши відповідний сервіс у кластері Kubernetes. Окрім того, можна зробити сервіс Prometheus публічно доступним.

```
kubectl expose service prometheus-kube-prometheus-prometheus --type=LoadBalancer --port=80 --target-port=9090 --name=prometheus-server-ext

kubectl expose service prometheus-grafana --type=LoadBalancer --target-port=3000 --name=grafana-ext
```

A	grafana	35.234.73.170	DNS only	Auto	Edit
A	prometheus	35.234.88.26	DNS only	Auto	Edit

Рисунок 3.56 – Прив’язка публічних IP-адрес відкритих сервісів Grafana та Prometheus.

Після того, як сервіси Grafana та Prometheus отримали зовнішні IP-адреси, їм можна призначити доменні імена. Це підвищує зручність доступу до цих інструментів.

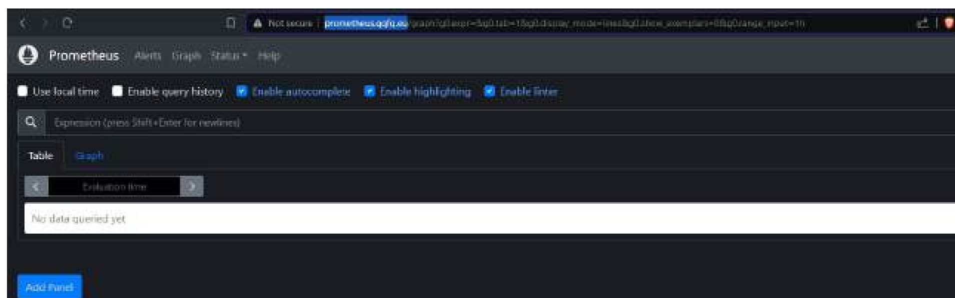

Рисунок 3.57 – Вигляд сторінки Prometheus за посиланням «prometheus.qqfq.eu».

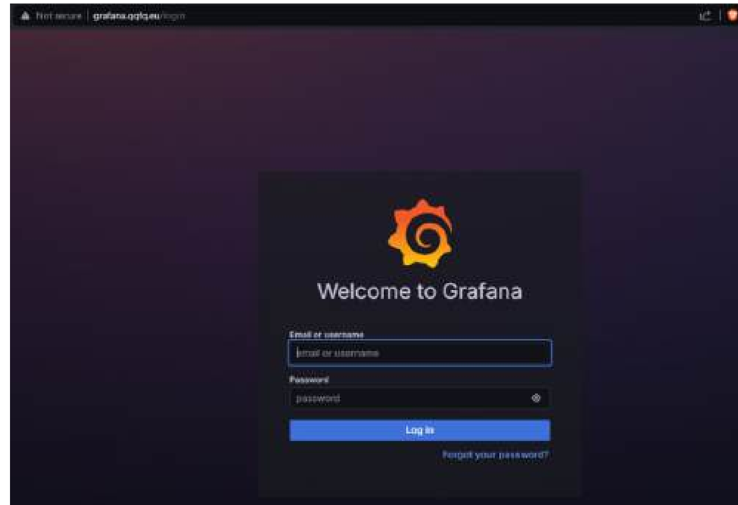

Рисунок 3.58 – Вигляд сторінки логіну Grafana за посиланням «grafana.qqfq.eu».

При першому вході до Grafana, після її встановлення, за замовчуванням використовується логін «admin». Однак, пароль не є стандартним і його потрібно отримати, виконавши спеціальну команду в кластері Kubernetes:

```
kubectl get secret --namespace default prometheus-grafana -o jsonpath="{.data.admin-password}" | base64 --decode ; echo
```

Grafana пропонує широкий набір готових панелей моніторингу (Dashboards). Кожна з них надає детальну інформацію про різні аспекти кластеру: використання ресурсів, стан мережі, здоров’я подів та нод. Ці панелі допомагають швидко виявляти вузькі місця, аномалії у споживанні ресурсів та потенційні проблеми з продуктивністю, що дозволяє вчасно реагувати на будь-які відхилення від норми.

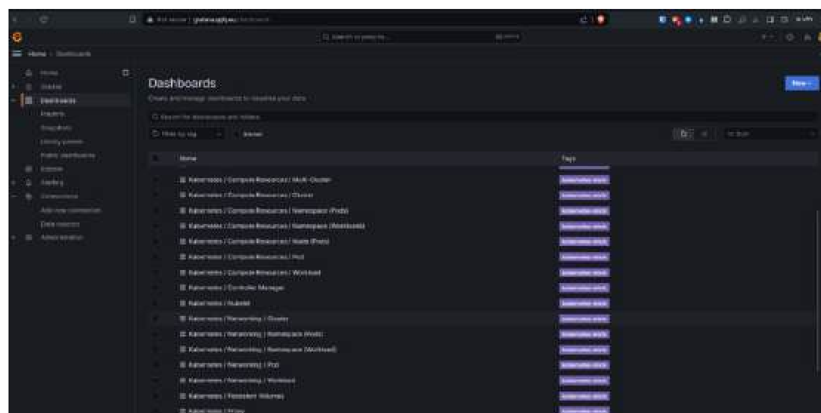

Рисунок 3.59 – Список Dashboards з моніторингу ресурсів кластеру.

Рисунок 3.60 – Дошка моніторингу обчислювальних ресурсів усього кластеру.

Рисунок 3.61 – Дошка моніторингу обчислювальних ресурсів для нод.

3.6.2 Моніторинг стану фронтенду у кластері Kubernetes

Для повноцінного моніторингу необхідно налаштувати збір метрик з фронтенду. Nginx у Pod-і може надавати статистику підключень через спеціальний запит «/nginx_status» та файл access.log. Ці джерела дають інформацію про виконані запити, включаючи типи відповідей, загальну кількість запитів, розподіл на читання і запис, та затримку Nginx. Така

інформація візуалізується у вигляді графіків, що надає тестувальникам глибше розуміння стану фронтенду.
Для збору метрик використовується Telegraf – агент для збору логів та запитів з контейнера Nginx. Telegraf перетворює ці дані на метрики і відправляє їх до Prometheus. Потім Grafana отримує ці метрики від Prometheus і візуалізує їх.

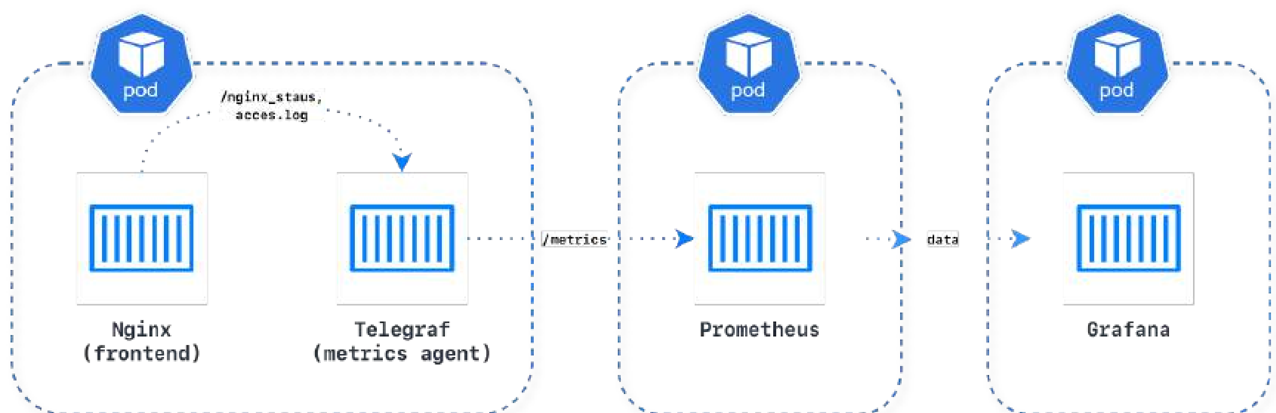

Рисунок 3.62 – Схема отримання метрик веб-серверу Nginx до Grafana.

Щоб Nginx повертав статистику по підключеннях, в його конфігурації необхідно дозволити відправку цієї інформації через запит «/nginx_status». Цю конфігурацію можна передати в контейнер з Nginx у Deployment фронтенду за допомогою configmap у **«nginx-config-map.yaml»**.

Налаштування включало конфігурацію Nginx для відправки статистики, додавання контейнера Telegraf у один под з Nginx, створення сервісу для збору метрик з Telegraf та налаштування Prometheus для відстеження цього сервісу через ServiceMonitor.

Для візуалізації метрик, що збираються через Telegraf, на офіційному сайті Grafana була знайдена спеціальна Dashboard. Її імпорт відбувається шляхом копіювання номера Dashboard [20] та вибору Prometheus як джерела даних. Цей метод дозволяє швидко отримати готове рішення для відображення метрик.

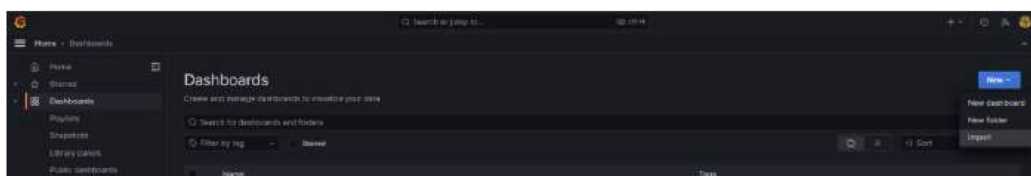

Рисунок 3.63 – На сторінці «Dashboards» обирається імпорт дошки.

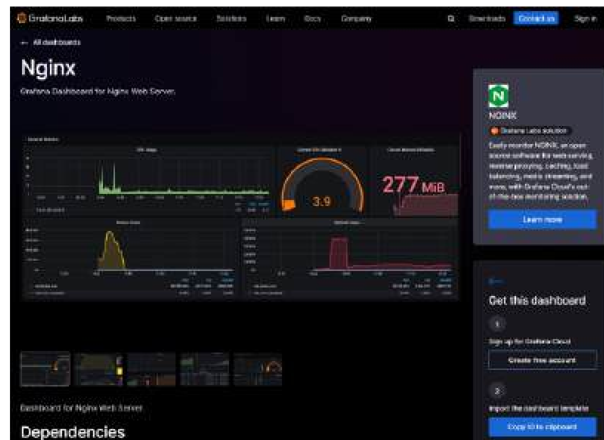

Рисунок 3.64 – Сторінка на офіційному сайті Grafana з шаблонами дошок. Номер шаблону було скопійовано для подальшого імпорту.

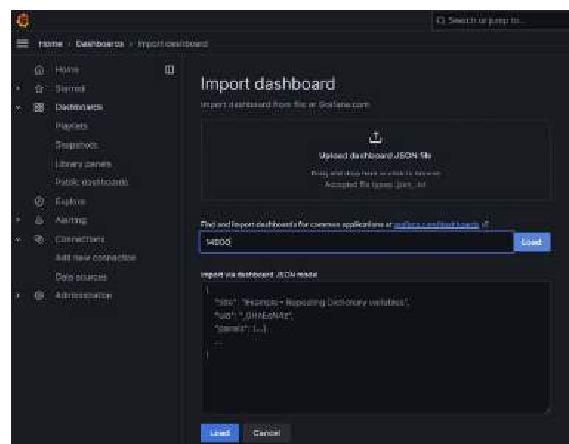

Рисунок 3.65 – Імпорт дошки через вказання номеру шаблону у Grafana.

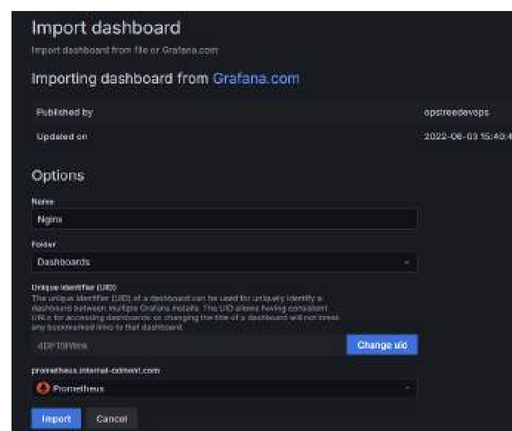

Рисунок 3.66 – Обрання назви дошки та Prometheus в якості джерела даних.

Рисунок 3.67 – Дані про використання ресурсів подами фронтенду.

Рисунок 3.68 – Графіки з кількості запитів за кодом: 2XX, 3XX, 4XX, 5XX.

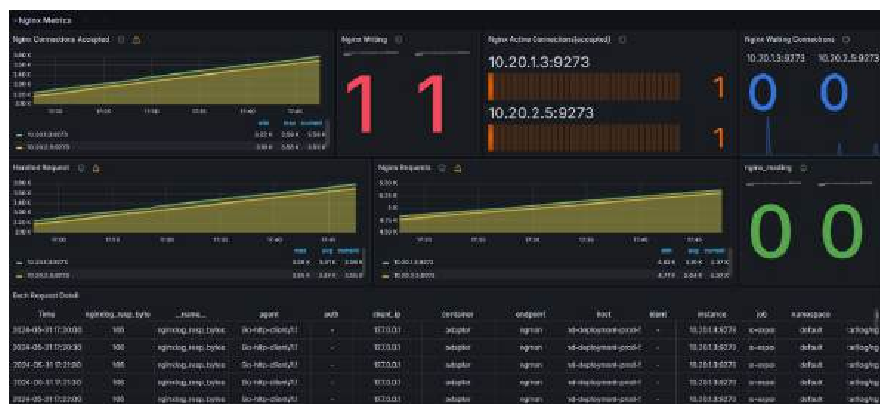

Рисунок 3.69 – Статистика за загальною кількістю запитів, затримкою Nginx, кількістю читання та запису. Наприкінці список останніх запитів до веб-серверів.

Було впроваджено комплексний моніторинг Kubernetes-кластера за допомогою kube-prometheus-stack з Grafana та Prometheus. Це забезпечило глибоке розуміння ресурсів кластера і дозволило оперативно реагувати на проблеми. Моніторинг фронтенду з Nginx та Telegraf надав детальну картину роботи веб-сервера, підвищивши контроль.

Висновки за розділом

У цьому розділі розглянуто створення комплексної та масштабованої інфраструктури для веб-додатків, від локального розгортання до хмарних рішень.

На першому етапі створено локальну інфраструктуру з використанням гіпервізора, де були розгорнуті екземпляри фронтенду і бекенду додатка. Це дало основні концепції та практичний досвід роботи з віртуальними машинами.

Другий етап передбачав контейнеризацію додатків за допомогою Docker та їх розгортання у Google Cloud Platform з використанням Terraform. Контейнеризовані додатки були розгорнуті на GCE, що забезпечило гнучкість та масштабованість.

Наступним кроком було створення кластера Kubernetes у GCP. У кластері розгорнуто контейнеризований застосунок для декількох середовищ, що забезпечило переваги оркестрації контейнерів та автоматичного масштабування.

Після цього побудовано систему CI/CD за допомогою GitLab, що автоматизувало збірку, тестування та розгортання додатків, скоротивши час виходу нових версій та підвищивши якість ПЗ.

Завершальним етапом стало налаштування моніторингу за допомогою Prometheus, Grafana та Telegraf для відстеження працездатності додатків і кластера Kubernetes. Це дозволило здійснювати детальний моніторинг ресурсів і вчасно реагувати на проблеми.

Таким чином, поступовий та систематичний підхід до розгортання інфраструктури дозволив успішно впровадити сучасні технології DevOps, забезпечуючи надійність, масштабованість та ефективність роботи веб-

додатків.

4 ТЕСТУВАННЯ ТА ВЕРИФІКАЦІЯ

4.1 Функції інфраструктури, які необхідно протестувати

В рамках оцінки надійності та продуктивності створеної DevOps-інфраструктури необхідно провести комплексне тестування її ключових функцій. Це включає перевірку як штатних режимів роботи, так і реакцію системи на нестандартні ситуації.

Першочергово слід протестувати роботу GitLab CI при некоректних змінах у коді. Очікується, що CI-система не лише зупинить процес, але й надасть вичерпну діагностичну інформацію. Перевіряється коректність роботи конвеєру CI/CD при успішному сценарії. Код повинен пройти всі стадії: компіляцію, юніт-тести, збірку Docker-образів, розгортання в dev-середовищі Kubernetes для інтеграційних тестів і, нарешті, оновлення production-середовища без простоїв.

4.2 Верифікація інфраструктури

Верифікація – це процес в інженерії програмного забезпечення, спрямований на підтвердження того, що система побудована правильно. На відміну від валідації, яка оцінює, чи відповідає система потребам користувачів, верифікація зосереджена на технічній точності та відповідності специфікаціям. Це систематичне оцінювання артефактів системи для визначення їх повної відповідності проєкту, без відхилень чи помилок у реалізації.

Далі верифікація зосереджується на послідовності та цілісності процесів. Перевіряється, чи не порушується ланцюг CI/CD: від комміту в GitLab через збірку та тестування до розгортання в Kubernetes.

Узагальнені результати верифікації представлено у таблиці 4.1.

Таблиця 4.1 – Результати верифікації

№	Найменування	Вимога	Підсумок	Результат
1	Створення інфраструктури у Kubernetes	Розгорнути застосунок у Kubernetes кластері	Застосунок успішно розгорнутий у Kubernetes за допомогою manifest файлів	Виконано
№	Найменування	Вимога	Підсумок	Результат
2	Моніторинг інфраструктури	Налаштувати моніторинг ресурсів Kubernetes	Встановлено Prometheus та Grafana для збору метрик та візуалізації	Виконано
3	Вхід до застосунку через Ingress	Забезпечити доступ до застосунку через Ingress	Ingress налаштований, застосунок доступний за доменом	Виконано
4	Створення CI/CD для фронтенду	Автоматизувати процес збірки та розгортання фронтенду	CI/CD налаштований в GitLab CI для фронтенду	Виконано
5	Створення CI/CD для бекенду	Автоматизувати процес збірки та розгортання бекенду	CI/CD налаштований в GitLab CI для бекенду	Виконано
6	Створення інфраструктури у хмарі з використанням Terraform	Створити інфраструктуру у GCP за допомогою Terraform	Інфраструктура успішно створена в GCP відповідно до Terraform конфігурацій	Виконано
7	Створення інфраструктури локально	Розгорнути локальну інфраструктуру на віртуальних машинах	Локальна інфраструктура розгорнута на VirtualBox з фронтендом, бекендом, базою даних та сховищем	Виконано

4.3 Тестування інфраструктури та CI/CD

Тестування програмного забезпечення – це багатогранний процес, який має на меті оцінку якості продукту. Цей процес включає спостереження за роботою ПЗ в різних ситуаціях та використання обмеженого набору тестів, обраних з певною метою. Сюди входить планування, розробка тестових сценаріїв, аналіз отриманих результатів, а також оцінка критеріїв завершення тестування. Крім того, процес включає написання звітів, рецензування документації та проведення статичного аналізу.

Тест-кейс є важливою складовою процесу тестування. Він включає детальний опис кроків, умов та параметрів, які необхідні для перевірки конкретної функції або її частини. Тестовий кейс дозволяє тестувальникам чітко розуміти, що і як слід перевіряти, забезпечуючи при цьому структурованість і повторюваність тестування. Завдяки цьому артефакту можна легко виявити помилки на ранніх етапах розробки, забезпечуючи високу якість кінцевого продукту. Тестові кейси також слугують документацією, яка може бути корисною для майбутніх тестувань і підтримки системи.

Вкладення:

```
FROM node:21-alpine as frontend-build

WORKDIR /app

COPY . .

ARG API_URL="http://dev.api.qgfq.eu/api/"
ENV REACT_APP_API_URL=$API_URL

RUN npm update -g npm \
    && npm ci --no-audit --maxsockets 1

RUN npm run build

RUN npm run test

FROM nginx:latest

COPY --from=frontend-build /app/build/ /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Рисунок 4.1 – Виконання unit-тестування у Dockerfile.

Тестування інфраструктури та pipeline CI/CD описано в таблицях 4.2 - 4.4:

Таблиця 4.2 – Тест-кейс «Виконання перевірки працездатності збірки фронтенду»

Назва перевірки:	Виконання перевірки працездатності збірки фронтенду
Середа та версії:	Windows 10 Pro 22H2, macOS 11, Chrome - 123.0.6312.105/106
Дії перед кроками:	Створити CI/CD для фронтенду
Кроки	**Очікувані результати**
1. У проєкті GitLab «Frontend» треба перейти до репозиторію з вихідним кодом;	1. На сторінці репозиторію має бути вихідний код фронтенду та вкладка «Pipeline»
2. запустити pipeline для «Frontend» у вкладці «Pipeline»;	2. Під час збірки фронтенду у логах буде повідомлення про проходження unit-тестів, а після неї почнеться збереження релізу фронтенду до Docker Hub.
3. Перейти до кластеру Kubernetes та перевірити список Pod.	3. У кластері Kubernetes з’являються нові поди з новим релізом фронтенду.

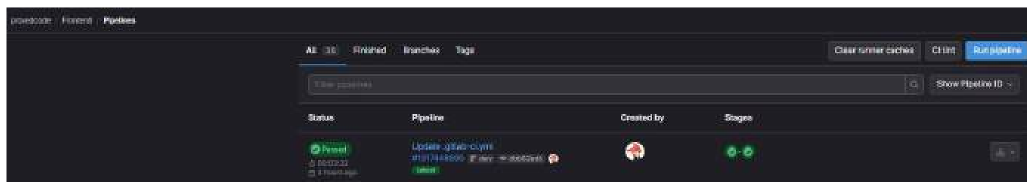

Рисунок 4.2 – Виконання pipeline для збірки, тестування та деплою фронтенду додатка, натискання на кнопку «Run pipeline».

```
151 #12 103.6
152 #12 103.6 added 2092 packages in 2m
153 #12 103.6
154 #12 103.6 244 packages are looking for funding
155 #12 103.6   run `npm fund` for details
156 #12 DONE 104.7s
157 #13 [frontend-build 5/6] RUN npm run test
158 #13 0.340
159 #13 0.340 > provedcode_st_1@0.1.0 test
160 #13 0.340 > node test.js
161 #13 0.340
162 #13 2.385 5/5 test(s) passed.
```

Рисунок 4.3 – Пройдені Unit-тести після збірки фронтенду.

```
provedcode-frontend-deployment-dev-55bf9448c7-klnnq     2/2   Running   0   3m57s
provedcode-frontend-deployment-dev-55bf9448c7-q8hqz     2/2   Running   0   3m44s
provedcode-frontend-deployment-prod-85c5fcf97f-2wvnc    2/2   Running   0   5h3m
provedcode-frontend-deployment-prod-85c5fcf97f-5r5w5    2/2   Running   0   5h3m
```

Рисунок 4.4 – Створені нові поди після збереження нової версії застосунку.

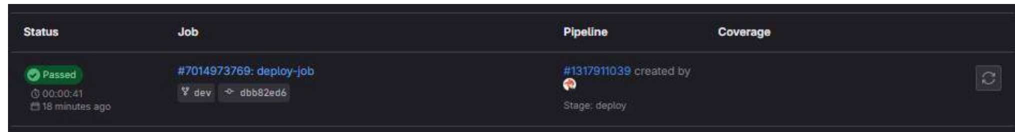

Рисунок 4.5 – Статус про успішне виконання pipeline.

Таблиця 4.3 – Тест-кейс «Виконання перевірки працездатності збірки бекенду»

Назва перевірки:	Виконання перевірки працездатності збірки бекенду
Середа та версії:	Windows 10 Pro 22H2, macOS 11, Chrome - 123.0.6312.105/106
Дії перед кроками:	Створити CI/CD для бекенду
Кроки	**Очікувані результати**
1. У проєкті GitLab «Backend» треба перейти до репозиторію з вихідним кодом;	1. На сторінці репозиторію має бути вихідний код бекенду та вкладка «Pipeline»
2. запустити pipeline для «Backend» у вкладці «Pipeline»;	2. Під час збірки бекенду у логах буде повідомлення про проходження unit-тестів, після неї почнеться збереження релізу бекенду до Docker Hub.
3. Перейти до кластеру Kubernetes та перевірити список Pod.	3. У кластері Kubernetes з’являються нові поди з новим релізом бекенду.

Вкладення:

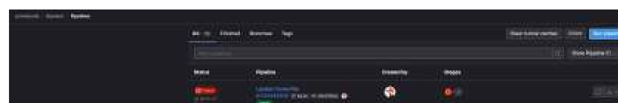

Рисунок 4.6 – Виконання pipeline для збірки, тестування та деплою бекенду додатка, натискання на кнопку «Run pipeline».

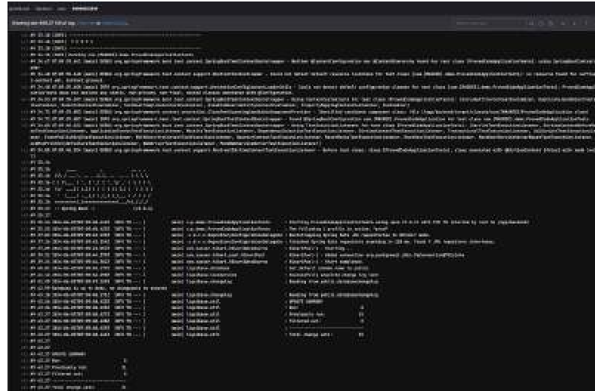

Рисунок 4.7 – Виконання застосунку у тестовому режимі під час виконання CI/CD.

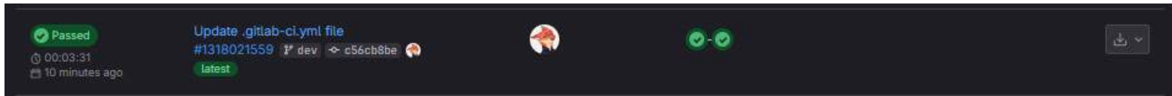

Рисунок 4.8 – Запуск нової поди з бекендом.

Рисунок 4.9 – Успішний статус про виконання збірки.

Таблиця 4.4 – Тест-кейс «Перевірка розгортання застосунку Proved Code на декількох середовищах»

Назва перевірки:	Перевірка розгортання застосунку Proved Code на декількох середовищах
Середа та версії:	Windows 10 Pro 22H2, macOS 11, Chrome - 123.0.6312.105/106
Дії перед кроками:	Створити Helm-Chart застосунку «Proved Code»
Кроки	**Очікувані результати**
1. Виконати команду для Helm у кластері Kubernetes: helm install provedcode-production ProvedCode-Deploy/ --set container.backendImage=mnemov/provedcode-backend:latest,container.backendReplicas=1	1. Мають створитись одразу усі необхідні ресурси для роботи «Proved Code» для середовища «production». Для

	середовища має автоматично створитись Ingress з публічною IP-адресою.
2. Виконати команду для Helm у кластері Kubernetes: helm install provedcode-development ProvedCode-Deploy/ --set enviroment=dev,ingress.frontendUrl=dev.x.qqfq.eu,ingress.backendUre=dev.api.qqfq.eu,container.backendImage=mnemov/provedcode-backend:go	2. Мають створитись одразу усі необхідні ресурси для роботи «Proved Code» для середовища «development». Для середовища має автоматично створитись Ingress з публічною IP-адресою.
3. Прив’язати публічні IP-адреси до доменних імен у Cloudflare	3. За утилітою ping мають на доменні імена мають з’явитись IP-адреси Ingress додатків.
4. Зайти за доменним іменем до сторінок для «production» та «development» середовищ.	4. До сторінок є доступ, при переході за доменним іменем.

Вкладення:

Рисунок 4.10 – Створення «production» та «development» копій застосунку через Helm у терміналі.

```
ubuntu@craft:~/helm-infra$ kubectl describe ingress
Name:             provedcode-ingress-hosts-dev
Labels:           app.kubernetes.io/managed-by=Helm
Namespace:        default
Address:          34.160.48.188
Ingress Class:    <none>
Default backend:  <default>
Rules:
  Host             Path  Backends
  ----             ----  --------
  dev.api.qqfq.eu
                   /   provedcode-backend-service-dev:80 (10.20.0.34:80,10.20.1.68:80)
  dev.x.qqfq.eu
                   /   provedcode-frontend-service-dev:80 (10.20.0.33:80,10.20.1.67:80)
Annotations:       ingress.kubernetes.io/backends:
                     {"k8s1-42bfe4c-default-provedcode-backend-service-dev-8-e35e90cb":"HEALTHY","k8s1-42bfe4c-default-provedcode-frontend-service-de-8-c85aa...
                   ingress.kubernetes.io/forwarding-rule: k8s2-fr-g8a9bzxt-default-provedcode-ingress-hosts-dev-fd0yecyg
                   ingress.kubernetes.io/target-proxy: k8s2-tp-g8a9bzxt-default-provedcode-ingress-hosts-dev-fd0yecyg
                   ingress.kubernetes.io/url-map: k8s2-um-g8a9bzxt-default-provedcode-ingress-hosts-dev-fd0yecyg
                   meta.helm.sh/release-name: provedcode-development
                   meta.helm.sh/release-namespace: default
Events:
  Type    Reason     Age                 From                     Message
  ----    ------     ----                ----                     -------
  Normal  Sync       55s                 loadbalancer-controller  UrlMap "k8s2-um-g8a9bzxt-default-provedcode-ingress-hosts-dev-fd0yecyg" created
  Normal  Sync       52s                 loadbalancer-controller  TargetProxy "k8s2-tp-g8a9bzxt-default-provedcode-ingress-hosts-dev-fd0yecyg" created
  Normal  Sync       39s                 loadbalancer-controller  ForwardingRule "k8s2-fr-g8a9bzxt-default-provedcode-ingress-hosts-dev-fd0yecyg" created
  Normal  IPChanged  39s                 loadbalancer-controller  IP is now 34.160.48.188
  Normal  Sync       32s (x4 over 4m44s) loadbalancer-controller  Scheduled for sync


Name:             provedcode-ingress-hosts-prod
Labels:           app.kubernetes.io/managed-by=Helm
Namespace:        default
Address:          34.111.106.189
Ingress Class:    <none>
Default backend:  <default>
Rules:
  Host         Path  Backends
  ----         ----  --------
  api.qqfq.eu
               /   provedcode-backend-service-prod:80 ()
  x.qqfq.eu
               /   provedcode-frontend-service-prod:80 (10.20.0.31:80,10.20.1.66:80)
Annotations:   ingress.kubernetes.io/backends:
                 {"k8s1-42b6fe4c-default-provedcode-backend-service-pro-8-4f18e472":"UNHEALTHY","k8s1-42b6fe4c-default-provedcode-frontend-service-pr-8-288...
               ingress.kubernetes.io/forwarding-rule: k8s2-fr-g8a9bzxt-default-provedcode-ingress-hosts-prod-dorz82by
               ingress.kubernetes.io/target-proxy: k8s2-tp-g8a9bzxt-default-provedcode-ingress-hosts-prod-dorz82by
               ingress.kubernetes.io/url-map: k8s2-um-g8a9bzxt-default-provedcode-ingress-hosts-prod-dorz82by
               meta.helm.sh/release-name: provedcode-production
               meta.helm.sh/release-namespace: default
Events:
  Type    Reason     Age                 From                     Message
  ----    ------     ----                ----                     -------
  Normal  Sync       2m36s               loadbalancer-controller  UrlMap "k8s2-um-g8a9bzxt-default-provedcode-ingress-hosts-prod-dorz82by" created
  Normal  Sync       2m33s               loadbalancer-controller  TargetProxy "k8s2-tp-g8a9bzxt-default-provedcode-ingress-hosts-prod-dorz82by" created
  Normal  Sync       2m19s               loadbalancer-controller  ForwardingRule "k8s2-fr-g8a9bzxt-default-provedcode-ingress-hosts-prod-dorz82by" created
  Normal  IPChanged  2m19s               loadbalancer-controller  IP is now 34.111.106.189
  Normal  Sync       48s (x5 over 5m12s) loadbalancer-controller  Scheduled for sync
```

Рисунок 4.11 – Отримання публічних адрес для «production» та «development» копій застосунку.

Type ▲	Name	Content	Proxy status	TTL	Actions
A	api	34.111.106.189	DNS only	Auto	Edit ▶
A	dev.api	34.160.48.188	DNS only	Auto	Edit ▶
A	dev.x	34.160.48.188	DNS only	Auto	Edit ▶
A	d	34.116.150.175	Proxied	Auto	Edit ▶
A	grafana	35.234.73.170	DNS only	Auto	Edit ▶
A	h	129.151.222.203	DNS only	Auto	Edit ▶
A	x	34.111.106.189	DNS only	Auto	Edit ▶
CNAME	www	qqfq.eu	DNS only	Auto	Edit ▶

Рисунок 4.12 – Прив’язка публічних адрес фронтенду та бекенду для «production» та «development» копій застосунку.

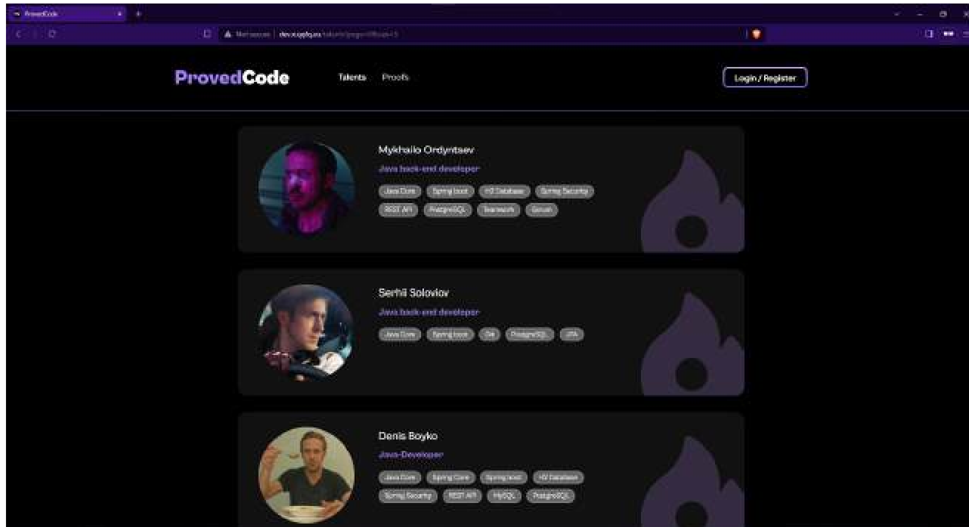

Рисунок 4.13 – Сторінка Proved Code середовища «development».

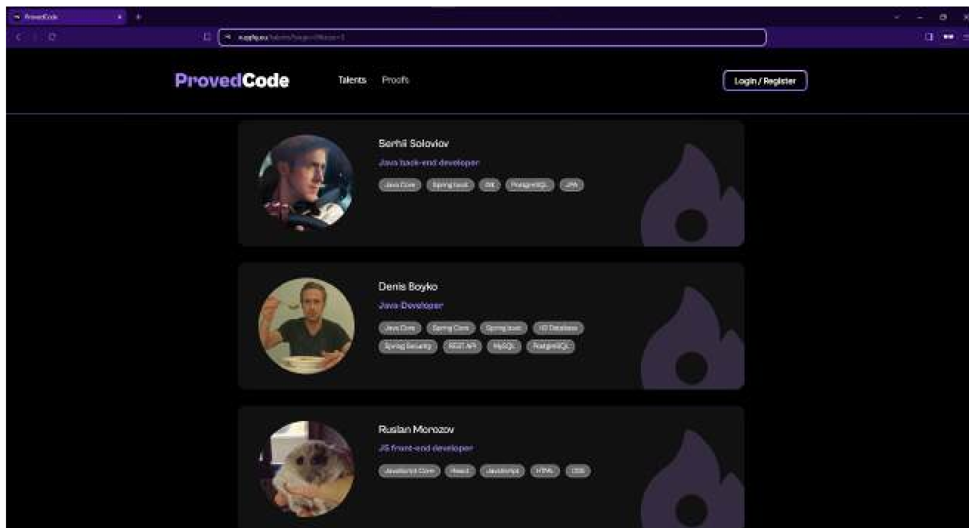

Рисунок 4.14 – Сторінка Proved Code середовища «production».

Висновки за розділом

Проведене комплексне тестування DevOps-інфраструктури, зосереджене на GitLab CI та процесах CI/CD, виявило стійкість та адаптивність системи. Верифікація інфраструктури, визначеної кодом, показала точну відповідність між специфікаціями та реальною системою. Успішне проходження кожного етапу CI/CD, від комміту через тестування до розгортання в Kubernetes – підтверджує цілісність та послідовність процесів, забезпечуючи надійне та безперебійне оновлення додатків.

ВИСНОВКИ

У процесі дослідження було встановлено, що створення ефективної та масштабованої інфраструктури для автоматизації життєвого циклу контейнеризованого веб-застосунку є надзвичайно важливим. Основна мета роботи полягала в розробці та впровадженні засобів для безперервної інтеграції, доставки, розгортання та моніторингу застосунку. Усі етапи розробки були ретельно проаналізовані та описані, включаючи вибір відповідних інструментів, налаштування системи безперервної інтеграції та моніторингу, а також тестування та верифікацію інфраструктури.

Проєкт довів, що використання сучасних технологій, таких як Kubernetes та Docker, дозволяє значно підвищити ефективність процесу розгортання додатків. Автоматизація цих процесів знижує ризики людських помилок і скорочує час виходу нових версій програмного забезпечення на ринок. Впровадження системи моніторингу забезпечує своєчасне виявлення та вирішення проблем, що сприяє стабільній роботі застосунку.

Економічний аналіз показав, що інвестиції в автоматизацію розгортання та моніторингу застосунку є виправданими та приносять значні переваги в довгостроковій перспективі. Проєкт продемонстрував позитивний очікуваний прибуток завдяки зниженню витрат на ручні процеси та підвищенню продуктивності розробників.

Отже, робота підтверджує важливість впровадження сучасних методів автоматизації у процес розробки програмного забезпечення для підвищення ефективності, надійності та економічної вигоди.

ПЕРЕЛІК ПОСИЛАНЬ

1. Хто такий DevOps [Електронний ресурс]. URL: https://dou.ua/forums/topic/33309/
2. Jobs in the DevOps category [Electronic resource]. URL: https://jobs.dou.ua/vacancies/?category=DevOps
3. Benefits of Cloud Migration [Electronic resource]. URL: https://azure.microsoft.com/en-in/resources/cloud-computing-dictionary/benefits-of-cloud-migration
4. Yet Another Brief History of container(d) [Electronic resource]. URL: https://erzeghi.medium.com/yet-another-brief-history-of-container-d-2962eac9679e
5. Що таке Kubernetes? [Електронний ресурс]. URL: https://kubernetes.io/uk/docs/concepts/overview/what-is-kubernetes/
6. Методології управління проєктами, або Що таке Waterfall, Agile та Scrum [Електронний ресурс]. URL: https://devisu.ua/uk/stattia/metodologii-upravlinnya-proktami-abo-shcho-take-waterfall-agile-ta-scrum
7. Trello [Electronic resource]. URL: https://trello.com/
8. What is Infrastructure as Code (IaC)? [Electronic resource]. URL: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac
9. What is Terraform? [Electronic resource]. URL: https://developer.hashicorp.com/terraform/intro
10. Dockerfile on Windows [Electronic resource]. URL: https://learn.microsoft.com/uk-ua/virtualization/windowscontainers/manage-docker/manage-windows-dockerfile
11. What is YAML? [Electronic resource]. URL: https://www.redhat.com/en/topics/automation/what-is-yaml
12. ProvedCode [Electronic resource]. URL: https://github.com/ProvedCode
13. React [Electronic resource]. URL: https://react.dev/

14. JDK [Electronic resource].
URL: https://www.oracle.com/java/technologies/javase/jdk22-readme-downloads.html
15. PostgreSQL [Electronic resource].
URL: https://www.postgresql.org/
16. AWS S3 [Electronic resource].
URL: https://aws.amazon.com/s3/
17. Telegraf [Electronic resource].
URL: https://www.influxdata.com/time-series-platform/telegraf/
18. How to create a Monitoring Stack using Kube-Prometheus-stack [Electronic resource].
URL: https://medium.com/israeli-tech-radar/how-to-create-a-monitoring-stack-using-kube-prometheus-stack-part-1-eff8bf7ba9a9
19. Helm [Electronic resource].
URL: https://github.com/helm/helm/releases
20. Grafana Dashboard for Nginx [Electronic resource].
URL: https://grafana.com/grafana/dashboards/14900-nginx/
21. Shared-core machine types [Electronic resource].
URL: https://cloud.google.com/compute/vm-instance-pricing#sharedcore_machine_types

ДОДАТОК А

Вихідні коди

Розгортання інфраструктури у хмарі з контейнеризованим додатком

install_docker.sh

```
# Додавання офіційного ключа GPG Docker:
apt-get update
apt-get install ca-certificates curl
install -m 0755 -d /etc/apt/keyrings
curl -fsSL
https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
chmod a+r /etc/apt/keyrings/docker.asc
# Додавання репозиторію до джерел Apt:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/debian \
  $(. /etc/os-release && echo
"$VERSION_CODENAME") stable" | \
  tee /etc/apt/sources.list.d/docker.list > /dev/null
apt-get update
apt-get install docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin
```

Dokerfile-backend

```
FROM eclipse-temurin:17-jdk-alpine as backend-
build

WORKDIR /app

RUN apk add --no-cache git

RUN git clone
https://github.com/ProvedCode/backend.git \
   && cd backend \
   && git switch dev

ENV S3_REGION=eu-central-1 \
   BUCKET=provedcode-backend \
   S3_ACCESS_KEY=****** \
   S3_SECRET_KEY==******  \
   DB_URL=*.*.*.*:5432/provedcode \
   DB_LOGIN==****** \
   DB_PASSWORD==******  \
   SPRING_PROFILES_ACTIVE=prod \
   EMAIL_USER=proved.code.team@gmail.com \
   EMAIL_PASSWORD==******

RUN chmod 744 /app/backend/mvnw \
   && cd /app/backend \
   && ./mvnw clean package

FROM eclipse-temurin:17-jre

COPY --from=backend-build
/app/backend/target/*.jar /app/provedcode.jar

CMD ["java", "-jar", "/app/provedcode.jar", "--
server.port=8080"]
```

Dokerfile-frontend

```
FROM node:21-alpine as frontend-build

WORKDIR /app

RUN apk add --no-cache git \
   && apk add --no-cache curl

RUN git clone
https://github.com/ProvedCode/frontend.git

WORKDIR /app/frontend

RUN git switch dev \
   && ip=$(curl -s https://ifconfig.me) \
   && export BACKEND_URL=http://$(echo
$ip)/:8080 \
   && escaped_ip=$(echo $ip | sed 's/\./\\\./g') \
   && sed -i
"s/18\\.194\\.159\\.42:8081/$escaped_ip:8080/g"
src/services/api-services.js

RUN npm install \
   && npm run build

FROM nginx:latest
```

```
COPY --from=frontend-build /app/frontend/build/
/usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

docker-compose-provedcode

```
version: '3.1'

services:
  backend:
    build:
      context: ./backend
      dockerfile: Dokerfile-backend
    restart: always
    container_name: java-backend
    env_file:
      - ./backend/.env
    ports:
      - 8080:8080

  frontend:
    build:
      context: ./frontend
      dockerfile: Dokerfile-frontend
    restart: always
    container_name: node-frontend
    ports:
      - 80:80
    depends_on:
      - backend
```

Інфраструктура Terraform

Dockerfile-frontend-terraform

```
FROM node:21-alpine as frontend-build

WORKDIR /app

COPY . .

ARG REACT_APP_API_URL
ENV
REACT_APP_API_URL=$REACT_APP_API_URL

RUN npm update -g npm \
    && npm ci --no-audit --maxsockets 1 \
    && npm run build

FROM nginx:latest

COPY --from=frontend-build /app/build/
/usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

install_terraform.sh

```
mkdir terraform
cd terraform/
sudo apt-get update && sudo apt-get install -y
gnupg software-properties-common
wget -O- https://apt.releases.hashicorp.com/gpg | \
gpg --dearmor | \
sudo tee /usr/share/keyrings/hashicorp-archive-
keyring.gpg > /dev/null

gpg --no-default-keyring \
--keyring /usr/share/keyrings/hashicorp-archive-
keyring.gpg \
--fingerprint

echo "deb [signed-
by=/usr/share/keyrings/hashicorp-archive-
keyring.gpg] \
https://apt.releases.hashicorp.com $(lsb_release -
cs) main" | \
sudo tee /etc/apt/sources.list.d/hashicorp.list

sudo apt update
sudo apt-get install terraform
```

Основний модуль

main.tf

```
# Провайдер GCP
provider "google" {
   credentials = file(var.gcp_svc_key)
   project     = var.gcp_project
   region      = var.gcp_region
}

# Модуль "network"
module "network" {
  source      = "./modules/network"
  gcp_project = var.gcp_project
  gcp_region  = var.gcp_region
}

# Модуль "gce"
module "gce" {
  source         = "./modules/gce"
  provedcode_subnet =
module.network.provedcode_subnet
  gcp_project     = var.gcp_project
  gcp_zone        = var.gcp_zone
}
```

variables.tf

```
variable "gcp_svc_key" {
   description = "Ключ служби для GCP"
   type        = string
}

variable "gcp_project" {
   description = "Ідентифікатор проекту GCP"
   type        = string
}

variable "gcp_region" {
   description = "Регіон GCP"
   type        = string
}

variable "gcp_zone" {
   description = "Зона GCP"
   type        = string
}
```

terraform.tfvars

```
gcp_project = "pure-hue-391415"
gcp_svc_key = "./keys/pure-hue-391415-
3181d55ca060.json"
gcp_region = "europe-west3"
gcp_zone = "europe-west3-a"
```

Модуль мережі

main.tf

```
// VPC
resource "google_compute_network" "main" {
  name                    = "main-vpc"
  auto_create_subnetworks = false
  project                 = var.gcp_project
}

// Підмережа
resource "google_compute_subnetwork"
"provedcode-subnet" {
  name          = "provedcode-subnet"
  ip_cidr_range = "10.0.0.0/16"
  region        = var.gcp_region
  network       =
google_compute_network.main.self_link
  project       = var.gcp_project
}
```

variables.tf

```
variable "gcp_project" {
  description = "Ідентифікатор проекту GCP"
  type        = string
}

variable "gcp_region" {
  description = "Регіон GCP"
  type        = string
}
```

firewalls.tf

```
resource "google_compute_firewall"
"frontend_firewall" {
  name    = "allow-http-frontend"
  project = var.gcp_project
  network = google_compute_network.main.name

  allow {
    protocol = "tcp"
    ports    = ["80"]
  }

  source_ranges = ["0.0.0.0/0"]
  target_tags   = ["frontend"]
}

resource "google_compute_firewall"
"backend_firewall" {
  name    = "allow-http-backend"
  project = var.gcp_project
  network = google_compute_network.main.name

  allow {
    protocol = "tcp"
    ports    = ["8080"]
  }

  source_ranges = ["0.0.0.0/0"]
  target_tags   = ["backend"]
}

resource "google_compute_firewall" "ssh_firewall" {
  name    = "allow-ssh"
  project = var.gcp_project
  network = google_compute_network.main.name

  allow {
    protocol = "tcp"
    ports    = ["22"]
  }
  source_ranges = ["0.0.0.0/0"]
}
```

outputs.tf

```
output "provedcode_subnet" {
  description = "Самостійне посилання на
підмережу, яку використовують екземпляри"
  value       =
google_compute_subnetwork.provedcode-
subnet.self_link
}
```

Модуль екземплярів обчислення

main.tf

```
// Віртуальна машина frontend
resource "google_compute_instance" "frontend" {
  // Основна конфігурація машини
  name         = "tf-frontend"
  machine_type = "e2-small"
  zone         = var.gcp_zone
  tags         = ["frontend"]
  depends_on   =
[google_compute_instance.backend]

  // Скрипти запуску
  metadata_startup_script = join("\n", [
    file("scripts/install_docker.sh"),
    format("export BACKEND_IP=%s",
google_compute_instance.backend.network_interfa
ce[0].access_config[0].nat_ip),
    file("scripts/run_frontend.sh")
  ])

  // Конфігурація загрузочного диска
  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }

  // Конфігурація мережі
  network_interface {
    subnetwork = var.provedcode_subnet
    access_config {}
  }

  metadata = {
    ssh-keys =
"provedcode:${file("keys/id_rsa.pub")}"
  }
}

// Віртуальна машина backend
resource "google_compute_instance" "backend" {
  // Основна конфігурація машини
  name         = "tf-backend"
  machine_type = "e2-small"
  zone         = var.gcp_zone
  project      = var.gcp_project
  tags         = ["backend"]

  // Скрипти запуску
  metadata_startup_script = join("\n", [
    file("scripts/install_docker.sh"),
    file("scripts/env.sh"),
    file("scripts/run_backend.sh")
  ])

  // Конфігурація загрузочного диска
```

```
  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }

  // Конфігурація мережі
  network_interface {
    subnetwork = var.provedcode_subnet
      access_config {}
    }

    metadata = {
      ssh-keys =
"provedcode:${file("keys/id_rsa.pub")}"
    }
}
```

variables.tf

```
variable "gcp_project" {
  description = "Ідентифікатор проекту GCP"
  type        = string
}

variable "gcp_zone" {
  description = "Зона GCP"
  type        = string
}

variable "provedcode_subnet" {
  description = "Самостійне посилання на
підмережу, яку використовують екземпляри"
  type        = string
}
```


variables.tf

```
variable "gcp_project" {
  description = "Ідентифікатор проекту GCP"
  type        = string
}

variable "gcp_zone" {
  description = "Зона GCP"
  type        = string
}

variable "provedcode_subnet" {
  description = "Самостійне посилання на
підмережу, яку використовують екземпляри"
  type        = string
}
```


Виконувані скрипти при запуску віртуальних машин

install_docker.sh

```
#!/bin/bash

# Додати офіційний GPG ключ Docker:
apt-get update
apt-get install ca-certificates curl
install -m 0755 -d /etc/apt/keyrings
curl -fsSL
https://download.docker.com/linux/debian/gpg -o
/etc/apt/keyrings/docker.asc
chmod a+r /etc/apt/keyrings/docker.asc

# Додати репозиторій до джерел Apt:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/debian \
  $(. /etc/os-release && echo
"$VERSION_CODENAME") stable" | \
  tee /etc/apt/sources.list.d/docker.list > /dev/null
apt-get update

# Встановити Docker:
apt-get install docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin -y
```

env.sh

```
#!/bin/bash

export S3_REGION=eu-central-1
export BUCKET=provedcode-backend
export S3_ACCESS_KEY=******
export S3_SECRET_KEY=******
export DB_LOGIN=provedcode
export DB_PASSWORD=******
export DB_URL=*.*.*.*:5432/provedcode
export SPRING_PROFILES_ACTIVE=prod
export EMAIL_USER=******
```

```
export EMAIL_PASSWORD=******
```

run_backend.sh

```
#!/bin/bash

# Запустити Docker контейнер
docker pull mnemov/provedcode-backend
docker run \
-p 8080:8080 \
-e S3_REGION=$S3_REGION \
-e BUCKET=$BUCKET \
-e S3_ACCESS_KEY=$S3_ACCESS_KEY \
-e S3_SECRET_KEY=$S3_SECRET_KEY \
-e DB_LOGIN=$DB_LOGIN \
-e DB_PASSWORD=$DB_PASSWORD \
-e DB_URL=$DB_URL \
-e
SPRING_PROFILES_ACTIVE=$SPRING_PROFIL
ES_ACTIVE \
-e EMAIL_USER=$EMAIL_USER \
-e EMAIL_PASSWORD=$EMAIL_PASSWORD \
mnemov/provedcode-backend
```

run_frontend.sh

```
#!/bin/bash

# Клонувати репозиторій frontend з GitHub
apt-get install git
git clone --branch dev --depth 1
https://github.com/SvarshikPlaton/frontend.git
cd frontend

# Отримати аргумент API_URL
API_URL=http://${BACKEND_IP}:8080/api/
echo "API_URL: $API_URL"

# Побудувати Docker образ
docker build -t provedcode-frontend --build-arg
REACT_APP_API_URL=$API_URL --network=host
.

# Запустити Docker контейнер
docker run -p 80:80 -d provedcode-frontend
```

Розгортання контейнеризованного застосунку за допомогою Kubernetes

Dockerfile-frontend-k8s

```
FROM node:21-alpine as frontend-build

WORKDIR /app

COPY . .

ARG API_URL="http://api.qqfq.eu/api/"

ENV REACT_APP_API_URL=$API_URL

RUN npm update -g npm \
    && npm ci --no-audit --maxsockets 1 \
    && npm run build

FROM nginx:latest

COPY --from=frontend-build /app/build/
/usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

build_frontend.sh

```
docker build -t  provedcode-frontend . --platform linux/x86_64
docker tag provedcode-frontend mnemov/provedcode-frontend
docker push mnemov/provedcode-frontend
```

Dockerfile-backend-k8s

```
FROM eclipse-temurin:17-jdk-alpine

WORKDIR /app

RUN apk add --no-cache git
```

```
RUN git clone --branch dev
https://github.com/ProvedCode/backend.git

WORKDIR /app/backend

ARG S3_REGION
ARG BUCKET
ARG S3_ACCESS_KEY
ARG S3_SECRET_KEY
ARG DB_URL
ARG DB_LOGIN
ARG DB_PASSWORD
ARG SPRING_PROFILES_ACTIVE
ARG EMAIL_USER
ARG EMAIL_PASSWORD

ENV S3_REGION=$S3_REGION \
   BUCKET=$BUCKET \
   S3_ACCESS_KEY=$S3_ACCESS_KEY \
   S3_SECRET_KEY=$S3_SECRET_KEY \
   DB_URL=$DB_URL \
   DB_LOGIN=$DB_LOGIN \
   DB_PASSWORD=$DB_PASSWORD \
SPRING_PROFILES_ACTIVE=$SPRING_PROFIL
ES_ACTIVE \
   EMAIL_USER=$EMAIL_USER \
   EMAIL_PASSWORD=$EMAIL_PASSWORD

RUN chmod 744 /app/backend/mvnw \
   && cd /app/backend \
   && ./mvnw clean package

FROM eclipse-temurin:17-jre

COPY --from=backend-build
/app/backend/target/*.jar /app/provedcode.jar

CMD ["java", "-jar", "/app/provedcode.jar", "--
server.port=80"]
```

build_backend.sh

```
docker build -t  provedcode-backend . --platform
linux/x86_64 \
   --build-arg S3_REGION=$S3_REGION \
   --build-arg BUCKET=$BUCKET \
   --build-arg
S3_ACCESS_KEY=$S3_ACCESS_KEY \
   --build-arg
S3_SECRET_KEY=$S3_SECRET_KEY \
   --build-arg DB_URL=$DB_URL \
   --build-arg DB_LOGIN=$DB_LOGIN \
   --build-arg DB_PASSWORD=$DB_PASSWORD \
   --build-arg
SPRING_PROFILES_ACTIVE=$SPRING_PROFIL
ES_ACTIVE \
   --build-arg EMAIL_USER=$EMAIL_USER \
   --build-arg
EMAIL_PASSWORD=$EMAIL_PASSWORD

docker tag provedcode-backend
mnemov/provedcode-backend
docker push mnemov/provedcode-backend
```

secrets.sh

```
export S3_REGION=eu-central-1
export BUCKET=provedcode-backend
export S3_ACCESS_KEY=******
export S3_SECRET_KEY=******
export DB_URL=*.*.*.*:5432/provedcode
export DB_LOGIN=******
export DB_PASSWORD=******
export SPRING_PROFILES_ACTIVE=prod
export
EMAIL_USER=proved.code.team@gmail.com
export EMAIL_PASSWORD=******
```

provedcode-backend-secret.yaml


```
apiVersion: v1
kind: Secret
metadata:
  name: provedcode-backend-secret
type: Opaque
data:
  S3_REGION: ZXUtY2VudHJhbC0x
  BUCKET: cHJvdmVkY29kZS1iYWNrZW5k
  S3_ACCESS_KEY: ********
  S3_SECRET_KEY: ********
  DB_URL: ******
  DB_LOGIN: ******
  DB_PASSWORD: ********
  SPRING_PROFILES_ACTIVE: cHJvZA==
  EMAIL_USER:
cHJvdmVkLmNvZGUudGVhbUBnbWFpbC5jb20=
  EMAIL_PASSWORD: ******
```

provedcode-backend-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: provedcode-backend-deployment
  labels:
    app: provedcode-backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: provedcode-backend
  template:
    metadata:
      labels:
        app: provedcode-backend
    spec:
      containers:
      - name: provedcode-backend
        image: mnemov/provedcode-backend
        ports:
        - containerPort: 80
        livenessProbe:
          httpGet:
            path: /api/v2/talents?page=0&size=5
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /api/v2/talents?page=0&size=5
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 10
        env:
          - name: S3_REGION
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret
                key: S3_REGION
          - name: BUCKET
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret
                key: BUCKET
          - name: S3_ACCESS_KEY
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret
                key: S3_ACCESS_KEY
          - name: S3_SECRET_KEY
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret
                key: S3_SECRET_KEY
          - name: DB_URL
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret
                key: DB_URL
          - name: DB_LOGIN
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret
                key: DB_LOGIN
          - name: DB_PASSWORD
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret
                key: DB_PASSWORD
          - name: SPRING_PROFILES_ACTIVE
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret
                key: SPRING_PROFILES_ACTIVE
          - name: EMAIL_USER
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret
                key: EMAIL_USER
          - name: EMAIL_PASSWORD
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret
                key: EMAIL_PASSWORD
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - provedcode-backend
              topologyKey: "kubernetes.io/hostname"
```

provedcode-frontend-deployment.yaml»

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: provedcode-frontend-deployment
  labels:
    app: provedcode-frontend
spec:
```

```
  replicas: 2
  selector:
    matchLabels:
      app: provedcode-frontend
  template:
    metadata:
      labels:
        app: provedcode-frontend
    spec:
      containers:
      - name: provedcode-frontend
        image: mnemov/provedcode-frontend
        ports:
        - containerPort: 80
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - provedcode-frontend
              topologyKey: "kubernetes.io/hostname"
```

provedcode-ingress-hosts.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: provedcode-ingress-hosts
spec:
  rules:
  - host: api.qqfq.eu
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: provedcode-backend-deployment
            port:
              number: 80
  - host: x.qqfq.eu
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: provedcode-frontend-deployment
            port:
              number: 80
```

Розгортання застосунку через Helm у кластері Kubernetes

prometheus-values.yaml

```
prometheus:
  prometheusSpec:
    ruleSelectorNilUsesHelmValues: false
    serviceMonitorSelectorNilUsesHelmValues: false
    podMonitorSelectorNilUsesHelmValues: false
    probeSelectorNilUsesHelmValues: false
```

nginx-config-map.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-configmap-{{ .Values.environment }}
data:
  default.conf: |
    server {
      listen 80;
      server_name _;

      location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
        error_page 404 = /;
      }
```

```
    location /nginx_status {
      stub_status on;
      allow 127.0.0.1;
      deny all;
    }
  }
```

provedcode-backend-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: provedcode-backend-deployment-{{
.Values.environment }}
  labels:
    app: provedcode-backend-{{ .Values.environment
}}
spec:
  replicas: {{ .Values.container.backendReplicas }}
  selector:
    matchLabels:
      app: provedcode-backend-{{
.Values.environment }}
  template:
    metadata:
      labels:
        app: provedcode-backend-{{
.Values.environment }}
    spec:
      containers:
      - name: provedcode-backend
        image: {{ .Values.container.backendImage }}
        imagePullPolicy: Always
        ports:
        - containerPort: 80
        livenessProbe:
          httpGet:
            path: /api/v2/talents?page=0&size=5
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /api/v2/talents?page=0&size=5
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 10

        env:
          - name: S3_REGION
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret-{{
.Values.environment }}
                key: S3_REGION
          - name: BUCKET
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret-{{
.Values.environment }}
                key: BUCKET
          - name: S3_ACCESS_KEY
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret-{{
.Values.environment }}
                key: S3_ACCESS_KEY
          - name: S3_SECRET_KEY
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret-{{
.Values.environment }}
                key: S3_SECRET_KEY
          - name: DB_URL
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret-{{
.Values.environment }}
                key: DB_URL
          - name: DB_LOGIN
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret-{{
.Values.environment }}
                key: DB_LOGIN
          - name: DB_PASSWORD
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret-{{
.Values.environment }}
                key: DB_PASSWORD
          - name: SPRING_PROFILES_ACTIVE
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret-{{
.Values.environment }}
                key: SPRING_PROFILES_ACTIVE
          - name: EMAIL_USER
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret-{{
.Values.environment }}
                key: EMAIL_USER
          - name: EMAIL_PASSWORD
            valueFrom:
              secretKeyRef:
                name: provedcode-backend-secret-{{
.Values.environment }}
```

```
            key: EMAIL_PASSWORD

      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - provedcode-backend-{{ .Values.environment }}
              topologyKey: "kubernetes.io/hostname"
---
apiVersion: v1
kind: Service
metadata:
  name: provedcode-backend-service-{{ .Values.environment }}
  labels:
    app: provedcode-backend-{{ .Values.environment }}
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
      name: http
  selector:
    app: provedcode-backend-{{ .Values.environment }}
```

provedcode-backend-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: provedcode-backend-secret-{{ .Values.environment }}
type: Opaque
data:
  S3_REGION: *
  BUCKET: *
  S3_ACCESS_KEY: *
  S3_SECRET_KEY: *
  DB_URL: *
  DB_LOGIN: *
  DB_PASSWORD: *
  SPRING_PROFILES_ACTIVE: *
  EMAIL_USER: *
  EMAIL_PASSWORD: *
```

helm-provedcode-frontend-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: provedcode-frontend-deployment-{{ .Values.environment }}
  labels:
    app: provedcode-frontend-{{ .Values.environment }}
spec:
  replicas: {{ .Values.container.frontendReplicas }}
  selector:
    matchLabels:
      app: provedcode-frontend-{{ .Values.environment }}
  template:
    metadata:
      labels:
        app: provedcode-frontend-{{ .Values.environment }}

    spec:
      containers:
      - name: provedcode-frontend
        imagePullPolicy: Always
        image: {{ .Values.container.frontendImage }}
        ports:
        - containerPort: 80
        volumeMounts:
          - name: nginx-config-{{ .Values.environment }}
            mountPath: /etc/nginx/conf.d/default.conf
            subPath: default.conf
          - name: nginx-logs-{{ .Values.environment }}
            mountPath: /var/log/nginx

      - name: adapter
        image: mnemov/telegraf-monitoring
        ports:
        - containerPort: 9273
        volumeMounts:
          - name: nginx-logs-{{ .Values.environment }}
            mountPath: /var/log/nginx

      volumes:
      - name: nginx-config-{{ .Values.environment }}
        configMap:
          name: nginx-configmap-{{ .Values.environment }}
          items:
```

```
      - key: default.conf
        path: default.conf
    - name: nginx-logs-{{ .Values.environment }}
      emptyDir: {}

    affinity:
      podAntiAffinity:

preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - provedcode-frontend
            topologyKey: "kubernetes.io/hostname"
---
apiVersion: v1
kind: Service
metadata:
  name: provedcode-frontend-service-{{
.Values.environment }}
  labels:
    app: provedcode-frontend-{{ .Values.environment
}}
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
      name: http
  selector:
    app: provedcode-frontend-{{ .Values.environment
}}
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-prometheus-exporter-service-{{
.Values.environment }}
  labels:
    app: nginx-prometheus-exporter-{{
.Values.environment }}
spec:
  type: ClusterIP
  ports:
    - port: 8080
      targetPort: 9273
      protocol: TCP
      name: ngmon
  selector:
    app: provedcode-frontend-{{ .Values.environment
}}
```

provedcode-ingress-hosts.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: provedcode-ingress-hosts-{{
.Values.environment }}
spec:
  rules:
  - host: {{ .Values.ingress.backendUrl }}
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: provedcode-backend-service-{{
.Values.environment }}
            port:
              number: 80
  - host: {{ .Values.ingress.frontendUrl }}
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: provedcode-frontend-service-{{
.Values.environment }}
            port:
              number: 80
```

service_monitor.yaml

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: provedcode-frontend-service-monitor-{{
.Values.environment }}
spec:
  selector:
    matchLabels:
      app: nginx-prometheus-exporter-{{
.Values.environment }}
  endpoints:
  - port: ngmon
    path: /metrics
    interval: 15s
```

Chart.yaml

```
apiVersion: v2
name: ProvedCode
description: The Helm chart for deploying the
ProvedCode application on Kubernetes
type: application
version: 0.1.0
appVersion: 1.16.0
keywords:
  - java
  - nodejs
  - nginx
  - telegraf
  - prometheus
maintainers:
  - name: Nikita Nemov
```

values.yaml

```
container:
  backendImage: "mnemov/provedcode-
backend:go"
  frontendImage: "mnemov/provedcode-
frontend:latest"
  backendReplicas: 2
  frontendReplicas: 2
  healthCheckPath: "/api/v2/talents?page=0&size=5"
ingress:
  frontendUrl: "x.qqfq.eu"
  backendUrl: "api.qqfq.eu"
environment: "prod"
```

loki-values.yaml

```
prometheus:
  enabled: true
  isDefault: false
  url: http://{{ include "prometheus.fullname" .}}:{{
.Values.prometheus.server.service.servicePort }}{{
.Values.prometheus.server.prefixURL }}
  datasource:
    jsonData: "{}"
loki:
  enabled: true
  isDefault: true
  url: http://{{(include "loki.serviceName" .)}}:{{
.Values.loki.service.port }}
  readinessProbe:
    httpGet:
      path: /ready
      port: http-metrics
    initialDelaySeconds: 45
  livenessProbe:
    httpGet:
      path: /ready
      port: http-metrics
    initialDelaySeconds: 45
  datasource:
    jsonData: "{}"
    uid: ""
promtail:
  enabled: true
  config:
    logLevel: info
    serverPort: 3101
    clients:
      - url: http://{{ .Release.Name
}}:3100/loki/api/v1/push

grafana:
  enabled: true
  sidecar:
    datasources:
      label: ""
      labelValue: ""
      enabled: true
      maxLines: 1000
  image:
    tag: 10.0.2
```

build.sh

```
#/bin/bash
# Build the monitoring image
docker build -t telegraf-monitoring . --platform linux/x86_64
docker tag telegraf-monitoring mnemov/telegraf-monitoring
docker push mnemov/telegraf-monitoring
```

Dockerfile-telegraf

```
FROM telegraf:alpine
COPY telegraf.conf /etc/telegraf/telegraf.conf
RUN chmod 777 /etc/telegraf/telegraf.conf
EXPOSE 9273
```

telegraf.conf

```
[[inputs.nginx]]
   urls = ["http://localhost:80/nginx_status"]
   response_timeout = "15s"
[[inputs.tail]]
   name_override = "nginxlog"
   files = ["/var/log/nginx/access.log"]
   from_beginning = true
   pipe = false
   data_format = "grok"
   grok_patterns =
["%{COMBINED_LOG_FORMAT}"]
[[inputs.cpu]]
   percpu = true
[[inputs.disk]]
[[inputs.diskio]]
[[inputs.nstat]]
[[inputs.mem]]
[[inputs.system]]
[[outputs.prometheus_client]]
   listen = ":9273
```

Навчальне видання

Узун Дмитро Дмитрович,
Тецький Артем Григорович,
Немов Микита Русланович

ТЕХНОЛОГІЇ ДЕВОПС.
РЕАЛІЗАЦІЯ CI/CD ПРОЕКТА З ВІДКРИТИМ ВИХІДНИМ КОДОМ

Навчальний посібник

[Electronic resource]

Національний аерокосмічний університет ім. М.Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@ khai.edu

Свідоцтво про внесення суб’єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розпосюджувачів видавничої продукції,
серія ДК № 391, від 30.03.2001 р.