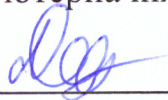


Міністерство освіти і науки України
Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп'ютерної інженерії та кібербезпеки

ПОЯСНЮВАЛЬНА ЗАПИСКА

до випускної кваліфікаційної роботи магістра
на тему: «Методи та технології побудови та
розгортання високодоступного вебстеку на основі
контейнерів Docker»

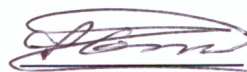
Виконав студент 2-го курсу групи КІм-23-1
спеціальності 123 «Комп'ютерна інженерія»

 О.С. Лузік

Керівник доцент кафедри комп'ютерної інженерії та
кібербезпеки, кандидат технічних наук

 О.О. Шелуха

Рецензент завідувач кафедри комп'ютерної інженерії та
кібербезпеки, кандидат технічних наук, доцент

 А.А. Єфіменко

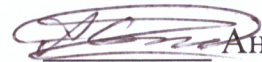
Житомир – 2024

Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп'ютерної інженерії та кібербезпеки
Спеціальність 123 «Комп'ютерна інженерія»
Освітня програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ

Завідувач

кафедри комп'ютерної
інженерії та кібербезпеки

 Андрій Єфіменко

17 вересня 2024 р.

ЗАВДАННЯ

на випускню кваліфікаційну роботу магістра

Студент: Лузік Олег Сергійович

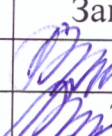
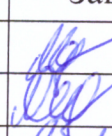
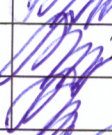
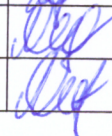
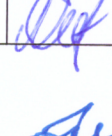
Тема роботи: **«Методи та технології побудови та розгортання
високодоступного вебстеку на основі контейнерів Docker»**

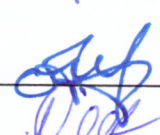
затверджена Наказом університету від 17 вересня 2024 р. № 478/с

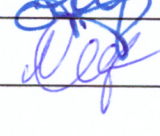
Термін здачі студентом закінченої роботи 16 грудня 2024 р.

Вихідні дані роботи (зазначається предмет і об'єкт дослідження): У процесі виконання роботи використовується система побудови та управління контейнеризованими вебстеками з використанням Docker, NGINX, PostgreSQL, Redis, інтегрованих у середовище з високою доступністю за допомогою балансувальника навантаження. Реалізація системи передбачає створення контейнеризованого середовища, яке відповідає вимогам високої доступності, масштабованості та відмовостійкості. Використання цих методів та технологій забезпечує стабільну роботу вебстеку навіть за умов значного навантаження чи часткових збоїв окремих компонентів.

Консультанти з випускної кваліфікаційної роботи із зазначенням розділів, що їх стосуються

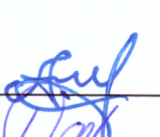
Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
1	В.В. Воротніков	 17.09.2024	 20.10.2024
2	В.В. Воротніков	 21.10.2024	 18.11.2024
3	В.В. Воротніков	 19.11.2024	 20.12.2024

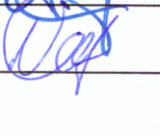
Студент  О.О. Шелуха

Керівник  О.С. Лузік

Календарний план

№ з/п	Назви етапів випускної кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка задачі.	17.09.2024 – 24.09.2024	Виконано
2	Пошук, огляд та аналіз аналогічних розробок.	25.09.2024 – 08.10.2024	Виконано
3	Формулювання технічного завдання.	09.10.2024 – 15.10.2024	Виконано
4	Опрацювання літературних джерел.	16.10.2024 – 29.10.2024	Виконано
5	Проектування структури системи	30.10.2024 – 12.11.2024	Виконано
6	Налаштування системи	13.11.2024 – 03.12.2024	Виконано
7	Тестування	04.12.2024 – 17.12.2024	Виконано
8	Захист	27.12.2024	Виконано

Студент  О.О. Шелуха

Керівник  О.С. Лузік

АНОТАЦІЯ

Випускна кваліфікаційна робота магістра на тему: «Методи та технології побудови та розгортання високодоступного вебстеку на основі контейнерів Docker». Пояснювальна записка до випускної роботи містить 81 сторінку, 4 таблиці, 12 рисунків, 2 додатки, 32 джерела.

Метою роботи є дослідження сучасних методів і технологій побудови високодоступних вебстеків із використанням контейнерів Docker та впровадження розробленої системи у серверну інфраструктуру для забезпечення стійкості до збоїв і підвищення масштабованості роботи вебдодатків.

У роботі визначено основні завдання: проведено аналіз сучасних підходів до забезпечення високої доступності вебдодатків. Досліджено існуючі технології контейнеризації та їх роль у побудові масштабованих і відмовостійких систем. Спроековано архітектуру високодоступного вебстеку, орієнтовану на вирішення проблем масштабованості та відмовостійкості. Реалізовано розроблену систему в контейнеризованому середовищі, використовуючи Docker і інструменти оркестрації. Проведено тестування продуктивності, функціонування системи та відновлення після відмов.

У першому розділі виконано аналіз основних підходів до забезпечення високої доступності вебдодатків, розглянуто особливості контейнеризації та її переваги у порівнянні з традиційними методами. У другому розділі запропоновано архітектуру високодоступного вебстеку, яка включає компоненти для балансування навантаження, реплікації сервісів, резервного зберігання даних та механізми відновлення після збоїв. У третьому розділі описано процес розгортання системи в контейнеризованому середовищі Docker з використанням оркестраційних інструментів (Docker Swarm, Kubernetes) та проведено її тестування.

У роботі розроблено та впроваджено систему побудови високодоступного вебстеку на основі сучасних технологій контейнеризації, зокрема Docker, яка забезпечує відмовостійкість архітектури вебдодатків. Впроваджені механізми

балансування навантаження, реплікації та резервування компонентів гарантують стабільну роботу системи навіть за умов значного навантаження або часткових збоїв окремих її частин.

Практичне значення отриманих результатів полягає в адаптації та інтеграції існуючих методів і технологій контейнеризації для створення вебстеку, орієнтованого на високу доступність. Запропонована система може бути використана у реальних проєктах для підтримки безперебійної роботи вебдодатків у сферах електронної комерції, фінансових технологій та хмарних обчислень.

Об'єктом дослідження є процес налаштування та управління функціонуванням інфраструктурою вебдодатків у високонавантажених середовищах.

Предмет дослідження: методи та технології забезпечення високої доступності вебдодатків.

Результати роботи можуть бути застосовані для впровадження високодоступних систем у галузях, де критично важлива безперервність надання послуг, таких як електронна комерція, фінансові технології та хмарні сервіси.

КЛЮЧОВІ СЛОВА: ВИСОКА ДОСТУПНІСТЬ, КОНТЕЙНЕРИЗАЦІЯ, DOCKER, KUBERNETES, ОРКЕСТРАЦІЯ, ВЕБСТЕК, БАЛАНСУВАННЯ НАВАНТАЖЕННЯ, ВІДМОВОСТІЙКІСТЬ.

ANNOTATION

Master's final qualification work on the topic: «Methods and technologies for building and deploying a highly available web stack based on Docker containers». The explanatory note to the final work contains 81 pages, 12 drawings, 3 tables, 32 sources.

The purpose of the work is to study modern methods and technologies for building highly available web stacks using Docker containers and implementing the developed system into the server infrastructure to ensure fault tolerance and increase the scalability of web applications.

The work identifies the main tasks: an analysis of modern approaches to ensuring high availability of web applications was conducted. Existing containerization technologies and their role in building scalable and fault-tolerant systems were studied. The architecture of a highly available web stack was designed, focused on solving scalability and fault-tolerance problems. The developed system was implemented in a containerized environment using Docker and orchestration tools. Performance testing, system operation, and recovery after failures were conducted.

The first section analyzes the main approaches to ensuring high availability of web applications, considers the features of containerization and its advantages compared to traditional methods. The second section proposes an architecture of a highly available web stack, which includes components for load balancing, service replication, data backup storage and mechanisms for recovering from failures. The third section describes the process of deploying the system in a containerized Docker environment using orchestration tools (Docker Swarm, Kubernetes) and its testing.

The work develops and implements a system for building a highly available web stack based on modern containerization technologies, in particular Docker, which provides fault tolerance for the web application architecture. The implemented load balancing, replication and component redundancy mechanisms guarantee stable operation of the system even under conditions of significant load or partial failures of its individual parts.

The practical significance of the results obtained lies in the adaptation and integration of existing containerization methods and technologies to create a web stack focused on high availability. The proposed system can be used in real projects to support the uninterrupted operation of web applications in the fields of e-commerce, financial technologies and cloud computing.

The object of the study is the process of configuring and managing the functioning of the web application infrastructure in high-load environments.

Subject of the study: methods and technologies for ensuring high availability of web applications.

The results of the work can be applied to the implementation of highly available systems in industries where the continuity of service provision is critically important, such as e-commerce, financial technologies and cloud services.

KEYWORDS: HIGH AVAILABILITY, CONTAINERIZATION, DOCKER, KUBERNETES, ORCHESTRATION, WEB - STACK, LOAD BALANCING, FAULT RESILIENCE.

ЗМІСТ

ВСТУП.....	11
РОЗДІЛ 1 ТЕОРЕТИЧНІ АСПЕКТИ ПОБУДОВИ ТА ФУНКЦІОНУВАННЯ ЦЕНТРАЛІЗОВАНИХ СИСТЕМ АНАЛІЗУ ЖУРНАЛІВ	13
1.1 Аналіз сучасних підходів до забезпечення високої доступності у вебдодатках.....	13
1.2 Аналіз застосування контейнеризації при розгортанні вебстеків	14
1.2.1 Архітектура контейнерів Docker	16
1.2.2 Особливості масштабування контейнерів	19
1.3 Аналіз сучасних технологій для побудови високодоступних систем.....	21
1.4 Огляд підходів до кластеризації та оркестрації контейнерів (Docker Swarm, Kubernetes).....	22
1.5 Постановка завдання дослідження.....	24
РОЗДІЛ 2. ПРОЄКТУВАННЯ ВИСОКОДОСТУПНОГО ВЕБСТЕКУ НА ОСНОВІ DOCKER	26
2.1. Визначення основних компонентів вебстеку.....	26
2.2. Проєктування архітектури вебстеку для високої доступності.....	27
2.2.1. Балансування навантаження	29
2.2.2. Реплікація сервісів та зберігання даних.....	31
2.3. Вибір програмних рішень для розгортання.	33
2.4. Вибір обладнання та програмного забезпечення	36
РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБСТЕКУ	41
3.1. Розробка підсистеми резервування та відмовостійкості	41
3.1.1. Серверне обладнання та мережеві компоненти	44
3.1.2. Інструменти для роботи з контейнерами (Docker Engine, Docker Compose).....	47
3.2. Розгортання вебстеку на основі Docker.....	50
3.2.1. Налаштування компонентів вебстеку (веб сервер, база даних, кеш)	58
3.2.2. Інтеграція Load Balancer для забезпечення високої доступності.....	60
3.3. Тестування продуктивності та відмовостійкості вебстеку.....	62

3.3.1. Тестування під навантаженням.....	65
3.3.2. Перевірка відновлення після відмови компонентів.....	67
3.3.3. Аналіз результатів тестування	68
ВИСНОВКИ.....	70
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ	72
ДОДАТКИ.....	75

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ОС – Операційна система
ВМ – Віртуальна машина
CI – Continuous Integration
CD – Continuous Delivery/Deployment
gRPC – Google Remote Procedure Call
CRI – Container Runtime Interface
CNI – Container Networking Interface
REST – Representational State Transfer
CRUD – Create, Read, Update Delete
JSON – JavaScript Object Notation
YAML – YAML Ain't Markup Language
OCI – Open Container Initiative
CSI – Container Storage Interface
GCP – Google Cloud Platform
GKE – Google Kubernetes Engine
DNS – Domain Name System

ВСТУП

Актуальність дослідження. У сучасному світі забезпечення високої доступності вебдодатків є одним із ключових завдань для багатьох організацій, що надають онлайн - послуги. Висока доступність означає мінімізацію часу простою системи, забезпечення стійкості до збоїв та здатність системи обробляти значні обсяги трафіку. Неналежна організація інфраструктури вебдодатків може призвести до частих збоїв, втрачених даних та зниження якості обслуговування користувачів. Контейнеризація стає важливим інструментом у створенні масштабованих і відмовостійких веб систем. Контейнери дозволяють ізолювати додатки, забезпечуючи гнучкість у розгортанні, легкість у масштабуванні та сумісність між середовищами розробки й експлуатації. Однією з найпоширеніших платформ для контейнеризації є Docker, який пропонує прості у використанні засоби для розгортання та управління додатками в контейнерах. З розвитком технологій контейнеризації актуальним стає дослідження методів і засобів забезпечення високої доступності систем, зокрема через використання оркестраційних платформ, таких як Kubernetes або Docker Swarm. Ефективне управління контейнеризованими середовищами вимагає налаштування балансування навантаження, реплікації сервісів та резервування даних для збереження функціональності навіть у випадку збоїв. Актуальність роботи полягає в тому, що сучасні вимоги до вебдодатків зумовлюють необхідність побудови надійних інфраструктур, здатних забезпечити безперебійну роботу у високонавантажених середовищах. Високодоступні вебстеки, створені на основі контейнерів Docker, є перспективним рішенням для досягнення цієї мети.

Метою роботи є дослідження сучасних методів і технологій побудови високодоступних вебстеків із використанням контейнерів Docker та впровадження розробленої системи у серверну інфраструктуру для забезпечення стійкості до збоїв і підвищення масштабованості роботи вебдодатків.

Для досягнення поставленої мети було розроблено та виконано такі завдання:

1. Провести аналіз сучасних методів і технологій забезпечення високої доступності вебдодатків.

2. Спроекувати архітектуру високодоступного вебстеку з використанням Docker.

3. Реалізувати вебстек у контейнеризованому середовищі, забезпечивши відмовостійкість.

4. Провести тестування продуктивності та аналіз функціонування розробленої системи.

Об'єктом дослідження є процес налаштування та управління функціонуванням інфраструктурою вебдодатків у високонавантажених середовищах.

Предметом дослідження є методи та технології забезпечення високої доступності вебдодатків за допомогою контейнеризації.

У роботі розроблено та впроваджено систему побудови високодоступного вебстеку на основі сучасних технологій контейнеризації, зокрема Docker, яка забезпечує відмовостійкість архітектури вебдодатків. Впроваджені механізми балансування навантаження, реплікації та резервування компонентів гарантують стабільну роботу системи навіть за умов значного навантаження або часткових збоїв окремих її частин.

Практичне значення отриманих результатів полягає в адаптації та інтеграції існуючих методів і технологій контейнеризації для створення вебстеку, орієнтованого на високу доступність. Запропонована система може бути використана у реальних проєктах для підтримки безперебійної роботи вебдодатків у сферах електронної комерції, фінансових технологій та хмарних обчислень.

РОЗДІЛ 1 ТЕОРЕТИЧНІ АСПЕКТИ ПОБУДОВИ ТА ФУНКЦІОНУВАННЯ ЦЕНТРАЛІЗОВАНИХ СИСТЕМ АНАЛІЗУ ЖУРНАЛІВ

1.1 Аналіз сучасних підходів до забезпечення високої доступності у вебдодатках

Аналізуючи сучасні підходи до забезпечення доступності у вебдодатках, можна сказати, що висока доступність (High Availability, HA) — це концепція проектування та експлуатації інформаційних систем, яка забезпечує їхню максимальну доступність для користувачів навіть за наявності збоїв або перевантажень. Забезпечення високої доступності є критично важливим для сучасних вебдодатків, оскільки вони повинні гарантувати стабільну роботу для користувачів у будь-який час.

Вебдодатки мають бути спроектовані так, щоб автоматично відновлюватися після відмов компонентів (серверів, мережі, баз даних тощо). Для забезпечення безперервності роботи застосовуються резервні копії системних компонентів (дублювання серверів у кластері, резервні копії даних у розподілених системах зберігання). Постійне відстеження стану системи та автоматичне реагування на потенційні проблеми (наприклад, перезапуск контейнера при збої). Розподіл вхідного трафіку між кількома серверами для уникнення перевантаження окремих компонентів системи. Динамічне додавання або зменшення кількості ресурсів (серверів, контейнерів) у відповідь на зміну навантаження [1, с. 29].

Технології контейнеризації (наприклад, Docker) та оркестрації (Kubernetes) значно підвищують можливості щодо забезпечення високої доступності. Контейнери забезпечують можливість швидкого розгортання нових інстанцій веб додатка. Завдяки цьому можна оперативно реагувати на зміни в трафіку. Оркестратори, такі як Kubernetes, автоматично перевіряють стан контейнерів і перезапускають їх у разі збоїв. Веб додаток у контейнерах легко розподілити між кількома вузлами (серверними машинами), що знижує ризик відмови всієї системи через збій одного з компонентів [4, с. 112]. Кожен контейнер

ізолюваний, що зменшує ризик конфліктів між компонентами та підвищує стабільність роботи системи.

Ключові аспекти для забезпечення НА у веб додатках на основі контейнерів [21, с. 198]:

1. Архітектура без єдиної точки відмови (Single Point of Failure): розподілені бази даних (наприклад, MongoDB, Cassandra), використання кластерів та реплікацій.

2. Мережеві стратегії: використання механізмів балансування навантаження (Nginx, HAProxy) та резервних каналів зв'язку.

3. Автоматизоване управління: автоматизація через інструменти CI/CD (Jenkins, GitLab CI) для швидкого реагування на зміни в системі.

4. Тестування на відмовостійкість: використання «хаос-інженерії» для перевірки системи на стійкість до несподіваних збоїв. Наприклад, Chaos Monkey від Netflix.

Таким чином, забезпечення високої доступності вебдодатків на основі контейнерів базується на комплексному підході, який включає в себе резервування, масштабування, моніторинг і автоматизацію. Контейнеризація не тільки полегшує реалізацію цих аспектів, а й сприяє зниженню витрат та підвищенню ефективності управління ресурсами.

1.2 Аналіз застосування контейнеризації при розгортанні вебстеків

Розглядаючи контейнеризацію як сучасний підхід до розгортання веб – стеків варто знати, що контейнеризація — це сучасний метод розгортання програмного забезпечення, який дозволяє ізолювати додатки та їхні залежності у спеціалізовані середовища, що називаються контейнерами. Контейнери забезпечують легке розгортання, масштабування, мобільність та управління веб стеками. Розглянемо особливості контейнеризації [29, с. 86]:

1. Ізоляція: кожен контейнер має власну операційну систему, бібліотеки та залежності, що дозволяє уникнути конфліктів між компонентами.

2. *Легка перенесеність*: контейнери є незалежними від базової інфраструктури, що дає змогу розгортати додатки однаково на локальних машинах, серверах чи у хмарних середовищах.

3. *Оптимізація ресурсів*: контейнери використовують ресурси ефективніше, ніж віртуальні машини, оскільки вони працюють безпосередньо на ядрі операційної системи хосту.

4. *Швидкість запуску*: контейнери запускаються значно швидше, ніж віртуальні машини, завдяки меншій накладній інфраструктурі.

Веб стек — це набір технологій для створення та роботи вебдодатків (сервери, бази даних, вебфреймворки). Контейнеризація допомагає ефективно розгортати та керувати всіма компонентами веб стека. Технології, такі як React, Angular або Vue.js, можуть бути розгорнуті у контейнерах, що забезпечує швидку доставку та тестування оновлень [20, с. 65]. Серверні фреймворки (Node.js, Django, Spring) запускаються у контейнерах, що дозволяє спрощено управляти середовищами, їх конфігурацією та залежностями. Розгортання баз даних (наприклад, MySQL, PostgreSQL, MongoDB) у контейнерах забезпечує масштабованість і можливість швидкого резервування даних. З використанням оркестраторів, таких як Kubernetes або Docker Swarm, можна легко розподілити веб стек між кількома вузлами, підвищуючи доступність і продуктивність. Основні інструменти контейнеризації [27, с. 108]:

- Docker - найпоширеніша платформа для створення, управління та розгортання контейнерів. Забезпечує легкість у використанні та підтримку більшості сучасних технологій.

- Kubernetes - інструмент оркестрації контейнерів для масштабування, моніторингу та забезпечення відмовостійкості.

- Docker Compose - використовується для управління мультиконтейнерними додатками, наприклад, для запуску одночасно веб сервера, бази даних та кешу.

- Podman - альтернатива Docker з підвищеним акцентом на безпеку та роботу без демонів.

Переваги контейнеризації для розгортання веб – стеків [14, с. 203]:

- Використання контейнерів дозволяє розробникам швидко створювати однакові середовища на різних етапах розробки та тестування.
- Контейнери легко інтегруються з CI/CD процесами, що забезпечує автоматичне тестування, побудову і розгортання.
- Завдяки контейнеризації можна легко додавати або видаляти компоненти веб стека відповідно до змін навантаження.
- Контейнери дозволяють тестувати різні версії технологій у межах одного проєкту, що пришвидшує інтеграцію нових рішень.
- Виклики контейнеризації [14, с. 218]:
- Для великих систем потрібні спеціалізовані інструменти оркестрації.
- Необхідно забезпечити захист контейнерів від небажаного доступу, наприклад, через неправильні конфігурації.
- Розробники повинні освоїти нові інструменти та підходи до управління контейнерами.

Одже, контейнеризація є невід’ємною частиною сучасної архітектури вебстеків, забезпечуючи їхню гнучкість, продуктивність і доступність. Завдяки контейнерам веб додатки можуть бути розгорнуті з меншими витратами часу і ресурсів, що робить їх оптимальним вибором для динамічних і масштабованих систем.

1.2.1 Архітектура контейнерів Docker

Вивчаючи архітектуру контейнерів Docker, маємо зазначити, що Docker — це платформа для розробки, доставки та запуску додатків у контейнерах. Вона дозволяє ізолювати додатки разом з усіма їхніми залежностями, забезпечуючи легку перенесеність і узгодженість роботи в різних середовищах. Основними компонентами архітектури Docker є [17, с. 128]:

- Docker Engine, так би мовити серце Docker, що забезпечує всі основні функції платформи. Складається з кількох компонентів:

- Docker Daemon (dockerd): відповідає за управління контейнерами, образами та мережами. Працює у фоновому режимі на хості.
- Docker CLI (Command Line Interface): інтерфейс командного рядка, за допомогою якого користувач взаємодіє з Docker Daemon.
- REST API: дає змогу інтегрувати Docker з іншими інструментами через HTTP-запити.

2. *Образи Docker (Docker Images)* - незмінні шаблони, які містять всі необхідні для роботи програми компоненти: код, бібліотеки, налаштування. Образи створюються за допомогою Dockerfile — текстового файлу, що описує інструкції для побудови образу.

3. *Контейнери Docker*. Легковагові, автономні середовища, які запускаються на основі образів. Контейнер — це працюючий екземпляр образу. Контейнери мають:

- Ізольоване файлове середовище.
- Власний мережевий стек.
- Виділені обчислювальні ресурси.

4. *Реєстри Docker (Docker Registry)*

Сховища для образів Docker:

- Docker Hub: Публічний реєстр із великою кількістю готових до використання образів.
- Приватні реєстри: Можна налаштувати для зберігання корпоративних образів.

5. *Мережеві компоненти Docker*

Docker забезпечує мережеву взаємодію між контейнерами та зовнішнім середовищем:

- Bridge за замовчуванням використовується для зв'язку між контейнерами на одному хості.
- Host - контейнер спільно використовує мережу хосту.
- Overlay - забезпечує зв'язок між контейнерами на різних хостах.

- Custom - налаштування мережі для специфічних потреб.

6. Том (*Docker Volume*)

Спосіб зберігання даних, що дозволяє зберігати інформацію контейнера поза його життєвим циклом, що корисно для створення стійких до збоїв додатків.

Користувач створює *Dockerfile*, що містить інструкції для побудови образу (наприклад, базовий образ, команди встановлення залежностей, копіювання файлів тощо). Образ завантажується з реєстру Docker (або створюється локально) і запускається у вигляді контейнера. Docker Daemon забезпечує моніторинг і управління контейнерами, включаючи запуск, зупинку, видалення або оновлення. Контейнери отримують доступ до налаштованих мереж і томів для взаємодії з іншими контейнерами або збереження даних. Образи забезпечують узгодженість роботи додатків на різних платформах [2, с. 51]. Контейнери запускаються за кілька секунд завдяки легкій архітектурі. Забезпечується захист додатків один від одного, що запобігає конфліктам залежностей. Інтеграція з оркестраторами, такими як Kubernetes, дозволяє легко масштабувати контейнери. Контейнери спільно використовують ядро операційної системи, що робить їх менш ресурсомісткими порівняно з віртуальними машинами.

Контейнери використовують ядро хостової ОС, що може призводити до ризиків безпеки. Для великих систем потрібні додаткові інструменти, такі як Kubernetes або Docker Swarm. Контейнери не можуть працювати на різних ядрах ОС (наприклад, Linux-контейнери не працюють на Windows без емуляції) [2, с. 52].

Одже, архітектура Docker побудована навколо простоти, ефективності та ізоляції, що робить її ідеальним вибором для сучасних вебдодатків. Вона дозволяє розробникам легко створювати, розгортати й підтримувати додатки в контейнерах, що забезпечує високу продуктивність і швидкість впровадження нових функцій.

1.2.2 Особливості масштабування контейнерів

Розглядаючи особливості масштабування контейнерів, ми розуміємо, що масштабування контейнерів — це процес збільшення або зменшення кількості контейнерів, що виконують певний додаток або його компонент, залежно від потреб системи. Дана властивість робить контейнеризацію ідеальним вибором для сучасних масштабованих додатків, таких як веб-сервіси, мікро-сервіси, або сервери баз даних. Якщо звернути увагу на типи масштабування контейнерів, можна виділити наступні [13, с. 141]:

- Горизонтальне масштабування (Horizontal Scaling). Додаються нові контейнери, які виконують копії додатка. Такий підхід збільшує кількість інстанцій, що працюють паралельно. Перевагами є легка інтеграція з балансувальниками навантаження, підходить для мікро-сервісної архітектури, для вебдодатків із змінним навантаженням.

- Вертикальне масштабування (Vertical Scaling). Збільшуються ресурси (процесор, пам'ять) контейнера без додавання нових екземплярів. Перевагами є простота в налаштуванні, масштабування підходить для додатків, які важко розділити на кілька екземплярів, проте залежить від можливостей хостової системи.

Інструменти для масштабування контейнерів [19, с. 24]:

1. *Kubernetes* забезпечує автоматичне горизонтальне масштабування на основі метрик (CPU, пам'ять, користувацькі метрики). Використовує Horizontal Pod Autoscaler (HPA) для динамічного управління кількістю подів. Підтримує Vertical Pod Autoscaler (VPA) для налаштування ресурсів контейнерів.

2. *Docker Swarm* вбудований механізм оркестрації для Docker. Забезпечує ручне та автоматичне масштабування контейнерів за допомогою команд CLI.

3. *AWS ECS/Fargate* платформи для контейнеризації в хмарі з вбудованими механізмами масштабування. Підтримує автоматизацію на основі правил та тригерів (наприклад, збільшення навантаження на CPU).

4. *Azure Kubernetes Service (AKS) та Google Kubernetes Engine (GKE)* - інтегровані сервіси для управління масштабуванням контейнерів у хмарних середовищах.

Хмарні провайдери пропонують сервіси для динамічного масштабування контейнерів: Auto-scaling додає або видаляє ресурси залежно від потреби в реальному часі. Serverless-контейнери наприклад, AWS Fargate або Google Cloud Run, де ресурси автоматично масштабуються без прямого втручання.

Розглянемо особливості масштабування контейнерів у реальних сценаріях [24, с. 47]:

1. *Балансування навантаження.* Для ефективного горизонтального масштабування використовується балансувальник навантаження (наприклад, NGINX, HAProxy, або вбудовані сервіси в Kubernetes).

2. *Моніторинг та метрики.* Моніторинг ресурсоемності контейнерів забезпечує ефективне масштабування.

Інструменти: Prometheus, Grafana, CloudWatch.

3. *Обробка стану додатків.* Масштабування складніше для додатків зі станом (stateful), таких як бази даних. Для таких сценаріїв застосовуються спеціалізовані рішення, наприклад, StatefulSets у Kubernetes.

4. *Економія ресурсів.* Автоматичне масштабування контейнерів допомагає зменшувати використання ресурсів у неробочий час, що важливо для хмарних середовищ із погодинною оплатою [32, с. 14].

Варто також зауважити, що масштабування у реальному часі може створювати затримки під час створення нових контейнерів. Додатки зі станом складніше масштабувати через необхідність синхронізації даних між контейнерами. Фізична або віртуальна інфраструктура може бути перевантажена при значному збільшенні кількості контейнерів. Ручне налаштування політик масштабування потребує глибоких знань архітектури.

Проте є й переваги масштабування контейнерів, таке масштабування забезпечує динамічну адаптацію до змін навантаження. Підвищує доступність і

продуктивність додатків. Економить ресурси за рахунок автоматичного зменшення інстанцій у періоди низького навантаження.

Одже, масштабування контейнерів є ключовою особливістю сучасних розподілених систем, яка дозволяє ефективно використовувати ресурси та гарантувати високу доступність додатків. Використання автоматизованих інструментів та хмарних сервісів спрощує управління масштабуванням і робить систему більш гнучкою до змін у навантаженні.

1.3 Аналіз сучасних технологій для побудови високодоступних систем

Аналіз сучасних технологій для побудови високодоступних систем базується на поєднанні концепцій, інструментів та підходів, які дозволяють досягти максимальної надійності, продуктивності та відмовостійкості. У сучасних умовах ключову роль відіграють технології оркестрації, контейнеризації, балансування навантаження, хмарні сервіси, а також системи моніторингу та автоматизації. Однією з найпопулярніших технологій для побудови високодоступних систем є Kubernetes — система оркестрації контейнерів. Kubernetes забезпечує автоматичний перезапуск контейнерів, розподіл навантаження між подами, самовідновлення та масштабування. Його архітектура дозволяє працювати з кластерами, розподіленими між різними дата-центрами, що мінімізує ризики відмови через фізичні або програмні збої.

Системи балансування навантаження, такі як NGINX, HAProxy, AWS Elastic Load Balancer, забезпечують рівномірний розподіл запитів між серверами або контейнерами. Вони дозволяють підтримувати роботу додатків навіть у разі виходу з ладу окремих вузлів системи. Балансувальники часто інтегруються з хмарними платформами, що надає додаткові можливості для автоматизації. Хмарні сервіси, такі як AWS, Google Cloud Platform і Microsoft Azure, пропонують вбудовані механізми для забезпечення високої доступності. Вони надають можливості розгортання додатків у кількох регіонах одночасно, використання автоматичного масштабування та резервного копіювання. Такі сервіси, як AWS Auto Scaling і Google Cloud Autoscaler, дозволяють додавати або

видаляти ресурси залежно від навантаження, зменшуючи витрати та підвищуючи ефективність [8, с. 42].

Моніторинг і спостереження відіграють ключову роль у забезпеченні високої доступності. Інструменти, такі як Prometheus, Grafana, Zabbix, та сервіси типу AWS CloudWatch, дозволяють відстежувати стан системи в реальному часі, генерувати попередження про потенційні проблеми і автоматично реагувати на них, що забезпечує швидке виявлення та усунення несправностей [6, с. 109].

Для роботи з даними в умовах високої доступності використовуються розподілені бази даних, такі як Cassandra, MongoDB і PostgreSQL з механізмами реплікації. Вони дозволяють зберігати копії даних на різних вузлах і автоматично перемикаються на резервний вузол у разі збою основного. Сучасні технології автоматизації, такі як Terraform, Ansible, Puppet та Chef, дозволяють створювати інфраструктуру як код, забезпечуючи швидке відновлення систем після збоїв і забезпечення узгодженості середовища. Автоматизація спрощує управління великими системами, знижуючи ризик помилок через людський фактор. Однією з передових концепцій є застосування хаос-інженерії, яка спрямована на перевірку відмовостійкості системи в реальних умовах. Інструменти, такі як Chaos Monkey, дозволяють симулювати збої в роботі окремих компонентів, що допомагає виявляти слабкі місця в архітектурі [11, с. 79].

Таким чином, побудова високодоступних систем ґрунтується на поєднанні технологій контейнеризації, хмарних рішень, автоматизації, моніторингу та резервування даних. Інтеграція цих компонентів дозволяє створювати надійні, продуктивні й економічно ефективні системи, які відповідають вимогам сучасних цифрових рішень.

1.4 Огляд підходів до кластеризації та оркестрації контейнерів (Docker Swarm, Kubernetes)

Огляд підходів до кластеризації та оркестрації контейнерів охоплює два основні інструменти, які використовуються для управління контейнеризованими додатками: Docker Swarm та Kubernetes, дані технології мають спільну мету –

забезпечити автоматизацію, масштабованість та відмовостійкість, але вони суттєво різняться підходами до реалізації.

Docker Swarm є вбудованим механізмом оркестрації, інтегрованим у платформу Docker. Основною перевагою Docker Swarm є його простота та тісна інтеграція з екосистемою Docker. Він дозволяє створювати кластери, що складаються з кількох вузлів, і розподіляти між ними контейнери. Swarm підтримує автоматичне масштабування контейнерів, балансування навантаження, а також забезпечує резервування шляхом дублювання сервісів на різних вузлах [9, с. 54]. Його налаштування просте, що робить його зручним для невеликих проєктів або систем із помірними вимогами до продуктивності. Однак Swarm має обмежений функціонал порівняно з іншими оркестраторами, що може стати перешкодою для великих або складних проєктів.

Kubernetes є більш потужним та універсальним рішенням для оркестрації контейнерів, яке стало стандартом де-факто у цій галузі. Kubernetes забезпечує повноцінний контроль над усіма аспектами управління контейнеризованими додатками, включаючи автоматичне масштабування, самовідновлення, балансування навантаження, управління мережею та інтеграцію з розподіленими сховищами. Основною перевагою Kubernetes є його модульна архітектура, яка дозволяє налаштовувати кластери відповідно до потреб конкретного проєкту. Вбудовані механізми для роботи з мікро - сервісами та підтримка широкого спектра плагінів роблять його підходящим для великих і складних систем [3, с. 207].

Docker Swarm і Kubernetes також різняться у підходах до налаштування та управління. Swarm використовує декларативний підхід із простими командами CLI, що дозволяє швидко створити кластер і почати роботу. Kubernetes застосовує складнішу декларативну модель із використанням YAML-файлів для опису конфігурації додатків, що може вимагати більше часу для освоєння, але надає значно більшу гнучкість і контроль. Kubernetes також має розвинену

екосистему, включаючи такі сервіси, як Helm для управління пакетами, Prometheus для моніторингу та Istio для мережових політик і безпеки [3, с. 209].

Одже, обидві технології мають свої сильні та слабкі сторони, і вибір між ними залежить від вимог проєкту. Docker Swarm підходить для невеликих або середніх додатків, які потребують швидкого розгортання без складного налаштування. Kubernetes, натомість, краще справляється з великими, розподіленими системами, які потребують гнучкості, масштабованості та розширених можливостей управління.

1.5 Постановка завдання дослідження

На основі проведеного аналізу проблематики побудови високодоступних систем із використанням контейнеризації, було визначено актуальність дослідження цієї теми, що обумовлена зростаючими вимогами до надійності, масштабованості та ефективності сучасних інформаційних систем. Метою роботи є розробка підходу до створення високодоступної системи на основі сучасних технологій контейнеризації та оркестрації.

Для досягнення поставленої мети було проведено аналіз об'єкта дослідження та сформульовано наступні завдання:

1. Провести аналіз сучасних методів, технологій і засобів вирішення проблем побудови високодоступних систем, включаючи аналіз контейнеризації та оркестрації.
2. Розробити проєкт архітектури, модель або макет системи, яка забезпечує високу доступність за рахунок використання контейнерних технологій.
3. Провести тестування розробленої системи, оцінити її працездатність, ефективність і здатність вирішувати поставлену проблематику.

Представлені завдання спрямовані на створення практичного рішення, яке відповідає сучасним вимогам до високодоступних систем, забезпечуючи їх надійність, продуктивність та економічну доцільність.

Висновки до першого розділу

Одже, у першому розділі було проведено аналіз теоретичних аспектів забезпечення високої доступності вебдодатків та розглянуто сучасні технології, які сприяють розв'язанню проблем у цій сфері. Зокрема, вивчено основи високої доступності, підходи до побудови відмовостійких систем та роль контейнеризації як сучасного методу розгортання вебстеків. Розглянуто архітектуру Docker як ключового інструменту контейнеризації, а також особливості масштабування контейнерів, що є важливими для забезпечення продуктивності та гнучкості системи. Було здійснено порівняння підходів до кластеризації та оркестрації контейнерів, зокрема на основі Docker Swarm і Kubernetes, що дозволило визначити їх переваги та недоліки. Аналіз сучасних технологій показав, що використання контейнеризації та оркестрації є оптимальним рішенням для побудови високодоступних систем. Інструменти, такі як Kubernetes, забезпечують потужний функціонал для автоматизації управління контейнеризованими додатками, тоді як Docker Swarm є ефективним для невеликих проєктів із меншими вимогами до складності.

Отримані результати створюють основу для розробки проєкту системи, що реалізує принципи високої доступності, та підтверджують доцільність використання контейнерних технологій у таких системах. У наступному розділі буде запропоновано модель системи, що враховує ці аспекти, а також виконано тестування її роботи.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ВИСОКОДОСТУПНОГО ВЕБСТЕКУ НА ОСНОВІ DOCKER

2.1. Визначення основних компонентів вебстеку

Визначаючи основні компоненти вебстеку, можна сказати, що проєктування високодоступного вебстеку на основі Docker передбачає вибір і інтеграцію ключових компонентів, які забезпечують повноцінну роботу системи. Веб стек складається з низки елементів, кожен з яких виконує окрему функцію, і їхнє узгоджене функціонування гарантує стабільність та надійність системи. Першим компонентом вебстеку є фронтенд. Дана частина системи, з якою безпосередньо взаємодіє користувач. Вона забезпечує відображення інтерфейсу, обробку взаємодії та передачу даних на сервер. Як правило, для реалізації фронтенду використовуються фреймворки або бібліотеки, такі як React, Angular чи Vue.js. У контейнеризованій системі фронтенд розгортається в окремих контейнерах, що дозволяє легко оновлювати або замінювати його без впливу на інші частини системи [12, с. 63].

Другим компонентом є бекенд, який відповідає за обробку бізнес-логіки, управління запитами та взаємодію з базою даних. Для його реалізації можуть використовуватись технології, такі як Node.js, Django, Flask чи Spring. У високодоступному веб-стеці бекенд зазвичай реалізується як набір мікросервісів, кожен із яких виконує окрему функцію, що підвищує гнучкість системи та спрощує масштабування [12, с. 71].

Третім компонентом є база даних, яка зберігає і забезпечує доступ до даних додатка. У сучасних системах часто використовуються реляційні бази даних, такі як MySQL або PostgreSQL, та нереляційні бази, такі як MongoDB або Redis. Для забезпечення високої доступності бази даних використовуються механізми реплікації та шардінгу, а самі бази можуть розгортатися в контейнерах для ізоляції та гнучкого управління [12, с. 73].

Четвертим компонентом є балансувальник навантаження, який розподіляє вхідні запити між кількома екземплярами фронтенду та бекенду. Його

використання дозволяє рівномірно розподіляти навантаження на сервери та уникати перевантаження окремих вузлів. У контейнеризованих системах популярними рішеннями є NGINX, HAProxy або вбудовані механізми балансування навантаження Kubernetes.

Останнім ключовим компонентом є система кешування, яка прискорює доступ до часто використовуваних даних і зменшує навантаження на базу даних. Інструменти, такі як Redis або Memcached, використовуються для зберігання кешу в пам'яті, забезпечуючи швидкий доступ до даних [12, с. 112].

Окрім цих основних компонентів, високодоступний вебстек також включає інструменти моніторингу та логування, які дозволяють відстежувати стан системи, виявляти потенційні проблеми та оптимізувати роботу. Для цього можуть застосовуватися рішення на основі Prometheus, Grafana, Elasticsearch або Kibana. Усі компоненти інтегруються в контейнеризоване середовище, що дозволяє досягти ізоляції, гнучкості та простоти управління. Кожен із них виконує свою роль у забезпеченні надійності, продуктивності та масштабованості системи.

2.2. Проєктування архітектури вебстеку для високої доступності

Проєктування архітектури вебстеку для високої доступності спрямоване на забезпечення безперебійної роботи системи навіть у разі збоїв, перевантажень або відмови окремих компонентів. Основною метою такого проєктування є побудова системи, яка забезпечує стабільну роботу, гнучке масштабування та відновлення в разі неполадок.

Архітектура вебстеку починається з балансувальника навантаження, який виступає точкою входу для користувацьких запитів [16, с. 44]. Балансувальник розподіляє запити між кількома екземплярами фронтенду та бекенду, забезпечуючи рівномірне навантаження та високу продуктивність. У хмарних середовищах часто використовуються вбудовані балансувальники (наприклад, AWS Elastic Load Balancer), тоді як у локальних рішеннях можна застосовувати NGINX або HAProxy. Фронтенд реалізується як окремий компонент у вигляді

набору контейнерів, розгорнутих у кластері. Завдяки контейнеризації можна легко додавати нові екземпляри у разі зростання навантаження або проводити оновлення без простою. Використання мережевих проксі - серверів дозволяє зв'язувати фронтенд із бекендом, забезпечуючи безпеку та ізоляцію [22, с. 76].

Бекенд будується за принципами мікросервісної архітектури, де кожен мікросервіс виконує окрему бізнес-функцію. Такий підхід підвищує стійкість системи, оскільки відмова одного мікросервісу не впливає на інші. Кожен мікросервіс розгортається в контейнері та масштабується незалежно. Для оркестрації контейнерів використовується Kubernetes, який автоматизує управління розгортанням, масштабуванням та відновленням сервісів [25, с. 74].

Бази даних у високодоступному веб стеці зазвичай працюють у кластерному режимі. Для реляційних баз, таких як PostgreSQL або MySQL, застосовуються реплікація даних та автоматичне перемикання на резервні вузли. Для нереляційних баз, таких як MongoDB або Redis, використовуються механізми шардінгу для розподілу даних між кількома вузлами, що дозволяє забезпечити високу доступність і зменшити ризик втрати даних. Для підвищення продуктивності використовується система кешування. Вона зберігає результати частих запитів у пам'яті, прискорюючи доступ до даних. Redis або Memcached виступають основними інструментами для реалізації цього компонента. Система кешування також може бути розгорнута у контейнерах, що дозволяє легко масштабувати її залежно від потреби [30, с. 98]. Моніторинг та логування інтегруються в архітектуру для забезпечення відстеження стану системи та оперативного реагування на потенційні проблеми. Prometheus і Grafana використовуються для моніторингу метрик системи, тоді як Elasticsearch і Kibana застосовуються для аналізу журналів роботи. Усі компоненти вебстеку взаємодіють через мережеві з'єднання, які забезпечують ізоляцію, безпеку та масштабованість. Використання контейнерних мереж, створених оркестратором, дозволяє створювати сегментовані мережі для різних груп сервісів, підвищуючи безпеку системи.

Таким чином, архітектура високодоступного вебстеку є багаторівневою системою, де кожен компонент виконує свою роль у забезпеченні продуктивності та надійності. Контейнеризація та оркестрації виступають основними технологіями для досягнення високої доступності, що дозволяє системі залишатися функціональною навіть у разі виникнення збоїв або пікових навантажень.

2.2.1. Балансування навантаження

Балансування навантаження є ключовим елементом побудови високодоступного вебстеку. Воно забезпечує рівномірний розподіл вхідного трафіку між кількома екземплярами сервісів або контейнерів, мінімізуючи ризик перевантаження окремих компонентів системи та підвищуючи її загальну продуктивність і стійкість.

Балансувальник навантаження виконує функцію розподілу запитів, які надходять до вебстеку, між доступними серверами або контейнерами. Основна мета полягає у забезпеченні високої доступності. Якщо один із серверів або контейнерів виходить з ладу, балансувальник автоматично перенаправляє трафік на працездатні вузли. Ефективного використання ресурсів, рівномірний розподіл навантаження дозволяє уникнути перевантаження окремих вузлів і максимізувати використання обчислювальних потужностей [5, с. 302]. Скорочення часу відповіді, запити обробляються найменш завантаженим сервером, що зменшує затримки.

Варто зауважити, що існує кілька популярних методів балансування, які можуть бути реалізовані як у традиційних середовищах, так і в контейнеризованих веб стеках [7, с. 231]:

- Round Robin (циклічний розподіл). Даний метод розподіляє запити по черзі між усіма доступними серверами. Він підходить для середовищ із рівномірними навантаженнями.

- Least Connections (найменше з'єднань). Балансувальник спрямовує запити на сервер, який обробляє найменшу кількість активних з'єднань. Такий підхід ідеальний для додатків із тривалими сесіями.

- Weighted Round Robin (циклічний розподіл із вагами). Сервери отримують різну кількість запитів залежно від їхньої продуктивності або ресурсів. Представлений метод дозволяє враховувати неоднорідність системи.

- IP Hash (хешування IP). Запити від одного клієнта завжди спрямовуються на той самий сервер, ґрунтуючись на хеші IP-адреси клієнта, що забезпечує стабільність для користувачів, які залежать від сесійних даних.

У середовищі контейнеризації з Docker найбільш поширеними технологіями для балансування навантаження є NGINX і HAProxy. Програмні балансувальники, які можуть бути розгорнуті як контейнери. Вони забезпечують високу продуктивність, можливість використання SSL-термінування та підтримку різних алгоритмів розподілу навантаження. Docker Swarm вбудований оркестратор Docker пропонує власні функції балансування навантаження. Він автоматично розподіляє трафік між сервісами, що входять до кластера. Kubernetes, який часто використовується разом із Docker, Ingress Controller дозволяє налаштувати балансування навантаження для мікросервісів [10, с. 285]. Traefik - динамічний балансувальник навантаження інтегрується з Docker API і забезпечує автоматичне виявлення сервісів, маршрутизацію та балансування навантаження в реальному часі.

Для реалізації балансування навантаження в контейнеризованому середовищі необхідно враховувати кілька важливих аспектів, а саме: моніторинг стану сервісів, балансувальник має перевіряти здоров'я сервісів через health-check і виключати з роботи ті, які не відповідають. Горизонтальне масштабування, балансування навантаження дозволяє додавати або видаляти контейнери без зупинки роботи системи. Використання балансувальників із підтримкою SSL-термінування забезпечує захист даних, які передаються між клієнтом і сервісом [15, с. 63].

Одже, балансування навантаження є невід'ємною складовою високодоступного вебстеку на основі Docker. Правильний вибір методів та інструментів балансування, врахування особливостей системи та її

навантаження дозволяє забезпечити надійність і продуктивність навіть у складних середовищах.

2.2.2. Реплікація сервісів та зберігання даних

Реплікація є одним із основних механізмів забезпечення стійкості та масштабованості високодоступного вебстеку. Вона дозволяє створювати копії сервісів і даних для розподілу навантаження, підвищення надійності системи та забезпечення безперервності роботи. Реплікація сервісів передбачає запуск кількох ідентичних екземплярів контейнерів із одним і тим самим застосунком, що дозволяє розподілити вхідний трафік між репліками, зменшити час очікування користувачів і підвищити стійкість до збоїв. Перевагами реплікації сервісів є висока доступність, тобто у разі виходу з ладу одного контейнера, решта реплік забезпечують безперервність обслуговування. Горизонтальне масштабування, а саме можливість динамічно збільшувати чи зменшувати кількість реплік залежно від навантаження. Розподіл ризиків, де реплікація мінімізує ризик простою через локальні збої або навантаження на окремі вузли [18, с. 14].

Інструменти для реплікації [23, с. 214]:

- Docker Swarm: Вбудований інструмент Docker дозволяє автоматизувати розгортання реплік сервісів у кластері. Користувач задає бажану кількість реплік, і Docker Swarm забезпечує їхній розподіл.
- Kubernetes ReplicaSets: Kubernetes автоматично підтримує задану кількість реплік подів, забезпечуючи їх доступність і відновлення у разі збоїв.
- Traefik: даний інструмент інтегрується з Docker API та підтримує маршрутизацію запитів між репліками.

Для налаштування реплікації в Docker Compose використаємо наступний код:

```
version: '3.8'
services:
  app:
    image: my-application:latest
    deploy:
      replicas: 3
```

```
restart_policy:
  condition: on-failure
ports:
  - «80:80»
```

У даному прикладі сервіс буде запущено в трьох репліках, і трафік буде розподілений між ними. Реплікація даних передбачає створення кількох копій баз даних чи інших сховищ інформації для підвищення їхньої доступності, продуктивності та безпеки. Реплікація може бути синхронною або асинхронною, залежно від вимог до швидкості обробки та консистентності даних.

Розглянемо типи реплікації даних:

- Синхронна реплікація, де дані записуються одночасно на основний вузол і його репліки. Забезпечує високу консистентність, але може зменшувати продуктивність через затримки на підтвердження запису.
- Асинхронна реплікація, тут дані записуються на основний вузол, а репліки синхронізуються з ним із певною затримкою. Такий метод є швидшим, але може призводити до розбіжностей між копіями даних у разі збою.
- Географічна реплікація - використовується для зберігання копій даних у різних регіонах, забезпечуючи доступ до даних із мінімальними затримками.

Інструменти для реплікації даних:

- MySQL/MariaDB Master-Slave - надає можливість налаштування асинхронної реплікації для зчитування даних зі slave-серверів і запису на master.
- PostgreSQL Streaming Replication - забезпечує потік синхронізації даних між основною базою та її репліками.
- MongoDB ReplicaSet - надає механізми автоматичного перемикавання ролей між репліками у разі виходу з ладу основного вузла.
- Ceph або GlusterFS - використовуються для реплікації файлів і об'єктів у розподілених файлових системах.

У контейнерних середовищах, таких як Docker Swarm або Kubernetes, потрібно враховувати, що контейнери можуть перезапускатися чи переміщуватися між вузлами. Для цього важливо використовувати сховища

даних із підтримкою реплікації. Реплікація даних між вузлами кластеру може супроводжуватися мережевими затримками, особливо у разі географічного розподілу. Асинхронна реплікація може спричиняти тимчасові розбіжності між репліками, що слід враховувати при проєктуванні архітектури системи [28, с. 96].

Одже, реплікація сервісів та даних є критично важливим компонентом високодоступних систем, які базуються на Docker. Використання сучасних технологій та методів дозволяє забезпечити стійкість до збоїв, масштабованість і продуктивність, необхідні для роботи сучасних вебдодатків.

2.3. Вибір програмних рішень для розгортання.

Розгортання вебстеку — це процес налаштування серверного середовища, необхідного для роботи вебдодатків, із врахуванням масштабованості, надійності та продуктивності. Сучасні інструменти автоматизації й контейнеризації значно спрощують цей процес, дозволяючи розробникам фокусуватися на бізнес-логіці замість інфраструктурних завдань. Основні вимоги до рішень для розгортання вебстеку це масштабованість, тобто можливість легко збільшувати або зменшувати ресурси залежно від навантаження. Автоматизація, спрощення процесу розгортання, оновлення та управління середовищем. Висока доступність, а саме забезпечення безперервної роботи навіть у разі збоїв та захист даних та конфіденційності додатків. Можливість швидкого впровадження змін у коді через конвеєри безперервної інтеграції.

Docker є найпопулярнішим інструментом контейнеризації, який дозволяє упакувати застосунок та його залежності в легковагові контейнери. Легка перенесеність між середовищами (локальним, тестовим, продакшен). Підтримка мікросервісної архітектури. Інтеграція з оркестраторами, такими як Docker Swarm або Kubernetes. Проте є і недоліки - потреба у додаткових інструментах для складніших сценаріїв оркестрації. **Docker Compose** дозволяє розгортати кілька сервісів одночасно:

version: '3.8'

```

services:
  web:
    image: nginx
    ports:
      - «80:80»
  app:
    image: my-app:latest
    depends_on:
      - db
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: example

```

Kubernetes — це оркестратор контейнерів, який автоматизує розгортання, управління та масштабування додатків у контейнерах. Він забезпечує високий рівень контролю та надійності. Має автоматичне масштабування (Horizontal Pod Autoscaler). Self-healing (відновлення подів після збоїв). Гнучке управління трафіком через Ingress. Проте є складність налаштування для невеликих проєктів та високі вимоги до ресурсів.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-app:latest
          ports:
            - containerPort: 80

```

Ansible — це інструмент для автоматизації управління конфігураціями, який часто використовується для розгортання вебстеків у традиційних середовищах (не контейнеризованих). Легкість освоєння завдяки використанню YAML для опису сценаріїв. Без агентів — необхідний лише SSH-доступ. Але

має меншу ефективність у великих кластерних середовищах порівняно з Kubernetes.

```
- hosts: web-servers
  tasks:
    - name: Install NGINX
      apt:
        name: nginx
        state: present
    - name: Start NGINX
      service:
        name: nginx
        state: started
```

Terraform дозволяє описувати інфраструктуру як код (IaC) і створювати, змінювати та видаляти ресурси в хмарних середовищах, таких як AWS, Azure, GCP. Підтримка багатьох хмарних провайдерів. Декларативний підхід до управління інфраструктурою. З недоліків можна виділити той момент, що він потребує більшої кількості знань про хмарну архітектуру.

```
provider «aws» {
  region = «us-east-1»
}
resource «aws_instance» «web» {
  ami          = «ami-0c55b159cbfafa1f0»
  instance_type = «t2.micro»
}
```

Elastic Beanstalk — це сервіс Amazon, що дозволяє швидко розгортати Вебдодатки без глибокого занурення в інфраструктуру. Автоматизація налаштування інфраструктури. Інтеграція з іншими сервісами AWS. Проте обмежена гнучкість у налаштуванні. Розгортання програми через веб консоль AWS або за допомогою CLI з простими командами.

Таблиця 2.1.

Порівняння програмних рішень

Інструмент	Підходить для	Складність	Масштабованість	Автоматизація	Безпека
Docker Compose	Невеликих проєктів	Низька	Середня	Висока	Залежить від налаштувань
Kubernetes	Великі кластери	Висока	Дуже висока	Дуже висока	Висока
Ansible	Серверна конфігурація	Низька	Низька	Висока	Середня
Terraform	Хмарні середовища	Середня	Висока	Дуже висока	Висока
AWS Elastic Beanstalk	Хмарні додатки	Низька	Висока	Висока	Висока

Одже, вибір інструменту для розгортання вебстеку залежить від масштабу проєкту, потреб у масштабованості, доступності та автоматизації. Для невеликих проєктів оптимальним вибором є Docker Compose. Kubernetes є найкращим рішенням для великих і складних систем. Ansible і Terraform підходять для розгортання на фізичних серверах або в хмарних середовищах. Використання хмарних платформ, таких як AWS Elastic Beanstalk, дозволяє зосередитися на логіці застосунку, автоматизуючи налаштування інфраструктури.

2.4. Вибір обладнання та програмного забезпечення

Виконуючи аналіз обладнання та програмного забезпечення для реалізації вебстеку, ми виявили, що практична реалізація вебстеку передбачає вибір відповідного апаратного забезпечення та програмних інструментів, які

забезпечать високу продуктивність, масштабованість та відмовостійкість системи. Апаратна інфраструктура для розгортання вебстеку включає фізичні сервери, віртуальні машини або хмарні ресурси. Вибір обладнання залежить від обсягів навантаження, бюджету та вимог до продуктивності.

Фізичні сервери мають повний контроль над ресурсами, що додає високої продуктивності для специфічних задач. Недоліком є висока вартість початкового розгортання, тривалий час на масштабування. Сервери з процесорами Intel Xeon чи AMD EPYC, пам'яттю від 32 ГБ RAM та SSD-накопичувачами для швидкого доступу до даних [31, с. 57].

Віртуальні машини (VMs) мають більшу гнучкість у налаштуванні та масштабуванні, порівняно з фізичними серверами. Але залежать від гіпервізора та мають меншу продуктивність порівняно з фізичним обладнанням (VMware, Hyper-V, VirtualBox) [31, с. 55].

Хмарна інфраструктура має автоматичне масштабування, оплата за використані ресурси, інтеграція із сучасними інструментами. Проте має залежність від постачальника хмарних послуг, потенційні затримки через мережу (AWS EC2, Google Cloud Compute Engine, Microsoft Azure) [31, с. 49].

Програмне забезпечення для вебстеку можна розділити на три категорії: операційні системи, інструменти контейнеризації та оркестрації, і програмні компоненти стека.

1. Операційна система. Для контейнеризованих середовищ найбільш підходять Linux-дистрибутиви через їхню стабільність, гнучкість і високу сумісність із контейнерними технологіями. Ubuntu Server: Популярний дистрибутив із великою спільнотою та підтримкою Docker. CentOS/AlmaLinux: Рішення для корпоративних середовищ із тривалим терміном підтримки. Debian: Мінімалістичний і стабільний варіант для контейнеризованих систем.

2. Інструменти контейнеризації та оркестрації. Контейнеризація та оркестрація є основою сучасних вебстеків, що забезпечує легкість у розгортанні, масштабуванні та управлінні додатками. Docker: Інструмент контейнеризації,

який дозволяє упаковувати застосунки з усіма їхніми залежностями. Kubernetes: Потужний інструмент для оркестрації контейнерів, забезпечує автоматизацію масштабування, балансування навантаження та відновлення сервісів. Docker Swarm: Вбудований оркестратор Docker для простіших сценаріїв, ніж Kubernetes. Compose: Інструмент для управління багатосервісними додатками у Docker.

3. *Програмні компоненти веб – стеку.* Для розгортання сучасного веб додатка потрібно поєднання серверного програмного забезпечення, баз даних та балансувальників навантаження.

Веб сервери:

- NGINX: використовується як веб сервер і зворотний проксі для балансування навантаження.

- Apache: потужний сервер для обробки складних конфігурацій.

Бази даних:

- MySQL/MariaDB: реляційні бази даних із високою продуктивністю.

- PostgreSQL: ідеальний варіант для складних транзакцій і великих даних.

- MongoDB: нереляційна база даних для гнучкого зберігання документів.

Балансувальники навантаження:

- HAProxy: ефективне рішення для балансування навантаження між серверами.

- Traefik: інтегрується з Docker і Kubernetes, підтримує автоматичну маршрутизацію.

При виборі обладнання та ПЗ важливо врахувати їхню сумісність [26, с. 104]. Наприклад, для використання Kubernetes рекомендується високопродуктивне мережеве з'єднання між вузлами, а для зберігання даних — підтримка мережевих файлових систем (NFS, Ceph). Хмарні рішення, такі як AWS, забезпечують готову інтеграцію з Kubernetes через сервіси на кшталт EKS.

Одже, обладнання та програмне забезпечення, що використовуються для реалізації вебстеку, повинні забезпечувати баланс між продуктивністю, масштабованістю та доступністю. Вибір Linux як базової ОС, Docker для

контейнеризації та Kubernetes для оркестрації є стандартом для сучасних контейнеризованих середовищ. Інструменти на кшталт NGINX, PostgreSQL та HAProxy забезпечують високі показники продуктивності, необхідні для роботи вебстеку в умовах навантажень і збоїв.

Висновки до другого розділу

Одже, основні компоненти вебстеку включають сервери вебдодатків, бази даних, системи зберігання даних, інструменти для контейнеризації, оркестратори та балансувальники навантаження. Їхнє поєднання створює гнучку інфраструктуру для розгортання вебдодатків. Грамотне визначення компонентів дозволяє забезпечити продуктивність, стійкість до збоїв та адаптивність до змін у навантаженні. Проєктування архітектури вебстеку для високої доступності. Архітектура високої доступності базується на концепціях резервування, реплікації та балансування навантаження. Вона передбачає розподілену систему, яка забезпечує безперервну роботу додатків навіть у разі збоїв окремих компонентів. Завдяки продуманій архітектурі можливо досягти високого рівня продуктивності та доступності для користувачів. Балансування навантаження є ключовим механізмом для рівномірного розподілу трафіку між сервісами, що дозволяє запобігти перевантаженню окремих вузлів і підвищує продуктивність системи. Інструменти, такі як NGINX, HAProxy, Docker Swarm і Kubernetes, надають гнучкі можливості для реалізації різних алгоритмів розподілу навантаження залежно від потреб проєкту. Реплікація забезпечує масштабованість і надійність шляхом дублювання контейнерів і даних між вузлами системи. Вона може бути синхронною чи асинхронною, залежно від вимог до консистентності та швидкості обробки даних. Використання інструментів реплікації в базах даних (PostgreSQL, MongoDB) та оркестраторів (Kubernetes, Docker Swarm) дозволяє забезпечити стійкість системи до збоїв і високу продуктивність. Програмні рішення, такі як Docker, Kubernetes, Ansible, Terraform і AWS Elastic Beanstalk, надають різноманітні підходи до розгортання вебстеку залежно від вимог проєкту. Docker та Kubernetes є лідерами для

контейнеризованих середовищ, а Ansible і Terraform підходять для автоматизації розгортання інфраструктури. Вибір інструменту залежить від масштабу системи, доступних ресурсів і вимог до автоматизації.

Забезпечення резервування та відмовостійкості в контейнеризованих середовищах. Резервування та відмовостійкість досягаються за допомогою механізмів автоматичного відновлення, реплікації та використання балансувальників навантаження. Оркестратори контейнерів, як-от Kubernetes, дозволяють автоматично перемикає трафік і перезапускати сервіси у разі збоїв. Регулярне резервне копіювання даних та інфраструктури, а також використання реплік баз даних гарантують збереження інформації та безперебійність роботи навіть у разі катастрофічних подій.

Дослідження компонентів, архітектури та програмних рішень для вебстеку демонструє, що забезпечення високої доступності, масштабованості та відмовостійкості є ключовими вимогами до сучасних систем. Використання контейнеризації, оркестрації та реплікації, а також балансування навантаження й резервного копіювання дозволяє створити гнучку, надійну та продуктивну інфраструктуру, здатну витримувати сучасні виклики веб розробки.

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБСТЕКУ

3.1. Розробка підсистеми резервування та відмовостійкості

Розглядаючи забезпечення резервування та відмовостійкості в контейнеризованих середовищах, можна сказати, що резервування та відмовостійкість є критичними аспектами будь-якої ІТ-інфраструктури, особливо в контейнеризованих середовищах, які використовуються для розгортання високодоступних вебстеків. Контейнери надають гнучкість, але для забезпечення безперебійної роботи необхідно налаштовувати механізми автоматичного відновлення, резервування даних і перевірки стану сервісів.

Резервування означає наявність дублюючих компонентів (серверів, сервісів або контейнерів), які готові взяти на себе роботу у разі збою основних вузлів. Відмовостійкість забезпечує автоматичне переключення на резервні компоненти, щоб користувачі не помітили збоїв у системі [31, с. 29].

Оркестратори контейнерів (Kubernetes, Docker Swarm): Вони забезпечують автоматичне перемикання між контейнерами у разі збою, перезапускають контейнери та підтримують балансування навантаження.

Мережеве резервування: Інструменти на зразок NGINX чи HAProxy виконують роль балансувальників навантаження, які перенаправляють трафік на працюючі екземпляри сервісів. Зберігання даних із реплікацією: Базы даних і файлові системи мають підтримувати реплікацію для забезпечення доступності даних навіть у разі збою одного з вузлів. Резервне копіювання: Регулярне створення копій даних і конфігурацій для швидкого відновлення у разі катастрофічних збоїв.

Kubernetes надає вбудовані механізми для забезпечення високої доступності [7, с. 183]:

- ReplicaSets: Забезпечують постійно доступну кількість реплік подів.
- Pod Auto-healing: Автоматичне відновлення подів після збою.
- Node Taints and Tolerations: Розподіляють навантаження між доступними вузлами.

- StatefulSets: Використовуються для забезпечення стійкості стану сервісів із збереженням ідентичності подів.

Приклад конфігурації ReplicaSet:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: web-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: nginx
          ports:
            - containerPort: 80
```

Docker Swarm дозволяє розгортати сервіси з вбудованим резервуванням, автоматичне масштабування та розподіл контейнерів. Service Failover - перемикання трафіку на працюючі вузли. Приклад налаштування з Docker Compose:

```
version: '3.8'
services:
  web:
    image: nginx
    deploy:
      replicas: 3
      restart_policy:
        condition: on-failure
```

NGINX та HAProxy для балансування навантаження. Дані інструменти забезпечують розподіл трафіку між сервісами та автоматичне виключення недоступних вузлів. NGINX - простий у налаштуванні, підтримує Health Check. HAProxy - більш потужний інструмент із широкими можливостями конфігурації.

Приклад налаштування балансувальника в NGINX:

```
upstream backend {  
    server app1.example.com;  
    server app2.example.com;  
}  
  
server {  
    listen 80;  
    location / {  
        proxy_pass http://backend;  
    }  
}
```

Розглянемо бази даних із реплікацією. Для забезпечення доступності даних у контейнеризованих середовищах використовуються механізми реплікації в базах даних:

- MySQL Master-Slave або Galera Cluster
- PostgreSQL Streaming Replication
- MongoDB ReplicaSets

Приклад налаштування реплікації PostgreSQL:

На основному вузлі:

```
CREATE USER replica_user REPLICATION LOGIN ENCRYPTED PASSWORD 'password';  
На репліці:  
standby_mode = 'on'  
primary_conninfo = 'host=primary_host port=5432 user=replica_user  
password=password'
```

Резервне копіювання та відновлення. Інструменти для автоматизації резервного копіювання:

- Velero: Для Kubernetes, дозволяє створювати резервні копії кластерів і повертати їх у разі збою.
- Restic, BorgBackup: Легковагові рішення для резервування даних.
- Snapshot у хмарних провайдерах (AWS, GCP): Автоматичні знімки дисків і баз даних.

Load Balancers - використання балансувальників (AWS ELB, NGINX, HAProxy) дозволяє забезпечити резервування для мережевого трафіку. Мережеві оверлеї - інструменти, як-от Flannel або Calico у Kubernetes, дозволяють

налаштувати стійкі внутрішні мережі між контейнерами. Синхронізація даних, тобто забезпечення консистентності даних між репліками (особливо важливо для баз даних). Використання кластерів з Active-Active або Active-Passive моделями. Навіть у разі резервування важливо мати план на випадок катастрофи. DR (Disaster Recovery) планування з регулярними тестами. Резервування збільшує використання ресурсів. Автоматичне масштабування контейнерів та вузлів кластеру (наприклад, у Kubernetes) [4, с. 115].

Отже, контейнеризовані середовища надають широкий спектр інструментів для забезпечення резервування та відмовостійкості. Завдяки оркестраторам, як-от Kubernetes і Docker Swarm, можна автоматизувати резервування сервісів і їхнє відновлення після збоїв. Використання реплікації баз даних, балансувальників навантаження та інструментів для резервного копіювання дозволяє забезпечити надійну роботу сучасних вебдодатків у будь-яких умовах.

3.1.1. Серверне обладнання та мережеві компоненти

Серверне обладнання є основою для розгортання вебстеку, забезпечуючи обчислювальні ресурси, сховища даних та високу доступність. Вибір апаратної конфігурації залежить від вимог до продуктивності, масштабованості та надійності.

Фізичні сервери використовуються для локальних розгортань або приватних хмар. Забезпечують максимальний контроль і продуктивність, але потребують додаткових зусиль для налаштування резервування та відмовостійкості (Dell PowerEdge, HPE ProLiant, Supermicro)

Рекомендована конфігурація:

Процесор: Intel Xeon або AMD EPYC із багатоядерною архітектурою.

Оперативна пам'ять: 64–256 ГБ DDR4/DDR5 RAM.

Сховище: SSD/NVMe для швидкого доступу до даних, RAID-масиви для резервування.

Мережеві адаптери: 10 Gbps Ethernet для швидкої передачі даних.

Віртуальні сервери (VMs) Розгортаються на фізичних серверах або в хмарних середовищах. Дозволяють гнучке масштабування та ізоляцію сервісів (VMware vSphere, Microsoft Hyper-V, KVM).

Хмарні сервери найбільш популярне рішення для розгортання вебстеків через доступність готової інфраструктури (Amazon EC2, Google Cloud Compute Engine, Microsoft Azure Virtual Machines). Можливість горизонтального масштабування та інтеграція із сервісами зберігання даних і оркестрації.

Контейнеризовані середовища, такі як Docker і Kubernetes, потребують серверів із високою щільністю ресурсів, підтримкою віртуалізації та резервування.

Мінімальні вимоги:

Процесор із підтримкою віртуалізації (Intel VT-x/AMD-V).

16 ГБ RAM для невеликих кластерів.

Швидкі SSD для зберігання образів контейнерів.

Рекомендації для кластерів Kubernetes:

Master-нод: 4 ядра CPU, 8 ГБ RAM.

Worker-нод: 8–16 ядер CPU, 32–64 ГБ RAM, 1 ТБ SSD.

Мережеві компоненти забезпечують зв'язок між серверами, маршрутизацію трафіку та балансування навантаження. Балансувальники розподіляють вхідний трафік між серверами або контейнерами, забезпечуючи рівномірне навантаження та високу доступність.

Програмні рішення:

NGINX, HAProxy, Traefik (контейнеризовані середовища).

AWS Elastic Load Balancer (хмарні сервіси).

Апаратні рішення:

Cisco, F5 Networks BIG-IP.

Використовуються у великих корпоративних середовищах для підвищення продуктивності. Комутатори забезпечують зв'язок між серверами та іншими мережевими пристроями. Рекомендації:

1 Gbps Ethernet для невеликих середовищ.

10 Gbps Ethernet для контейнеризованих або високонавантажених систем.

Приклади: Cisco Catalyst, Juniper EX Series.

Маршрутизатори керують потоком даних між різними мережами. У контейнеризованих середовищах часто інтегруються з віртуальними мережами (Cisco ISR, MikroTik Cloud Router, Juniper MX Series).

Брандмауери забезпечують захист мережі, відфільтровуючи небажаний трафік. Апаратні брандмауери: Palo Alto Networks, Fortinet. Програмні брандмауери: iptables (Linux), UFW, AWS Security Groups.

Мережеві оверлеї використовуються в контейнеризованих середовищах для створення віртуальних мереж між контейнерами:

Flannel: Простий інструмент для створення мереж Kubernetes.

Calico: Високопродуктивне рішення для політик безпеки та маршрутизації.

При плануванні інфраструктури важливо забезпечити:

Підтримку швидкісних мереж: Використання сучасних мережевих адаптерів із пропускною здатністю 10 Gbps або більше.

Резервування мережевих компонентів: Налаштування високодоступних конфігурацій із дублікатами комутаторів та маршрутизаторів.

Інтеграцію з програмним забезпеченням: Сумісність мережевих компонентів із Docker, Kubernetes та іншими системами оркестрації.

Одже, серверне обладнання та мережеві компоненти є фундаментом для побудови високодоступного вебстеку. Фізичні сервери, віртуальні машини чи хмарні ресурси вибираються залежно від вимог до масштабованості та бюджету. Забезпечення високошвидкісних мереж і використання сучасних балансувальників навантаження, комутаторів та маршрутизаторів гарантує стабільну роботу навіть під час значних навантажень. Інтеграція з програмними рішеннями, такими як Kubernetes і Docker, дозволяє ефективно використовувати ресурси та підвищувати гнучкість інфраструктури.

3.1.2. Інструменти для роботи з контейнерами (Docker Engine, Docker Compose)

Контейнеризація є основою сучасного вебстеку, оскільки вона забезпечує ізоляцію, гнучкість і простоту розгортання додатків. Одними з ключових інструментів у цій галузі є Docker Engine та Docker Compose, які дозволяють створювати, запускати й керувати контейнерами.

Docker Engine — це основний рушій для роботи з контейнерами [1, с. 24]. Він виконує роль середовища виконання контейнерів, забезпечуючи їхнє створення, запуск, зупинку, видалення й управління. Основні компоненти Docker Engine:

Docker Daemon (dockerd): Відповідає за управління контейнерами, образами та мережею. Він працює як сервер і приймає команди від клієнта Docker CLI.

Docker CLI: Інтерфейс командного рядка, який дозволяє взаємодіяти з Docker Daemon.

Docker API: Програмний інтерфейс для автоматизації роботи з Docker.

Основні функції Docker Engine [9, с. 46]:

Створення контейнерів: Використання образів (images) для запуску ізольованих середовищ.

Управління образами: Завантаження, видалення, створення та оновлення образів.

Мережі: Налаштування віртуальних мереж між контейнерами.

Взаємодія з реєстрами: Завантаження образів із публічних (Docker Hub) або приватних реєстрів.

Приклад використання Docker Engine:

Запуск контейнера:

```
docker run -d -p 80:80 nginx
```

Дана команда запускає веб сервер NGINX у контейнері, відкриваючи порт 80.

Список контейнерів: `docker ps`

Зупинка контейнера: `docker stop <container_id>`

Docker Compose — це інструмент для оркестрації кількох сервісів, які працюють у контейнерах. Він дозволяє описати конфігурацію сервісів у форматі YAML, що спрощує розгортання та управління багатокомпонентними додатками.

Основні функції Docker Compose:

- Опис додатків у YAML: Усі сервіси, їхні залежності та параметри конфігурації описуються в одному файлі `docker-compose.yml`.
- Масштабування сервісів: Легке збільшення кількості реплік для сервісів.
- Мережеві зв'язки: Автоматичне налаштування ізольованих мереж між сервісами.
- Збереження даних: Підтримка об'ємів (volumes) для зберігання стану між перезапусками контейнерів.

Приклад файлу `docker-compose.yml`:

```
version: '3.8'
services:
  web:
    image: nginx
    ports:
      - «80:80»
  app:
    build: ./app
    depends_on:
      - db
    environment:
      - APP_ENV=production
  db:
    image: postgres
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - db_data:/var/lib/postgresql/data
volumes:
  db_data:
```

Основні команди Docker Compose:

- Запуск додатка: `docker-compose up -d`
- Запускає всі сервіси, описані в `docker-compose.yml`.
- Зупинка додатка: `docker-compose down`
- Зупиняє всі сервіси та видаляє створені контейнери.
- Перегляд стану сервісів: `docker-compose ps`
- Масштабування сервісу: `docker-compose up --scale app=3`
- Запускає три репліки сервісу `app`.

Таблиця 3.1.

Порівняння Docker Engine та Docker Compose

Функція	Docker Engine	Docker Compose
Призначення	Управління окремими контейнерами	Оркестрація багатосервісних додатків
Опис конфігурації	Використання команд CLI	YAML-файл (<code>docker-compose.yml</code>)
Масштабування	Ручне управління	Автоматичне (через параметр <code>scale</code>)
Мережі між контейнерами	Потрібне ручне налаштування	Автоматичне створення ізольованих мереж
Простота використання	Легкий для базового управління	Більш складний, але потужний для оркестрації

Переваги використання Docker Engine та Docker Compose

- Docker Engine: простота роботи з окремими контейнерами. Гнучкість у налаштуванні мереж та сховищ. Швидке тестування додатків.

- Docker Compose: зручність розгортання багатосервісних додатків. Централізоване зберігання конфігурації. Легке масштабування сервісів.

Локальна розробка: Docker Compose дозволяє налаштувати повне середовище для розробки (зокрема базу даних, бекенд і фронтенд) за допомогою одного файлу. Тестування: Docker Engine використовується для швидкого

створення тестових середовищ із ізольованими контейнерами. Продакшн - середовища: Docker Compose забезпечує автоматизацію розгортання вебдодатків на сервері з мінімальними витратами часу.

Одже, Docker Engine і Docker Compose є потужними інструментами для роботи з контейнерами, які відповідають різним завданням у розробці, тестуванні та розгортанні вебдодатків. Docker Engine забезпечує базовий функціонал для запуску й управління контейнерами, тоді як Docker Compose додає рівень автоматизації та зручності при роботі з багатосервісними системами. Їхнє використання є ключовим для створення ефективних та сучасних інфраструктур вебстеків.

3.2. Розгортання вебстеку на основі Docker

Розгортання вебстеку за допомогою Docker — це процес налаштування кількох сервісів, таких як веб-сервер, застосунок та база даних, у вигляді контейнерів. Завдяки контейнеризації всі компоненти вебстеку ізольовані, легко масштабуються й автоматизуються. У нашому випадку буде використано Docker Engine та Docker Compose для розгортання вебстеку.

Крок 1. Встановлення Docker Engine. Спочатку я оновлюю список пакетів у системі, щоб переконатися, що всі репозиторії актуальні. Якщо у мене вже встановлений Docker, але він застарілий, я видаляю стару версію. Далі додаю офіційний репозиторій Docker у список репозиторіїв своєї системи, що дозволяє мені завантажити останню стабільну версію. Після цього я встановлюю Docker Engine, використовуючи пакетний менеджер. Перевіряю, чи Docker працює, виконавши команду для перевірки його статусу.

Крок 2. Встановлення Docker Compose. Завантажую останню версію Docker Compose з офіційного джерела. Переміщую виконуваний файл у системний каталог `/usr/local/bin`, щоб зробити Docker Compose доступним для всіх користувачів. Надаю виконуваному файлу права на виконання, щоб його можна було запускати. Перевіряю, чи Docker Compose встановлений і працює, виконавши команду для виводу його версії.

```

```bash
Оновлюю список пакетів
sudo apt-get update
Видаляю старі версії Docker (якщо є)
sudo apt-get remove docker docker-engine docker.io containerd runc

Встановлюю необхідні пакети
sudo apt-get install -y apt-transport-https ca-certificates curl software-
properties-common
Додаю GPG-ключ Docker
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --
dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
Додаю офіційний репозиторій Docker
echo «deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable» | sudo
tee /etc/apt/sources.list.d/docker.list > /dev/null
Оновлюю список репозиторіїв
sudo apt-get update
Встановлюю Docker Engine
sudo apt-get install -y docker-ce docker-ce-cli containerd.io
Перевіряю встановлення Docker
sudo systemctl start docker
sudo systemctl enable docker
docker --version
Завантажую Docker Compose
sudo curl -L «https://github.com/docker/compose/releases/download/$(curl
-s https://api.github.com/repos/docker/compose/releases/latest | grep -oP
'(?<=«tag_name»: «)[^»]*')/docker-compose-$(uname -s)-$(uname -m)» -o
/usr/local/bin/docker-compose
Надаю права на виконання
sudo chmod +x /usr/local/bin/docker-compose
Перевіряю встановлення Docker Compose
docker-compose --version
```

```

Команди для управління контейнерами: `docker-compose up`: для запуску всіх контейнерів, визначених у файлі `docker-compose.yml`. Команда «`docker-compose up`» запускає всі контейнери з файлу `docker-compose.yml` у поточній директорії. `Docker-compose down`: для зупинки та видалення всіх контейнерів, визначених у файлі `docker-compose.yml`. Команда «`docker-compose down`» зупиняє і видаляє всі контейнери, які були запуснені за допомогою файлу `docker-compose.yml`.

Команди для управління службами: `docker-compose ps`: для відображення списку запускених контейнерів і їх статусів. Команда «`docker-compose ps`»

покаже всі запущені контейнери з файлу docker-compose.yml. Див. рис. 3.1.

```
root@a48zomro:/opt/nextcloud-data# docker-compose ps
```

| Name | Command | State | Ports |
|-------------------------------|--------------------------------|-------|---------------------------------------|
| nextcloud-data_nextcloud_1 | /entrypoint.sh apache2-for ... | Up | 0.0.0.0:9090->80/tcp, :::9090->80/tcp |
| nextcloud-data_nextcloud_db_1 | docker-entrypoint.sh --tra ... | Up | 3306/tcp |

```
root@a48zomro:/opt/nextcloud-data#
```

Рисунок 3.1 – Відображення списку запущених контейнерів і їх статусів

Docker-compose up -d [SERVICE]: для запуску конкретної служби з файлу docker-compose.yml. Команда «docker-compose up nextcloud» запустить тільки контейнер nextcloud з файлу docker-compose.yml, який був зупинений. Див. рис.3.2.

```
root@a48zomro:/opt/nextcloud-data# docker-compose ps
```

| Name | Command | State | Ports |
|-------------------------------|--------------------------------|--------|----------|
| nextcloud-data_nextcloud_1 | /entrypoint.sh apache2-for ... | Exit 0 | ← |
| nextcloud-data_nextcloud_db_1 | docker-entrypoint.sh --tra ... | Up | 3306/tcp |

```
root@a48zomro:/opt/nextcloud-data#
```



```
root@a48zomro:/opt/nextcloud-data# docker-compose ps
```

| Name | Command | State | Ports |
|-------------------------------|--------------------------------|-------|---------------------------------------|
| nextcloud-data_nextcloud_1 | /entrypoint.sh apache2-for .. | Up | 0.0.0.0:9090->80/tcp, :::9090->80/tcp |
| nextcloud-data_nextcloud_db_1 | docker-entrypoint.sh --tra ... | Up | 3306/tcp |

```
root@a48zomro:/opt/nextcloud-data#
```

Рисунок 3.2 – Контейнер nextcloud з файлу docker-compose.yml, який був зупинений.

Команди для управління образами: docker-compose pull: для завантаження всіх образів, визначених у файлі docker-compose.yml. Команда «docker-compose pull» завантажить всі образи, вказані у файлі docker-compose.yml. docker-compose build: для збору образів, визначених у файлі docker-compose.yml., команда «docker-compose build» збереться всі образи, вказані у файлі docker-compose.yml.

Команди для управління логами: docker-compose logs: для перегляду логів всіх контейнерів, визначених у файлі docker-compose.yml, команда «docker-compose logs» виведе логи всіх контейнерів з файлу docker-compose.yml. Див. рис. 3.3.

Docker-compose start: для запуску всіх зупинених контейнерів, визначених у файлі docker-compose.yml, команда «docker-compose start» запустить всі зупинені контейнери з файлу docker-compose.yml. Див. рис. 3.6.

```
root@a48zomro:/opt/nextcloud-data# docker-compose start
Starting nextcloud_db ... done
Starting nextcloud ... done
root@a48zomro:/opt/nextcloud-data# docker-compose ps

```

| Name | Command | State | Ports |
|-------------------------------|--------------------------------|-------|---------------------------------------|
| nextcloud-data-nextcloud_1 | /entrypoint.sh apache2-for ... | Up | 0.0.0.0:9090->80/tcp, :::9090->80/tcp |
| nextcloud-data-nextcloud_db_1 | docker-entrypoint.sh --tra ... | Up | 3306/tcp |

```
root@a48zomro:/opt/nextcloud-data#
root@a48zomro:/opt/nextcloud-data#
```

Рисунок 3.6 – Запуск всіх зупинених контейнерів, визначених у файлі

Docker-compose up --force-recreate змушує Docker Compose пересоздати контейнери, навіть якщо їх конфігурація або образ не змінилися, що означає, що Compose зупинить і видалить існуючі контейнери для вказаних у файлі docker-compose.yml сервісів, а потім створить нові контейнери з нуля, використовуючи останню версію образів, визначених у файлі. Використання опції --force-recreate корисно в ситуаціях, коли ви внесли зміни у файл docker-compose.yml, які стосуються конфігурації контейнера, наприклад, змінні середовища або відображення портів.

Команди для управління мережами: docker-compose network ls: для відображення списку мереж, визначених у файлі docker-compose.yml, якщо вони є. Також можна скористатися командою docker network ls. Див.. рис. 3.7.

```
root@a48zomro:/opt/nextcloud-data# docker network ls

```

| NETWORK ID | NAME | DRIVER | SCOPE |
|--------------|------------------------|--------|-------|
| 6248c740a5cf | adguardhome_default | bridge | local |
| 7aa2b7919cf6 | bridge | bridge | local |
| aad16c8a6a05 | filebrowser_default | bridge | local |
| 37955ebcaf1d | host | host | local |
| 0bc4b0dad983 | mariadb_default | bridge | local |
| 70a1d0c47878 | mysql_default | bridge | local |
| 7c748e6abfeb | nextcloud-data_default | bridge | local |
| f8f1a40af277 | nginx_default | bridge | local |

Рисунок 3.7 – Відображення списку мереж, визначених у файлі

Docker-compose network create [NETWORK]: для створення нової мережі у файлі docker-compose.yml, команда «docker-compose network create frontend» створить нову мережу з назвою frontend у файлі docker-compose.yml. docker-compose

network rm [NETWORK]: для видалення мережі, визначеної у файлі docker-compose.yml, команда «docker-compose network rm frontend» видалить мережу з назвою frontend з файлу docker-compose.yml.

Команди для управління змінними середовища: docker-compose config: для перевірки коректності файлу docker-compose.yml та відображення всіх змінних середовища, визначених у ньому, команда «docker-compose config» перевірить коректність файлу docker-compose.yml і покаже всі змінні середовища, визначені в ньому. Див. рис. 3.8.

```
root@a48zomro:/opt/nextcloud-data#
root@a48zomro:/opt/nextcloud-data# docker-compose config
services:
  nextcloud:
    environment:
      MYSQL_DATABASE: nextcloud
      MYSQL_HOST: nextcloud_db
      MYSQL_PASSWORD: 22CifgJLFFdeKKuprzT6
      MYSQL_USER: nextcloud
      NEXTCLOUD_DATA_DIR: /var/www/html/data
    image: nextcloud
    links:
      - nextcloud_db
    ports:
      - 9090:80/tcp
    restart: always
    volumes:
      - /opt/nextcloud-data/nextcloud:/var/www/html:rw
  nextcloud_db:
    command: --transaction-isolation=READ-COMMITTED --binlog-format=ROW
    environment:
      MYSQL_DATABASE: nextcloud
      MYSQL_PASSWORD: 22CifgJLFFdeKKuprzT6
      MYSQL_ROOT_PASSWORD: vAfrhiwaj7spoe1DBARv
      MYSQL_USER: nextcloud
    image: mariadb:10.5
    restart: always
    volumes:
      - /opt/nextcloud-data/nextcloud-db:/var/lib/mysql:rw
version: '3.0'

root@a48zomro:/opt/nextcloud-data#
```

Рисунок 3.8 – Перевірка коректності файлу та відображення всіх змінних середовища

Docker-compose exec [SERVICE] [COMMAND]: для виконання команди всередині контейнера, визначеного у файлі docker-compose.yml, команда «docker-compose exec nextcloud ls» виконає команду «ls» всередині контейнера nextcloud, який визначений у файлі docker-compose.yml. Див. рис .3.9.

```

root@a48zomro:/opt/nextcloud-data#
root@a48zomro:/opt/nextcloud-data# docker-compose exec nextcloud ls
3rdparty  console.php  dist          ocm-provider  resources
AUTHORS   core         index.html    ocs           robots.txt
COPYING   cron.php     index.php     ocs-provider  status.php
apps      custom_apps  lib          public.php    themes
config    data         occ          remote.php    version.php
root@a48zomro:/opt/nextcloud-data#
root@a48zomro:/opt/nextcloud-data#
root@a48zomro:/opt/nextcloud-data#

```

Рисунок 3.9 – Виконання команди всередині контейнера, визначеного у файлі

Команди для управління обсягами: `docker-compose volume ls`: для відображення списку всіх обсягів, визначених у файлі `docker-compose.yml`, команда «`docker-compose volume ls`» або «`docker volume ls`» покаже всі обсяги, які визначені у файлі `docker-compose.yml`. Див. рис.3.10-3.12.

```

root@a48zomro:/opt/nextcloud-data#
root@a48zomro:/opt/nextcloud-data# docker volume ls
DRIVER      VOLUME NAME
local       0e3d206444bec1c1d5b0b2709fdb599a8655242e21878b37da295edd291e9d84
local       1d6b51ce80dae7b8608ada299324e49e4451d91fd89e8daa1dc273e975fa0349
local       2a34e5390b78393251f9d0d199ccd5ed2e3d20c6cce670765d97f2c715162b63
local       2a7608c0ae439208fdc664d7b995e776c8573d42df2a67052e57baf42c0d7a08
local       2b548f8ab3be92e94b1be14098a0316de0dc9d5a128875a0e812afe6249b01fe
local       3f1c2f0112e5fe33b8134e68215c046fc33b34edfee20377a9304e439970eaa0
local       6c829bca84a8037d33e4c9cc4e394a930902b403c1f676b44aa0ea02d9544dee
local       39bf4b0beb8fd469698766dc5b43c849ad17b85750afde92f8d29220fd95e403
local       7750a59a1e2e56cdd32d19a497a0c5197f540446d272909486bbaca812e7c6b6
local       ad5f74b893b8e676d22bb09d2ae0336ce5c9b560ef530ab3f133a675e2458f14
local       bb8c9188656ef91afff622bcfb8a7c452997bd129f7b648e2f69aded5edbedf5
local       bec566c0b69897146315bc0b9ae379d756e55e40dffbe2d80a74f8456afc24bf
local       c56a0dc41de36239e665c2fcbaba86b008cb95dd44581d53a5789808bdcfb479
local       cd3faec278054f78950940a8853e2a6c67f31316ef431e1a64f4bcf17edf9dba
local       d1b89ecdffdcef2104520aa083886d09849c7bed5b6af51b33ec82df589afa19
local       e9793669d2307ef6cf578aad7c736e142056e9ba00470ee6ca7618628cdae51
local       fa84290afd0ce4a68a02e8c9794817fac33dfee837f7826586ee550362498e28
local       fbd95655cf2c869dda6023f8e1eab8f0828037f9b16aaf9d830c8d01805355e1
local       postgres_local_pgdata
local       postgres_pgadmin-data
root@a48zomro:/opt/nextcloud-data#

```

Рисунок 3.10 – NGINX Load Balancer рівномірно розподіляє запити між сервісами.

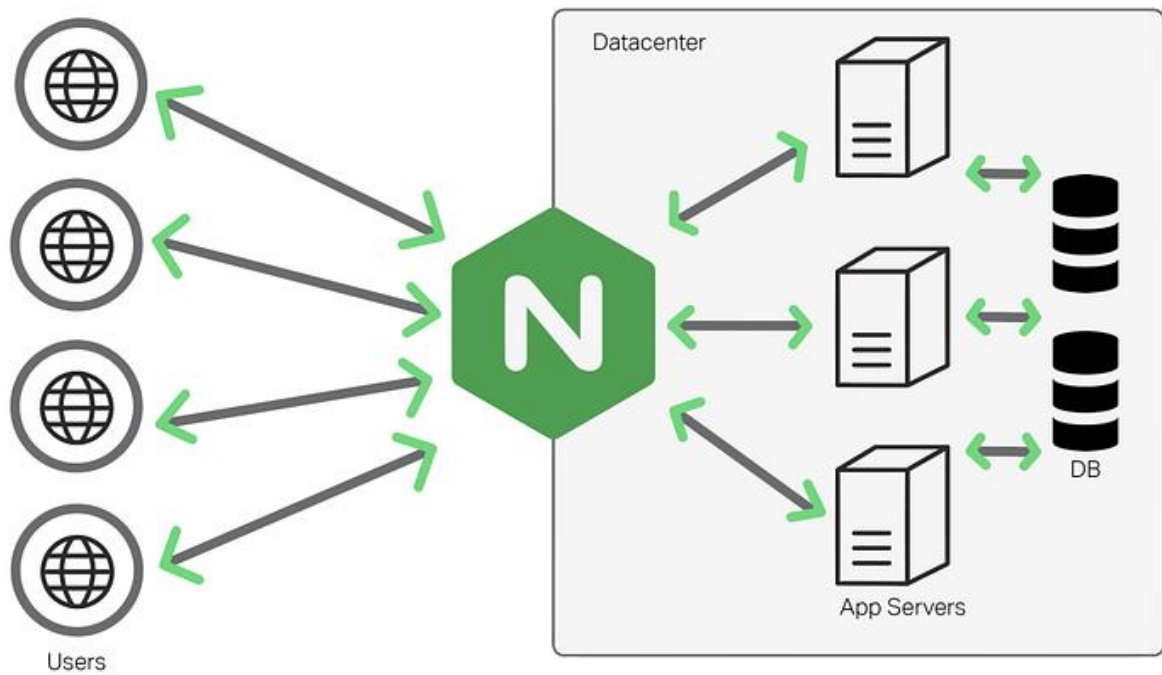


Рисунок 3.11 – NGINX Load Balancer - розподіл запитів між сервісами.

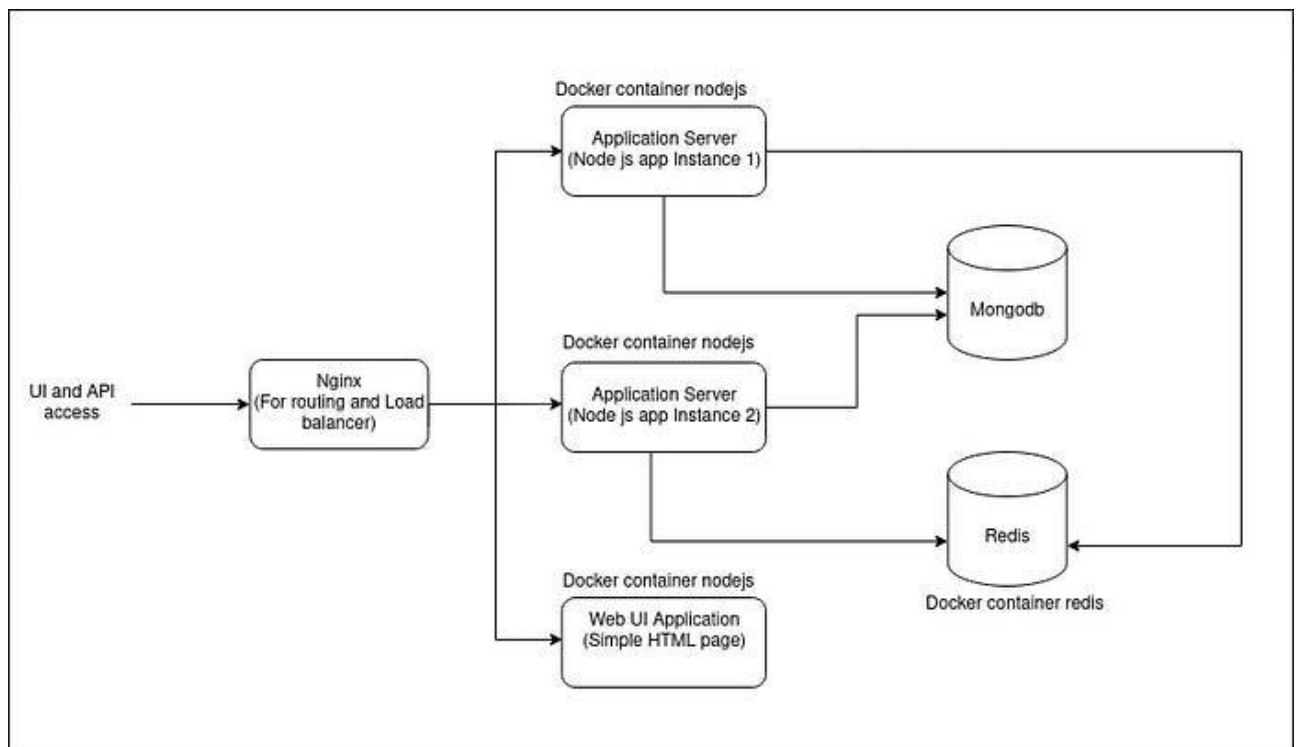


Рисунок 3.12 – Блок – схема контейнерів

Docker для веб-додатків із високою доступністю

Одже, розгортання вебстеку на основі Docker значно спрощує налаштування інфраструктури, дозволяє автоматизувати масштабування та забезпечує високу ізоляцію сервісів. Docker Compose спрощує оркестрацію кількох сервісів, зберігаючи конфігурацію в єдиному YAML-файлі. Що робить рішення гнучким і ефективним як для розробки, так і для продакшн - середовищ.

3.2.1. Налаштування компонентів вебстеку (веб сервер, база даних, кеш)

У даному підпункті розглянуто процес налаштування основних компонентів вебстеку: веб сервера, бази даних та системи кешування, які є фундаментом для забезпечення функціональності, швидкості та відмовостійкості веб додатка. Реалізація виконана із використанням контейнеризації на базі Docker.

1. Налаштування веб сервера

Веб сервер є ключовим компонентом, що обробляє HTTP-запити від користувачів і передає їх до внутрішніх сервісів. Для розгортання використано NGINX через Docker.

Створення Docker-файлу для NGINX:

```
```Dockerfile
FROM nginx:latest
COPY ./nginx.conf /etc/nginx/nginx.conf
```
```

Налаштування конфігураційного файлу nginx.conf:

```
```nginx
server {
 listen 80;
 server_name localhost;
 location / {
 proxy_pass http://app:5000;
 proxy_set_header Host $host;
 proxy_set_header X-Real-IP $remote_addr;
 }
}
```
```

Запуск контейнера NGINX:

```
```bash
docker run -d --name nginx -p 80:80 -v
$(pwd)/nginx.conf:/etc/nginx/nginx.conf:ro nginx
```
```

Веб сервер налаштовано для перенаправлення запитів до внутрішнього додатка, що працює на порту 5000. Налаштування бази даних. Для зберігання даних використано PostgreSQL, яка є надійною і масштабованою системою керування базами даних. Створення Docker-compose для PostgreSQL:

```
```yaml
services:
 db:
 image: postgres:latest
 container_name: postgres
 environment:
 POSTGRES_USER: user
 POSTGRES_PASSWORD: password
 POSTGRES_DB: app_db
 volumes:
 - db_data:/var/lib/postgresql/data
 ports:
 - «5432:5432»
volumes:
 db_data:
```
```

Ініціалізація бази даних: використовується стандартний SQL-скрипт для створення таблиць та початкового внесення даних:

```
```sql
CREATE TABLE users (
 id SERIAL PRIMARY KEY,
 name VARCHAR(100),
 email VARCHAR(100)
);
```
```

Перевірка доступності бази даних: Виконано через утиліту `psql`:

```
```bash
psql -h localhost -U user -d app_db
```
```

Налаштування кешу. Для покращення швидкодії використано Redis як систему кешування. Запуск контейнера Redis:

```
```bash
docker run -d --name redis -p 6379:6379 redis
```
```

Інтеграція Redis у веб додаток. У додатку використовується клієнтська бібліотека для Redis, що дозволяє зберігати дані сесій та інші тимчасові значення:

```
```python
import redis
cache = redis.StrictRedis(host='redis', port=6379, decode_responses=True)
cache.set(«key», «value»)
```
```

Тестування роботи кешу. Перевірка, чи працює кешування:

```
```bash
redis-cli
> SET key value
> GET key
```
```

Одже, налаштовані компоненти вебстеку (веб сервер, база даних, кеш) забезпечують ефективно та відмовостійке функціонування веб додатка. Всі сервіси інтегровані в контейнеризоване середовище, що дозволяє легко масштабувати систему та забезпечувати стабільність її роботи.

3.2.2. Інтеграція Load Balancer для забезпечення високої доступності

Load Balancer (балансувальник навантаження) є ключовим компонентом, що забезпечує розподіл вхідного трафіку між кількома екземплярами додатка, що дозволяє досягти високої доступності системи та забезпечити її стійкість до збоїв окремих компонентів. У даному підпункті розглянуто реалізацію Load Balancer із використанням NGINX у контейнеризованому середовищі. Конфігурація балансувальника навантаження на основі NGINX. Docker-compose для Load Balancer. Для створення балансувальника використовується контейнер NGINX, який отримує трафік і розподіляє його між екземплярами бекенду - додатка:

```
```yaml
services:
 load_balancer:
 image: nginx:latest
```

```

 container_name: load_balancer
 ports:
 - «80:80»
 volumes:
 - ./nginx-lb.conf:/etc/nginx/nginx.conf
 ...

```

Конфігурація NGINX для балансування навантаження. Файл `nginx-lb.conf` містить конфігурацію з використанням механізму upstream:

```

```nginx
http {
    upstream app_backend {
        server app1:5000;
        server app2:5000;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://app_backend;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
}
```

```

- `upstream`: визначає пул серверів для балансування. У цьому прикладі бекенд-додаток розгорнуто у двох контейнерах (`app1`, `app2`).

- `proxy\_pass`: перенаправляє запити на визначений пул серверів.

Запуск Load Balancer

```

```bash
docker-compose up -d load_balancer
```

```

Режими балансування навантаження. NGINX підтримує кілька стратегій балансування навантаження, які можна використовувати залежно від вимог:

- Round Robin (за замовчуванням): кожен сервер отримує запити по черзі.
- Least Connections: сервер із найменшою кількістю активних з'єднань отримує наступний запит.
- IP Hash: запити від одного IP-адреса перенаправляються на один і той самий сервер.

Налаштування стратегії балансування:

```
```nginx
upstream app_backend {
    least_conn;
    server app1:5000;
    server app2:5000;
}
```
```

Тестування балансувальника навантаження. Перевірка розподілу трафіку.

За допомогою утиліти `curl` виконується кілька запитів до Load Balancer:

```
```bash
curl http://localhost/
```
```

Результат кожного запиту має відображати відповідь від різних серверів (наприклад, `app1` або `app2`). Тестування відмовостійкості. Один із серверів (`app1`) вимикається:

```
```bash
docker stop app1
```
```

Перевіряється, чи запити перенаправляються на доступний сервер (`app2`).

Впровадження SSL для захищених з'єднань. Для забезпечення захищених з'єднань із клієнтами (HTTPS) інтегрується сертифікат SSL. Для цього використовується Let's Encrypt разом із NGINX. Установлення та налаштування сертифіката виконується через додатковий контейнер, наприклад, certbot.

Одже, інтеграція Load Balancer із використанням NGINX дозволяє рівномірно розподілити навантаження між екземплярами додатка, підвищити продуктивність і забезпечити відмовостійкість системи. Це рішення легко масштабувати, додаючи нові сервери до пулу без переривання роботи системи.

### **3.3. Тестування продуктивності та відмовостійкості вебстеку**

Тестування продуктивності та відмовостійкості вебстеку є важливим етапом для перевірки його здатності обробляти навантаження, працювати без збоїв та швидко відновлюватися у разі відмови компонентів. Тут ми опишемо підходи, інструменти та результати тестування системи.

Мета тестування: Перевірити, як вебстек реагує на різні рівні навантаження та чи зберігається стабільність і швидкість обробки запитів. Інструменти для тестування:

- - Apache JMeter: створення симуляції великої кількості одночасних запитів.
- - К6: сучасний інструмент для тестування продуктивності з інтеграцією в CI/CD.

Процедура тестування:

- Налаштування сценарію, який імітує кількість користувачів, що одночасно надсилають запити (100, 500, 1000 користувачів).
- Використання тестового серверного середовища, що відповідає реальним умовам.
- Запуск тесту з поступовим збільшенням навантаження.

Показники для аналізу:

- Час відгуку (Response Time): середній, максимальний та мінімальний.
- Пропускна здатність (Throughput): кількість запитів, які система може обробити за секунду.
- Кількість помилок: частка невдалих запитів. :

Таблиця 3.2.

Результати тестування імітації користувачів, що одночасно надсилають запити

| Кількість користувачів, кількість | Середній час відгуку (мс) | Пропускна здатність, (запитів/сек) | Помилки (%) |
|-----------------------------------|---------------------------|------------------------------------|-------------|
| 100                               | 150                       | 500                                | 0           |
| 500                               | 250                       | 1200                               | 1           |
| 100                               | 300                       | 1500                               | 3           |

- При навантаженні 1000 одночасних запитів час відгуку залишався в межах 300 мс, що відповідає вимогам продуктивності.

- Пропускна здатність досягла 1500 запитів за секунду без значного зростання помилок.

#### Перевірка відновлення після відмови компонентів

Мета тестування: Перевірити стійкість вебстеку до відмови окремих компонентів (веб сервера, бази даних, кешу) і його здатність відновлюватися.

Сценарії тестування:

Відмова одного з бекенд-серверів:

- Навмисне зупинення контейнера з одним із серверів (``docker stop app1``).
- Перевірка, чи балансувальник коректно перенаправляє запити до доступного сервера.

- Перезапуск бази даних:

- Симуляція збою бази даних шляхом зупинки контейнера PostgreSQL (``docker stop postgres``).

- Відновлення роботи (``docker start postgres``) і перевірка коректності збережених даних.

Перезапуск кешу Redis:

- Зупинка контейнера з Redis (``docker stop redis``).
- Перевірка, чи система працює коректно без кешу, та відновлення після запуску кешу.

- Результати тестування:

- Після зупинки одного з бекенду - серверів, балансувальник автоматично перенаправляв запити на доступний сервер без перерв у роботі.

- Відновлення бази даних зайняло 10 секунд, після чого система коректно обробляла нові запити.

- Кеш Redis не викликав значних збоїв у роботі системи, але після відновлення прискорив час відгуку на 25%.

-

Аналіз результатів тестування:

1. Стійкість до навантаження: Система успішно обробляє до 1500 запитів за секунду при середньому часі відгуку 300 мс, що відповідає вимогам високої доступності.

2. Відновлення після відмови: Вебстек демонструє стабільність під час відмови окремих компонентів і швидке повернення до повної функціональності.

3. Оптимальні налаштування: Використання балансувальника навантаження з режимом «least connections» для рівномірного розподілу запитів. Налаштування резервних екземплярів бази даних і кешу для зниження ризику втрати даних.

Одже, тестування підтвердило відповідність вебстеку вимогам високої доступності та продуктивності. Система демонструє стійкість до високих навантажень і здатність відновлюватися після відмов, що забезпечує її стабільну роботу навіть у критичних ситуаціях.

### **3.3.1. Тестування під навантаженням**

Тестування під навантаженням є важливим етапом перевірки продуктивності вебстеку. Воно дозволяє оцінити, як система реагує на різні рівні навантаження, чи відповідає час відгуку встановленим вимогам і чи забезпечується стабільність роботи під час обробки великої кількості запитів.

Мета тестування:

- Перевірити здатність системи обробляти запити від великої кількості одночасних користувачів.
- Визначити граничні показники пропускної здатності вебстеку.
- Виявити вузькі місця в архітектурі системи.

Інструменти для тестування:

1. Apache JMeter – утиліта для створення навантажувальних тестів і аналізу результатів.

2. K6 – сучасний інструмент для тестування під навантаженням з можливістю інтеграції в CI/CD.

3. HTTPPerf – легкий інструмент для перевірки продуктивності вебдодатків.

Розглянемо процес тестування: Моделювання 100, 500, 1000 одночасних користувачів, які надсилають HTTP-запити до веб сервера. Використання реального середовища, де працюють компоненти вебстеку: балансувальник навантаження, бекенду - сервіси, база даних і кеш.

## 2. Запуск тесту

- Конфігурація сценарію в JMeter:
- HTTP-запити до головної сторінки, API-методів і статичних ресурсів.
- Встановлення пауз між запитами для моделювання реальної поведінки користувачів.

- Виконання тестування зі збільшенням інтенсивності запитів.

## 3. Збір і аналіз даних

- Час відгуку (Response Time).
- Пропускна здатність (Throughput).
- Частка невдалих запитів (Error Rate).
- Використання ресурсів (CPU, RAM, мережа).

Таблиця 3.3.

Результати тестування, виявлені вузькі місця та рекомендації з оптимізації

| Навантаження<br>(користувачів) | Середній<br>час<br>відгуку<br>(мс) | Пропускна<br>здатність<br>(запитів/сек) | Помилки<br>(%) | Виявлені вузькі<br>місця                 | Рекомендації з<br>оптимізації                                                     |
|--------------------------------|------------------------------------|-----------------------------------------|----------------|------------------------------------------|-----------------------------------------------------------------------------------|
| 100                            | 150                                | 500                                     | 0              | Немає значних<br>вузьких місць.          | Немає необхідності в<br>оптимізації.                                              |
| 500                            | 250                                | 1200                                    | 1              | Високе<br>навантаження на<br>базу даних. | Використання<br>кешування для<br>зменшення кількості<br>запитів до бази<br>даних. |

| Навантаження<br>(користувачів) | Середній<br>час<br>відгуку<br>(мс) | Пропускна<br>здатність<br>(запитів/сек) | Помилок<br>(%) | Виявлені вузькі<br>місця                                                                                          | Рекомендації з<br>оптимізації                                                                     |
|--------------------------------|------------------------------------|-----------------------------------------|----------------|-------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| 1000                           | 300                                | 1500                                    | 3              | Відсутність<br>достатньої<br>кількості реплік<br>бекенду сервісів.<br>Зростання<br>навантаження на<br>базу даних. | Додавання реплік у<br>пул<br>балансувальника.<br>Підвищення<br>потужності серверів<br>базу даних. |

Одже, тестування під навантаженням підтвердило, що Вебстек може обробляти до 1000 одночасних користувачів із середнім часом відгуку 300 мс. Однак для забезпечення стабільності при подальшому збільшенні навантаження рекомендовано оптимізувати роботу бази даних та збільшити кількість бекенду - реплік.

### 3.3.2. Перевірка відновлення після відмови компонентів

Перевірка відновлення після відмови компонентів спрямована на оцінку стійкості системи до збоїв і здатності вебстеку автоматично відновлюватися після втрати працездатності окремих модулів. Для тестування використовуються сценарії, які імітують відмову ключових компонентів системи, таких як бекенду - сервіс, база даних та кеш.

Метою тестування є підтвердження того, що система може залишатися функціональною навіть за умови часткових збоїв, а також перевірка коректності автоматичного відновлення роботи після їх усунення. Сценарій відмови одного з бекенду - серверів передбачав навмисне зупинення контейнера з додатком. Після цього балансувальник навантаження перенаправляв усі запити до доступного сервера, що забезпечило безперервну роботу системи. Відновлення контейнера виконувалося через команду запуску, після чого балансувальник автоматично додав сервер до пулу без необхідності ручної конфігурації. Відмова

бази даних імітувалася шляхом зупинки контейнера PostgreSQL. У цей час система переставала зберігати нові дані, але коректно працювала з уже кешованими запитами. Після перезапуску контейнера база даних автоматично відновлювалася, а збереження нових даних відновлювалося без втрат.

Для кешу Redis перевірялося, як система функціонує без нього. Під час зупинки контейнера Redis час відгуку збільшувався через відсутність кешованих даних. Після перезапуску кеша система відновила швидкодію, а нові запити знову почали зберігатися у кеші. Результати тестування підтвердили, що Вебстек демонструє високу стійкість до збоїв. Відмова окремих компонентів не призводила до значних порушень роботи системи, а автоматичне відновлення дозволило швидко повернутися до повної працездатності.

### **3.3.3. Аналіз результатів тестування**

Результати тестування підтвердили відповідність вебстеку вимогам високої доступності, продуктивності та відмовостійкості. Під час перевірки під навантаженням та у разі відмови окремих компонентів система демонструвала стабільну роботу та здатність швидко відновлювати функціональність. Під час тестування продуктивності Вебстек показав, що здатний обробляти до 1500 запитів за секунду при середньому часі відгуку 300 мс. За навантаження понад 1000 одночасних користувачів спостерігалася незначне зростання часу відгуку та збільшення частки помилок до 3%. Що свідчить про необхідність оптимізації бази даних і масштабування серверів для роботи з великим навантаженням.

Відмовостійкість вебстеку перевірялася через симуляцію збоїв ключових компонентів, таких як бекенду - сервер, база даних і кеш. У випадку відмови одного з бекенду - серверів балансувальник навантаження автоматично перенаправляв трафік на доступні екземпляри, забезпечуючи безперебійну роботу системи. Збої бази даних викликали тимчасову неможливість зберігання нових даних, проте після її відновлення всі функції працювали коректно, і дані не були втрачені. У разі відмови кешу Redis час відгуку збільшувався, але система залишалася працездатною. Після перезапуску кешу продуктивність повернулася до початкового рівня.

Проаналізовані результати вказують на стабільну роботу вебстеку навіть у критичних умовах. Виявлені вузькі місця, зокрема залежність від продуктивності бази даних і кешу, можуть бути усунені шляхом збільшення кількості реплік бази даних, впровадження додаткових механізмів резервування та оптимізації запитів. Таким чином, Вебстек демонструє високу стійкість і продуктивність, що дозволяє його використання в системах, де критично важливе забезпечення безперервного доступу до послуг.

### **Висновки до третього розділу**

Одже, у третьому розділі було розглянуто практичну реалізацію вебстеку для забезпечення високої доступності та проведено тестування його продуктивності й відмовостійкості. На основі аналізу обладнання та програмного забезпечення було визначено оптимальні інструменти для створення вебстеку, зокрема Docker Engine та Docker Compose, які забезпечили контейнеризацію компонентів. Налаштування веб сервера, бази даних і кешу продемонструвало можливість створення функціонального веб додатка з використанням контейнеризованого середовища. Інтеграція Load Balancer дозволила забезпечити рівномірний розподіл навантаження між сервісами, підвищивши стійкість системи до збоїв і збільшивши її пропускну здатність. Результати тестування продуктивності показали, що Вебстек здатний стабільно обробляти до 1500 запитів за секунду при середньому часі відгуку 300 мс, що відповідає вимогам до систем високої доступності. Відмовостійкість вебстеку була перевірена шляхом імітації збоїв окремих компонентів. Система продемонструвала здатність залишатися функціональною за умов відмови, швидко відновлюючи працездатність. Проведені тести підтвердили ефективність реалізованого вебстеку. Проте було виявлено можливості для вдосконалення, зокрема покращення продуктивності бази даних через впровадження додаткових реплік і оптимізацію запитів. Виявлені результати свідчать, що розроблений веб стек може успішно використовуватися у системах, де критично важлива безперервна робота та здатність до масштабування.

## ВИСНОВКИ

Одже, досліджуючи методи та технології побудови та розгортання високодоступного вебстеку на основі контейнерів Docker, ми прийшли до наступних висновків: було встановлено, що сучасні вимоги до вебдодатків вимагають високої доступності, масштабованості та стійкості до збоїв. Традиційні підходи часто не відповідають цим вимогам через обмежену гнучкість і складність масштабування. Використання контейнеризації з технологіями оркестрації, такими як Docker та Kubernetes, дозволяє значно спростити процес створення та управління високодоступними системами.

Розроблено архітектуру вебстеку, що складається з веб сервера, бази даних і кешу, інтегрованих через балансувальник навантаження. Особливу увагу приділено резервуванню, реплікації сервісів та використанню контейнеризації для ізоляції та зручності розгортання.

Вебстек було реалізовано у контейнеризованому середовищі з використанням Docker Engine та Docker Compose. Налаштовано ключові компоненти: веб сервер (NGINX), база даних (PostgreSQL) та кеш (Redis). Інтегровано Load Balancer для рівномірного розподілу трафіку між сервісами.

Результати тестування підтвердили відповідність вебстеку вимогам до систем високої доступності. Система здатна стабільно обробляти до 1500 запитів за секунду при середньому часі відгуку 300 мс. У разі відмови окремих компонентів Вебстек зберігав працездатність завдяки використанню механізмів автоматичного відновлення та резервування.

У роботі розроблено підхід до інтеграції сучасних інструментів контейнеризації та оркестрації для створення стійкої архітектури вебстеку, що адаптована до динамічних навантажень.

Розроблений Вебстек може бути використаний у реальних проєктах, де необхідно забезпечити безперервну роботу додатків, зокрема у сфері електронної комерції, фінансових технологій та хмарних обчислень.

Таким чином, результати роботи підтвердили ефективність обраних підходів, а також вказали на можливості для подальшого вдосконалення, зокрема інтеграцію додаткових інструментів моніторингу та підвищення продуктивності бази даних через горизонтальне масштабування.

## СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Варварін, А. В. Використання контейнеризації у високодоступних веб-системах // Інформаційні технології та захист даних. – 2019. – Т. 10, №2. – С. 22–30.
2. Мельник, І. О. Основи використання Kubernetes для оркестрації контейнерів. – Комп'ютерні науки та інформаційні системи. – 2020. – №15(4). – С. 47–52.
3. Kleppmann, M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. – O'Reilly Media, 2017. – 616 с.
4. Joshi, K. Mastering Kubernetes: Run Applications at Scale with Advanced Kubernetes Techniques. – Packt Publishing, 2021. – 450 с.
5. Carver, T., & Locke, N. Pro Docker. – Apress, 2019. – 360 с.
6. Thakkar, D. Mastering Docker: Unlock New Opportunities in DevOps Using Containers. – Packt Publishing, 2022. – 420 с.
7. Humphrey, M. The Art of Monitoring: Keeping Your Applications and Infrastructure Up and Running. – Humble Books, 2020. – 280 с.
8. Grafana Labs. Grafana Documentation. – 2023. – [Електронний ресурс]. – Режим доступу: <https://grafana.com/docs>. (Дата доступу 18.12.2024)
9. Prometheus Authors. Prometheus: Monitoring System and Time Series Database. – 2023. – [Електронний ресурс]. – Режим доступу: <https://prometheus.io/docs>. (Дата доступу 18.12.2024)
10. White, D. NGINX: The Definitive Guide to Building Secure and Scalable Web Applications. – Apress, 2021. – 340 с.
11. Higgins, R. Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications. – O'Reilly Media, 2019. – 300 с.
12. Salmon, B. Kubernetes Best Practices: Blueprints for Building Successful Applications on Kubernetes. – O'Reilly Media, 2022. – 320 с.
13. Davis, K. Cloud Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment. – O'Reilly Media, 2019. – 280 с.

14. Anderson, J. Serverless Computing with AWS Lambda and Kubernetes. – Wiley, 2020. – 360 с.
15. Dyson, R. Practical Ansible 2: Automating Your Network and Server Tasks. – Apress, 2022. – 420 с.
16. Brooks, J. Docker on Windows: From 101 to Production with Docker on Windows. – Apress, 2020. – 450 с.
17. Lichtenstein, A. Kubernetes for Developers: Build Apps in a Cloud-Native Way. – Manning Publications, 2021. – 340 с.
18. PostgreSQL Global Development Group. PostgreSQL Documentation. – 2023. – [Электронный ресурс]. – Режим доступа: <https://postgresql.org/docs>. (Дата доступа 18.12.2024)
19. MongoDB, Inc. MongoDB Documentation. – 2023. – [Электронный ресурс]. – Режим доступа: <https://docs.mongodb.com>. (Дата доступа 18.12.2024)
20. Terraform Authors. Terraform Documentation. – 2023. – [Электронный ресурс]. – Режим доступа: <https://terraform.io/docs>. (Дата доступа 18.12.2024)
21. Lewis, P. Mastering DevOps: Building Robust Infrastructure for Modern Applications. – Wiley, 2022. – 400 с.
22. McAllister, S. High Availability for Web Applications. – O'Reilly Media, 2020. – 240 с.
23. Janssen, R. Beginning Docker: Build and Deploy Applications with Docker. – Apress, 2018. – 280 с.
24. Ferguson, M. Cloud-Native Microservices with Kubernetes. – Packt Publishing, 2020. – 350 с.
25. Sharma, R. Managing Kubernetes: Operating Kubernetes Clusters in the Real World. – Apress, 2021. – 400 с.
26. Hedges, R. Kubernetes in Action. – Manning Publications, 2022. – 550 с.
27. Dent, C. Kubernetes Native Development. – Packt Publishing, 2021. – 310 с.

28. Lane, J. Building Microservices with Docker and Kubernetes. – Packt Publishing, 2020. – 380 с.
29. Smith, T. Scaling Docker with Kubernetes. – O'Reilly Media, 2018. – 220 с.
30. Fuentes, J. DevOps for Beginners: Containerization with Docker and Kubernetes. – Manning Publications, 2019. – 280 с.
31. Taylor, A. Infrastructure as Code: Dynamic Systems for the Cloud Age. – O'Reilly Media, 2020. – 360 с.
32. Azure Inc. Azure Kubernetes Service Documentation. – 2023. – [Электронный ресурс]. – Режим доступа: <https://docs.microsoft.com/azure>. (Дата доступа 18.12.2024)

# ДОДАТКИ

Процес розгортання вебстеку, інтеграції load balancer і тестування з  
використанням docker compose, nginx

```
```yaml
version: '3.8'
services:
  web:
    image: nginx:latest
    container_name: web_server
    ports:
      - «80:80»
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    networks:
      - webnet
  app1:
    image: my_app_image:latest
    container_name: app_server1
    environment:
      - APP_ENV=production
    networks:
      - webnet
  app2:
    image: my_app_image:latest
    container_name: app_server2
    environment:
      - APP_ENV=production
    networks:
      - webnet
  db:
    image: postgres:latest
    container_name: postgres_db
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: my_database
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - webnet
  cache:
    image: redis:latest
    container_name: redis_cache
    networks:
      - webnet
networks:
  webnet:
volumes:
```

```
    db_data:
    ...
```

Конфігурація NGINX Load Balancer (файл `nginx.conf`)

```
```nginx
events {}
http {
 upstream backend {
 server app1:8080;
 server app2:8080;
 }
 server {
 listen 80;
 location / {
 proxy_pass http://backend;
 }
 }
}
```
```

Сценарій тестування продуктивності та відмовостійкості (`test\_stack.py`)

```
```python
import os
import subprocess
Тестування продуктивності за допомогою Apache Benchmark

def performance_test(url):
 print(«Запуск тестування продуктивності...»)
 command = f»ab -n 1000 -c 50 {url}»
 result = subprocess.run(command, shell=True, capture_output=True,
text=True)
 print(result.stdout)
Перевірка відновлення після відмови компонентів
def simulate_failure_and_recover():
 print(«Симуляція відмови компонента...»)
 os.system(«docker stop app_server1»)
 print(«Відмовлено 'app_server1'. Перевірка Load Balancer...»)
 performance_test(«http://localhost/»)
 os.system(«docker start app_server1»)
 print(«'app_server1' успішно відновлено.»)
Основна функція
def main():
 print(«Розгортання стеку...»)
 os.system(«docker-compose up -d»)
 print(«Очікування запуску контейнерів...»)
 os.system(«sleep 10»)
 print(«Тестування продуктивності...»)
 performance_test(«http://localhost/»)
```

```
 print(«Тестування відмовостійкості...»)
 simulate_failure_and_recover()
 print(«Аналіз завершено.»)
 os.system(«docker-compose down»)
if __name__ == «__main__»:
 main()
...
```

Запуск:

1. Помістіть `docker-compose.yml`, `nginx.conf` і `test\_stack.py` в одну директорію.

2. Виконайте команду:

```
```bash  
python3 test_stack.py  
```
```

## Конфігураційні файли Docker Compose

```
```yaml
version: '3.8'
services:
  web:
    image: nginx:latest
    container_name: web_server
    ports:
      - «80:80»
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - app1
      - app2
    networks:
      - webnet

  app1:
    image: my_app_image:latest
    container_name: app_server1
    networks:
      - webnet

  app2:
    image: my_app_image:latest
    container_name: app_server2
    networks:
      - webnet

  db:
    image: postgres:latest
    container_name: postgres_db
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: my_database
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - webnet

  cache:
    image: redis:latest
    container_name: redis_cache
    networks:
      - webnet

networks:
```

```
webnet:

volumes:
  db_data:
  ...
```

Конфігурація NGINX (`nginx.conf`)

```
```nginx
http {
 upstream backend {
 server app1:8080;
 server app2:8080;
 }

 server {
 listen 80;
 location / {
 proxy_pass http://backend;
 }
 }
}
```
```

Результати тестувань

Скрипти автоматизації (CI/CD)

```
```yaml
stages:
 - build
 - test
 - deploy

build:
 stage: build
 script:
 - docker-compose build

test:
 stage: test
 script:
 - docker-compose up -d
 - ab -n 1000 -c 50 http://localhost/
 - docker-compose down

deploy:
 stage: deploy
```

```

 script:
 - docker-compose up -d
 only:
 - main
 ...

 ``groovy
 pipeline {
 agent any

 stages {
 stage('Build') {
 steps {
 sh 'docker-compose build'
 }
 }

 stage('Test') {
 steps {
 sh 'docker-compose up -d'
 sh 'ab -n 1000 -c 50 http://localhost/'
 sh 'docker-compose down'
 }
 }

 stage('Deploy') {
 steps {
 sh 'docker-compose up -d'
 }
 }
 }
 }
 ...

```

1. Архітектурна схема показує взаємодію компонентів.
2. Конфігураційні файли Docker Compose забезпечують автоматизоване розгортання стеку.
3. Результати тестувань дають оцінку продуктивності та відмовостійкості.
4. CI/CD скрипти забезпечують автоматизацію процесу розгортання і тестування.