НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

В.о. завідувача кафедри ОТ
__________ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _________2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

на тему: Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері

Виконав: студент 4 курсу, групи ІО-16
(шифр групи)

Попов Михайло Володимирович ____________
(прізвище, ім’я, по батькові) (підпис)

Керівник асистент, Пономаренко А. М. ____________
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) асистент, Гончаренко О. О. ____________
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент асистент, Пустовіт Л. М. ____________
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.

Студент ____________________

(підпис)

Київ – 2025 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО"

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

"Комп'ютерні системи та мережі"

спеціальності 123 "Комп'ютерна інженерія"

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри ОТ
__________ Михайло НОВОТАРСЬКИЙ
(підпис)
"__" __________2025 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Попова Михайла Володимировича

1. Тема проєкту *Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері*

2. К*ерівник проєкту Пономаренко Артем Миколайович, асистент,*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від 23 травня 2025 року 1705-с

3. Термін здачі студентом закінченого проєкту *13 червня 2025 р.*

4. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

5. *Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)*

Розділ 1. Теоретичні основи мікросервісної архітектури та масштабованості веб-застосунків.

Розділ 2. Огляд технологій та інструментів для реалізації мікросервісів у Kubernetes-кластері.

Розділ 3. Проєктування та розробка мікросервісної системи.

Розділ 4. Тестування, дослідження масштабованості та аналіз ефективності розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) модель процесу обробки зображень, архітектурна модель системи, модель структури бази даних.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Гончаренко О. О.	07.10.24	04.06.25

7. Дата видачі завдання «7» жовтня 2024 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*20.12.2024-25.12.2024*	
2.	*Вивчення та аналіз завдання*	*08.01.2025-05.02.2025*	
3.	*Розробка архітектури та загальної структури системи*	*05.02.2025-15.02.2025*	
4.	*Розробка структур окремих підсистем*	*15.02.2025-26.02.2025*	
5.	*Програмна реалізація системи*	*26.02.2025-01.04.2025*	
6.	*Оформлення пояснювальної записки*	*01.04.2025-17.05.2025*	
7.	*Захист програмного продукту*	*03.06.2025*	
8.	*Передзахист*	*10.06.2025*	
9.	*Захист*	*21.06.2025*	

Студент-дипломник __________________________ Михайло ПОПОВ
(підпис)

Керівник проєкту __________________________ Артем ПОНОМАРЕНКО
(підпис)

АНОТАЦІЯ

У даній роботі було детально розглянуто концепцію мікросервісної архітектури та її застосування для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері. Окрему увагу приділено основним принципам роботи мікросервісів, їх взаємодії через API та використанню контейнеризації для організації ефективного управління сервісами. Також були проаналізовані основні переваги та недоліки мікросервісної архітектури в контексті високої доступності та масштабованості.На основі цього аналізу була розроблена мікросервісна система, яка забезпечує автоматичне масштабування веб-застосунків у Kubernetes-кластері. Було проведено дослідження ефективності роботи системи в залежності від різних конфігурацій та навантажень, а також здійснено налаштування автоматичного масштабування сервісів з урахуванням поточних вимог до ресурсів. Розроблена система дає можливість ефективно керувати веб-застосунками в умовах змінного навантаження та високих вимог до доступності.Програмний продукт був розроблений на мові Python та контейнеризований за допомогою Docker для подальшого розгортання в середовищі Kubernetes.

ANNOTATION

In this work, the concept of microservices architecture and its application for ensuring high scalability of web applications in a Kubernetes cluster were thoroughly discussed. Special attention was given to the core principles of microservices, their interaction via APIs, and the use of containerization for effective management of services. The main advantages and disadvantages of the microservices architecture in the context of high availability and scalability were also analyzed. Based on this analysis, a microservices system was developed to ensure the automatic scaling of web applications in a Kubernetes cluster. The system's performance was studied depending on various configurations and workloads, and automatic scaling of services was configured to meet the current resource demands. The developed system allows efficient management of web applications under changing loads and high availability requirements.The software product was developed in Python and containerized using Docker for deployment in a Kubernetes environment.

Справки	Формат	Значення	Найменування	Кіл. листів	№ екземп.	Додаток
			Документація загальна			
	A4	ІАЛЦ.467200.002 ТЗ	Мікросервісна система для	4		
			забезпечення високої			
			масштабованості веб-застосунків у			
			Kubernetes-кластері			
			Технічне завдання			
	A4	ІАЛЦ.467200.003 ПЗ	Мікросервісна система для	65		
			забезпечення високої			
			масштабованості веб-застосунків у			
			Kubernetes-кластері			
			Пояснювальна записка			
	A4	ІАЛЦ.467200.004 Д1	Мікросервісна система для	1		
			забезпечення високої			
			масштабованості веб-застосунків у			
			Kubernetes-кластері			
			Модель процесу обробки			
			зображень			
	A4	ІАЛЦ.467200.007 Д2	Мікросервісна система для	1		
			забезпечення високої			
			масштабованості веб-застосунків у			
			Kubernetes-кластері			
			Архітектурна модель системи			

Зм	Лист	№ докум.	Підп	Дата	ІАЛЦ.467200.001 ОА		
Розроб		Попов М .В.			Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері Опис альбому	Літ.	Аркуш	Аркушів
Перев		Пономаренко А.М.					1	2
						“КПІ ім. Ігоря Сікорського” ФІОТ ІО-16		

Справки	Формат	Значення	Найменування	Кіл. листів	№ екземп.	Додаток
	А4	*ІАЛЦ.467200.007 Д3*	Мікросервісна система для	1		
			забезпечення високої			
			масштабованості веб-застосунків у			
			Kubernetes-кластері			
			Модель структури бази даних			
	А4	*ІАЛЦ.467200.007 Д4*	Мікросервісна система для	19		
			забезпечення високої			
			масштабованості веб-застосунків у			
			Kubernetes-кластері			
			Текст програмного коду			

ТЕХНІЧНЕ ЗАВДАННЯ

ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері»

Київ – 2025

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ............................2
2. ПРИЧИНИ ДЛЯ РОЗРОБКИ..2
3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ 2
4. ІНФОРМАЦІЙНА БАЗА ТА ВИХІДНІ МАТЕРІАЛИ 3
5. ТЕХНІЧНІ ХАРАКТЕРИСТИКИ ТА ВИМОГИ 3
5.1 Функціональні та нефункціональні вимоги.......................................3
5.1 Визначення пріоритетів і ризиків 3
5.3 Функціональні та нефункціональні вимоги.......................................4
6. ЕТАПИ РОЗРОБКИ...4

Зм.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.467200.002 ТЗ		
Розробив		Попов М.В.			Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері Технічне завдання	Літ.	Аркуш	Аркушів
Перевірив		Пономаренко А.М.					1	4
Реценз.		Пустовіт Л.М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІО-16		
Н. Контр.		Гончаренко О. О.						
Затвердив		Новотарський М.А.						

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Цей проєкт присвячений створенню мікросервісної системи, яка дозволяє досягти високої масштабованості, гнучкості та відмовостійкості веб-застосунків, що працюють у хмарному середовищі Kubernetes.

2. ПРИЧИНИ ДЛЯ РОЗРОБКИ

Необхідність розробки зумовлена сучасними вимогами до гнучкості та ефективності веб-систем, які повинні адаптуватися до змінних умов навантаження та забезпечувати безперебійну роботу. Проєкт дозволить створити систему, яка автоматично масштабуватиметься залежно від навантаження.

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою розробки є створення надійної, гнучкої та масштабованої мікросервісної системи, яка працює в Kubernetes-кластері та забезпечує безперебійну роботу веб-застосунків.

Джерелом розробки є проєктування архітектури системи з використанням контейнеризації, впровадження механізмів автоматичного масштабування, забезпечення моніторингу та логування, налаштування CI/CD процесів, а також тестування системи в умовах змінного навантаження для оцінки її ефективності.

4. ІНФОРМАЦІЙНА БАЗА ТА ВИХІДНІ МАТЕРІАЛИ

Під час розробки системи використовуються сучасні відкриті технології та інструменти: Kubernetes, Docker, Helm, Prometheus, Grafana, Git, GitHub Actions (або інші CI/CD системи). Також залучаються офіційна документація Kubernetes, практичні рекомендації з розгортання мікросервісів у продакшн-середовищах

5. ТЕХНІЧНІ ХАРАКТЕРИСТИКИ ТА ВИМОГИ

5.1. ФУНКЦІОНАЛЬНІ МОЖЛИВОСТІ СИСТЕМИ

Розроблена система має виконувати такі вимоги:

- Підтримка автоматичного масштабування мікросервісів.
- Моніторинг стану контейнерів та кластеру.
- Інтеграція з CI/CD-процесами для автоматичного розгортання оновлень.
- Просте додавання або оновлення мікросервісів без зупинки всієї системи.
- Висока доступність та безперервна робота при змінному навантаженні

5.2. ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

- Kubernetes версії 1.20 або вище.
- Контейнерний рушій Docker або сумісний (наприклад, containerd).
- Helm для управління розгортаннями та конфігураціями.
- CI/CD-система (GitHub Actions, GitLab CI/CD, Jenkins тощо).

5.3. ВИМОГИ ДО АПАРАТНОГО ЗАБЕЗПЕЧЕННЯ

Для розробки та тестування:

- 4 ядра CPU.
- 8 ГБ оперативної пам’яті.

Для продуктивного середовища (мінімально):

- Масштабовані вузли відповідно до навантаження.
- Наявність балансувальника навантаження (наприклад, NGINX).
- Захищене з’єднання (SSL/TLS), резервне копіювання тощо.

6. ЕТАПИ РОЗРОБКИ

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту
1.	Затвердження теми проєкту	20.12.2024-25.12.2024
2.	Вивчення та аналіз завдання	08.01.2025-05.02.2025
3.	Розробка архітектури та загальної структури системи	05.02.2025-15.02.2025
4.	Розробка структур окремих підсистем	15.02.2025-26.02.2025
5.	Програмна реалізація системи	26.02.2025-01.04.2025
6.	Оформлення пояснювальної	01.04.2025-17.05.2025
7.	Захист програмного продукту	03.06.2025
8.	Передзахист	10.06.2025
9.	Захист	21.06.2025

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 3

ВСТУП 4

РОЗДІЛ 1. ОГЛЯД МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА ІНФРАСТРУКТУРИ ДЛЯ ХМАРНИХ СЕРВІСІВ 6

1.1 Сучасні підходи до побудови веб-систем 6

1.2 Огляд технологій для хмарної інфраструктури 7

1.3 Порівняльний аналіз сучасних інструментів 11

ВИСНОВОК ДО РОЗДІЛУ 1 13

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА ІНФРАСТРУКТУРНА ПІДГОТОВКА СИСТЕМИ 14

2.1 Аналіз вимог до мікросервісної системи 14

2.1.1 Функціональні та нефункціональні вимоги 14

2.1.2 Визначення пріоритетів і ризиків 15

2.2 Вибір архітектурного стилю та способу взаємодії 16

2.2.1 REST, gRPC та GraphQL: критерії вибору 16

2.2.2 Розподіл відповідальності між сервісами 18

2.3 Технологічний стек і засоби реалізації 19

2.3.1 Django, Docker, PostgreSQL: основні компонентии 19

2.3.2 Інструменти для безпеки та CI/CD 20

2.4 Оркестрація та конфігурація Kubernetes 22

2.4.1 Побудова середовища та керування конфігураціями 22

2.4.2 Налаштування кластеру, Secrets 23

ВИСНОВОК ДО РОЗДІЛУ 2 26

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ІНТЕГРАЦІЯ СИСТЕМИ 28
3.1 Побудова серверної логіки Django-застосунку....28
3.1.1 Архітектура проєкту: шаблон Model-View-Template 28
3.1.2 Обробка зображень і взаємодія з користувачем 30
3.2 Контейнеризація й Docker-інтеграція....32
3.2.1 Створення Dockerfile: образи та залежності....32
3.2.2 Команди запуску, .dockerignore і безпека....35
3.3 CI/CD: автоматизація життєвого циклу....37
3.3.1 Kubernetes: інтеграція, токени, secrets....37
3.3.2 GitLab CI: workflow, деплой....40
3.4 Тестування: надійність та масштабованість 42
3.4.1 Функціональне та інтерфейсне тестування 42
3.4.2 Навантажувальне тестування у Kubernetes....43
ВИСНОВОК ДО РОЗДІЛУ 3....46
РОЗДІЛ 4. ТЕХНІЧНА РЕАЛІЗАЦІЯ ІНФРАСТРУКТУРИ....48
4.1 Автоматизація розгортання через GitLab CI/CD 48
4.1.1 Побудова пайплайну: build, test, dockerize, deploy 48
4.1.2 Інтеграція з Kubernetes через YAML-манифести 49
4.2 Масштабування та балансування в Kubernetes....50
4.2.1 HPA та управління ресурсами 50
4.2.2 Ingress-контролер і балансування навантаження 53
4.3 Забезпечення безпеки інфраструктури....55
4.3.1 Конфігурація секретів і ролей доступу 55
4.3.2 Мережеві політики та безпечна взаємодія 57
4.4 Тестування: надійність та масштабованість 58
ВИСНОВОК ДО РОЗДІЛУ 4....60
ВИСНОВКИ....62

					ІАЛЦ.467200.003 ПЗ	Арк
						2
Зм.	Арк.	№ докум.	Підпис	Дата		

ПЕРЕЛІК СКОРОЧЕНЬ

- **API** – Application Programming Interface
- **CI/CD** – Continuous Integration / Continuous Deployment
- **CPU** – Central Processing Unit
- **DB** – Database
- **DBMS** – Database Management System
- **DNS** – Domain Name System
- **K8s** – Kubernetes
- **HTTP** – HyperText Transfer Protocol
- **JSON** – JavaScript Object Notation
- **Microservices** – Мікросервіси
- **RAM** – Random Access Memory
- **REST** – Representational State Transfer
- **SPA** – Single Page Application
- **YAML** – Yet Another Markup Language
- **ORM** – Object-Relational Mapping
- **IDE** – Integrated Development Environment
- **DNS** – Domain Name System

ВСТУП

Розвиток цифрових сервісів і зростаючі вимоги до доступності, гнучкості й продуктивності програмних рішень висувають нові стандарти до архітектури веб-застосунків. Усе більше компаній зіштовхуються з необхідністю створення систем, здатних стабільно функціонувати за умов змінного навантаження, підтримувати швидке оновлення функціоналу та забезпечувати високу відмовостійкість. У цьому контексті дедалі більшої актуальності набуває мікросервісний підхід до проєктування та реалізації програмного забезпечення. Мікросервісна архітектура передбачає побудову системи як сукупності незалежних сервісів, кожен з яких реалізує чітко окреслену бізнес-функцію та взаємодіє з іншими компонентами через стандартизовані інтерфейси. Щоб забезпечити ефективну роботу мікросервісної архітектури, доцільним є впровадження засобів контейнеризації (Docker) та оркестрації (Kubernetes), які дозволяють автоматизувати процеси розгортання, масштабування, оновлення та моніторингу.

Особливу роль у таких середовищах відіграє Kubernetes як універсальна платформа для управління життєвим циклом контейнеризованих сервісів. Завдяки своїй модульності, декларативному управлінню станом, а також інтеграції з CI/CD-процесами, ця система забезпечує гнучке розгортання та високу надійність навіть у разі складних міжсервісних взаємодій. У рамках цієї роботи було спроєктовано та реалізовано веб-застосунок, заснований на мікросервісному принципі, з фокусом на автоматизоване розгортання в кластері Kubernetes. Розробка охоплює побудову серверної логіки, контейнеризацію, налаштування пайплайну безперервної інтеграції та доставки (CI/CD), впровадження механізмів масштабування, а також моніторинг та логування.

Основним завданням стало створення інфраструктури, яка дозволяє ефективно керувати сервісами, гнучко адаптуватися до змін навантаження та мінімізувати вплив помилок на загальну стабільність системи.

Додатково у дослідженні розглянуто технічні аспекти взаємодії мікросервісів через REST API, специфіку конфігурацій у Kubernetes, а також типові виклики, пов'язані з забезпеченням цілісності даних, контролем версій сервісів та автоматичним відкатом у разі збоїв. Робота також містить аналіз переваг мікросервісної архітектури у порівнянні з традиційними монолітними підходами, із прикладами реалізації на практиці. Запропоноване рішення є універсальним каркасом для створення стійких до навантажень веб-застосунків, здатних масштабуватися відповідно до потреб бізнесу та швидко впроваджувати інновації.

РОЗДІЛ 1. ОГЛЯД МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА ІНФРАСТРУКТУРИ ДЛЯ ХМАРНИХ СЕРВІСІВ

1.1 Сучасні підходи до побудови веб-систем

Сучасні підходи до створення масштабованих і гнучких веб-систем базуються на принципах мікросервісної архітектури. Ця концепція передбачає розподіл єдиної системи на незалежні, вузькоспеціалізовані сервіси, кожен з яких реалізує окрему бізнес-функцію. Завдяки цьому досягається ізоляція компонентів, що спрощує процес розробки, обслуговування й оновлення окремих частин без ризику вплинути на всю систему. Організація мікросервісної структури дозволяє сформувати автономні команди, які можуть працювати незалежно, паралельно реалізуючи нові функції. Це сприяє швидкому впровадженню змін і адаптації до нових ринкових вимог. Зв'язок між сервісами зазвичай забезпечується через API або системи обміну повідомленнями, що дозволяє інтегрувати різні технології та платформи. Управління конфігураціями централізовано, що дає можливість гнучко реагувати на зміни середовища без перезапуску сервісів. Кожен мікросервіс зазвичай має власну базу даних, що підвищує незалежність, але також вимагає спеціальних підходів до узгодженості даних. Автоматизація процесів, таких як CI/CD і тестування, є обов'язковою складовою, оскільки дозволяє підтримувати якість системи й оперативно доставляти нові функціональні можливості. Графічна візуалізація архітектури часто використовується для полегшення аналізу та розуміння взаємозв'язків між сервісами, що допомагає у прийнятті технічних рішень.

Отже, мікросервісний підхід є ефективним засобом створення сучасних IT-рішень, що поєднує масштабованість, адаптивність і технологічну незалежність.

Популярність мікросервісної архітектури пояснюється її численними практичними перевагами. Насамперед, вона дозволяє розробляти окремі компоненти системи у відносній ізоляції, що скорочує час розробки та дозволяє командам працювати паралельно над різними сервісами. Це значно зменшує залежності між модулями та прискорює загальний цикл розгортання. Масштабування в мікросервісній архітектурі можливе на рівні окремих сервісів, що дозволяє ефективно використовувати ресурси та адаптуватися до зміни навантаження. Гнучка структура сприяє безпечному оновленню компонентів без ризику зупинки всієї системи. Разом із тим, реалізація мікросервісної моделі вимагає додаткових зусиль для координації між сервісами, обробки помилок і забезпечення сумісності API. Забезпечення цілісності даних у розподілених базах, підтримка журналювання, моніторинг і безпечна комунікація — усе це вимагає ретельної інженерної підготовки.

Також слід враховувати збільшені витрати на створення інфраструктури для оркестрації, CI/CD і захисту, особливо на початковому етапі впровадження. Водночас технологічна різноманітність, яку дозволяє мікросервісна архітектура, забезпечує високий рівень інновацій та адаптації. Кожен сервіс може бути реалізований на відповідній мові або фреймворку, що оптимально відповідає його задачам. Завдяки цьому команди можуть впроваджувати нові технології точково, не порушуючи цілісність системи. Крім цього, незалежність сервісів підвищує загальну надійність: збій одного не впливає на інші. Це особливо актуально для критичних сфер, як-от медицина чи фінанси. Таким чином, мікросервісний підхід поєднує інженерну ефективність із організаційною гнучкістю, забезпечуючи конкурентні переваги в динамічному IT-середовищі.

1.2 Огляд технологій для хмарної інфраструктури

Kubernetes, також відомий як K8s, — це потужне open-source рішення для автоматизації розгортання, масштабування та управління контейнеризо-ваними додатками. Його архітектура і концепції були значною мірою натхненні внутрішньою системою Google — Borg, що багато років використовувалась для управління мільйонами контейнерів у дата-центрах компанії. Kubernetes надає гнучкий і розширюваний підхід до оркестрації контейнерів, забезпечуючи високу надійність, масштабованість і ефективне використання ресурсів. Однією з ключових особливостей Kubernetes є декларативне управління станом системи. Це означає, що розробник задає бажаний стан (наприклад, кількість реплік додатку), а Kubernetes автоматично підтримує цей стан, реагуючи на збої або зміни в інфраструктурі. Такий підхід суттєво підвищує стійкість додатків до збоїв та спрощує їх підтримку. Крім того, система дозволяє здійснювати горизонтальне масштабуван-ня — як вручну, так і автоматично, на основі заданих метрик (наприклад, CPU або пам'яті), що забезпечує оптимальну продуктивність навіть при пікових навантаженнях. Для забезпечення постійного зберігання даних, Kubernetes інтегрується з різними рішеннями, такими як Persistent Volumes (PV) та Persistent Volume Claims (PVC), що дозволяють додаткам зберігати стан між перезапусками.

Також велика увага приділяється спостереженню за системою: вбудовані інструменти та сторонні рішення забезпечують ефективний моніторинг і логування, дозволяючи вчасно виявляти проблеми та оптимізувати роботу сервісів. Для спрощення встановлення та керування додатками використовується інструмент Helm, який дозволяє створювати, публікувати та повторно використовувати шаблони розгортання, зменшуючи кількість рутинної роботи.

Додаткові можливості на рівні мережі надають Istio, Linkerd та інші Service Mesh-рішення. Вони дозволяють реалізовувати складну маршрутизацію трафіку, контроль доступу, спостережуваність, а також захищену взаємодію між мікросервісами. Активна підтримка Kubernetes з боку відкритої спільноти, великої кількості навчальних ресурсів, а також його гнучкість та модульність зробили цю платформу стандартом де-факто у світі хмарних обчислень і мікросервісної архітектури. Вивчення Kubernetes стало важливою частиною професійної підготовки для сучасних розробників, DevOps-інженерів і системних адміністраторів, що працюють з масштабованими та динамічними програмними системами.

Docker та інші OCI-сумісні контейнери стали невіддільною частиною сучасної розробки програмного забезпечення завдяки своїй здатності забезпечувати зручне створення, ізоляцію, упаковку та розгортання додатків. Основна ідея полягає у тому, щоб «запакувати» програму разом з усіма її залежностями у єдиний контейнер. Це усуває проблему несумісностей між середовищами, даючи змогу запускати контейнер однаково як на локальній машині розробника, так і у хмарі чи на сервері. На відміну від класичних віртуальних машин, контейнери не потребують окремої операційної системи, оскільки використовують спільне ядро з хостовою ОС. Це робить їх значно легшими, швидшими та менш ресурсоємними, а процес створення та запуску — практично миттєвим. Стандартизацію в цій галузі забезпечує Open Container Initiative (OCI), яка визначає єдині специфікації формату контейнерів і механізмів їх запуску.

Завдяки цьому рішення, побудовані на Docker, Podman, CRI-O чи інших сумісних платформах, залишаються взаємозамінними та портативними. Окрім основної утиліти, Docker надає низку супутніх інструментів, які суттєво спрощують роботу з контейнерами. Наприклад, Docker Compose дозволяє визначати і запускати багатоконтейнерні додатки з єдиного конфігураційного файлу, а Docker Swarm забезпечує побудову кластерів для масштабованого та високодоступного розгортання.

Це особливо корисно в мікросервісних архітектурах, де кожен сервіс можна окремо оновлювати, масштабувати або замінювати. Контейнеризація також радикально змінила підходи до реалізації CI/CD-процесів. Тестування, збірка і розгортання тепер можуть відбуватись в однаковому середовищі, що знижує ймовірність помилок і скорочує час між написанням коду та його доставкою у production. Ізольовані середовища дозволяють безпечно перевіряти зміни, не зачіпаючи основний додаток.

Втім, активне використання контейнерів піднімає і нові виклики, передусім у сфері безпеки. Контейнери можуть містити вразливі залежності або мати некоректні конфігурації, що відкриває потенційні шляхи для атак. Тому критично важливо застосовувати практики безпечної розробки: обмеження прав доступу, сканування образів на вразливості, використання тільки перевірених репозиторіїв, а також регулярне оновлення контейнерів та базових образів. Таким чином, Docker і OCI-сумісні рішення стали основою сучасної розробки, пропонуючи високу гнучкість, швидкість, масштабованість та стабільність. Вони дозволяють командам ефективно адаптуватися до змін ринку, пришвидшують впровадження нових технологій та полегшують створення складних, розподілених систем.

Continuous Integration і Continuous Deployment (CI/CD) для мікросервісної архітектури — це важливий підхід, що дозволяє автоматизувати процеси розробки, тестування та розгортання програмного забезпечення. Його основна перевага полягає в тому, що він дозволяє скорочувати час виводу нових функцій на ринок, швидко виправляти помилки та при цьому зберігати високу якість і стабільність системи. Такий підхід дозволяє розгортати й оновлювати окремі частини системи без зупинки всієї інфраструктури, що ідеально поєднується з принципами CI/CD. Процес CI/CD включає кілька ключових етапів. Першим є інтеграція, під час якої код, написаний різними командами, об'єднується в єдину кодову базу. Це зазвичай виконується за допомогою систем контролю версій, таких як Git.

Системи автоматизації деплою та CI/CD процеси є невід'ємною частиною сучасної розробки програмного забезпечення, особливо в рамках мікросервісної архітектури. Вони дозволяють суттєво прискорити розгортання додатків, мінімізуючи вплив людського фактора та ймовірність помилок у процесі. Багато розробницьких команд впроваджують практики безперервної інтеграції та безперервної доставки (CI/CD), що допомагає зробити розробку більш ефективною та контрольованою.

Категорія	Технологія	Опис	Переваги
Контейнери	Docker	Упаковка додатків в контейнери	Швидке розгортання, ізоляція
Оркестрація	Kubernetes (K8s)	Автоматизація розгортання, масштабування	Самовідновлення, масштабованість
Сервісна сітка	Istio, Linkerd	Управління трафіком, безпека, спостереження	Розширений контроль за мікросервісами
API Gateway	Kong, Ambassador	Маршрутизація трафіку до мікросервісів	Аутентифікація, логування, кешування
CI/CD	Jenkins, Argo CD	Безперервна інтеграція та доставка	Автоматизація деплою

Рисунок. 1.1 - Порівняльна таблиця технологій для мікросервісної архітектури в Kubernetes

На ринку існує широкий вибір рішень для автоматизації деплою, серед яких популярним є Jenkins — система з відкритим кодом, що підтримує численні плагіни і дозволяє створювати гнучкі пайплайни, адаптовані до потреб проекту. Хоча Jenkins є безкоштовним, його налаштування може вимагати значних зусиль та знань від команди. Ще одним популярним вибором є GitLab CI/CD і GitHub Actions, які інтегруються із системами контролю версій Git і надають зручні інтерфейси, що спрощує роботу навіть для новачків. GitLab CI/CD також має вбудовані інструменти для моніторингу та тестування, що допомагає підтримувати стабільність коду.

Постійне об'єднання змін дозволяє швидше виявляти конфлікти та знижує ризик помилок при злитті коду. Наступним кроком є автоматизоване тестування, яке відіграє критичну роль у забезпеченні надійності. Тестування може бути різного рівня: юніт-тести перевіряють окремі функції, інтеграційні — взаємодію між компонентами, а end-to-end тести імітують поведінку користувачів. Завдяки цьому забезпечується виявлення помилок ще до розгортання коду в продуктивне середовище.

Коли тести проходять успішно, наступає етап автоматичного деплойменту. Для мікросервісів це особливо важливо, адже кожен сервіс може оновлюватися незалежно, що дає змогу оперативно вносити зміни без порушення роботи всієї системи. Важливу роль у CI/CD відіграє контейнеризація. Використання Docker дозволяє стандартизувати середовище виконання мікросервісів, усуваючи різницю між розробницьким та виробничим середовищами. Контейнери легко масштабуються та переносяться, що значно спрощує управління мікросервісною архітектурою.

1.3 Порівняльний аналіз сучасних інструментів

Аналіз платформ для керування контейнерами та їхній вплив на продуктивність є надзвичайно важливим у сучасній розробці програмного забезпечення. У світі, де хмарні сервіси та мікросервісна архітектура займають провідні позиції, правильний вибір інструментів для контейнеризації суттєво впливає на загальну ефективність системи, стабільність роботи та швидкість розробки. Для початку розглянемо найбільш поширені технології — Docker, Kubernetes, OpenShift — і їхні ключові особливості. Docker став базовою технологією для упаковки додатків разом із всіма необхідними залежностями у легкі, самодостатні контейнери, що полегшує їхнє розгортання і транспортування між середовищами. Водночас Kubernetes виступає як комплексна платформа для оркестрації контейнерів, що автоматизує розгортання, масштабування та управління контейнеризованими додатками в різних інфраструктурах.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було проведено глибокий огляд мікросервісної архітектури як сучасного підходу до побудови масштабованих та гнучких програмних систем. Було визначено ключові принципи мікросервісної моделі, включно з незалежністю сервісів, їхнім розмежуванням за бізнес-функціями, використанням стандартизованих інтерфейсів взаємодії та автономією команд розробників. Особливу увагу приділено технічним аспектам: контейнеризації, автоматизації процесів CI/CD, а також інструментам для управління хмарною інфраструктурою, зокрема Kubernetes та Docker.

Окрім технічних переваг, підкреслено організаційні та стратегічні вигоди мікросервісного підходу: пришвидшення розробки, підвищення надійності систем, спрощення масштабування і можливість швидко реагувати на зміни ринку. Водночас розкрито низку викликів, пов'язаних із впровадженням такої архітектури: складність інтеграції сервісів, забезпечення цілісності даних, підвищені вимоги до автоматизації та безпеки, необхідність кваліфікованих команд.

Розгляд сучасних технологій, таких як Kubernetes, Docker та CI/CD-платформи, дозволив оцінити інструментарій, який використовується для реалізації хмарних сервісів. Порівняльний аналіз інструментів управління контейнерами висвітлив переваги та обмеження кожного рішення, що дозволяє здійснювати обґрунтований вибір залежно від потреб проєкту. Таким чином, розділ сформував цілісне уявлення про архітектурні, технологічні та організаційні особливості сучасних мікросервісних систем та створив основу для практичної реалізації описаних концепцій у наступних розділах.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА ІНФРАСТРУКТУРНА ПІДГОТОВКА СИСТЕМИ

2.1 Аналіз вимог до мікросервісної системи

2.1.1 Функціональні та нефункціональні вимоги

Визначення вимог до майбутньої системи є надзвичайно важливим етапом проєктування, оскільки саме від цього залежить ефективність усієї розробки. На цьому етапі формується цілісне бачення того, які функціональні та нефункціональні характеристики має забезпечувати система. Для цього першочергово необхідно зібрати дані від усіх ключових учасників проєкту — замовників, кінцевих користувачів та членів команди розробки. Такий підхід дозволяє точно з'ясувати очікування щодо функціоналу системи та умов її експлуатації. До функціональних вимог зазвичай відносять здатність системи виконувати запити, працювати з базами даних, а також взаємодіяти з іншими інформаційними сервісами. У той час як нефункціональні вимоги охоплюють параметри продуктивності, масштабованості, безпеки, доступності та зручності використання.

Наприклад, якщо система повинна обслуговувати велику кількість одночасних користувачів, вона має відповідати високим вимогам до швидкості обробки запитів. Також надзвичайно важливо передбачити механізми забезпечення безпеки, оскільки захист персональних або комерційних даних сьогодні є пріоритетним завданням. На цьому етапі доцільно проводити аналіз потенційних ризиків, що дозволяє заздалегідь передбачити можливі загрози та сформувати стратегії їх уникнення чи нейтралізації. Окрім збору та аналізу вимог, доцільно використовувати візуальні інструменти для моделювання, зокрема діаграми варіантів використання (use case diagrams), які відображають взаємодію користувачів із системою. Це сприяє кращому розумінню архітектури рішення всіма учасниками розробки.

Крім того, гнучкий підхід до зміни вимог, запропонований у рамках методології Agile, дозволяє оперативно адаптувати систему до нових умов, що виникають у процесі реалізації проєкту. Надзвичайно важливим також є встановлення пріоритетності вимог: критичні функції повинні реалізовуватися насамперед, тоді як другорядні — на наступних етапах. Це дозволяє оптимізувати витрати ресурсів і зосередитися на головному. Загалом, аналіз вимог є основоположним етапом, який суттєво впливає на подальшу реалізацію системи. Його результати слугують підґрунтям для всіх наступних етапів розробки, тому виконувати цей процес слід з особливою відповідальністю.

2.1.2 Визначення пріоритетів і ризиків

У рамках аналізу важливу роль відіграє налагодження ефективної комунікації між усіма зацікавленими учасниками проєкту. Використання методології Scrum у гнучких процесах розробки дозволяє оперативно обговорювати зміни в очікуваннях і вчасно вносити правки у вимоги, що сприяє підвищенню якості фінального рішення. Одним з наступних важливих кроків є чітке формулювання вимог. На цьому етапі створюються документи, в яких зафіксовані бізнес-завдання та детальний опис функціоналу системи. Формулювання має бути точним, однозначним і зрозумілим для всіх учасників, щоб уникнути тлумачень та помилок у реалізації.

Не менш важливо класифікувати вимоги за ступенем важливості, поділяючи їх на критичні та додаткові. Такий підхід дає змогу зосередитися насамперед на реалізації основних функцій, відклавши другорядні завдання на наступні етапи.

Наприклад, нестача технічної документації, застарілі платформи або проблеми з актуальністю даних можуть суттєво ускладнити розробку. Тому на цьому етапі важливо виявити всі можливі внутрішні й зовнішні залежності, що можуть вплинути на хід реалізації.

Окрім технічних аспектів, необхідно також перевірити відповідність системи чинним нормативним актам і стандартам. Це стосується, зокрема, вимог щодо збереження даних, конфіденційності та надання доступу. Постійне оновлення правового поля зумовлює необхідність гнучкої адаптації рішень до нових умов. Документ із вимогами, складений на початку, часто потребує змін і має виконувати роль динамічного документа, що оновлюється протягом усього циклу розробки. Регулярний зворотний зв'язок від кінцевих користувачів відіграє особливу роль у процесі коригування вимог. Навіть незначні уточнення можуть значно покращити загальну якість програмного продукту.

2.2 Вибір архітектурного стилю та способу взаємодії

2.2.1. REST, gRPC та GraphQL: критерії вибору

Визначення архітектурного стилю для побудови мікросервісів є одним із ключових етапів у процесі створення програмного рішення. Від обраної моделі залежить, наскільки ефективною, масштабованою, продуктивною та адаптивною буде система в умовах змін бізнес-вимог і технологічного середовища. У цьому контексті важливо враховувати як технічні параметри, так і цілі, що стоять перед бізнесом. Сучасна практика пропонує розробникам кілька основних підходів до організації взаємодії між сервісами, серед яких найбільш поширеними є REST, GraphQL і gRPC — кожен із них має як сильні сторони, так і певні обмеження.

REST є найбільш відомим та стабільним підходом, який базується на використанні стандартних HTTP-методів для реалізації взаємодії між клієнтом і сервером. Цей стиль добре підтримується більшістю технологій та інструментів, включаючи Django та Python. Однією з головних переваг REST є простота розробки і зручність читання запитів, що робить його зручним вибором для більшості веб-застосунків. REST також дозволяє ефективно реалізовувати кешування, що зменшує навантаження на сервер і прискорює відповідь клієнту.

Однак, у проєктах з високими вимогами до швидкодії та мінімізації затримок, REST не завжди є оптимальним. У таких випадках варто звернути увагу на gRPC — сучасний стиль взаємодії, що працює поверх HTTP/2 і підтримує двосторонній стрімінг даних. Завдяки бінарному протоколу передачі та ефективному стисненню, gRPC дозволяє значно скоротити об’єм переданої інформації. Це робить його надзвичайно привабливим варіантом для інтенсивного обміну даними між мікросервісами. Проте реалізація gRPC вимагає глибших знань з боку розробників, і є менш інтуїтивно зрозумілою у порівнянні з REST. Ще одним сучасним підходом є GraphQL, який особливо добре підходить для сценаріїв з частими й динамічними запитами до різних типів даних. На відміну від REST, де фіксовані кінцеві точки повертають заздалегідь визначені набори даних, GraphQL дозволяє клієнту самостійно формулювати структуру запиту, отримуючи рівно ту інформацію, яка йому потрібна. Це мінімізує надлишковість у відповідях і зменшує кількість звернень до сервера. Але цей підхід вимагає специфічного підходу до проєктування API та належного рівня експертизи в команді. Підсумовуючи, обґрунтований вибір архітектурного стилю є передумовою для ефективної реалізації мікросервісної моделі. Саме на цьому етапі формується фундамент, який вплине на стабільність, швидкість та якість усієї системи в цілому.

2.2.2 Розподіл відповідальності між сервісами

Розподіл відповідальності між мікросервісами є визначальним фактором у забезпеченні ефективної роботи децентралізованої архітектури. Замість реалізації всіх бізнес-процесів в одному застосунку, функціональність розбивається на окремі сервіси, кожен з яких обслуговує власну логіку, працює автономно і виконує чітко окреслене завдання. Це дозволяє зменшити міжсервісні залежності, що спрощує впровадження змін, оновлення та налагодження системи без впливу на інші компоненти. Наприклад, сервіс, відповідальний за управління користувачами, не повинен містити механізмів авторизації платежів — кожен з них функціонує незалежно, комунікуючи через API або інші механізми обміну повідомленнями.

Ізоляція відповідальності також полегшує масштабування: можна збільшити ресурси лише для тих сервісів, які зазнають високого навантаження. Крім того, автономність дозволяє розробляти й розгортати сервіси різними командами, навіть на різних мовах програмування, без необхідності глибоко розуміти роботу суміжних частин. У межах великої системи це формує більш гнучку й адаптивну структуру, що легше піддається контролю, відстеженню та вдосконаленню.

Окрему увагу слід приділити розмежуванню зон відповідальності при проєктуванні — неправильне визначення меж сервісів може призвести до дублювання логіки, складності синхронізації даних або зайвої залежності між модулями.

Для уникнення цього застосовуються підходи на кшталт domain-driven design (DDD), які дозволяють визначити коректні межі між сервісами відповідно до реальних бізнес-потреб. У результаті формується архітектура, де кожен сервіс є незалежним функціональним блоком, що легко інтегрується, змінюється і повторно використовується без шкоди для решти системи.

2.3 Технологічний стек і засоби реалізації

2.3.1 Django, Docker, PostgreSQL: основні компонентии

Визначення відповідного стеку технологій для реалізації мікросервісної архітектури є одним із найважливіших кроків у розробці сучасного програмного забезпечення. Від цього вибору безпосередньо залежать не лише ефективність і стабільність роботи системи, а й її адаптивність до змін, можливість масштабування та тривала підтримка. Початковим етапом має бути глибокий аналіз наявних рішень, які відповідають як технічним вимогам, так і стратегічним цілям проєкту.

Особливу увагу при цьому слід звернути на мову програмування. Серед численних варіантів Python вирізняється простою, читабельною структурою коду, багатою екосистемою бібліотек та активною спільнотою. Саме ці характеристики роблять його ідеальним вибором для швидкої розробки мікросервісів. У поєднанні з Django або Flask можна реалізувати API з чіткою структурою та підтримкою масштабування. Django, зокрема, забезпечує високий рівень інтеграції інструментів, що спрощує створення повноцінних веб-сервісів, тоді як Flask підходить для розробки легших та більш ізольованих сервісів.

Вибір бази даних — ще один критичний аспект. У багатьох випадках ефективним рішенням буде використання реляційних баз даних, таких як PostgreSQL або MySQL, які мають потужну підтримку у Django через ORM. Однак при роботі з великою кількістю неструктурованих або напівструктурованих даних варто розглядати NoSQL-рішення, наприклад MongoDB. Для інтеграції таких БД у Python існує цілий спектр бібліотек, зокрема mongoengine або pymongo. Крім того, технології контейнеризації стали фундаментом сучасних підходів до мікросервісної архітектури. Використання Docker дозволяє упакувати кожен мікросервіс у незалежний контейнер із власними залежностями. Це значно полегшує розгортання, тестування і масштабування сервісів. У свою чергу, Kubernetes забезпечує централізоване управління кластерами, автоматичний баланс навантаження, моніторинг стану контейнерів і можливість безперервного оновлення.Не менш важливим є і підхід до комунікації між мікросервісами. Django REST Framework дозволяє реалізовувати RESTful API з мінімальними зусиллями, забезпечуючи валідацію, серіалізацію та обробку запитів на високому рівні. Водночас, при складних запитах, які потребують вибіркової передачі даних, варто звернути увагу на GraphQL, для якого існує підтримка через бібліотеки типу Graphene.

Окрему увагу слід приділяти безпеці, особливо в умовах розподіленої архітектури. Django надає широкі можливості для захисту від типових загроз — CSRF, XSS, SQL-ін'єкцій. Також доцільним буде використання авторизації через JWT або OAuth2, що полегшує управління доступом між сервісами. Підсумовуючи, можна сказати, що побудова мікросервісної архітектури з використанням Python і Django — це ефективний шлях до створення гнучких, масштабованих і підтримуваних систем. Правильний вибір інструментів на кожному етапі розробки забезпечить не лише технічну надійність, а й бізнес-стійкість рішення в цілому.

2.3.2 Інструменти для безпеки та CI/CD

Інтеграція безпеки та автоматизації розгортання є критичним елементом сучасної мікросервісної архітектури. Забезпечення конфіденційності даних, захист каналів взаємодії між сервісами, контроль доступу до ресурсів та відстеження змін — усе це вимагає використання спеціалізованих інструментів, здатних органічно вписатися у CI/CD-процеси. Такий підхід дозволяє реалізувати модель «безпека як код», де контроль безпечної конфігурації стає частиною життєвого циклу розробки. Одним із ключових аспектів є захищене зберігання конфіденційної інформації, зокрема паролів, токенів доступу чи ключів до баз даних. У середовищі Kubernetes для цього використовуються ресурси типу Secrets, які можна додавати до контейнерів як змінні середовища або через томи. Вони дозволяють розділяти доступ за принципом мінімальних прав і уникати зберігання чутливих даних безпосередньо у коді або відкритих конфігураційних файлах. Для додаткового шифрування можуть застосовуватись сторонні системи, як-от HashiCorp Vault або Sealed Secrets.

Ще одним важливим завданням є контроль доступу на рівні сервісів. У Kubernetes для цього застосовується механізм RBAC (Role-Based Access Control), який дозволяє задавати детальні права для користувачів, сервісних облікових записів або додатків. Це унеможливлює несанкціоновані дії в кластері, обмежуючи взаємодію лише необхідними правами. Весь процес описується в конфігураційному файлі .gitlab-ci.yml, що надає повний контроль над етапами розробки і мінімізує людський фактор.

У процесі налаштування пайплайну особливу увагу приділяють перевірці наявності вразливостей у залежностях. Це можна реалізувати через інтеграцію з такими інструментами, як Snyk, Trivy або SonarQube, які автоматично перевіряють безпеку контейнерів і коду. Таким чином, ще до етапу деплою можна запобігти проникненню потенційно шкідливих компонентів у систему.

Завдяки поєднанню механізмів безпеки та засобів безперервної інтеграції забезпечується не лише стабільність релізів, а й відповідність сучасним стандартам розробки та експлуатації хмарних мікросервісів. Це дозволяє оперативно доставляти оновлення, контролювати ризики й дотримуватися вимог до надійності та цілісності інформації в розподіленому середовищі.

2.4 Оркестрація та конфігурація Kubernetes

2.4.1 Побудова середовища та керування конфігураціями

Конфігурація Kubernetes-середовища є ключовим етапом при впровадженні мікросервісної архітектури, особливо якщо вона реалізується із використанням Python та Django. Саме на цьому етапі формується інфраструктура, що забезпечує стабільність, масштабованість і гнучке управління сервісами в продакшн-середовищі.

Першочергово слід визначити кластерну архітектуру Kubernetes — кількість та ролі вузлів, зокрема master-нод і worker-нод. Це дозволяє створити надійне середовище, в якому кожен мікросервіс (наприклад, Django REST API, сервіс обробки платежів на Flask або асинхронний сервіс на FastAPI) функціонує у власному ізольованому контейнері. Для кожного такого сервісу потрібно задати ресурси CPU та RAM у YAML-файлах розгортання (Deployment), з урахуванням очікуваних навантажень.

Особливу увагу слід приділити налаштуванню мережевої взаємодії між сервісами. Kubernetes забезпечує гнучке управління мережею через Service, Ingress та можливість використання Service Mesh-інструментів, таких як Istio. Це дає змогу Django-сервісам взаємодіяти з іншими мікросервісами через стабільні DNS-імена або маршрути HTTP/HTTPS, захищені TLS. Збереження чутливої інформації, такої як ключі доступу до баз даних, токени API або налаштування Django (SECRET_KEY, облікові дані SMTP тощо), здійснюється через Secrets або ConfigMaps, які можна легко підключити до контейнерів як змінні середовища або файли конфігурації.

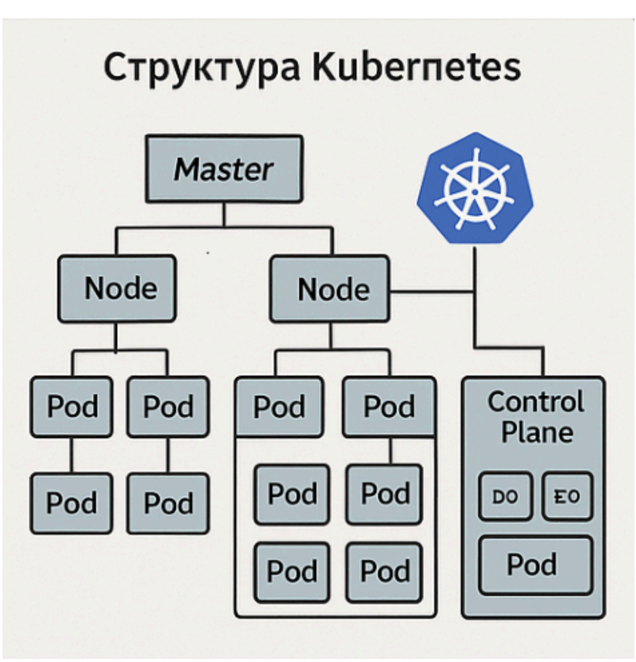


Рисунок. 1.2 - Загальна архітектура Kubernetes-кластера з мікросервісами

Для автоматизації розгортання Python-сервісів доцільно використовувати Helm Charts. Це дозволяє шаблонізувати всі ресурси Kubernetes та керувати оновленнями, масштабуванням і конфігурацією з єдиного місця. У зв’язці з CI/CD (наприклад, GitLab CI) можна налаштувати повний цикл безперервної інтеграції та деплойменту: запуск тестів для Django, створення Docker-образу, пуш у реєстр і автоматичне оновлення у кластері.

Таким чином, правильно спроєктоване Kubernetes-середовище для мікросервісів на Python/Django забезпечує не лише високу доступність і надійність, а й дозволяє оперативно масштабувати систему, впроваджувати нові сервіси та підтримувати високий рівень безпеки. Це створює надійну основу для довготривалого розвитку як технічної, так і бізнес-частини продукту.

2.4.2 Налаштування кластеру, Secrets

Налаштування Kubernetes-кластеру є ключовим етапом у впровадженні мікросервісної архітектури, оскільки саме ця платформа відповідає за управління життєвим циклом контейнеризованих застосунків. На початковому етапі необхідно сформувати базову інфраструктуру, яка забезпечить ізольоване середовище для кожного сервісу, гнучке масштабування, автоматичне перезапускання у разі збоїв та підтримку конфігурацій у вигляді окремих об’єктів.

Одним із основних механізмів конфігурації в Kubernetes є ресурси типу ConfigMap та Secret. Якщо ConfigMap призначені для зберігання відкритих налаштувань, таких як порти, URL або режими запуску, то Secret використовується для безпечного зберігання чутливих даних, які не мають потрапити у вихідний код чи журнали.

Дані у Secret зберігаються у кодованому вигляді (base64), і їх можна передавати у контейнери через змінні середовища або файлову систему.

Для створення Secret можливо використовувати як YAML-маніфести, так і команду kubectl create secret.

Наприклад, базова команда для додавання секрету з логіном та паролем виглядає як kubectl create secret generic db-secret --from-literal=username=admin —from-literal=password=securepass. Після цього об'єкт можна використати в описі Deployment-ресурсу для передачі конфіденційної інформації в поди. Це дозволяє чітко контролювати, які сервіси мають доступ до яких саме даних, і виключає ймовірність помилкового поширення секретів у межах кластера.

Крім того, важливим аспектом є управління доступом до Secret за допомогою RBAC (Role-Based Access Control), що дозволяє надавати права лише необхідним компонентам або користувачам. Це підвищує загальний рівень безпеки кластера, мінімізуючи ризик несанкціонованого доступу до критичних налаштувань. Таким чином, використання Secrets у поєднанні з грамотним налаштуванням кластеру формує безпечне та стабільне середовище для розгортання мікросервісної архітектури у масштабованому хмарному середовищі.

Налаштування самого кластера також включає визначення кількості нод, налаштування мережі, зберігання логів, використання ingress-контролерів для зовнішнього доступу та моніторинг ресурсів. Більшість сучасних інфраструктур реалізується через інструменти інфраструктури як коду (наприклад, Helm або Terraform), що дозволяє зробити процес розгортання кластерів відтворюваним та керованим. Застосування Helm для керування шаблонами конфігурацій забезпечує повторюваність налаштувань, а також спрощує оновлення версій сервісів та компонентів.

Таким чином, другий розділ заклав міцну теоретичну та технічну основу для реалізації проєкту. На підставі аналізу вимог, архітектурних підходів та обґрунтованого вибору інструментів, розроблено структуру майбутньої системи, що дозволить забезпечити її масштабованість, ефективність та надійність у продакшн-середовищі.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі було проведено комплексний аналіз вимог до проєктованої мікросервісної системи, а також здійснено архітектурне та технологічне обґрунтування її побудови. Розділ охопив як теоретичні, так і практичні аспекти розробки сучасного веб-застосунку, що функціонує в хмарному середовищі з використанням Kubernetes.

Перш за все було сформульовано функціональні та нефункціональні вимоги до системи. Було визначено, що система повинна забезпечувати горизонтальну масштабованість, стійкість до збоїв, незалежність сервісів, спрощене розгортання, а також безперервну інтеграцію та доставку. Окрему увагу приділено вимогам до безпеки, моніторингу, логування та ефективної взаємодії сервісів. Далі проведено аналіз зазначених вимог, що дозволив обґрунтовано перейти до архітектурного дизайну. Було визначено, що для даного типу системи найдоцільнішим є використання мікросервісної архітектури, яка забезпечує гнучкість, модульність та масштабованість. Розглянуто підходи до побудови сервісів, визначено їхні ролі, межі відповідальності та способи взаємодії між ними.

Окремо описано процес конфігурації середовища Kubernetes — від розробки manifest-файлів до налаштування подів, сервісів, ingress-контролерів, секретів, config map'ів та autoscaler'ів. Було розглянуто також варіанти деплою кластера. Також визначено загальний технологічний стек, до якого входять інструменти CI/CD, системи управління версіями (Git), GitLab CI, Helm (для шаблонізації), а також Kubernetes-орієнтовані рішення для моніторингу та безпеки. Останнім етапом розглянуто побудову повноцінного процесу CI/CD, включаючи автоматичне тестування, збірку, створення контейнерів, їх завантаження у реєстр, деплой у кластер, перевірку здоров'я сервісів, та автоматичне масштабування в залежності від навантаження.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ІНТЕГРАЦІЯ СИСТЕМИ

3.1 Побудова серверної логіки Django-застосунку

3.1.1 Архітектура проєкту: шаблон Model-View-Template

Для реалізації серверної частини веб-застосунку, призначеного для видалення фону зображень, було обрано фреймворк Django — один із найпоширеніших інструментів для створення веб-додатків мовою Python. Django забезпечує швидку розробку, чітку логічну організацію коду, вбудовані механізми роботи з формами, а також засоби автентифікації користувачів і захисту від атак типу CSRF. Архітектура додатку базується на шаблоні «Model-View-Template», що забезпечує поділ функцій між рівнями системи. Головною складовою є модуль обробки зображень: користувачі можуть завантажити графічний файл, здійснити над ним операцію видалення фону й отримати оброблене зображення.

Основна функціональність серверної частини включає механізми реєстрації та входу в систему, CSRF-захист, прийом файлів через веб-форму, їхню обробку на сервері, збереження результатів і відображення у веб-інтерфейсі. Система автентифікації використовує стандартний функціонал Django: після реєстрації або входу користувач отримує сесію, що зберігається через cookie, а всі POST-запити перевіряються на наявність CSRF-токена, який автоматично додається до шаблонів. Після входу в обліковий запис користувач перенаправляється на сторінку з формою завантаження зображення, яка побудована за допомогою поля типу file. Серверна обробка полягає в отриманні зображення, його збереженні у тимчасовому сховищі та передачі до функції обробки. Для модифікації зображень використовуються бібліотеки Python, зокрема Pillow або власноруч реалізований алгоритм.

Результат зберігається окремо, а у веб-інтерфейсі виводиться посилання для перегляду. У системі використовується проста модель даних — ProcessedImage, яка містить шляхи до оригінального та обробленого файлів, а також посилання на власника зображення. На етапі розробки база даних реалізована за допомогою SQLite, однак у майбутньому передбачено можливість переходу на PostgreSQL. Структура застосунку охоплює файли views.py (логіка обробки форм і зображень), models.py (опис моделі зображень), forms.py (реалізація веб-форми завантаження), templates (шаблони HTML-сторінок), а також urls.py (визначення маршрутів).

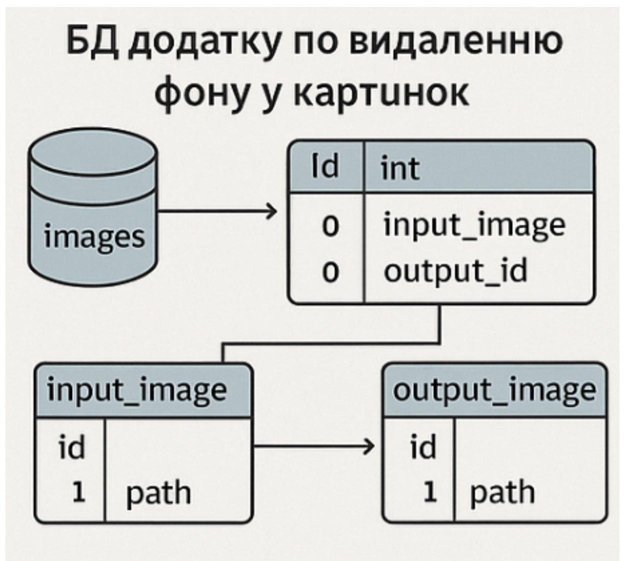


Рисунок. 1.3 - Приклад структури бази даних, створеної на основі описаної django-моделі

Усе це об'єднується в один Django-проєкт без використання REST API дозволяє сфокусуватися на класичній серверній логіці без розділення на мікросервіси. Завдяки використанню поля ImageField у формі, користувачі можуть вибирати зображення для обробки, а перевірка вхідних даних гарантує коректність файлу. Для спрощення підтримки проекту шаблони успадковуються від базового шаблону (base.html), який включає загальні елементи дизайну.

3.1.2 Обробка зображень і взаємодія з користувачем

Фронтенд-складова створеної мікросервісної системи реалізована із застосуванням базових вебтехнологій, таких як HTML5 для структурування вмісту та CSS3 для візуального оформлення. Такий вибір є обґрунтованим у випадках, коли функціональні вимоги до користувацького інтерфейсу не передбачають високого рівня динамічності або складної інтерактивності, що зазвичай реалізується засобами JavaScript або за допомогою сучасних frontend-фреймворків. Простота у реалізації, кросбраузерна сумісність і висока швидкодія інтерфейсу стали ключовими чинниками, які визначили вибір саме цієї технологічної бази для клієнтської частини застосунку.

Головним завданням фронтенду було створити зрозуміле та доступне середовище для взаємодії користувача із сервісом — від реєстрації та входу, до завантаження файлів, перегляду оброблених зображень і управління власними ресурсами. Логіка навігації повністю відповідає послідовності дій у межах бізнес-процесів: кожна сторінка виконує конкретну функцію, що полегшує орієнтування у межах веб-інтерфейсу. Для досягнення узгодженості дизайну в межах усього проєкту використано шаблонну систему Django, яка дозволяє реалізувати повторне використання спільних інтерфейсних компонентів. Завдяки цьому заголовки, панелі навігації, підвали та повідомлення не дублюються вручну у кожному шаблоні, а автоматично наслідуються від базового шаблону, що полегшує підтримку і забезпечує єдність стилю усього вебзастосунку.

У структурі HTML-шаблонів чітко дотримано принципів семантичної верстки — кожен блок інтерфейсу має окреме призначення, що спрощує його сприйняття як користувачем, так і системами доступності. Основні сторінки поділяються на кілька логічних частин: верхній блок (header) містить назву сервісу та елементи автентифікації, центральна частина (main) охоплює основний функціонал, зокрема форми завантаження, таблиці з файлами та статусні повідомлення, а нижня панель (footer) завершує сторінку та містить допоміжну інформацію. Усі елементи інтерфейсу розроблені з урахуванням сучасних принципів UX-дизайну, з акцентом на простоту, функціональність і зручність у користуванні. Оформлення реалізовано через окремий CSS-файл, у якому визначено ключові стилістичні характеристики: кольорову палітру, шрифти, розміри відступів, вигляд кнопок, стилі форм тощо.

Основний акцент зроблено на візуальну контрастність, що забезпечує легку читабельність тексту та чітке виділення інтерактивних елементів. Застосування різних фонів для окремих блоків дозволяє візуально структурувати сторінку, а стилізовані кнопки надають інтерфейсу сучасного вигляду навіть без застосування JavaScript. Хоча система розроблялася з орієнтацією на використання з комп’ютера, базові адаптивні властивості все ж були реалізовані. Завдяки гнучкій сітці, процентному масштабуванню та універсальному контейнеру, інтерфейс коректно відображається на екранах різного розміру — від десктопів до планшетів. Це дозволяє користувачам взаємодіяти із системою без суттєвих втрат у зручності. Безпека форм забезпечується за рахунок вбудованого механізму захисту від CSRF-атак, де у всі форми автоматично додається відповідний токен за допомогою Django-шаблонів. Це запобігає можливості зовнішнього несанкціонованого надсилання POST-запитів і є важливим елементом захисту, особливо у контексті обробки персональних даних.

Повідомлення про дії користувача — такі як помилки при авторизації, успішне завантаження файлу або завершення обробки — реалізовані через систему повідомлень Django. Вони відображаються безпосередньо на сторінках і мають колірне виділення, що дозволяє ефективно інформувати користувача про стан виконання дій та підтримувати зворотний зв'язок. У результаті реалізовано лаконічний, зручний та надійний користувацький інтерфейс, який не перевантажений складною логікою і не залежить від зовнішніх бібліотек. Простота верстки, інтеграція з шаблонною системою Django та відповідність вимогам користувацького досвіду забезпечили ефективну реалізацію фронтенду для цього проєкту. Такий підхід дозволяє швидко вносити зміни, підтримувати єдину структуру всього застосунку та полегшує масштабування функціоналу в майбутньому, зберігаючи водночас зрозумілий та доступний інтерфейс для кінцевого користувача.

3.2 Контейнеризація й Docker-інтеграція

3.2.1 Створення Dockerfile: образи та залежності

У мікросервісній архітектурі ключовим елементом є визначення ефективних способів взаємодії між незалежними компонентами системи. Одним із найпоширеніших рішень у цьому контексті є використання протоколу HTTP у поєднанні з REST-подібними API, що забезпечують просту, гнучку та стандартизовану модель обміну даними між сервісами. У межах розробленої системи, попри її невеликий масштаб, вже на етапі первинної реалізації було закладено принципи, що сприяють майбутньому масштабуванню.

Поточна структура ґрунтується на базовій клієнт-серверній взаємодії в межах одного Django-застосунку, але передбачає потенційне розділення функціональних блоків на окремі сервіси, які зможуть обмінюватися інформацією за допомогою HTTP-запитів.

Закладена логіка розділення дає змогу в майбутньому винести обробку зображень, авторизаційний модуль, управління даними та інші вузькоспеціалізовані задачі в окремі сервіси. Після переходу до повноцінної мікросервісної архітектури кожен функціональний блок (наприклад, сервіс ідентифікації, сервіс обробки зображень чи керування користувацькими файлами) працюватиме автономно, обмінюючись даними виключно через API.

Цей підхід має високу гнучкість, не залежить від реалізації сервісів на одній мові програмування та добре масштабується завдяки можливості використання проксі-серверів і балансувальників навантаження у кластерному середовищі на базі Kubernetes. Особливе значення в такій архітектурі має реалізація надійної автентифікації та авторизації. Якщо в поточній реалізації застосовується традиційна модель на базі CSRF-токенів та сесій, то у разі переходу до міжсервісної взаємодії доцільним буде впровадження токен-орієнтованих механізмів — таких як JWT або OAuth 2.0. Це дозволить безпечно передавати права доступу, мінімізувати залежність від сесій та централізовано управляти авторизацією у розподіленому середовищі. Визначення базового образу для контейнера є одним із ключових етапів у створенні Dockerfile. Базовий образ виступає фундаментом, на якому формується усе середовище для виконання або розробки програмного забезпечення. Вибір відповідного образу повинен враховувати низку факторів: мову програмування, специфіку застосунку, системні вимоги та бажаний рівень безпеки.

У сховищі Docker Hub доступні тисячі готових образів, серед яких широке використання мають alpine, ubuntu, debian, а також спеціалізовані образи для конкретних мов програмування, як-от python, node, golang. Залежно від цілей проєкту, доцільно обирати між мінімалістичними або повноцінними образами. Наприклад, alpine приваблює розробників через надзвичайно малий розмір, що скорочує час завантаження контейнера та зменшує обсяг використаних ресурсів. Водночас образ ubuntu або debian забезпечує більш повне середовище, що може бути необхідним для застосунків, які залежать від наявності широкого набору системних утиліт або пакетів.

Окрім вибору типу образу, важливо звертати увагу на його версію: найкращою практикою є вказування конкретної стабільної версії, наприклад, python:3.9-slim, замість використання позначки latest, яка може спричинити непередбачувані зміни в середовищі. Крім функціональності та сумісності, вагоме значення має аспект безпеки. При виборі образу слід перевіряти його на наявність відомих вразливостей. У цьому можуть допомогти спеціалізовані інструменти, такі як Trivy, Grype або Docker Bench for Security, які дозволяють проводити аналіз контейнерів і виявляти потенційні ризики. Водночас, іноді необхідність у використанні специфічних бібліотек або залежностей може виправдовувати обрання більш "важкого", але функціонально насиченого образу.

Таким чином, вибір базового образу — це завжди компроміс між компактністю, безпекою та функціональністю. Успішне визначення цього компонента дозволяє не лише оптимізувати процес конфігурації контейнера, але й забезпечити надійність та стабільність застосунку в умовах реального середовища. Після вибору базового образу наступним кроком є встановлення потрібних залежностей, що формують остаточне середовище для роботи програмного продукту.

3.2.2 Команди запуску, .dockerignore і безпека

Після вибору базового образу одним із ключових етапів створення Dockerfile є додавання необхідних бібліотек і залежностей, що безпосередньо впливає на функціональність контейнера. На цьому етапі важливо чітко розуміти, які саме пакети потрібні для коректної роботи додатку. Dockerfile є текстовим документом, який містить покрокові інструкції для побудови середовища виконання, тож правильна організація його вмісту є критично важливою.

Якщо, наприклад, ви використовуєте Python, доцільно встановлювати такі бібліотеки як Flask, NumPy або Pandas за допомогою інструкції RUN, яка виконує команду всередині контейнера під час створення образу. Проте ефективніше використовувати файл requirements.txt, в якому вказано повний список залежностей з відповідними версіями. Такий підхід спрощує підтримку проєкту, забезпечує повторюваність і зменшує ризик несумісності. Для цього спершу копіюють сам файл за допомогою COPY requirements.txt ., а потім виконують інсталяцію через RUN pip install --no-cache-dir -r requirements.txt, де параметр --no-cache-dir допомагає уникнути збереження тимчасових файлів та зменшити розмір контейнера.

Іноді, окрім Python-бібліотек, можуть знадобитися й системні пакети, наприклад компілятори або заголовкові файли. Їх можна встановити через менеджер пакетів операційної системи, вбудований у базовий образ, як-от apt-get у випадку з Debian або Ubuntu. Наступним кроком є копіювання вихідного коду проєкту в контейнер. Це робиться за допомогою команди COPY, яка приймає два аргументи: шлях до файлів на локальній машині та директорію у контейнері.

Наприклад, COPY . /app переміщує всі файли з поточної директорії до /app у контейнері. Однак варто дбати про безпеку та чистоту середовища — небажано копіювати зайві або конфіденційні файли. Щоб уникнути цього, створюють файл .dockerignore, в якому зазначають шаблони для виключення файлів, як-от тимчасові каталоги, кеші, логи, ключі доступу тощо. Цей файл працює аналогічно до .gitignore у Git і дозволяє суттєво оптимізувати процес збирання образу, зменшити його розмір і виключити випадкове потрапляння чутливої інформації до контейнера.

В залежності від структури проєкту, замість повного копіювання можна вказати лише окремі каталоги або файли, що особливо актуально для великих систем. Наприклад, можна використати COPY src/ /app/src/ для переносу лише вихідного коду. При цьому важливо дотримуватися правильного контексту — тобто переконатися, що всі необхідні файли знаходяться в межах доступного для Docker каталогу, оскільки система збирання не зможе працювати з елементами, що лежать поза ним. Завершальним етапом є налаштування команди запуску застосунку, яка визначає поведінку контейнера після його старту.

У Dockerfile для цього використовуються інструкції CMD або ENTRYPOINT. Обидві вказують, яка саме команда повинна бути виконана під час запуску контейнера, проте мають відмінності. ENTRYPOINT задає базову команду, яка виконується завжди, тоді як CMD використовується для передачі додаткових параметрів або вказівки команди за замовчуванням, яка може бути змінена при запуску. Наприклад, для запуску Node.js-додатку можна вказати ENTRYPOINT ["node", "app.js"], а параметри передати через CMD ["--port", "8080"]. У деяких випадках може бути доцільно використовувати скрипт запуску, особливо якщо необхідно ініціалізувати декілька процесів або виконати попередні налаштування перед стартом програми.

Такі скрипти пишуться на shell або bash і додаються до образу з подальшим виконанням як основної команди. В результаті, чітко прописані інструкції запуску забезпечують стабільне і передбачуване функціонування додатку в контейнерному середовищі. Таким чином, усі розглянуті етапи — додавання залежностей, копіювання коду та налаштування запуску — є критичними для створення надійного Docker-образу. Правильне їх виконання забезпечить не лише працездатність програми, але й її стабільну роботу під час реального використання.

3.3 CI/CD: автоматизація життєвого циклу

3.3.1 Kubernetes: інтеграція, токени, secrets

Створення файлу Deployment є критично важливим кроком у процесі розгортання застосунку в середовищі Kubernetes, адже саме цей документ описує, як саме буде функціонувати основний додаток у кластері. У ньому вказуються ключові параметри — кількість реплік, образ контейнера, налаштування ресурсів і конфігурацій. Як правило, Deployment створюється у форматі YAML, що робить його зручним для читання й редагування. На початку необхідно визначити метадані, такі як назва додатку та мітки (labels), які надалі використовуються для ідентифікації компонентів у межах кластера.

Наприклад, можна задати ім'я на кшталт my-app-deployment, а у мітках зазначити app: my-app, що дозволить Kubernetes правильно ідентифікувати та групувати пов'язані ресурси. Далі потрібно визначити, скільки екземплярів застосунку має бути запущено одночасно — це здійснюється через параметр replicas. Наприклад, для забезпечення високої доступності додатку варто запускати щонайменше три репліки: replicas: 3. Наступним етапом є налаштування шаблону поду, в якому вказуються контейнери, що запускатимуться.

У секції containers задається ім’я контейнера, образ (наприклад, image: myapp-image:latest), а також порти, які будуть відкриті для зовнішнього чи внутрішнього доступу — зокрема, containerPort: 8080, якщо додаток працює на відповідному порту. Для стабільної роботи застосунку необхідно чітко задати межі ресурсів, які він споживатиме. Це здійснюється через параметри resources, де задаються requests (мінімальні ресурси) та limits (максимальні). Такий підхід дозволяє кластеру розподіляти ресурси ефективно й уникати конфліктів між подами. Окрім цього, важливо врахувати зовнішні налаштування, які можуть змінюватися без потреби модифікації образу контейнера. Для цього у Kubernetes існують об'єкти ConfigMap і Secret, які дозволяють підключити конфігураційні дані або чутливу інформацію (наприклад, API-ключі) через змінні середовища або монтування у файлову систему контейнера.

Завершальним елементом у файлі Deployment є стратегія оновлення, яка описує, як саме система має оновлювати поди при внесенні змін — чи це буде повне перезавантаження (Recreate), чи поступова заміна (RollingUpdate), що дозволяє уникнути простоїв. Таким чином, файл Deployment — це основа стабільної роботи вашого сервісу в Kubernetes, і від його правильного налаштування залежить не лише коректність розгортання, але й ефективність усієї інфраструктури. Окрім цього, налаштування Service у Kubernetes забезпечує можливість зв’язку між подами та зовнішніми або внутрішніми клієнтами.

Service виступає як логічна абстракція, що дозволяє отримати стабільну точку доступу до змінних за своєю природою подів, оскільки останні при кожному перезапуску можуть мати нову IP-адресу. Існує кілька типів сервісів, і вибір залежить від потреб застосунку: ClusterIP — для доступу лише всередині кластера, NodePort — для зовнішнього доступу через фіксований порт вузла, LoadBalancer — для автоматичної інтеграції з хмарними балансувальниками навантаження, ExternalName — для перенаправлення запитів на зовнішні сервіси.

Автоматизована генерація YAML-маніфестів дозволяє систематизувати та стандартизувати процес налаштування всіх компонентів кластера — починаючи від подів, сервісів, конфігурацій, закінчуючи секретами та маршрутами. YAML, завдяки простій ієрархічній структурі, забезпечує прозору інтеграцію з CI/CD-пайплайнами, інфраструктурою як кодом і системами управління змінами. Правильно сформовані маніфести містять основні поля: apiVersion, kind, metadata, spec, які є основою будь-якого об'єкта Kubernetes. Саме ці файли забезпечують передбачуваність поведінки системи, прозорість змін і контроль за середовищем виконання застосунків.

Створення ConfigMap можливе або через команду kubectl create config-map, де значення вказуються безпосередньо в командному рядку, або шляхом опису у YAML-файлі, що зручно для великих обсягів конфігурацій та подальшої версійності. Наприклад, при роботі з підключенням до бази даних у ConfigMap можна зберігати адресу сервера, ім'я користувача та порт, передаючи їх додатку без шифрування (на відміну від Secret). Зміни в ConfigMap можуть бути застосовані до запущених контейнерів, хоча зазвичай для цього потрібно перезапустити поди, якщо конфігурація зчитується при старті. Це може бути реалізовано через стратегії оновлення, які Kubernetes підтримує за замовчуванням.

Перевагою такого підходу є чітке розділення налаштувань від бізнес-логіки, а також зменшення часу на оновлення й усунення помилок. Конфігурації, винесені у ConfigMap, можуть версіонуватись і зберігатись у системах контролю версій, що полегшує відновлення після збоїв та відстеження змін. Тому використання ConfigMap у Kubernetes є надзвичайно ефективним інструментом для централізованого управління налаштуваннями застосунків.

3.3.2 GitLab CI: workflow, деплой

Побудова процесу CI/CD (Continuous Integration / Continuous Deployment) є критичним етапом розробки будь-якої сучасної програмної системи, зокрема у випадку мікросервісної архітектури, що передбачає автономну розробку, тестування та розгортання окремих компонентів. У контексті створеного застосунку, який реалізує базовий функціонал авторизації користувачів і обробки зображень у веб-інтерфейсі на основі Django, впровадження CI/CD дозволяє досягти стабільності, повторюваності та надійності всього процесу доставки оновлень до Kubernetes-кластера. Безперервна інтеграція (CI) — це процес регулярного об’єднання змін до спільного репозиторію, з подальшим автоматичним виконанням перевірок: тестів, статичного аналізу, перевірки форматування тощо. Головна мета — якомога раніше виявити потенційні помилки та забезпечити стабільність основної гілки розробки. У проєкті це дозволяє гарантувати, що після кожного внесення змін (наприклад, додавання нового алгоритму обробки зображень або оновлення шаблонів HTML/CSS) система залишається працездатною та не потребує ручного тестування на кожному етапі.

Безперервне розгортання (CD) передбачає автоматичну доставку змін на тестові або продуктивні середовища після проходження усіх необхідних перевірок. Це усуває «людський фактор» при розгортанні, забезпечує прогнозованість процесу, пришвидшує вихід нових версій застосунку, дозволяє швидко реагувати на знайдені помилки чи потреби користувачів. Мікросервісна архітектура значно ускладнює ручне керування процесами оновлення, оскільки кожен сервіс потенційно має власні залежності, версію та логіку деплою. Тому саме автоматизований підхід до інтеграції та доставки є оптимальним рішенням, яке дозволяє незалежно розгортати компоненти, знижуючи ризики міжсервісних конфліктів.

Навіть у рамках відносно монолітного Django-застосунку, який у перспективі може бути розділений на окремі сервіси (наприклад, обробка зображень, модуль авторизації, база даних тощо), вже зараз впровадження CI/CD формує фундамент для масштабування та підтримки. У розробленому проєкті як основну платформу CI/CD було обрано Gitlab CI. Це зручний і гнучкий інструмент, що дозволяє створювати так звані «workflow» — сценарії, які запускаються автоматично після певної події, наприклад: коміту в репозиторій, відкриття pull request, або тегування релізу.

Кожен workflow виконується в ізольованому середовищі, що дозволяє повторно відтворювати результат незалежно від локального середовища розробника. У типовому сценарії проєкту передбачено такі етапи: клонування репозиторію та ініціалізація залежностей Python-проєкту, збірка Docker-образу на основі Dockerfile, перевірка наявності синтаксичних помилок або недопустимих змін, пуш Docker-образу до реєстру (наприклад, DockerHub або GitHub Container Registry), автоматичне оновлення застосунку в Kubernetes-кластері шляхом оновлення відповідного Deployment. Ще однією важливою перевагою CI/CD є забезпечення контролю якості. Кожна зміна, що вноситься в репозиторій, проходить через однакові перевірки. Це створює довіру до процесу, зменшує ризик помилок у продуктивному середовищі та дозволяє скоротити цикл розробки. Крім того, автоматичне логування усіх етапів CI/CD сприяє кращому виявленню та аналізу помилок: у випадку невдалого розгортання розробник отримує чітку інформацію про причину збою без потреби відтворювати ситуацію вручну.

Отже, впровадження CI/CD у розробленій системі дозволило зробити процес розробки більш передбачуваним, контрольованим і стійким до помилок. Це стало важливою частиною інфраструктури, яка забезпечує автоматичне тестування, збірку та доставку застосунку у кластер Kubernetes, з можливістю подальшого масштабування під окремі мікросервіси. Навіть для невеликого Django-проєкту така автоматизація є важливим кроком у напрямку професійної експлуатації та підтримки веб-застосунків.

3.4 Тестування: надійність та масштабованість

3.4.1 Функціональне та інтерфейсне тестування

Тестування у мікросервісній системі відіграє ключову роль у забезпеченні надійності, стабільності та передбачуваної поведінки кожного окремого сервісу. Навіть якщо проєкт має відносно просту архітектуру — наприклад, у вигляді одного Django-застосунку, що функціонує як окремий сервіс — перевірка функціональності залишається критично важливою. У межах реалізованої системи основною метою тестування є перевірка коректної роботи основного функціоналу: завантаження зображень, обробки (видалення фону), авторизації користувача, коректної реакції інтерфейсу та відсутності помилок у бізнес-логіці.

Оскільки API як окремий інтерфейс у проєкті не використовувався, тестування переважно зосереджено на рівні веб-інтерфейсу (HTML-сторінки) та взаємодії з серверною частиною через шаблонізований frontend Django. Тестування функціональності у проєкті здійснювалося переважно за допомогою вбудованих засобів Django (django.test.TestCase), що дозволяють перевірити поведінку застосунку у режимі емуляції HTTP-запитів. У рамках проєкту були охоплені такі основні аспекти: тестування авторизації — перевірка коректної роботи login-форми, CSRF-захисту, правильності перенаправлення після входу/виходу, обмеження доступу до функціоналу для неавторизованих користувачів; тестування завантаження файлів — перевірка, чи система приймає графічні файли, чи правильно обробляє їх розмір та тип, чи виконується валідація; перевірка бізнес-логіки обробки зображень — тестування результатів виконання основної функції (видалення фону) шляхом аналізу вихідних файлів, статусів операцій або наявності очікуваних артефактів у файловій системі; тестування виводу шаблонів (views) — перевірка правильності відображення сторінок, наявності необхідних форм, обробки GET/POST-запитів і реакції на помилки.

У рамках CI/CD-конвеєра, побудованого за допомогою Gitlab CI, функціональні тести були інтегровані до конвеєра як один з обов’язкових етапів перед створенням та публікацією Docker-образу. Це дозволило забезпечити гарантовану перевірку працездатності застосунку після кожного внесення змін у кодову базу.

Таким чином, будь-які порушення функціональності миттєво виявляються та блокують подальше розгортання. Навіть за відсутності складної мікросервісної архітектури або зовнішнього API, покриття основного функціоналу тестами дозволяє: забезпечити базовий рівень стабільності; впевнено вносити зміни в код без ризику порушення ключової логіки; мінімізувати вплив людського фактору; підтримувати стандарти якості навіть у невеликому або навчальному проєкті. Таким чином, у реалізованому проєкті тестування функціональності відіграло роль гаранта надійної роботи сервісу, забезпечуючи базову впевненість у кожному релізі системи.

3.4.2 Навантажувальне тестування у Kubernetes

У контексті сучасних мікросервісних архітектур, особливо коли застосунки розгортаються у Kubernetes-кластерах, важливо не лише забезпечити базову функціональність кожного сервісу, але й ретельно протестувати їхню стабільність, продуктивність та здатність витримувати навантаження. Навіть якщо мова йде про відносно простий сервіс, побудований на Django, який не передбачає високих обсягів трафіку, проведення таких тестів у середовищі Kubernetes є критично необхідним для виявлення потенційних проблем, які не завжди очевидні на етапі розробки. Тестування допомагає виявити конфігураційні помилки, «вузькі місця» у продуктивності або непередбачені збої, що можуть виникати через особливості контейнерного середовища або взаємодії компонентів кластера.

Одним із ключових аспектів тестування в такому середовищі є оцінка поведінки сервісу під час різних операцій підтримки та масштабування. Наприклад, було проведено перевірки, як сервіс реагує на повторні перезапуски контейнерів, що можуть відбуватися після оновлення конфігурації або іміджів. Така перевірка дає впевненість, що сервіс коректно ініціалізується і запускається у будь-який момент без ручного втручання. Для забезпечення автоматичного моніторингу стану сервісу в Kubernetes активно застосовувалися readiness та liveness probes.

Ці механізми дозволяють кластеру самостійно визначати, чи готовий сервіс обробляти запити, чи його необхідно перезапустити, що істотно підвищує загальну стійкість системи до збоїв. Крім того, перевіряли, як система поводиться під час rolling update — оновлення, що відбувається поетапно без зупинки роботи сервісу. Важливо, що при такому підході забезпечується відсутність простоїв та безперервність обслуговування користувачів.
Враховуючи можливості Kubernetes, оцінка продуктивності сервісу здійснювалася комплексно. Тестування включало завантаження зображень великого розміру, що дозволяло імітувати реальне навантаження і виявляти межі обробки даних в рамках одного pod-а. Паралельно аналізувалося споживання ресурсів — оперативної пам'яті і процесорного часу — в умовах, коли ресурси були навмисно обмежені відповідними параметрами у YAML-конфігураціях.

Такі тести важливі, щоб переконатися, що сервіс не виходить за межі виділених йому ресурсів, не створює ризику деградації через перевантаження і стабільно працює навіть в умовах конкуренції за обчислювальні потужності. Окрім цього, було проведено імітацію одночасного завантаження зображень, що дало змогу оцінити, чи здатний сервіс витримувати паралельні запити і не втрачати працездатність при зростанні кількості одночасних користувачів.

Kubernetes надає потужні інструменти для забезпечення відмовостійкості систем, але це можливо лише за умови правильного налаштування. У ході тестування було змодельовано сценарії аварійного вимкнення pod-ів, щоб перевірити, чи відновлюється сервіс автоматично за допомогою механізмів реплікації і перезапуску. Важливо було також оцінити, як застосунок реагує на тимчасову втрату мережевого з'єднання або доступу до зовнішніх томів зберігання, що є критичними моментами у реальному продакшн-середовищі. Ці сценарії допомогли зрозуміти, наскільки стабільно Django-застосунок працює в умовах несподіваних збоїв інфраструктури і які заходи потрібно впровадити для підвищення його надійності.

Попри простоту самого застосунку, комплексне тестування у Kubernetes значно підвищило якість його розгортання та експлуатації. Воно дозволило виявити потребу у коректному налаштуванні probes, які відповідають за моніторинг стану подів, що суттєво впливає на підтримання їх життєздатності. Також під час тестування було визначено мінімальні межі ресурсів, необхідних для стабільної роботи, що допомагає уникнути перевитрат та забезпечити оптимальне використання доступних потужностей.

Важливим результатом стало підтвердження того, що налаштований CI/CD-конвеєр, який автоматично виконує деплой після успішного проходження тестів, не порушує стабільність і працездатність сервісу. У цілому, таке тестування є фундаментом для майбутнього масштабування та розвитку проекту. Воно допомагає знизити ризики виникнення несподіваних збоїв у реальному продакшн-середовищі, навіть якщо поточний рівень навантаження не є високим. Впровадження комплексної практики тестування в Kubernetes дозволяє системі адаптуватися до зміни обсягів роботи, гарантує плавне оновлення і безперервну доступність, що є ключовими аспектами сучасних хмарних сервісів.

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі було проведено детальний аналіз процесу створення та впровадження мікросервісної архітектури для веб-застосунку, розробленого на основі фреймворку Django та розгорнутого в Kubernetes-кластері. Головною метою цього етапу стала розробка стабільної, масштабованої та автоматизованої системи, здатної обробляти користувацькі запити в реальному часі з мінімальними затримками та високою надійністю. Аналіз починався із вивчення взаємодії між складовими проєкту. Незважаючи на відсутність зовнішнього API чи окремого сервісного шару, було показано, що навіть у межах одного Django-сервісу можливо забезпечити архітектурну ізоляцію функціональних компонентів.

Поділ бізнес-логіки, шаблонів HTML, обробки CSRF-токенів та внутрішніх запитів дозволив створити чітку модульність, яка забезпечує прозору структуру системи. Це відкриває можливості для подальшого масштабування архітектури до повноцінного мікросервісного середовища. Особливу увагу було приділено організації CI/CD-процесів — невід’ємної частини сучасних DevOps-практик. У межах проєкту реалізовано повністю автоматизований цикл безперервної інтеграції і доставки, який забезпечує послідовне проходження кожного оновлення через етапи тестування, збірки Docker-образу, публікації у реєстр і автоматичного оновлення деплойменту в Kubernetes. Це мінімізує ризики людських помилок, прискорює процес розробки та впровадження змін.

Також у розділі було розглянуто механізми керування конфігураціями та середовищами виконання. Система підтримує розмежування середовищ (локальне, тестове, продакшн) за допомогою змінних середовища, які зберігаються у Kubernetes Secret та ConfigMap.

Такий підхід забезпечує централізоване управління секретами, паролями, ключами та параметрами підключення до баз даних, не жертвуючи при цьому безпекою і гнучкістю. Django-застосунок налаштований так, щоб адаптуватися до змін конфігурації без потреби втручання в код.

Окрему увагу приділено стратегіям розгортання та масштабування. Використання Rolling Update у Kubernetes гарантує безперервну роботу сервісу під час оновлень, усуваючи необхідність повного простою навіть при частих релізах або змінах у функціоналі. Такий підхід забезпечує безперебійну доступність системи, що є критичним для сучасних веб-застосунків.

Таким чином, третій розділ створює фундамент для сучасної, масштабованої та гнучкої інфраструктури, яка відповідає вимогам мікросервісної архітектури. Навіть у рамках одного Django-сервісу можна застосовувати принципи, притаманні повноцінним мікросервісним системам: модульність, ізоляцію компонентів, автоматизоване оновлення, централізоване керування конфігураціями та масштабованість. Описані підходи створюють платформу для подальшого розвитку системи: додавання нових сервісів, інтеграцію зовнішніх API, впровадження кешування, використання CDN і балансування навантаження.

Цей рівень архітектурної зрілості є важливим кроком для забезпечення високої масштабованості, відмовостійкості та продуктивності сучасних веб-застосунків у хмарному середовищі на базі Kubernetes, що має особливе значення в умовах швидкого розвитку IT-інфраструктури та зростаючих вимог до якості обслуговування користувачів.

РОЗДІЛ 4. ТЕХНІЧНА РЕАЛІЗАЦІЯ ІНФРАСТРУКТУРИ

4.1 Автоматизація розгортання через GitLab CI/CD

4.1.1 Побудова пайплайну: build, test, dockerize, deploy

У контексті сучасної розробки веб-застосунків, особливо побудованих за мікросервісною архітектурою, автоматизація процесів розгортання є ключовим фактором забезпечення стабільності та масштабованості системи. Зокрема, реалізація CI/CD стала стандартом, що дозволяє скоротити час між розробкою нових функціональних змін і їхньою появою у продуктивному середовищі. У межах реалізованої системи, побудованої на фреймворку Django з орієнтацією на контейнеризоване розгортання в Kubernetes, було впроваджено повноцінний конвеєр CI/CD за допомогою засобів GitLab CI.

Завдяки використанню конфігураційного файлу .gitlab-ci.yml, розміщеного у корені репозиторію, кожен етап автоматизованого процесу розгортається у вигляді логічно пов’язаних стадій. Такий файл є декларативним описом усіх дій, які повинні виконуватися під час кожного коміту або злиття змін. У розробленій архітектурі пайплайн умовно поділяється на чотири основні етапи: build, test, dockerize та deploy. На початковій стадії відбувається базова перевірка проектної структури та валідація синтаксису коду. Наступним етапом передбачено запуск тестів — наразі реалізовано виконання юніт-тестів, але архітектура дозволяє масштабувати перевірки до інтеграційного або навантажувального рівня.

Рисунок. 1.4 - Приклад процесу тестування у Gitlab CI

Стадія dockerize відповідає за створення Docker-образу, який маркується автоматично на основі хешу коміту або заданої версії, після чого зберігається у відповідному реєстрі образів.

4.1.2 Інтеграція з Kubernetes через YAML-манифести

Одним з найважливіших етапів є deploy, який передбачає оновлення середовища виконання у Kubernetes. Завдяки вбудованій підтримці GitLab CI для роботи з Kubernetes, стало можливим автоматично застосовувати оновлені YAML-манифести та оновлювати існуючі ресурси (зокрема, Deployment) з новою версією контейнера.

Важливо підкреслити гнучкість GitLab CI у контексті розмежування середовищ. У реалізованій системі передбачено наявність окремих середовищ для тестування (staging) та продуктивної експлуатації (production). Стабільність пайплайну підтверджується зручним механізмом логування — у межах кожного виконання CI/CD можна переглянути покрокове виконання інструкцій, швидко виявити помилки або регресії.

Окрім технічних переваг, автоматизація процесів розгортання забезпечує послідовність дій, незалежність від людського фактору та прискорює релізний цикл. Кожне оновлення відбувається ідентичним чином — згідно з попередньо визначеним сценарієм, що забезпечує стабільність функціонування всієї системи. У поєднанні з можливостями Kubernetes, де всі ресурси описуються декларативно у YAML-файлах, це створює передумови для повного контролю над середовищем та його прогнозованою поведінкою. Загалом, інтеграція GitLab CI у процес розробки та обслуговуваннязастосунку у рамках мікросервісної архітектури виявилася обґрунтованим та ефективним рішенням. Незважаючи на відносну простоту самого застосунку, впровадження CI/CD стало вирішальним фактором у підвищенні надійності, гнучкості та структурованості всієї системи. Така автоматизація сприяє не лише швидшому впровадженню змін, але й створює основу для подальшого масштабування системи, розширення її функціоналу та інтеграції з новими сервісами. У підсумку, реалізований підхід дозволяє забезпечити сучасний рівень якості, швидкості розробки та стабільності мікросервісної веб-архітектури у хмарному середовищі.

4.2 Масштабування та балансування в Kubernetes

4.2.1 HPA та управління ресурсами

Масштабування — критичний аспект життєздатності веб-системи, який набуває особливого значення на етапі її експлуатації. У мікросервісній архітектурі, особливо при використанні Kubernetes, можливості масштабування суттєво розширюються порівняно з традиційними монолітними застосунками. Це досягається завдяки розділенню застосунку на незалежні компоненти (сервіси), кожен з яких може масштабуватись автономно — залежно від навантаження, яке він обслуговує.

У розробленій інфраструктурі система не розрахована на обслуговування великої кількості користувачів з перших днів, однак її архітектура закладена з урахуванням майбутнього зростання. Django-застосунок, який реалізує функції авторизації користувачів та обробки зображень, контейнеризовано з використанням Docker і розгорнуто у Kubernetes у вигляді окремого об’єкта Deployment. Такий підхід дозволяє масштабувати backend незалежно від змін у коді — лише за рахунок зміни конфігураційних параметрів Kubernetes.

Базова конфігурація передбачає одну репліку пода. Проте параметр replicas у YAML-описі можна змінити в будь-який момент — вручну або автоматизовано. Kubernetes здатен динамічно запускати додаткові екземпляри подів на доступних вузлах кластера у відповідь на зростання кількості запитів. Завдяки сервісу типу ClusterIP, який виконує роль внутрішнього балансувальника, кожен вхідний запит автоматично спрямовується на доступний под — прозоро для кінцевого користувача.

Серед вбудованих у Kubernetes можливостей масштабування особливу увагу заслуговує Horizontal Pod Autoscaler (HPA) — компонент, який автоматично змінює кількість реплік на основі споживання ресурсів (CPU, пам’ять). У поєднанні з системою метрик, HPA забезпечує динамічну реакцію на зміну навантаження без необхідності втручання з боку адміністратора. Це дозволяє інфраструктурі залишатися адаптивною навіть у випадках непередбачуваних піків трафіку.

Масштабування бази даних на поточному етапі не реалізоване, однак архітектура передбачає її винесення у керований сервіс, з можливістю реалізації реплікації або шардінгу в майбутньому. Це відкриває шлях до побудови більш стійкої до навантаження та відмов інфраструктури з високою доступністю.

Важливою перевагою є повна автоматизація масштабування на рівні CI/CD. Зміни в конфігурації кластера, зокрема зміна кількості реплік або налаштувань autoscaling, контролюються через GitLab CI. Кожен коміт до основної гілки викликає пайплайн, що включає етап автоматичного деплою із застосуванням оновлених YAML-файлів. Такий підхід гарантує узгодженість середовищ, дозволяє виявляти конфлікти до потрапляння змін у production і підтримує повторюваність усіх дій. У разі масштабування всієї системи, наприклад, при значному зростанні трафіку або збільшенні кількості запитів на обробку зображень, інфраструктура дозволяє розширення у кілька кліків — додавання нових реплік, масштабування сервісів або винесення частин логіки у спеціалізовані сервіси. При цьому кодова база залишається незмінною, а всі зміни відбуваються лише на рівні конфігурацій або додаткових компонентів. Це суттєво спрощує підтримку та розвиток системи.

Крім цього, інфраструктура повністю описана у вигляді YAML-манифестів. Це забезпечує портативність — розгортання в новому середовищі (наприклад, staging чи інший кластер) відбувається без додаткових маніпуляцій. Такий підхід є важливим фактором у забезпеченні масштабованості на організаційному рівні: при зміні команди, середовища чи умов роботи система зберігає свою гнучкість.

У підсумку, навіть для порівняно простого Django-застосунку, що включає авторизацію користувачів і базову обробку зображень, було реалізовано гнучку інфраструктуру, яка готова до масштабування у різних напрямках. Це демонструє переваги поєднання контейнеризації, Kubernetes і CI/CD як інструментів, що дозволяють системам залишатися стабільними, передбачуваними та адаптивними до будь-яких змін навантаження чи бізнес-вимог.

4.2.2 Ingress-контролер і балансування навантаження

Балансування навантаження є ключовим елементом у сучасних розподілених системах, особливо в архітектурах, що базуються на мікросервісах та використовують Kubernetes. Основна його функція — рівномірно розподіляти вхідний трафік між кількома екземплярами сервісів, зокрема подами, аби запобігти перевантаженню окремих компонентів і забезпечити стабільну продуктивність. У рамках даного проєкту, що реалізований на базі Django у середовищі Kubernetes, механізми балансування реалізовані на кількох рівнях системи.

На першому рівні за балансування відповідають Kubernetes-сервіси з типами ClusterIP або LoadBalancer. Вони автоматично розподіляють трафік між активними подами, використовуючи алгоритми типу Round Robin або Least Connections, завдяки чому система може динамічно масштабуватись без необхідності змін у програмному коді. Таким чином, кожен запит автоматично спрямовується до одного з доступних екземплярів застосунку, забезпечуючи рівномірне навантаження.

Як зовнішній балансувальник, у проєкті застосовується Ingress-контролер, зокрема рішення на базі NGINX. Він виконує роль централізованої точки доступу для всіх вхідних HTTP-запитів і забезпечує маршрутизацію до відповідних сервісів у кластері. Крім того, Ingress виконує низку додаткових функцій: термінацію TLS-з'єднань, переписування URL-шляхів, обмеження доступу та ведення логів запитів. Такий підхід дозволяє централізовано адмініструвати політику доступу та безпеки для всієї інфраструктури мікросервісів.

Kubernetes також забезпечує масштабування шляхом реплікації подів. У маніфесті Deployment можна задати бажану кількість реплік для кожного сервісу, зокрема для Django-застосунку. Kubernetes автоматично слідкує за тим, щоб кожна репліка залишалась у працездатному стані. Коли надходить новий запит, він перенаправляється до однієї з активних реплік через Kubernetes-сервіс або Ingress, завдяки чому забезпечується висока доступність і продуктивність. У разі зростання навантаження система може бути масштабована вручну або автоматично через HPA (Horizontal Pod Autoscaler), після чого балансувальник автоматично починає враховувати нові поди в розподілі запитів. Інтеграція механізмів балансування з CI/CD-процесом — ще один критичний аспект. Під час розгортання нових версій застосунку через GitLab CI використовуються rolling updates, які дозволяють оновлювати систему без простоїв. Нові поди поступово вводяться в експлуатацію, а старі відключаються лише після того, як нові екземпляри повністю готові до обробки трафіку. Це гарантує безперервну роботу сервісів навіть у період активних змін.

Підсумовуючи, варто наголосити, що балансування навантаження у Kubernetes — це не лише технічна функція, а стратегічний підхід до побудови стійкої, гнучкої та масштабованої інфраструктури. У контексті навіть відносно простих систем, як-от Django-застосунок для обробки зображень, продумане балансування забезпечує високу доступність, адаптивність до змін навантаження та легкість масштабування без значних зусиль з боку розробників.

4.3 Забезпечення безпеки інфраструктури

4.3.1 Конфігурація секретів і ролей доступу

Процес розгортання програмного забезпечення є критично важливою складовою життєвого циклу розробки, особливо коли йдеться про мікросервісну архітектуру. У таких системах потрібно не лише швидко й ефективно впроваджувати нові функції, але й робити це без ризику порушення доступності сервісу. Тому основна мета деплою полягає в автоматизації, контрольованості та безперервності. У рамках даного проєкту, побудованого на базі Django та контейнеризованого за допомогою Docker, основним інструментом оркестрації є Kubernetes, а за CI/CD відповідає GitLab CI. Запуск процесу розгортання ініціюється зі змінами у вихідному коді — зазвичай при коміті в одну з головних гілок, таких як main або production. Ці зміни автоматично активують GitLab CI пайплайн, який виконує послідовність завдань: перевірку якості коду, створення Docker-образу, публікацію образу в реєстр, оновлення Kubernetes-манифестів і безпосереднє застосування оновлень у кластері. Уся процедура відбувається без ручного втручання, що дозволяє досягти високої повторюваності та мінімізувати ризик помилок людини.

На першому етапі збирається Docker-образ застосунку. Для цього використовується Dockerfile, який описує усі необхідні кроки — від встановлення залежностей до копіювання коду проєкту. Після завершення цього етапу образ отримує тег, наприклад, у вигляді хешу коміту або номеру версії, після чого зберігається у Docker-реєстрі (GitLab Container Registry або альтернативному).

Далі в пайплайні відбувається оновлення описів розгортання у Kubernetes. Це може бути як пряме редагування YAML-файлів, так і використання Helm-чартів, що дають більше гнучкості у налаштуванні. Основне завдання цього етапу — вказати Kubernetes, що потрібно використовувати нову версію образу. GitLab Runner має доступ до кластера через kubeconfig, і застосування змін виконується за допомогою kubectl apply. Завдяки механізму rolling update, Kubernetes поступово замінює застарілі поди новими, не порушуючи роботи сервісу. При цьому використовуються readinessProbe та livenessProbe, які дозволяють системі перевіряти, чи кожен новий екземпляр сервісу готовий до роботи, і вчасно реагувати на помилки.

Не менш важливою складовою є правильна робота з конфігураціями. Усі змінні середовища, ключі, паролі й інші налаштування виносяться в Kubernetes-об'єкти ConfigMap та Secret. Це дозволяє змінювати поведінку застосунку без необхідності збирати новий образ або редагувати код. Крім того, у GitLab CI можна передбачити крок, який оновлює ці об'єкти при розгортанні нової версії. Загалом, описаний підхід має кілька ключових переваг. По-перше, він забезпечує передбачуваність і контрольованість — кожен реліз фіксується, логіку деплою можна відтворити в будь-який момент. По-друге, автоматизація знижує вплив людського чинника — це особливо критично при роботі з продакшен-середовищем. По-третє, система легко масштабується — змінити параметри розгортання або кількість подів можна просто через оновлення конфігурацій, без складних маніпуляцій з кодом.

У підсумку, використання зв'язки GitLab CI, Docker і Kubernetes дозволяє побудувати ефективний, безпечний і масштабований процес розгортання, навіть якщо йдеться про відносно просту архітектуру.

Автоматизація не тільки спрощує підтримку, а й наближає рівень операційної стабільності до стандартів великих продакшен-платформ. Завдяки цьому кожна зміна доставляється у середовище швидко, без збоїв та з повним контролем над якістю — що є критично важливим для сучасних веб-застосунків.

4.3.2 Мережеві політики та безпечна взаємодія

Комунікація між користувачем і системою захищена протоколом HTTPS. Цей захист реалізується через Ingress-контролер або зворотний проксі, що відповідає за шифрування трафіку між клієнтом та сервером. Веб-застосунок на Django додатково використовує механізм захисту від підробки міжсайтових запитів (CSRF), який перевіряє наявність дійсного токена в кожному запиті, що змінює стан системи. Таким чином, навіть при простій авторизації через сесію, користувач надійно захищений від сторонніх атак, спрямованих на зміну його даних.

Для забезпечення міжсервісної безпеки всередині кластеру застосовано політику мережевих обмежень (NetworkPolicy). Цей механізм дозволяє чітко визначити, які сервіси можуть взаємодіяти один з одним на рівні мережі. Наприклад, веб-додаток має дозвіл на з’єднання з базою даних, але не може звертатися до інших сервісів, які не є необхідними для його роботи. Такий підхід значно знижує ризики розповсюдження атаки в разі, якщо один з подів буде скомпрометовано. Не менш важливою складовою загальної безпеки є захист процесів безперервної інтеграції та доставки (CI/CD). У системі використовується GitLab CI, де кожен пайплайн виконується в ізольованому середовищі. GitLab Runner працює в контейнері з обмеженим доступом до зовнішньої мережі, що знижує ризики втручання під час виконання скриптів.

Конфіденційні дані — такі як токени для доступу до Kubernetes або SSH-ключі — зберігаються в CI Variables і не виводяться у відкриті логи. Перед розгортанням застосунку перевіряється цілісність і достовірність контейнерного образу за допомогою підпису та контрольних сум. Це запобігає випадковому чи навмисному втручанню в процес побудови та доставки коду.

4.4 Тестування: надійність та масштабованість

Автоматизоване тестування є ключовим елементом сучасного процесу створення веб-застосунків, особливо у межах реалізації DevOps-підходів і мікросервісної архітектури. Його головна мета полягає у своєчасному виявленні помилок, забезпеченні стабільності роботи системи, контролі якості програмного коду та змін в інфраструктурі ще до потрапляння релізу в продуктивне середовище. У рамках реалізованого проєкту впроваджено базову модель автоматизованого тестування, інтегровану в CI/CD-процес за допомогою GitLab CI, що дозволяє реалізувати безперервну перевірку та автоматичне розгортання застосунку у Kubernetes-кластер. GitLab CI відіграє ключову роль у життєвому циклі розробки: кожен новий коміт у репозиторій активує запуск пайплайну, який складається з послідовних стадій, серед яких тестування передує побудові Docker-образу та деплою в кластер. Такий підхід гарантує, що нестабільний або помилковий код не потрапить до продакшн-середовища.

У рамках проєкту тестування реалізовано на ранньому етапі, одразу після lint-перевірки. Воно охоплює перевірку працездатності Django-застосунку, коректності рендерингу основних сторінок, роботи middleware, валідності форм та функціональності базових дій користувача. Для запуску тестів використовується стандартний механізм Django — команда manage.py test, яка виконує юніт-тести основних модулів системи.

Зокрема, тестується завантаження сторінки входу, коректна обробка, поведінка форм при передачі валідних і невалідних зображень, базова логіка видалення фону із застосуванням тестових зображень або мок-даних, а також робота з базою даних — створення, збереження та отримання об'єктів. Усі тести є ізольованими, а база даних створюється автоматично в CI-середовищі на основі SQLite або PostgreSQL залежно від конфігурації, що дозволяє уникнути залежності від зовнішніх сервісів.

У GitLab CI пайплайні передбачено окрему стадію test, яка виконується перед етапом deploy. Вона включає ініціалізацію Python-середовища, встановлення всіх залежностей через pip install -r requirements.txt, запуск тестів за допомогою Django-команди, а також обробку результатів — при потребі з формуванням логів або звітів. Якщо хоча б один тест не проходить успішно, увесь пайплайн переривається, і жодні подальші дії не виконуються. Це дозволяє гарантувати, що до кластеру не потрапить версія застосунку з відомими помилками.

ВИСНОВОК ДО РОЗДІЛУ 4

У четвертому розділі було реалізовано безпосереднє впровадження мікросервісної системи з акцентом на розгортання, автоматизацію процесів CI/CD, безпеку інфраструктури, контроль версій та підтримку стабільності веб-застосунку. У межах цієї частини роботи здійснено повний цикл підготовки та налаштування середовища для безперервної інтеграції та доставки, адаптованої до реального стеку, що включає Django, Docker, GitLab CI та Kubernetes. Особливу увагу приділено тому, щоб кожен крок від коду до деплою був автоматизований, відтворюваний та підконтрольний.

Реалізація GitLab CI-пайплайну дозволила автоматизувати основні етапи життєвого циклу застосунку — збірку Docker-образу, прогін базових перевірок, деплой до Kubernetes-кластера. Таким чином, кожна зміна в кодовій базі автоматично обробляється, збирається та може бути розгорнута у відповідне середовище. Це значно знижує ризик людського фактору, прискорює цикл доставки та забезпечує стабільну інтеграцію змін.

Завдяки збереженню Docker-образів у GitLab Registry впроваджується чіткий зв’язок між версією коду й відповідною інстанцією застосунку в Kubernetes. У процесі реалізації розгортання особливу роль відіграв Kubernetes — як основна платформа для управління мікросервісною інфраструктурою. Була налаштована система опису застосунку за допомогою YAML-конфігурацій, що описують об’єкти типу Deployment, Service та інші необхідні ресурси. Усі сервіси було ізольовано та забезпечено їхнє незалежне масштабування, що в перспективі дозволяє системі адаптуватися до змін у навантаженні. Не менш важливою складовою розділу стала робота з безпекою та конфігурацією середовища.

Усі критичні параметри, такі як токени доступу або ключі, було винесено у Kubernetes Secrets, що мінімізує ризики витоку інформації. Крім того, впроваджено базову логіку поділу середовищ (наприклад, stage/production) з можливістю незалежного тестування та релізу, що дозволяє уникати прямого впливу нестабільного коду на основне середовище користувачів. Особливої уваги заслуговує реалізована стратегія контролю версій, що базується на Git-тегах. Такий підхід дозволяє точно ідентифікувати кожен реліз, забезпечувати зв'язність між кодом, Docker-образами та деплоєм, а також здійснювати швидкий rollback у разі виявлення помилок. Можливість розгортання саме за тегованою версією значно підвищує надійність системи, зменшує час на вирішення інцидентів та покращує процес супроводу. Крім основних технічних аспектів, у межах реалізації було приділено увагу підтримці якості та надійності.

Навіть за відсутності розширених інструментів моніторингу чи логування, системі забезпечено базову стабільність завдяки структурованості процесів, наявності тестів, автоматичному деплою та фіксації версій. Це створює базис для подальшого масштабування проєкту як у технічному, так і в архітектурному вимірах. У підсумку розділ продемонстрував, як на практиці побудувати повний життєвий цикл роботи над мікросервісним веб-застосунком — від коду й контейнеризації до деплою, захисту та експлуатації у кластері Kubernetes. Незважаючи на обмежену складність проєкту, реалізовані процеси є універсальними та масштабованими, що дозволяє у подальшому розвивати систему у бік підвищення надійності, автоматизації й масштабованості.

ВИСНОВКИ

У результаті проведеної роботи було реалізовано повноцінну мікросервісну систему для автоматизованого оброблення зображень, зокрема — для видалення фону. Розроблений застосунок демонструє ефективне поєднання сучасних підходів до архітектури, інфраструктури та розробки хмарно-орієнтованих сервісів. Основна увага приділялася тому, щоб створене рішення було не лише функціональним, але й здатним масштабуватися, автоматично оновлюватися, бути надійним і безпечним у реальному продакшн-середовищі.

У першому розділі було проведено аналітичний огляд сучасних архітектурних підходів у сфері веб-розробки. Розглянуто особливості мікросервісної моделі, переваги децентралізованої структури, а також ключові технології, що підтримують реалізацію такої архітектури — контейнеризацію, інструменти автоматизації, оркестрацію, моніторинг та CI/CD. На основі порівняльного аналізу було обґрунтовано вибір стеку технологій, серед яких — Docker, Kubernetes, GitLab CI, Django, PostgreSQL, HTML/CSS.

У другому розділі сформовано вимоги до функціоналу системи, окреслено архітектурну модель та взаємодію між компонентами. Було обґрунтовано використання REST API для міжсервісної комунікації, застосування контейнеризації для ізоляції середовища кожного сервісу, а також обрано CI/CD як основний принцип автоматизації життєвого циклу застосунку. Особливу увагу приділено оркестрації в Kubernetes: налаштування кластеру, управління конфігураціями та секретами, контроль доступу на основі ролей. Розділ завершено технічною підготовкою до розгортання та запуску системи в умовах хмарного середовища.

У третьому розділі реалізовано серверну логіку на основі Django. Було створено API-сервіси, що обробляють запити на завантаження та обробку зображень, зокрема з використанням алгоритмів для видалення фону. Реалізовано базову валідацію вхідних файлів, обробку помилок, підключено засоби логування й тестування. Застосунок було контейнеризовано, конфігурацію описано у Dockerfile, що дозволяє запускати його на будь-якому середовищі без додаткових налаштувань. Особливе значення мало впровадження CI/CD, який автоматизує збірку, перевірку коду, створення контейнерів та їх розгортання у кластері.

Четвертий розділ був присвячений реалізації інфраструктурної частини проєкту. Через пайплайни GitLab CI було реалізовано автоматизоване тестування та розгортання сервісів у Kubernetes-кластері. Налаштовано правила горизонтального масштабування, балансування навантаження та моніторингу ресурсів. Для безпеки конфігурації застосовано механізми зберігання секретів та розмежування прав доступу. Паралельно з деплоєм виконувалося тестування стабільності сервісів, перевірка відповідності ресурсних обмежень і коректної роботи health checks. Це дозволило забезпечити повноцінне функціонування додатку в умовах реального навантаження.

У результаті розробки вдалося реалізувати масштабовану, модульну та легко розширювану систему з автоматизованими процесами оновлення й контролю якості. Додаток виконує свою основну функцію — видалення фону з графічних зображень — ефективно, із мінімальним часом обробки, підтримуючи стабільність при підвищених навантаженнях. Застосований підхід і реалізовані рішення можуть бути масштабовані або адаптовані для інших задач з обробки мультимедійного контенту. Таким чином, розроблена система не лише відповідає поставленим вимогам, але й демонструє потенціал для подальшого розвитку, інтеграції нових сервісів та переходу до повноцінної платформи обробки візуальних даних у хмарному середовищі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bass, L., Weber, I., & Zhu, L. *DevOps: A Software Architect's Perspective*. Addison-Wesley, 2015. – Друк.
2. Newman, S. *Building Microservices: Designing Fine-Grained Systems*. 2-ге вид. O'Reilly Media, 2021. – Друк.
3. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J. Borg, Omega, and Kubernetes // *Communications of the ACM*. – 2016. – Т. 59, № 5. – С. 50–57. – DOI: https://doi.org/10.1145/2890784. – Електрон. джерело.
4. Turnbull, J. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014. – Друк.
5. Red Hat. *Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications*. O'Reilly Media, 2022. – Друк.
6. GitLab Documentation. GitLab CI/CD Pipelines [Електронний ресурс]. – Режим доступу: https://docs.gitlab.com/ee/ci/.
7. Django Software Foundation. Django Documentation [Електронний ресурс]. – Режим доступу: https://docs.djangoproject.com/.
8. Садовий М. І. Архітектурні патерни в мікросервісних системах // *Інформаційні технології та комп'ютерна інженерія*. – 2021. – № 3. – С. 22–27.

9. Глущенко В. В. Впровадження безпеки в CI/CD пайплайнах з використанням Kubernetes // *Науковий вісник НТУУ «КПІ»*. – 2020. – № 5. – С. 41–47.

10. Левченко С. С. Автоматизація розгортання веб-додатків з використанням Docker та Kubernetes // *Системи обробки інформації*. – 2022. – № 2. – С. 34–39.

11. Олійник Ю. Б. Аналіз платформ для оркестрації контейнеризованих застосунків // *Вісник ЖДТУ. Серія: Технічні науки*. – 2020. – № 3. – С. 67–73.

12. Fowler, M. Microservices [Електронний ресурс]. – Режим доступу: https://martinfowler.com/articles/microservices.html.

13. HashiCorp. Vault Documentation [Електронний ресурс]. – Режим доступу: https://developer.hashicorp.com/vault.

14. Hightower, K., Burns, B., Beda, J. *Kubernetes: Up and Running*. O'Reilly Media, 2017. – Друк.

15. Vehent, J. *Securing DevOps: Security in the Cloud*. O'Reilly Media, 2020. – Друк.

ДОДАТОК А

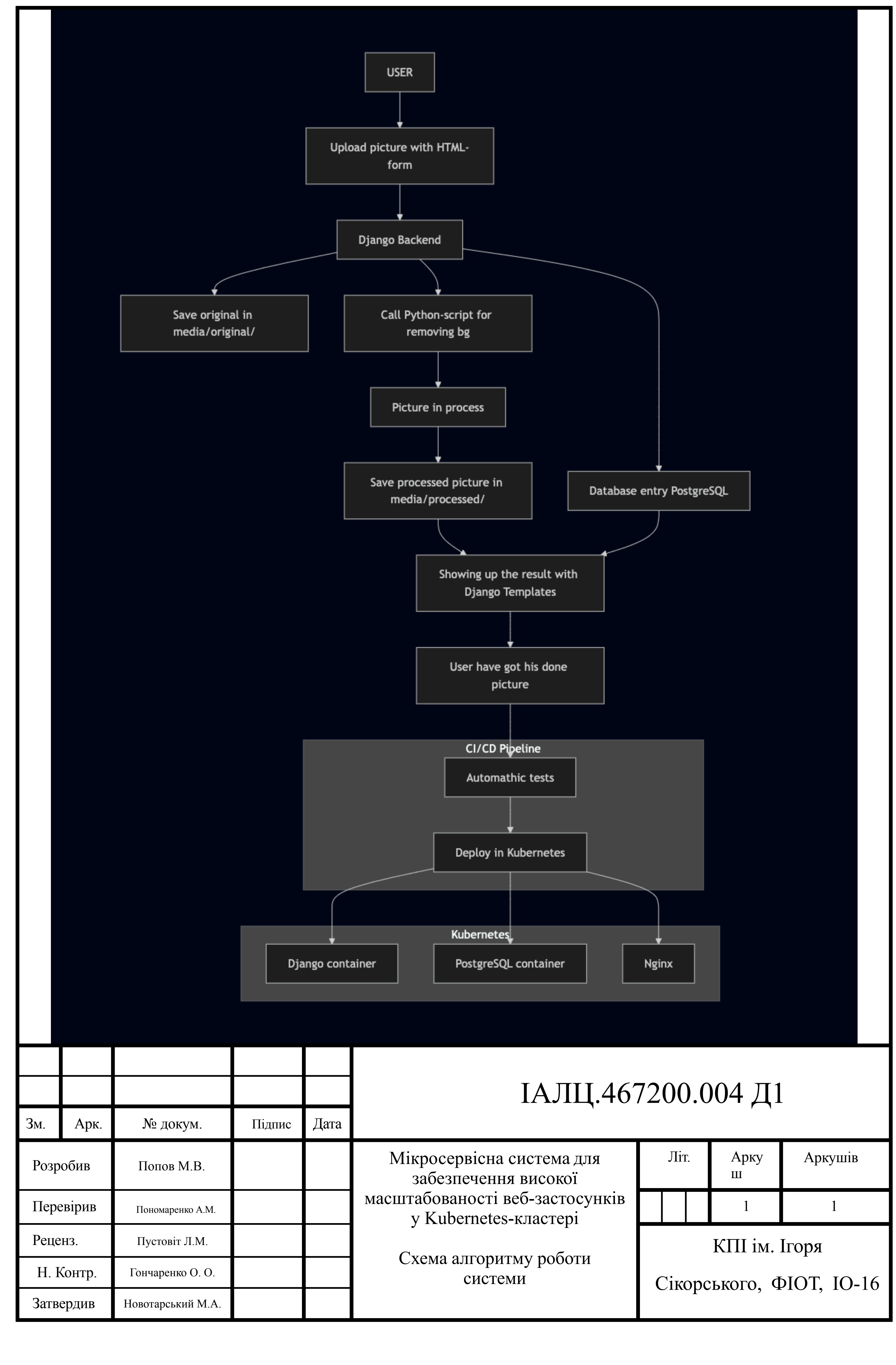
Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері

Модель процесу обробки зображень

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2025 р


USER
Upload picture with HTML-form
Django Backend
Save original in media/original/
Call Python-script for removing bg
Picture in process
Save processed picture in media/processed/
Database entry PostgreSQL
Showing up the result with Django Templates
User have got his done picture
CI/CD Pipeline
Automathic tests
Deploy in Kubernetes
Kubernetes
Django container
PostgreSQL container
Nginx
ІАЛЦ.467200.004 Д1
Зм.
Арк.
№ докум.
Підпис
Дата
Розробив
Попов М.В.
Перевірив
Пономаренко А.М.
Реценз.
Пустовіт Л.М.
Н. Контр.
Гончаренко О. О.
Затвердив
Новотарський М.А.
Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері
Схема алгоритму роботи системи
Літ.
Аркуш
Аркушів
1
1
КПІ ім. Ігоря Сікорського, ФІОТ, ІО-16

ДОДАТОК Б

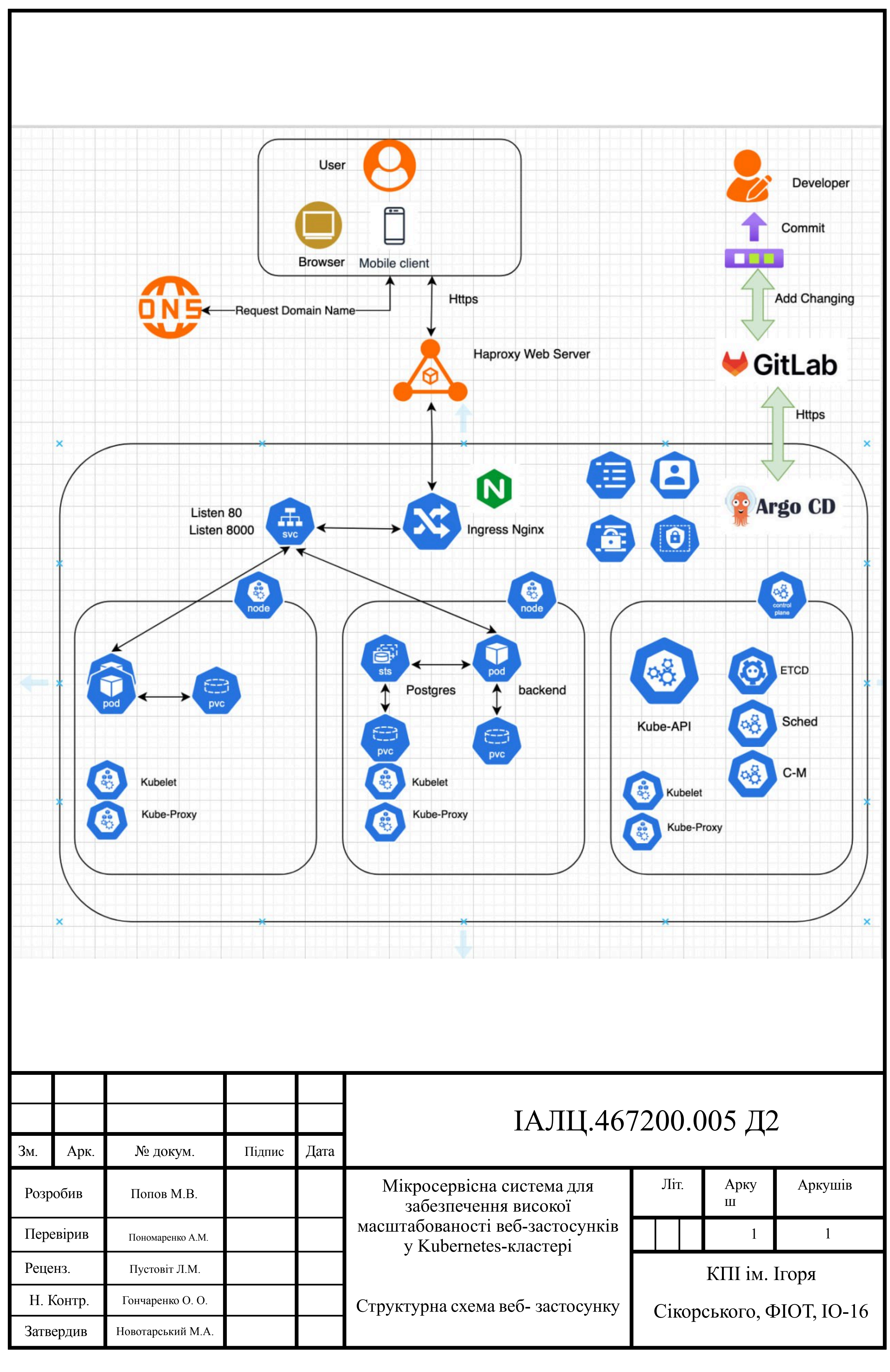
Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері

Архітектурна модель системи

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2025 р



Зм.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.467200.005 Д2		
Розробив		Попов М.В.			Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері	Літ.	Аркуш	Аркушів
Перевірив		Пономаренко А.М.					1	1
Реценз.		Пустовіт Л.М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІО-16		
Н. Контр.		Гончаренко О. О.			Структурна схема веб- застосунку			
Затвердив		Новотарський М.А.						

ДОДАТОК В

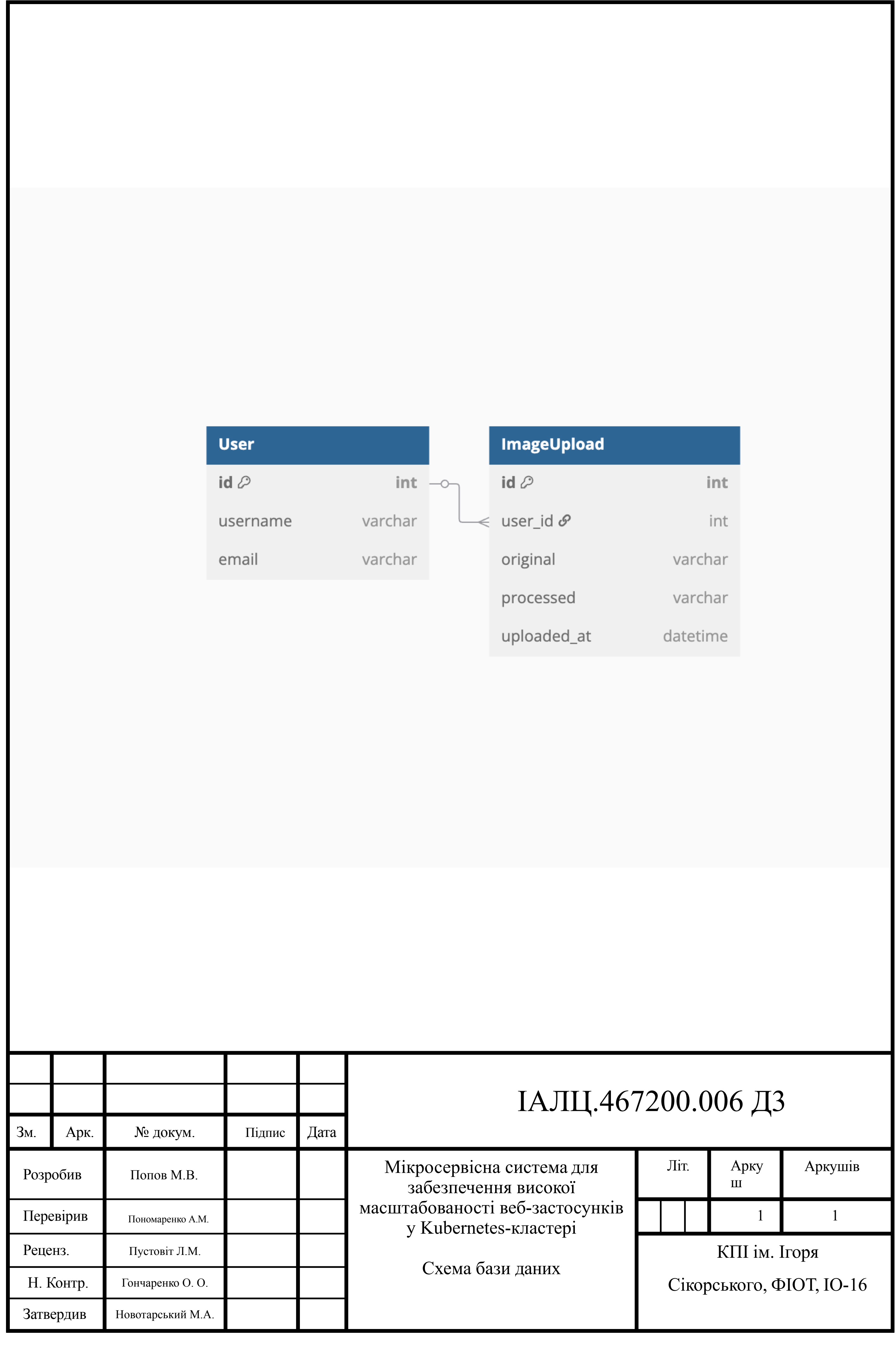
Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері

Модель структури бази даних

ІАЛЦ.472500.006 Д3

Аркушів 1

Київ 2025 р


User
id int
username varchar
email varchar
ImageUpload
id int
user_id int
original varchar
processed varchar
uploaded_at datetime
ІАЛЦ.467200.006 Д3
Зм.
Арк.
№ докум.
Підпис
Дата
Розробив
Попов М.В.
Перевірив
Пономаренко А.М.
Реценз.
Пустовіт Л.М.
Н. Контр.
Гончаренко О. О.
Затвердив
Новотарський М.А.
Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері
Схема бази даних
Літ.
Аркуш
Аркушів
1
1
КПІ ім. Ігоря Сікорського, ФІОТ, ІО-16

ДОДАТОК Г

Мікросервісна система для забезпечення високої масштабованості веб-застосунків у Kubernetes-кластері

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 19

Київ 2025 р

```python
from django.shortcuts import render, redirect, get_object_or_404
from django.contrib.auth.decorators import login_required
from django.contrib.auth import login
from django.views.decorators.csrf import csrf_exempt

from .forms import RegisterForm, ImageUploadForm
from .models import ImageUpload
from PIL import Image
from io import BytesIO
from django.core.files.base import ContentFile

def register(request):
    if request.method == 'POST':
        form = RegisterForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)
            return redirect('index')
    else:
        form = RegisterForm()
    return render(request, 'registration/register.html', {'form': form})

@login_required
def index(request):
    from rembg import remove
    if request.method == 'POST':
        form = ImageUploadForm(request.POST, request.FILES)
        if form.is_valid():
            instance = form.save(commit=False)
            instance.user = request.user
            # Удаляем фон
            input_image = Image.open(instance.original).convert("RGBA")
            output_image = remove(input_image)
            buffer = BytesIO()
            output_image.save(buffer, format='PNG')
            instance.processed.save(f"processed_{instance.original.name}", ContentFile(buffer.getvalue()), save=False)
            instance.save()
            return redirect('index')
    else:
        form = ImageUploadForm()

    images = ImageUpload.objects.filter(user=request.user).order_by('-uploaded_at')
    return render(request, 'index.html', {'form': form, 'images': images})

@login_required
def delete_image(request, pk):
    image = get_object_or_404(ImageUpload, pk=pk, user=request.user)
    image.original.delete()
    if image.processed:
        image.processed.delete()
    image.delete()
    return redirect('index')
```

```
from django.urls import path
from .views import index, register, delete_image

urlpatterns = [
    path('', index, name='index'),
    path('register/', register, name='register'),
    path('delete/<int:pk>/', delete_image, name='delete')
]
```

————

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User
from .models import ImageUpload

class RegisterForm(UserCreationForm):
    email = forms.EmailField(required=True)

    class Meta:
        model = User
        fields = ['username', 'email', 'password1', 'password2']

class ImageUploadForm(forms.ModelForm):
    class Meta:
        model = ImageUpload
        fields = ['original']  # только поле загрузки

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fields['original'].widget.attrs.update({
            'class': 'upload-input',
            'accept': 'image/*',
        })
```

———

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('web.urls')),
    path('accounts/', include('django.contrib.auth.urls')),
]

if not settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
    urlpatterns += static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
else:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

```
{% load static %}
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8" />
  <title>Удаление фона</title>
  <link rel="stylesheet" href="{% static 'web/css/style.css' %}">
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
</head>
<body>

    <header>
        <div class="header-logo">
            <span class="logo">BG REMOVER</span>
        </div>
        <div class="header-header">
            <div class="user-block">
                <span style="font-size: 24px; color: gray;">👤</span>
                <p class="username">{{ request.user.username }} </p>
            </div>
            <a href="{% url 'logout' %}" class="logout-btn">Log Out</a>
        </div>
    </header>
    <div class="container">
      <div class="sidebar">
        <div class="image">
          <img src="{% static 'web/img/images.png' %}" alt="" class="side_image">
        </div>
        <div class="description">
            <h1>Free Image<br>Background Remover</h1>
            <br>
            <h3>Instantly remove background from any image using Pixelcut’s AI<br>background remover. It’s free and
no sign-up required!</h3>
        </div>
      </div>
      <div class="form">
        <form method="POST" enctype="multipart/form-data">
          {% csrf_token %}

          <div class="file-upload-wrapper">
            <label for="id_original" class="custom-file-label">📁 UPLOAD PICTURE</label>
            {{ form.original }}

          </div>

          <div class="name">
            <span id="file-name">File haven't chosen yet</span>
          </div>

          <button type="submit" class="upload-btn">REMOVE BG</button>
        </form>
      </div>
    </div>
```

```
    <div class="done_images">
      <div class="clear_images">
        {% for img in images %}
          <div class="card">
            <img src="{{ img.processed.url }}" alt="Обработанное изображение" />
            <div class="actions">
              <a href="{{ img.processed.url }}" download class="download-btn">DOWNLOAD</a>
              <a href="{% url 'delete' img.pk %}" class="delete-btn">REMOVE</a>
            </div>
          </div>
        {% empty %}
          <p class="no-images">No uploaded pictures yet.</p>
        {% endfor %}
      </div>
    </div>
</body>
  <script>
    document.addEventListener('DOMContentLoaded', function () {
      const fileInput = document.getElementById('id_original');
      const fileName = document.getElementById('file-name');

      fileInput.addEventListener('change', function () {
        if (fileInput.files.length > 0) {
          fileName.textContent = "File was chosen";
        } else {
          fileName.textContent = "File haven't chosen yet";
        }
      });
    });
  </script>
</html>

——
{% load static %}
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="UTF-8">
<title>Вход</title>
<link rel="stylesheet" href="{% static 'web/css/style.css' %}">
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
</head>
<body>
  <div class="sign-up">
    <h1>Вход</h1>
    <form method="post">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit" class="sign">Войти</button>
    </form>
    <p>Нет аккаунта? <a href="{% url 'register' %}">Зарегистрироваться</a></p>
  </div>
</body>
</html>
```

```
{% load static %}
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="UTF-8">
<title>Регистрация</title>
<link rel="stylesheet" href="{% static 'web/css/register.css' %}">
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
</head>
<body>
  <div class="sign-in">
    <h1>Регистрация</h1>
    <form method="post">
      {% csrf_token %}

      <div class="form-group">
        <label for="{{ form.username.id_for_label }}">Имя пользователя</label>
        {{ form.username }}
        {% if form.username.errors %}
          <div class="error">{{ form.username.errors.0 }}</div>
        {% endif %}
      </div>

      <div class="form-group">
        <label for="{{ form.email.id_for_label }}">Email</label>
        {{ form.email }}
        {% if form.email.errors %}
          <div class="error">{{ form.email.errors.0 }}</div>
        {% endif %}
      </div>

      <div class="form-group">
        <label for="{{ form.password1.id_for_label }}">Пароль</label>
        {{ form.password1 }}
        {% if form.password1.errors %}
          <div class="error">{{ form.password1.errors.0 }}</div>
        {% endif %}
      </div>

      <div class="form-group">
        <label for="{{ form.password2.id_for_label }}">Подтвердите пароль</label>
        {{ form.password2 }}
        {% if form.password2.errors %}
          <div class="error">{{ form.password2.errors.0 }}</div>
        {% endif %}
      </div>

      <button type="submit">Зарегистрироваться</button>
    </form>
    <p>Уже есть аккаунт? <a href="{% url 'login' %}">Войти</a></p>
  </div>
</body>
</html>
```

```
* {
    margin: 0;
    padding: 0;
    font-family: Arial, sans-serif;
}

.sign-in {
  background: #ffffff;
  padding: 40px 30px;
  border-radius: 12px;
  box-shadow: 0 10px 25px rgba(0, 0, 0, 0.1);
  width: 80%;
  max-width: 400px;
  margin: 0 auto;
  margin-top: 80px;

}

.sign-in h1 {
  margin-bottom: 24px;
  text-align: center;
  font-size: 28px;
  color: #333;
}

.form-group {
  margin-bottom: 18px;
}

label {
  display: block;
  margin-bottom: 6px;
  font-weight: bold;
  color: #333;
}

input {
  width: 100%;
  padding: 10px 5px;
  border: 1px solid #ccc;
  border-radius: 6px;
  font-size: 15px;
  transition: border-color 0.3s;
}

input:focus {
  border-color: #007bff;
  outline: none;
}
```

```
button {
  width: 100%;
  padding: 12px;
  background: #007bff;
  color: white;
  border: none;
  border-radius: 6px;
  font-size: 16px;
  cursor: pointer;
  transition: background 0.3s;
}

button:hover {
  background: #0056b3;
}

p {
  margin-top: 16px;
  text-align: center;
  font-size: 14px;
}

p a {
  color: #007bff;
  text-decoration: none;
}

p a:hover {
  text-decoration: underline;
}

.error {
  color: #ff3333;
  font-size: 14px;
  margin-top: 4px;
}
```

```
* {
   margin: 0;
   padding: 0;
   font-family: Arial, sans-serif;
}

input[type="file"] {
  display: none;
}

#file-name {
   color: grey;
   font-size: 13px;
   font-weight: 700;
}

.custom-file-label {
  display: inline-block;
  padding: 12px 20px;
  background-color: #6C63FF;
  color: white;
  border-radius: 8px;
  cursor: pointer;
  font-weight: bold;
  margin-bottom: -13px;
}

.upload-btn {
   background: linear-gradient(135deg, #4da6ff, #0055ff);
   color: #fff;
   border-radius: 15px;
   width: 150px;
   height: 37px;
   border: none;
   cursor: pointer;
}

.sign-up {
  background: #ffffff;
  padding: 40px 30px;
  border-radius: 12px;
  box-shadow: 0 10px 25px rgba(0, 0, 0, 0.1);
  width: 80%;
  max-width: 400px;
  text-align: center;
  margin: 0 auto;
  margin-top: 80px;
}

.sign-up h1 {
  margin-bottom: 24px;
  font-size: 28px;
  color: #333;
}

form {
  display: flex;
  flex-direction: column;
  gap: 15px;
}
```

```
form input[type="text"],
form input[type="password"] {
  padding: 12px;
  border: 1px solid #ccc;
  border-radius: 6px;
  font-size: 16px;
  transition: border 0.3s ease;
}

form input[type="text"]:focus,
form input[type="password"]:focus {
  border-color: #007bff;
  outline: none;
}

button.sign {
  background: #007bff;
  color: white;
  border: none;
  border-radius: 6px;
  padding: 12px;
  font-size: 16px;
  cursor: pointer;
  transition: background 0.3s ease;
}

button.sign:hover {
  background: #0056b3;
}

p {
  margin-top: 20px;
  font-size: 14px;
}

p a {
  color: #007bff;
  text-decoration: none;
}

p a:hover {
  text-decoration: underline;
}
header {
    width: 80%;
    margin: 0 auto;
    height: 80px;
    margin-top: 50px;
    display: flex;
    justify-content: space-between;
    align-items: center;
    border-radius: 5px;
}
```

```
header .header-logo .logo {
   font-size: 27px;
   font-weight: 900;
}

header .header-header {
   display: flex;
   align-items: center;
   gap: 25px;
}

header .logout-btn {
   background: linear-gradient(135deg, #1a1a1a, #000000);
   color: #fff;
   border-radius: 15px;
   padding: 10px 35px;
   border: none;
   cursor: pointer;
   text-align: center;
   text-decoration: none;
   margin-right: 15px;
}

header .username {
   padding: 0;
   margin: 0;
   font-weight: 700;
   font-size: 18px;
}

header .user-block {
   display: flex;
   flex-direction: column;
   align-items: center;
   justify-content: center;
}
.container {
   width: 80%;
   margin: 0 auto;
   height: 70vh;
   margin-top: 10vh;
   margin-bottom: 10vh;
   display: flex;
   flex-direction: row;
   align-items: center;
   justify-content: space-between;
   gap: 20px;
}

.container .sidebar {
   height: 80vh;
   width: 50%;
   display: flex;
   flex-direction: column;
   justify-content: center;
   gap: 35px;
}
```

```
.container .sidebar .image .side_image {
    width: 150px;
    height: 150px;
    box-shadow: 0 0 30px rgba(0, 0, 0, 0.1);
    border-radius: 25px;
}

.container .form {
    padding-top: 120px;
    padding-bottom: 120px;
    width: 40%;
    border: dashed 1px #000;
    border-radius: 25px;
}

.container .form form {
    width: 100%;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
}

.container .form form .name {
    margin-bottom: 30px;
}


.done_images {
    width: 80%;
    margin: 0 auto;
    margin-bottom: 50px;
}

.no-images {
    text-align: center;
    margin-top: 25px;
    margin-bottom: 25px;
}
.clear_images {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
    gap: 20px;
}

.clear_images .card {
  border-radius: 25px;
  padding: 10px;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  box-shadow: 5px 5px 15px rgba(0, 0, 0, 0.2);
}

.clear_images .card img {
  max-width: 300px;
  height: auto;
  border-radius: 6px;
  margin-bottom: 10px;
}
```

```
.actions {
   margin-bottom: 25px;
}

.download-btn {
  background-color: #6C63FF;
  color: #fff;
  padding: 10px 15px;
  border: none;
  cursor: pointer;
  text-decoration: none;
  border-radius: 10px;
  margin-right: 25px;
}

.delete-btn {
   background: linear-gradient(135deg, #ff4d4d, #ff0000);
   color: #fff;
   border-radius: 10px;
   padding: 10px 15px;
   border: none;
   cursor: pointer;
   text-decoration: none;
}

@media (max-width: 600px) {
  body {
   display: flex;
   flex-direction: column;
   gap: 20px;
  }
  .container {
   width: 100%;
   display: flex;
   flex-direction: column;
   gap: 20px;
  }

  .container .sidebar {
   width: 90%;
   margin: 0 auto;
   flex: 1;
   align-items: center;
  }
  .container .form {
   width: 90%;
   margin: 0 auto;
   flex: 1;
  }

  .done_images {
   width: 90%;
   margin: 0 auto;
   margin-top: 75px;
  }

  header {
   width: 90%;
   margin: 0 auto;
   justify-content: space-between;
   margin-top: 20px;
  }
```

```
  header .logout-btn {
    margin-right: 0px;
    padding: 7px 20px;
  }

  header .username {
    font-size: 15px;
  }

  header .header-logo .logo {
    font-size: 22px;
    font-weight: 900;
  }
}
```

————

```
from pathlib import Path
import os

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent


# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/4.2/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = "django-insecure-_$^w932&&qb4&go@(#eg#9f&$@91&!=)*^6ar*-_eq13m)+h_r"

# SECURITY WARNING: don't run with debug turned on in production!

STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')

DEBUG = True

ALLOWED_HOSTS = ['*']

LOGOUT_REDIRECT_URL = '/accounts/login/'
LOGIN_REDIRECT_URL = '/'



INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "web.apps.WebConfig"
]
```

```
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    'whitenoise.middleware.WhiteNoiseMiddleware',
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.common.CommonMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "django.middleware.clickjacking.XFrameOptionsMiddleware",
]

ROOT_URLCONF = "bg_remover.urls"

TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        'DIRS': [BASE_DIR / 'templates'],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    },
]

WSGI_APPLICATION = "bg_remover.wsgi.application"

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.getenv('POSTGRES_DB', 'bg_remover'),
        'USER': os.getenv('POSTGRES_USER', 'misha'),
        'PASSWORD': os.getenv('POSTGRES_PASSWORD', 'qwerty1234'),
        'HOST': os.getenv('POSTGRES_HOST', 'postgres'),
        'PORT': '5432',
    }
}

# Password validation
# https://docs.djangoproject.com/en/4.2/ref/settings/#auth-password-validators

AUTH_PASSWORD_VALIDATORS = [
    {
        "NAME": "django.contrib.auth.password_validation.UserAttributeSimilarityValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.MinimumLengthValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.CommonPasswordValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.NumericPasswordValidator",
    },
]
```

```
# Internationalization
# https://docs.djangoproject.com/en/4.2/topics/i18n/

LANGUAGE_CODE = "en-us"

TIME_ZONE = "UTC"

USE_I18N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/4.2/howto/static-files/


# Default primary key field type
# https://docs.djangoproject.com/en/4.2/ref/settings/#default-auto-field

DEFAULT_AUTO_FIELD = «django.db.models.BigAutoField"
```

————

```
FROM python:3.11-slim

ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

WORKDIR /app

# Устанавливаем зависимости
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        postgresql-client \
        iputils-ping \
        dnsutils \
        netcat-traditional \
        curl \
        net-tools && \
    rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["gunicorn", "bg_remover.wsgi:application", "--bind", "0.0.0.0:8000"]
```

```python
#!/usr/bin/env python
"""Django's command-line utility for administrative tasks."""
import os
import sys


def main():
    """Run administrative tasks."""
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "bg_remover.settings")
    try:
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(
            "Couldn't import Django. Are you sure it's installed and "
            "available on your PYTHONPATH environment variable? Did you "
            "forget to activate a virtual environment?"
        ) from exc
    execute_from_command_line(sys.argv)


if __name__ == "__main__":
    main()
```

————

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc
  namespace: tester
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

———

```yaml
apiVersion: v1
kind: Service
metadata:
  name: postgres
  namespace: tester
spec:
  selector:
    app: postgres
  ports:
  - port: 5432
    targetPort: 5432
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
  namespace: tester
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:15
        env:
        - name: POSTGRES_DB
          value: bg_remover
        - name: POSTGRES_USER
          value: misha
        - name: POSTGRES_PASSWORD
          value: qwerty1234
        volumeMounts:
        - name: pgdata
          mountPath: /var/lib/postgresql/data
        ports:
        - containerPort: 5432
      volumes:
      - name: pgdata
        persistentVolumeClaim:
          claimName: postgres-pvc

————

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: django-media-pvc
  namespace: tester
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: django-static-pvc
  namespace: tester
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: django
  namespace: tester
spec:
  replicas: 1
  selector:
    matchLabels:
      app: django
  template:
    metadata:
      labels:
        app: django
    spec:
      volumes:
        - name: media
          persistentVolumeClaim:
            claimName: django-media-pvc
        - name: staticfiles
          emptyDir: {}

      initContainers:
        - name: migrate-and-setup
          image: misha525/bg-remover:latest
          workingDir: /app
          command:
          - sh
          - -c
          - |
            set -e
            echo "Waiting for DB (initial sleep)..."
            sleep 5
            until pg_isready -h "$POSTGRES_HOST" -p 5432; do
              echo "Waiting for DB..."
              sleep 2
            done
            echo "Running migrations..."
            python manage.py migrate --noinput
            python manage.py collectstatic --noinput
          env:
            - name: POSTGRES_DB
              value: bg_remover
            - name: POSTGRES_USER
              value: misha
            - name: POSTGRES_PASSWORD
              value: qwerty1234
            - name: POSTGRES_HOST
              value: postgres
          volumeMounts:
            - name: staticfiles
              mountPath: /app/staticfiles
```

```
containers:
    - name: django
      image: misha525/bg-remover:latest
      ports:
        - containerPort: 8000
      env:
        - name: POSTGRES_DB
          value: bg_remover
        - name: POSTGRES_USER
          value: misha
        - name: POSTGRES_PASSWORD
          value: qwerty1234
        - name: POSTGRES_HOST
          value: postgres
      volumeMounts:
        - name: media
          mountPath: /app/media
        - name: staticfiles
          mountPath: /app/staticfiles


---
apiVersion: v1
kind: Service
metadata:
  name: django
  namespace: tester
spec:
  selector:
    app: django
  ports:
    - port: 80
      targetPort: 8000
      protocol: TCP
  type: ClusterIP
```