ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Кваліфікаційна робота

на тему: «**МЕТОДИ РОЗГОРТАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА МЕТОДІВ БЕЗПЕКИ ДЛЯ ВЕБ-ДОДАТКІВ»**

на здобуття освітнього ступеня магістра
зі спеціальності __122 Комп'ютерні науки_____
(код, найменування спеціальності)
освітньо-професійної програми __Комп'ютерні науки _____
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ ___Євгеній СОНЕЦЬ_____
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. КНДМ - 62

Євгеній СОНЕЦЬ
(Ім'я, ПРІЗВИЩЕ)

Керівник: к. т. н., доцент Володимир ВАСИЛЕНКО
Науковий ступінь, вчене звання *(Ім'я, ПРІЗВИЩЕ)*

Рецензент: _____
Науковий ступінь, вчене звання *(Ім'я, ПРІЗВИЩЕ)*

Київ – 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально–науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук
Ступінь вищої освіти − «Магістр»
Спеціальність підготовки − «122 Комп'ютерні науки»
Освітньо-професійна програма − «Комп'ютерні науки»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук
_____Віктор Вишнівський
"____" _____2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Сонець Євгеній Анатолійович
(прізвище, ім'я, по батькові)

1.Тема кваліфікаційної роботи: «Методи розгортання мікросервісної архітектури та методів безпеки для веб-додатків»
Керівник кваліфікаційної роботи __Володимир Василенко, к.т.н., доцент.,
(ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу № 320 від 15.10.2024р.
2. Строк подання студентом роботи 15.12.2024 р.
3. Вихідні дані до роботи: Параметри веб-сервера та бази даних, Мікросервісна архітектура та оркестрація, Методи забезпечення безпеки веб-додатків, Автоматизація розгортання, Моніторинг та тестування продуктивності, Науково-технічна література.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Основні принципи розгортання мікросервісної архітектури.
2. Використання контейнеризації для забезпечення ізоляції мікросервісів.
3. Оркестрація мікросервісів за допомогою Kubernetes: переваги та можливості.
4. Методи забезпечення безпеки в мікросервісах: аутентифікація, авторизація та шифрування.
5. Інструменти автоматизації розгортання: Ansible, Terraform, Helm.
6. Вплив методів моніторингу та логування на продуктивність і стабільність системи.

7. Аналіз тестування продуктивності мікросервісних веб-додатків.
8. Рекомендації щодо покращення продуктивності та безпеки мікросервісів.
9. Висновки за результатами розробки системи.

5. Перелік ілюстративного матеріалу: *презентація*
1. Мета роботи, об`єкт та предмет дослідження
2. Архітектурна модель мікросервісного веб-додатка
3. Схема оркестрації контейнерів за допомогою Kubernetes
4. Конфігурація системи аутентифікації та авторизації
5. Використання інструментів для автоматизації розгортання (Ansible, Terraform)
6. Інтеграція інструментів моніторингу (Prometheus, Grafana) у систему
7. Методи тестування продуктивності мікросервісів
8. Рекомендації щодо оптимізації безпеки даних у мікросервісній архітектурі
9. Механізми резервного копіювання та відновлення даних
10. Висновки та подальші перспективи розвитку роботи
6. Дата видачі завдання 05.09.2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	09.10.2024	вик.
2	Сучасні методи розгортання мікросервісної архітектури для веб-додатків	23.10.2024	вик.
3	Автоматизація розгортання мікросервісів за допомогою Kubernetes, Ansible, Terraform	04.11.2024	вик.
4	Моніторинг і тестування продуктивності веб-додатків	22.11.2024	вик.
5	Методи аутентифікації та авторизації: протоколи OpenID Connect, OAuth 2.0, JWT	02.12.2024	вик.
6	Вступ, висновки, реферат	10.12.2024	вик.
7	Розробка демонстраційних креслень	14.12.2024	вик.

Здобувач вищої освіти _____ Євгеній СОНЕЦЬ
(підпис) (Ім'я, ПРІЗВИЩЕ)

Керівник кваліфікаційної роботи _____ Володимир ВАСИЛЕНКО
(підпис) (Ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 82 стор., 5 рис., 1 таблиця, 20 джерел.

Мета роботи – проведення глибокого аналізу існуючих методів розгортання мікросервісної архітектури, включаючи контейнери, оркестрацію та автоматизацію, а також дослідження сучасних підходів до забезпечення її безпеки.

Об`єкт дослідження – мікросервісна архітектура, яка представляє собою сучасний підхід до створення веб-додатків шляхом поділу на незалежні сервіси, кожен з яких виконує конкретні функції.

Предмет дослідження – методи розгортання мікросервісної архітектури, що включають аналіз контейнеризації, оркестрації та інтеграції сервісів, а також сучасні підходи до забезпечення безпеки, зокрема застосування аутентифікації, авторизації та шифрування для захисту даних і запобігання можливим загрозам.

Короткий зміст роботи:

Магістерська робота присвячена розробці методів розгортання мікросервісної архітектури та забезпечення безпеки для веб-додатків. У роботі розглянуто актуальність використання мікросервісної архітектури в умовах зростання складності програмних систем та сучасні підходи до її впровадження. Досліджено автоматизацію розгортання з використанням таких інструментів, як Docker, Kubernetes, Ansible та Terraform, а також методи забезпечення безпеки, включаючи шифрування, аутентифікацію та авторизацію на основі OAuth 2.0, OpenID Connect та JWT.Особливу увагу приділено оптимізації продуктивності системи шляхом впровадження кешування, балансування навантаження та інструментів моніторингу (Prometheus, Grafana). У роботі також представлено рекомендації щодо тестування продуктивності мікросервісних додатків та забезпечення їхньої стійкості до навантаження. Результатом дослідження є набір рекомендацій та практичних рішень для створення надійних, безпечних і масштабованих веб-додатків.

КЛЮЧОВІ СЛОВА: МІКРОСЕРВІСИ, РОЗГОРТАННЯ, ВЕБ-ДОДАТКИ, АВТОМАТИЗАЦІЯ, БЕЗПЕКА, ПРОДУКТИВНІСТЬ.

ABSTRACT

Text part of the master's qualification work: 82 pages, 5 pictures, 1 tables, 20 sources.

The purpose of the work – conduct an in-depth analysis of existing methods for deploying microservice architecture, including containerization, orchestration, and automation, as well as to explore modern approaches to ensuring its security.

Object of research – is microservice architecture, which represents a modern approach to developing web applications by dividing them into independent services, each performing specific functions.

Subject of research – is the methods for deploying microservice architecture, including an analysis of containerization, orchestration, and service integration, as well as modern approaches to ensuring security, specifically the application of authentication, authorization, and encryption to protect data and prevent potential threats.

Summary of the work:

The master's thesis focuses on developing methods for deploying microservice architecture and ensuring security for web applications. The work highlights the relevance of using microservice architecture in the context of increasing software system complexity and modern approaches to its implementation. It explores deployment automation using tools such as Docker, Kubernetes, Ansible, and Terraform, along with security methods including encryption, authentication, and authorization based on OAuth 2.0, OpenID Connect, and JWT. Particular attention is given to optimizing system performance through caching, load balancing, and monitoring tools (Prometheus, Grafana). The work also presents recommendations for performance testing of microservice applications and ensuring their resilience to load. The result of the study is a set of recommendations and practical solutions for creating reliable, secure, and scalable web applications.

KEYWORDS: MICROSERVICES, DEPLOYMENT, WEB APPLICATIONS, AUTOMATION, SECURITY, PERFORMANCE

ЗМІСТ

ВСТУП

Сучасний розвиток інформаційних технологій та зростаючі потреби бізнесу вимагають створення гнучких, масштабованих та безпечних веб-додатків. Інтенсивний розвиток цифровізації та збільшення кількості онлайн-сервісів призводить до зростання попиту на ефективні рішення для управління інформаційними системами. Традиційна монолітна архітектура, яка довгий час була стандартом у розробці програмного забезпечення, демонструє низку недоліків. Серед них: складність адаптації до нових вимог, обмеженість масштабування, труднощі з інтеграцією нових компонентів та високі ризики відмови всієї системи через помилку в одній її частині. У відповідь на ці виклики з'явилася мікросервісна архітектура, яка стала революційним підходом до проєктування програмних продуктів. Основна ідея цієї архітектури полягає в поділі додатка на набір незалежних сервісів, кожен з яких виконує одну конкретну функцію і може бути розроблений, протестований, розгорнутий і масштабований окремо. Такий підхід забезпечує вищу гнучкість у розробці, можливість швидкого впровадження нових функцій і покращення ефективності розподілених команд розробників. Мікросервісна архітектура також сприяє інтеграції з новими технологіями, дозволяючи системам еволюціонувати разом із бізнес-потребами, що постійно змінюються. Разом із перевагами мікросервісної архітектури зростає необхідність глибокого вивчення методів її розгортання та забезпечення безпеки, адже ця модель супроводжується низкою технічних викликів. Одним із головних аспектів є інтеграція сервісів, що вимагає ефективного управління інтерфейсами API, обробки великого обсягу даних і забезпечення високої продуктивності системи. Управління інфраструктурою в умовах мікросервісної архітектури також стає складнішим через необхідність використання сучасних оркестраційних платформ, таких як Kubernetes, що забезпечують автоматизацію розгортання та моніторинг системи.

Ще одним критичним аспектом є забезпечення безпеки мікросервісів, яка вимагає впровадження комплексного підходу. Кожен окремий сервіс стає потенційною точкою атаки, тому необхідно впроваджувати методи аутентифікації, авторизації, шифрування даних і засоби для виявлення та запобігання загрозам. Особлива увага приділяється захисту API, оскільки саме через нього здійснюється більшість комунікацій між сервісами. Використання API Gateway дозволяє централізовано управляти потоками даних, захищати їх від небажаних доступів і реалізовувати контроль доступу до сервісів.

Об'єктом дослідження є мікросервісна архітектура, яка представляє собою сучасний підхід до створення веб-додатків шляхом поділу на незалежні сервіси, кожен з яких виконує конкретні функції. Такий підхід дозволяє оптимізувати розробку, масштабування та експлуатацію програмного забезпечення, знижуючи ризики збоїв та полегшуючи адаптацію до нових бізнес-потреб. Крім того, мікросервісна архітектура стає важливим інструментом для забезпечення ефективної роботи великих систем, які вимагають інтеграції з різноманітними

сторонніми сервісами та технологіями. Особливий акцент у цьому дослідженні зроблено на її застосуванні в контексті безпеки даних, зокрема захисту від кібератак, що набувають усе більшої актуальності в сучасному цифровому середовищі. Завдяки своїй гнучкості та адаптивності, мікросервісна архітектура дедалі частіше використовується в різних галузях, таких як електронна комерція, фінансові послуги, медична інформатика та багато інших, що робить її ключовим об'єктом для вивчення та вдосконалення.

Предметом дослідження є методи розгортання мікросервісної архітектури, що включають аналіз контейнеризації, оркестрації та інтеграції сервісів, а також сучасні підходи до забезпечення безпеки, зокрема застосування аутентифікації, авторизації та шифрування для захисту даних і запобігання можливим загрозам.

Метою дослідження є проведення глибокого аналізу існуючих методів розгортання мікросервісної архітектури, включаючи контейнери, оркестрацію та автоматизацію, а також дослідження сучасних підходів до забезпечення її безпеки. Особлива увага приділяється питанням аутентифікації, авторизації, моніторингу вразливостей і використання шифрування для захисту переданих даних. Мета також охоплює визначення ефективності цих методів у різних середовищах розгортання, зокрема у великих веб-додатках з інтеграцією сторонніх сервісів, щоб надати рекомендації щодо оптимального вибору рішень.

Результати дослідження можуть бути корисними для розробників програмного забезпечення, системних архітекторів, фахівців із забезпечення безпеки та IT-інженерів, які працюють над створенням та вдосконаленням веб-додатків. Отримані висновки допоможуть оптимізувати процес розгортання мікросервісної архітектури, враховуючи специфіку різних платформ і середовищ. Крім того, дослідження пропонує детальний огляд методів захисту, які дозволять зменшити ризики кібератак, а також покращити управління інфраструктурою і масштабованість систем. Це забезпечує інтеграцію сучасних технологій у різноманітні галузі, від фінансового сектора до охорони здоров’я, сприяючи розвитку більш надійних і ефективних рішень.

1 АНАЛІЗ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

1.1 Сучасні тенденції у розробці веб-додатків

У сучасному світі інформаційні технології стрімко змінюють способи взаємодії людей, бізнесу та організацій. Зокрема, розробка веб-додатків зазнає постійної трансформації, обумовленої постійним зростанням обсягу даних, підвищенням вимог до надійності систем та необхідністю забезпечення високої продуктивності, гнучкості й безпеки. Монолітні підходи до створення програмного забезпечення, які довгий час домінували, поступово поступаються місцем мікросервісним і хмарним моделям, які дозволяють ефективніше управляти складними системами та скорочувати витрати на їх підтримку. Особливо актуальним є використання інноваційних методологій, таких як DevOps і Agile, які забезпечують тісну взаємодію між командами розробників і експлуатації. Це сприяє швидшому впровадженню змін, зниженню ризиків помилок та забезпеченню стабільності роботи додатків. Розвиток технологій контейнеризації, зокрема Docker, став фундаментом для розгортання додатків у середовищах, які забезпечують швидке масштабування та гнучкість. Впровадження хмарних технологій значно змінило підхід до інфраструктури веб-додатків. Завдяки платформам, таким як AWS, Google Cloud та Microsoft Azure, з’явилася можливість використовувати модель "інфраструктури як послуги" (IaaS), що дозволяє бізнесу оптимізувати витрати на апаратне забезпечення та концентруватися на розробці інноваційних продуктів. Сучасні веб-додатки також зіштовхуються з викликами у сфері безпеки, оскільки зростає кількість кіберзагроз, які стають дедалі складнішими та різноманітнішими. До найбільш поширених загроз належать атаки типу DDoS, фішинг, ін'єкції SQL, атаки на рівні API, а також вразливості, пов'язані з людським фактором, наприклад, використання слабких паролів або небезпечних каналів зв'язку. Ці фактори змушують розробників приділяти ще більше уваги розробці комплексних систем захисту, які включають багаторівневу аутентифікацію (наприклад, двофакторну або з використанням біометрії),

ефективне шифрування даних у стані спокою та під час передачі, а також використання сертифікатів SSL/TLS. Автоматизовані інструменти моніторингу безпеки, такі як системи виявлення вторгнень (IDS) та системи управління інформацією та подіями безпеки (SIEM), допомагають у реальному часі виявляти потенційні загрози та реагувати на них. Інтеграція штучного інтелекту та машинного навчання дозволяє аналізувати величезні обсяги даних для виявлення аномалій у поведінці користувачів чи системи, що може свідчити про спробу атаки. Наприклад, системи на базі AI здатні прогнозувати атаки, базуючись на історичних даних, і пропонувати превентивні заходи. Крім технічних аспектів, важливу роль відіграє створення політик інформаційної безпеки в організаціях, які включають регулярне навчання співробітників, аудит безпеки та оновлення програмного забезпечення для закриття відомих вразливостей. Тому тільки поєднання сучасних технологій, інноваційних методів розробки та управління безпекою дозволяє створювати ефективні, масштабовані та надійні веб-додатки, які відповідають сучасним викликам та вимогам.

Основні тенденції у розробці веб-додатків включають

Перехід на мікросервісну архітектуру. Цей підхід забезпечує поділ програмного забезпечення на незалежні сервіси, кожен з яких виконує свою унікальну функцію. Це дозволяє ефективно масштабувати систему, впроваджувати нові функціональні можливості без значних змін у наявній структурі та забезпечувати високу гнучкість у роботі з різними компонентами. Кожен сервіс є автономним, що спрощує не лише його розгортання та тестування, але й дозволяє різним командам одночасно працювати над окремими частинами системи, не створюючи залежностей. Мікросервісна архітектура також сприяє поліпшенню стійкості системи: збої в одному сервісі не впливають на загальну працездатність, оскільки інші сервіси продовжують працювати. Це особливо важливо для великих високонавантажених систем, таких як інтернет-магазини, фінансові платформи чи сервіси потокового відео. Завдяки цим перевагам, мікросервісна архітектура стала стандартом для створення сучасних масштабованих та надійних веб-додатків.

Контейнеризація та оркестрація. Використання Docker, Kubernetes та інших платформ дозволяє автоматизувати процеси розгортання та забезпечити стабільну роботу додатків навіть у складних середовищах, де необхідно дотримуватися високих вимог до масштабованості, доступності та відмовостійкості. Контейнеризація дозволяє ізолювати кожен мікросервіс у власному середовищі, що забезпечує узгодженість роботи незалежно від інфраструктури, на якій він розгорнутий. Оркестраційні системи, такі як Kubernetes, додають можливість автоматичного балансування навантаження, відстеження стану сервісів і автоматичного відновлення роботи у випадку збоїв. Крім того, ці технології сприяють оптимізації використання ресурсів, знижують витрати на обслуговування та дозволяють зосередитися на розробці нового функціоналу замість управління інфраструктурою. Такі можливості роблять контейнеризацію та оркестрацію ключовими інструментами для створення сучасних веб-додатків.

Хмарні технології. Веб-додатки дедалі частіше інтегруються з хмарними сервісами, такими як AWS, Azure чи Google Cloud, що сприяє їх глобальній доступності та підвищенню продуктивності. Хмарні платформи надають широкий спектр можливостей, включаючи автоматичне масштабування ресурсів, гнучке управління інфраструктурою та використання технологій штучного інтелекту для аналітики та автоматизації. Завдяки моделі "плати за використання", хмарні сервіси дозволяють компаніям оптимізувати витрати на інфраструктуру, забезпечуючи при цьому високу продуктивність і доступність систем. Хмарні технології також значно підвищують рівень резервування даних та відмовостійкості завдяки багаторегіональним копіям і автоматичному відновленню систем після збоїв. Такі функції є критично важливими для сучасних веб-додатків, які обслуговують мільйони користувачів у режимі реального часу. Наприклад, сервіси Google Cloud дозволяють легко інтегрувати штучний інтелект для обробки великих даних, що допомагає створювати прогнози та адаптуватися до змінних умов ринку. AWS, у свою чергу, пропонує потужні інструменти для машинного навчання та

автоматизації інфраструктури, забезпечуючи розробникам свободу у створенні інноваційних рішень. Завдяки хмарним технологіям компанії можуть швидше реагувати на зміни у вимогах бізнесу, забезпечуючи конкурентоспроможність та адаптивність у сучасному цифровому середовищі. Інтеграція хмарних сервісів із процесами DevOps дозволяє прискорити цикл розробки та доставки програмного забезпечення, що є ключовим фактором успіху на ринку.

Впровадження DevOps. Інтеграція процесів розробки та експлуатації створює основу для безперервного вдосконалення програмного забезпечення через автоматизацію, тісну співпрацю між командами та спрощення процесів тестування та впровадження. DevOps дозволяє організаціям швидко адаптуватися до змін у вимогах бізнесу, зменшити час виходу на ринок нових продуктів і послуг, а також покращити загальну якість програмного забезпечення завдяки зниженню кількості помилок. Основними компонентами DevOps є CI/CD (Continuous Integration/Continuous Deployment), автоматизація тестування, моніторинг продуктивності та інтеграція з хмарними сервісами, що забезпечують безперебійну роботу додатків навіть у середовищах з високим навантаженням. Завдяки DevOps, організації можуть досягти більшої гнучкості, продуктивності та надійності своїх систем, що є ключовими вимогами сучасного ринку.

Фокус на безпеку. З огляду на зростаючу кількість кіберзагроз, сучасні веб-додатки впроваджують комплексні механізми захисту на рівні архітектури, які включають багаторівневу аутентифікацію, ефективне шифрування даних та постійний моніторинг уразливостей. Важливим елементом є використання API Gateway для централізованого контролю доступу та захисту потоків даних, а також інтеграція з системами виявлення загроз, такими як IDS та SIEM. Крім цього, застосовуються сучасні стандарти безпеки, наприклад, OAuth 2.0 та OpenID Connect, які забезпечують безпечну авторизацію користувачів. У поєднанні з регулярним аудитом безпеки, впровадженням політик оновлення та автоматизованими інструментами для аналізу загроз, це дозволяє ефективно протистояти сучасним викликам у сфері кібербезпеки.

Дані тенденції зумовлені необхідністю відповідати вимогам сучасного бізнесу, що постійно змінюються, а також очікуванням користувачів, які прагнуть швидких, надійних і безпечних рішень. Високий рівень конкуренції на ринку веб-додатків стимулює компанії постійно вдосконалювати свої рішення, інтегруючи сучасні технології та адаптуючись до нових викликів. Одним із ключових аспектів є здатність веб-додатків забезпечувати безперервність роботи навіть за умов пікових навантажень або в разі збоїв у роботі окремих компонентів. Це стає можливим завдяки мікросервісній архітектурі, яка підтримує гнучкість і стійкість систем. Крім того, зростаюча інтеграція хмарних платформ дозволяє компаніям економити ресурси на утриманні власної інфраструктури, спрямовуючи більше зусиль на інновації. З іншого боку, зростаючі кіберзагрози вимагають від розробників впровадження складних систем захисту, які можуть адаптуватися до нових типів атак. У цьому контексті особливе значення мають алгоритми штучного інтелекту, які допомагають у прогнозуванні та запобіганні загрозам.

1.2 Основні принципи побудови мікросервісної архітектури

Мікросервісна архітектура є сучасним підходом до розробки програмного забезпечення, який базується на поділі системи на невеликі, незалежні сервіси. Цей підхід надає змогу організаціям створювати високоефективні, адаптивні та масштабовані системи, які легко адаптуються до змін у бізнес-вимогах. Завдяки мікросервісній архітектурі розробники можуть швидко впроваджувати нові функціональні можливості, мінімізуючи ризики збоїв та проблем, що виникають через взаємозалежності компонентів. Кожен мікросервіс має чітко визначену відповідальність за виконання конкретного набору функцій. Це забезпечує зручність у підтримці та розвитку програмного забезпечення. Наприклад, якщо виникає необхідність у зміні або оновленні певної функції, розробники можуть зосередитися лише на відповідному сервісі, не порушуючи роботу всієї системи. Такий підхід також дозволяє організаціям масштабувати окремі сервіси залежно від їх завантаження. Наприклад, сервіси, які відповідають за процесинг транзакцій,

можуть бути масштабовані незалежно від інших, менш завантажених компонентів. Це не лише оптимізує використання ресурсів, але й знижує витрати на інфраструктуру.

Мікросервісна архітектура надає можливість використовувати різноманітні технології та інструменти для кожного сервісу, обираючи оптимальні рішення залежно від конкретних завдань. Це відкриває широкі можливості для інтеграції інноваційних рішень, таких як штучний інтелект, машинне навчання чи обробка великих даних. Не лише покращує якість програмного забезпечення, але й робить його більш стійким до змін у вимогах та здатним ефективно реагувати на виклики сучасного бізнес-середовища. Основні принципи побудови мікросервісної архітектури включають:

Декомпозиція за функціональністю. Кожен мікросервіс має чітко визначену роль у системі і виконує лише одну функцію. Це забезпечує простоту управління, тестування і розширення системи. Крім того, декомпозиція дозволяє залучати вузькоспеціалізовані команди до роботи над окремими мікросервісами, що покращує загальну якість продукту.

Автономність сервісів. Мікросервіси працюють незалежно один від одного, що дозволяє уникнути проблем із взаємозалежністю. Це сприяє стійкості системи до збоїв і полегшує масштабування. Наприклад, у випадку збою одного сервісу, інші компоненти можуть продовжувати виконувати свої функції без переривання роботи всієї системи.

Комунікація через API. Взаємодія між сервісами здійснюється через стандартизовані інтерфейси API, що дозволяє легко інтегрувати нові компоненти і забезпечує сумісність. Використання RESTful або gRPC API сприяє більш ефективній передачі даних між сервісами, знижуючи затримки та спрощуючи розробку.

Децентралізоване управління даними. Кожен мікросервіс має власну базу даних або доступ до сегмента даних, що знижує ризик конфліктів і спрощує управління даними. Цей підхід також дозволяє вибирати різні технології баз даних

для різних сервісів, враховуючи їх унікальні вимоги до продуктивності та зберігання даних.

Контейнеризація та оркестрація. Для розгортання і управління мікросервісами використовуються контейнери, такі як Docker, а оркестраційні платформи, як-от Kubernetes, допомагають автоматизувати процеси. Контейнеризація дозволяє ізолювати кожен мікросервіс у власному середовищі, що спрощує його управління, тестування та масштабування.

Безперервна інтеграція і доставка (CI/CD). Цей підхід дозволяє автоматизувати процеси тестування, розгортання і оновлення мікросервісів, забезпечуючи швидкість і якість розробки. CI/CD забезпечує зменшення ризиків помилок у процесі оновлення та спрощує додавання нових функцій.

Моніторинг і логування. Постійний моніторинг роботи мікросервісів та аналіз журналів дозволяє виявляти потенційні проблеми на ранніх етапах і забезпечує високу надійність системи. Використання таких інструментів, як Prometheus або Grafana, сприяє прозорості у відстеженні стану системи та швидкому виявленню аномалій.

Висока масштабованість. Архітектура мікросервісів дозволяє масштабувати окремі компоненти системи відповідно до потреб бізнесу, що оптимізує використання ресурсів. Наприклад, сервіси з високим навантаженням можуть бути масштабовані окремо від інших, що зменшує витрати.

Гнучкість у виборі технологій. Різні мікросервіси можуть бути розроблені на різних мовах програмування або використовувати різні фреймворки, що дозволяє вибирати оптимальні інструменти для кожної задачі.

Дотримання цих принципів є запорукою успішного впровадження мікросервісної архітектури, яка відповідає вимогам сучасних високонавантажених систем. Вона забезпечує стабільність завдяки ізоляції кожного сервісу, що зменшує ризик поширення помилок. Кожен мікросервіс функціонує автономно, що дозволяє зменшити вплив потенційних збоїв на загальну систему. Безпека досягається через впровадження багаторівневих механізмів, таких як сучасні протоколи передачі

даних (HTTPS, TLS), інтеграція методів аутентифікації OAuth, OpenID Connect, а також реалізація міжсервісного шифрування з використанням SSL-сертифікатів. Завдяки цьому мікросервісна архітектура забезпечує конфіденційність, цілісність та доступність даних. Адаптивність до змінних умов підтримується можливістю горизонтального та вертикального масштабування сервісів. Наприклад, сервіси з високим навантаженням можуть бути швидко розгорнуті у декількох екземплярах для обробки пікових навантажень, тоді як менш важливі компоненти працюють у стандартному режимі, що дозволяє економити ресурси. Крім того, використання таких технологій, як контейнеризація (Docker) та оркестрація (Kubernetes), сприяє підвищенню адаптивності системи, дозволяючи швидко оновлювати програмне забезпечення та зменшувати час простою. Інтеграція нових технологій є ключовою перевагою мікросервісної архітектури. Завдяки модульності системи, організації можуть легко впроваджувати нові фреймворки, бібліотеки та сервіси, що відповідають їх бізнес-цілям. Це дозволяє швидше реагувати на зміни ринку та підвищувати конкурентоспроможність.

У результаті мікросервісна архітектура стає важливим інструментом для створення надійних, масштабованих та ефективних IT-рішень. Вона дозволяє організаціям адаптуватися до зростаючих вимог бізнесу, оптимізувати операційні процеси та забезпечувати найвищий рівень обслуговування для своїх клієнтів..

1.3 Переваги та недоліки використання мікросервісів

Мікросервісна архітектура є популярним підходом до розробки програмного забезпечення завдяки її численним перевагам, які дозволяють створювати гнучкі, масштабовані та стійкі до збоїв системи. Вона базується на модульності, що забезпечує незалежність компонентів та спрощує їхню інтеграцію. Мікросервіси стали стандартом у розробці складних додатків завдяки можливості швидко адаптуватися до змін бізнес-потреб, підтримувати високий рівень доступності та забезпечувати гнучкість у виборі технологій. Однак мікросервісна архітектура має й певні виклики, які виникають через розподілену природу системи. Управління

численними сервісами вимагає використання складних інструментів оркестрації, таких як Kubernetes, що може бути складним для невеликих команд. Додатково, інтенсивна взаємодія між сервісами через мережу створює потребу в оптимізації протоколів зв'язку, таких як HTTP/2 або gRPC, для зменшення затримок та підвищення продуктивності. Успішне впровадження мікросервісної архітектури залежить від правильного вибору технологій, ефективного дизайну інтерфейсів API, забезпечення безпеки міжсервісних комунікацій і впровадження системи моніторингу для відстеження стану сервісів у реальному часі. Лише ретельне планування та врахування потреб проєкту дозволяють досягти максимального ефекту від використання мікросервісної архітектури.

Переваги використання мікросервісів

Масштабованість. Кожен мікросервіс може бути масштабований незалежно, залежно від навантаження. Наприклад, якщо сервіс обробки платежів починає отримувати значно більше запитів через акційну кампанію, лише цей компонент може бути масштабований, залишаючи інші частини системи незмінними. Такий підхід зменшує витрати на ресурси та підвищує ефективність роботи системи в цілому.

Гнучкість у розробці. Завдяки розподілу системи на невеликі сервіси, команди розробників можуть працювати паралельно, незалежно одна від одної. Це означає, що декілька команд можуть одночасно займатися розробкою нових функцій, тестуванням або усуненням помилок, що значно скорочує час розробки продукту та дозволяє швидше реагувати на бізнес-запити.

Стійкість до збоїв. Збої в одному сервісі не впливають на роботу інших, що підвищує загальну надійність системи. Наприклад, якщо сервіс обробки зображень виходить з ладу, це не зупинить роботу сервісів, які відповідають за авторизацію чи перегляд контенту.

Легкість оновлень. Мікросервіси дозволяють оновлювати окремі компоненти системи без необхідності перевипуску всього додатку. Наприклад,

виправлення помилки в одному сервісі не потребує зупинки всіх інших компонентів, що зменшує ризики і спрощує процес оновлення.

Технологічна різноманітність. Кожен сервіс може бути реалізований з використанням різних мов програмування, фреймворків або баз даних. Наприклад, сервіс обробки великих даних може використовувати Hadoop, тоді як сервіс взаємодії з клієнтами може базуватися на більш легкому Node.js. Це дозволяє оптимізувати роботу кожного компонента.

Полегшення інтеграції. Завдяки стандартизованим інтерфейсам API мікросервіси легко інтегруються з іншими системами. Наприклад, інтеграція з платіжними шлюзами або сторонніми сервісами аналітики стає швидшою та ефективнішою.

Недоліки використання мікросервісів

Складність управління. Зі збільшенням кількості сервісів зростає складність їхнього адміністрування, моніторингу та забезпечення взаємодії між ними. Для великих систем необхідні спеціальні інструменти, такі як Kubernetes для оркестрації, і комплексні системи моніторингу на кшталт Prometheus або Grafana.

Витрати на інфраструктуру. Для роботи кожного сервісу потрібні окремі ресурси, що може збільшити витрати, особливо у невеликих проєктах. Наприклад, кожен контейнер може вимагати окремого серверного ресурсу, навіть якщо його навантаження незначне.

Проблеми з комунікацією. Інтенсивна взаємодія між сервісами через мережу може призводити до затримок. Наприклад, високочастотні запити між сервісами можуть створювати додаткове навантаження на мережеву інфраструктуру, що вимагає оптимізації протоколів комунікації.

Підвищена складність тестування. Перевірка роботи всіх мікросервісів у їхній взаємодії потребує складніших підходів до тестування. Наприклад, розробникам доводиться моделювати взаємодію сотень сервісів, щоб забезпечити стабільність роботи системи.

Безпека. Розподіленість системи вимагає додаткових заходів безпеки, таких як шифрування міжсервісних комунікацій, контроль доступу та моніторинг загроз. Наприклад, необхідно забезпечувати перевірку кожного запиту, що проходить через API Gateway.

Проблеми узгодженості даних. Розподіл баз даних між сервісами може ускладнити підтримку узгодженості даних. Наприклад, у випадках оновлення інформації одночасно в кількох сервісах можуть виникати конфлікти або затримки, які важко синхронізувати.

Мікросервісна архітектура забезпечує значні переваги для розробки сучасного програмного забезпечення. Вона сприяє створенню гнучких, масштабованих і адаптивних систем, що відповідають вимогам сучасного бізнесу. Гнучкість дозволяє компаніям швидко адаптуватися до змін у ринкових умовах, масштабованість забезпечує стабільну роботу навіть під час пікових навантажень, а адаптивність дозволяє ефективно впроваджувати нові технології без значних збоїв у роботі системи. Проте впровадження мікросервісної архітектури є складним процесом, який вимагає ретельного планування та використання сучасних інструментів. Зокрема, необхідно правильно вибрати інструменти для управління сервісами, такі як Kubernetes або Docker Swarm, які дозволяють автоматизувати процеси оркестрації та контейнеризації. Для забезпечення безпеки потрібне впровадження стандартів шифрування даних, таких як TLS, а також побудова системи контролю доступу з використанням OAuth чи OpenID Connect. Мікросервіси вимагають створення ефективної інфраструктури моніторингу. Це включає використання інструментів на зразок Prometheus або Grafana для відстеження продуктивності та стану сервісів, а також впровадження систем логування, таких як ELK Stack, для аналізу журналів і виявлення проблем.

Завдяки правильно розробленій стратегії впровадження, мікросервісна архітектура може стати основою для побудови складних, високонавантажених систем, які відповідають сучасним вимогам бізнесу та технологічним трендам.

1.4 Приклади застосування мікросервісної архітектури у реальних проєктах

Мікросервісна архітектура знаходить широке застосування у сучасних високотехнологічних проєктах, що охоплюють різні галузі. Одним із найяскравіших прикладів є використання мікросервісів у великих e-commerce платформах, таких як Amazon. Завдяки розподіленій архітектурі, Amazon забезпечує обробку мільйонів замовлень щодня з високою швидкістю та надійністю. Наприклад, під час пікових навантажень, таких як "Чорна п’ятниця", платформа може динамічно масштабувати сервіси обробки замовлень або пошукові механізми, щоб відповідати зростаючому попиту. Кожна частина платформи, така як обробка платежів, управління складом або персоналізовані рекомендації, реалізована як окремий мікросервіс із чіткими межами відповідальності. Такий підхід не лише знижує витрати на інфраструктуру, але й дозволяє швидко впроваджувати нові функції, тестувати їх у реальному часі та оперативно реагувати на проблеми.

Іншим прикладом є стрімінговий сервіс Netflix, який використовує мікросервісну архітектуру для забезпечення безперебійного доступу до контенту для мільйонів користувачів у всьому світі. Завдяки мікросервісам, Netflix підтримує високу продуктивність і надійність системи. Кожен мікросервіс виконує свою конкретну роль: обробка запитів на трансляцію контенту, управління користувачами, створення персоналізованих рекомендацій, аналіз поведінки глядачів та оптимізація потоків даних. Наприклад, система рекомендацій використовує аналіз великих даних для передбачення вподобань користувачів, що підвищує їх залученість. Мікросервіси Netflix також відповідають за забезпечення географічної доступності контенту через інтеграцію з Content Delivery Networks (CDN), що дозволяє мінімізувати затримки для користувачів у різних регіонах світу. Netflix активно використовує автоматизовані системи моніторингу, такі як "Chaos Monkey", для перевірки стійкості мікросервісів до збоїв. Це допомагає запобігти можливим проблемам у роботі платформи та гарантує безперервний

доступ до контенту навіть за умов високого навантаження, наприклад, під час прем’єри популярних серіалів.

Фінансовий сектор також активно використовує мікросервісну архітектуру, демонструючи її здатність відповідати вимогам сучасних високонавантажених систем. Наприклад, PayPal застосовує мікросервіси для обробки мільйонів транзакцій щодня, забезпечуючи безперервну роботу навіть у пікові періоди, такі як святкові дні чи глобальні розпродажі. Кожен мікросервіс у системі PayPal відповідає за виконання специфічного завдання, такого як управління обліковими записами користувачів, перевірка транзакцій на відповідність фінансовим регуляціям, управління шахрайством або інтеграція з іншими платіжними платформами. Це забезпечує гнучкість і адаптивність платформи до нових фінансових продуктів і послуг. Завдяки мікросервісній архітектурі PayPal може швидко впроваджувати інновації, наприклад, додавати підтримку криптовалют чи нових способів оплати. Система забезпечує високий рівень безпеки завдяки шифруванню даних, багаторівневій аутентифікації та використанню машинного навчання для виявлення шахрайства у реальному часі. Мікросервіси також дозволяють оптимізувати інфраструктуру, динамічно розподіляючи ресурси між компонентами, що підвищує продуктивність та знижує експлуатаційні витрати.

У галузі охорони здоров’я мікросервісна архітектура допомагає створювати інноваційні рішення для управління медичними даними, що є критично важливим для покращення якості медичних послуг і оптимізації ресурсів. Системи електронних медичних записів (EMR) використовують мікросервіси для забезпечення децентралізованого доступу до даних пацієнтів, зберігання історій хвороб, обробки результатів лабораторних аналізів, а також координації роботи між різними підрозділами медичних закладів. Наприклад, мікросервіс, відповідальний за розклад прийому лікарів, може автоматично синхронізувати дані з іншими сервісами, такими як реєстратура або лабораторія.

Інтеграція мікросервісів у сфері охорони здоров’я дозволяє зменшити ризик втрати важливої інформації завдяки резервуванню даних та підвищує ефективність

роботи медичних працівників через автоматизацію рутинних процесів. Наприклад, системи підтримки прийняття клінічних рішень, засновані на мікросервісах, можуть аналізувати історії хвороб пацієнтів і пропонувати лікарям рекомендації щодо діагностики чи лікування, базуючись на останніх дослідженнях. Завдяки мікросервісній архітектурі стало можливим створення телемедичних платформ, які забезпечують дистанційну консультацію лікарів, моніторинг стану здоров'я пацієнтів за допомогою носимих пристроїв та інтеграцію з хмарними сервісами для обробки великих обсягів даних. Це значно покращує доступність медичних послуг для пацієнтів у віддалених регіонах та оптимізує роботу медичних установ.

Мікросервісна архітектура довела свою ефективність у численних проєктах, що вимагають високої продуктивності, гнучкості та масштабованості. Її застосування дозволяє організаціям впроваджувати інноваційні рішення, адаптуватися до змін та підтримувати високу якість послуг для користувачів.

1.5 Висновки по розділу

Мікросервісна архітектура є потужним підходом до створення сучасних інформаційних систем, який пропонує високий рівень гнучкості, масштабованості та надійності. У першому розділі було розглянуто основні аспекти цієї архітектури, включаючи її принципи, переваги та виклики. Серед ключових переваг мікросервісної архітектури можна виділити її здатність підтримувати незалежність компонентів, що забезпечує швидкість розробки, простоту оновлення та стійкість до збоїв. Також підхід дозволяє інтегрувати різноманітні технології та забезпечує гнучкість у виборі інструментів для кожного сервісу.

Розглянуті реальні приклади, такі як Amazon, Netflix, PayPal та інші, продемонстрували, як мікросервіси допомагають організаціям досягати високої продуктивності та адаптуватися до змінних умов ринку. Водночас мікросервісна архітектура вимагає ретельного планування, особливо у сферах управління складністю системи, забезпечення узгодженості даних та впровадження систем моніторингу й безпеки.

Мікросервісна архітектура є ефективним рішенням для створення високонавантажених і складних систем, проте її впровадження повинно супроводжуватися виваженим підходом до проектування та вибору інструментів. Цей розділ закладає основу для подальшого дослідження методів її розгортання та забезпечення безпеки.

2 МЕТОДИ РОЗГОРТАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1 Використання контейнеризації: Docker та OCI

Контейнеризація є ключовим методом у розгортанні мікросервісної архітектури, що забезпечує ізоляцію, гнучкість і масштабованість сервісів. Ця технологія дозволяє запускати кожен мікросервіс у своєму ізольованому середовищі, де всі необхідні залежності, бібліотеки та конфігурації вже включені. Це створює стандартизоване середовище виконання, незалежне від апаратного забезпечення чи операційної системи хосту. Завдяки цьому команди розробників можуть розробляти, тестувати та розгортати додатки швидше, оскільки відсутність залежності від конкретного середовища знижує ймовірність помилок. Крім того, контейнеризація сприяє значному спрощенню процесів CI/CD, дозволяючи інтегрувати та доставляти нові версії програмного забезпечення з високою швидкістю та передбачуваністю. Таким чином, використання контейнерів забезпечує узгодженість роботи мікросервісів у різних середовищах, включаючи розробку, тестування та продуктивне розгортання.

Одним із найпоширеніших інструментів для контейнеризації є Docker (рис.2.1). Docker надає можливість створювати, розгортати та управляти контейнерами з мінімальними витратами ресурсів. Його ключова перевага полягає у зручності створення стандартних контейнерних образів, які включають всі необхідні бібліотеки, залежності та конфігурації для виконання певного мікросервісу. Це значно спрощує процес розгортання, роблячи його швидким і передбачуваним. Docker також забезпечує тісну інтеграцію з сучасними системами оркестрації, такими як Kubernetes, які дозволяють автоматизувати управління тисячами контейнерів одночасно. Це особливо важливо для великих систем, де потрібно підтримувати стабільну роботу під час пікових навантажень. Крім того, Docker надає можливості для версіонування образів, що спрощує процес оновлення програмного забезпечення та відкату до попередніх версій у разі виявлення помилок.

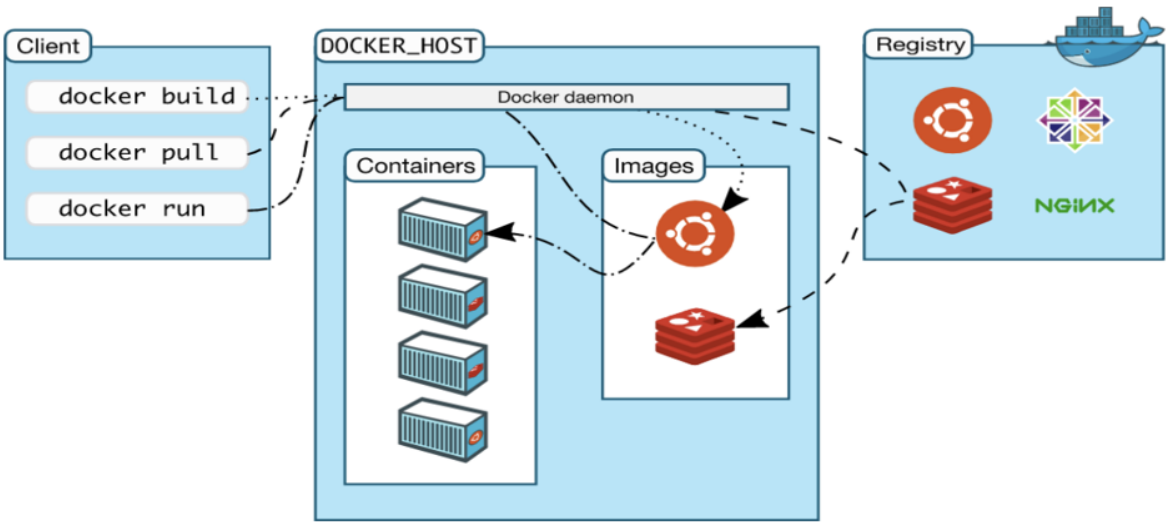

Рисунок 2.1 – Docker

Однією з унікальних особливостей Docker є його екосистема, яка включає Docker Hub — хмарний реєстр контейнерних образів. Це дозволяє розробникам швидко обмінюватися образами, використовувати готові рішення або створювати свої власні на основі доступних шаблонів. Завдяки цьому команди можуть значно прискорити розробку та розгортання нових мікросервісів, скорочуючи час на налаштування середовища.

Ще однією важливою технологією є Open Container Initiative (OCI) (рис.2.2). Це стандарт для контейнерів, що встановлює чіткі правила для створення форматів контейнерних образів і середовищ виконання. Головною перевагою OCI є забезпечення сумісності між різними платформами контейнеризації, такими як Docker, Podman чи CRI-O, завдяки чому компанії можуть уникнути прив'язки до одного постачальника технологій. Стандарти OCI включають специфікації для контейнерних образів (Image Specification) та середовищ виконання (Runtime Specification), що дозволяє використовувати одні й ті самі контейнери на різних платформах без необхідності адаптації чи модифікацій. Наприклад, організація може розробляти додатки з використанням Docker, а розгортати їх на Kubernetes із застосуванням CRI-O без порушення функціональності. Завдяки стандартизації OCI підприємства отримують гнучкість у виборі інструментів, що відповідають їхнім потребам. Це також сприяє зменшенню витрат на адаптацію та інтеграцію

нових технологій. Використання OCI дозволяє зберігати високу продуктивність систем, мінімізуючи ризики, пов'язані зі зміною постачальників або платформ контейнеризації. Таким чином, ця ініціатива забезпечує стабільність та передбачуваність процесів у межах мікросервісної архітектури.

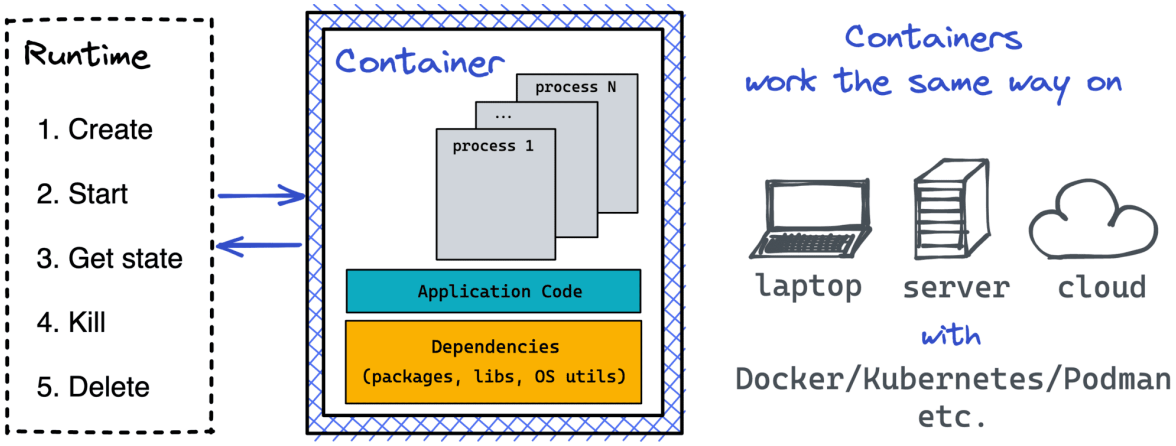

Рисунок 2.2 – Open Container Initiative (OCI)

Контейнеризація також значно підвищує рівень безпеки мікросервісів завдяки декільком критично важливим механізмам. Насамперед, ізоляція контейнерів забезпечує чіткий поділ середовищ виконання, що унеможливлює прямий вплив одного мікросервісу на інший навіть у разі компрометації. Це особливо важливо у високонавантажених системах, де безпека є ключовим аспектом.

Сучасні платформи контейнеризації пропонують розширені засоби безпеки, серед яких:

Обмеження ресурсів: Контейнери можуть бути налаштовані з обмеженням використання процесорного часу, пам’яті та інших ресурсів, що запобігає їх перевантаженню.

Контроль доступу: Використання механізмів Role-Based Access Control (RBAC) дозволяє забезпечити доступ до контейнерів лише авторизованим користувачам і процесам.

Шифрування даних: Секрети, такі як ключі API, паролі та сертифікати, можуть зберігатися в захищених сховищах і бути доступними лише для контейнерів із необхідними правами.

Існують інструменти для постійного моніторингу контейнерів, які аналізують активність у реальному часі та можуть виявляти аномальні дії, характерні для можливих атак. Завдяки впровадженню таких механізмів контейнеризація стає не лише інструментом для спрощення розгортання, а й засобом забезпечення комплексної безпеки системи.

Завдяки використанню Docker, OCI та інших технологій контейнеризації, організації отримують можливість ефективно адаптуватися до викликів сучасного бізнесу. Контейнеризація забезпечує стабільність за рахунок ізоляції кожного сервісу, що мінімізує вплив помилок одного мікросервісу на інші. Продуктивність систем досягається через швидке розгортання та автоматизацію процесів за допомогою таких інструментів, як Kubernetes, які оптимізують розподіл ресурсів між контейнерами. Завдяки OCI забезпечується універсальність і сумісність різних платформ, що дозволяє інтегрувати нові технології без значних витрат на адаптацію. Контейнеризація також надає потужні засоби безпеки, включаючи контроль доступу, шифрування даних і моніторинг активності. Це робить контейнеризацію незамінною для побудови масштабованих, надійних і безпечних систем у межах мікросервісної архітектури.

2.2 Оркестрація мікросервісів за допомогою Kubernetes

Оркестрація мікросервісів (рис.2.3) є фундаментальним етапом у розгортанні мікросервісної архітектури, оскільки вона забезпечує автоматизацію управління контейнерами, балансування навантаження, моніторинг і високу доступність системи. Kubernetes, як одна з провідних платформ оркестрації, надає розробникам і адміністраторам широкий спектр інструментів, які дозволяють не лише ефективно керувати контейнеризованими додатками, але й забезпечувати стійкість до збоїв та динамічне масштабування системи. Платформа стала стандартом для управління

контейнерними середовищами завдяки її здатності інтегруватися з багатьма сучасними технологіями, такими як хмарні платформи, системи CI/CD і засоби моніторингу. Kubernetes не тільки забезпечує автоматизацію повторюваних завдань, але й дозволяє підвищити продуктивність команд розробників, зосереджуючи їх зусилля на створенні нових функцій, а не на вирішенні операційних питань. Kubernetes забезпечує автоматизоване розгортання, масштабування та відновлення роботи контейнерів, що є ключовою перевагою для забезпечення стійкості і безперервності роботи системи. Завдяки концепції "кластерів", платформа дозволяє ефективно розподіляти контейнери між вузлами, забезпечуючи оптимальне використання ресурсів серверів. Kubernetes постійно моніторить стан вузлів і контейнерів, визначаючи потенційні проблеми або збої у роботі компонентів. У разі виявлення таких збоїв, система автоматично перезапускає або переміщує контейнер на доступний вузол, мінімізуючи час простоїв і втрати даних.

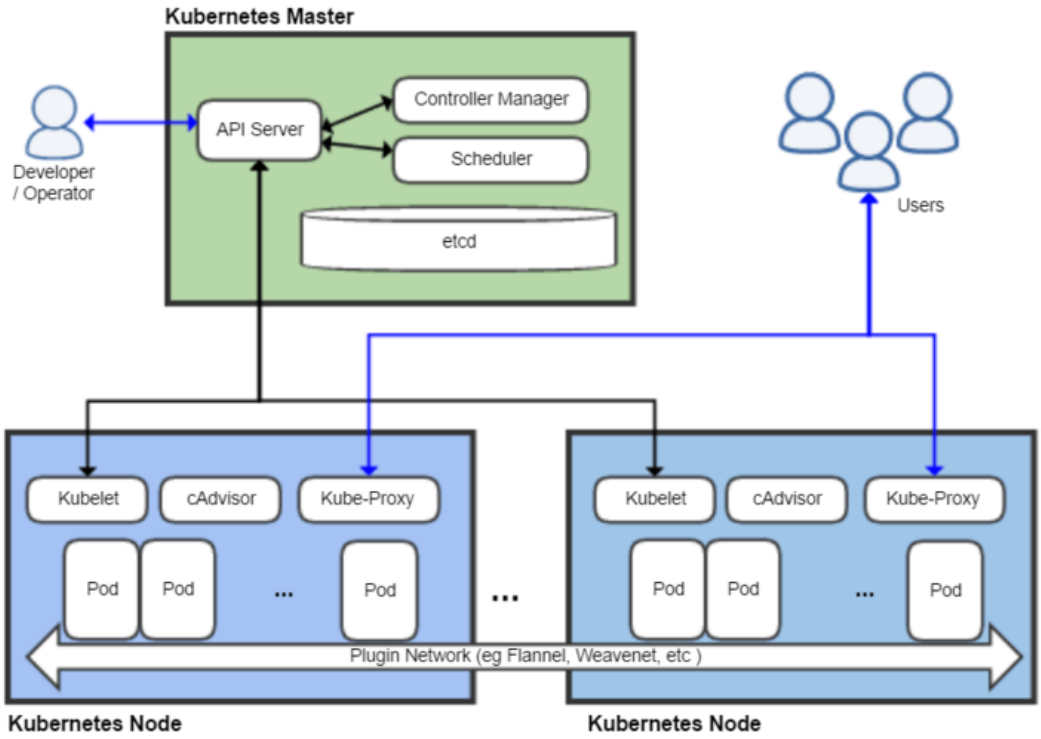

Рисунок 2.3 – Оркестрація мікросервісів Kubernetes

Платформа також підтримує динамічну реорганізацію ресурсів, дозволяючи автоматично адаптувати систему до змін у навантаженні. Наприклад, у період пікової активності Kubernetes може створювати додаткові екземпляри контейнерів для підтримки високого рівня продуктивності. Водночас, у періоди низької активності система здатна автоматично зменшувати кількість активних контейнерів, що знижує витрати на інфраструктуру. Ця здатність до гнучкої адаптації та автоматичного відновлення робить Kubernetes незамінним інструментом для забезпечення стабільності та продуктивності мікросервісних додатків у будь-якому середовищі. Використовуючи Kubernetes, компанії можуть зменшити складність управління інфраструктурою, гарантуючи стабільну роботу систем навіть у періоди високого навантаження. Гнучкість платформи дозволяє динамічно змінювати ресурси в реальному часі, забезпечуючи безперебійну роботу додатків. Автоматичне відновлення системи після збоїв, інтеграція з інструментами моніторингу та можливість використання хмарних платформ дозволяють значно оптимізувати операційні процеси та підвищити загальну продуктивність додатків. Важливою функцією Kubernetes є забезпечення високої доступності через використання Load Balancing. Це дозволяє рівномірно розподіляти трафік між сервісами, мінімізуючи ризики перевантаження окремих компонентів системи. Ця функція забезпечує плавний і безперебійний досвід для кінцевих користувачів навіть у разі значного збільшення кількості запитів. Наприклад, у випадку мікросервісів, які обслуговують критично важливі запити, таких як процесинг платежів або реєстрація користувачів, Load Balancing розподіляє навантаження між кількома екземплярами сервісу, забезпечуючи їх стабільну роботу.

Kubernetes підтримує горизонтальне масштабування, яке дозволяє додавати або видаляти екземпляри мікросервісів у залежності від поточного навантаження на систему. Це масштабування здійснюється автоматично за допомогою Horizontal Pod Autoscaler, що аналізує такі метрики, як використання процесора, пам'яті чи кількість активних запитів. Наприклад, якщо система фіксує зростання

навантаження на 80% протягом визначеного часу, Kubernetes автоматично створює додаткові екземпляри для підтримки продуктивності. Коли навантаження спадає, надлишкові ресурси вивільняються, що дозволяє знижувати витрати на інфраструктуру. Горизонтальне масштабування та Load Balancing у комплексі забезпечують надзвичайно важливий рівень стійкості та надійності мікросервісної архітектури. Горизонтальне масштабування дозволяє оперативно реагувати на зміну навантаження, додаючи або видаляючи екземпляри сервісів у відповідь на поточний обсяг запитів. Наприклад, під час пікових навантажень, таких як розпродажі чи великі маркетингові кампанії, автоматичне масштабування забезпечує збереження продуктивності, дозволяючи користувачам отримувати стабільний доступ до сервісів. Load Balancing, у свою чергу, розподіляє вхідний трафік між різними екземплярами сервісів, запобігаючи перевантаженню окремих компонентів. Це сприяє рівномірному використанню ресурсів і знижує ризик виникнення вузьких місць у системі. Крім того, інтеграція з механізмами автоматичного перенаправлення трафіку на здорові вузли підвищує загальну надійність системи, зменшуючи ризики простоїв і втрати даних. Разом ці механізми дозволяють мікросервісній архітектурі не лише адаптуватися до змін у навантаженні, але й забезпечувати високий рівень безперервності роботи навіть у найскладніших умовах.

Kubernetes також пропонує потужні засоби для управління конфігураціями і секретами, що є важливим аспектом у забезпеченні безпеки та ефективності роботи мікросервісних систем. Зберігання конфігурацій у вигляді ConfigMaps дозволяє централізовано керувати налаштуваннями додатків, розділяючи їх від контейнерних образів. Це спрощує процес оновлення параметрів, оскільки зміни в ConfigMaps автоматично застосовуються до відповідних сервісів без необхідності перевипуску контейнерів. Секрети, які зберігаються у Secret, забезпечують захищений спосіб управління конфіденційною інформацією, такою як ключі доступу, токени аутентифікації та сертифікати. Завдяки використанню механізмів шифрування та обмеження доступу лише для авторизованих контейнерів,

Kubernetes мінімізує ризики витоку критичних даних. Наприклад, секректи можуть використовуватися для передачі сертифікатів TLS до мікросервісів, що гарантує безпечну комунікацію між компонентами системи. Платформа підтримує динамічне оновлення конфігурацій і секретів, що дозволяє вносити зміни до системи без її перезавантаження. Це особливо важливо у високонавантажених додатках, де простої можуть призводити до значних фінансових втрат. Використання ConfigMaps і Secret у Kubernetes не тільки спрощує управління, але й підвищує рівень безпеки, адаптивності та гнучкості мікросервісної архітектури. Моніторинг і логування в Kubernetes реалізуються через інтеграцію з такими інструментами, як Prometheus, Grafana і Fluentd, що забезпечує глибоке розуміння роботи системи та її компонентів. Prometheus відповідає за збір і обробку метрик, надаючи розробникам детальну інформацію про використання ресурсів, продуктивність контейнерів і стан кластерів. Grafana використовується для візуалізації зібраних даних, створення інтерактивних панелей і звітів, що спрощує аналіз стану системи в реальному часі. Fluentd, у свою чергу, забезпечує централізоване логування, що дозволяє акумулювати логи з різних компонентів у єдиному сховищі, спрощуючи їх аналіз і діагностику проблем.

Завдяки інтеграції цих інструментів, адміністратори мають можливість автоматично виявляти відхилення у роботі сервісів, наприклад, надмірне використання пам’яті чи процесора, а також відстежувати помилки у роботі додатків. Це дозволяє не лише швидко реагувати на потенційні проблеми, але й проактивно запобігати можливим збоям через впровадження алертів і автоматичних дій на основі визначених правил. Такий підхід суттєво підвищує стійкість системи та зменшує час простоїв навіть у найскладніших робочих середовищах.

Kubernetes виступає як універсальне рішення для оркестрації мікросервісів, що забезпечує комплексну підтримку життєвого циклу контейнеризованих додатків. Платформа забезпечує гнучкість через можливість динамічного масштабування, адаптацію до різноманітних навантажень та інтеграцію з широким

спектром технологій. Її надійність ґрунтується на автоматичному відновленні роботи, балансуванні навантаження та багаторівневому моніторингу стану системи. Масштабованість Kubernetes дозволяє ефективно управляти системами, що включають тисячі контейнерів, забезпечуючи стабільну роботу навіть у періоди пікового навантаження. Це робить Kubernetes незамінним інструментом для реалізації сучасних високонавантажених додатків, які потребують високої продуктивності та стійкості до збоїв..

2.3 CI/CD процеси у мікросервісній архітектурі

Процеси CI/CD (Continuous Integration/Continuous Deployment) (рис.2.4) є невід'ємною частиною мікросервісної архітектури, оскільки вони забезпечують повну автоматизацію розробки, тестування, розгортання та моніторингу програмного забезпечення, значно спрощуючи управління складними розподіленими системами. Впровадження цих процесів дозволяє компаніям скоротити час між розробкою нових функцій і їхнім запуском у продуктивному середовищі, забезпечуючи високу якість, стабільність і надійність програмних рішень. Однією з основних переваг CI/CD є можливість інтеграції змін у коді в реальному часі з автоматичним тестуванням на кожному етапі життєвого циклу. Це дозволяє розробникам оперативно виявляти помилки, уникати конфліктів між командами, які працюють над різними мікросервісами, та підтримувати узгодженість роботи всіх компонентів системи. Інструменти автоматизації, такі як Jenkins, GitLab CI/CD, Travis CI та CircleCI, дозволяють налаштовувати складні пайплайни, які включають збірку, тестування та розгортання, що забезпечує передбачуваність результатів і стабільність функціонування. У мікросервісній архітектурі CI/CD також сприяє ефективній співпраці між різними командами, дозволяючи їм синхронізувати свої зусилля через прозору інтеграцію змін. Завдяки цьому компанії можуть підтримувати високу швидкість розробки, швидко адаптуватися до змін у вимогах ринку та впроваджувати інновації, які відповідають потребам користувачів. Крім того, автоматизовані процеси CI/CD знижують вплив

людського фактора, що мінімізує ризики помилок і забезпечує постійне вдосконалення програмного забезпечення.

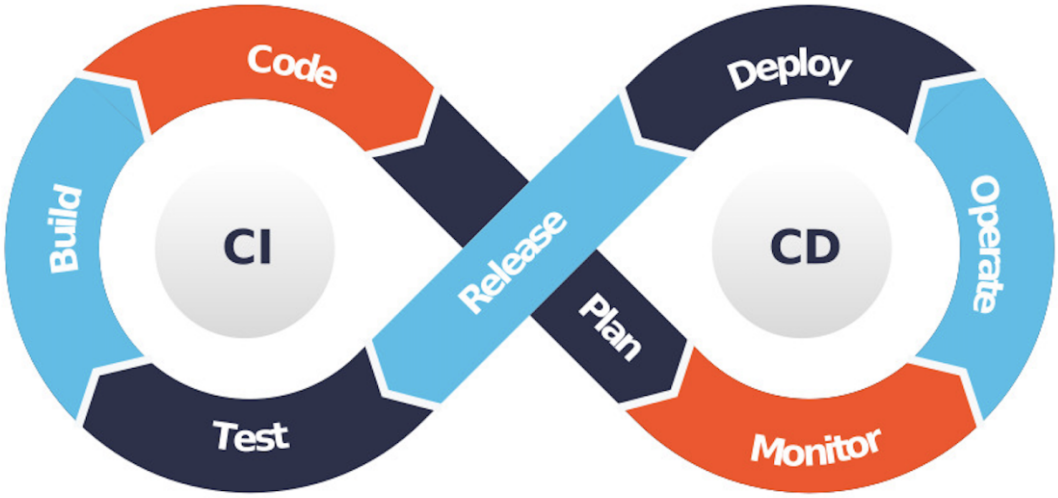

Рисунок 2.4 – CI/CD (Continuous Integration/Continuous Deployment)

CI (Continuous Integration) (рис.2.5) забезпечує безперервну інтеграцію змін, які вносяться до репозиторію, автоматично активуючи процеси збірки та тестування коду. Це дозволяє розробникам одразу побачити, як їхні зміни впливають на загальну стабільність системи. Кожна зміна автоматично перевіряється через набір тестів, включаючи юніт-тести, інтеграційні тести та статичний аналіз коду, що допомагає швидко виявляти та виправляти помилки ще до їхнього впровадження у продуктивне середовище. Для мікросервісної архітектури це особливо важливо, оскільки різні мікросервіси можуть розроблятися незалежними командами, які працюють паралельно. Регулярна інтеграція змін забезпечує синхронізацію між цими командами, що мінімізує ризики конфліктів у коді. Використання інструментів, таких як Jenkins, GitLab CI/CD, Travis CI або CircleCI, додає можливості автоматизації процесів, зменшуючи ручну працю та пришвидшуючи розгортання нових версій. Крім того, ці інструменти дозволяють налаштовувати складні пайплайни, які включають декілька етапів перевірки, гарантуючи якість кожного нового внесення в систему.

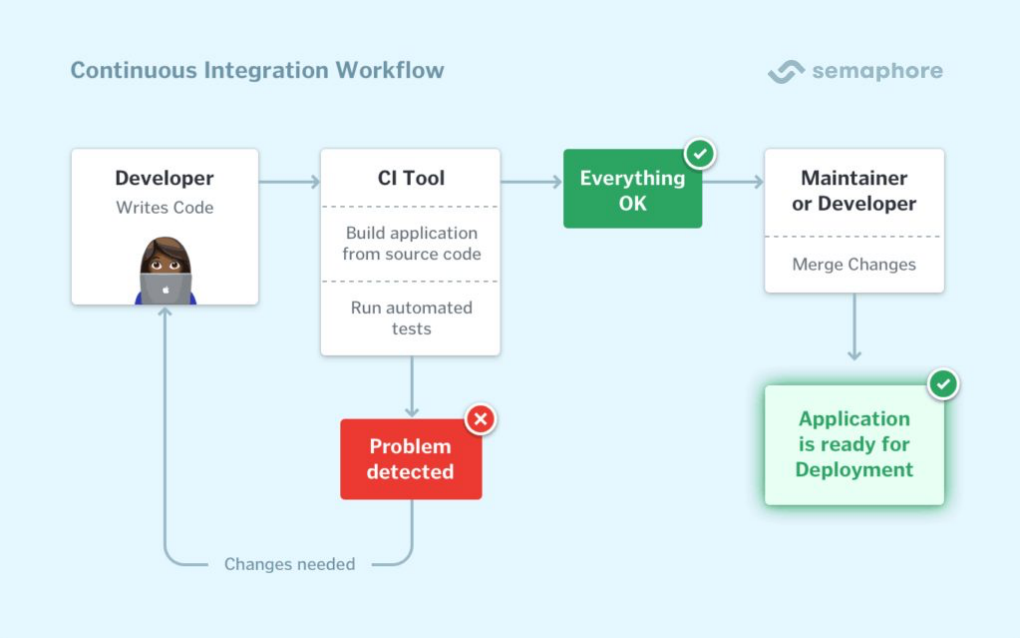

Рисунок 2.5 – CI (Continuous Integration)

CD (Continuous Deployment) дозволяє автоматично розгортати перевірений код у середовищах тестування, staging або production, що значно скорочує час між розробкою нової функції та її впровадженням у продуктивне середовище. Це забезпечує швидке впровадження оновлень і змін без необхідності втручання людини, тим самим знижуючи ймовірність людської помилки. У контексті мікросервісної архітектури CD спрощує процес оновлення окремих сервісів завдяки їхній автономності та ізоляції. Кожен мікросервіс може бути оновлений незалежно, без впливу на інші компоненти системи, що забезпечує безперервну роботу всієї архітектури. Це особливо важливо у випадках, коли деякі сервіси потребують частих оновлень через змінні бізнес-вимоги або необхідність швидкої реакції на виявлені помилки. Одним із ключових факторів, які роблять CD ефективним у мікросервісній архітектурі, є використання контейнеризації.

Контейнеризація дозволяє створювати ізольовані середовища для кожного сервісу, які включають усі необхідні залежності та конфігурації. Це забезпечує узгодженість і стабільність роботи сервісів у різних середовищах, від локального розробника до продуктивного серверу. Завдяки цьому процес розгортання стає

передбачуваним і легко контрольованим. Однією з ключових переваг CI/CD у мікросервісній архітектурі є автоматизоване тестування, яке забезпечує повний цикл перевірки на кожному етапі розробки та розгортання програмного забезпечення. Юніт-тести перевіряють окремі компоненти мікросервісів, гарантуючи їхню коректну роботу. Інтеграційні тести забезпечують синхронізацію та правильну взаємодію між мікросервісами, що є критично важливим у розподілених системах. Тестування взаємодії між сервісами дозволяє виявляти потенційні конфлікти в логіці роботи, особливо у випадках, коли сервіси залежать від спільних API чи обмінюються даними. Перевірка навантаження дозволяє оцінити, як система поводиться під час пікових запитів, визначити її обмеження та оптимізувати продуктивність.

Автоматизація цих етапів за допомогою таких інструментів, як Selenium, Postman чи Apache JMeter, значно скорочує час, необхідний для тестування, підвищує якість коду та зменшує ризики помилок у продуктивному середовищі. Крім того, використання паралельного тестування дозволяє одночасно перевіряти кілька мікросервісів, що забезпечує ефективність і точність процесу. Завдяки такому підходу CI/CD стає основою для створення надійних, масштабованих і високопродуктивних мікросервісних систем. Інструменти оркестрації, такі як Kubernetes, забезпечують багаторівневу підтримку CI/CD процесів, включаючи автоматичне масштабування ресурсів, управління оновленнями сервісів і забезпечення їхньої доступності навіть під час масштабних змін у системі. Kubernetes дозволяє динамічно виділяти додаткові ресурси для мікросервісів, які зазнають пікового навантаження, забезпечуючи стабільну роботу всієї архітектури. Крім того, автоматизовані механізми оновлення, такі як Rolling Updates і Canary Deployments, мінімізують ризики виникнення збоїв під час впровадження нових версій сервісів, забезпечуючи безперервність роботи. Інтеграція Kubernetes із системами моніторингу, такими як Prometheus і Grafana, створює потужну екосистему для збору, аналізу та візуалізації даних у реальному часі. Prometheus відповідає за збір метрик продуктивності, таких як використання CPU, пам’яті чи

кількість запитів до сервісів, тоді як Grafana перетворює ці дані у зрозумілі графіки та панелі. Це дозволяє адміністраторам і командам DevOps оперативно відстежувати ефективність внесених змін, швидко ідентифікувати проблеми та здійснювати коригувальні дії без значних затримок. Завдяки такій інтеграції компанії можуть забезпечувати високу стабільність, продуктивність і надійність своїх мікросервісних рішень.

Завдяки впровадженню CI/CD процесів у мікросервісній архітектурі, компанії можуть не лише значно підвищити швидкість розробки та оновлення своїх програмних рішень, але й створити більш стійку до змін інфраструктуру. Автоматизація цих процесів сприяє зниженню впливу людського фактора, що мінімізує ймовірність помилок під час інтеграції чи розгортання. Постійне тестування на різних етапах життєвого циклу забезпечує високу якість коду, що критично важливо у розподілених системах. Крім того, CI/CD дозволяє розробникам оперативно адаптуватися до нових вимог ринку, швидко впроваджувати інновації та зберігати конкурентоспроможність. У контексті масштабних мікросервісних архітектур це стає основою для ефективного управління складними програмними екосистемами..

2.4 Інструменти автоматизації розгортання мікросервісів

Автоматизація розгортання мікросервісів є одним із ключових аспектів успішного впровадження мікросервісної архітектури. Інструменти автоматизації дозволяють значно скоротити час розгортання, підвищити стабільність та передбачуваність процесів, а також забезпечити узгодженість роботи різних компонентів системи. Ці інструменти дозволяють уникнути ручних помилок під час конфігурації та оновлення, забезпечують повторюваність процесів і сприяють зниженню витрат на технічне обслуговування. Використання автоматизації також дозволяє компаніям швидко реагувати на зміни ринкових вимог, забезпечуючи високу гнучкість і адаптивність у розвитку мікросервісних систем. Одним із найбільш популярних інструментів автоматизації є Ansible. Цей інструмент надає

можливість створювати автоматизовані сценарії для розгортання мікросервісів, використовуючи просту та зрозумілу синтаксичну структуру YAML. Завдяки цьому Ansible легко адаптується до потреб різних середовищ, забезпечуючи швидкість і ефективність налаштувань. Він широко використовується для конфігурації серверів, управління контейнерами та оркестрації мікросервісів у масштабних системах. Інтеграція з Docker і Kubernetes розширює функціональність Ansible, дозволяючи створювати сценарії, які автоматично запускають контейнери, управляють їхніми мережевими налаштуваннями та забезпечують синхронізацію між сервісами. Наприклад, Ansible може автоматично оновлювати образи контейнерів у Docker Registry, перезапускати сервіси або налаштовувати їхню взаємодію у Kubernetes-кластерах. Ansible підтримує модульну структуру, що дає змогу легко розширювати його можливості за допомогою плагінів та кастомних модулів. Це робить інструмент надзвичайно гнучким і потужним у контексті автоматизації, дозволяючи забезпечити стабільну роботу навіть у найскладніших інфраструктурних сценаріях.

Terraform (рис.2.6) є ще одним потужним інструментом автоматизації, який дозволяє визначати інфраструктуру як код (Infrastructure as Code, IaC), що є ключовим принципом сучасного управління інфраструктурою. За допомогою Terraform можна створювати, модифікувати та управляти ресурсами в різних хмарних середовищах, таких як AWS, Azure чи Google Cloud Platform, а також у гібридних і локальних середовищах. Однією з головних переваг Terraform є його здатність забезпечувати повторюваність і передбачуваність інфраструктурних змін. За допомогою декларативного підходу розробники та адміністратори можуть описувати бажаний стан інфраструктури, після чого Terraform автоматично реалізує ці зміни. Це значно спрощує управління складними системами та дозволяє уникнути помилок, пов’язаних із ручними налаштуваннями. Terraform підтримує управління не лише обчислювальними ресурсами, такими як віртуальні машини або контейнери, але й мережевими налаштуваннями, системами зберігання даних, базами даних та іншими ключовими компонентами інфраструктури. Наприклад,

розробник може за кілька хвилин налаштувати цілісну інфраструктуру для мікросервісів, включаючи балансувальники навантаження, автошкалування та підключення до бази даних, використовуючи лише кілька рядків коду. Інструмент також пропонує функціонал управління версіями, що дозволяє відстежувати зміни в інфраструктурі та легко повертатися до попередніх конфігурацій у разі необхідності. Terraform активно інтегрується з іншими інструментами автоматизації, такими як Jenkins або Ansible, що робить його надзвичайно потужним у контексті комплексних сценаріїв розгортання. Це рішення підвищує ефективність роботи команд DevOps і забезпечує стабільну та масштабовану підтримку мікросервісної архітектури.

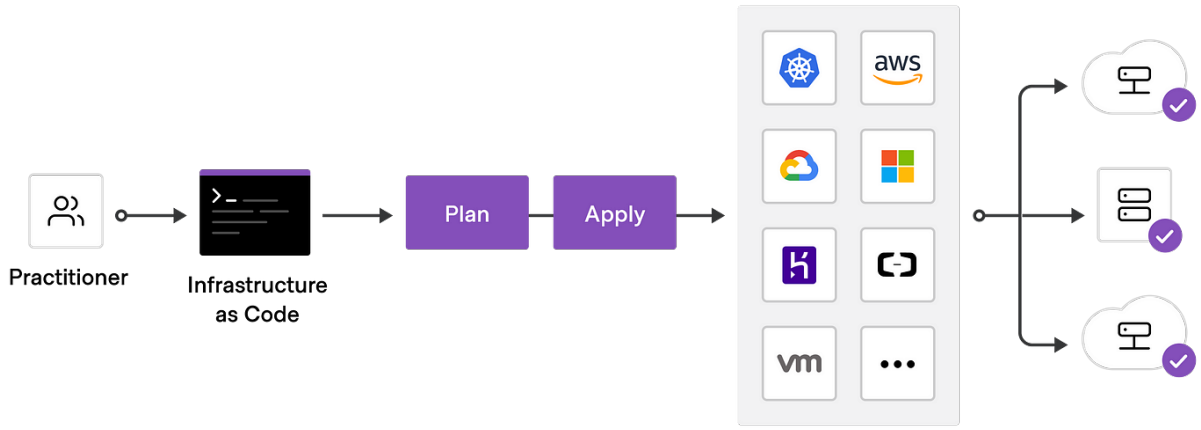

Рисунок 2.6 – Terraform

Інструмент Helm спеціалізується на спрощенні розгортання мікросервісів у середовищі Kubernetes, надаючи ефективний спосіб управління складними кластерами. Використовуючи Helm Charts, розробники можуть створювати повторювані та стандартизовані шаблони для розгортання додатків, що значно знижує складність ручного управління. Helm дозволяє легко управляти як окремими мікросервісами, так і повними додатками, забезпечуючи їхню інтеграцію з іншими компонентами системи. Helm також підтримує динамічне управління конфігураціями, дозволяючи вносити зміни до налаштувань додатків без необхідності перезапуску сервісів. Наприклад, зміни в налаштуваннях бази даних

або мережевих підключень можна швидко застосувати через оновлення Helm Charts, що скорочує час на адаптацію до нових умов. Ще однією важливою перевагою Helm є його здатність автоматизувати розгортання складних систем із взаємозалежними мікросервісами. Завдяки цьому інструменту можна швидко впроваджувати зміни, перевіряти їх у staging-середовищах і плавно переносити оновлення в production, мінімізуючи ризики збоїв. Helm Charts також підтримують версіонування, що дозволяє відстежувати зміни в конфігураціях і, за потреби, повертатися до попередніх версій, забезпечуючи надійність і передбачуваність процесів.

Інструмент Jenkins широко використовується для автоматизації CI/CD процесів і є особливо корисним для інтеграції з мікросервісними додатками завдяки своїй гнучкості та підтримці масштабованих сценаріїв. Використовуючи Jenkins Pipelines, розробники можуть налаштовувати багатоступеневі сценарії, які охоплюють повний цикл розробки — від автоматичного збирання коду до його тестування і розгортання в середовищах тестування, staging або production. Jenkins забезпечує інтеграцію з такими інструментами, як Docker, Kubernetes, Ansible та Terraform, що дозволяє легко керувати контейнеризованими мікросервісами та оркеструвати складні розгортання. Наприклад, Jenkins може автоматично створювати образи контейнерів, оновлювати їх у Docker Registry та розгортати у Kubernetes-кластерах, забезпечуючи узгодженість усіх компонентів системи. Ще однією важливою перевагою Jenkins є його здатність підтримувати динамічне масштабування ресурсів під час виконання пайплайнів, що дозволяє ефективно розподіляти навантаження між вузлами. Крім того, Jenkins активно використовує плагіни, які розширюють його можливості, додаючи підтримку моніторингу, аналізу безпеки коду, управління версіями та багато іншого. Завдяки цьому Jenkins є універсальним інструментом для автоматизації, який підходить як для невеликих команд, так і для масштабних корпоративних проектів.

Автоматизація розгортання мікросервісів за допомогою цих інструментів дозволяє компаніям значно оптимізувати свої процеси, зменшити витрати часу на

розгортання, мінімізувати ризики, пов’язані з ручними помилками, та досягти високої продуктивності навіть у масштабних системах. Використання таких рішень забезпечує стабільність роботи, полегшує інтеграцію нових компонентів та сприяє адаптивності інфраструктури до змін. Завдяки автоматизації компанії можуть зосередитися на стратегічних цілях і прискорювати інноваційні процеси, одночасно підтримуючи високу якість програмних продуктів..

2.5 Висновки по розділу

Методи розгортання мікросервісної архітектури відіграють ключову роль у забезпеченні ефективності, стабільності та масштабованості сучасних програмних систем. Розглянуті технології та інструменти автоматизації, такі як Docker, Kubernetes, Ansible, Terraform, Helm та Jenkins, забезпечують не лише зручність розгортання, але й мінімізують людські помилки, підвищують безпеку та забезпечують передбачуваність процесів. Контейнеризація, яка лежить в основі мікросервісної архітектури, забезпечує ізоляцію кожного сервісу, що значно підвищує надійність роботи системи. Оркестрація за допомогою Kubernetes дає змогу автоматизувати управління контейнерами, ефективно розподіляти ресурси, забезпечувати високу доступність та безперервну роботу системи. Інструменти автоматизації, такі як Ansible і Terraform, полегшують управління інфраструктурою та забезпечують повторюваність процесів розгортання.

CI/CD процеси, інтегровані з використанням Jenkins, сприяють швидкому та надійному впровадженню змін, що є критично важливим для динамічних середовищ. Інструмент Helm дозволяє стандартизувати і спростити розгортання у великих кластерах, забезпечуючи гнучке управління конфігураціями. Усі ці технології працюють у синергії, створюючи потужну екосистему для розробки, тестування та підтримки мікросервісів.

Отже, автоматизація розгортання мікросервісів дозволяє організаціям скоротити витрати, знизити ризики, прискорити впровадження нових функцій і забезпечити стабільну роботу навіть у найскладніших інфраструктурних сценаріях.

Це робить впровадження мікросервісної архітектури не лише технічно доцільним, але й стратегічно вигідним рішенням.

3 МЕТОДИ БЕЗПЕКИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

3.1 Аутентифікація та авторизація у мікросервісах

Аутентифікація та авторизація є критично важливими аспектами забезпечення безпеки у мікросервісній архітектурі, оскільки ці механізми запобігають несанкціонованому доступу до системи та забезпечують контроль над правами доступу. У мікросервісній архітектурі кожен компонент працює автономно, що ускладнює централізоване управління безпекою. Тому перевірка автентичності запитів і визначення прав доступу мають відбуватися на рівні кожного мікросервісу, що вимагає застосування розподілених і сучасних підходів. Для досягнення цієї мети широко використовуються протоколи OAuth 2.0 та OpenID Connect, які забезпечують стандартизовану взаємодію між клієнтами та серверами. OAuth 2.0 надає механізм делегованої авторизації, що дозволяє користувачам надавати стороннім додаткам доступ до своїх ресурсів без передачі паролів. OpenID Connect додає рівень аутентифікації до OAuth 2.0, дозволяючи перевіряти особу користувача за допомогою стандартизованих методів. Завдяки цим протоколам мікросервісна архітектура стає не лише більш захищеною, але й адаптивною до сучасних вимог.

Основні підходи до аутентифікації у мікросервісах

JSON Web Tokens (JWT) — це стандарт для безпечної передачі інформації між клієнтами та серверами у вигляді компактного та самодостатнього токена, який може бути використаний для аутентифікації та авторизації. JWT складається з трьох частин:

- Header (Заголовок): Визначає тип токена (зазвичай "JWT") та алгоритм підпису (наприклад, HMAC SHA256 або RSA).
- Payload (Тіло): Містить інформацію про користувача, яку можна декодувати, але не захищена від перегляду, якщо токен не зашифрований. У ньому зберігаються різні дані, такі як ідентифікатор користувача, його ролі або інші атрибути, що необхідні для доступу до ресурсів.

- Signature (Підпис): Використовується для перевірки того, що токен не було змінено. Підпис створюється шляхом застосування алгоритму до закодованого заголовка та тіла токена, за допомогою секретного ключа або публічного/приватного ключа, якщо застосовується асиметричне шифрування.

JWT зазвичай використовуються для:

Аутентифікації: Користувач отримує токен після успішної автентифікації і використовує його для доступу до захищених ресурсів. Токен зберігається на стороні клієнта (зазвичай в localStorage або cookies).

Авторизації: Токен перевіряється при кожному запиті до сервера, щоб визначити, чи має користувач доступ до конкретних ресурсів або мікросервісів. Завдяки своєму формату, JWT дозволяють серверу не зберігати стан про сесії користувачів, що робить їх ідеальними для масштабованих, розподілених систем.

Мікросервісних архітектур: Взаємодія між мікросервісами часто вимагає обміну аутентифікаційними даними, і JWT дозволяє здійснювати цю передачу без необхідності повторної автентифікації кожного разу. Токен може бути перевірений кожним мікросервісом, що забезпечує масштабованість та безпеку.

Переваги JWT:

Безшовний досвід користувача: Оскільки JWT є самодостатнім і містить всю необхідну інформацію для перевірки доступу, кожен запит, що містить токен, не потребує додаткових запитів до бази даних для перевірки користувача.

Масштабованість: Використання JWT дозволяє реалізувати безсерверні стратегії автентифікації, що знижує навантаження на сервери та бази даних.

Безпека: Оскільки підпис токена гарантує, що дані не були змінені, він є надійним інструментом для забезпечення автентичності.

Завдяки своїм характеристикам, JWT є широко використовуваним стандартом для автентифікації в сучасних веб-застосунках, особливо в умовах мікросервісних архітектур та розподілених систем.

Приклад структури JWT:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022,
  "roles": ["admin", "user"]
}
```

Сервіс аутентифікації: Для централізації перевірки облікових даних створюється окремий сервіс, відповідальний за аутентифікацію. Він видає токени доступу, які клієнти використовують для доступу до інших сервісів.

Код прикладу аутентифікації:

```
from flask import Flask, request, jsonify  # Імпортуємо необхідні бібліотеки для створення Flask додатку, обробки запитів та відповіді у форматі JSON
import jwt  # Бібліотека для роботи з JSON Web Tokens
import datetime  # Для роботи з датами та часом
app = Flask(__name__)  # Створюємо екземпляр Flask додатку
app.config['SECRET_KEY'] = 'supersecretkey'  # Налаштовуємо секретний ключ для підпису JWT токенів. Важливо, щоб цей ключ був конфіденційним.
# Маршрут для логіну, на якому користувачі можуть отримати токен після правильної автентифікації
@app.route('/login', methods=['POST'])
def login():
    # Отримуємо дані з запиту у форматі JSON
    auth = request.json
```

```
        # Перевіряємо, чи є дані та чи співпадають логін та пароль
        if auth and auth['username'] == 'admin' and auth['password']
== 'password':
            # Якщо дані правильні, створюємо JWT токен
            token = jwt.encode({
                'user':  auth['username'],     #  Зберігаємо  ім'я
користувача в токені
                'exp':          datetime.datetime.utcnow()          +
datetime.timedelta(hours=1)  # Термін дії токену - 1 година
            },  app.config['SECRET_KEY'],  algorithm="HS256")     #
Підписуємо токен за допомогою секретного ключа та алгоритму HS256
            # Повертаємо токен у відповіді
            return jsonify({'token': token})
        # Якщо логін або пароль неправильні, повертаємо повідомлення
про помилку
        return jsonify({'message': 'Invalid credentials'}), 401
    if __name__ == '__main__':
        # Запускаємо Flask сервер
        app.run(debug=True)
```

Імпорти:

- Flask використовується для створення веб-сервера.
- request дозволяє отримувати дані з HTTP запиту.
- jsonify дозволяє формувати відповіді у форматі JSON.
- jwt — бібліотека для роботи з JSON Web Tokens.
- datetime — для роботи з датами та часом, зокрема для визначення терміну дії токену.

Маршрут /login:

1. Запит до цього маршруту приймає тільки POST метод, тобто дані про логін та пароль передаються у тілі запиту.

2. Після отримання запиту перевіряється, чи правильно введено ім'я користувача та пароль.
3. Якщо дані правильні, створюється JWT токен, який включає:
4. user: ім'я користувача.
5. exp: час закінчення терміну дії токену (1 година з моменту генерації).
6. Токен підписується за допомогою секретного ключа SECRET_KEY та алгоритму HS256.
7. Якщо дані неправильні, повертається відповідь з кодом 401 (Unauthorized).

Запуск сервера:

Викликається метод run() для запуску сервера на локальному хості.

Цей код дозволяє користувачам отримати JWT токен після автентифікації, який можна використовувати для доступу до захищених маршрутів у додатку.

Основні підходи до авторизації у мікросервісах:

Рольова модель доступу (RBAC)

Рольова модель доступу (Role-Based Access Control, RBAC) є одним з найпоширеніших підходів до авторизації в мікросервісах. Згідно з цією моделлю, доступ користувача до певних ресурсів або операцій визначається на основі ролей, які йому призначені.

Ключові особливості RBAC:

Ролі користувача: Роль визначає набір дозволених або заборонених дій. Наприклад, роль "Адміністратор" може мати доступ до всіх ресурсів, тоді як роль "Користувач" може мати обмежений доступ.

Призначення ролей: Кожен користувач може бути пов'язаний з однією або кількома ролями, а ці ролі визначають рівень доступу до ресурсів.

Політики доступу: Ролі можуть бути налаштовані таким чином, щоб дозволяти або забороняти доступ до певних дій чи ресурсів у системі.

Переваги RBAC:

Простота в управлінні: Всі користувачі з однією роллю мають однаковий рівень доступу.

Масштабованість: Легко масштабувати та додавати нові ролі без необхідності переналаштовувати доступ кожному окремому користувачу.

Безпека: Мінімізує ризик помилок, оскільки доступ контролюється через ролі, а не для кожного окремого користувача.

Недоліки RBAC:

Обмеження гнучкості: У випадку, якщо потрібно встановити доступ на основі дуже конкретних умов (наприклад, час доступу або місцезнаходження), RBAC може бути недостатнім.

Політики доступу (ABAC)

Політики доступу, або атрибутивний контроль доступу (Attribute-Based Access Control, ABAC), визначають доступ до ресурсів не лише на основі ролей, а й за допомогою атрибутів користувача, ресурсу або середовища.

Ключові особливості ABAC:

Атрибути користувача: Включають такі характеристики, як роль, місцезнаходження, час доби, тип пристрою або навіть статус роботи.

Політики доступу: Політики описують правила, які використовують ці атрибути для прийняття рішення щодо дозволу чи заборони доступу.

Механізми перевірки: Кожен ресурс може мати свою політику доступу, і при кожному запиті оцінюється, чи відповідає запитуваний доступ наявним атрибутам користувача та іншими умовами.

Переваги ABAC:

Висока гнучкість: ABAC дозволяє реалізувати дуже детальне управління доступом, враховуючи не тільки роль користувача, але й інші динамічні фактори.

Контекстуальний доступ: Врахування таких факторів, як місцезнаходження, час доступу або рівень терміновості запиту, дозволяє більш точно визначати, хто має доступ до конкретних ресурсів у конкретних умовах.

Недоліки ABAC:

Складність: Моделі ABAC потребують більш складної реалізації політик, оскільки кожен доступ залежить від різних атрибутів, що може ускладнити адміністрування.

Перевантаження: За великої кількості атрибутів і політик може зростати складність і навантаження на систему під час прийняття рішень про доступ.

Гібридний підхід: Комбінація RBAC та ABAC

У реальних системах часто застосовується гібридний підхід, де використовується комбінація ролей і атрибутів для визначення доступу. Це дозволяє поєднати простоту ролей з гнучкістю атрибутивного контролю доступу, що забезпечує більшу масштабованість та точність.

Приклад гібридного підходу:

Користувач може бути частиною певної ролі, але додатково можуть бути враховані його атрибути, наприклад, його час доступу або місцезнаходження, щоб надати або обмежити доступ до конкретних ресурсів.

Наприклад, роль "Менеджер" може мати доступ до певних ресурсів, але тільки якщо він знаходиться в офісі (атрибут місцезнаходження) або тільки в робочі години (атрибут часу).

Централізована автентифікація та розподілена авторизація

У мікросервісах часто застосовуються централізовані методи автентифікації з розподіленими механізмами авторизації. У цьому випадку автентифікація виконується через окремий сервіс (наприклад, через OAuth або OpenID Connect), який видає токени (JWT). Ці токени потім використовуються для доступу до мікросервісів.

Централізоване управління автентифікацією: Користувачі проходять автентифікацію в одному центральному сервісі.

Розподілені механізми авторизації: Кожен мікросервіс може здійснювати авторизацію за допомогою JWT токенів, не потребуючи додаткових запитів до центрального сервісу. Це дозволяє забезпечити масштабованість і зменшити

навантаження на центральний сервіс, одночасно забезпечуючи високий рівень безпеки завдяки розподіленим перевіркам доступу.

У мікросервісних архітектурах важливо обрати підхід до авторизації, який забезпечить як безпеку, так і зручність масштабування. Використання ролей і атрибутів дозволяє реалізувати гнучкі і ефективні стратегії управління доступом, а також дає змогу швидко адаптувати систему до змінюваних умов..

3.2 Використання API Gateway для захисту даних

API Gateway — це шаблон проектування, який виступає як єдина точка входу для клієнтів до різноманітних мікросервісів у системі. Всі запити, що надходять до системи, проходять через API Gateway, який виконує роль "проксі" для перехоплення запитів і направлення їх до відповідних мікросервісів. Це дозволяє централізовано керувати усіма запитами, моніторингом і безпекою в межах мікросервісної архітектури. API Gateway не тільки зручний з точки зору управління запитами, але й має потужний потенціал для забезпечення безпеки системи. Нижче розглянемо основні переваги та функції API Gateway в контексті безпеки мікросервісної архітектури.

Ключові функції API Gateway для безпеки

Централізоване управління доступом: API Gateway дозволяє централізовано реалізувати механізми аутентифікації та авторизації для всіх мікросервісів. Це означає, що незалежно від того, скільки мікросервісів у вас є, всі вони можуть перевіряти доступ користувача або клієнта через єдину точку — API Gateway.

Аутентифікація: API Gateway може бути налаштований для перевірки JWT (JSON Web Tokens), OAuth2, API ключів або інших методів аутентифікації для кожного запиту.

Авторизація: На основі аутентифікаційних даних API Gateway може передавати запити лише тим користувачам або сервісам, які мають право доступу до певних ресурсів.

Масштабованість: API Gateway дозволяє масштабувати систему, додаючи нові мікросервіси без необхідності змінювати політики безпеки чи доступу. Оскільки весь трафік проходить через API Gateway, з його допомогою можна інтегрувати нові сервіси, не змінюючи структуру безпеки для існуючих мікросервісів.

Шифрування та захист даних: API Gateway може забезпечити шифрування даних за допомогою SSL/TLS на рівні комунікацій. Це означає, що дані, що передаються між клієнтом і мікросервісами, будуть захищені від перехоплення та атак типу «Man-in-the-Middle».

Обмеження частоти запитів (Rate Limiting): API Gateway дозволяє обмежувати кількість запитів до мікросервісів. Це важливо для запобігання атак типу Denial of Service (DoS) або Distributed Denial of Service (DDoS), які можуть знизити продуктивність системи або навіть призвести до її відмови. API Gateway може застосовувати ліміти на кількість запитів від одного клієнта за певний період часу (наприклад, 100 запитів на хвилину).

Логування та моніторинг: API Gateway забезпечує централізоване логування всіх запитів і відповідей. Це дозволяє отримати детальну інформацію про активність користувачів та мікросервісів і допомагає швидко виявити аномалії або потенційні загрози. Можна збирати дані про частоту запитів, типи помилок, час відповіді і багато іншого.

Обробка помилок та відповіді: API Gateway може централізовано обробляти помилки, що виникають у процесі взаємодії з мікросервісами. Це дозволяє уникнути розкриття чутливої інформації про внутрішню структуру мікросервісів, такої як стек помилок або технології, що використовуються.

Інтерфейс для мікросервісів: API Gateway може виконувати агрегацію відповідей від кількох мікросервісів в одну єдину відповідь для клієнта. Це значно спрощує інтерфейс для клієнтів і дозволяє зменшити кількість запитів, які вони повинні зробити, одночасно знижуючи навантаження на мікросервіси.

Розподілена перевірка доступу та аутентифікації: У випадку з мікросервісною архітектурою, де кожен сервіс може бути незалежним, використання API Gateway дозволяє виконувати перевірку аутентифікації один раз при вході до системи і застосовувати її до всіх мікросервісів. Це допомагає уникнути дублювання механізмів перевірки доступу в кожному окремому мікросервісі.

Архітектурні рішення для API Gateway

API Gateway може бути реалізований з використанням різних технологій. Ось кілька популярних варіантів:

Kong: Kong є одним з найбільш популярних рішень для API Gateway. Він надає потужні механізми для керування трафіком, аутентифікацією, моніторингом і обмеженням частоти запитів. Kong підтримує різні плагіни для безпеки, зокрема для JWT, OAuth2, базових аутентифікаційних методів, шифрування, моніторингу та інше.

Nginx: Nginx може бути використаний як API Gateway для обробки запитів, маршрутизації трафіку до мікросервісів і виконання додаткових функцій безпеки. Він є дуже швидким і ефективним для великих навантажень і має підтримку SSL, балансування навантаження та інші корисні функції.

Spring Cloud Gateway: Для мікросервісних архітектур, заснованих на Java, Spring Cloud Gateway є потужним рішенням для маршрутизації і забезпечення безпеки. Spring Cloud Gateway забезпечує інтеграцію з іншими продуктами Spring, такими як Spring Security для аутентифікації та авторизації.

AWS API Gateway: Якщо ваша система працює в хмарі Amazon Web Services (AWS), AWS API Gateway є відмінним вибором для керування запитами і забезпечення безпеки мікросервісів. Цей сервіс інтегрується з іншими інструментами AWS для аутентифікації (наприклад, через AWS Cognito), шифрування, моніторингу та обмеження частоти запитів.

Переваги використання API Gateway для безпеки мікросервісної архітектури:

Централізоване управління безпекою: API Gateway дозволяє централізувати управління політиками безпеки, такими як аутентифікація, авторизація, обмеження запитів, і застосовувати їх до всіх мікросервісів одночасно.

Знижує складність: API Gateway дозволяє уникнути дублювання логіки безпеки в кожному мікросервісі, що знижує загальну складність системи.

Масштабованість: З API Gateway можна додавати нові мікросервіси без зміни існуючих політик безпеки, що дозволяє легко масштабувати систему.

Поліпшений контроль доступу: API Gateway може забезпечити детальне налаштування доступу до мікросервісів на основі різних параметрів (наприклад, ролей користувачів або атрибутів запиту), що підвищує безпеку.

Універсальний моніторинг: Завдяки централізованому моніторингу API Gateway дозволяє швидко виявляти аномалії або підозрілу активність, що дозволяє знижувати ризик атак на систему.

Приклад реалізації API Gateway для безпеки

Можна реалізувати API Gateway за допомогою таких інструментів, як Kong, Nginx або Spring Cloud Gateway. Розглянемо приклад реалізації простого API Gateway на Flask з перевіркою JWT для забезпечення безпеки.

```
    from flask import Flask, request, jsonify  # Імпортуємо необхідні бібліотеки Flask для створення додатку, обробки запитів і відповіді у форматі JSON
    import jwt  # Бібліотека для роботи з JSON Web Tokens (JWT)
    import datetime  # Для роботи з датами та часом
    app = Flask(__name__)  # Створюємо екземпляр Flask додатку
    app.config['SECRET_KEY'] = 'supersecretkey'    # Налаштовуємо секретний ключ для підпису JWT токенів. Він має бути конфіденційним.
    # Декоратор для перевірки наявності і дійсності токену
    def token_required(f):
        @wraps(f)   # Забезпечує збереження метаданих оригінальної функції
        def decorated_function(*args, **kwargs):
```

```
            token = None  # Ініціалізуємо змінну для зберігання токену
            # Перевіряємо наявність токену в заголовку запиту
            if 'Authorization' in request.headers:
                token = request.headers['Authorization']
            # Якщо токен відсутній, повертаємо помилку 401
            if not token:
                return jsonify({'message': 'Token is missing!'}), 401
            try:
                # Декодуємо токен, використовуючи секретний ключ
                data = jwt.decode(token, app.config['SECRET_KEY'],
algorithms=["HS256"])
                current_user = data['user']    # Отримуємо ім'я
користувача з токену
            except jwt.ExpiredSignatureError:
                # Якщо токен прострочений
                return jsonify({'message': 'Token has expired!'}),
401
            except jwt.InvalidTokenError:
                # Якщо токен є некоректним
                return jsonify({'message': 'Token is invalid!'}), 401
            # Передаємо користувача в обробник маршруту
            return f(current_user, *args, **kwargs)
        return decorated_function
    # Захищений маршрут, доступ до якого є тільки для користувачів з
валідним токеном
    @app.route('/protected', methods=['GET'])
    @token_required  # Застосовуємо декоратор для перевірки токену
    def protected_route(current_user):
        # Якщо токен валідний, повертаємо відповідь з привітанням
        return jsonify({'message': f'Hello, {current_user}. This is
a protected route!'})
    # Маршрут для логіну та отримання JWT токену
    @app.route('/login', methods=['POST'])
    def login():
```

```
        # Отримуємо дані з тіла запиту в форматі JSON
        auth = request.json
        # Перевіряємо логін і пароль
        if auth and auth['username'] == 'admin' and auth['password']
== 'password':
            # Створюємо JWT токен, який містить ім'я користувача і
термін дії токену (1 година)
            token = jwt.encode({
                'user': auth['username'],
                'exp':          datetime.datetime.utcnow()          +
datetime.timedelta(hours=1)  # Термін дії токену - 1 година
            },  app.config['SECRET_KEY'],  algorithm="HS256")     #
Підписуємо токен секретним ключем
            return jsonify({'token': token})  # Повертаємо токен у
відповіді
        # Якщо дані неправильні, повертаємо помилку
        return jsonify({'message': 'Invalid credentials'}), 401
    if __name__ == '__main__':
        # Запускаємо Flask сервер
        app.run(debug=True)
```

Імпорти:

- Flask: Використовується для створення веб-додатку.
- request: Дозволяє отримувати запити та їхні дані.
- jsonify: Функція для формування відповідей у форматі JSON.
- jwt: Бібліотека для роботи з JSON Web Tokens (JWT), яка використовується для аутентифікації.
- datetime: Модуль для роботи з датами та часом, який використовується для визначення терміну дії токену.

Декоратор token_required:

- Це функція, яка перевіряє наявність і правильність токену в заголовках запиту.
- Якщо токен відсутній або неправильний, повертається помилка 401 (Unauthorized).
- Якщо токен правильний, передаємо дані користувача (ім'я) у функцію маршруту.

Маршрут /protected:

- Цей маршрут доступний лише користувачам з валідним токеном. Він використовує декоратор @token_required, який перевіряє правильність токену перед тим, як надати доступ.
- Після успішної перевірки токену повертається повідомлення з ім'ям користувача.

Маршрут /login:

- Цей маршрут дозволяє користувачеві отримати токен після введення правильного імені користувача та пароля.
- Токен містить ідентифікаційну інформацію про користувача та термін його дії.
- Якщо введені дані неправильні, повертається помилка 401.

Запуск серверу:

- Сервер запускається з використанням app.run(debug=True), що дозволяє відладку додатку під час розробки.

Додаткові можливості:

Обробка помилок: Додано обробку помилок для прострочених і некоректних токенів. У разі простроченого токену буде повернено повідомлення "Token has expired", а для некоректного токену — "Token is invalid".

Безпека: Секретний ключ SECRET_KEY необхідно тримати в безпеці, оскільки він використовується для підпису токенів. Рекомендується зберігати його в окремому конфігураційному файлі або використовувати змінні середовища.

Даний приклад демонструє основи використання JWT для аутентифікації в Flask додатку, де всі запити до захищених ресурсів перевіряються на наявність і дійсність токену.

API Gateway є потужним інструментом для забезпечення безпеки в мікросервісних архітектурах. Він дозволяє централізовано контролювати доступ до всіх мікросервісів, здійснювати шифрування даних, обмежувати частоту запитів і вести моніторинг активності, що значно покращує захист системи в цілому..

3.3 Механізми шифрування у передачі даних між сервісами

Механізми шифрування в передачі даних між сервісами є важливим аспектом безпеки в мікросервісних архітектурах, оскільки вони забезпечують захист конфіденційності та цілісності даних під час їх переміщення через мережу. В умовах сучасних мікросервісних систем, де сервіси можуть бути розміщені на різних фізичних або віртуальних машинах, а також мати окремі точки доступу через відкриті мережі, захист даних стає першочерговим завданням. Кожен з мікросервісів часто комунікує з іншими через API або міжсервісні виклики, що робить необхідним застосування ефективних методів шифрування для запобігання перехопленню або маніпуляції даними в процесі їх передачі. У сучасних мікросервісних архітектурах, де застосовуються множинні компоненти для забезпечення функціональності, дані можуть передаватися між сервісами по різних каналах — таких як HTTP, WebSockets, gRPC або інші протоколи. Кожен із цих каналів може бути уразливим до атак, таких як перехоплення або несанкціоноване модифікування даних, що передаються. Це особливо важливо, якщо мікросервіси працюють у різних фізичних або хмарних середовищах, де зв'язок між компонентами відбувається через публічні або напівпублічні мережі, що збільшує ймовірність несанкціонованого доступу. Зазвичай у мікросервісних системах використовуються API або міжсервісні виклики для того, щоб обмінюватися інформацією між компонентами. Оскільки мікросервіси часто складаються з безлічі взаємодіючих сервісів, без належного захисту дані можуть бути вразливими

до атак. Наприклад, у разі перехоплення пакету інформації під час передачі через незахищену мережу, зловмисники можуть отримати доступ до чутливої інформації або змінити її, що призведе до серйозних проблем безпеки.

Методи шифрування забезпечують захист на рівні передачі даних, дозволяючи зашифрувати інформацію таким чином, що вона стане недоступною або неможливою для зміни без ключа дешифрування. Це не тільки захищає від перехоплення інформації, але й гарантує, що дані не будуть змінені в процесі передавання.

Додаткові аспекти

Захист конфіденційності: Шифрування дозволяє уникнути витоку конфіденційної інформації. Це особливо важливо, якщо сервіси обробляють особисті дані користувачів або фінансову інформацію, де кожен витік може призвести до серйозних наслідків, як для користувачів, так і для організацій.

Шифрування в реальному часі: У випадку, коли дані обробляються в реальному часі, шифрування забезпечує те, що навіть при обміні даними між сервісами, вони будуть захищені від несанкціонованого доступу протягом усього процесу.

Інтеграція шифрування в CI/CD: Враховуючи, що в мікросервісних архітектурах розгортання нових версій сервісів є частим, інтеграція механізмів шифрування на етапі безперервної інтеграції та доставки (CI/CD) дозволяє автоматизувати процес захисту даних на всіх етапах — від тестування до виробничого середовища.

Тому застосування шифрування між сервісами не лише підвищує безпеку, але й дозволяє створювати більш стійкі та захищені інфраструктури, де передача чутливої інформації відбувається в безпечному середовищі, навіть коли дані передаються через відкриті або непередбачувані мережі.

Мікросервісні архітектури, як правило, складаються з великої кількості незалежних компонентів, що мають свої власні ресурси, бази даних і логіку обробки запитів. Ці сервіси часто взаємодіють між собою через відкриті канали

зв'язку, такі як HTTP або gRPC, що робить передачу чутливої інформації уразливою для атак, таких як перехоплення трафіку або його модифікація. Шифрування, у свою чергу, запобігає подібним загрозам, забезпечуючи необхідний рівень захисту даних.

Мікросервісна архітектура передбачає побудову додатків як сукупності малих, незалежних компонентів (мікросервісів), кожен з яких виконує певну функцію або серію функцій. Такі компоненти можуть бути розгорнуті на окремих серверах, віртуальних машинах або контейнерах, часто в різних мережах або хмарних середовищах. Це дає величезну гнучкість і масштабованість системи, оскільки кожен мікросервіс можна масштабувати або оновлювати незалежно від інших. Однак така архітектура також має суттєві вразливості. Мікросервіси зазвичай спілкуються між собою за допомогою REST API, gRPC, SOAP або інших протоколів, що дозволяє їм обмінюватися даними та викликати функції один одного. Проте ці канали зв'язку можуть бути відкритими, що робить передану інформацію вразливою до атак, таких як перехоплення трафіку або модифікація даних. Коли трафік проходить через незахищені канали, зловмисники можуть спробувати «прослухати» або навіть змінити дані, що передаються між сервісами. Вразливості, пов'язані з передачею даних, можуть стати серйозною загрозою для конфіденційності та цілісності інформації. Наприклад, якщо в одному з сервісів передається пароль користувача або фінансова інформація через незахищений канал, хакери можуть легко перехопити ці дані. Ще однією потенційною загрозою є атаки типу man-in-the-middle, коли зловмисник перехоплює та змінює дані, що передаються між двома сервісами.

Як шифрування вирішує ці проблеми

Шифрування даних між сервісами є потужним засобом захисту від перерахованих загроз. Оскільки при шифруванні дані перетворюються на недоступний для розуміння текст, зловмисники, які перехоплюють такий трафік, не зможуть прочитати або змінити дані без відповідного ключа дешифрування.

Таким чином, шифрування гарантує, що навіть якщо хтось зможе перехопити трафік, він не зможе отримати корисну інформацію або змінити її.

Шифрування передавання даних між сервісами забезпечує не лише конфіденційність, але й цілісність даних. Під час шифрування додається так звана контрольна сума (наприклад, HMAC — Hash-based Message Authentication Code), що дозволяє виявити навіть незначні зміни в даних під час їх передачі. Таким чином, шифрування в поєднанні з аутентифікацією повідомлень забезпечує комплексний захист від атак, орієнтованих на перехоплення або модифікацію інформації.

Додаткові уразливості та їх рішення:

Зловживання API: Якщо сервіс надає відкриті API без належного контролю доступу, зловмисники можуть скористатися ними для атаки на систему. Протокол шифрування (як SSL/TLS) у поєднанні з механізмами аутентифікації, такими як OAuth 2.0 або API-ключі, допомагає контролювати доступ до цих API та захищає сервіси від несанкціонованих запитів.

Використання спільного доступу до даних: У деяких випадках мікросервіси можуть мати спільний доступ до баз даних або ресурсів, що може створити небезпеку, якщо дані не будуть належним чином захищені. Для таких ситуацій рекомендується використовувати шифрування даних на рівні бази даних або застосовувати спеціалізовані інструменти для захисту даних на всіх етапах обробки.

Одним з найбільш поширених методів шифрування, що використовується в мікросервісах, — це протокол SSL/TLS, який гарантує, що вся інформація, що передається між клієнтами та сервісами, а також між самими мікросервісами, буде захищена від несанкціонованого доступу. Враховуючи, що сервіси можуть працювати в різних середовищах і мати різні точки доступу, TLS шифрування гарантує, що канали зв'язку будуть захищені навіть у разі перехоплення трафіку зловмисниками.

SSL (Secure Sockets Layer) і TLS (Transport Layer Security) — це криптографічні протоколи, які забезпечують захищене з'єднання для передачі даних по незахищених мережах, таких як Інтернет. SSL був початковим протоколом, однак через виявлені вразливості його було замінено на більш надійний TLS, хоча термін SSL часто використовують для позначення обох протоколів.

Основна мета TLS — забезпечити конфіденційність та цілісність переданих даних. Коли два мікросервіси обмінюються інформацією, використовуючи TLS, вони спочатку виконують процес рукопожаття (handshake), який включає в себе:

Перевірку сертифікатів: кожен сервіс підтверджує свою ідентичність за допомогою цифрового сертифіката, що дозволяє уникнути атак типу "man-in-the-middle".

Обмін ключами: для шифрування даних використовується сесійний ключ, який генерується під час рукопожаття, що забезпечує швидке та безпечне шифрування даних.

Шифрування даних: після цього обмін інформацією відбувається через зашифровані канали, що захищає її від перехоплення і модифікацій.

Переваги використання SSL/TLS:

Конфіденційність: усі дані, що передаються між сервісами, шифруються. Це означає, що навіть якщо зловмисник перехопить трафік, він не зможе отримати доступ до змісту даних.

Цілісність: SSL/TLS використовує контрольні суми (hashing), що дозволяє перевірити, чи не були змінені дані під час передачі.

Аутентифікація: завдяки сертифікатам, що використовуються під час рукопожаття, кожен сервіс може підтвердити свою особу. Це значно знижує ймовірність атак на типу "man-in-the-middle", коли зловмисник підробляє один із сервісів і перехоплює або змінює передані дані.

Безпека при відкритих каналах: у разі, якщо зловмисник намагається перехопити трафік у публічних або відкритих мережах, використання TLS забезпечує захист від подібних атак.

Використання SSL/TLS у мікросервісах:

У контексті мікросервісів, де сервіси часто взаємодіють через відкриті API або інші протоколи, TLS є стандартним способом забезпечення безпечної комунікації між компонентами системи. Всі API-запити та відповіді, що проходять через мікросервіси, можуть бути зашифровані, забезпечуючи їхній захист на кожному етапі передавання.

Застосування TLS гарантує, що між сервісами передаються тільки захищені дані, і навіть у випадку витоку даних із сервісу або зловмисного втручання в мережу, інформація залишатиметься недоступною для сторонніх осіб.

Інтеграція TLS у мікросервісну архітектуру:

Автоматичне шифрування: Багато інструментів і платформ для мікросервісів автоматично підтримують TLS. Наприклад, Kubernetes має вбудовану підтримку TLS для забезпечення безпеки між контейнерами, а також дозволяє автоматично генерувати та обмінювати сертифікати для захищених з'єднань.

API Gateway: У мікросервісних архітектурах часто використовується API Gateway, який виступає єдиною точкою входу для всіх запитів. Всі з'єднання з клієнтами через API Gateway можуть бути зашифровані за допомогою TLS, забезпечуючи високий рівень захисту. Це дозволяє централізовано керувати безпекою всіх комунікацій між сервісами.

Секрети та сертифікати: Важливо правильно керувати сертифікатами та секретами, що використовуються для TLS. Рішення для автоматичного оновлення та управління сертифікатами, наприклад, Let's Encrypt, можуть бути інтегровані в CI/CD процеси, щоб гарантувати актуальність і безпеку сертифікатів.

Використання SSL/TLS в розподілених системах:

У великих розподілених системах, де сервіси можуть бути фізично розміщені на різних географічних локаціях, TLS дозволяє забезпечити захист даних не лише між мікросервісами всередині одного дата-центру, але й під час передачі між різними регіонами чи навіть хмарними провайдерами. Це особливо важливо для систем, які потребують високої безпеки та надійності, таких як фінансові або медичні сервіси, де кожен витік або модифікація даних може мати серйозні наслідки.

SSL/TLS є основним протоколом для забезпечення безпеки передачі даних у мікросервісах. Завдяки своїм механізмам шифрування, аутентифікації та цілісності, він забезпечує захист від перехоплення та зміни даних, що робить його критично важливим компонентом для захищених мікросервісних архітектур.

3.4 Захист від загроз: моніторинг, виявлення та реагування

У мікросервісних архітектурах, де система складається з великої кількості незалежних компонентів, забезпечення безпеки стає однією з головних задач. Кожен мікросервіс взаємодіє з іншими через відкриті канали зв'язку, тому важливими складовими захисту є моніторинг, виявлення загроз та реагування на них. Без належного контролю та здатності швидко реагувати на інциденти навіть найкраще спроектовані системи можуть стати вразливими до атак. Моніторинг безпеки в мікросервісних архітектурах є важливою частиною загальної стратегії захисту. Зважаючи на те, що ці архітектури складаються з безлічі мікросервісів, кожен з яких може перебувати в різних середовищах або бути розгорнутий на різних фізичних чи віртуальних машинах, для своєчасного виявлення загроз необхідно постійно відслідковувати активність усіх компонентів. Це дозволяє швидко виявляти потенційні загрози, як-от спроби несанкціонованого доступу до ресурсів або аномалії в поведінці сервісів. Одним із основних завдань є виявлення підозрілих дій або аномальних патернів трафіку, таких як надмірне навантаження на конкретний сервіс, збільшення кількості помилок у запитах або підозрілі IP-адреси, які можуть свідчити про спроби зловмисників атакувати систему.

Моніторинг реалізується через використання спеціалізованих інструментів для збору та аналізу даних, таких як платформи для збору логів або системи для моніторингу метрик. Вони дають змогу отримувати повну картину активності всіх компонентів системи. Наприклад, платформи як ELK Stack (Elasticsearch, Logstash, Kibana) дозволяють збирати логи з усіх мікросервісів, що дає змогу швидко виявити підозрілі дії або аномалії. Крім того, системи моніторингу, такі як Prometheus і Grafana, дозволяють отримувати детальну інформацію про стан кожного сервісу, показники його продуктивності і завантаженості, що також допомагає ідентифікувати потенційні загрози. Що стосується виявлення загроз, то тут мова йде про використання спеціалізованих систем для автоматичного виявлення аномалій або відомих атак. Виявлення аномалій дозволяє виявляти нові, раніше невідомі типи атак, які можуть не бути зафіксовані традиційними сигнатурами. Це дозволяє швидко реагувати на зміни в поведінці системи і вчасно нейтралізувати загрози. Однак, для більш ефективного виявлення відомих атак можна використовувати сигнатурні методи, які шукають заздалегідь визначені шаблони, такі як спроби SQL-ін'єкцій, XSS-атаки або інші типи атак, що вже відомі в галузі безпеки.

Реагування на загрози є критично важливим етапом, який визначає, наскільки швидко і ефективно можна усунути небезпеку та мінімізувати шкоду від інциденту. Коли загроза виявлена, система повинна мати змогу швидко відреагувати, щоб захистити дані та обмежити поширення атаки. Реагування може включати в себе автоматичне блокування доступу до скомпрометованих сервісів або ресурсів, відновлення безпеки системи, а також сповіщення адміністраторів або відповідальних осіб для подальшого розслідування інциденту. Швидке реагування на загрози дозволяє не тільки захистити дані, але й мінімізувати час, протягом якого зловмисник може впливати на систему. Наприклад, автоматичне блокування доступу до скомпрометованих компонентів або обмеження їх функціональності може істотно знизити ризик поширення атаки на інші частини системи. Важливою

складовою реагування є також можливість швидкого відновлення роботи сервісів після інциденту, що дозволяє зберігати безперервність функціонування системи.

Використання інструментів для моніторингу, виявлення та реагування на загрози дозволяє створити більш стійку і безпечну інфраструктуру. Це дає змогу не лише вчасно виявляти потенційні проблеми, а й ефективно і швидко їх усувати, забезпечуючи безпеку даних і стабільну роботу мікросервісної системи в цілому. Важливою частиною цього процесу є створення плану реагування на інциденти, що визначає конкретні кроки і процедури, які повинні бути виконані в разі виникнення загрози.

Це дає можливість не тільки швидко виявляти та реагувати на загрози, а й значно покращує рівень захисту системи в цілому, забезпечуючи стійкість до потенційних атак і зловживань. Оперативне реагування та постійний моніторинг дозволяють створити безпечну інфраструктуру, де кожен інцидент може бути виявлений і нейтралізований у найкоротші терміни, забезпечуючи таким чином цілісність і конфіденційність даних..

3.5 Висновки по розділу

У цьому розділі було розглянуто основні методи забезпечення безпеки в мікросервісних архітектурах, зокрема важливість шифрування даних, захисту каналів зв'язку та ефективного моніторингу і виявлення загроз. Використання таких методів, як SSL/TLS шифрування, асиметричне і симетричне шифрування, а також централізований моніторинг і виявлення аномалій, дозволяє забезпечити високий рівень безпеки в умовах складних і масштабованих систем. Важливим аспектом є також оперативне реагування на загрози, що дозволяє швидко локалізувати інциденти та мінімізувати їхні наслідки.

Без належного захисту, моніторингу і швидкої реакції, мікросервісні архітектури можуть бути уразливими до атак, що призводить до потенційних витоків даних або збоїв у роботі системи. Тому інтеграція всіх цих методів є необхідною для створення надійної і стійкої інфраструктури..

4 ВПРОВАДЖЕННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

4.1 Принципи розгортання мікросервісів за допомогою Kubernetes

Розгортання мікросервісів за допомогою Kubernetes є однією з найефективніших методик управління складними розподіленими системами, яка дозволяє організаціям реалізовувати масштабовані, стабільні та безпечні програмні рішення. Kubernetes надає широкий набір можливостей, зокрема автоматизацію управління ресурсами, забезпечення високої доступності додатків і оперативне масштабування інфраструктури залежно від потреб. Завдяки своїй здатності оркеструвати контейнери, Kubernetes забезпечує гнучке управління робочими навантаженнями, яке включає балансування навантаження між вузлами, відновлення роботи після збоїв та оптимізацію використання ресурсів. Це створює умови для підтримки стабільної роботи сервісів навіть під час пікових навантажень. Крім того, інтеграція Kubernetes із системами моніторингу (Prometheus, Grafana) та засобами забезпечення безпеки дозволяє компаніям оперативно реагувати на потенційні проблеми, аналізувати ефективність роботи та захищати свої дані.

Особливе значення має його здатність працювати як у локальних середовищах, так і в хмарних інфраструктурах, забезпечуючи узгодженість і повторюваність процесів незалежно від середовища розгортання. Це робить Kubernetes універсальним рішенням для сучасних програмних систем, що дозволяє компаніям швидко адаптуватися до змін ринку, підвищувати конкурентоспроможність та знижувати витрати на підтримку інфраструктури.

Основні принципи розгортання:

Контейнеризація: Усі мікросервіси повинні бути упаковані в контейнери, що забезпечує їхню ізоляцію та незалежність від основного середовища. Docker є найпоширенішим інструментом для створення контейнерів, які можна легко інтегрувати з Kubernetes.

Опис інфраструктури як коду: Kubernetes використовує YAML-файли для опису конфігурації мікросервісів, включаючи налаштування реплік, зберігання,

мережевих з'єднань і політик доступу. Це забезпечує повторюваність і передбачуваність процесу розгортання.

Приклад YAML-файлу для розгортання мікросервісу:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-microservice
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example-app
  template:
    metadata:
      labels:
        app: example-app
    spec:
      containers:
      - name: example-container
        image: example/microservice:latest
        ports:
        - containerPort: 80
        env:
        - name: ENVIRONMENT
          value: production
```

Оркестрація ресурсів: Kubernetes автоматично розподіляє ресурси між вузлами кластера, забезпечуючи оптимальне використання процесора, пам'яті та інших ресурсів. Завдяки цьому розгортання нових мікросервісів або масштабування існуючих компонентів здійснюється без впливу на стабільність системи.

Моніторинг і логування: Важливим аспектом є інтеграція Kubernetes із системами моніторингу, такими як Prometheus і Grafana, що дозволяє відстежувати стан мікросервісів у реальному часі, аналізувати продуктивність і оперативно реагувати на збої.

Розгортання мікросервісів за допомогою Kubernetes забезпечує високу надійність і масштабованість системи завдяки автоматизованим механізмам управління ресурсами та безперервного моніторингу. Це дозволяє організаціям не лише підтримувати стабільну роботу додатків у хмарних середовищах, але й ефективно розподіляти навантаження, уникати простоїв та швидко реагувати на непередбачувані ситуації. Інтеграція з такими інструментами, як Prometheus для моніторингу та Grafana для візуалізації, забезпечує прозорість у роботі додатків і сприяє оперативному прийняттю рішень. Такий підхід також дозволяє компаніям легко масштабувати свої додатки у відповідь на зростання попиту, використовуючи горизонтальне та вертикальне масштабування, що оптимізує використання ресурсів і забезпечує безперервність роботи. Горизонтальне масштабування дозволяє додавати нові екземпляри мікросервісів для збільшення їх пропускної здатності, тоді як вертикальне масштабування оптимізує потужність існуючих вузлів, додаючи більше обчислювальних ресурсів.

Орієнтованість Kubernetes на стандарти, такі як контейнеризація та інфраструктура як код, сприяє спрощенню інтеграції з широким спектром інструментів і платформ, зокрема хмарних провайдерів, систем моніторингу та CI/CD. Наприклад, інтеграція з Terraform дозволяє автоматизувати процес розгортання інфраструктури, а Helm спрощує управління конфігураціями мікросервісів. Крім того, завдяки підтримці Rolling Updates і Canary Deployments Kubernetes забезпечує плавний перехід на нові версії додатків без ризику збоїв у роботі. Крім масштабування, Kubernetes активно підтримує механізми автоматичного відновлення, які дозволяють мінімізувати наслідки можливих збоїв. У разі виходу з ладу одного з вузлів платформа автоматично перерозподіляє ресурси та відновлює роботу сервісів. Завдяки цій гнучкості Kubernetes є ключовим

компонентом для побудови надійних і адаптивних IT-систем, що сприяє підвищенню продуктивності бізнесу, зниженню експлуатаційних витрат і забезпеченню конкурентних переваг на ринку..

4.2 Аналіз сучасних методів забезпечення безпеки: OpenID Connect та JWT

Для забезпечення безпеки у мікросервісній архітектурі використовуються різні підходи та інструменти, які допомагають знизити ризики та забезпечити безпечну взаємодію між компонентами системи. OpenID Connect (OIDC) та JSON Web Tokens (JWT) є ключовими технологіями, що дозволяють ефективно вирішувати завдання як аутентифікації, так і авторизації, забезпечуючи баланс між простотою інтеграції, безпекою та продуктивністю. OpenID Connect (OIDC) працює на основі OAuth 2.0, додаючи рівень перевірки особи користувача через централізованого Identity Provider. Це дозволяє створювати єдину точку входу для різних сервісів і забезпечувати єдині політики безпеки в рамках розподіленої архітектури. JSON Web Tokens (JWT), у свою чергу, забезпечують легку та швидку передачу аутентифікаційних даних між компонентами системи за рахунок безстанової моделі та компактного формату. Вибір між OpenID Connect та JWT залежить від багатьох факторів, включаючи масштаб системи, вимоги до рівня безпеки, типи додатків, а також необхідність централізованого управління доступом. Обидва методи мають свої сильні сторони та обмеження, які варто враховувати при розробці сучасних мікросервісних систем. Нижче наведено детальний аналіз у вигляді порівняльної таблиці (табл.4.1).

Переваги OpenID Connect:

Забезпечує вищий рівень безпеки через централізований механізм аутентифікації, який дозволяє централізовано перевіряти особу користувача та управляти доступом до різних сервісів у межах розподіленої архітектури. Це забезпечує консистентність політик безпеки, можливість управління сесіями користувачів у реальному часі та захист від несанкціонованих спроб доступу.

Централізація також спрощує інтеграцію з багатофакторними методами аутентифікації, такими як SMS-коди чи біометрія, що додатково підвищує рівень захисту корпоративних систем. Підходить для комплексних корпоративних систем, які вимагають високого рівня безпеки, централізованого управління доступом та підтримки складних сценаріїв аутентифікації. Завдяки інтеграції OpenID Connect з багатьма сучасними технологіями, такими як багатофакторна аутентифікація (MFA), управління сесіями та моніторинг безпеки, ця технологія дозволяє легко масштабувати системи, адаптувати їх до вимог міжнародних стандартів безпеки, таких як GDPR чи ISO/IEC 27001, і знижувати операційні ризики.

Підтримує різні типи токенів (ID Token, Access Token), які забезпечують гнучкість у використанні та адаптацію до різних сценаріїв взаємодії між сервісами. ID Token служить для передачі інформації про аутентифікованого користувача, що дозволяє сервісам отримувати дані про його особу та права доступу. Access Token забезпечує доступ до ресурсів або API, спрощуючи процес авторизації та зменшуючи ризик розкриття конфіденційної інформації. Ця багатофункціональність дозволяє OpenID Connect ефективно інтегруватися з системами будь-якого масштабу, від невеликих веб-додатків до великих корпоративних платформ.

Переваги JWT:

Простота використання та швидкість обробки токенів роблять JWT надзвичайно зручним інструментом для розробників, які працюють із розподіленими системами. Завдяки компактному формату, токени легко передаються через мережу, що мінімізує затримки та оптимізує продуктивність API. Крім того, їхня безстановість дозволяє уникати складного управління сесіями на сервері, що особливо важливо для систем із високою кількістю запитів. Використання стандартних алгоритмів шифрування, таких як HS256 чи RS256, забезпечує належний рівень безпеки при мінімальних витратах ресурсів. Підходить для безстанового обміну даними між сервісами, забезпечуючи ефективну передачу інформації між компонентами без потреби у збереженні стану на стороні сервера.

Завдяки цій властивості, JWT ідеально підходить для систем із великою кількістю запитів, таких як API, оскільки зменшує навантаження на сервер і підвищує швидкість обробки даних. Токени містять всю необхідну інформацію для автентифікації та авторизації, що робить їх незалежними від серверних сесій і дозволяє легко масштабувати систему, зберігаючи її продуктивність навіть під час пікових навантажень.

Легко інтегрується в API, що дозволяє розробникам без додаткових витрат на складну конфігурацію інтегрувати JWT у існуючі системи. Завдяки цьому токени стають ефективним засобом автентифікації та авторизації для різноманітних додатків, таких як мобільні додатки, веб-додатки, а також системи з високою частотою API-запитів. Простота інтеграції також сприяє швидшому впровадженню нових функцій, що є критично важливим для швидкорозвиваючихся проектів.

Таблиця 4.1 – Порівняльна таблиця

Характеристика	OpenID Connect (OIDC)	JSON Web Tokens (JWT)
Призначення	Аутентифікація користувачів і делегування доступу	Перенесення аутентифікаційних даних
Стандарт/Протокол	Побудований на основі OAuth 2.0	Відкритий стандарт для створення токенів
Механізм роботи	Використовує Identity Provider для перевірки особи	Токени кодуються у форматі Base64
Підтримка авторизації	Є	Часткова (залежить від реалізації)
Зберігання стану	Підтримує стан через Identity Provider	Безстанова структура
Термін дії токенів	Динамічно керується	Фіксований (залежить від налаштувань)
Рівень безпеки	Високий (включає багаторівневі механізми)	Залежить від алгоритму шифрування (HS256, RS256)
Сценарії використання	Мобільні додатки, веб-додатки, корпоративні системи	API, взаємодія між сервісами

Залежно від потреб системи, вибір між OpenID Connect та JWT може залежати від багатьох факторів, які варто ретельно враховувати при проєктуванні архітектури. Для великих корпоративних систем, де важливим є централізований контроль доступу, управління сесіями користувачів і суворе дотримання політик безпеки, OpenID Connect стає оптимальним вибором. Його можливість інтеграції з

існуючими системами аутентифікації та підтримка багатофакторної автентифікації робить його незамінним для організацій з високими вимогами до безпеки. JWT, зі свого боку, демонструє значну перевагу у швидкості обробки запитів і простоті реалізації. Це рішення ідеально підходить для систем, де важливий безстанний обмін даними, наприклад, для інтеграції мікросервісів або забезпечення доступу до API. Його легка інтеграція з мовами програмування та високий рівень масштабованості роблять його ефективним інструментом для проєктів з невеликим обсягом користувацьких даних або систем, орієнтованих на API-запити.

Враховуючи сучасні тенденції розробки, багато компаній комбінують використання OpenID Connect та JWT для досягнення оптимального балансу між безпекою, продуктивністю та зручністю інтеграції. Наприклад, OpenID Connect використовується для первинної аутентифікації користувачів, після чого JWT забезпечує безпечний обмін даними між сервісами. Такий підхід дозволяє не лише підвищити загальний рівень безпеки системи, але й зменшити затримки при взаємодії між її компонентами.

4.3 Оцінка методів тестування навантаження мікросервісів

Тестування навантаження є ключовим аспектом забезпечення продуктивності, надійності та масштабованості мікросервісної архітектури. У розподілених системах кожен мікросервіс виконує специфічну функцію, і його ефективність має безпосередній вплив на стабільність, час відгуку та загальну продуктивність системи. Перевірка поведінки мікросервісів під різними типами навантаження, включаючи раптові пікові запити, тривале високонавантажене використання або нестандартні сценарії взаємодії, є критично важливою для забезпечення стабільної роботи системи. Тестування дозволяє не лише виявляти потенційні вузькі місця, але й формувати стратегії для покращення управління ресурсами та вдосконалення архітектурного дизайну. Наприклад, аналіз точок відмови під час стрес-тестів може допомогти визначити необхідність у додаткових обчислювальних ресурсах або зміні механізмів балансування навантаження.

Водночас тестування продуктивності сприяє оптимізації часу відгуку сервісів, що є критичним для забезпечення позитивного досвіду користувачів. Завдяки регулярному тестуванню навантаження можна забезпечити відповідність роботи мікросервісів сучасним стандартам продуктивності, створюючи передумови для їхньої безперебійної роботи навіть у найскладніших умовах експлуатації. Це стає особливо важливим для організацій, які прагнуть масштабувати свої системи, реагуючи на зростаючі потреби клієнтів, а також для компаній, що працюють у сферах із високими вимогами до доступності та швидкості обслуговування.

Основні методи тестування навантаження

Стрес-тестування: Цей метод дозволяє оцінити, як мікросервіси працюють під умовами максимального навантаження, включаючи симуляцію надзвичайно високого трафіку або одночасних запитів від великої кількості користувачів. Під час тестування визначається точка відмови системи, тобто максимальне навантаження, яке вона може витримати до збоїв. Також аналізуються механізми відновлення після перевантаження, такі як автоматичне масштабування ресурсів, балансування навантаження та відновлення стану мікросервісів. Крім того, стрес-тестування дозволяє перевірити, як система поводиться за умов довготривалого навантаження, що може виявити проблеми, пов’язані з витоками пам’яті чи накопиченням некоректних даних у чергах запитів.

Тестування продуктивності: Використовується для оцінки часу відгуку, швидкості обробки запитів та ефективності роботи сервісу за нормальних умов експлуатації. Цей вид тестування дозволяє визначити, наскільки система відповідає встановленим SLA (Service Level Agreement), а також виявити потенційні вузькі місця у її роботі. Наприклад, тестування може показати, чи здатний сервіс обробляти запити в межах заданого часу, зберігаючи при цьому стабільність і точність результатів. Тестування продуктивності також дозволяє проводити порівняльний аналіз різних версій програмного забезпечення, оцінюючи вплив оптимізацій або змін в архітектурі на загальну продуктивність. Регулярне

проведення цього тестування допомагає забезпечити надійну роботу системи, навіть якщо умови експлуатації змінюються або навантаження зростає.

Тестування навантаження: Використовується для симуляції реальних умов експлуатації, що дозволяє визначити, як система обробляє запити за різних рівнів активності. Цей метод тестування допомагає імітувати поведінку реальних користувачів, перевіряючи, як система справляється з типовими та атиповими сценаріями взаємодії. Наприклад, можна моделювати одночасні запити від великої кількості користувачів або створювати навантаження у різних часових інтервалах. Це дозволяє не лише оцінити поточну продуктивність, але й визначити, наскільки система готова до масштабування або роботи під піковим навантаженням. Крім того, тестування навантаження дозволяє прогнозувати потенційні точки відмови та оптимізувати ресурси перед запуском системи у реальну експлуатацію.

Тестування на стійкість: Перевіряється, як система поводиться у разі часткової відмови одного чи кількох компонентів, включаючи аналіз її здатності відновлюватися та забезпечувати безперебійну роботу. Цей тип тестування є критичним для визначення надійності системи в умовах реальних збоїв, таких як відмова окремих серверів, проблеми з мережею чи недоступність зовнішніх ресурсів. Завдяки тестуванню на стійкість виявляються вузькі місця, що можуть впливати на стабільність мікросервісів, а також перевіряються механізми автоматичного масштабування, повторних спроб підключення та балансування навантаження. Такий підхід дозволяє забезпечити, що навіть під час критичних відмов основні сервіси залишаються доступними для користувачів, а система здатна самостійно адаптуватися до непередбачуваних умов.

Інструменти для тестування навантаження

Apache JMeter: Один із найпоширеніших інструментів для моделювання навантаження, що забезпечує широкий спектр можливостей для тестування як окремих мікросервісів, так і комплексних систем. JMeter дозволяє створювати сценарії для оцінки продуктивності під різними типами навантаження, включаючи стрес-тестування, тестування на стійкість та симуляцію реальних користувачів.

Завдяки своїй підтримці протоколів HTTP, HTTPS, FTP, JDBC та інших, інструмент є універсальним вибором для багатьох сценаріїв. Крім того, його гнучкість у налаштуванні, можливість інтеграції з CI/CD пайплайнами та підтримка плагінів значно розширюють його функціонал.

Gatling: Високопродуктивний інструмент, орієнтований на тестування веб-додатків і API, який дозволяє моделювати реалістичні сценарії взаємодії користувачів із системою. Основними перевагами Gatling є його здатність обробляти великий обсяг запитів за короткий час, що робить його ідеальним для стрес-тестування і оцінки продуктивності високонавантажених систем. Інструмент підтримує створення сценаріїв на основі мови Scala, що дозволяє легко налаштовувати складні сценарії взаємодії. Gatling також має вбудовані функції візуалізації результатів у реальному часі, що сприяє оперативному аналізу роботи системи. Інтеграція з CI/CD пайплайнами дозволяє автоматизувати тестування, забезпечуючи безперервне вдосконалення продуктивності системи.

Locust: Потужний і гнучкий інструмент для створення сценаріїв навантаження, написаних на Python. Завдяки своїй простоті у використанні та масштабованості, Locust дозволяє імітувати реальні умови експлуатації систем, моделюючи поведінку тисяч одночасних користувачів. Основними перевагами є можливість динамічного створення складних сценаріїв, інтеграція з іншими інструментами Python, а також підтримка розподіленого тестування для великих систем. Locust широко застосовується для перевірки продуктивності API, веб-додатків і мікросервісів, дозволяючи розробникам і адміністраторам оперативно виявляти вузькі місця та оптимізувати роботу системи. Вбудовані механізми збору статистики й візуалізації результатів сприяють швидкому аналізу продуктивності.

Приклад сценарію для Locust

```
from locust import HttpUser, TaskSet, task
# Визначаємо набір завдань для моделювання поведінки користувачів
class UserBehavior(TaskSet):
    @task
```

```
        def index(self):
            # Запит до головної сторінки
            self.client.get("/")
        @task
        def about_page(self):
            # Запит до сторінки 'Про нас'
            self.client.get("/about")
    # Основний клас, що визначає користувачів і час очікування між запитами
    class WebsiteUser(HttpUser):
        tasks = [UserBehavior]
        # Встановлюємо мінімальний і максимальний час очікування між запитами
        min_wait = 5000
        max_wait = 15000
```

Етапи проведення тестування навантаження:

1. Визначення цілей тестування (наприклад, максимальне навантаження, при якому система залишається стабільною).
2. Створення сценаріїв для тестування.
3. Запуск тестів із поступовим збільшенням навантаження.
4. Аналіз отриманих результатів та виявлення вузьких місць.
5. Оптимізація системи на основі отриманих даних.

Тестування навантаження дозволяє забезпечити високу продуктивність мікросервісної архітектури, адаптуючи її до реальних умов експлуатації. Воно допомагає виявляти слабкі місця в роботі системи, наприклад, неефективне використання ресурсів, неочікувані збої під час пікових навантажень або повільну обробку запитів. Завдяки цьому організації можуть уникнути простоїв, що мають критичне значення для бізнесу, особливо в умовах жорсткої конкуренції. Регулярне тестування сприяє підвищенню рівня задоволеності користувачів завдяки мінімізації затримок у роботі сервісів і забезпеченню надійності. Наприклад, у

сфері електронної комерції або фінансових послуг кожна секунда затримки може призводити до втрати клієнтів і доходів. Тестування навантаження також дозволяє прогнозувати необхідні ресурси для масштабування, що важливо для компаній, які планують розширення свого бізнесу.

Даний підхід гарантує, що мікросервісна архітектура відповідатиме сучасним вимогам ринку, забезпечуючи високу доступність, продуктивність і стабільність системи навіть у складних умовах експлуатації..

4.4 Рекомендації щодо покращення безпеки та продуктивності

Покращення безпеки та продуктивності є критичними аспектами для забезпечення ефективності, стабільності та масштабованості роботи мікросервісної архітектури. У сучасному світі, де цифрові технології постійно вдосконалюються, забезпечення безпеки та продуктивності стає ключовим фактором успіху будь-якої системи. Зростання складності додатків та обсягу даних вимагає ретельного підходу до захисту інформації, управління ресурсами та оптимізації процесів. Ефективна безпека передбачає багаторівневий підхід, який включає аутентифікацію користувачів, шифрування даних, реалізацію політик доступу та контроль за станом системи. Продуктивність, у свою чергу, залежить від правильної організації ресурсів, автоматизації процесів масштабування, забезпечення швидкого відгуку системи та рівномірного розподілу навантаження між компонентами. Дотримання сучасних стандартів безпеки, таких як ISO/IEC 27001, а також інтеграція інструментів моніторингу та аналітики дозволяють підвищити прозорість процесів і забезпечити своєчасну реакцію на потенційні загрози. Нижче наведено детальні рекомендації, які сприяють підвищенню надійності, адаптивності та продуктивності системи у різноманітних умовах експлуатації:

1. Впровадження багатофакторної аутентифікації: Забезпечує додатковий рівень захисту, вимагаючи від користувачів підтвердження через кілька незалежних методів, таких як SMS-коди або біометричні дані.

2. Використання шифрування: Усі дані, що передаються між мікросервісами, мають бути зашифровані за допомогою сучасних алгоритмів, таких як TLS 1.3, для захисту від перехоплення чи несанкціонованого доступу.
3. Реалізація політик безпеки на рівні API: Використовуйте токени доступу, такі як OAuth 2.0 або JWT, для аутентифікації та авторизації запитів до API.
4. Моніторинг і логування: Інтегруйте системи моніторингу (Prometheus, Grafana) та централізованого логування (ELK Stack), що дозволяють виявляти загрози в реальному часі та швидко реагувати на інциденти.
5. Масштабування ресурсів: Використовуйте автоматичне масштабування в Kubernetes для адаптації системи до змінних умов навантаження, забезпечуючи стабільну роботу навіть під час пікових запитів.
6. Оптимізація часу відгуку: Регулярно проводьте тестування продуктивності, виявляючи та усуваючи вузькі місця в роботі мікросервісів.
7. Розподіл навантаження: Реалізуйте балансування навантаження за допомогою таких інструментів, як NGINX або HAProxy, що забезпечує рівномірний розподіл запитів між сервісами.
8. Впровадження політик доступу: Використовуйте рольові моделі (RBAC) для забезпечення доступу до критичних даних лише авторизованим користувачам.
9. Резервне копіювання та відновлення: Регулярно створюйте резервні копії даних та перевіряйте їхню цілісність, щоб забезпечити швидке відновлення у разі збоїв.
10. Використання контейнерної ізоляції: Кожен мікросервіс має працювати у власному контейнері з обмеженими правами доступу, що зменшує ризик поширення загроз у системі.

Реалізація цих рекомендацій дозволить не лише підвищити рівень безпеки та продуктивності, а й забезпечить формування стійкої, адаптивної та довготривалої архітектури системи. Інтеграція сучасних методів захисту, таких як багатофакторна аутентифікація та контейнерна ізоляція, дозволить мінімізувати ризики, пов’язані з несанкціонованим доступом або витоком даних. Автоматизація управління ресурсами, реалізована через інструменти, такі як Kubernetes, забезпечить оперативну адаптацію системи до змін навантаження без шкоди для стабільності. Прозорість процесів, яку забезпечують сучасні системи моніторингу та логування, дозволить швидко ідентифікувати потенційні проблеми та усувати їх до того, як вони вплинуть на продуктивність. Крім того, реалізація політик доступу на основі RBAC сприятиме чіткому розмежуванню ролей і відповідальностей, що важливо для дотримання стандартів безпеки. Масштабування системи з урахуванням прогнозованих навантажень дає змогу забезпечити безперебійну роботу навіть у періоди пікової активності користувачів. Завдяки регулярному резервному копіюванню та тестуванню сценаріїв відновлення організації можуть уникнути тривалих простоїв і мінімізувати втрати у разі технічних збоїв.

У підсумку, виконання цих заходів дозволить організаціям створити ефективну інфраструктуру, яка не лише відповідає сучасним викликам, а й має значний потенціал для розвитку в майбутньому. Це забезпечить не тільки довіру користувачів, але й високу конкурентоспроможність на ринку..

4.5 Висновки по розділу

У розділі було проаналізовано ключові аспекти впровадження мікросервісної архітектури, зокрема принципи її розгортання, тестування навантаження, забезпечення безпеки та рекомендації щодо оптимізації продуктивності. Визначено, що успішне впровадження мікросервісів вимагає комплексного підходу, який охоплює як технічні аспекти, так і організаційні заходи.

Розгортання мікросервісів за допомогою Kubernetes: забезпечує автоматизацію управління ресурсами, масштабованість і стабільність роботи систем навіть у складних умовах.

Методи тестування навантаження: дозволяють оцінити продуктивність та виявити потенційні вузькі місця для їхньої подальшої оптимізації.

Аналіз методів безпеки: підтверджує важливість впровадження багаторівневих механізмів захисту, таких як шифрування, політики доступу та багатофакторна аутентифікація.

Рекомендації щодо покращення продуктивності: акцентують увагу на важливості інтеграції моніторингових інструментів, автоматичного масштабування та оптимізації ресурсів.

Виконання перелічених заходів не лише підвищить надійність та ефективність мікросервісної архітектури, але й створить основу для її подальшого розвитку, відповідаючи сучасним вимогам безпеки, продуктивності та адаптивності.

ВИСНОВКИ

Дослідження методів розгортання мікросервісної архітектури та методів безпеки для веб-додатків підтвердило, що сучасні технології дозволяють значно підвищити ефективність, надійність і продуктивність програмного забезпечення. Мікросервісна архітектура, яка базується на використанні контейнеризації, оркестрації та автоматизації розгортання, демонструє високий рівень гнучкості, що забезпечує адаптацію до швидко змінних умов ринку та вимог користувачів. Впровадження багаторівневих механізмів безпеки дозволяє захищати дані та запобігати несанкціонованому доступу, що є критично важливим у сучасних реаліях кіберзагроз. Дана архітектура також сприяє оптимальному використанню ресурсів, зниженню витрат на обслуговування інфраструктури та забезпеченню стабільної роботи навіть у періоди високих навантажень. Завдяки застосуванню таких технологій, як Docker і Kubernetes, компанії отримують інструменти для ефективного управління своїми додатками, забезпечуючи їхню масштабованість та ізоляцію сервісів. Водночас автоматизація процесів розгортання за допомогою Ansible, Terraform, Helm і Jenkins спрощує операції та зменшує ризики, пов'язані з людським фактором. Впроваджені механізми безпеки, включаючи шифрування, токенізацію та рольові моделі доступу, гарантують конфіденційність і цілісність даних, що є базовими вимогами для сучасних веб-додатків. Результати дослідження підтверджують, що інтеграція систем моніторингу та тестування продуктивності дозволяє оперативно виявляти вузькі місця та адаптувати систему до зростання навантаження.Використання мікросервісної архітектури разом із сучасними методами безпеки є основою для створення конкурентоспроможних, стійких та інноваційних рішень, які здатні відповідати вимогам як поточного, так і майбутнього розвитку інформаційних технологій.

Контейнеризація та оркестрація: Технології, такі як Docker і Kubernetes, забезпечують ізоляцію сервісів, ефективне управління ресурсами та гнучке масштабування систем.

Методи автоматизації: Інструменти Ansible, Terraform, Helm та Jenkins оптимізують процеси розгортання та забезпечують повторюваність і передбачуваність операцій.

Безпека в мікросервісах: Використання протоколів OpenID Connect та JWT, шифрування даних і рольові моделі доступу гарантують захист інформації та запобігають несанкціонованому доступу.

Тестування навантаження: Методи оцінки продуктивності дозволяють виявити вузькі місця в роботі системи, оптимізувати її ресурси та забезпечити стабільну роботу навіть за умов пікових навантажень.

Запропоновані підходи та інструменти є основою для створення стійких веб-додатків, які здатні адаптуватися до змін умов експлуатації, забезпечувати високий рівень безпеки та відповідати сучасним вимогам продуктивності. Реалізація цих рішень не лише підвищує конкурентоспроможність програмних продуктів, але й формує основу для довгострокового розвитку інформаційних технологій.

ПЕРЕЛІК ПОСИЛАНЬ

1. Newman, S. Building Microservices: Designing Fine-Grained Systems. – O'Reilly Media, 2015. – 280 p.
2. Burns, B., Beda, J., Hightower, K. Kubernetes: Up and Running: Dive into the Future of Infrastructure. – O'Reilly Media, 2019. – 280 p.
3. Turnbull, J. The Docker Book: Containerization is the new virtualization. – James Turnbull, 2014. – 348 p.
4. Linders, B., Skelton, M. Continuous Delivery with Docker and Jenkins. – Packt Publishing, 2016. – 372 p.
5. Boyd, A. Terraform: Up & Running: Writing Infrastructure as Code. – O'Reilly Media, 2022. – 300 p.
6. Richardson, C. Microservices Patterns: With examples in Java. – Manning Publications, 2018. – 520 p.
7. Fitzgerald, A. Learning Helm: Managing Apps on Kubernetes. – O'Reilly Media, 2021. – 320 p.
8. Fowler, M. Refactoring: Improving the Design of Existing Code. – Addison-Wesley Professional, 2018. – 448 p.
9. Fowler, M., Lewis, J. Microservices: Principles and Practices. – Addison-Wesley Professional, 2020. – 296 p.
10. Evans, E. Domain-Driven Design: Tackling Complexity in the Heart of Software. – Addison-Wesley, 2003. – 560 p.
11. Beal, A. Jenkins 2: Up and Running: Evolve Your Deployment Pipeline for Next Generation Automation. – O'Reilly Media, 2018. – 306 p.
12. Weiss, G. Introduction to Microservices: A Hands-on Guide for Developers. – Packt Publishing, 2020. – 250 p.
13. Sutton, B. Security and Privacy in Microservices: Practical Techniques for Safe, Scalable Development. – O'Reilly Media, 2020. – 320 p.

14. Potts, T. Kubernetes Security: Protecting Cloud-Native Applications. – Packt Publishing, 2021. – 284 p.
15. Frankel, A. Effective Logging with the ELK Stack: Learn How to Log, Analyze, and Visualize Using Elastic Stack. – Packt Publishing, 2017. – 300 p.
16. Maciaszek, L.A. Requirements Analysis and System Design. – Pearson Education, 2005. – 600 p.
17. McKendrick, J. Cloud-Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment. – O'Reilly Media, 2019. – 260 p.
18. Haeberle, J. Prometheus: Up & Running: Infrastructure and Application Performance Monitoring. – O'Reilly Media, 2021. – 250 p.
19. Annesley, R. Learning ELK Stack. – Packt Publishing, 2016. – 374 p.
20. Parry, B. Distributed Systems Observability: A Guide to Building Robust Systems. – O'Reilly Media, 2019. – 290 p.

ДОДАТОК А. ПРЕЗЕНТАЦІЯ

1

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

ДИПЛОМНА РОБОТА
на ступінь вищої освіти магістр
із спеціальності 122 Комп'ютерні науки

МЕТОДИ РОЗГОРТАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА МЕТОДІВ БЕЗПЕКИ ДЛЯ ВЕБ-ДОДАТКІВ

Виконав: студент 6 курсу, групи КНДМ-62
Сонець Євгеній Анатолійович
Керівник: викладач кафедри Комп'ютерних наук
к.т.н, доцент Василенко В.В.

Київ - 2025

2

ЗАГАЛЬНА ХАРАКТЕРИСТИКА ДИПЛОМНОЇ РОБОТИ

Тема	Методи розгортання мікросервісної архітектури та методів безпеки для веб-додатків
Мета дослідження	проведення глибокого аналізу існуючих методів розгортання мікросервісної архітектури, включаючи контейнери, оркестрацію та автоматизацію, а також дослідження сучасних підходів до забезпечення її безпеки
Наукове завдання	Розробка теоретичних основ і практичних рекомендацій щодо методів розгортання микросерверної архітектури
Об'єкт дослідження	WEB-ресурси, що сприяють розвитку дронів, включаючи їх функціональні можливості та інструменти інтерактивної взаємодіїмікросервісна архітектура, яка представляє собою сучасний підхід до створення веб-додатків шляхом поділу на незалежні сервіси, кожен з яких виконує конкретні функції
Предмет дослідження	Є методи розгортання мікросервісної архітектури, що включають аналіз контейнеризації, оркестрації та інтеграції сервісів, а також сучасні підходи до забезпечення безпеки, зокрема застосування аутентифікації

3

ОСНОВНІ ТЕНДЕНЦІЇ У РОЗРОБЦІ ВЕБ-ДОДАТКІВ ВКЛЮЧАЮТЬ

Переход на мікросервісну архітектуру. Цей підхід забезпечує поділ програмного забезпечення на незалежні сервіси, кожен з яких виконує свою унікальну функцію. Це дозволяє ефективно масштабувати систему, впроваджувати нові функціональні можливості без значних змін у наявній структурі та забезпечувати високу гнучкість у роботі з різними компонентами.

Контейнеризація та оркестрація. Використання Docker, Kubernetes та інших платформ дозволяє автоматизувати процеси розгортання та забезпечити стабільну роботу додатків навіть у складних середовищах, де необхідно дотримуватися високих вимог до масштабованості, доступності та відмовостійкості. Контейнеризація дозволяє ізолювати кожен мікросервіс у власному середовищі, що забезпечує узгодженість роботи незалежно від інфраструктури, на якій він розгорнутий.

Хмарні технології. Веб-додатки дедалі частіше інтегруються з хмарними сервісами, такими як AWS, Azure чи Google Cloud, що сприяє їх глобальній доступності та підвищенню продуктивності. Хмарні платформи надають широкий спектр можливостей, включаючи автоматичне масштабування ресурсів, гнучке управління інфраструктурою та використання технологій штучного інтелекту для аналітики та автоматизації.

Впровадження DevOps. Інтеграція процесів розробки та експлуатації створює основу для безперервного вдосконалення програмного забезпечення через автоматизацію, тісну співпрацю між командами та спрощення процесів тестування та впровадження.

Фокус на безпеку. З огляду на зростаючу кількість кіберзагроз, сучасні веб-додатки впроваджують комплексні механізми захисту на рівні архітектури, які включають багаторівневу аутентифікацію, ефективне шифрування даних та постійний моніторинг уразливостей. Важливим елементом є використання API Gateway для централізованого контролю доступу та захисту потоків даних, а також інтеграція з системами виявлення загроз, такими як IDS та SIEM.

4

ОСНОВНІ ПРИНЦИПИ ПОБУДОВИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

Декомпозиція за функціональністю. Кожен мікросервіс має чітко визначену роль у системі і виконує лише одну функцію. Це забезпечує простоту управління, тестування і розширення системи. Крім того, декомпозиція дозволяє залучати вузькоспеціалізовані команди до роботи над окремими мікросервісами, що покращує загальну якість продукту.

Автономність сервісів. Мікросервіси працюють незалежно один від одного, що дозволяє уникнути проблем із взаємозалежністю. Це сприяє стійкості системи до збоїв і полегшує масштабування. Наприклад, у випадку збою одного сервісу, інші компоненти можуть продовжувати виконувати свої функції без переривання роботи всієї системи.

Комунікація через API. Взаємодія між сервісами здійснюється через стандартизовані інтерфейси API, що дозволяє легко інтегрувати нові компоненти і забезпечує сумісність. Використання RESTful або gRPC API сприяє більш ефективній передачі даних між сервісами, знижуючи затримки та спрощуючи розробку.

Децентралізоване управління даними. Кожен мікросервіс має власну базу даних або доступ до сегмента даних, що знижує ризик конфліктів і спрощує управління даними. Цей підхід також дозволяє вибирати різні технології баз даних для різних сервісів, враховуючи їх унікальні вимоги до продуктивності та зберігання даних.

Контейнеризація та оркестрація. Для розгортання і управління мікросервісами використовуються контейнери, такі як Docker, а оркестраційні платформи, як-от Kubernetes, допомагають автоматизувати процеси. Контейнеризація дозволяє ізолювати кожен мікросервіс у власному середовищі, що спрощує його управління, тестування та масштабування.

Безперервна інтеграція і доставка (CI/CD). Цей підхід дозволяє автоматизувати процеси тестування, розгортання і оновлення мікросервісів, забезпечуючи швидкість і якість розробки. CI/CD забезпечує зменшення ризиків помилок у процесі оновлення та спрощує додавання нових функцій.

Моніторинг і логування. Постійний моніторинг роботи мікросервісів та аналіз журналів дозволяє виявляти потенційні проблеми на ранніх етапах і забезпечує високу надійність системи. Використання таких інструментів, як Prometheus або Grafana, сприяє прозорості у відстеженні стану системи та швидкому виявленню аномалій.

Висока масштабованість. Архітектура мікросервісів дозволяє масштабувати окремі компоненти системи відповідно до потреб бізнесу, що оптимізує використання ресурсів. Наприклад, сервіси з високим навантаженням можуть бути масштабовані окремо від інших, що зменшує витрати.

Гнучкість у виборі технологій. Різні мікросервіси можуть бути розроблені на різних мовах програмування або використовувати різні фреймворки, що дозволяє вибирати оптимальні інструменти для кожної задачі.

ПЕРЕВАГИ ТА НЕДОЛІКИ ВИКОРИСТАННЯ МІКРОСЕРВІСІВ

Переваги

Масштабованість. Кожен мікросервіс може бути масштабований незалежно, залежно від навантаження. Наприклад, якщо сервіс обробки платежів починає отримувати значно більше запитів через акційну кампанію, лише цей компонент може бути масштабований, залишаючи інші частини системи незмінними. Такий підхід зменшує витрати на ресурси та підвищує ефективність роботи системи в цілому.

Гнучкість у розробці. Завдяки розподілу системи на невеликі сервіси, команди розробників можуть працювати паралельно, незалежно одна від одної. Це означає, що декілька команд можуть одночасно займатися розробкою нових функцій, тестуванням або усуненням помилок, що значно скорочує час розробки продукту та дозволяє швидше реагувати на бізнес-запити.

Стійкість до збоїв. Збої в одному сервісі не впливають на роботу інших, що підвищує загальну надійність системи. Наприклад, якщо сервіс обробки зображень виходить з ладу, це не зупинить роботу сервісів, які відповідають за авторизацію чи перегляд контенту.

Легкість оновлень. Мікросервіси дозволяють оновлювати окремі компоненти системи без необхідності перевипуску всього додатку. Наприклад, виправлення помилки в одному сервісі не потребує зупинки всіх інших компонентів, що зменшує ризики і спрощує процес оновлення.

Технологічна різноманітність. Кожен сервіс може бути реалізований з використанням різних мов програмування, фреймворків або баз даних. Наприклад, сервіс обробки великих даних може використовувати Hadoop, тоді як сервіс взаємодії з клієнтами може базуватися на більш легкому Node.js. Це дозволяє оптимізувати роботу кожного компонента.

Полегшення інтеграції. Завдяки стандартизованим інтерфейсам API мікросервіси легко інтегруються з іншими системами. Наприклад, інтеграція з платіжними шлюзами або сторонніми сервісами аналітики стає швидшою та ефективнішою.

Недоліки

Складність управління. Зі збільшенням кількості сервісів зростає складність їхнього адміністрування, моніторингу та забезпечення взаємодії між ними. Для великих систем необхідні спеціальні інструменти, такі як Kubernetes для оркестрації, і комплексні системи моніторингу на кшталт Prometheus або Grafana.

Витрати на інфраструктуру. Для роботи кожного сервісу потрібні окремі ресурси, що може збільшити витрати, особливо у невеликих проєктах. Наприклад, кожен контейнер може вимагати окремого серверного ресурсу, навіть якщо його навантаження незначне.

Проблеми з комунікацією. Інтенсивна взаємодія між сервісами через мережу може призводити до затримок. Наприклад, високочастотні запити між сервісами можуть створювати додаткове навантаження на мережеву інфраструктуру, що вимагає оптимізації протоколів комунікації.

Підвищена складність тестування. Перевірка роботи всіх мікросервісів у їхній взаємодії потребує складніших підходів до тестування. Наприклад, розробникам доводиться моделювати взаємодію сотень сервісів, щоб забезпечити стабільність роботи системи.

Безпека. Розподіленість системи вимагає додаткових заходів безпеки, таких як шифрування міжсервісних комунікацій, контроль доступу та моніторинг загроз. Наприклад, необхідно забезпечувати перевірку кожного запиту, що проходить через API Gateway.

Проблеми узгодженості даних. Розподіл баз даних між сервісами може ускладнити підтримку узгодженості даних. Наприклад, у випадках оновлення інформації одночасно в кількох сервісах можуть виникати конфлікти або затримки, які важко синхронізувати.

ВИКОРИСТАННЯ КОНТЕЙНЕРИЗАЦІЇ: DOCKER

Одним із найпоширеніших інструментів для контейнеризації є Docker (рис.6.1). Docker надає можливість створювати, розгортати та управляти контейнерами з мінімальними витратами ресурсів. Його ключова перевага полягає у зручності створення стандартних контейнерних образів, які включають всі необхідні бібліотеки, залежності та конфігурації для виконання певного мікросервісу. Це значно спрощує процес розгортання, роблячи його швидким і передбачуваним. Docker також забезпечує тісну інтеграцію з сучасними системами оркестрації, такими як Kubernetes, які дозволяють автоматизувати управління тисячами контейнерів одночасно. Це особливо важливо для великих систем, де потрібно підтримувати стабільну роботу під час пікових навантажень. Крім того, Docker надає можливості для версіонування образів, що спрощує процес оновлення програмного забезпечення та відкату до попередніх версій у разі виявлення помилок. Однією з унікальних особливостей Docker є його екосистема, яка включає Docker Hub — хмарний реєстр контейнерних образів. Це дозволяє розробникам швидко обмінюватися образами, використовувати готові рішення або створювати свої власні на основі доступних шаблонів. Завдяки цьому команди можуть значно прискорити розробку та розгортання нових мікросервісів, скорочуючи час на налаштування середовища.

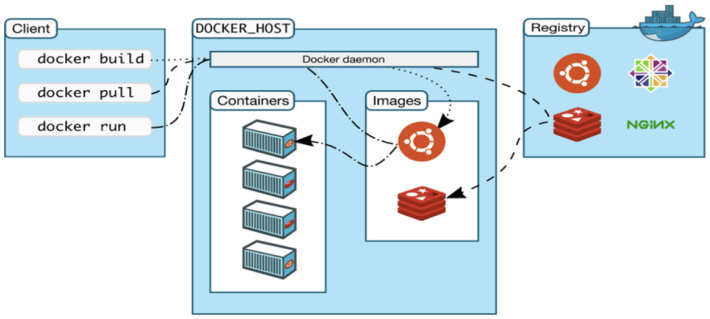

Рисунок 6.1 – Docker

ВИКОРИСТАННЯ КОНТЕЙНЕРИЗАЦІЇ: OCI

7

Ще однією важливою технологією є Open Container Initiative (OCI) (рис.7.1). Це стандарт для контейнерів, що встановлює чіткі правила для створення форматів контейнерних образів і середовищ виконання. Головною перевагою OCI є забезпечення сумісності між різними платформами контейнеризації, такими як Docker, Podman чи CRI-O, завдяки чому компанії можуть уникнути прив'язки до одного постачальника технологій. Стандарти OCI включають специфікації для контейнерних образів (Image Specification) та середовищ виконання (Runtime Specification), що дозволяє використовувати одні й ті самі контейнери на різних платформах без необхідності адаптації чи модифікацій. Наприклад, організація може розробляти додатки з використанням Docker, а розгортати їх на Kubernetes із застосуванням CRI-O без порушення функціональності. Завдяки стандартизації OCI підприємства отримують гнучкість у виборі інструментів, що відповідають їхнім потребам. Це також сприяє зменшенню витрат на адаптацію та інтеграцію нових технологій. Використання OCI дозволяє зберігати високу продуктивність систем, мінімізуючи ризики, пов'язані зі зміною постачальників або платформ контейнеризації. Таким чином, ця ініціатива забезпечує стабільність та передбачуваність процесів у межах мікросервісної архітектури.

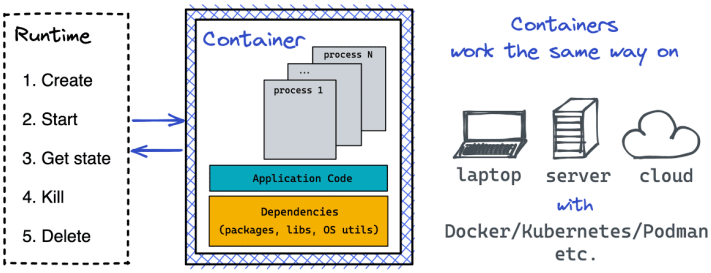

Рисунок 7.1 – Open Container Initiative (OCI)

ОРКЕСТРАЦІЯ МІКРОСЕРВІСІВ ЗА ДОПОМОГОЮ KUBERNETES

8

Оркестрація мікросервісів (рис.8.1) є фундаментальним етапом у розгортанні мікросервісної архітектури, оскільки вона забезпечує автоматизацію управління контейнерами, балансування навантаження, моніторинг і високу доступність системи. Kubernetes, як одна з провідних платформ оркестрації, надає розробникам і адміністраторам широкий спектр інструментів, які дозволяють не лише ефективно керувати контейнеризованими додатками, але й забезпечувати стійкість до збоїв та динамічне масштабування системи. Платформа стала стандартом для управління контейнерними середовищами завдяки її здатності інтегруватися з багатьма сучасними технологіями, такими як хмарні платформи, системи CI/CD і засоби моніторингу. Kubernetes не тільки забезпечує автоматизацію повторюваних завдань, але й дозволяє підвищити продуктивність команд розробників, зосереджуючи їх зусилля на створенні нових функцій, а не на вирішенні операційних питань. Kubernetes забезпечує автоматизоване розгортання, масштабування та відновлення роботи контейнерів, що є ключовою перевагою для забезпечення стійкості і безперервності роботи системи. Завдяки концепції "кластерів", платформа дозволяє ефективно розподіляти контейнери між вузлами, забезпечуючи оптимальне використання ресурсів серверів. Kubernetes постійно моніторить стан вузлів і контейнерів, визначаючи потенційні проблеми або збої у роботі компонентів. У разі виявлення таких збоїв, система автоматично перезапускає або переміщує контейнер на доступний вузол, мінімізуючи час простоїв і втрати даних.

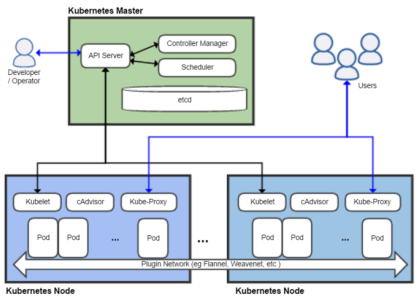

Рисунок 8.1 – Оркестрація мікросервісів Kubernetes

ПРИНЦИПИ РОЗГОРТАННЯ МІКРОСЕРВІСІВ ЗА ДОПОМОГОЮ KUBERNETES

9

Особливе значення має його здатність працювати як у локальних середовищах, так і в хмарних інфраструктурах, забезпечуючи узгодженість і повторюваність процесів незалежно від середовища розгортання. Це робить Kubernetes універсальним рішенням для сучасних програмних систем, що дозволяє компаніям швидко адаптуватися до змін ринку, підвищувати конкурентоспроможність та знижувати витрати на підтримку інфраструктури.

Основні принципи розгортання:

Контейнеризація: Усі мікросервіси повинні бути упаковані в контейнери, що забезпечує їхню ізоляцію та незалежність від основного середовища. Docker є найпоширенішим інструментом для створення контейнерів, які можна легко інтегрувати з Kubernetes.

Опис інфраструктури як коду: Kubernetes використовує YAML-файли для опису конфігурації мікросервісів, включаючи налаштування реплік, зберігання, мережевих з'єднань і політик доступу. Це забезпечує повторюваність і передбачуваність процесу розгортання.

Приклад YAML-файлу для розгортання мікросервісу:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-microservice
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example-app
  template:
    metadata:
      labels:
        app: example-app
    spec:
      containers:
      - name: example-container
        image: example/microservice:latest
        ports:
        - containerPort: 80
        env:
        - name: ENVIRONMENT
          value: production
```

АНАЛІЗ СУЧАСНИХ МЕТОДІВ ЗАБЕЗПЕЧЕННЯ БЕЗПЕКИ: OPENID CONNECT ТА JWT

10

Переваги OpenID Connect:

Забезпечує вищий рівень безпеки через централізований механізм аутентифікації, який дозволяє централізовано перевіряти особу користувача та управляти доступом до різних сервісів у межах розподіленої архітектури. Це забезпечує консистентність політик безпеки, можливість управління сесіями користувачів у реальному часі та захист від несанкціонованих спроб доступу.

Переваги JWT:

Простота використання та швидкість обробки токенів роблять JWT надзвичайно зручним інструментом для розробників, які працюють із розподіленими системами. Завдяки компактному формату, токени легко передаються через мережу, що мінімізує затримки та оптимізує продуктивність API. Крім того, їхня безстановість дозволяє уникати складного управління сесіями на сервері, що особливо важливо для систем із високою кількістю запитів. Використання стандартних алгоритмів шифрування, таких як HS256 чи RS256, забезпечує належний рівень безпеки при мінімальних витратах ресурсів.

Легко інтегрується в API, що дозволяє розробникам без додаткових витрат на складну конфігурацію інтегрувати JWT у існуючі системи. Завдяки цьому токени стають ефективним засобом автентифікації та авторизації для різноманітних додатків, таких як мобільні додатки, веб-додатки, а також системи з високою частотою API-запитів. Простота інтеграції також сприяє швидшому впровадженню нових функцій, що є критично важливим для швидкорозвиваючихся проектів.

Таблиця 10.1 – Порівняльна таблиця

Характеристика	OpenID Connect (OIDC)	JSON Web Tokens (JWT)
Призначення	Аутентифікація користувачів і делегування доступу	Перенесення аутентифікаційних даних
Стандарт/Протокол	Побудований на основі OAuth 2.0	Відкритий стандарт для створення токенів
Механізм роботи	Використовує Identity Provider для перевірки особи	Токени кодуються у форматі Base64
Підтримка авторизації	Є	Часткова (залежить від реалізації)
Зберігання стану	Підтримує стан через Identity Provider	Безстанова структура
Термін дії токенів	Динамічно керується	Фіксований (залежить від налаштувань)
Рівень безпеки	Високий (включає багаторівневі механізми)	Залежить від алгоритму шифрування (HS256, RS256)
Сценарії використання	Мобільні додатки, веб-додатки, корпоративні системи	API, взаємодія між сервісами

ОСНОВНІ МЕТОДИ ТЕСТУВАННЯ НАВАНТАЖЕННЯ

11

Стрес-тестування: Цей метод дозволяє оцінити, як мікросервіси працюють під умовами максимального навантаження, включаючи симуляцію надзвичайно високого трафіку або одночасних запитів від великої кількості користувачів.

Тестування продуктивності: Використовується для оцінки часу відгуку, швидкості обробки запитів та ефективності роботи сервісу за нормальних умов експлуатації.

Тестування навантаження: Використовується для симуляції реальних умов експлуатації, що дозволяє визначити, як система обробляє запити за різних рівнів активності.

Тестування на стійкість: Перевіряється, як система поводиться у разі часткової відмови одного чи кількох компонентів, включаючи аналіз її здатності відновлюватися та забезпечувати безперебійну роботу.

Інструменти для тестування навантаження

Apache JMeter: Один із найпоширеніших інструментів для моделювання навантаження, що забезпечує широкий спектр можливостей для тестування як окремих мікросервісів, так і комплексних систем.

Gatling: Високопродуктивний інструмент, орієнтований на тестування веб-додатків і API, який дозволяє моделювати реалістичні сценарії взаємодії користувачів із системою.

Locust: Потужний і гнучкий інструмент для створення сценаріїв навантаження, написаних на Python. Завдяки своїй простоті у використанні та масштабованості, Locust дозволяє імітувати реальні умови експлуатації систем, моделюючи поведінку тисяч одночасних користувачів.

Приклад сценарію для Locust

```
from locust import HttpUser, TaskSet, task
# Визначаємо набір завдань для моделювання поведінки користувачів
class UserBehavior(TaskSet):
    @task
    def index(self):
        # Запит до головної сторінки
        self.client.get("/")
    @task
    def about_page(self):
        # Запит до сторінки 'Про нас'
        self.client.get("/about")
# Основний клас, що визначає користувачів і час очікування між запитами
class WebsiteUser(HttpUser):
    tasks = [UserBehavior]
    # Встановлюємо мінімальний і максимальний час очікування між запитами
    min_wait = 5000
    max_wait = 15000
```

РЕКОМЕНДАЦІЇ ЩОДО ПОКРАЩЕННЯ БЕЗПЕКИ ТА ПРОДУКТИВНОСТІ

12

Покращення безпеки та продуктивності є критичними аспектами для забезпечення ефективності, стабільності та масштабованості роботи мікросервісної архітектури.

Нижче наведено детальні рекомендації, які сприяють підвищенню надійності, адаптивності та продуктивності системи у різноманітних умовах експлуатації:

Впровадження багатофакторної аутентифікації: Забезпечує додатковий рівень захисту, вимагаючи від користувачів підтвердження через кілька незалежних методів, таких як SMS-коди або біометричні дані.

Використання шифрування: Усі дані, що передаються між мікросервісами, мають бути зашифровані за допомогою сучасних алгоритмів, таких як TLS 1.3, для захисту від перехоплення чи несанкціонованого доступу.

Реалізація політик безпеки на рівні API: Використовуйте токени доступу, такі як OAuth 2.0 або JWT, для аутентифікації та авторизації запитів до API.

Моніторинг і логування: Інтегруйте системи моніторингу (Prometheus, Grafana) та централізованого логування (ELK Stack), що дозволяють виявляти загрози в реальному часі та швидко реагувати на інциденти.

Масштабування ресурсів: Використовуйте автоматичне масштабування в Kubernetes для адаптації системи до змінних умов навантаження, забезпечуючи стабільну роботу навіть під час пікових запитів.

Оптимізація часу відгуку: Регулярно проводьте тестування продуктивності, виявляючи та усуваючи вузькі місця в роботі мікросервісів.

Розподіл навантаження: Реалізуйте балансування навантаження за допомогою таких інструментів, як NGINX або HAProxy, що забезпечує рівномірний розподіл запитів між сервісами.

Впровадження політик доступу: Використовуйте рольові моделі (RBAC) для забезпечення доступу до критичних даних лише авторизованим користувачам.

Резервне копіювання та відновлення: Регулярно створюйте резервні копії даних та перевіряйте їхню цілісність, щоб забезпечити швидке відновлення у разі збоїв.

Використання контейнерної ізоляції: Кожен мікросервіс має працювати у власному контейнері з обмеженими правами доступу, що зменшує ризик поширення загроз у системі.

ВИСНОВКИ

Було успішно виконано дослідження, спрямоване на досягнення поставленої мети:

1. Контейнеризація та оркестрація: Технології, такі як Docker і Kubernetes, забезпечують ізоляцію сервісів, ефективне управління ресурсами та гнучке масштабування систем .

2. Методи автоматизації: Інструменти Ansible, Terraform, Helm та Jenkins оптимізують процеси розгортання та забезпечують повторюваність і передбачуваність операцій.

3. Безпека в мікросервісах: Використання протоколів OpenID Connect та JWT, шифрування даних і рольові моделі доступу гарантують захист інформації та запобігають несанкціонованому доступу.

4. Тестування навантаження: Методи оцінки продуктивності дозволяють виявити вузькі місця в роботі системи, оптимізувати її ресурси та забезпечити стабільну роботу навіть за умов пікових навантажень.