Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи магістра

галузь знань *12 Інформаційні технології*
(шифр і назва галузі знань)

спеціальність *125 Кібербезпека*
(код і назва спеціальності)

освітній ступень *магістр*

освітньо-наукова програма *Кібербезпека*
(назва освітньої програми)

на тему: « Розробка та розгортання системи моніторингу для забезпечення безпеки мікросервісної архітектури»

Виконавець: студентка ІІ курсу, групи КБм-21

(підпис) Гонтковська Єлизавета Андріївна
(Ім'я, ПРІЗВИЩЕ)

	Ім'я, ПРІЗВИЩЕ	**Підпис**
Науковий керівник	Сергій ТОЛЮПА	
Нормоконтроль	Юрій ЩЕБЛАНІН	

Київ 2023

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Сергій ТОЛЮПА
«24» жовтня 2022 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *125 Кібербезпека*
(код і назва спеціальності)

освітній ступень *магістр*

Здобувачки КБм-21 (група) Гонтковської Єлизавети Андріївни (прізвище ім'я по-батькові)

Тема кваліфікаційної роботи Розробка та розгортання системи моніторингу для забезпечення безпеки мікросервісної архітектури.

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Рішення засідання кафедри кібербезпеки та захисту інформації факультету інформаційних технологій протокол № 3 від 20.10.2022

2. МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень Процес створення захищеної мікросервісної архітектури.

Предмет досліджень Методи захисту мікросервісної архітектури.

Мета Розробка програмного забезпечення з моніторингу мікросервісної архітектури задля оцінки її поточного стану захищеності.

Вихідні дані для проведення роботи	Методи захисту мікросервісів, контейнеризація, безперервний моніторинг.

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна	Створення нового програмного продукту з моніторингу мікросервісів та вдосконалення наявних способів моніторингу додатків.
Практична цінність	Створене програмне забезпечення може бути використаним у рамках покращення захисту мікросервісної архітектури.

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Робота виконана у повному обсязі відповідно до теми.

5. ЕТАПИ ВИКОНАННЯ РОБОТИ

Найменування етапів робіт	Строки виконання робіт (початок-кінець)
Огляд інформаційних джерел, аналіз нормативно-правової бази	17.01.2023 – 25.01.2023
Досліджено принципи роботи мікросервісної архітектури	26.01.2023 – 3.02.2023
Визначено основні вразливості та структуровано методи захисту.	4.02.2023 – 15.03.2023
Створено програмне забезпечення	16.03.2023 – 28.04.2023

6. РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект	Зниження збитків через зменшення виникнення інцидентів безпеки.
Соціальний ефект	Покращення технологій забезпечення захисту програмних продуктів на мікросервісній архітектурі.

7. ДОДАТКОВІ ВИМОГИ

Завдання видав _____ Сергій ТОЛЮПА
(підпис) (Ім'я, ПРІЗВИЩЕ)

Завдання прийняв до виконання _____ Єлизавета Гонтковська
(підпис) (Ім'я, ПРІЗВИЩЕ)

Дата видачі завдання: 24.10.2022 р.
Термін подання кваліфікаційної роботи до ЕК 19.05.2023 р.

УДК. 004.432.16

РЕФЕРАТ

Пояснювальна записка: 82 с. основного тексту, 39 рис., 21 джерело та 2 додатки.

Об'єктом дослідження є процес створення захищеної мікросервісної архітектури.

Метою даної роботи є розробка програмного забезпечення з моніторингу мікросервісної архітектури задля оцінки її поточного стану захищеності.

Методи дослідження: аналіз літератури, структурування отриманих даних, системний підхід, оцінка досліджуваних методів захисту, структурний аналіз, методи порівняння.

Завдання дослідження:

1. Провести огляд мікросервісної архітектури, виділити особливості роботи.
2. Провести порівняння мікросервісної архітектури з монолітною. Проаналізувати переваги і недоліки мікросервісів;
3. Виявити основні вразливості додатку на мікросервісній архітектурі.
4. Проаналізувати та систематизувати методи захисту мікросервісів;
5. Виділити основні принципи роботи безперервного моніторингу як способу покращення захищеності та забезпечення спостережності процесів роботи програмного продукту;
6. Виділити найкращі показники для моніторингу.
7. Створити програмне забезпечення. Розробити веб додаток на базі мікросервісної архітектури. Розробити систему моніторингу веб додатку на мікросервісах.

Предметом дослідження є методи захисту мікросервісної архітектури.

Практична цінність: структурування методів захисту мікросервісів може бути використане для вирішення задач або проблем у сфері захисту інформаційних систем,

а створене програмне забезпечення може бути використаним у рамках покращення захисту мікросервісної архітектури та вдосконалене при необхідності.

Від успіху запровадження методів безпеки може залежати безперервність та якість роботи інформаційної системи, адже процеси захисту та мінімізації ризиків забезпечують стабільність роботи і підвищують ефективність компанії.

Від успіху побудови системи моніторингу залежить швидкість і ефективність виявлення можливих ризиків у інформаційній системі, наслідком чого є можливість усунення викликів безпеки до того як вони зможуть стати шкідливими і принести збитки.

Наукова новизна дослідження полягає у створенні нового програмного продукту з моніторингу мікросервісів та вдосконаленні наявних способів моніторингу додатків.

Апробація результатів роботи:

- Гонтковська Є. Методи захисту мікросервісної архітектури / Є. Гонтковська, С. Толюпа // “Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS). 27 - 28 жовтня 2022, Київ, Україна.
- Гонтковська Є. Безпека API та мікросервісів / Є. Гонтковська, С. Толюпа // “Новітні технологічні тенденції інтелектуальної індустрії та Інтернету речей” (TTSIIT). 24-25 січня 2023 р, Київ, Україна.
- Гонтковська Є., Мостовенко А., Толюпа С. / Problematic aspects of solving issues of cyber influence on critical infrastructure objects. Scientific and Practical Cyber Security Journal (SPCSJ) № 3 (02) September 2023. (Грузія).
- Гонтковська Є. Вплив моніторингу на захист мікросервісної архітектури / Є. Гонтковська, С. Толюпа // “Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS). 27 квітня 2023, Київ, Україна.

Ключові слова: мікросервісна архітектура, безпека, моніторинг, програмне забезпечення.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API	–	Application Programming Interface
DevSecOps	–	Development, Security, Operations
POLP	–	Principle of Least Privilege
SSL	–	Secure Sockets Layer
SBOM	–	Software Bill of Materials
Kubernetes	–	Додаток з оркестрування контейнерів
IT	–	Information Technology
SLA	–	Service Level Agreement
APM	–	Application performance monitoring
SIEM	–	Security information and event management
Ping	–	Утиліта для перевірки з’єднань в мережах
ЦП	–	Центральний процесор
NIST	–	Національний інститут стандартів і технологій
APM	–	Application performance monitoring
SRMP	–	Sustainable Resource Management Plan
ПЗ	–	Програмне забезпечення
ISM	–	International Safety Management

ЗМІСТ

РЕФЕРАТ 4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 6
ЗМІСТ 7
ВСТУП 9
РОЗДІЛ 1 МІКРОСЕРВІСНА АРХІТЕКТУРА 10
1.1 Загальний огляд мікросервісної архітектури 10
1.1.1 Принципи роботи мікросервісів 10
1.2 Порівняння мікросервісної та монолітної архітектур 12
1.2.1 Монолітна архітектура 13
1.2.2 Порівняння архітектур 14
1.3 Плюси і мінуси використання мікросервісів 15
1.3.1 Сильні сторони 15
1.3.2 Слабкі сторони 16
1.3.3 Переваги вибору мікросервісної архітектури 17
Висновки за розділом 1 19
РОЗДІЛ 2 МЕТОДИ ЗАХИСТУ МІКРОСЕРВІСІВ 20
2.1 Вразливості мікросервісів 20
2.1.1 Основні вразливості додатків побудованих на мікросервісах 20
2.1.2 Особливості розробки та впровадження мікросервісної архітектури 21
2.2 Проблематика захисту мікросервісної архітектури 24
2.2.1 Покриття великої поверхні ризиків 24
2.2.2 Ізоляція сервісів 24
2.2.3 Логування 25
2.2.4 Взаємодія команд розробки 25
2.2.5 Тестування сервісів 26
2.2.6 Забезпечення відмовостійкості 26
2.2.7 Безпека API 27
2.3 Сучасні методи захисту 27
2.3.1 Контроль процесів розробки 28

2.3.2 Контроль доступу 28
2.3.3 Захист точок входу 29
2.3.4 Використання SBOM 30
2.3.5 Забезпечення спостережності служб додатку 31
2.3.6 Ізоляція 32
2.3.7 Автоматизація та тестування 32
Висновки за розділом 2 33
РОЗДІЛ 3 ВИКОРИСТАННЯ МОНІТОРИНГУ ДЛЯ ОПТИМАЛЬНОГО ЗАХИСТУ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ 34
3.1 Огляд роботи систем моніторингу 34
3.1.1 Моніторинг мікросервісів 36
3.1.2 Найкращі метрики для моніторингу 40
3.2 Вплив моніторингу на безпеку мікросервісів 41
Висновки за розділом 3 53
РОЗДІЛ 4 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 54
4.1 Проектування програмного забезпечення 54
4.2 Розробка сайту побудованого на мікросервісній архітектурі 57
4.2.1 Створення серверної частини додатку на мікросервісах 58
4.2.2 Створення візуальної частини додатку на мікросервісах 67
4.2.3 Перевірка працездатності веб продукту 68
4.3 Розробка системи моніторингу мікросервісів 70
4.3.1 Створення агенту з моніторингу. 71
4.3.2 Створення додатку з моніторингу. 72
4.3.3 Перевірка працездатності системи моніторингу мікросервісів 75
4.3.4 Перспективи розвитку додатку 77
Висновки за розділом 4 78
ВИСНОВКИ 79
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 81

ВСТУП

У сучасному світі розвиток програмних продуктів є дуже популярним, як і поширеними є загрози та ризики, які негативно впливають на них. Архітектура побудови програмних продуктів дуже сильно відображається на роботі додатків, тому забезпечення її захисту є необхідним.

Однією з найпопулярніших архітектур побудови є мікросервісна архітектура, що базується на невеликих контейнерних службах додатків, які виконують завдання для вирішення тих чи інших задач у системі.

Багато програмних продуктів можуть бути побудованими без урахування специфіки їх архітектури та без урахування спеціальних методик захисту. Особливо критично це може проявитися у роботі з мікросервісами які мають багато вразливостей які необхідно брати до уваги кожній компанії яка вирішила розробити додаток на мікросервісній архітектурі.

Для усунення проблем захисту часто використовуються різноманітні методи захисту мікросервісів, які допомагають покращити безпеку всієї архітектури та у наслідок підвищити надійність програмного забезпечення. Хоча існує дуже багато систем захисту, не всі вони можуть осягнути різноманіття ризиків, що існують у сучасному світі.

Враховуючи постійний розвиток загроз які можуть значно впливати на роботу інформаційних ресурсів, необхідним є розвиток систем захисту та вдосконалення наявних методів захисту архітектури мікросервісів для покращення безпеки додатків та забезпечення безпепервності їх роботи.

РОЗДІЛ 1
МІКРОСЕРВІСНА АРХІТЕКТУРА

1.1 Загальний огляд мікросервісної архітектури

Архітектура мікросервісів відноситься до архітектурного стилю для розробки програм, у даній архітектурі додатки розроблюються як набір служб, що забезпечують основу для незалежної розробки, розгортання та підтримки служб архітектури мікросервісів.

Мікросервіси — це архітектурний стиль, який структурує програму як набір сервісів. Мікросервіси [1]:

- розгортаються самостійно;
- слабко пов'язані;
- організовані навколо можливостей бізнесу;
- належать невеликим командам або сукупності команд;
- мають високу ремонтопридатність і можливість проведення тестування.

Мікросервіси дозволяють розділити велику програму на менші незалежні частини, кожна з яких має свою сферу відповідальності. Щоб обслуговувати один запит користувача, програма на основі мікросервісів може звертатися до багатьох внутрішніх мікросервісів, щоб створити свою відповідь. Таким чином дана архітектура забезпечує швидку, часту та надійну доставку великих складних програм. Це також дозволяє організації розвивати свій стек технологій.

1.1.1 Принципи роботи мікросервісів

Архітектура мікросервісів (рис. 1.1) працює за допомогою зосередження на класифікації великих, громіздких програм. Кожна мікрослужба стосується певного

аспекту та функції програми, наприклад журналювання, пошук даних тощо. Разом ці мікросервіси утворюють єдину програму.

На рівні користувачів клієнт може використати інтерфейс для створення запиту, і один або декілька мікросервісів у залежності від функцій які вони виконують розпочнуть виконувати свою роботу через шлюз API для виконання запитаного завдання.

Рисунок 1.1 – Схематичний приклад роботи мікросервісної архітектури

Серед прикладів використання мікросервісної архітектури можна виділити [2]:

- міграцію сайту;
- медіаконтент;
- використання контейнерів;
- обробку даних.

За допомогою мікросервісів складний веб-сайт, розміщений на монолітній платформі, можна перенести на хмарну та контейнерну платформу мікросервісів.

Організацію медіаконтенту зробити доволі просто використовуючи архітектуру мікросервісів. Зображення та відеоресурси можна зберігати в масштабованій системі зберігання об’єктів і обслуговувати безпосередньо в Інтернеті чи на мобільному пристрої.

Використання контейнерів дозволяє зосередитися на розробці сервісів, не турбуючись про залежності. Сучасні хмарні додатки зазвичай будуються як поєднання мікросервісів з контейнерами.

Для покращення обробки даних платформа мікросервісів може розширити хмарну підтримку для існуючих модульних служб обробки даних.

Системи мікросервісів є дуже поширеними у використанні, адже вони не тільки полегшують незалежне створення, роботу, масштабування але і покращують процеси розгортання кожної компонентної служби.

За допомогою мікросервісів у команд розробки немає спільного використання кодів або функцій з іншими службами, тобто вони працюють чітко над своєю частиною коду що полегшує розробку та тестування.

Чітко визначені API поширюють зв’язок між різними компонентами програми та поєднують мікросервісну архітектуру в єдине ціле пов'язуючи між собою функції різноманітніх мікросервісів за допомогою API запитів [3].

Залежно від конкретної проблеми кожна послуга в системі адаптована до унікального набору навичок. Це дає розробникам багато варіантів вирішення потенційних викликів безпеки та у наслідку мінімізує ризики виникнення інцидентів інформаційної безпеки.

1.2 Порівняння мікросервісної та монолітної архітектур

Повноцінний вибір архітектури програмного застосунку неможливий без порівняння.

Для того щоб зрозуміти можливості архітектури, оцінити її переваги і недоліки необхідно звернути увагу на існуючі альтернативи.

1.2.1 Монолітна архітектура

Монолітна архітектура (рис. 1.2) - це традиційна модель програмного забезпечення, яка є єдиним модулем, що працює автономно і незалежно від інших додатків.

Рисунок 1.2 – Схематичний приклад роботи монолітної архітектури

У монолітних архітектурах усі процеси тісно пов’язані між собою та виконуються як єдиний сервіс. Це означає, що якщо один процес програми відчуває сплеск попиту, необхідно масштабувати всю архітектуру.

Збільшення або вдосконалення функцій монолітної програми стає складнішим зі зростанням бази коду. Ця складність обмежує експерименти та ускладнює реалізацію нових ідей. Монолітні архітектури підвищують ризик доступності додатків, оскільки багато залежних і тісно пов’язаних процесів посилюють вплив відмови одного процесу.

1.2.2 Порівняння архітектур

За допомогою архітектури мікросервісів основна проблема монолітної архітектури вирішується, оскільки програма будується на незалежних компонентах, які запускають кожен процес програми як службу. Ці служби спілкуються через чітко визначений інтерфейс із використанням легких API запитів. Сервіси створені для бізнес-можливостей, і кожен сервіс виконує певну функцію. Оскільки вони працюють незалежно, кожну службу можна оновлювати, розгортати та масштабувати відповідно до попиту на певні функції програми.

Порівнюючи сильні та слабкі сторони архітектур можна сказати що кожна із них може бути побудована і використана від потреб тих чи інших компаній, тому кожен розробник повинен розуміти з чим буде працювати та обрати свій найкращий шлях розробки. Розглянувши мікросервісну та монолітну архітектури можна виділити певні особливості які їх відрізняють і обрати для себе найзручнішу у використанні.

З одного боку монолітна архітектура є автономною та незалежною від інших програм, забезпечує легке розгортання та розробку, спрощене тестування, легке налагодження та оптимізовану продуктивність. Однак монолітна архітектура має свої недоліки, зокрема повільнішу швидкість розробки, проблеми з масштабованістю та той факт, що окремі помилки можуть вплинути на доступність усієї програми.

З іншого боку, архітектура мікросервісів є більш спритним, гнучким і масштабованим рішенням. Основою мікросервісів є те, що вони забезпечують безперервне розгортання та швидші цикли випуску. Мікросервісна архітектура також зручна у обслуговуванні та перевірці, а також забезпечує більшу гнучкість у

технологічних параметрах. Однак архітектура мікросервісів також може призвести до розповсюдження розробки, збільшення витрат, а також до проблем з налагодженням через великі обсяги даних журналу.

1.3 Плюси і мінуси використання мікросервісів

Перед тим як будувати мікросервісну архітектуру у програмному застосунку, розробники додатку повинні мати уявлення про її переваги і недоліки.

Як і монолітна, мікросервісна архітектура має ряд сильних і слабких сторін, які мають бути розглянутими перед початком побудови архітектури майбутньої програми. Це означає, що під час процесу розробки необхідно враховувати специфіку роботи мікросервісів, щоб програмний застосунок у результаті був якісним і надійно захищеним.

1.3.1 Сильні сторони

Впровадження мікросервісної архітектури має багато позитивних сторін у розробці та переважно використанні. Робота з гнучкою та ізольованою архітектурою може стати дуже корисною для великих програмних застосунків, у тому разі, коли використання монолітної архітектури вже не є доцільним через громіздкий код та складність масштабування.

Враховуючи аспекти роботи з мікросервісної архітектури можна виділити такі переваги використання мікросервісів [4]:

- незалежні служби мають велику гнучкість;
- свобода вибору розробниками програмного та апаратного забезпечення;
- легкість дотримання командами принципів проектування SOLID (наприклад, принципу єдиної відповідальності);
- підтримка мікросервісами розподіленої розробки;

- легкість використання мікросервісів разом із сучасними технологіями (наприклад, контейнеризацією та хмарними обчисленнями);
- швидкий час виходу на ринок;
- легкий процес масштабування;
- швидкі цикли розробки;
- легкість налаштування та підтримки ізольованих служб.

Популярнсть використання мікросервісів зумовлена її великим спектром позитивних сторін які сприяють розвитку бізнесу.

З одного боку належним чином налаштовані та захищені мікросервісні архітектури забезпечують швидку та безперебійну роботу додатків. З іншого боку треба ніколи не забувати про внутрішні складнощі які можуть бути пов’язані з процесами розробки та процесами забезпечення безпеки програми. Враховуючи це, розробникам працюючим з мікросервісною архітектурою необхідно приділяти увагу не тільки перевагам, а і недолікам, щоб мати змогу без проблем справлятися з труднощами, які можуть виникнути під час роботи з мікросервісами.

1.3.2 Слабкі сторони

Не зважаючи на великий перелік переваг, мікросервісна архітектура має ряд слабкостей, на які треба звернути увагу кожному розробнику додатків на мікросервісах.

Серед недоліків мікросервісів можна виділити[4]:

- додаткові складності, до яких менші команди не завжди готові;
- обмін повідомленнями між службами має накладні витрати на продуктивність (наприклад, затримка мережі, обробка повідомлень тощо);
- мікросервіси важко підтримувати без автоматизації та передових методів управління проектами;
- прийняття методології має круту криву навчання;

- додаткові проблеми безпеки (наприклад, безпека транзакцій потребує особливої уваги).

Розглянувши недоліки мікросервісної архітектури можна сказати, що розробникам треба приділяти багато зусиль комунікації між командами під час роботи з кодом, щоб уникнути додаткових вразливостей на програмному рівні. Також для того, щоб програмний продукт був якісним, командам необхідно працювати над процесами оптимізації та автоматизації роботи кожного сервісу. Команди інформаційної безпеки у свою чергу повинні бути пильними із захистом компонентів програмного продукту, захистом даних та з захистом міжсервісних зв'язків для того щоб зменшити ризики виникнення інцидентів з інформаційної безпеки. Також використання безперервного моніторингу додатку надає можливість компанії уникнути багатьох проблем з додатком та значно підвищити його шанси на забезпечення найкращої продуктивності роботи.

1.3.3 Переваги вибору мікросервісної архітектури

Мікросервіси сприяють організації невеликих незалежних команд, які беруть на себе власність за їхні послуги. Команди діють у невеликому та добре зрозумілому контексті та мають змогу працювати незалежно та швидко, що значно скорочує час циклу розробки та підвищує загальну продуктивність.

Мікрсервіси дозволяють незалежно масштабувати кожну службу для задоволення попиту на функцію програми, яку вона підтримує. Це дає змогу командам правильно оцінювати потреби в інфраструктурі, точно вимірювати вартість функції та підтримувати доступність, якщо попит на послугу різко зросте.

Мікросервіси забезпечують постійну інтеграцію та безперервну доставку, спрощуючи випробування нових ідей і відкат, якщо щось не працює. Низька вартість відмови дозволяє експериментувати, полегшує оновлення коду та пришвидшує час виходу на ринок нових функцій.

Архітектури мікросервісів не дотримуються підходу монолітної архітектури в якій переважно обирається основна мова програмування, шаблон наробок та певні

інструменти для всього коду. Команди мають свободу вибирати свій найкращий спосіб для вирішення своїх конкретних проблем. Як наслідок, команди, що створюють мікросервіси, можуть вибрати найзручніший набір інструментів для кожної роботи.

Розподіл програмного забезпечення на невеликі, чітко визначені модулі дозволяє командам використовувати функції для багатьох цілей. Службу, написану для певної функції, можна використовувати як будівельний блок для іншої функції. Це дозволяє програмі самозавантажуватися, оскільки розробники можуть створювати нові можливості без написання коду з нуля.

Незалежність сервісу підвищує стійкість програми до збоїв. У монолітній архітектурі, якщо один компонент виходить з ладу, це може призвести до збою всієї програми. За допомогою мікросервісів програми справляються з повним збоєм служби, погіршуючи функціональність, а не виводячи з ладу всю програму [5].

Не зважаючи на те, що мікросервісна архітектура і має дуже багато переваг все одно розробниками необхідно бути дуже пильними при побудові архітектури. Іншими словами, всі бонуси, які можна отримати від простоти мікросервісів з єдиною відповідальністю, можна втратити на складності всієї програми, накладних витратах на продуктивність і додаткових потребах безпеки системи обміну повідомленнями, яка з’єднує служби.

Хоча архітектура мікросервісів і дозволяє кожній команді вибирати, яку мову програмування та платформу використовувати, члени команди також повинні намагатися краще співпрацювати, щоб керувати всім життєвим циклом свого мікросервісу і для того щоб забезпечити найкращу працездатність архітектури.

Також дуже важливим у якості та безперервності роботи всієї мікросервісної архітектури є питання забезпечення належного рівня захисту системи. Найчастіше компанії для цього використовують поєднання технічних та програмних методів захисту таких як моніторинг мікросервісної архітектури, розмежування доступу, використання шлюзу API, використання зображення змісту програм та інші способи захисту. Така сукупність способів дбайливого нагляду за чистотою коду, оптимізацією, моніторингу та захисту програмного застосунку покращує

безперервність і якість роботи. Забезпечення надійного захисту додатку і безперебійної роботи сервісів у наслідок гарантує покращення репутації компанії у клієнтів та буде сприяти розвитку бізнесу.

Висновки за розділом 1

1. Проведено загальний огляд мікросервісної архітектури. Пояснено поняття мікросервіси.
2. Проведено порівняння мікросервісної та монолітної архітектур. Виділено їх особливості.
3. Проведено аналіз сильних сторін мікросервісної архітектури, наведено перелік недоліків використання мікросервісів.
4. Розглянуто наслідки використання саме мікросервісної архітектури під час побудови додатків. Наведено позитивні наслідки для бізнесу компанії після успішного впровадження мікросервісів.

РОЗДІЛ 2
МЕТОДИ ЗАХИСТУ МІКРОСЕРВІСІВ

2.1 Вразливості мікросервісів

Архітектура мікросервісів є дуже популярною. Побудова додатків які базуються на мікросервісах є чудовим і зручним способом для створення програмного забезпечення сучасними розробниками. Дана архітектура має багато якісних сторін, які розкриваються при її впровадженні і використанні, але не зважаючи на всі здобутки мікросервісів, вони також можуть мати власну Ахіллесову п’яту.

При використанні мікросервісів треба завжди приділяти увагу вразливостям, які вони можуть принести у створений ІТ продукт. Щоб мати змогу подолати незручності забезпечення захисту такого додатку необхідно розумітися на основних проблемах безпеки мікросервісів та мати уяву про особливості роботи з мікросервісною архітектурою на всіх етапах життя програмного забезпечення.

2.1.1 Основні вразливості додатків побудованих на мікросервісах

Вразливість програми визначається як слабка сторона, яка може бути вадою програми, реалізацією помилки або неправильною конфігурацією програми, що може дозволити зловмиснику завдати шкоди зацікавленим сторонам продукту.

Серед найпоширеніших вразливостей додатків побудованих на мікросервісній архітектурі можна виділити [6]:

- порушений контроль доступу;
- криптографічні збої;
- загрозу ін'єкційних атак;
- небезпечний дизайн;

- неправильну конфігурацію безпеки;
- вразливі та застарілі компоненти;
- помилки ідентифікації та автентифікації;
- порушення цілісності програмного забезпечення та даних;
- помилки реєстрації та моніторингу безпеки;
- підробку запитів на стороні сервера.

Для того щоб мінімізувати ризики втілення загроз необхідно враховувати усі особливості мікросервісної архітектури та перекрити всі вразливості додатку за допомогою безперервного та надійного захисту даних, контролю користувачів, контролю середи розробки, моніторингу та інших методів захисту мікросервісів які у своєму поєднанні допоможуть нівелювати вразливості архітектури.

2.1.2 Особливості розробки та впровадження мікросервісної архітектури

Мікросервісна архітектура може мати дуже багато ризиків, які зумовлені конкурентним і швидким характером розробки програмного забезпечення, це може спонукати команди приймати рішення, які в кінцевому підсумку іноді спричиняють більше проблем, ніж виправляють.

Якщо розробники не будуть в змозі зрозуміти як подолати виклики та зменшити ризики, пов’язані з мікросервісами, вони можуть стикнутися з великою кількістю складностей, що призведе до різкої зупинки прогресу.

Деякі з проблем, з якими стикаються розробники з мікросервісами, включають [7]:

- операційну складність;
- складність масштабування;
- складність забезпечення відмовостійкості;
- обмежений контроль середовища;
- загрози безпеці;
- уразливості відкритого вихідного коду;

- ліцензійні зобов'язання.

Операційна складність зумовлена якістю взаємодій та операцій мікросервісів. Оскільки кожна служба розгортається незалежно, координація операцій на шляху служб часто буває ускладненою. Крім того, під час виконання запиту користувача системою може бути задіяно кілька служб, тому аналіз першопричини може бути неможливим за допомогою традиційних засобів моніторингу.

Масштабування та оптимізація може також стати викликом для розробників, оскільки розширення або оптимізація незалежно розміщених і розгорнутих служб вимагає складної координації різних компонентів на окремих серверах. Хоча масштабованість є однією з часто згадуваних переваг архітектури мікросервісів, досягти її може бути складно.

Забезпечення відмовостійкості може бути складною і цікавою задачею адже вона має бути оптимізована для кожного мікросервісу. Збій одного компонента може вплинути на всю систему, проте чим краще відмовостійкість забезпечена, тим менше шансів на те, що при відмові тої чи іншої служби постраждають інші.

Забезпечення контролю середовища є необхідним для захисту будь-якої архітектури побудови програмних застосунків. В архітектурі мікросервісів існує мінімум централізованого керування, що ускладнює контроль середовища. У той час як один набір процедур застосовується до монолітного середовища, мікросервісне середовище потребує різних наборів процедур контролю для кожного середовища. Тестування окремих компонентів архітектури мікросервісу разом із їх взаємозалежностями посилює складність контролю середовища. Оскільки тестування має бути включено на всіх етапах життєвого циклу розробки програмного забезпечення, чим раніше буде введений контроль середовищ тим краще буде вплив на архітектуру мікросервісів.

Мікросервіси розгортаються в різних хмарних середовищах і спілкуються один з одним через різні рівні інфраструктури, що призводить до зменшення контролю та збільшення неясності компонентів. Структурна особливість мікросервісів може не тільки збільшити вразливості системи безпеки, але й зробити складнішим тестування на наявність вразливостей і безпеку мережі. Якщо рівні інфраструктури, які сприяють

спільному використанню ресурсів між службами, погано налаштовані, результатом буде неоптимальна програма з повільним часом відгуку.

Якщо захист мікросервісів не буде досконалим, це може збільшити шанси на появи загроз, які зможуть нашкодити програмному продукту. Отже при роботі з мікросервісною архітектурою треба приділяти багато уваги контролю і захисту інформаційних ресурсів програмного забезпечення.

Розподілена структура ускладнює захист даних, оскільки технічно важко контролювати доступ і адмініструвати захищену авторизацію для окремих служб. Конфіденційність, доступність і цілісність даних також важче підтримувати за допомогою розподіленої системи.

Дані в мікросервісах постійно переміщуються та використовуються, інформаційні ресурси зберігаються в різних місцях для різних цілей, що відкриває більше точок входу для зловмисників і породжує більше ризиків.

Програмне забезпечення з відкритим кодом часто є компонентом мікросервісів, у наслідок чого мікросервісна архітектура також може мати уразливості, пов’язані зі стороннім і відкритим кодом. Найважливішими проблемами, які слід розглянути з відкритим вихідним кодом, є вразливі місця в безпеці та ліцензійні зобов’язання.

Програмне забезпечення з відкритим вихідним кодом часто має слабкі місця, які є загальновідомими, як і для фахівців безпеки так і для зловмисників. Ці уразливості дуже сильно впливають на забезпечення захисту програмного продукту. Хоча латки зазвичай швидко випускаються для усунення цих ризиків безпеки, але на практиці майже 80% інформаційних бібліотек ніколи не оновлюються. Як наслідок, застаріле програмне забезпечення з відкритим кодом може спричинити серйозні проблеми з продуктивністю та викликати загрози безпеки.

Найпоширенішими проблемами ліцензійних зобов’язань пов’язаними з відкритим кодом є порушення та обмеження. Існує понад 200 різних типів ліцензій на програмне забезпечення з відкритим кодом, які часто конфліктують одна з одною. Розробники, які використовують відкритий код у своїх збірках, повинні дотримуватися цих ліцензій.

2.2 Проблематика захисту мікросервісної архітектури

Розробники програмних засносунків побудованих на мікросервісній архітектурі часто стикаються з проблемами безпеки, які необхідно вирішувати. Серед таких викликів автор даної роботи виділяє:

- покриття великої поверхні ризиків;
- ізоляцію сервісів;
- використання журналів подій або логування не тільки на рівні одного сервісу, а з прорахунком співвідносин подій між сервісами;
- взаємодію команд розробки;
- належне тестування сервісів;
- забезпечення відмовостійкості;
- безпеку API.

2.2.1 Покриття великої поверхні ризиків

В мікросервісній архітектурі використовується зв'язок великої кількості служб, що працюють на різних портах. Використяння великої кількості портів та API, особливо відкритих, значно збільшує площу атаки для зловмисників.

У програмі на основі мікросервісів кожен окремий сервіс спілкується з іншими через чітко визначені API, що збільшує поверхню атаки та робить API вразливими до загроз безпеки.

Щоб мінімізувати ризики виникнення загрози безпеці, вкрай важливо, щоб усі мікросервіси були належним чином захищені.

2.2.2 Ізоляція сервісів

Ізоляція є однією з основних функцій програми на основі мікросервісів.

У типовій програмі на основі мікросервісів розробники повинні мати можливість створювати, тестувати, розширювати, розгортати та підтримувати програму ізольовано, тобто жодна з цих дій не повинна впливати на функціонування будь-якого іншого сервісу в програмі.

Для покращення захисту ізоляцію найкраще реалізовувати на рівні бази даних. Тобто кожен мікросервіс повинен мати свою копію даних і не повинен мати доступ до даних, що стосуються інших мікросервісів у програмі.

Впроваджуючи ізоляцію на всіх рівнях, можна зробити свою програму на основі мікросервісів більш безпечною.

2.2.3 Логування

Логування монолітних додатків традиційним способом використовується вже давно. Однак той самий старий традиційний спосіб журналювання неефективний у програмі на основі мікросервісів. Це пов’язано з тим, що в додатку на основі мікросервісів, як правило використовуються дані про розподілені автономні служби без збереження стану.

Щоб журналювання було ефективним, програма повинна мати можливість агрегувати журнали та співвідносити події між кількома службами та платформами.

2.2.4 Взаємодія команд розробки

Щоб архітектура мікросервісів була успішною в організації, команди розробки та операцій повинні тісно співпрацювати – вони повинні мати можливість взаємодіяти одна з одною за потреби.

Для успішної роботи мікросервісної архітектури, незважаючи на те, що команди розробки та операцій мають об’єднатися, вони також мусять добре розуміти роботу внутрішніх задіяних процесів та знати способи пом’якшення ризиків інформаційної безпеки.

2.2.5 Тестування сервісів

Незважаючи на переваги створення додатків шляхом розробки, розгортання та керування службами, незалежними одна від одної, уразливостей у системі безпеки багато, оскільки програми випускаються без ретельного тестування. У наслідок, покращена гнучкість мікросервісної архітектури створюється за рахунок безпеки – оскільки у програмі наявні часті збірки, додатки можуть не пройти належне тестування перед випуском.

2.2.6 Забезпечення відмовостійкості

Під час роботи з мікросервісами необхідною частиною розробки є забезпечення відмовостійкості кожної ланки програми. Відмовостійкість - це здатність програми продовжувати роботу в разі відмови одного або кількох компонентів. Така функція досягається за допомогою реалізації резервного механізму, наприкляд схеми автоматичного вимикача, що часто використовується в таких програмах.

Впровадження відмовостійкості в додатку на основі мікросервісів є складним завданням – це набагато складніше та важче реалізувати, ніж у монолітному додатку. Завдяки збільшенню кількості служб з’являється більше запитів, а коли кількість служб, що спілкуються через мережу, збільшується, складність також зростає. Саме тому сервіси мають бути відмовостійкими, тобто вони повинні подолати проблеми зі збоями сервісів, які можуть виникнути.

Для того щоб мінімізувати виклики безпеки, розробникам необхідно використовувати стратегію поглибленого захисту, щоб захистити мікросервіси.

Підхід глибинного захисту передбачає додавання багатьох рівнів контролю безпеки до програми. Рівні захисту використовуються як і для захисту критично важливих служб так і для інших служб, а методи захисту обираються відповідно до рівня небезпеки реалізації виклику надійності роботи системи. Це означає, що якщо потенційний зловмисник зможе атакувати один рівень захисту в додатку, інші рівні додатка будуть складними для зламу.

2.2.7 Безпека API

У типовій програмі на основі мікросервісів споживачі послуг не можуть спілкуватися з мікросервісами безпосередньо. Натомість використовується шлюз API, який забезпечує єдину точку входу для спрямування трафіку до різних мікросервісів. Таким чином, клієнти не мають прямого доступу до послуг і не можуть використовувати їх. Якщо шлюз API розташований за брандмауером, то необхідно додати рівень захисту навколо поверхні атаки.

Шлюзи API зазвичай захищені за допомогою автентифікації на основі маркерів, але додатково необхідно приділяти увагу тому щоб мікросервіси були захищеними, наприклад розташуванням шлюзу API за брандмауером.

Доступ до API має бути захищеним – ніхто, крім авторизованих користувачів, не повинен мати до них доступ. Також слід підтримувати сертифікати безпеки, а дані, що передаються, мають бути зашифровані та захищені.

Знаючи про проблеми безпеки мікросервісів, розробникам легше їх подолати. За допомогою впровадження глибинного захисту архітектури, використання шлюзів API та захисту інформації можна дуже сильно покращити стан захищенності програмного продукту, що у наслідок буде впливати на безперервність та якість його роботи.

2.3 Сучасні методи захисту

Проаналізувавши основні проблеми захисту мікросервісної архітектури можна сказати, що без досконалого захисту, додаток може мати велику кількість дір безпеки, які мають бути усуненими. Саме тому знаходження найкращих методів захисту мікросервісів є необхідним.

У сучасному світі існує дуже багато методів захисту мікросервісної архітектури. Серед найефективніших з них можна виділити поєднання: захисту

даних, контролю доступу, захисту середовища розробки та програмного коду, ізоляції, захисту точок комунікації сервісів, та моніторингу.

2.3.1 Контроль процесів розробки

Мікросервіси вимагають великої співпраці між командами розробки додатків протягом всього життєвого циклу додатків. Команди розробників повинні добре розуміти всі задіяні процеси для того, щоб вони могли мінімізувати ризики безпеки.

Оскільки служби створюються окремо різними командами, незалежними одна від одної, відокремлені процеси розробки сприяють швидкому розгортанню сервісів. Однак це розділення та швидкість часто призводять до відсутності тестування безпеки. Нехтування безпекою з ціллю швидкого виходу на ринок може призвести до наявності вразливостей у додатку.

Щоб забезпечити безпеку програм, команди DevSecOps повинні постійно тестувати написаний код та перевіряти процеси доставки/розгортання сервісів на всіх етапах розробки.

Командою розробників DevSecOps мають бути інтегровані заходи безпеки та зниження ризиків у свою практику на кожному етапі розробки з відкритим кодом.

2.3.2 Контроль доступу

Розробники також можуть вирішувати проблеми безпеки додатків, дотримуючись найкращих практик проектування безпеки в архітектурі мікросервісів, зокрема за допомогою прийняття принципів найменших привілеїв (POLP).

Багато мікросервісів використовують технологію контейнерів на основі зображень, які можуть містити вразливості. Команди повинні регулярно сканувати зображення, щоб переконатися, що вони не містять вразливостей.

Крім того, для вирішення внутрішніх і зовнішніх загроз контейнерів розробники можуть реалізувати POLP за допомогою [8]:

- надання мінімальних прав користувачам;

- усунення привілейованого доступу до запуску служб;
- обмеження використання доступних ресурсів, наприклад обмеження доступу контейнера до операційної системи хоста;
- заборони зберігання конфіденційної інформації на контейнері.

Також чудовим методом підвищення захисту мікросервісної архітектури є практика використання маркерів доступу та ідентифікації.

Ефективна автентифікація та авторизація мають вирішальне значення для забезпечення безпечного доступу до мікросервісу. Такий захист доступу може включати безпеку серверних служб, коду проміжного програмного забезпечення та інтерфейсу користувача. Прикладами систем аутентифікації та авторизації, які можуть бути використані у мікросервісній архітектурі є OpenID і OAuth 2.0, що являють собою системи автентифікації, які генерують маркери користувача для безпечного доступу в розподілених системах. Ефективний контроль автентифікації та авторизації дозволяє відстежувати контроль доступу в режимі реального часу та дозволяє командам розробників бачити вплив користувачів на систему і виявляти порушення перед розгортанням.

2.3.3 Захист точок входу

Велика кількість компонентів архітектури мікросервісів веде до більшої поверхні атаки, ніж у монолітних програмах. Кожен мікросервіс має додаткові точки входу та обробляє запити на доступ незалежно. Кожен мікросервіс спілкується за допомогою API, які зловмисники можуть відкрити, що призводить до значного збільшення площі атаки. Тому вкрай важливо захистити всі рівні інфраструктури програми мікросервісу [9].

Щоб зменшити загальну поверхню атаки, розробникам слід захистити не лише точки входу та виходу програми, але й усі API та точки входу мікросервісів. Використання автоматизованого наскрізного сканування може допомогти командам відстежувати загрози в усіх службах.

Створення шлюзу API є дуже важливим методом забезпечення захисту мікросервісів, адже створення однієї точки входу для всіх клієнтів і систем у мікросервісах допоможе подолати проблеми безпеки хмари, які можуть бути пов’язані з використанням кількох технологій, інтерфейсів і протоколів. Наприклад, якщо всі системи підключено до шлюзу API, його можна використовувати для фільтрації запитів до конфіденційних ресурсів і виконання автентифікації та авторизації.

На думку автора даної роботи шлюз API може надавати такі функції безпеки, як:

- шифрування та дешифрування SSL на системі балансування навантажень;
- перетворення протоколу;
- моніторингу;
- маршрутизації;
- кешування запитів.

Використання шлюзу API може значно допомогти підвищити захист додатку.

2.3.4 Використання SBOM

Корисним у використанні також є створення номенклатури матеріалів програмного забезпечення (SBOM).

SBOM — це стандартизований список усіх компонентів, які використовуються в збірці програмного забезпечення. Документації SBOM можуть містити у собі: дані щодо джерел, ліцензії, залежності, версії та іншу інформацію.

SBOM забезпечує видимість вразливостей відкритого коду та ліцензійних вимог.

Аналіз складу програмного забезпечення інформуватиме всіх членів організації про останні версії програмного забезпечення сторонніх розробників, які використовуються, будь-які виправлення, які потрібно впровадити, і відповідні

ліцензії. SBOM має включати поля даних, підтримку автоматизації, а також задокументовані практики та процеси, які використовує організація.

SBOM є основою управління ризиками в ланцюжку постачання програмного забезпечення та допомагає організаціям [10]:

- прискорити процес виявлення та усунення вразливостей;
- дотримуватися державних постанов;
- відповідати вимогам бібліотек із відкритим вихідним кодом;
- усунути зайву роботу для розробників;
- легко виявляти зламані компоненти;
- виконувати запити на аудит.

Збірка настільки безпечна, наскільки безпечні компоненти, на яких вона побудована.

2.3.5 Забезпечення спостережності служб додатку

Однією з найкращих методик захисту мікросервісів є використання журналів подій для покращення можливостей моніторингу процесів виконання сервісів.

Традиційні методи журналювання недостатні для програм, що складаються з кількох розподілених незалежних компонентів. Сервіси, створені з використанням різних технологій, часто є мультихмарними і призводять до великої кількості структур журналу. Кожна служба підтримує власний набір даних, і керування механізмами журналювання подій може стати складним через збільшення кількості служб. У команд можуть виникнути труднощі з визначенням проблеми та потоку запитів, якщо служба не працює. Щоб уникнути цих проблем, журнал мікросервісів має бути централізованим і стандартизованим для всіх сервісів. Програмне забезпечення також має мати можливість не тільки збирати журнали з усіх служб, але і представляти всю відповідну інформацію у візуальному форматі так, щоб команди могли швидко знайти та усунути проблему при потребі. Унікальний ідентифікатор

кореляції, призначений кожній службі, також може допомогти розробникам визначити, де сталася помилка, за умови, що вона включена в дані журналу.

2.3.6 Ізоляція

Ізоляція мікросервісів може сильно допомогти у забезпеченні захисту програмного застосунку побудованого на мікросервісній архітектурі. Сегментація мікросервісів являється одним із основних методів захисту мікросервісної архітектури. Враховуючи те, що кожна служба є ізольованою частиною програми, кожну частину сервісу можна впроваджувати, змінювати, підтримувати, розширювати та оновлювати, не впливаючи на інші мікросервіси.

Інші рівні інфраструктури, такі як база даних, також повинні бути ізольовані. Мікросервіси повинні мати доступ лише до своїх власних даних, щоб зловмисник, який отримав до них доступ, не міг здійснити атаку на інші сервіси.

Розгортання мікросервісів, реалізованих як контейнери, є найпростішим способом створити ізоляцію та застосувати спеціальні заходи безпеки. Такі контейнери засновані на зображеннях, які можуть містити вразливості та можуть бути скомпрометовані. Один скомпрометований контейнер може вплинути на всі інші контейнери, що працюють у тій самій операційній системі [11].

Прийняття принципу безпеки найменших привілеїв разом із обмеженням і моніторингом доступу до доступних ресурсів може допомогти забезпечити безпеку контейнера. Також використання інструменту оркестрування контейнерів може значно пришвидшити розгортання, одночасно автоматизуючи процеси безпеки.

2.3.7 Автоматизація та тестування

Багатьом розробникам, що працюють з мікросервісною архітектурою варто приділити увагу автоматизації та тестуванню мікросервісів.

Автоматизація аналізу коду та управління тестуванням зробить процес захисту мікросервісів набагато ефективнішим.

Невід’ємною частиною розробки є сканування програмного коду на вразливості. Необхідність перевірки наявних дефектів вихідного коду, , помилок надає можливість у наслідку після завершення процесу розробки сервісу мінімізувати ризики викликів безпеки та покращити продуктивність програмного забезпечення.

На думку автора даної роботи захист коду має забезпечувати такі якості програмного забезпечення:

- ефективність;
- ремонтопридатність;
- портативність;
- надійність;
- безпеку.

Статистика дозволяє розробникам приділяти більше уваги сповіщенням про вразливість системи безпеки або конфлікту ліцензій у наслідок чього розробники можуть усунути ризики безпеки. Невикористані функції в розгортаннях з відкритим кодом можуть спричинити проблеми із залежностями. Аналіз якості коду може допомогти визначити невикористаний код, щоб його можна було видалити, зменшуючи ризик проблем із залежністю мікросервісів та оптимізувати код.

Висновки за розділом 2

1. Проведено загальний огляд вразливостей мікросервісної архітектури.
2. Виділено основні проблеми з якими стикаються розробники мікросервісів.
3. Визначено виклики безпеки, які можуть вплинути на захист мікросервісної архітектури.
4. Розглянуто найкращі методи захисту мікросервісної архітектури. Проведено аналіз даних методів захисту.

РОЗДІЛ 3
ВИКОРИСТАННЯ МОНІТОРИНГУ ДЛЯ ОПТИМАЛЬНОГО ЗАХИСТУ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

3.1 Огляд роботи систем моніторингу

Зі зростанням попиту на використання мікросервісної архітектури зростає попит на її захист. Одною з найкращих та найпоширеніших практик захисту мікросервісної архітектури вважається забезпечення відстежуваності внутрішніх процесів системи за допомогою моніторингу. Оскільки необхідність відстежувати всі процеси та події є дуже великою, то важливість побудови системи моніторингу стає лише питанням часу.

Моніторинг являє собою систему постійного спостереження за внутрішніми та зовнішніми процесами роботи додатку. Моніторинг спрямований на виявлення несправностей у системі для їх подальшого усунення. Методи, що використовуються в моніторингу інформаційних систем, перетинаються з обробкою в реальному часі, статистикою та аналізом даних. Набір програмних компонентів, які використовуються для збору даних, їх обробки та представлення, називають системою моніторингу.

Найчастіше безперервний моніторинг відповідає за:

- колекцію даних;
- зберігання даних;
- візуалізацію та аналіз зібраних показників, побудову графіків/метрик;
- оповіщення користувачів про події.

Невід’ємною складовою частиною моніторингу є показники та попередження.

Показники — це необроблені дані про використання ресурсів або поведінку, які система моніторингу збирає з будь-яких програм або служб в інфраструктурі.

Попередження (alert) — це здатність системи моніторингу виявляти й повідомляти команди про значущі події, які вказують на серйозну зміну стану

показників того чи іншого внутрішнього процесу роботи додатку. Сповіщення являє собою просте повідомлення, яке може приймати різні форми: повідомлення електронної пошти, SMS, миттєве повідомлення або телефонний дзвінок.

Система безперервного моніторингу допомагає: швидко виявляти проблеми, підтримувати безперебійність роботи додатку та покращувати його доступність і продуктивність роботи. Також моніторинг безперечно впливає на безпеку програмного забезпечення за яким впроваджено нагляд.

Впровадження системи моніторингу для спостереження за додатком на мікросервісній архітектури є дуже корисним і необхідним заходом підвищення безпеки програмного забезпечення, що дає можливість розробникам краще розумітися на процесах сервісів та швидше знаходити помилки за допомогою відстежуваності програмного застосунку [12].

На думку автора даної роботи моніторинг мікросервісів є необхідним для того щоб:

- зменшити кількість ризиків виникнення інцидентів з інформаційної безпеки;
- збільшити спостережність сервісів та успіх їх роботи;
- зменшити кількість неоптимізованих сервісів;
- допомогти компанії дотримуватися угод про рівень обслуговування;
- оптимізувати взаємодії з кінцевим користувачем.

Специфіка мікросервісної архітектури пов’язана на багатьох маленьких сервісах які виконують певні задачі.

Враховуючи структуру побудови, моніторинг і керування мікросервісів може бути особливо складним, оскільки в програмі можуть бути сотні сервісів і тисячі екземплярів сервісів.

Моніторинг усіх компонентів служб та їх взаємодії може бути не простим, він потребує покращення спостережуваності в окремих службах, створення підґрунтя для виявлення збоїв, впровадження централізованого збору журналів і можливості збору показників із журналів для виявлення ряду виробничих проблем.

3.1.1 Моніторинг мікросервісів

Моніторинг мікросервісів (рис. 3.1) фактично являє собою впроваджену систему спостереження у мікросервісну архітектуру для постійного нагляду за процесами роботи програмного забезпечення.

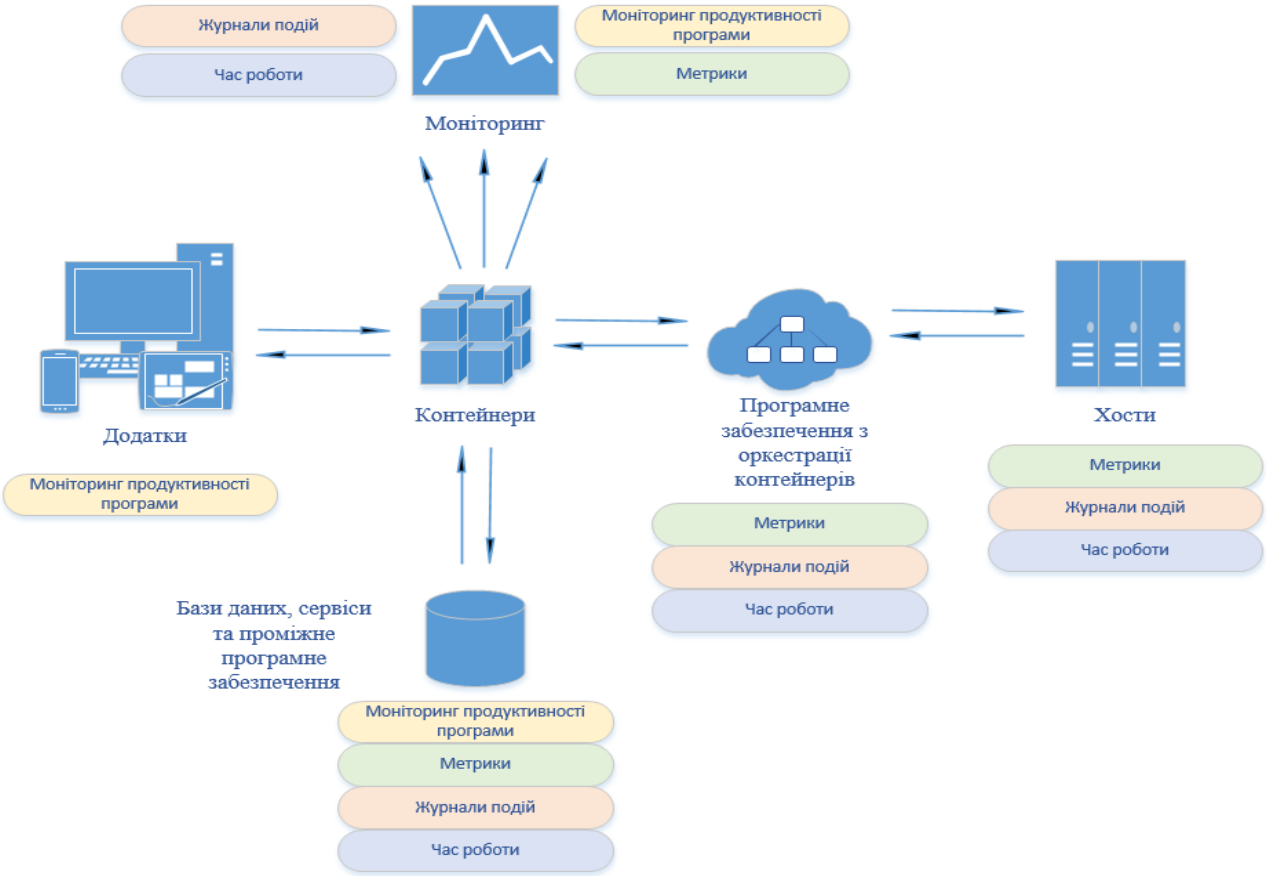

Рисунок 3.1 – Схематичний приклад роботи моніторингу мікросервісів

На думку автора даної роботи моніторинг мікросервісів є необхідним для того щоб:

- зменшити кількість ризиків виникнення інцидентів з інформаційної безпеки;
- збільшити спостережність сервісів та успіх їх роботи;
- зменшити кількість неоптимізованих сервісів;
- допомогти компанії дотримуватися SLA;

- оптимізувати взаємодії з кінцевим користувачем.

Оскільки архітектура мікросервісів розбиває програму на кілька незалежних частин, роль захищенності мережі у даній архітектурі стає дуже важливою. Більшість мікросервісних служб покладаються на виклики API для взаємодії один з одним, і моніторинг цих викликів може дати чудове уявлення про продуктивність роботи програми [13].

Існує велика ймовірність того, що при розробці програмного продукту розробниками буде обрано контейнеризацію як спосіб ефективного розгортання своєї програми для кінцевих користувачів. Такий підхід має свої позитивні сторони, у разі вибору контейнеризації набагато легше можна отримати додаткову інформацію про продуктивність додатка, відстежуючи специфічні для контейнера показники. Крім того, моніторинг контейнерної програми, яка керується інструментом оркестрації, дуже відрізняється від моніторингу традиційної програми, оскільки такий моніторинг може керуватися оркестратором контейнеру, наприклад таким як Kubernetes. Звичайний моніторинг може не завжди підходити для контейнеризації, адже масштабування та моніторинг програми яка може складати до тисяч контейнерів може бути дуже складним та дорогим способом захисту, що додасть непотрібних накладних витрат до вартості та продуктивності програми.

Моніторинг контейнерів зазвичай включає у себе [14]:

- моніторинг контейнерів і внутрішніх сервісів;
- збирання даних про APM (моніторинг та управління продуктивністю і доступністю додатку);
- API моніторинг;
- відображення моніторингу в організаційній структурі.

Як і будь-яка інша форма моніторингу, моніторинг мікросервісів життєво важливий для того, щоб програма на основі мікросервісів не виходила з ладу.

Окрім простого запобігання збою програми, моніторинг мікросервісів може допомагати отримати широке уявлення про продуктивність мікросервісної

архітектури і навіть спонукати розробників зробити оптимізацію, яка призведе до більшого результату або зниження витрат.

Якщо моніторинг правильно проведений можна навіть передбачити проблеми з продуктивністю ще до їх виникнення. Це, у свою чергу, може допомогти компанії запобігти повним збоям, що забезпечує продуктивність та безперервність роботи, і у наслідок підвищує репутацію компанії у клієнтів сервісів.

Кожна компанія яка володіє власним програмним забезпеченням не залежно від архітектури яку вона використовує має дотримуватися угод про рівень обслуговування, які встановлюються для продуктивності програми. Угоди про рівень обслуговування (SLA) – це угоди, які компанія укладаєте зі своїми клієнтами/користувачами щодо надання якості послуг, які їм пропонуються. Угоди про рівень обслуговування мають юридичні наслідки, і їх недотримання може призвести до юридичних проблем із зацікавленими сторонами.

Моніторинг мікросервісів не тільки допомагає компаніям дотримуватися угод про рівень обслуговування, але й допомагає краще визначити рівень обслуговування. Без ефективного налаштування моніторингу неможливо було б оцінити якість послуг, які пропонуються.

Для розробників мікросервісної архітектури необхідно визначити різні шаблони для спостереження за роботою своїх сервісів на відміну від моноліту, де зазвичай використовується в основному один шаблон моніторингу.

Оскільки моноліти пов’язані тісніше, ніж програми на основі мікросервісів, у мікросервісній архітектурі може бути важко відстежувати все, що відбувається всередині них. Без цієї видимості можна втратити важливу інформацію про те, як можна додатково оптимізувати продуктивність системи.

Відстеження продуктивності та поведінки мікросервісів допоможе компанії зрозуміти поведінку кожного компонента окремо. Це створює високий шанс командам розробників помітити будь-які колективні чи незалежні тенденції у програмі та використовувати їх для подальшої оптимізації роботи сервісів.

Оптимізація завжди є основою моніторингу. Відстежуючи мікросервіси, компанії набагато легше зрозуміти, як саме зусилля щодо розробки відображаються

на користувачах. Таким чином можна старатися дуже сильно розвивати додаток досліджуючи функції, які найбільше подобаються користувачам, і дотримуватися всіх можливих передових методів розробки, але невеликий витік пам’яті чи збій кінцевої точки можуть змарнувати всі зусилля. Враховуючи ситуацію відсутності моніторингу, компанія може і не знати про втрати продуктивності, але користувачі можуть зіткнутися з ними під час використання послуг.

Дуже важливо переконатися, що налаштований як і внутрішній моніторинг, так і зовнішній моніторинг. Тобто мають бути налаштованими синтетичні агенти, які будуть запускати програму через фіксовані проміжки часу з усього світу та вмикати заздалегідь визначені сценарії тестів, щоб переконатися, що все працює належним чином. Це не тільки покращує користувальницький досвід програми, але й надає компанії цінні відгуки під час процесу розробки.

При налаштуванні моніторингу мікросервісів треба звертати увагу на деякі показники такі як системні події та інші специфічні показники.

Специфічні показники програми це в основному показники, пов’язані з бізнес-логікою програми. Наприклад, кількість активних користувачів буде показником додатку для прямих трансляцій, або кількість покупців, які переглядають програмний продукт буде показником для електронної комерції.

Метою відстеження цих показників є збір даних високого рівня з програми для підтримки бізнес-операцій, а також для підтримки розробників у виявленні аномалій у використанні програми.

Також необхідно приділити увагу специфічним для платформи показникам, які зосереджені на базовій інфраструктурі програми. Вони можуть включати будь-що: від часу виконання запиту до бази даних до середнього часу відповіді кінцевої точки. Ці показники не враховують бізнес-логіку, а націлені на надання низькорівневої системної статистики.

Окрім специфічних для програми та платформи, існує ряд зовнішніх факторів, які можуть впливати на продуктивність програми. Системні події – це події, які відбуваються в системі, у якій працює програма, і які можуть призвести до перебоїв

у її роботі. Наприклад, нове розгортання вважається досить витратним за ресурсами і може вплинути на роботу кінцевого користувача під час їх виконання.

3.1.2 Найкращі метрики для моніторингу

Для успішного впровадження моніторингу у систему захисту мікросервісів необхідно розумітися на найважливіших показниках програми, за якими варто вести спостереження: показниками платформи, показниками ресурсів, золотими метриками.

Моніторинг показників платформи має вирішальне значення для безперебійної роботи інфраструктури мікросервісів. Показники платформи це низькорівневі дані, які можуть вказувати на проблеми в базовому комп'ютері, сховищі чи мережевому обладнанні. Ретельний моніторинг цих показників може виявити зниження продуктивності та запобігти системним збоям.

Показники платформи зазвичай включають [15]:

- кількість запитів в секунду/хвилину;
- кількість невдалих запитів за секунду;
- середній час відповіді на кінцеву точку служби;
- розподіл часу, необхідного для кожного запиту;
- середній час виконання для найшвидших 10% і найповільніших 10% запитів;
- відсоток успіхів/невдач за послугами.

Показники ресурсів також є важливими для моніторингу адже вони дозволяють розробникам чітко визначати продуктивність програми та при необхідності підвищувати оптимізацію.

Постачальники інфраструктури зазвичай можуть надавати показники ресурсів, корисні для моніторингу стану інфраструктури. У хмарі ці показники надає така система, як AWS CloudWatch; у локальному середовищі це можуть бути показники Kubernetes.

Приклади показників ресурсів включають [16]:

- використання ЦП і пам’яті вузлів і контейнерів;
- кількість хостів або модулів, на яких запущена система;
- кількість живих потоків;
- використання пам’яті купи.

Концепція «золотих сигналів» відноситься до показників, які дуже корисні для моніторингу працездатності мікросервісу або всієї програми мікросервісу, виявлення та вирішення проблем.

Приклади золотих сигналів включають [15]:

- доступність — стан системи, який вимірюється з точки зору клієнта, наприклад, співвідношенням помилок до загальної кількості запитів;
- справність — стан системи, який вимірюється за допомогою регулярних запитів ping;
- частота запитів — швидкість надходження запитів до системи;
- насиченість — ступінь вільності чи завантаження системи під час простою або завантаження системи (наприклад, доступна пам’ять або глибина черги);
- використання — рівень використання системи (навантаження ЦП, використання пам’яті тощо), виражений у відсотках.
- частота помилок — частота помилок, які виникають у системі.
- затримка — час відгуку системи.

Безперервний моніторинг цих показників може значно допомогти підвищити продуктивність роботи додатку та позитивно вплинути на забезпечення його безпеки завдяки великому рівню спостережності сервісів.

3.2 Вплив моніторингу на безпеку мікросервісів

Моніторинг мікросервісної архітектури має величезний вплив на безпеку всього додатку. Забезпечення безперервного спостереження за внутрішніми

сервісами та ресурсами є необхідним для створення дуже важливого фундаменту видимості процесів, ресурсів, помилок для забезпечення найкращого захисту додатку.

Постійний моніторинг — це процес і технологія, які використовуються для виявлення проблем відповідності та ризиків, пов’язаних із фінансовим і операційним середовищем додатку. Фінансове та операційне середовище складається з процесів і систем, які працюють разом для підтримки ефективної та якісної діяльності програмного продукту. Для усунення ризиків у цих компонентах використовуються засоби контролю. Завдяки безперервному моніторингу операцій і засобам контролю можна виправити або замінити погано розроблені ланки додатку або зміцнити його слабкі сторони. [17]

Під час впровадження системи безперервного моніторингу для нагляду за додатком на мікросервісній архітектурі необхідним є визначення не тільки основних сервісів, які необхідно моніторити, не тільки метрик продуктивності роботи самого додатку, за якими має бути впроваджений постійний нагляд, а і мають бути чітко сформульовані та описані заходи контролю безпеки додатку.

Для продуктивного використання безперервного моніторингу для спостереження за роботою додатку керівництвом з інформаційної безпеки (ISM) має бути визначено [18]:

- періодичність проведення додаткового сканування сервісів (принаймні раз на місяць);
- періодичність проведення оцінки вразливості або тестів на проникнення для систем (принаймні раз на рік);
- вразливі місця безпеки, для аналізу їх потенційного впливу і відповідних засобів пом'якшення на основі ефективності, вартості та існуючих засобів контролю безпеки
- підхід, що ґрунтується на оцінці викликів безпеки, щоб визначити пріоритетність впровадження заходів пом’якшення ризиків.

За документацією NIST також має бути сформульовано стандартний процес для проведення безперервного моніторингу [19]:

- визначено стратегію безперервного моніторингу, засновану на толерантності до ризику, яка забезпечує чітку видимість активів і усвідомлення вразливостей, актуальну інформацію про загрози та вплив на бізнес;
- встановлено показники метрик, частоти моніторингу стану додатку, частоти оцінки контролю безпеки;
- запроваджено програми постійного моніторингу для збору даних, пов’язаних із безпекою;
- введено дію аналізу зібраних даних у постійну практику та запроваджено ведення документацій;
- запроваджено перегляд звітів моніторингів та відповідне реагування на висновки системи спостережності;
- забезпечено постійне оновлення програми моніторингу, перегляд стратегій безперервного моніторингу, вдосконалення можливостей забезпечено видимості активів та підвищення рівню усвідомлення вразливостей; посилено керування даними контролю безпеки інформаційної інфраструктури організації; та підвищено організаційну гнучкість.

За NIST 800-55 компанією мають бути розроблені заходи контролю безпеки на організаційному, програмному та системному рівнях. Також має бути проведена оцінка наступних аспектів контролю безпеки [19]:

- оцінка прогресу у впровадженні програм інформаційної безпеки, оцінка конкретних заходів безпеки та відповідних політик і процедур;
- оцінка ефективності захищенності додатку за допомогою перевірки правильності реалізації засобів контролю безпеки враховуючи перевірку того, що вони працюють за призначенням і досягають бажаного результату;
- оцінка впливу контролю безпеки на компанію.

При використанні безперервного моніторингу мають бути визначені ролі та обов'язки. Вони безперечно мають визначальне значення при побудові системи безперервного моніторингу.

Одними з ключових ролей у системі моніторингу на думку автора виділяються:

- головний спеціаліст з інформаційної безпеки;
- власник системи;
- операційні центри безпеки;
- групи управління вразливістю;
- групи підтримки робочого столу;
- групи хмарних операцій.

Обов’язки, пов’язані з безперервним моніторингом, можуть включати збір, аналіз і звітування даних постійного моніторингу. Ці обов’язки мають бути розподілені на відповідні визначені ролі.

Якщо ролі та обов'язки розподілені правильно і якщо кожен працівник виконує свою роботу відповідно коректно сформульованим правилам роботи з сервісами додатку, то у наслідок програмне забезпечення буде не тільки оптимізованим і якісним, а і захищеним від зовнішніх загроз. Компанією обов'язково має бути створений чіткій і зрозумілий перелік обов'язків для кожної ролі у системі. Особливо для ролей пов'язаних з безпекою додатку.

Розуміння працівниками своєї роботи і сумлінне її виконання є важливим, але нічого не було б можливим без сервісів які необхідно моніторити та захищати.

При впроваджені безперервного моніторингу додатку на мікросервісах дуже важливо розуміти специфіку мікросервісної архітектури. Дана архітектура зазвичай генерує величезну кількість цифрової інформації з таких джерел, як мережеві пристрої, бази даних, сервери та кінцеві точки. Отже необхідним є визначення джерел інформації потрібних для моніторингу, з ціллю підтримки обізнаності про поточний стан середовища програми.

Компанії, яка будує додаток на мікросервісній архітектурі та впроваджує її моніторинг, необхідно чітко визначити активи та ресурси, які є критичними і мають бути захищеними.

Після визначення даних ресурсів необхідно переконатися що всі вони підпорядковуються єдиній політиці захисту та мають бути обрані певні інструменти захисту критичних ресурсів задля забезпечення найкращої стабільності і безпеки

додатку та його внутрішньої структури. Даний підхід дає змогу створити зручний менеджмент критичних ресурсів та підвищити контроль безпеки.

Забезпечення постійної спостережності за критичними ресурсами додатку (рис. 3.2) надає можливість власникам мікросервісної архітектури отримати додаткову інформацію щодо потенційних загроз безпеки та надає визначити відправні пункти для підвищення захисту програмного забезпечення.

Сукупність активів та ресурсів безперервного моніторингу які мають бути захищеними	Інструменти захисту
Керування вразливостями та виправленнями	Сканер вразливостей База даних вразливостей База даних з управління виправленнями Реєстр програмного забезпечення
Управління подіями та інцидентами	База даних інцидентів SIEM
Виявлення шкідливих програм	База даних інцидентів Антивірусне програмне забезпечення
Управління активами	Реєстр управління активами
Управління конфігураціями	База даних з конфігураціями Реєстр сертифікацій та авторизацій
Мережевий менеджмент	Міжмережевий екран
Управління ліцензіями	Реєстр програмного забезпечення
Управління інформацією	Політика класифікації та категоризації інформації Обмеження доступу

Рисунок 3.2 – Сукупність об'єктів та ресурсів додатку що підпорядковуються єдиній політиці безпеки.

Для покращення захищенності програмного забезпечення на мікросервісній архітектурі у поєднанні з безперервним моніторингом можуть бути використані методики отримання інформації які будуть вказувати на стан якості забезпечення захисту додатку (рис. 3.3).

Методики отримання інформації щодо захищеності системи	
Оцінки ефективності контролю	Ця методика дозволяє отримати детальну інформацію про охоплення, вплив і ефективність роботи засобів контролю безпеки
Сканування вразливостей	Сканування вразливостей використовується для того щоб виявити відомі вразливості в розгорнутому програмному забезпеченні або слабкі місця в конфігурації системи
Управління вразливостями та тестування на проникнення	Оцінка вразливостей та тестування використовуються для того щоб мінімізувати ризики викликів безпеки, виявити відомі вразливості в розгорнутому програмному забезпеченні або слабкі місця в конфігурації системи

Рисунок 3.3 – Методики оцінки захищенності додатку

Також компанією мають бути впроваджені перевірки ефективності роботи додатку на мікросервісах та тестування.

Для покращення принципів роботи безперервного моніторингу мікросервісної архітектури має бути окреслено час та частота проведення перевірки ефективності та тестування для кожного контролю безпеки (рис. 3.4).

Визначаючи часові рамки та частоту, компанія має встановити які контролі безпеки вона застосовує, як вони працюють, як легко їх скомпроментувати та як часто контролі безпеки необхідно оновлювати.

Перевірка ефективності роботи контролів безпеки		
Критичність елементів керування	Чи є контроль безпеки критичним для забезпечення ефективності захисту додатку? Чи є контроль критичним для пріоритетів компанії?	Елементи керування, що забезпечують високу критичну функціональність мають бути під безперервним моніторингом
Непостійність елементів контролю	Чи зміниться ефективність контролю безпеки по мірі розвитку додатку?	Непостійні елементи системи мають найчастіше бути під наглядом системи моніторингу
Надійність елементів керування	Чи має контроль якісь недоліки, які можуть гальмувати ефективність забезпечення захисту додатку?	Елементи керування, які можуть мати недоліки необхідно тримати під пильним наглядом поки недоліки не будуть виправлені.
Толерантність до ризиків	Чи узгоджуються часові рамки вирішення інцидентів безпеки з допустимим ризиком компанії?	Компанії з більш високою толерантністю до ризиків можуть менше проводити моніторинг
Попередження щодо загроз/ інформація про вразливості	Чи отримується компанією інформація щодо загроз та вразливостей програмного продукту?	Додаток має бути постійно сканованим на можливі виклики безпеки, та при виникненні таких мають бути створені звіти та запущений процес аналізу та мінімізації ризиків
Нормативні вимоги	Чи існують обов'язкові часові терміни в межах нормативних актів компанії?	Для узгодження з вимогами можуть бути змінені часові рамки для певних функцій моніторингу (час сканування вразливостей, перегляд документацій)
Вимоги до звітності	Чи визначено вимоги до звітності?	Для узгодження з вимогами можуть бути змінені методи ведення звітності, обліку, аналізу, часу проведення звітності

Рисунок 3.4 – Методики оцінки ефективності роботи контролів безпеки

Багато інструментів моніторингу безпеки збирають і записують інформацію моніторингу, яка є корисною для оцінки загального стану безпеки системи.

Для покращення процесу моніторингу в мікросервісній архітектурі, інформація, необхідна для безперервного моніторингу, має належно зберігатися та управлятися. Часто для таких цілей компанії використовують поєднання безперервного моніторингу з моніторингом безпеки у своїх системах для більш досконалого захисту додатків. Таким прикладом може бути використання системи безпеки та управління подіями (SIEM) для агрегування інформації моніторингу з метою виявлення слабких місць у безпеці додатку. Щоб покращити здатність ідентифікувати неприйнятну або

незвичайну діяльність, компанії за допомогою SIEM можуть інтегрувати аналіз інформації зі сканування вразливостей, моніторинг мережі та інформацію системного журналу.

Також мають бути задокументовані процедури проведення аналізу зібраної інформації щодо визначених заходів. Ці процеси можуть набувати форми робочих інструкцій. Такий підхід полегшує оцінку потенційних вразливостей або слабких місць у повторюваний і послідовний спосіб.

Разом з аналізом та документаціями має бути чітким і зрозумілим процес визначення пріоритетів для реагування на виявлені слабкі місця та вразливі місця системи. Після завершення аналізу та виявлення вразливостей або засобів контролю, які не відповідають очікуваній меті вимірювання, компанія має оцінити пов’язаний ризик і керувати ним, використовуючи методи, детально описані в плані управління ризиками безпеки (SRMP).

Аналіз потенційних наслідків виявлених вразливостей і ризиків на думку автора даної роботи може включати такі запитання, як:

- Яка ймовірність використання вразливості?
- Що станеться/який буде вплив, якщо вразливість буде використана?
- Наскільки ефективні поточні засоби контролю для зниження потенційного ризику?

Перш ніж визначити прийоми зниження ступеня ризику, компанія має розглянути свої рівні толерантності до ризику та визначити, чи доречні ті чи інші дії відносно ризику, наприклад [20]:

- уникнення ризику;
- прийняття ризику;
- запобігання ризику;
- зниження ризику.

Оцінювання має проводитися відповідним кваліфікованим персоналом незалежним від власника чи розробника системи, або третьою стороною, яка не залежить від об’єкта оцінювання.

Компанією мають бути задокументовані відповідні дії відносно ризикових ситуацій та інцидентів пов'язаних з роботою додатку. Вони можуть включати такі дії, як зміни конфігурації системи, навчання, придбання інструментів безпеки, зміна архітектури системи, встановлення нових процедур або оновлення документації політики безпеки.

Визначаючи відповідні заходи реагування, на думку автора роботи, компанією можуть бути поставлені такі запитання:

- Які доступні методи пом’якшення?
- Наскільки ефективними можуть бути ці пом’якшення?
- Хто несе відповідальність за впровадження будь-яких заходів пом’якшення?
- Якою буде вартість впровадження пом’якшення?

Приймаючи рішення про реагування на виклики безпеки, компанія повинна враховувати вимоги щодо управління змінами. Також за допомогою ISM мають бути визначеними слабкі місця та вразливі місця додатку за пріоритетністю на основі їх оціненого ризику або впливу, а також визначені часові рамки, протягом яких мають бути вжиті дії при потребі (рис. 3.5).

Рівень ризику	Час
Екстремальний	48 годин
Високий	1-2 неділі
Середній	1 місяць
Низький	1 місяць

Рисунок 3.5 – Пріоритезація відповіді на ризики

Залежно від виявленої вразливості та її серйозності дії можуть знадобитися негайно або можуть бути реалізовані протягом певного періоду часу.

Агентства повинні розглянути свої рівні толерантності до ризику та перевірити, чи існують процеси для відстеження прогресу дій із відновлення, щойно вони відбуваються.

Разом з документацією, у якій визначається пріорітетність ризиків та відповідей на виклики безпеки, може бути створена документація яка регламентує звітності безперервного моніторингу (рис. 3.6).

Звітності постійного моніторингу мають бути задокументованими, а вимоги звітності визначеними.

Вимого до звітності щодо постійного моніторингу	
Формування звітності	Конкретний персонал і його відповідальності за створення звітів
Зацікавлені сторони	Персонал, ролі, які отримують звітність
Зміст і формат	Вміст звіту та очікуваний формат звіту
Частота звітності	Як часто треба створювати звіти та надсилати їх
Інструменти, що використовуються для звітності	Перелік інструментів для створення і/або автоматизації звітності

Рисунок 3.6 – Вимоги до звітності

Враховуючи побудову додатку на мікросервісній архітектурі та його особливості, при використанні системи безперервного моніторингу необхідно

розуміти важливість оновлень. Процеси безперервного моніторингу не повинні бути статичними, вони повинні адаптуватися на основі змін у загрозах і ризиках, а також при змінах в технологіях та архітектурі робочого середовища. Це дозволить контролям захисту програмного продукту протистояти різноманітним викликам безпеки, які можуть вплинути на роботу додатку на мікросервісній архітектурі.

Визначивши необхідність безперервного моніторингу та виділивши деякі його аспекти використання для покращення безпеки додатку можна зрозуміти що моніторинг мікросервісної архітектури дійсно являється дуже важливим для забезпечення захищеності додатку. Постійний та безперервний моніторинг надає можливість командам, які працюють над сервісами виявляти можливі проблеми продуктивності роботи та потенційні діри безпеки.

При успішному впровадженні системи моніторингу для спостереження за додатком можна зрозуміти як він працює безпосередньо у системі (рис. 3.7).

Рисунок 3.7 – Безперервний моніторинг мікросервісів

За допомогою додатку з моніторингу команди отримують сповіщення про незвичні події у показниках – відповіді програми, які у наслідку переглядаються відповідними за даний сервіс командами.

Після отримання відповіді від програми система моніторингу надає визначення для аномалії за попередніми показниками норми роботи сервісів створеними працівниками. Тобто моніторингова програма оцінює норму поведінки тих чи інших метрик, які показують стабільність роботи програмного забезпечення, та надає відповідь належним командам щодо працездатності їх сервісів.

Якщо графіки відхиляються від норми, часто запускається процес визначення та встановлення критичності поведінки тієї чи іншої служби після цього надсилається робітникам сервісів для аналізу. В наслідок, залежно від впливу на програмний продукт команди вирішують чи потрібно їм створювати додаткові формуляри для вирішення проблеми. Тобто якщо аномалії графіків являються наслідками закономірних подій (наприклад збільшення продаж продуктів, знижки, тощо) то вони просто переглядаються командами та беруться до уваги, але якщо аномалії не закономірні і вплив інциденту може стати небезпечним для роботи додатку, мають бути запущені процеси реалізації мінімізації ризиків та усунення проблеми.

Після моніторингу та перевірки незвичної поведінки показників додатку для кращого забезпечення захисту програми запускається аналіз і звітування. Для звичайних аномалій зазвичай робляться нотатки щодо виникнення та усунення подібної зміни показників, це може бути, наприклад, час відкриття повідомлення про незвичну подію яка зумовлена зменшенням користувачів ввечері, і час закриття повідомлення про аномалію вранці коли показники приходять в норму.

Наслідком небезпечних інцидентів створюються нотатки з визначеннями коренів проблеми та методами усунення подібних ситуацій у майбутньому. Прикладом подібного інциденту може бути ситуація оновлення коду, що зламало можливість користувачів користуватися пошуком на сайті. У даному випадку команди, які займаються забезпеченням стабільності роботи пошуку бачуть небезпечний вплив недосконалого оновлення у системі моніторингу на своїх графіках, створюють інцидент та починають застосовувати дії для зменшення впливу

на додаток. Після усунення впливу має бути запущений процес аналізу інциденту та документування інформації.

Таким чином процеси безперервного моніторингу допомагають відслідковувати роботу додатку та покращувати його продуктивність у наслідок плідної роботи команд (оптимізація ресурсів, виправлення помилок, покращення захисту).

Моніторинг мікросервісної архітектури є не тільки необхідним для забезпечення спостережності роботи сервісів, а й забезпечення їх безперервної роботи та захисту додатку. Отже безперервний моніторинг має бути невід'ємною частиною захисту програмних продуктів побудованих на мікросервісах.

Висновки за розділом 3

1. Проведено загальний огляд моніторингу мікросервісів. Пояснено поняття моніторингу мікросервісів.
2. Пояснено принципи роботи моніторингу мікросервісів.
3. Наведено список найважливіших показників метрик для моніторингу.
4. Проведено аналіз роботи безперервного моніторингу. Виділено особливості впровадження безперервного моніторингу у систему забезпечення безпеки програмного продукту на мікросервісах.
5. Пояснено принципи роботи системи звітування безперервного моніторингу. Виділено якісні сторони використання системи безперервного спостереження за програмними продуктами.

РОЗДІЛ 4
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Проектування програмного забезпечення

Створення програмного забезпечення неможливе без процесу проектування. Для розробки якісного та ефективного програмного забезпечення необхідним є визначення мови програмування та допоміжних сервісів, які можуть бути використані. Також має бути сформованою архітектура побудови додатку.

Для написання програмного забезпечення обрано мову програмування Python. Для побудови Frontend частини веб сторінок обрано JavaScript, HTML, CSS. Такий набір підібрано через те що Python дозволяє швидко і ефективно побудувати програмний продукт також Python постійно оновлюються і залишається актуальним та широко росповсюдженим завдяки простоті та великої кількості внутрішніх модулів та функцій які можуть бути використані при побудові будь-яких додатків. JavaScript, HTML, CSS у свою чергу є дуже зручними для побудови візуальної частини веб продуктів і сервісів.

Для чіткого та зручного розподілу ресурсів створено дві віртуальні машини з операційною системою дистрибутиву Linux Ubuntu. На першій віртуальній машині (VM1) має знаходитись веб сторінка побудована на мікросервісній архітектурі, на другій віртуальній машині (VM2) має знаходитись система моніторингу роботи веб додатку на мікросервісах.

При побудові веб додатку на мікросервісній архітектурі визначається загальна структура програми (рис. 4.1). Вона складається з головної частини та адміністраторської частини.

Адміністраторська частина веб додатку побудованого на мікросервісах створена за допомогою Docker та Django. Вона дозволяє адміністратору додавати та видаляти зміст головної частини веб додатку, це означає що адмін може корегувати контент, який бачить користувач відносно потреби.

Головна частина веб додатку побудованого на мікросервісах створена за допомогою Docker та Flask. Дана частина веб додатку відповідає за контент який бачить користувач веб продукту. Тобто це частина видима клієнту додатку, яка дозволяє йому переглядати та оцінювати контент.

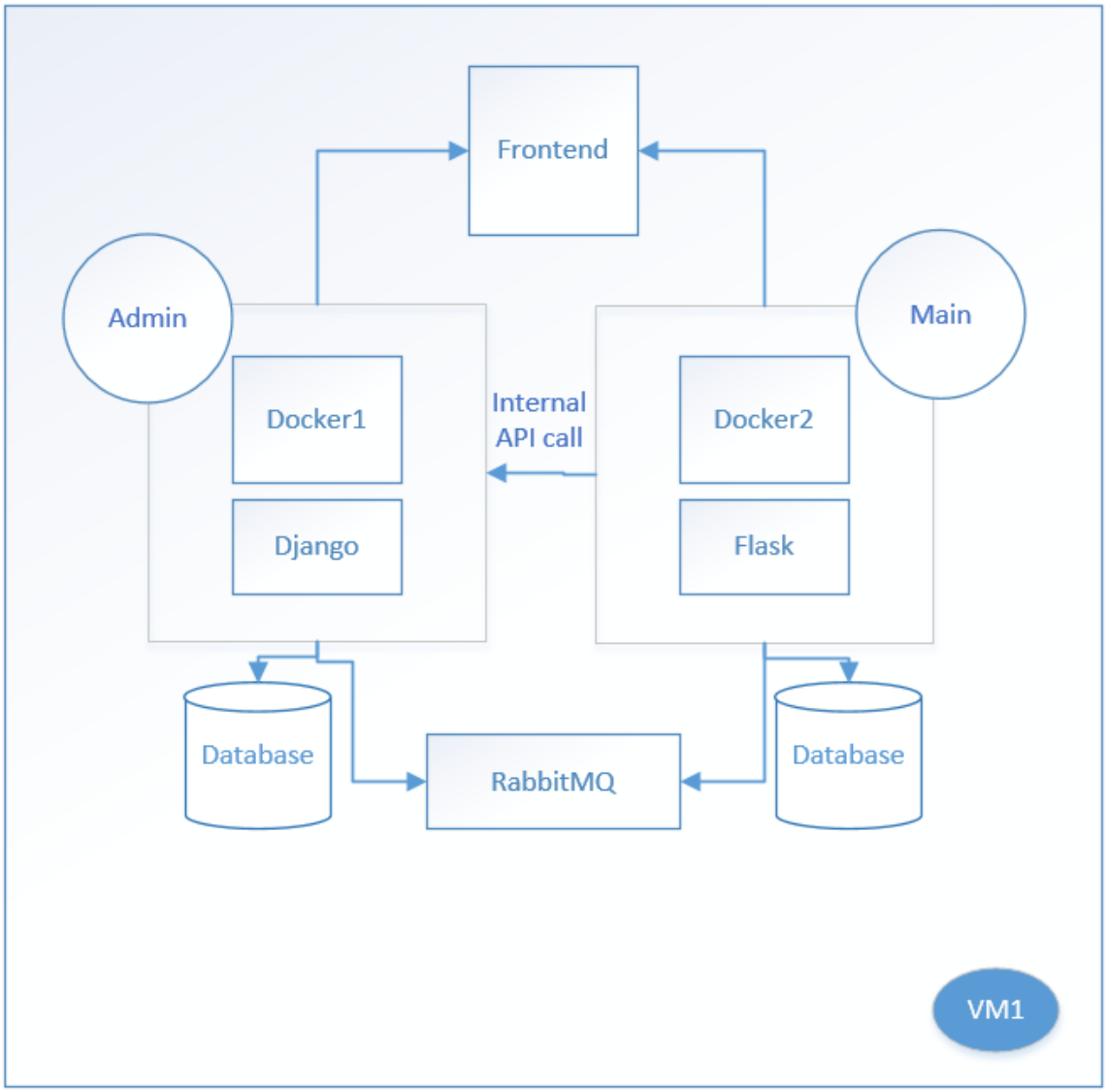

Рисунок 4.1 – Структура веб додатку побудованого на мікросервісах

Обидві з цих двох частин Admin та Main працюють за допомогою Docker та під'єднані до бази даних в MySql. Вони використовують комунікації між собою за допомогою подій RabbitMQ.

При створенні системи моніторингу на другій віртуальній машині (VM2) встановлюються Docker та Django. Програмне забезпечення з моніторингу має

власний інтерфейс виводу інформації та базу даних, з якої воно може брати показники для метрик (рис. 4.2).

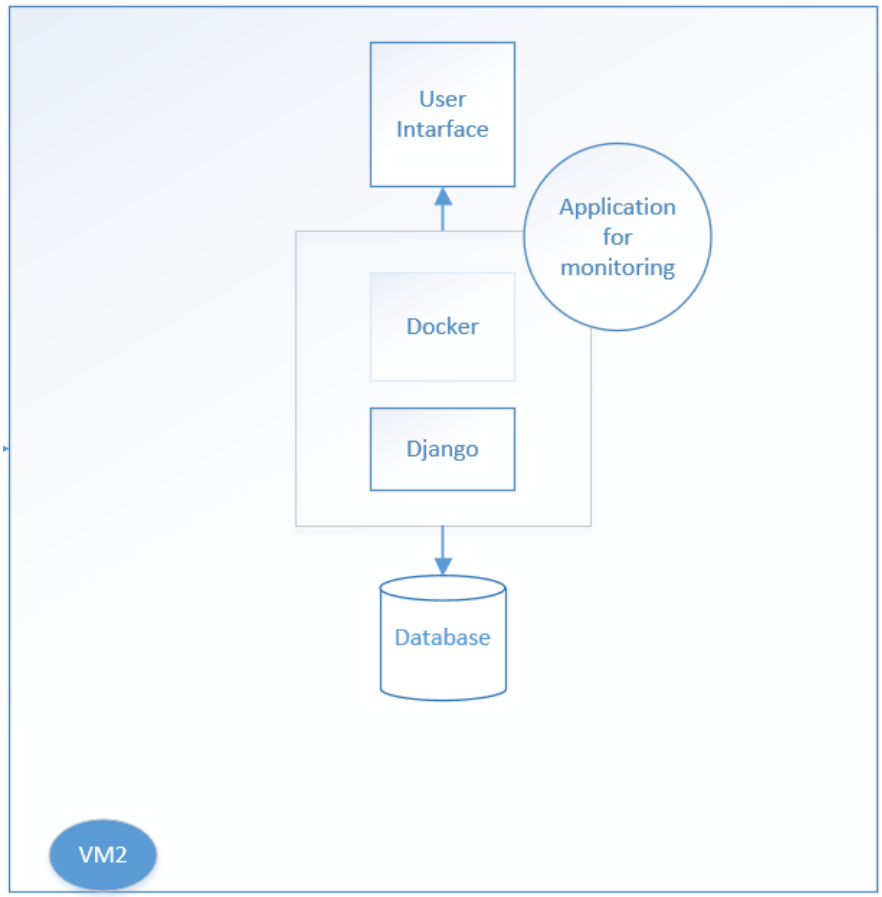

Рисунок 4.2 – Структура системи моніторінгу

Основна ідея полягає зборі даних з першої віртуальної машини (VM1) за допомогою агенту моніторингу, який відправляє інформацію на віртуальну машину (VM2) на програмне забезпечення з моніторингу використовуючи API.

Подібне програмне забезпечення для моніторингу у поєднанні з мікросервісними архітектурами (рис. 4.3) може бути дуже корисним для малих компаній яким потрібна оптимізована, не громіздка система, яка не потребує багатої кількості ресурсів так як потребують сучасні системи моніторингу.

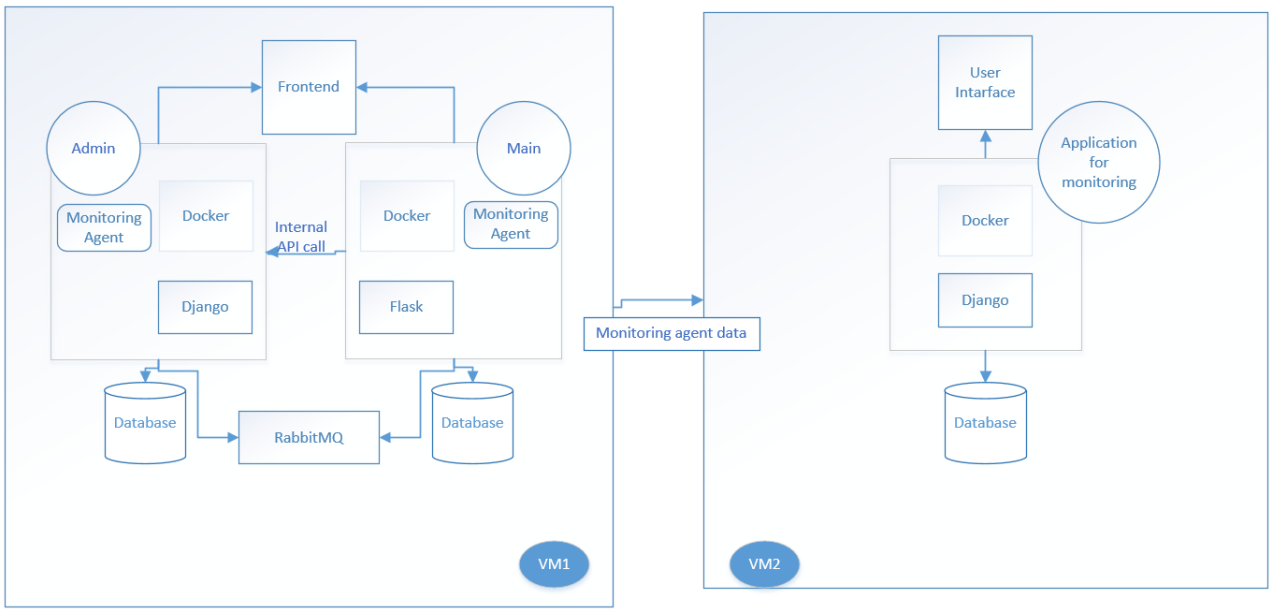

Рисунок 4.3 – Структура взаємодії програмного забезпечення з моніторингу та веб додатку побудованого на мікросервісах

Дана система моніторингу створена для використання у цілях впровадження системи спостереження за якістю і продуктивністю роботи контейнерів програмного продукту задля його безперервної роботи та майбутнього удосконалення.

4.2 Розробка сайту побудованого на мікросервісній архітектурі

Для розробки програмного продукту спочатку необхідно підготувати середовище роботи. На середовище віртуалізації VMware було встановлено віртуальну машину з дистрибутивом операційної системи Linux – Ubuntu. Після цього треба було встановити всі необхідні служби та сервіси.

Для роботи було встановлено Pycharm що є зручним інтегрованим середовищем розробки для мови програмування Python. Також для тестування API було встановлено Postman, і для роботи з мікросервісами було виконано інсталяцію Docker.

Основне середовище розробки програмного забезпечення є підготовленим. Розпочато створення адміністраторської та головної частини веб продукту.

4.2.1 Створення серверної частини додатку на мікросервісах

На початку створення веб додатку побудованого на мікросервісах для роботи обрано першу віртуальну машину (VM1) та на ній встановлено безкоштовний фреймворк Django для веб-проектів на Python з офіційного сайту (рис. 4.4).

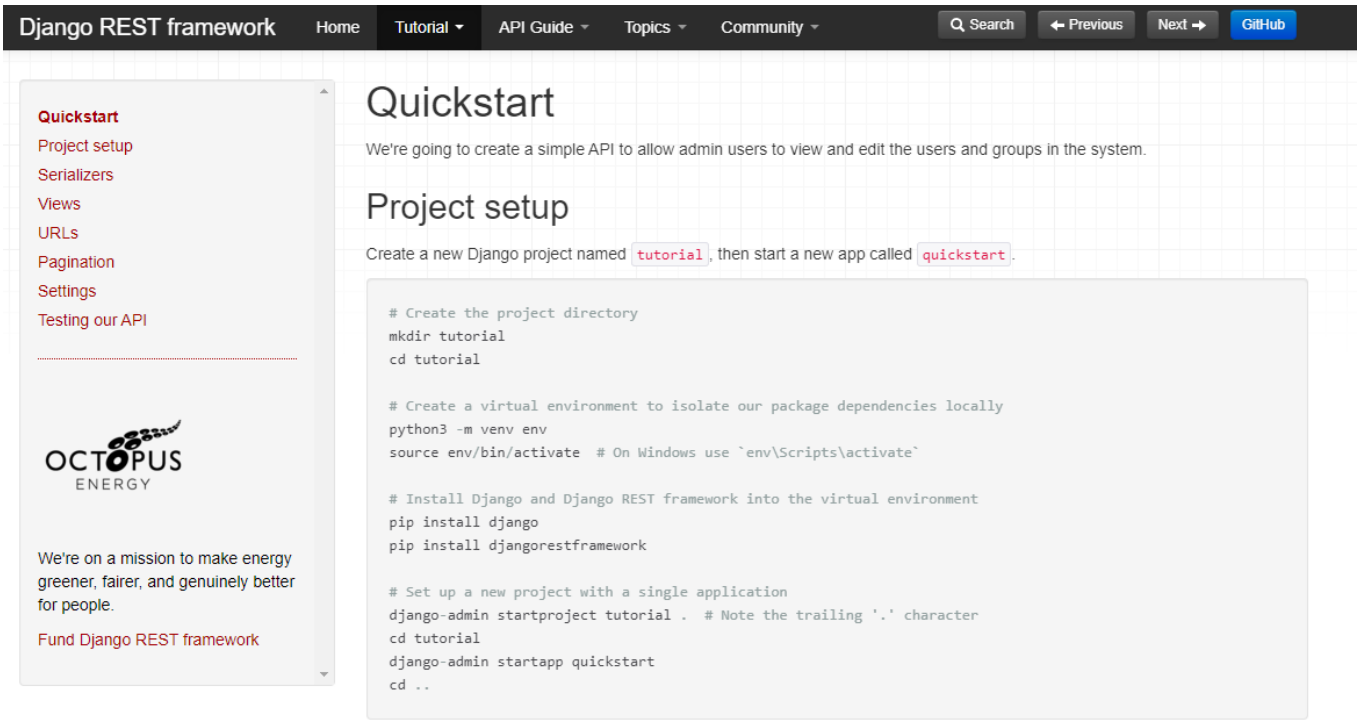

Рисунок 4.4 – Створення веб проекту на Django за допомогою офіційного сайту Django REST Framework

Після створення проекту було створено базу для роботи веб додатку на Docker.

Docker являється програмним забезпеченням для автоматизації розгортання та управління додатками в середовищах з підтримкою контейнеризації, контейнеризатор додатків [21].

У проекті створено 2 файли Dockerfile та docker-compose.

У Dockerfile (рис. 4.5) визначено назву веб додатку (app), визначено хост та визначені основні вимоги для допоміжних сервісів, які буде використовувати контейнеризатор додатків Docker.

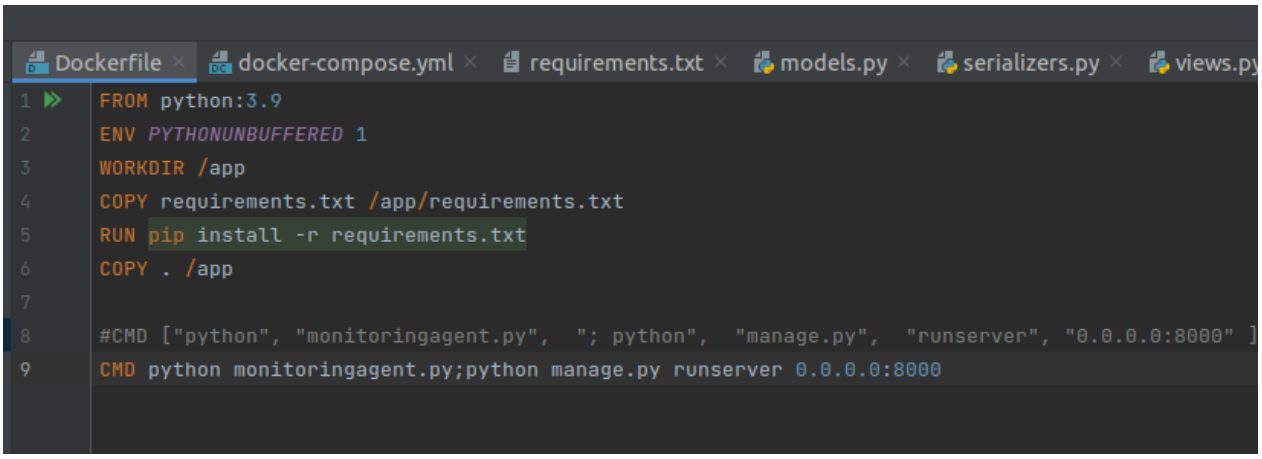

Рисунок 4.5 – Вміст файлу Dockerfile

У docker-compose (рис. 4.6) визначено версію docker-compose, підв'язано Dockerfile файл, виділено сервіси серверної частини, та визначено базу даних. Всі файли всередині даного проекту підв'язані під контейнер Docker.

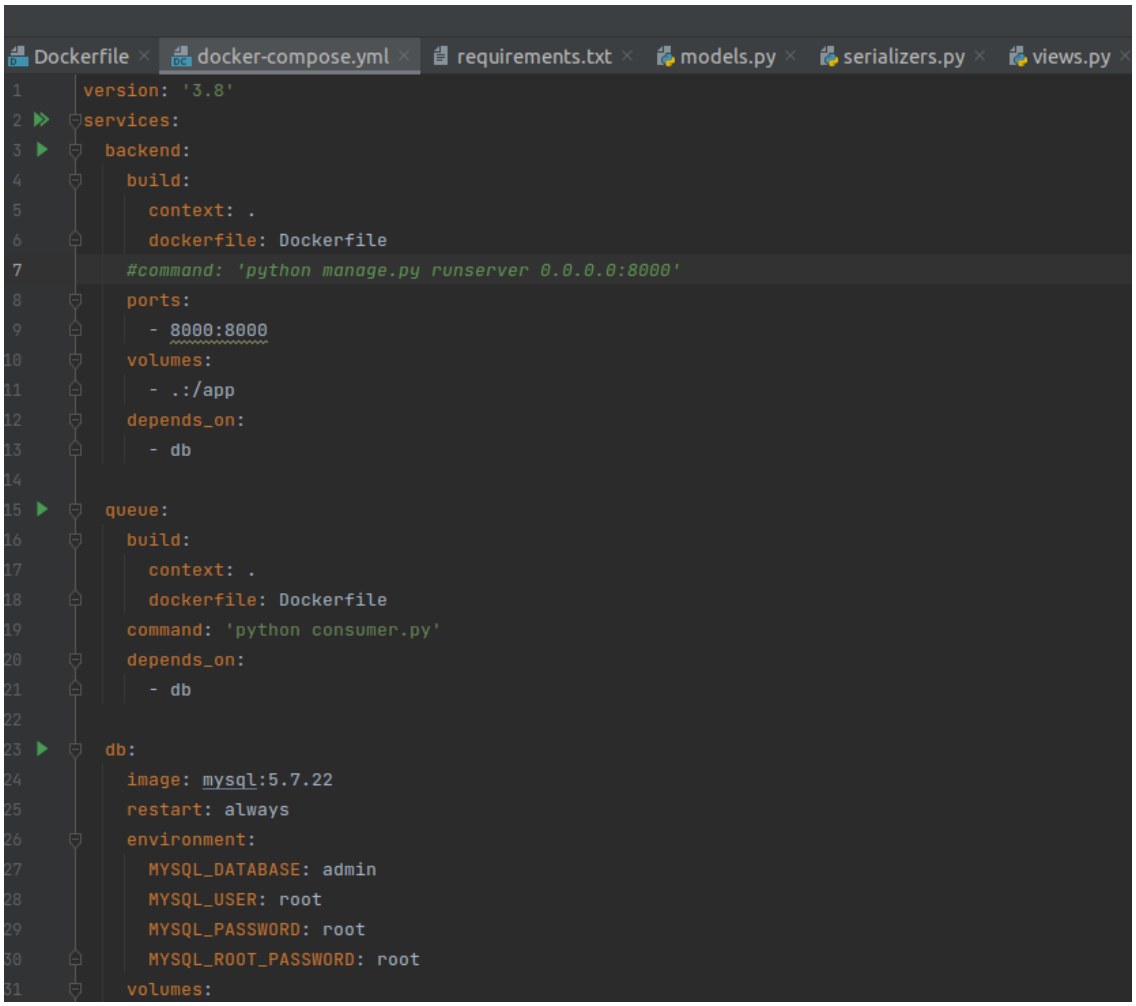

Рисунок 4.6 – Вміст файлу docker-compose

Далі необхідно створити базу даних, яку буде використовувати веб додаток. База даних була створена та вона відображається у основних файлах проекту.

Наступним кроком даної роботи є під’єднання Django з MySQL та з Docker.

У файли налаштувань проекту додано інформацію щодо бази даних (рис. 4.7). Визначено її ім’я, користувач, пароль, хост, порт тощо.

```
# Database
# https://docs.djangoproject.com/en/4.1/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'admin',
        'USER': 'root',
        'PASSWORD': 'root',
        'HOST': 'db',
        'PORT': '3306',
    }
}
```

Рисунок 4.7 – Інформація БД додана до файлів проекту

Для перевірки чи правильно працює база даних запущено сервер за допомогою команди sudo docker-compose up.

За результатами рисунку 4.8 можна побачити, що база даних підтягнулася і працює як треба.

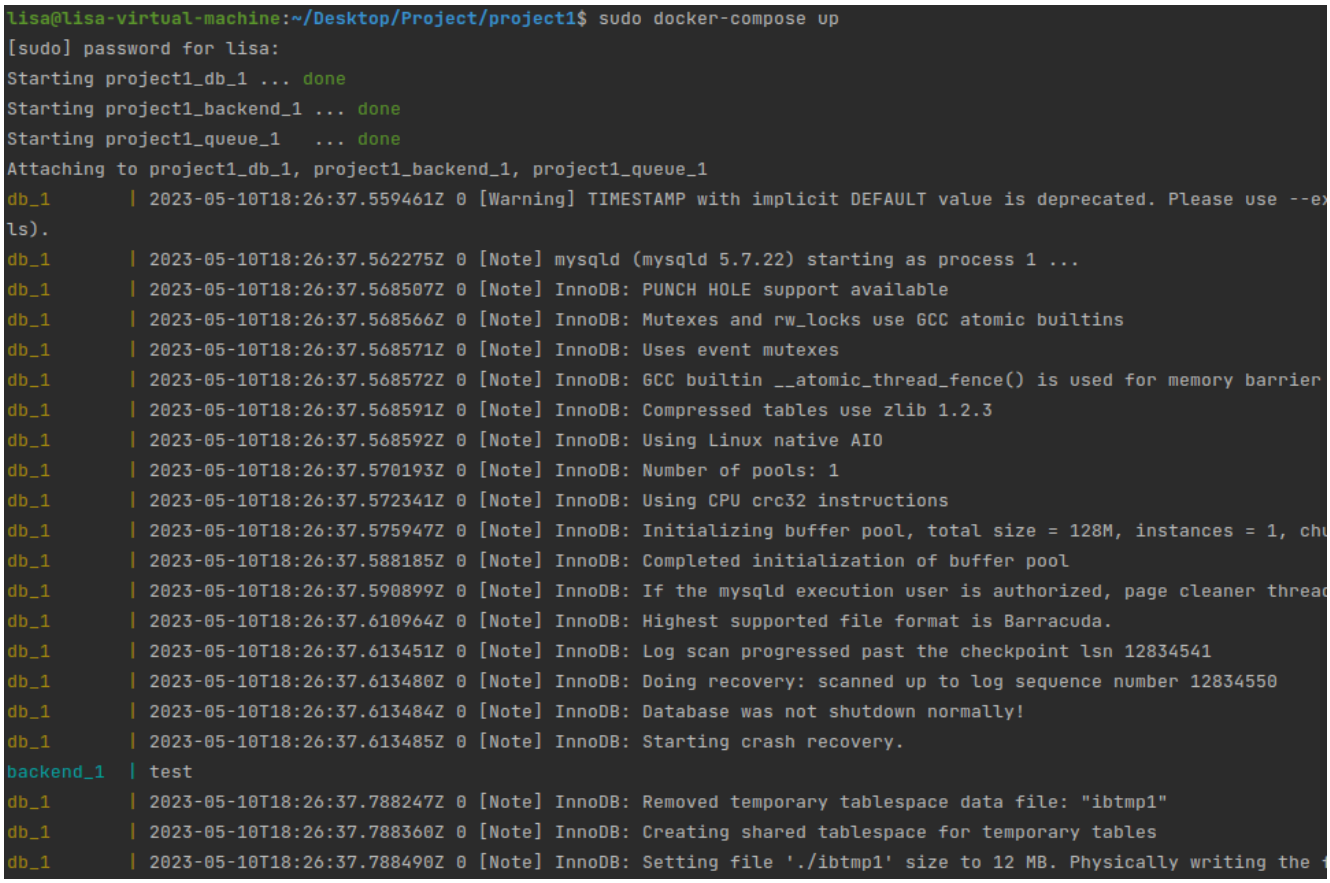

Рисунок 4.8 – Запуск сервісу за допомогою команди sudo docker-compose up.

Після створення у проекті бази даних було написано внутрішні класи та таблиці продуктів бази даних.

При успішному під’єднанні бази даних можна побачити як у контейнері Docker оновлюються дані таблиць після визначення інформації щодо продуктів веб сторінки.

Вся інформація відображається як очікується, отже базу даних можна вважати успішно під’єднаною з Django та з Docker.

Наступним кроком створення адміністраторської частини мікросервісного веб додатку є визначення API та визначення запитів та відповідей для набору переглядів продуктів (рис. 4.9 та рис.4.10). Це необхідно зробити щоб описати взаємодію запитів та відповідей додатку та покращити передачу даних за допомогою API.

```python
from rest_framework import viewsets, status
from rest_framework.response import Response
from rest_framework.views import APIView
from .models import Product, User
from .serializers import ProductSerializer
from.producer import publish
import random


class ProductViewSet(viewsets.ViewSet):

    def list(self, request):
        products = Product.objects.all()
        serializer = ProductSerializer(products, many=True)
        return Response(serializer.data)


    def create(self, request):
        serializer = ProductSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        publish('product_created', serializer.data)
        return Response(serializer.data, status=status.HTTP_201_CREATED)


    def retrieve(self, request, pk=None):
        product = Product.objects.get(id=pk)
        serializer = ProductSerializer(product)
        return Response(serializer.data)
```

Рисунок 4.9 – Визначення API та визначення запитів та відповідей у файлі views.py

```python
    def update(self, request, pk=None):
        product = Product.objects.get(id=pk)
        serializer = ProductSerializer(instance=product, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        publish('product_updated', serializer.data)
        return Response(serializer.data, status=status.HTTP_202_ACCEPTED)

    def destroy(self, request, pk=None):
        product = Product.objects.get(id=pk)
        product.delete()
        publish('product_deleted', pk)
        return Response(status=status.HTTP_204_NO_CONTENT)


usages
lass UserAPIView(APIView):
    4 usages (4 dynamic)
    def get(self, _):
        users = User.objects.all()
        user = random.choice(users)
        return Response({
            'id': user.id
        })
```

Рисунок 4.10 – Вміст файлу views.py

Після формування адміністраторської частини веб додатку було перейдено до роботи над головною частиною сайту.

Для розробки частини, яку буде бачити безпосередньо клієнт було обрано Docker + Flask фреймворк для створення веб-застосунків на Python.

Каркасна частина коду головної частини веб додатку що складається з Dockerfile та docker-compose була написана подібно до адміністраторської, а у Docker вимоги додаткових сервісів (файл requirements.txt) додано Flask.

Після встановлення Flask на Docker контейнер було виконано під'єднання Flask до бази даних за аналогією до адміністраторської частини.

В основному файлі головної частини (main.py) описано модель опрацювання продуктів веб додатку та взаємодію з користувачами і також з’єднано програмний продукт з базою даних. Програмний код даного файлу можна побачити на рисунках 4.11 та 4.12

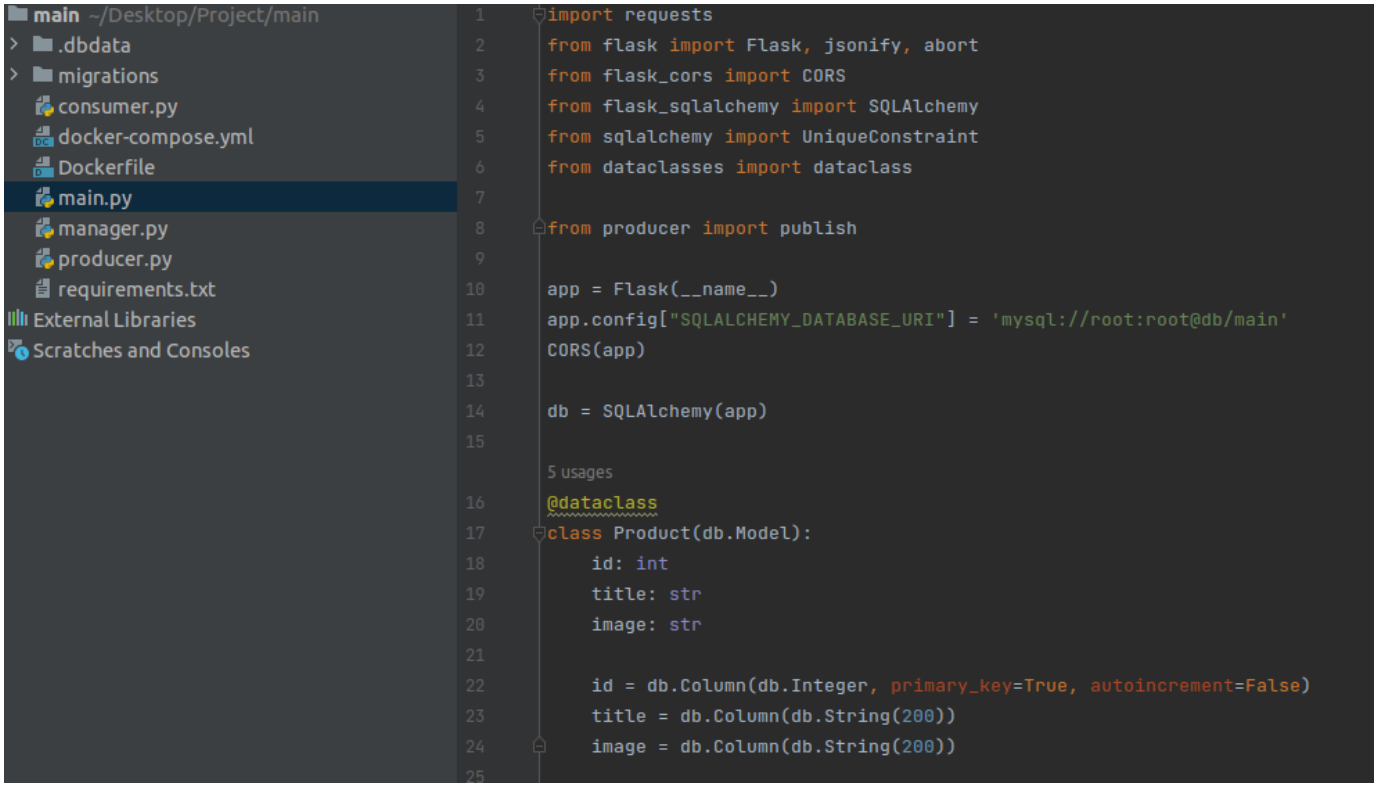

Рисунок 4.11 – Вміст файлу main.py

```
1 usage
@dataclass
class ProductUser(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer)
    product_id = db.Column(db.Integer)

    UniqueConstraint('user_id', 'product_id', name='user_product_unique')

@app.route('/api/products')
def index():
    return jsonify(Product.query.all())

@app.route('/api/products/<int:id>/like', methods=['POST'])
def like(id):
    req = requests.get('http://172.17.0.1:8000/api/user')
    json = req.json()

    try:
        productUser = ProductUser(user_id=json['id'], product_id=id)
        db.session.add(productUser)
        db.session.commit()

        publish('product_liked', id)
    except:
        abort(400, 'You already liked this product')

    return jsonify({
        'message': 'success'
    })
```

Рисунок 4.12 – Вміст файлу main.py

Наступним кроком після створення головної частини веб додатку є під’єднання проекту до RabbitMQ.

RabbitMQ являє собою програмне забезпечення орієнтоване на обробку повідомлень, тому воно було обрано для забезпечення взаємодії адміністраторської частини додатку та головної частини додатку.

На початку роботи з RabbitMQ створено акаунт на сайті CloudAMQP та використано посилання з сайту у проекті (рис. 4.13).

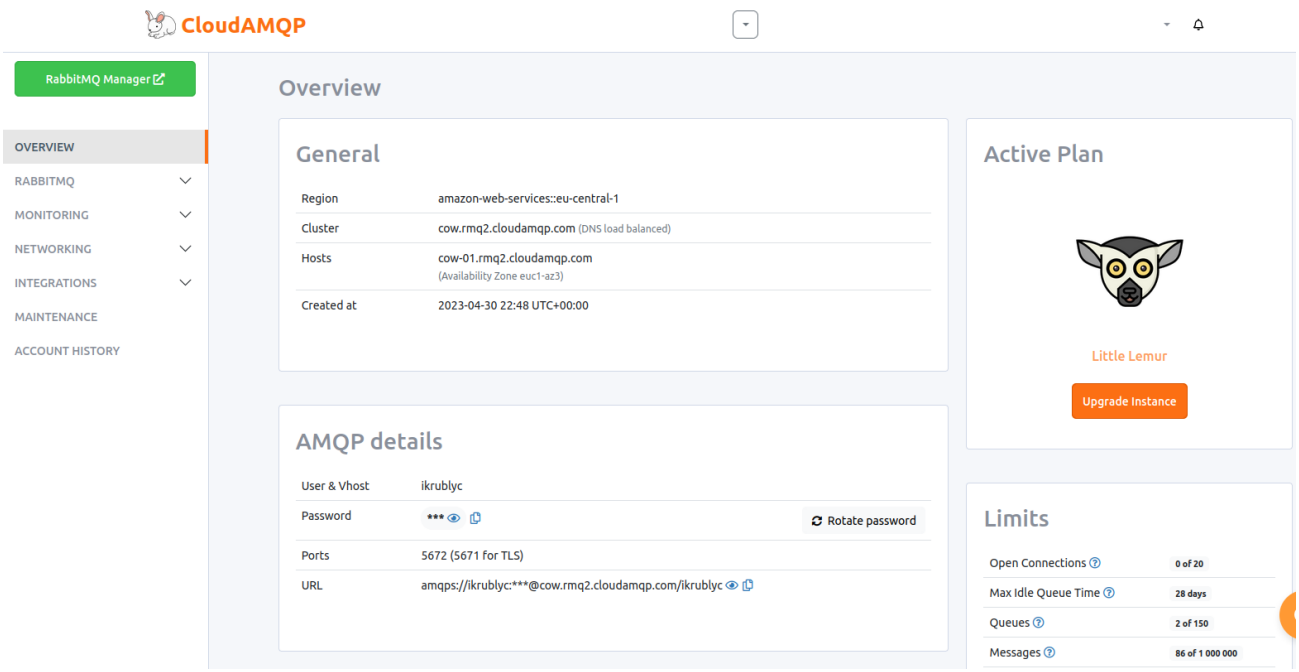

Рисунок 4.13 – Офіційний сайт CloudAMQP

З допомогою використаного посилання створено два файли які регламентують обробку повідомлень за допомогою RabbitMQ.

Файл виробник продуктів (producer.py) знаходиться на адміністраторській частині веб додатку (рис. 4.14) та файл споживач продуктів (consumer.py) знаходиться на головній частині сайту, яку зможуть бачити користувачі (рис. 4.15 та рис 4.16).

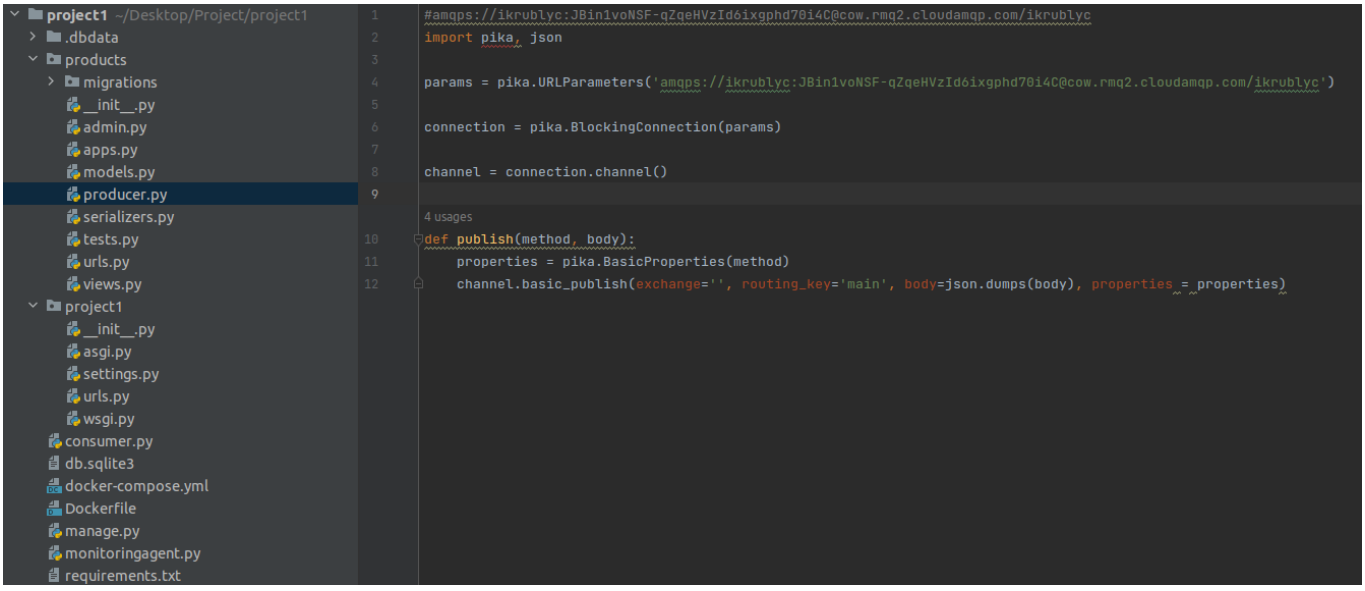

Рисунок 4.14 – Вміст файлу producer.py

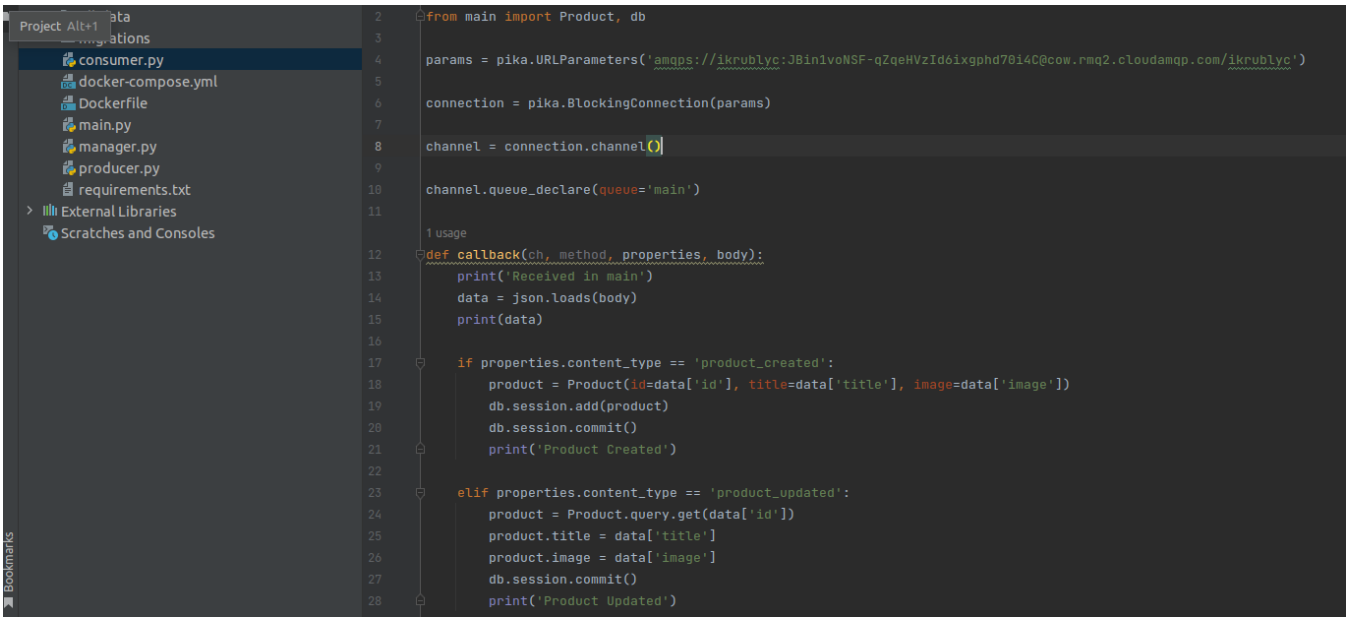

Рисунок 4.15 – Вміст файлу consumer.py

Рисунок 4.16 – Вміст файлу consumer.py

Головна ідея взаємодії двох частин додатку на мікросервісній архітектурі полягає у тому, що продукти на веб додатку створюються завдяки адміністраторській частині за допомогою Django та Docker, а головна частина веб додатку на Flask та Docker ловить подію (створення/видалення/оновлення нового продукту) за допомогою RabbitMQ і у наслідок відображає результати виконання даної події на головному сайті.

Після налаштування головної та адміністраторської частин веб додатку виконано перевірку взаємодії API за допомогою платформи Postman (рис 4.17).

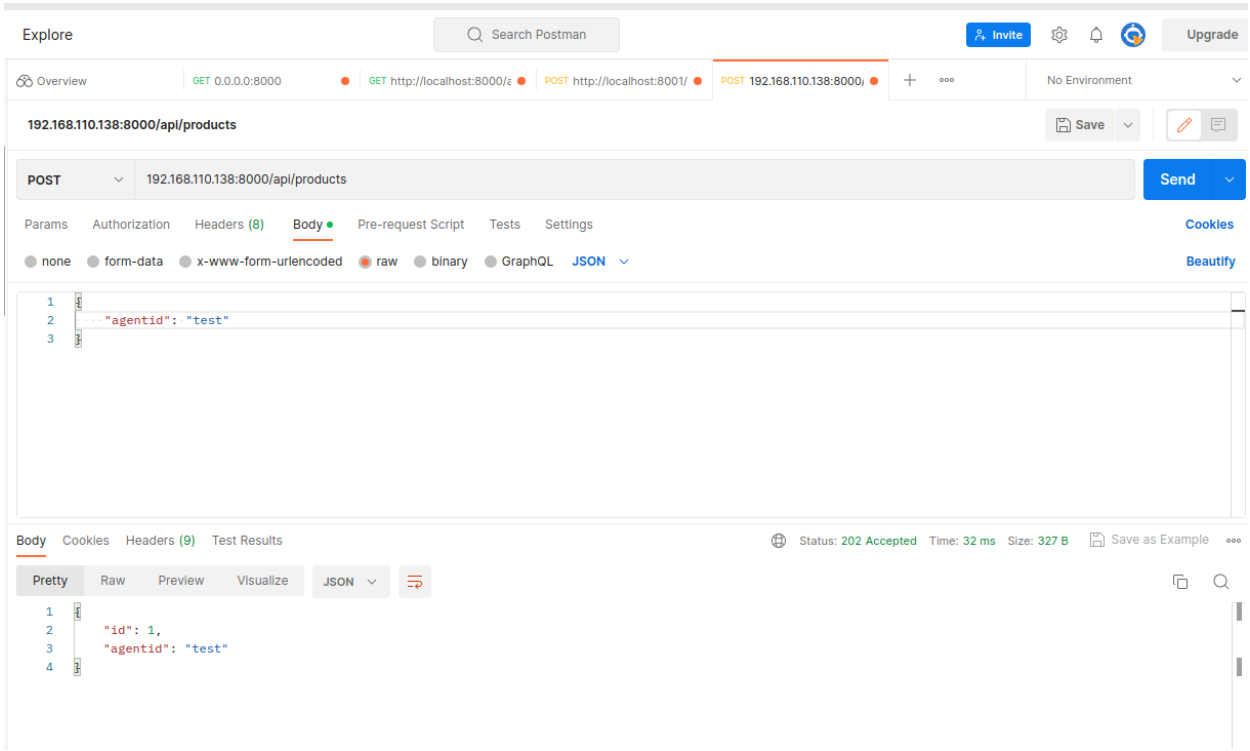

Рисунок 4.17 – Перевірка API взаємодій за допомогою платформи API Postman

У наслідок перевірки можна побачити успішну тестову відповідь (202 accepted) при використанні методу POST, отже можна зробити висновок що все працює.

4.2.2 Створення візуальної частини додатку на мікросервісах

Серверна частина веб продукту на мікросервісах працює як очікується, отже доцільним є робота над візуальною частиною додатку.

Для створення візуального користувацького інтерфейсу обрано React що являє собою JavaScript-бібліотеку з відкритим вихідним кодом, яка є зручною для розробки інтерфейсів користувача.

Для роботи з React створено проект (рис. 4.18) який став базою для формування візуальної частини продукту.

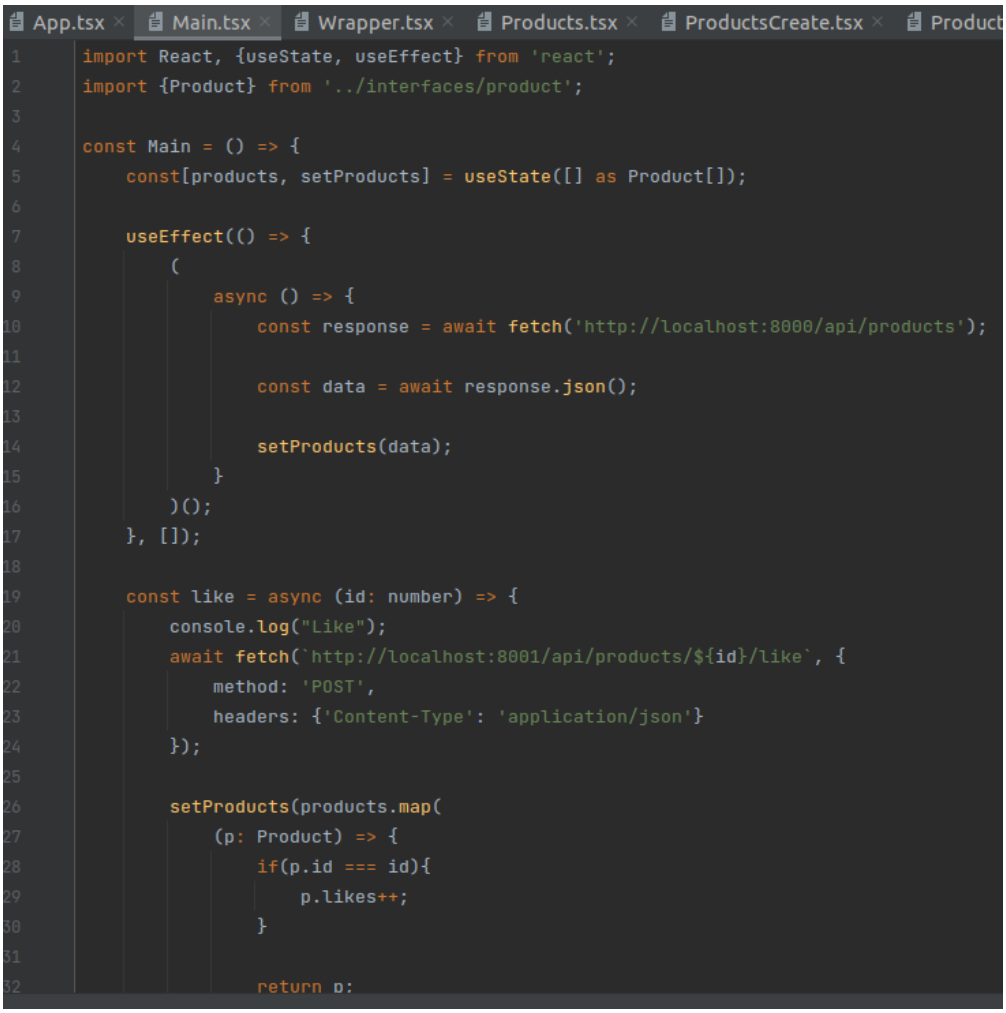

Рисунок 4.18 – Створення візуального інтерфейсу за допомогою React

Візуальна частина створена за допомогою JavaScript. Після завершення роботи над зовнішнім виглядом програми можна переходити до перевірки працездатності серверної та візуальної частини веб додатку на мікросервісах.

4.2.3 Перевірка працездатності веб продукту

Після закінчення розробки візуальної частини додатку на мікросервісах для того щоб перевірити роботу веб продукту, були запущені проекти серверної частини на Docker та було запущено проект що відповідає за реалізацію користувацького інтерфейсу.

На рисунку 4.19 можна побачити, що служби запущені успішно.

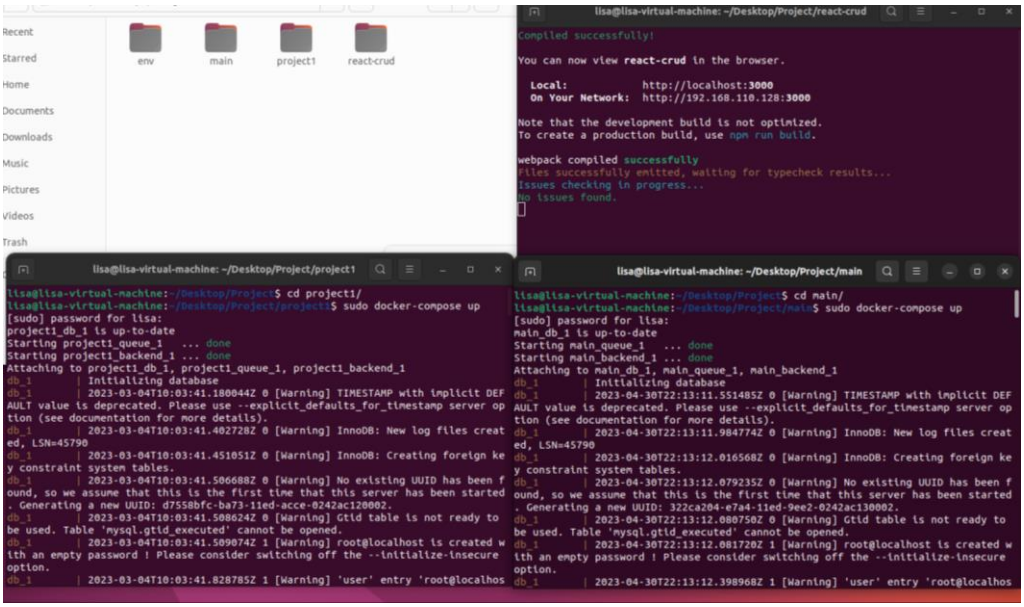

Рисунок 4.19 – Запуск та перевірка працездатності веб продукту

Наслідком запуску необхідних частин проекту можна побачити готовий веб продукт створений на мікросервісах. На рисунку 4.20 зображена адміністраторська частина сайту де можна створювати продукти, надавати їм опис, та при необхідності його коригувати або видаляти.

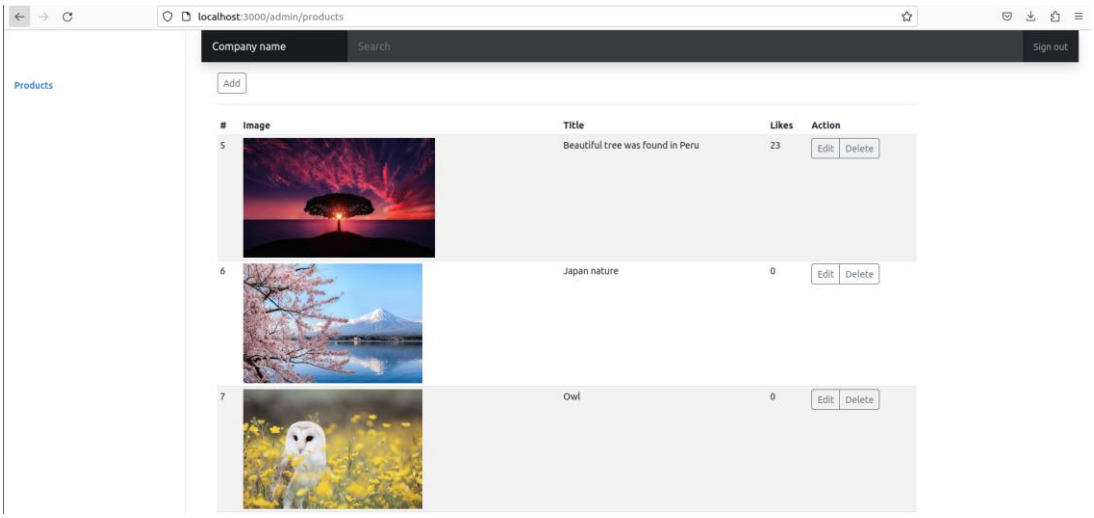

Рисунок 4.20 – Адміністраторська частина веб продукту

Для перевірки роботи було створено 6 фотокарток з описами на адміністраторській частині сайту. На рисунку 4.21 можна побачити що на сторінці,

яку бачать користувачі всі 6 фотокарток відображаються правильно і їм можна поставити лайки.

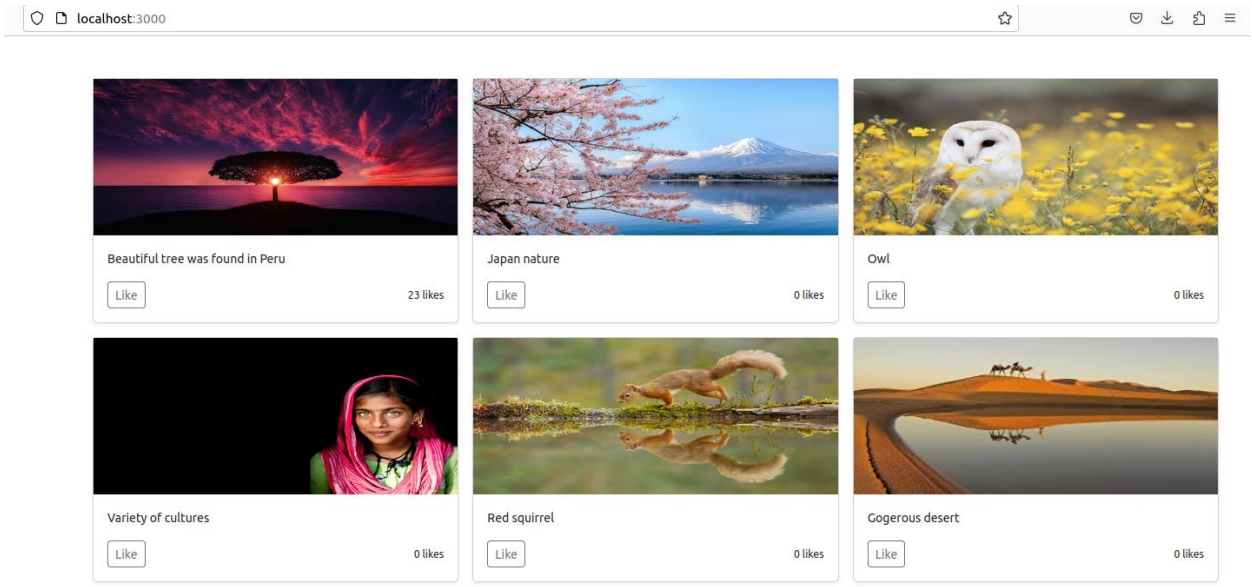

Рисунок 4.21 – Головна частина веб продукту

Перевірку роботи додатку на мікросервісах успішно завершено, отже можна сказати, що все працює належним чином. Тобто мікросервіси на Docker працюють і правильно взаємодіють.

4.3 Розробка системи моніторингу мікросервісів

Для розробки системи мікросервісів обрана друга віртуальна машина (VM2). Віртуальні машини знаходяться в одній мережі та пінгуються.

По аналогії до першої віртуальної машини, було підготовлене середовище розробки. Серед додаткових служб і сервісів встановлені Python, Pycharm та Docker.

Для того щоб створити систему моніторингу необхідно створити те, що буде відправляти дані та те, що буде їх приймати. Отже на серверній частині мікросервісного веб додатку треба створити агент з моніторингу, який буде збирати дані стану контейнеру і який буде відправляти ці дані безпосередньо на додаток з

моніторингу. Додаток з моніторингу у свою чергу вже буде розташований на іншій віртуальній машині буде отримувати інформацію та візуалізувати її.

4.3.1 Створення агенту з моніторингу.

На серверній частині мікросервісного додатку створено файл який збирає дані контейнеру у реальному часі та відправляє інформацію до програмного забезпечення з моніторингу за допомогою API.

На рисунку 4.22 можна побачити програмний код агенту з моніторингу, який є частиною як і адміністраторської частини так і головної частини проекту веб додатку на мікросервісах.

```
import requests
import os
import json
from datetime import datetime

import threading

2 usages
def startTimer():
  threading.Timer(5.0, startTimer).start()
  sendData()
  print("Data sent!")

def createTable():
    x = requests.post('http://192.168.110.138:8000/api/agent', data={'agentid':'test1'})

1 usage
def sendData():
    stream = os.popen('sudo docker stats --no-stream --format "{{ json . }}" main_db_1 project1_db_1')
    output = stream.read()
    result = json.loads(output)
    result['agentid'] = "test1"
    currentTime = datetime.now()
    stringTime = currentTime.strftime("%Y-%m-%d %H:%M:%S")
    result['datetime'] = stringTime
    x = requests.put('http://192.168.110.138:8000/api/agent', data=result)

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

Рисунок 4.22 – Вміст файлу monitoringagent.py

Основна ідея роботи агенту з моніторингу полягає в тому, що він відправляє інформацію про стан контейнеру з мікросервісами на другу віртуальну машину де знаходиться система моніторингу мікросервісів яка сприймає ці дані та відображає їх.

4.3.2 Створення додатку з моніторингу.

При написанні додатку з моніторингу на другій віртуальній машині створено новий проект, програма якого працює на Docker і отримує дані від додатку на мікросервісах.

Для того щоб отримати дані, треба визначити які саме дані контейнерів програма моніторингу буде обирати для спостереження. З ціллю визначення даної задачі у файлі models.py було створено класс даних які буде отримувати система моніторингу та описано їх максимальну довжину (рис. 4.23).

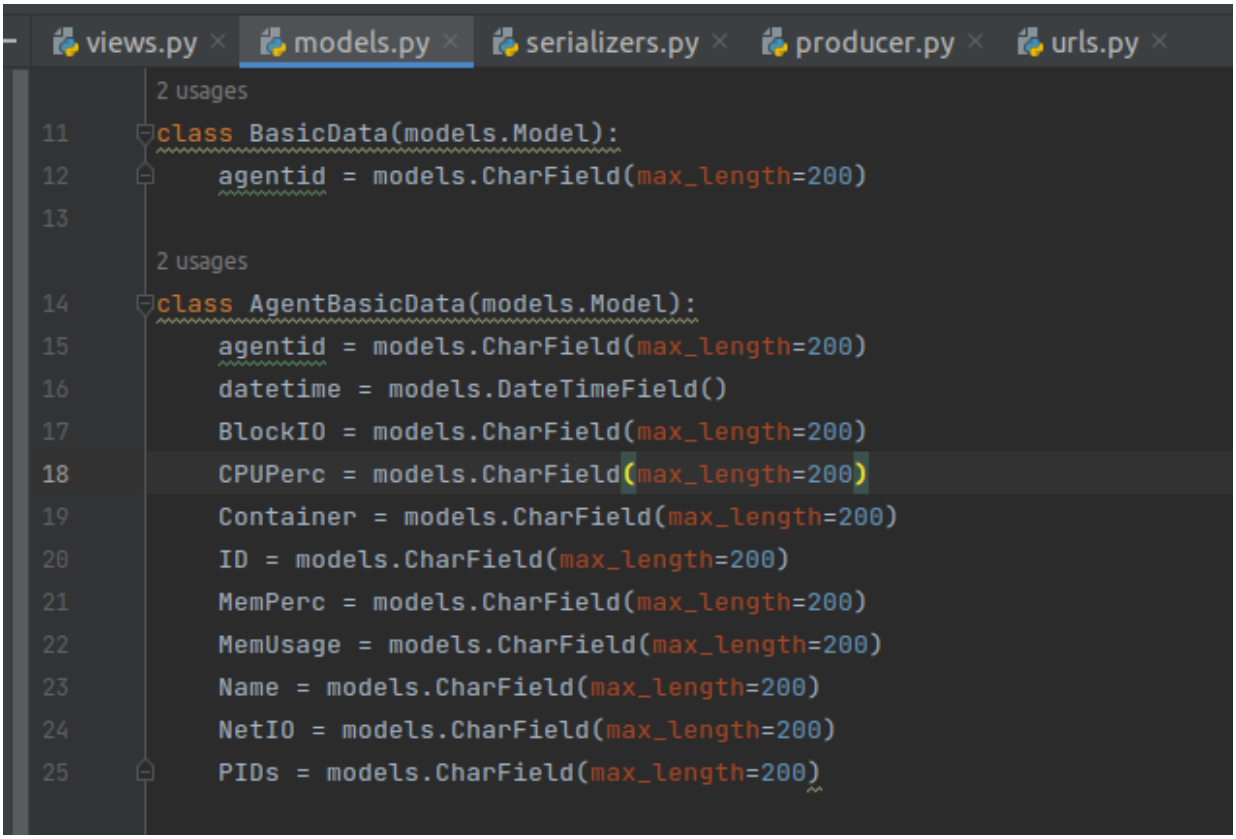

Рисунок 4.23 – Вміст файлу models.py

Після визначення інформації, яку можна отримати про стан контейнеру створено основне тіло програми з моніторингу.

До програми під’єднано базу даних де може зберігатися інформація щодо моніторингу, визначено таблиці всередині цієї бази даних (рис. 4.24 та рис.4.25).

views.py | models.py | serializers.py | producer.py | urls.py

```
from rest_framework import viewsets, status
from rest_framework.response import Response
from rest_framework.views import APIView
from .models import Product, User
from .serializers import ProductSerializer, BasicDataSerializer, AgentBasicDataSerializer
from.producer import publish
import random
from django.db import connection
from datetime import datetime

4 usages
class ProductViewSet(viewsets.ViewSet):
    def list(self, request):
        products = Product.objects.all()
        serializer = ProductSerializer(products, many=True)
        return Response(serializer.data)

    def create(self, request):
        serializer = BasicDataSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        print(serializer.data)
        print(serializer.data['agentid'])
        #serializer.save()
        cursor = connection.cursor()
        cursor.execute("CREATE TABLE " + serializer.data['agentid'] + "(timestamp DATETIME, "
                                                                      "BlockIO varchar(255),"
                                                                      "CPUPerc varchar(255),"
                                                                      "Container varchar(255),"
                                                                      "ID varchar(255),"
                                                                      "MemPerc varchar(255),"
                                                                      "MemUsage varchar(255),"
```

Рисунок 4.24 – Код основного тіла програмного забезпечення з моніторингу

Також описано процес запиту та отримання даних з агенту моніторингу (рис. 4.25 та рис.4.26).

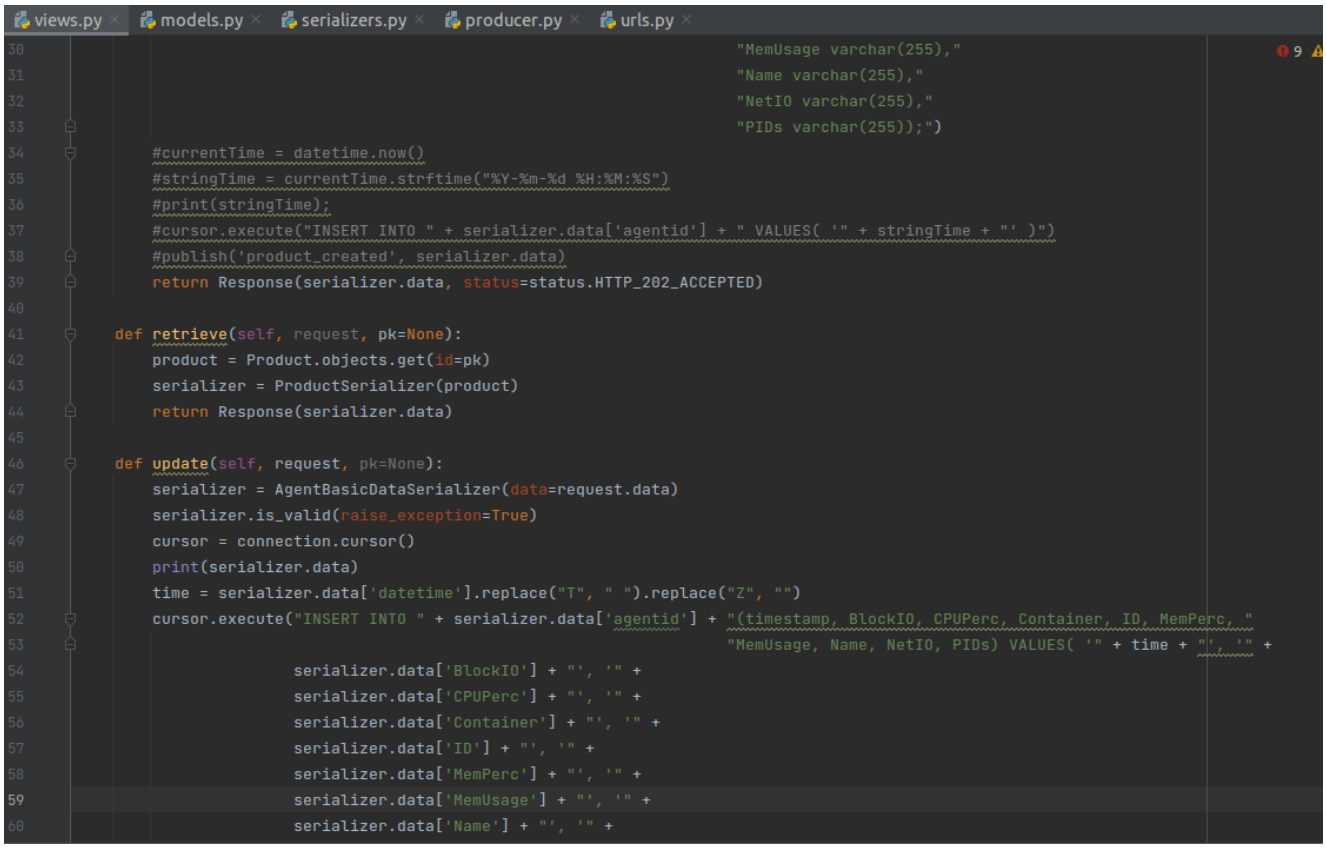

Рисунок 4.25 – Продовження коду програмного забезпечення з моніторингу

Рисунок 4.26 – Продовження коду програмного забезпечення з моніторингу

Всю необхідну інформацію для отримання даних з агенту додано, код програми написано, отже доцільним є перехід до перевірки його працездатності.

4.3.3 Перевірка працездатності системи моніторингу мікросервісів

Для перевірки спочатку були запущені програми на Docker за допомогою команди docker-compose up. Ця дія була виконана на двох віртуальних машинах для кожного проекту. Після цього додатки були успішно запущені.

Потім, за допомогою команди docker stats перевірено живий потік даних для контейнерів у стані роботи. Результат можна побачити на рисунку 4.27.

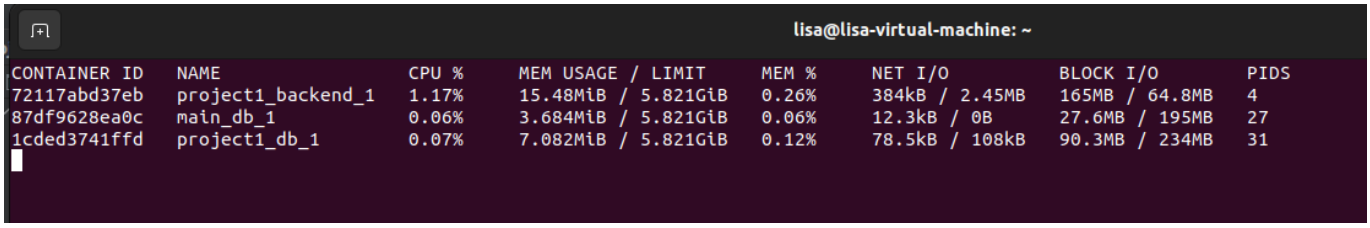

Рисунок 4.27 – Виведено інформацію про стан контейнерів Docker за допомогою команди docker stats

Наступним кроком перевірки є запуск агенту моніторингу на серверній частині веб додатку на мікросервісах, який знаходиться на першій віртуальній машині (рис. 4.28).

Результатом запуску агенту моніторингу можна побачити повідомлення про те що дані були відправлені, отже додаток з моніторингу на другій віртуальній машині (VM2) має змогу ці дані отримати.

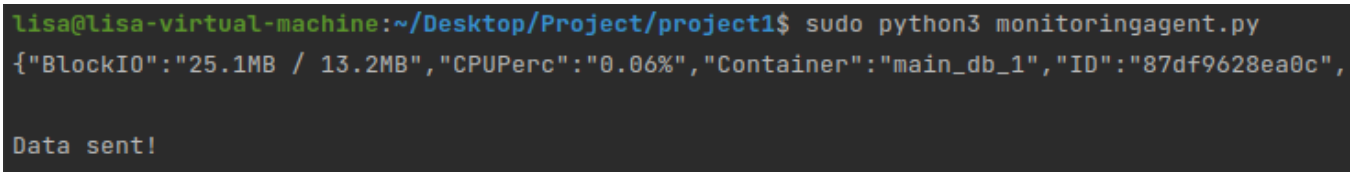

Рисунок 4.28 – Запуск агенту моніторингу

Щоб перевірити чи була отримана надіслана інформація запущено базу даних (рис. 4.29), яка знаходиться на другій віртуальній машині та взаємодіє безпосередньо з додатком моніторингу.

```
lisa@lisa-virtual-machine:~/Desktop/Project/project1$ sudo docker-compose exec db sh
# mysql -u root -p
```

Рисунок 4.29 – Виконано відкриття бази даних додатку з моніторингу

Інформація надсилається агентом моніторингу додатку кожні 5 секунд, після її отримання додаток з моніторингу робить запис у базу даних.

На рисунку 4.30 можна побачити результат запису стану контейнеру.

```
15 rows in set (0.00 sec)

mysql> select * from test1;
+---------------------+----------------+---------+-----------+--------------+---------+--------------------+-----------+-------------+------+
| timestamp           | BlockIO        | CPUPerc | Container | ID           | MemPerc | MemUsage           | Name      | NetIO       | PIDs |
+---------------------+----------------+---------+-----------+--------------+---------+--------------------+-----------+-------------+------+
| 2023-05-12 17:51:59 | 25.5MB / 195MB | 0.07%   | main_db_1 | 87df9628ea0c | 0.07%   | 4.004MiB / 5.821GiB | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:04 | 25.5MB / 195MB | 0.09%   | main_db_1 | 87df9628ea0c | 0.07%   | 4.004MiB / 5.821GiB | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:09 | 25.5MB / 195MB | 0.06%   | main_db_1 | 87df9628ea0c | 0.07%   | 4.004MiB / 5.821GiB | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:14 | 25.5MB / 195MB | 0.06%   | main_db_1 | 87df9628ea0c | 0.07%   | 4.004MiB / 5.821GiB | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:19 | 25.5MB / 195MB | 0.07%   | main_db_1 | 87df9628ea0c | 0.07%   | 4.004MiB / 5.821GiB | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:24 | 25.5MB / 195MB | 0.06%   | main_db_1 | 87df9628ea0c | 0.07%   | 4.004MiB / 5.821GiB | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:29 | 25.5MB / 195MB | 0.09%   | main_db_1 | 87df9628ea0c | 0.07%   | 4.004MiB / 5.821GiB | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:34 | 25.5MB / 195MB | 0.06%   | main_db_1 | 87df9628ea0c | 0.07%   | 4.004MiB / 5.821GiB | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:39 | 25.5MB / 195MB | 0.07%   | main_db_1 | 87df9628ea0c | 0.07%   | 4.004MiB / 5.821GiB | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:44 | 25.5MB / 195MB | 0.08%   | main_db_1 | 87df9628ea0c | 0.07%   | 4MiB / 5.821GiB     | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:49 | 25.5MB / 195MB | 0.07%   | main_db_1 | 87df9628ea0c | 0.07%   | 4MiB / 5.821GiB     | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:54 | 25.5MB / 195MB | 0.08%   | main_db_1 | 87df9628ea0c | 0.07%   | 4MiB / 5.821GiB     | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:52:59 | 25.5MB / 195MB | 0.06%   | main_db_1 | 87df9628ea0c | 0.07%   | 4MiB / 5.821GiB     | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:53:04 | 25.5MB / 195MB | 0.06%   | main_db_1 | 87df9628ea0c | 0.07%   | 4MiB / 5.821GiB     | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-12 17:53:09 | 25.5MB / 195MB | 0.08%   | main_db_1 | 87df9628ea0c | 0.07%   | 4MiB / 5.821GiB     | main_db_1 | 16.8kB / 0B | 27   |
| 2023-05-13 07:07:59 | 25.1MB / 13.2MB | 0.06%   | main_db_1 | 87df9628ea0c | 3.07%   | 183.2MiB / 5.821GiB | main_db_1 | 11.2kB / 0B | 27   |
| 2023-05-13 07:08:04 | 25.1MB / 13.2MB | 0.05%   | main_db_1 | 87df9628ea0c | 3.07%   | 183.2MiB / 5.821GiB | main_db_1 | 11.2kB / 0B | 27   |
| 2023-05-13 07:08:09 | 25.1MB / 13.2MB | 0.05%   | main_db_1 | 87df9628ea0c | 3.07%   | 183.2MiB / 5.821GiB | main_db_1 | 11.2kB / 0B | 27   |
+---------------------+----------------+---------+-----------+--------------+---------+--------------------+-----------+-------------+------+
18 rows in set (0.00 sec)
```

Рисунок 4.30 – Результат моніторингу записаний у базу даних

Оскільки інформація успішно записується та зберігається у базі даних, можна сказати, що програмне забезпечення з моніторингу мікросервісів працює як очікується.

Розробку програмного забезпечення завершено.

4.3.4 Перспективи розвитку додатку

Моніторинг саме мікросервісів дозволяє відобразити дані про роботу контейнерів у Docker, показати продуктиність їх роботи. Такий спосіб відрізняється від звичайного і поширеного способу моніторингу додатків тим, що він направлений на перевірку стану саме мікросервісів і він не потребує такої великої кількості ресурсів і даних, які потребують великі системи моніторингу.

Дана система моніторингу створена для перевірки продуктивності і якості роботи мікросервісів у невеликих додатках які не мають потреби у великих системах спостереження за всіма процесами програмного продукту.

Ця система може бути корисною для веб продуктів на мікросервісній архітектурі які розвиваються і не мають системи моніторингу.

Розроблена програма з моніторингу мікросервісних додатків може бути вдосконалена при потребі.

Вдосконалення можуть бути досягнуті за допомогою:

- створення додаткових ланок захисту, наприклад таких як шифрування та захист API;
- масштабування сервісів додатку за яким ведеться моніторинг (дана система моніторингу мікросервісів може буди поєднана з системою моніторингу додатку, а необхідні метрики візуалізовані);
- створення системи сповіщень, яка буде реагувати на збої роботи у контейнері та повідомляти власника веб продукту про наявні проблеми.

Дане програмне забезпечення має багато перспектив розвитку, а фокус уваги розробників програмного забезпечення може бути зосередженим як на захисті даних так і на розширенні функціоналу моніторингу.

Розроблений програмний продукт є актуальним і дуже корисним, адже він підвищує шанси знаходження проблем у середовищі розробки за допомогою якісного моніторингу стану контейнерів та їх внутрішніх процесів, що у наслідок впливає на швидкість знаходження проблем у системи та підвищує її захищеність завдяки

спостереженню за безперервністю та продуктивністю роботи програмних продуктів на мікросервісах та швидкому реагуванню на аномалії поведінки контейнерів.

Висновки за розділом 4

1. Сформовано проект програмного забезпечення. Пояснені основні принципи роботи.
2. Описано внутрішню структуру взаємодії веб додатку на мікросервісах та системи моніторингу.
3. Розроблено веб додаток на базі мікросервісної архітектури. Створено серверну частину веб додатку. Створено візуальну частину веб додатку. Перевірено взаємодію внутрішніх елементів веб продукту.
4. Розроблено систему моніторингу веб додатку на мікросервісах. Створено агент моніторингу на серверній частині веб додатку за яким ведеться моніторинг. Створено додаток з моніторингу який отримує і опрацьовує отримані дані. Інформацію про стан контейнеру у процесі моніторингу отримано.
5. Виділено перспективи розвитку програмного продукту.

ВИСНОВКИ

У сучасному світі мікросервісні технології активно захоплюють ринок інформаційних технологій. З поширенням розвитку інформаційних систем мікросервісні архітектури стають все більш популярними та часто використовуваними, збільшується не тільки їх розвиток і попит на них, а і стає більшим шанс виникнення різноманітних загроз що можуть вплинути на безпеку мікросервісних додатків.

Хоча мікросервісна архітектура має дуже багато сильних сторін які роблять розробку програмного забезпечення швидшою і простішою, та завдяки мікросервісній архітектурі підтримка великих і складних додатків стає кращою, а можливості гнучкого масштабування сервісів дозволяють оперативно реагувати на додаткові навантяження і покращувати продуктивність програмного забезпечення. Все таки мікросервісна архітектура - не є «панацеєю».

З частим використанням, збільшується не тільки розвиток попит на мікросервісні технології, а і стає більшим шанс виникнення різноманітних загроз що можуть вплинути на безпеку мікросервісної архітектури.

Оскільки загрози інформаційної безпеки постійно розвиваються, то захист мікросервісної архітектури є необхідним.

Основою продуктивного та якісного захисту додатку на мікросервісній архітектурі може стати впровадження системи безперервного моніторингу.

Моніторинг є важливою складовою системи безпеки програми. За допомогою цього процесу організації збирають і аналізують дані та визначають, чи досягла програма своїх цілей.

Постійний моніторинг надає можливість відстежувати середовище мікросервісів у режимі реального часу, щоб швидко виявляти інциденти безпеки та реагувати на них.

Моніторинг забезпечує кращу комунікацію команд між собою при розробці та огляді роботи, також моніторинг полегшує тестування коду завдяки забезпеченню

покращеної спостережності ресурсів. Отже завдяки безперервному моніторингу командами покращуються процеси прийняття рішень.

Безперервний моніторинг забезпечує кращу прозорість роботи додатку і підзвітність. Полегшує роботу завдяки графікам з показниками поточних даних додатку та журналам подій.

Завдяки моніторингу можна зрозуміти шляхи найефективнішого використання ресурсів. У наслідок підвищується продуктивність роботи додатку.

Для досягнення головної мети роботи були виконані такі завдання дослідження:

1. Проведено огляд мікросервісної архітектури. Виділено основні особливості роботи мікросервісів.
2. Проведено порівняння мікросервісної архітектури з монолітною. Проаналізовано переваги і недоліки мікросервісної архітектури.
3. Виявлено основні вразливості додатку на мікросервісній архітектурі.
4. Проаналізовано та систематизовано методи захисту мікросервісів.
5. Виділено основні принципи роботи безперервного моніторингу. Визначено вплив моніторингу на безпеку мікросервісів. Проведено огляд критичних ресурсів і процесів додатку корисних для системи моніторингу. Виділено вимоги щодо звітності для кращої роботи системи безперервного спостереження за роботою додатку на мікросервісах.
6. Визначено найкращі метрики та показники для моніторингу.
7. Сформовано проект програмного забезпечення. Розроблено веб додаток на базі мікросервісної архітектури. Розроблено систему моніторингу веб додатку на мікросервісах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Електронні джерела

1. Microservice architecture. [Електронний ресурс]. – Режим доступу: https://microservices.io/

2. What is Microservices Architecture? [Електронний ресурс]. – Режим доступу: https://cloud.google.com/learn/what-is-microservices-architecture

3. What are Microservices? How Microservices architecture works [Електронний ресурс]. – Режим доступу: https://middleware.io/blog/microservices-architecture/

4. What are microservices? How microservices work. The pros, cons. [Електронний ресурс]. – Режим доступу: https://raygun.com/blog/what-are-microservices/

5. Characteristics of Microservices. [Електронний ресурс]. – Режим доступу: https://aws.amazon.com/microservices/

6. How To Detect and Prevent Vulnerabilities in Microservices and Application Programming Interface. [Електронний ресурс]. – Режим доступу: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiRnYPqu8T9AhXpo4sKHThICfE4ChAWegQIChAB&url=https%3A%2F%2Fijrpr.com%2Fuploads%2FV3ISSUE11%2FIJRPR8061.pdf&usg=AOvVaw0UYLYjQvpjwZE0wHh4-2wB

7. Overcoming Microservices Architecture Risks. [Електронний ресурс]. – Режим доступу: https://www.kiuwan.com/blog/overcoming-microservices-architecture-risks/

8. Principle of least privilege. [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Principle_of_least_privilege

9. API and microservice security. [Електронний ресурс]. – Режим доступу: https://portswigger.net/burp/vulnerability-scanner/api-security-testing/guide-to-api-microservice-security

10. What is an SBOM. (Software Bill of Materials)? [Електронний ресурс]. – Режим доступу: https://www.aquasec.com/cloud-native-academy/supply-chain-security/sbom/

11. Security Challenges in Microservices. [Електронний ресурс]. – Режим доступу: https://www.styra.com/blog/security-challenges-in-microservices/

12. Best Practices. Microservices Monitoring: Challenges, Metrics, and Tips for Success. [Електронний ресурс]. – Режим доступу: https://lumigo.io/microservices-monitoring/

13. Microservice Monitoring Tools + Best Practices. [Електронний ресурс]. – Режим доступу: https://scoutapm.com/blog/microservice-monitoring

14. Tools for Monitoring Microservices. [Електронний ресурс]. – Режим доступу: https://www.decipherzone.com/blog-detail/top-best-tools-monitoring-microservices

15. Metrics to Monitor in Microservices. [Електронний ресурс]. – Режим доступу: https://lumigo.io/microservices-monitoring/

16. Monitoring Microservices. Performance monitoring. [Електронний ресурс]. – Режим доступу: https://smartbear.com/learn/performance-monitoring/monitoring-microservices/

17. Continuous monitoring. [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Continuous_monitoring

18. Continuous monitoring plan. [Електронний ресурс]. – Режим доступу: https://desktop.gov.au/blueprint/security/continuous-monitoring-plan.html

19. NIST. [Електронний ресурс]. – Режим доступу: https://www.nist.gov/

20. Мінімізація ризиків. [Електронний ресурс]. – Режим доступу: https://pidru4niki.com/86213/ekonomika/minimizatsiya_rizikiv

21. Docker. [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Docker_(software)

Статті в іноземних виданнях

22. Serhii Toliupa, Anna Mostovenko, Liza Hontkovska. Problematic aspects of solving issues of cyber influence on critical infrastructure objects. Scientific and Practical Cyber Security Journal (SPCSJ) № 3 (02) September 2023. (Грузія).

Тези наукових доповідей

23. Гонтковська Є. Методи захисту мікросервісної архітектури / Є. Гонтковська, С. Толюпа // "Проблеми кібербезпеки інформаційно-телекомунікаційних систем" (PCSITS)" 27 - 28 жовтня 2022, Київ, Україна.
24. Гонтковська Є. Безпека API та мікросервісів / Є. Гонтковська, С. Толюпа // "Новітні технологічні тенденції інтелектуальної індустрії та Інтернету речей" (TTSIIT). 24-25 січня 2023 р, Київ, Україна.
25. Гонтковська Є. Вплив моніторингу на захист мікросервісної архітектури / Є. Гонтковська, С. Толюпа // "Проблеми кібербезпеки інформаційно-телекомунікаційних систем" (PCSITS)" 27 квітня 2023, Київ, Україна.

ДОДАТОК А

Програмний код додатку з моніторингу:

```
from rest_framework import viewsets, status
from rest_framework.response import Response
from rest_framework.views import APIView
from .models import Product, User
from .serializers import ProductSerializer, BasicDataSerializer, AgentBasicDataSerializer
from.producer import publish
import random
from django.db import connection
from datetime import datetime

class ProductViewSet(viewsets.ViewSet):
    def list(self, request):
        products = Product.objects.all()
        serializer = ProductSerializer(products, many=True)
        return Response(serializer.data)

    def create(self, request):
        serializer = BasicDataSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        print(serializer.data)
        print(serializer.data['agentid'])
        #serializer.save()
        cursor = connection.cursor()
        cursor.execute("CREATE TABLE " + serializer.data['agentid'] + "(timestamp DATETIME, "
                                                "BlockIO varchar(255),"
                                                "CPUPerc varchar(255),"
                                                "Container varchar(255),"
                                                "ID varchar(255),"
                                                "MemPerc varchar(255),"
                                                "MemUsage varchar(255),"
                                                "Name varchar(255),"
                                                "NetIO varchar(255),"
                                                "PIDs varchar(255));")
        #currentTime = datetime.now()
        #stringTime = currentTime.strftime("%Y-%m-%d %H:%M:%S")
        #print(stringTime);
        #cursor.execute("INSERT INTO " + serializer.data['agentid'] + " VALUES( '" + stringTime + "' )")
        #publish('product_created', serializer.data)
```

```
            return Response(serializer.data, status=status.HTTP_202_ACCEPTED)

        def retrieve(self, request, pk=None):
            product = Product.objects.get(id=pk)
            serializer = ProductSerializer(product)
            return Response(serializer.data)

        def update(self, request, pk=None):
            serializer = AgentBasicDataSerializer(data=request.data)
            serializer.is_valid(raise_exception=True)
            cursor = connection.cursor()
            print(serializer.data)
            time = serializer.data['datetime'].replace("T", " ").replace("Z", "")
            cursor.execute("INSERT INTO " + serializer.data['agentid'] + "(timestamp, BlockIO, CPUPerc, Container, ID,
MemPerc, "
                                                "MemUsage, Name, NetIO, PIDs) VALUES( '" + time + "', '" +
                           serializer.data['BlockIO'] + "', '" +
                           serializer.data['CPUPerc'] + "', '" +
                           serializer.data['Container'] + "', '" +
                           serializer.data['ID'] + "', '" +
                           serializer.data['MemPerc'] + "', '" +
                           serializer.data['MemUsage'] + "', '" +
                           serializer.data['Name'] + "', '" +
                           serializer.data['NetIO'] + "', '" +
                           serializer.data['PIDs'] + "')")
            #serializer.save()
            #publish('product_updated', serializer.data)
            return Response(serializer.data, status=status.HTTP_202_ACCEPTED)

        def destroy(self, request, pk=None):
            product = Product.objects.get(id=pk)
            product.delete()
            publish('product_deleted', pk)
            return Response(status=status.HTTP_204_NO_CONTENT)

    class UserAPIView(APIView):
        def get(self, _):
            users = User.objects.all()
            user = random.choice(users)
            return Response({
                'id': user.id
})
```

ДОДАТОК Б

Програмний код агенту з моніторингу:

```
import requests
import os
import json
from datetime import datetime

import threading

def startTimer():
  threading.Timer(5.0, startTimer).start()
  sendData()
  print("Data sent!")

def createTable():
    x = requests.post('http://192.168.110.138:8000/api/agent', data={'agentid':'test1'})

def sendData():
    stream = os.popen('sudo docker stats --no-stream --format "{{ json . }}" main_db_1')
    output = stream.read()
    result = json.loads(output)
    result['agentid'] = "test1"
    currentTime = datetime.now()
    stringTime = currentTime.strftime("%Y-%m-%d %H:%M:%S")
    result['datetime'] = stringTime
    x = requests.put('http://192.168.110.138:8000/api/agent', data=result)

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