МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Факультет: **ННІ Каразінський банківський інститут**
Кафедра: **Інформаційних технологій та математичного моделювання**
Спеціальність: **122 Комп'ютерні науки**
Освітня програма: **Комп'ютерні науки та інформаційні технології в бізнесі**

Група: АК-21М **денна форма навчання**

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

«ДОСЛІДЖЕННЯ МОЖЛИВОСТІ ВИКОРИСТАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ В СИСТЕМАХ АВТОМАТИЗОВАНОГО ДОКУМЕНТООБІГУ»
ЗА НАКАЗОМ № 4601-5/3045 ВІД 25 ВЕРЕСНЯ 2024 РОКУ

здобувача вищої освіти **Рибалки Руслана Анатолійовича**

Робота допущена до захисту в ЕК
протокол кафедри ІТММ №____від_____2024 р.

Завідувач кафедри ІТММ
к. п. н., доцент
_____**Н. І. Стяглик**

Науковий керівник
д.е.н., к.т.н., професор
_____**Б. В. Самородов**

м. Харків 2024 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В. Н. Каразіна

Факультет навчально-науковий інститут “Каразінський банківський інститут”
Кафедра інформаційних технологій та математичного моделювання
Рівень вищої освіти другий (магістерський)
Спеціальність 122 Комп’ютерні науки
Освітня програма Комп’ютерні науки та інформаційні технології в бізнесі

ЗАТВЕРДЖУЮ

Завідувач кафедри

Н. І. Стяглик

“25” вересня 2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ (ПРОЕКТ)

Рибалка Руслан Анатолійович
(прізвище, ім’я, по батькові студента)

1. Тема роботи «Дослідження можливості використання мікросервісної архітектури в системах автоматизованого документообігу»

Керівник роботи д. е. н., професор Б. В. Самородов

затверджені наказом по університету від “25” вересня 2024 року №4601-5/3045

2. Строк подання студентом роботи 20 листопада 2024 р.

3. Перелік питань, які потрібно розробити:
У розділі 1: розглянути основи автоматизація документообігу в банківській системі та визначити переваги та недоліки мікросервісної архітектури.
У розділі 2: Провести вивчення концепції мікросервісів та аналіз архітектурної моделі мікросервісної системи документообігу.
У розділі 3: Дослідити можливості використання мікросервісів в системах автоматизованого документообігу.

4. План роботи

№ з/п	Назви етапів роботи
1	Вибір здобувачем теми кваліфікаційної магістерської роботи
2	Затвердження плану і завдання кваліфікаційної магістерської роботи
3	Здача кваліфікаційної магістерської роботи керівнику
4	Підпис кваліфікаційної магістерської роботи у керівника
5	Підпис кваліфікаційної магістерської роботи у нормо контролера
6	Допуск завідувачем кафедри до захисту кваліфікаційної магістерської роботи
7	Захист кваліфікаційної магістерської роботи

5. Дата видачі завдання 25 вересня 2024 року

Студент _____ Р. А. Рибалко
Підпис ініціали, прізвище

Керівник роботи _____ Б. В. Самородов
підпис ініціали, прізвище

РЕФЕРАТ
НА КВАЛІФІКАЦІЙНУ МАГІСТЕРСЬКУ РОБОТУ
«ДОСЛІДЖЕННЯ МОЖЛИВОСТІ ВИКОРИСТАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ В СИСТЕМАХ АВТОМАТИЗОВАНОГО ДОКУМЕНТООБІГУ»
Рибалка Руслан Анатолійович

Кваліфікаційна магістерська робота містить 83 сторінки, 1 таблиця, 19 рисунків, список літератури з 21 найменування.

Об’єктом дослідження є процес дослідження технологій та стандартів мікросервісної архітектури в системах автоматизованого документообігу**.**

Предметом дослідження є мікросервісна архитектура, як сучасний підхід до розробки складних систем автоматизованого документообігу.

Мета кваліфікаційної магістерської роботи полягає у дослідженні можливості використання мікросервісної архітектури в системах автоматизованого документообігу для подальшого практичного їх використання, визначення можливих викликів та обмежень мікросервісної архітектури та удосконалення систем автоматизованого документообігу.

Завданнями кваліфікаційної магістерської роботи є:

- у першому розділі розглянути основи автоматизації документообігу в банківській системі та визначити переваги та недоліки мікросервісного підходу для документообігу;
- у другому розділі провести вивчення концепції мікросервісів та аналіз архітектурної моделі мікросервісної системи документообігу;
- у третьому розділі дослідити можливості використання мікросервісів в системах автоматизованого документообігу.

Актуальність дослідження: У зв'язку з швидким зростанням обсягу електронної документації та переходом на хмарні рішення, необхідністю інтеграції з іншими системами та високими вимоги до продуктивності, стає важливим використання мікросервісної архітектури в автоматизованому документообігу.

За результатами дослідження: сформовано основні теоретичні аспекти використання мікросервісної архітектури в системах автоматизованого документообігу.

Практична новизна: запропоновано архітектуру та програмну реалізацію системи автоматизованого документообігу в банківській системі з використанням мікросервісів.

Одержані результати можуть бути використані: в системах автоматизованого документообігу банківського сектору.

КЛЮЧОВІ СЛОВА: МІКРОСЕРВІСНА АРХІТЕКТУРА, СИСТЕМИ АВТОМАТИЗОВАНОГО ДОКУМЕНТООБІГУ, АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, API GATEWAY, DOCER, KUBERNETES, REACT, KONG, NODE.JS, JAVASPRING, KAFKA, POSTGRESQL, MONGODB.

ABSTRACT
AT QUALIFICATION MAGISTER WORK

«RESEARCH OF THE POSSIBILITY OF USING MICROSERVICE ARCHITECTURE IN AUTOMATED DOCUMENTATION SYSTEMS»

Ruslan Rybalka

The qualifying master's thesis contains 83 pages, 1 table, 19 figures, a list of references with 21 names.

The object of research is the process of researching technologies and standards of microservice architecture in automated document management systems.

The subject of research is microservice architecture, as a modern approach to the development of complex automated document management systems.

The purpose of the qualifying master's thesis is to investigate the possibility of using microservice architecture in automated document flow systems for their further practical use, to identify possible challenges and limitations of microservice architecture, and to improve automated document flow systems.

The tasks of the qualifying master's thesis are:

- in the first section, consider the basics of document flow automation in the banking system and determine the advantages and disadvantages of the microservice approach for document flow;
- in the second section, study the concept of microservices and analyze the architectural model of the microservice document flow system;
- in the third section, explore the possibilities of using microservices in automated document management systems.

Relevance of the research: In connection with the rapid growth of the volume of electronic documentation and the transition to cloud solutions, the need for integration with other systems and high performance requirements, the use of microservice architecture in automated document flow becomes important.

According to the results of the research: the main theoretical aspects of the use of microservice architecture in automated document management systems have been formed.

Practical innovation: the architecture and software implementation of the automated document flow system in the banking system using microservices is proposed.

The obtained results can be used: in automated document flow systems of the banking sector.

KEYWORDS: MICROSERVICE ARCHITECTURE, AUTOMATED DOCUMENTATION SYSTEMS, SOFTWARE ARCHITECTURE, API GATEWAY, DOCER, KUBERNETES, REACT, KONG, NODE.JS, JAVASPRING, KAFKA, POSTGRESQL, MONGODB.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧОК, СИМВОЛІВ І ТЕРМІНІВ

API - Прикладний програмний інтерфейс;

CI/CD - Continuous Integration/Continuous Delivery;

CRM - Customer relationship management (управління відносинами з клієнтами);

ERP - enterprise resource planning system (планування ресурсів підприємства);

gRPC - Google Remote Procedure Call;

OCR - Optical Character Recognition (технології розпізнавання тексту);

PKI - Інфраструктура відкритих ключів;

REST - Representational State Transfer;

RPA - Robotic Process Automation (автоматизація повторюваних завдань);

RPC - Remote procedure call (виклик віддалених процедур).

ВСТУП

Системи електронного документообігу призначені для автоматизації процесів створення, обробки, зберігання, передачі та контролю документів у цифровій формі. Вони допомагають підприємствам та організаціям ефективно управляти документацією, забезпечують зручний доступ до даних, підвищують продуктивність роботи та знижують ризики втрати інформації. Використання мікросервісної архітектури в системах автоматизованого документообігу суттєво підвищує гнучкість, масштабованість і надійність таких систем. Переваги роблять мікросервісну архітектуру привабливим вибором для реалізації автоматизованої системи електронного документообігу. Крім того, мікросервісна архітектура сприяє спрощенню оновлення окремих сервісів без порушення роботи всієї системи, що особливо важливо для великих організацій з високими вимогами до доступності та стабільності.

Актуальність роботи. Сучасні банківські системи постійно стикаються з високими вимогами до обробки великих обсягів даних, зумовленими зростанням кількості транзакцій, взаємодій із клієнтами та змінами в регуляторних вимогах. Традиційні монолітні системи автоматизованого документообігу обмежені в можливостях масштабування, ускладнюють процеси оновлення та інтеграцію з новими технологіями. Мікросервісна архітектура дає змогу вирішити ці проблеми шляхом розподілу системи на незалежні компоненти, що полегшує їх розвиток, обслуговування та масштабування.

Метою роботи є дослідження можливостей використання мікросервісної архітектури для побудови ефективних систем автоматизованого документообігу в банківських установах. Дослідження охоплює вивчення архітектурних підходів, розгляд вимог до безпеки, продуктивності, надійності та масштабованості, а також визначення ключових переваг і потенційних ризиків впровадження мікросервісної

архітектури в банківській сфері.

Завдання дослідження:

– розглянути основи автоматизації документообігу в банківській системі та визначити переваги та недоліки мікросервісного підходу для документообігу;

– дослідити концепцію мікросервісів та провести аналіз архітектурної моделі мікросервісної системи документообігу;

– дослідити можливість використання мікросервісів в системах автоматизованого документообігу;

– запропонувати архітектуру та програмну реалізацію системи автоматизованого документообігу в банківській системі з використанням мікросервісів.

Об’єктом дослідження є процес дослідження технологій та стандартів мікросервісної архітектури в системах автоматизованого документообігу.

Предметом дослідження є мікросервісна архитектура, як сучасний підхід до розробки складних систем автоматизованого документообігу.

У вступі представлено актуальність роботи, сформульовано мету та відповідні завдання, об’єкт та предмет дослідження, наведено загальну структуру роботи.

У першому розділі «Основи архітектури системи автоматизованого документообігу» розглянуто основи автоматизації документообігу в банківській системі та визначено переваги та недоліки мікросервісного підходу для документообігу.

У другому розділі «Архітектурна модель мікросервісної системи документообігу» проведено дослідження концепції мікросервісів та аналіз архітектурної моделі мікросервісної системи документообігу.

У третьому розділі «Розробка системи автоматизованого документообігу з використанням мікросервісної архітектури» досліджено можливості використання мікросервісів в системах автоматизованого документообігу, запропоновано архітектуру та програмну реалізацію системи

автоматизованого документообігу в банківській системі з використанням мікросервісів. Одержані результати можуть бути використані в системах автоматизованого документообігу банківського сектору.

Висновок висвітлює інформацію щодо підсумків дослідження, його наукової та практичної значущості, можливі перспективи подальшого розвитку.

РОЗДІЛ 1
ОСНОВИ АРХІТЕКТУРИ СИСТЕМИ АВТОМАТИЗОВАНОГО ДОКУМЕНТООБІГУ

1.1. Автоматизація документообігу в банківській системі

У сучасному світі банки обробляють величезні обсяги документів: фінансові звіти, договори, платіжні доручення, кредити, депозити, звітність перед регуляторами, документи по взаємодії з клієнтами тощо. Обробка цих документів вручну потребує великих ресурсів, часу і створює ризики для точності даних та відповідності правовим вимогам.

Автоматизація документообігу — це процес впровадження інформаційних систем для створення, обробки, зберігання та передачі документів у цифровому форматі. В банківській системі це є особливо важливим через високий обсяг операцій, строгі регуляторні вимоги, необхідність точного обліку даних, а також зручність обслуговування клієнтів [1, 2].

Автоматизація документообігу дозволяє:

- прискорити обробку документів;
- знизити помилки, пов’язані з людським фактором;
- підвищити контроль та відповідність регуляторним вимогам;
- підвищити рівень безпеки та конфіденційності даних;
- покращити клієнтський досвід за рахунок швидшого та ефективнішого обслуговування.

Систем автоматизованого документообігу в банках виконує такі основні функції:

1. централізоване зберігання документів (зберігання всіх документів в електронному вигляді у централізованому сховищі, яке дозволяє легкий доступ до документів для співробітників банку);
2. автоматизований облік та класифікація документів (документи

автоматично класифікуються за типами (кредитні договори, фінансові звіти, юридичні документи тощо), що спрощує їх обробку та пошук. Кожен документ може мати індекс або унікальний ідентифікатор);

3. розмежування доступу до документів (налаштовування права доступу до документів для різних співробітників на основі їх ролей та обов'язків, що підвищує рівень безпеки та конфіденційності даних);

4. автоматизація банківських процесів (узгодження документів, підписання, затвердження, відправлення клієнтам або партнерам. Це значно пришвидшує документообіг та забезпечує прозорість процесів);

5. інтеграція з іншими системами (системи управління відносинами з клієнтами, планування ресурсів підприємства, платіжні системи, що дозволяє уникати дублювання інформації та автоматизувати обмін даними);

6. електронний підпис та аутентифікація (підтримка електронних підписів, що дозволяє офіційно підписувати документи в електронному вигляді, забезпечуючи їх юридичну силу. Системи включають механізми аутентифікації користувачів, що підвищує рівень захисту).

Автоматизація документообігу в банківській систем має ряд переваг. Вона знижує витрати на зберігання паперових документів, скорочує час, необхідний для обробки, і зменшує витрати на робочу силу, пов'язану з ручною обробкою документів. Завдяки автоматизації працівники банку можуть швидше обробляти документи, зосереджуючи свою увагу на більш важливих завданнях, замість ручної обробки рутинних операцій. Системи автоматизації забезпечують високий рівень захисту документів від несанкціонованого доступу або витоку інформації. Дані можуть бути захищені шифруванням, а доступ до них регулюється суворими правами доступу. Банківські установи підпадають під різноманітні регуляторні вимоги, такі як боротьба з відмиванням грошей та інші стандарти безпеки. Завдяки швидкій обробці документів, клієнти банку отримують швидші відповіді на свої запити, наприклад, оформлення кредитів або випуск нових карт. Це покращує загальне враження від взаємодії з банком. Автоматизація

також дозволяє уникнути помилок, які можуть виникнути через людський фактор, такі як неправильна обробка документів або дублювання інформації.

Незважаючи на переваги система автоматизованого документообігу стикається з проблемами. Впровадження сучасних систем автоматизованого документообігу вимагає значних початкових інвестицій, зокрема у програмне забезпечення, серверну інфраструктуру та навчання персоналу. Працівники можуть чинити спротив впровадженню нових технологій через необхідність навчання або звикання до нових інструментів. Важливо забезпечити відповідне навчання і підтримку, щоб мінімізувати цей ризик. Банки часто використовують різні системи управління даними, і інтеграція нових автоматизованих рішень з існуючими системами може бути складною та вимагати додаткових ресурсів і часу. Оскільки банківські документи містять таємну інформацію, забезпечення безпеки системи автоматизованого документообігу є критично важливим. Будь-які вразливості в системі можуть призвести до витоку даних або втрати конфіденційної інформації [2, 3].

Рисунок 1.1 – Переваги та недоліки автоматизованого документообігу

Система автоматизованого документообігу повинна відповідати певним вимогам, щоб забезпечити ефективне управління документами та підвищити продуктивність роботи.

1. Функціональні вимоги:

– зберігання електронних документів у різних форматах (doc, pdf, xls), з можливістю їх швидкого пошуку та перегляду;

– підтримування маршрутизації документів між користувачами, затвердження, узгодження та передача документів на підпис;

– можливість зберігати різні версії одного й того ж документа, з відстеженням змін та можливістю повернення до попередніх версій;

– забезпечення швидкого та зручного пошуку документів за метаданими (назва, автор, дата) або за вмістом документа (повнотекстовий пошук);

– розмежування прав доступу до документів на основі ролей користувачів (читання, редагування, видалення, затвердження);

– можливість інтеграції з іншими програмами (CRM, ERP, електронна пошта тощо) для забезпечення більш зручної та безперебійної роботи;

– підтримка електронного підпису для документів, що мають юридичну силу;

– автоматичне надсилання повідомлень, нагадувань та сповіщень про статус документів, завдань, що потребують дії (наприклад, затвердження або підпису).

2. Технічні вимоги:

– стійкість до збоїв і забезпечення збереження даних навіть у випадку неполадок;

– масштабованість з урахуванням зростання обсягу документів та кількості користувачів;

– швидке оброблення великих обсягів документів та метаданих,

мінімальний час на пошук та передачу даних;

– шифрування документів, захист від несанкціонованого доступу, журналювання дій користувачів;

– можливість створення резервних копій документів і швидке їх відновлення у випадку аварійних ситуацій.

Для автоматизації документообігу в банках використовують наступні технологічні рішення:

1. системи електронного документообігу – дозволяють створювати, зберігати, редагувати та обмінюватися документами в електронному вигляді;

2. цифровий підпис – забезпечує юридичну значимість електронних документів та підвищує безпеку транзакцій. Підпис підтверджує автентичність документа і захищає його від несанкціонованих змін;

3. OCR (Optical Character Recognition) – технології розпізнавання тексту, які дозволяють автоматично перетворювати скановані документи на редаговані електронні файли;

4. хмарні рішення – платформи для зберігання та управління документами. Вони знижують витрати на інфраструктуру та підвищує доступність документів;

5. RPA (Robotic Process Automation) – автоматизація повторюваних завдань, пов’язаних з обробкою документів, що знижує навантаження на працівників та мінімізує людські помилки;

6. Blockchain – для безпечного та прозорого зберігання і підтвердження автентичності документів. Технологія блокчейн гарантує незмінність даних і підвищує довіру до процесів документообігу;

7. CRM-системи – використовуються для зберігання та управління інформацією про клієнтів, включаючи документи. Інтеграція CRM з системами документообігу дозволяє ефективніше керувати клієнтськими документами та історіями взаємодії;

8. електронні архіви – рішення для довготривалого зберігання великих обсягів документів у цифровому форматі з можливістю швидкого пошуку.

1.2. Основи мікросервісної архітектури

Системи автоматизованого документообігу є головними компонентами банківської інфраструктури, що забезпечують ефективне управління документами, транзакціями та інформацією. Банківські системи завжди висувають високі вимоги до безпеки, надійності та продуктивності. Системи автоматизованого документообігу, що обробляють тисячі транзакцій щоденно, потребують швидкої адаптації до змін у законодавстві, нових вимог щодо безпеки та захисту даних, а також до інтеграції з іншими системами. Зі зростанням вимог до швидкості обробки, безпеки та гнучкості все більше уваги приділяється мікросервісной архітектурі. Цей підхід дозволяє розподілити великі, монолітні системи на незалежні, взаємодіючі мікросервіси, що значно покращують масштабованість та адаптивність [4, 5].

Мікросервісна архітектура — це підхід до розробки програмного забезпечення, коли система складається з набору невеликих, незалежних сервісів (мікросервісів), що взаємодіють через чітко визначені API. Кожен мікросервіс відповідає за певну функціональність, і його можна розгортати, масштабувати та оновлювати окремо від інших. Основними характеристиками мікросервісної архітектури є:

– незалежність сервісів (мікросервіси функціонують автономно, кожен мікросервіс є незалежним компонентом, що виконує конкретну бізнес-логіку, наприклад, обробку платежів, керування обліковими записами або генерацію звітів. Ця незалежність дозволяє розробникам працювати над різними частинами системи одночасно швидко та без ризику для всіх інших частин системи оновлювати або змінювати окремі сервіси);

– масштабованість (окремі мікросервіси можуть бути масштабовані незалежно один від одного. Якщо якийсь сервіс потребує більше ресурсів (наприклад, обробка транзакцій під час пікових навантажень), його можна масштабувати без необхідності розширення всіх інших сервісів);

– гнучкість (легке впровадження нових технологій та інструментів у

різних сервісах без потреби переписувати весь код. Оскільки мікросервіси є незалежними один від одного, для кожного з них можна використовувати різні технології або мови програмування, що відповідають їхнім специфічним вимогам. Наприклад, для аналітики можна використовувати Python, для обробки транзакцій — Java, а для зберігання даних — бази даних, які найкраще підходять для конкретних цілей);

– легкість модифікації (зміни в одному мікросервісі не впливають на інші. Оскільки кожен мікросервіс може бути розгорнутий окремо, це дозволяє випускати нові версії або вносити виправлення в один сервіс без перезавантаження всієї системи);

– чітке розмежування функціональності (кожен мікросервіс відповідає за конкретну задачу або набір задач, може мати свою власну базу даних або модель зберігання даних, що дозволяє уникнути централізації та підвищує продуктивність за рахунок паралельної обробки даних).

У проектуванні програмного забезпечення існує безліч підходів, але два найбільш поширених — це монолітна архітектура та мікросервісна архітектура (рис. 1.2).

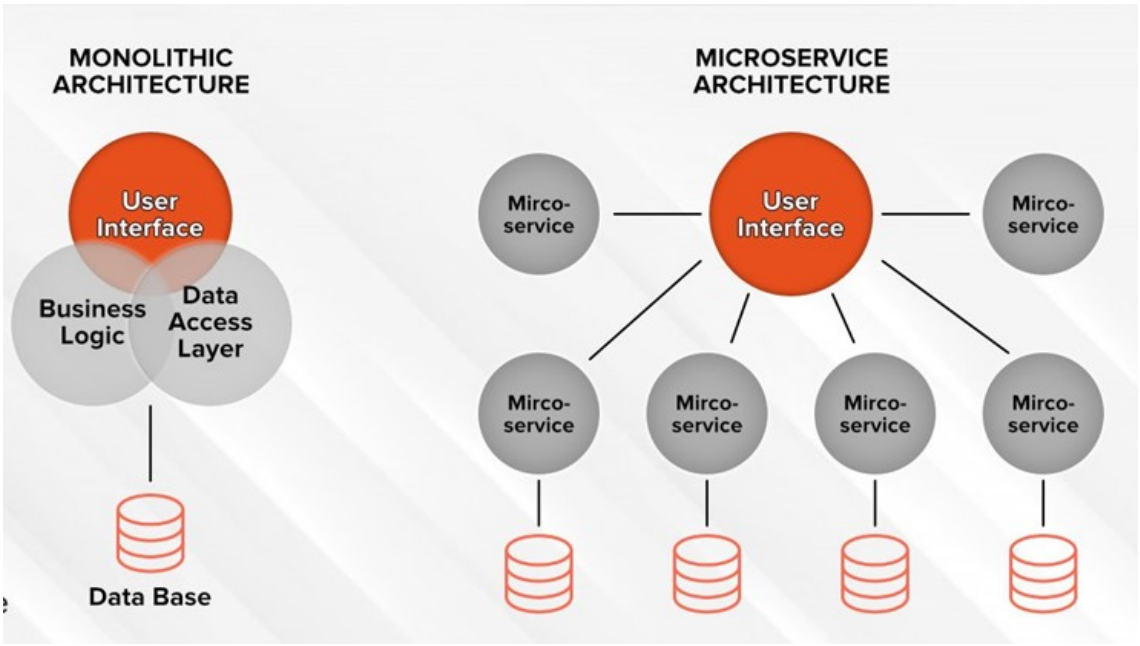

Рисунок 1.2 - Різниця між архітектурами

Монолітна архітектура — це підхід, при якому всі компоненти програми об'єднуються в єдине ціле. Уся функціональність програми реалізована в одній кодовій базі, і всі частини тісно пов'язані між собою [4-6].

Проведемо порівняння традиційної архітектури з мікросервісной (табл. 1.1).

Таблиця 1.1

Порівняння мікросервісного підходу з монолітним

Монолітна архітектура	Мікросервісна архітектура
Масштабованість	
Додаток функціонує як єдиний блок, що ускладнює його масштабування. Щоб підвищити продуктивність, необхідно масштабувати весь додаток, навіть якщо лише одна його частина потребує додаткових ресурсів.	Кожен компонент (сервіс) можна масштабувати незалежно від інших, що робить систему більш гнучкою та економічною.
Розгортанням та оновленнями	
Зміни в одному модулі вимагають перезавантаження та повторного розгортання всього додатка. Це збільшує ризик помилок і може призвести до простою системи.	Мікросервіси дозволяють розгортати та оновлювати окремі сервіси незалежно один від одного, мінімізуючи ризики збоїв та простоїв.
Гнучкість технологій	
Всі частини системи зазвичай написані на одній технології чи мовах програмування, що обмежує можливість використання нових інструментів або мов для вирішення конкретних завдань	Мікросервіси дозволяють використовувати різні технології для різних компонентів системи. Це надає більшу гнучкість при виборі інструментів, що найкраще підходять для кожної задачі.

Продовження табл. 1.1

Підтримка та розвиток	
З часом монолітна система може стати важкою для підтримки, оскільки збільшення коду та залежність ускладнює внесення змін, тестування та налагодження.	Завдяки розділенню на окремі сервіси, підтримка та розвиток стають простішими. Це дозволяє командам працювати автономно над окремими частинами системи.
Масштабування команд	
У великих монолітних системах важко розділити відповідальність між кількома командами, оскільки всі працюють з єдиною базою коду, що може призвести до конфліктів і уповільнення процесу розробки.	Кожна команда може відповідати за окремий мікросервіс, працювати незалежно і розгортати зміни автономно. Це сприяє масштабуванню команд і прискорює розробку.
Стійкість до помилок	
Якщо виникає помилка в одній частині програми, це може призвести до збою всієї системи. Немає чіткого розмежування між різними частинами системи.	Завдяки ізоляції мікросервісів, помилка в одному сервісі не впливає на інші. Це підвищує стійкість системи до помилок.
Труднощі з інноваціями	
Оновлення або впровадження нових функцій може бути довгим та ризикованим процесом, оскільки будь-яка зміна потребує ретельного тестування всього додатка.	Завдяки автономності кожного сервісу нові функції можна швидко впроваджувати в окремі компоненти, не ризикуючи стабільністю всієї системи.

Мікросервісний підхід відрізняється від традиційної монолітної архітектури, в якій усі компоненти системи з’єднані в один великий блок. У мікросервісах кожен компонент працює автономно і може бути розгорнутий,

масштабований та оновлений незалежно від інших [5, 6].

Вибір між монолітною та мікросервісною архітектурою залежить від потреб проекту, розміру команди, типу застосування та вимог до масштабованості. Монолітна архітектура може бути кращим вибором для малих проектів або стартапів, тоді як мікросервісна архітектура більше підходить для великих і складних систем, де потрібна гнучкість і масштабованість. На відміну від мікросервісної архітектури, традиційна (монолітна) архітектура має ряд проблем, які можуть ускладнювати масштабування та управління додатками [6, 7].

Таким чином, традиційна архітектура має проблеми, пов'язані з масштабованістю, гнучкістю, стійкістю до помилок та оновленістю. Мікросервісна архітектура вирішує більшість з цих проблем, забезпечуючи гнучкіший та надійніший підхід до розробки складних систем.

Мікросервісна архітектура набула популярності завдяки своїм численним перевагам, особливо в складних і великих системах. Основними прикладами її впровадження є такі галузі, як електронна комерція, фінанси, телекомунікації та банківські сервіси.

1.3. Переваги мікросервісної архітектури в автоматизованому документообігу

Масштабованість:

- динамічне масштабування окремих сервісів залежно від навантаження (наприклад, обробка великої кількості документів у години пік);
- використання хмарних рішень для масштабування.

Використання мікросервісів дозволяє масштабувати конкретні компоненти системи, наприклад, сервіс зберігання документів, залежно від навантаження. Оскільки кожен мікросервіс може бути масштабований незалежно, можна виділяти більше ресурсів для сервісів, що обробляють

найбільші обсяги запитів. Це дозволить банку краще управляти піковими навантаженнями, наприклад, під час обробки великої кількості платіжних документів.

Гнучкість та незалежність модулів:

- оновлення та модифікація окремих сервісів без зупинки системи;
- простота розробки та тестування нових функцій.

Банківська система постійно змінюється через нові вимоги регуляторів та інновації. Завдяки можливості оновлення та розгортання кожного мікросервісу окремо, компанії можуть швидше впроваджувати нові функції та реагувати на зміни ринку. Мікросервісна архітектура дозволяє легко впроваджувати нові технології та модифікації без необхідності зміни всієї системи, окремо розвивати кожен функціональний блок системи автоматизованого документообігу. Наприклад, сервіс керування документами оновлюється або замінюється без впливу на інші частини системи.

Різні функції системи автоматизованого документообігу (зберігання документів, обробка метаданих, робота з електронним підписом) можуть бути реалізовані як окремі сервіси. Це дає можливість командам розробників працювати над різними компонентами незалежно, що прискорює час розробки і випуску нових функцій.

Інтеграція з іншими системами:

- легка інтеграція з іншими сервісами та платформами через API;
- підтримка різних форматів документів та платформ.

Мікросервісна архітектура полегшує інтеграцію з іншими системами та сервісами, оскільки мікросервіси взаємодіють через чітко визначені API, що робить їх відкритими для зовнішніх взаємодій. Вони легко інтегруються з зовнішніми сервісами, такими як платіжні шлюзи, системи аутентифікації або сервіси сторонніх банків, що важливо для взаємодії системи автоматизованого документообігу з іншими корпоративними інформаційними системами. Мікросервісна архітектура також дозволяє

інтегрувати системи автоматизованого документообігу з CRM, ERP, системи електронного підпису тощо.

Кожен мікросервіс може бути написаний на різних мовах програмування і використовувати різні бази даних, технології і фреймворки, залежно від його потреб що дозволяє легко використовувати новітні технології для окремих частин системи. Це також знижує залежність від єдиної технологічної платформи. Наприклад, сервіс обробки зображень може бути написаний на Python з використанням спеціалізованих бібліотек для комп'ютерного зору, тоді як сервіс управління базами даних може працювати на SQL або NoSQL рішеннях.

Мікросервісна архітектура ідеально підходить для інтеграції з хмарними сервісами, що дозволяє банкам використовувати хмарні рішення для зберігання та обробки даних.

Стійкість до збоїв:

– розподілені сервіси з ізольованими контурами безпеки;
– відповідність сучасним вимогам щодо захисту даних.

У випадку збою одного з мікросервісів, інші продовжують функціонувати. Наприклад, збій у сервісі генерації звітів не вплине на роботу сервісів, що відповідають за авторизацію користувачів або збереження документів.

Висока надійність.

Якщо один мікросервіс виходить з ладу, це не призводить до зупинки всієї системи, інші мікросервіси можуть продовжувати працювати, що підвищує загальну надійність системи. Система в цілому залишається працездатною, навіть якщо один або кілька сервісів тимчасово недоступні. Мікросервіси можуть бути розміщені на різних серверах, що підвищує стійкість до відмов.

1.4. Можливі виклики та обмеження мікросервісної архітектури

Попри значних переваг, перехід на мікросервісну архітектуру вимагає ретельної підготовки та врахування певних ризиків [6, 7].

1. Складність управління

Кількість сервісів значно збільшується, що ускладнює їх моніторинг і адміністрування, особливо у великих банківських системах, де можуть бути сотні мікросервісів. Управління великою кількістю мікросервісів може бути складним завданням, оскільки потрібно стежити за їхньою взаємодією, версіями, та узгодженістю даних. Для цього необхідно впроваджувати спеціальні інструменти для управління мікросервісами, такі як Kubernetes або Docker.

2. Мережеві затримки

Оскільки мікросервіси взаємодіють через мережу, їх продуктивність залежить від стабільності мережевих з'єднань. Це може додавати затримки і збільшувати ризик виникнення проблем із продуктивністю. Потрібно забезпечити надійність та швидкість мережевої інфраструктури.

3. Розподілена обробка даних

Необхідність забезпечення узгодженості даних між сервісами, особливо у випадках, коли один документ обробляється кількома сервісами одночасно. Для цього можуть використовуватися шаблони, такі як SAGA або Event Sourcing.

4. Безпека

Банківські системи стикаються з жорсткими вимогами до безпеки, і мікросервісна архітектура, де багато сервісів взаємодіють через мережу, збільшує поверхні атаки. Кожен мікросервіс має власні точки входу, що збільшує кількість потенційної вразливості у системі. Тому важливо впроваджувати комплексні механізми безпеки, кожен мікросервіс повинен мати належний рівень безпеки, включаючи шифрування даних, захист від несанкціонованого доступу і забезпечення аутентифікації та авторизації для

кожного з них.

5. Можливі проблеми з консистентністю даних

Банківські системи потребують високого рівня консистентності даних, особливо у фінансових транзакціях. Оскільки мікросервіси можуть працювати з різними базами даних, важливо забезпечити узгодженість даних між ними, особливо у випадках транзакцій, які потребують синхронізації між кількома мікросервісами.

6. Складність тестування

Автономність мікросервісів ускладнює комплексне тестування системи. Необхідно тестувати як окремі мікросервіси, так і їх взаємодію.

Попри значну кількість недоліків, мікросервісний підхід залишається беззаперечним лідером для створення складних додатків, які мають на меті активний розвиток і розширення функціональних можливостей у майбутньому.

РОЗДІЛ 2
АРХІТЕКТУРНА МОДЕЛЬ МІКРОСЕРВІСНОЇ СИСТЕМИ ДОКУМЕНТООБІГУ

2.1. Концепції побудови мікросервісів

Концепції побудови мікросервісів охоплюють різні аспекти проектування, реалізації, розгортання та підтримки мікросервісних архітектур. Розглянемо основні концепції, які враховуються при розробці мікросервісів [8, 9].

1. Розділення на сервіси:

- мікросервіс є незалежним модулем, який реалізує одну конкретну функцію;
- сервіси повинні бути простими, з чітко визначеними межами відповідальності, що полегшує їх розробку, тестування та підтримку.

2. Технологічна автономність:

- кожен мікросервіс може бути реалізований за допомогою різних мов програмування та технологій, що дозволяє командам використовувати найкращі інструменти для конкретних задач;
- кожен сервіс має власну базу даних (переважно «Database per Service»), що знижує ризик узгодження схем між сервісами.

3. Комунікація між сервісами:

- синхронна комунікація (зазвичай REST або gRPC);
- асинхронна комунікація, для зменшення зв’язності між сервісами та покращення продуктивності використовуються асинхронні повідомлення через черги (наприклад, RabbitMQ, Apache Kafka).

4. Скалярність та розширюваність

- горизонтальне масштабування - додавання нових екземплярів сервісів за потреби без зміни коду;
- вертикальне масштабування - збільшення потужності одного

серверу або контейнера, не потребує змінювання коду або структуру системи;

- керування трафіком (API Gateway) дозволяє ефективно управляти навантаженням, маршрутизацією запитів і безпекою.

5. Надійність і стійкість

- Circuit Breaker - захист системи від каскадних збоїв, коли один сервіс виходить з ладу;
- відновлення після збоїв, включаючи повторні спроби та резервування.

6. Моніторинг і обслуговування

- логування для збору інформації про дії та помилки в системі;
- моніторинг для збору метрик продуктивності (наприклад, Prometheus, Grafana) допомагає виявляти проблеми до їхнього впливу на користувачів.

7. Автоматизація тестування та розгортання:

- CI/CD - впровадження безперервної інтеграції та безперервного розгортання (CI/CD) дозволяє автоматизувати процеси тестування, побудови та розгортання мікросервісів;
- контейнеризація - використання контейнерів (Docker) спрощує розгортання та управління залежностями, створюючи ізольовані середовища для кожного мікросервісу.

8. Безпека

- аутентифікація та авторизація - використання стандартів OAuth2 або OpenID Connect, для захисту доступу до мікросервісів, що дозволяє впроваджувати багаторівневу безпеку;
- забезпечення шифрування даних як у процесі передачі, так і при зберіганні.

Ці концепції допомагають створити ефективну, гнучку та масштабовану систему, яка може швидко реагувати на зміни бізнес-вимог та потреби користувачів.

2.2. Етапи розробки системи автоматизованого документообігу

Розробка системи автоматизованого документообігу охоплює кілька етапів, які забезпечують успішне впровадження, функціональність та підтримку системи. Процес розробки включає вимоги до архітектури, вибір технологій, побудову мікросервісів, розробку базових модулів та інтеграцію з іншими системами [9].

1) Проектування архітектури.

Мікросервісна архітектура системи автоматизованого документообігу складається з окремих сервісів (рис. 2.1), які відповідають за конкретні функції:

- сервіс документообігу (основний модуль для зберігання, передачі, обробки та пошуку документів);
- сервіс управління користувачами та доступом (модуль автентифікації та авторизації, який забезпечує захист даних і керування правами доступу);
- сервіс робочих процесів (відповідає за автоматизацію маршрутизації документів між користувачами, погодження та затвердження);
- сервіс звітності (генерація звітів про статус документів, дії користувачів та інші метрики);
- інтеграційні модулі (модулі для взаємодії з зовнішніми системами (електронна пошта, системи електронного підпису, хмарні сервіси зберігання тощо).

При проектуванні архітектури використовують патерни проектування, такі як API Gateway для централізованої обробки запитів або Event-Driven Architecture для асинхронної взаємодії між сервісами.

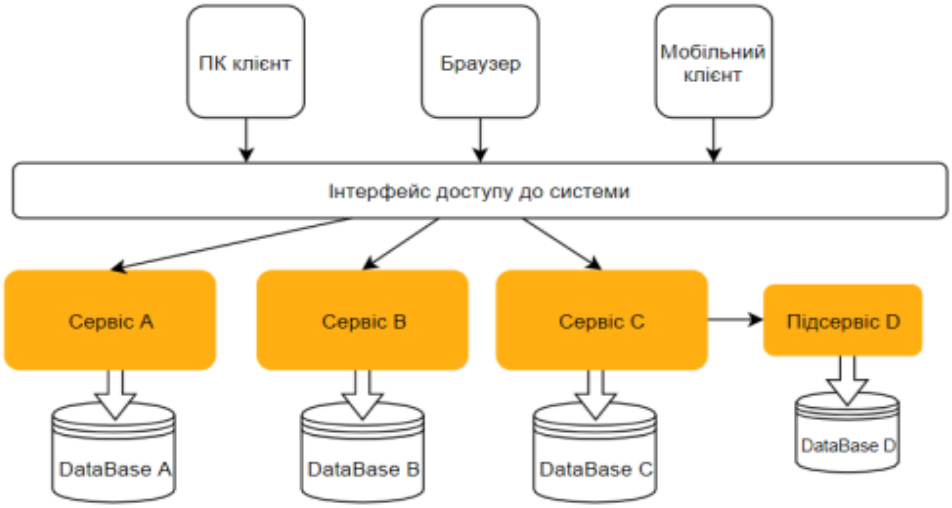

Рисунок 2.1 – Базова схема мікросервісної архітектури

2) Розробка мікросервісів.

Після завершення архітектурного проектування розпочинається розробка мікросервісів:

– кожен мікросервіс має бути автономним і виконувати лише одну функцію. Наприклад, сервіс пошуку може обробляти запити щодо пошуку за метаданими або повнотекстового пошуку в документах;

– API кожного мікросервісу має бути чітко визначене для забезпечення коректної взаємодії з іншими сервісами. Тут використовуються такі технології, як REST або GraphQL;

– захист API, усі сервіси повинні бути захищені за допомогою механізмів автентифікації (наприклад, OAuth 2.0) та шифрування переданих даних.

3) Розробка базових модулів.

Базові модулі є фундаментальними компонентами системи, і до їх розробки входять:

– модуль управління документами (забезпечує створення, редагування, затвердження та архівування документів. Важливо, щоб модуль підтримував різні формати документів (pdf, docx, xlsx);

– модуль версіонування документів (зберігає всі зміни, що вносяться в документи, та дозволяє користувачам повертатися до попередніх версій);

– модуль пошуку (інструменти індексації документів для швидкого пошуку, включно з метаданими (назва документа, автор, дата створення) і контекстним пошуком у вмісті документа);

– модуль електронного підпису (реалізація електронного підпису з використанням сертифікатів і ключів (наприклад, через PKI);

– модуль автоматизації робочих процесів (автоматичне переміщення документів між етапами погодження та затвердження. Використовуються правила для маршрутизації документів).

4) Інтеграція з іншими системами.

Інтеграція є важливою частиною розробки системи автоматизованого документообігу, оскільки вона дозволяє системі взаємодіяти з іншими бізнес-системами:

– інтеграція з ERP та CRM (для забезпечення безперервного обміну даними);

– використання API Gateway для управління взаємодією між клієнтами та мікросервісами системи;

– інтеграція з системами електронного підпису (використання зовнішніх сервісів для накладання підписів на документи);

– інтеграція з поштовими сервісами (відправка документів через електронну пошту або отримання нових документів через інтеграцію з SMTP/IMAP).

5) Забезпечення безпеки.

Безпека є одним із ключових аспектів системи автоматизованого документообігу, особливо при роботі з конфіденційними документами:

– шифрування даних під час передавання та зберігання документів (використання протоколів HTTPS та алгоритмів шифрування AES-256);

– розмежування прав доступу (забезпечення належного рівня контролю за доступом користувачів до різних документів і функцій системи,

включає механізми автентифікації та авторизації користувачів (наприклад, використання технологій, таких як OAuth 2.0, JWT для забезпечення захищеного доступу до ресурсів);

– фіксація всіх дій користувачів у системі для відстеження змін у документах і виявлення потенційних проблем безпеки.

6) Тестування

Тестування — це невід'ємна частина процесу розробки системи автоматизованого документообігу:

– модульне тестування (перевірка кожного окремого мікросервісу чи компонента на коректність виконання його функцій. Для цього використовуються інструменти автоматизованого тестування, такі як JUnit або pytest);

– інтеграційне тестування (тестування взаємодії між мікросервісами для перевірки сумісності та коректної передачі даних. Особлива увага приділяється тестуванню API та його інтеграції з іншими системами);

– End-to-End тестування (повне тестування системи від початку до кінця, включаючи всі функціональні можливості: створення, зберігання, пошук та обробка документів);

– тестування продуктивності (вимірювання здатності системи справлятися з великими обсягами документів та запитів, щоб забезпечити масштабованість).

7) Розгортання та підтримка

– контейнеризація (для забезпечення гнучкості використовується контейнеризація за допомогою Docker. Кожен мікросервіс запускається в окремому контейнері, що полегшує його розгортання і підтримку);

– оркестрація контейнерів (використання Kubernetes для автоматизації розгортання, управління та масштабування контейнерів. Kubernetes допомагає забезпечити безперебійну роботу та автоматичне відновлення мікросервісів у разі збоїв);

– моніторинг та логування (налаштування системи моніторингу для

відстеження стану кожного мікросервісу, збору логів та аналітики продуктивності (наприклад, Prometheus, Grafana, ELK Stack)).

8) Підтримка та масштабування

Після розгортання системи важливо забезпечити її підтримку та можливість подальшого масштабування:

- автоматичне масштабування (система повинна динамічно збільшувати ресурси при підвищеному навантаженні і знижувати їх при зменшенні активності);
- оновлення без простоїв (використання патернів розгортання, таких як Blue-Green Deployment або Canary Release, для оновлення системи без переривання її роботи);
- регулярні оновлення безпеки (забезпечення регулярного оновлення модулів безпеки та застосування патчів для уникнення вразливості);
- резервне копіювання та відновлення (налаштування автоматичних резервних копій документів та метаданих, а також процесів швидкого відновлення системи у разі збою).

Таким чином, розробка системи автоматизованого документообігу є комплексним процесом, який охоплює планування архітектури, розробку мікросервісів, тестування, інтеграцію з іншими системами та забезпечення надійного розгортання.

2.3. Види комунікації між мікросервісами

Кожен мікросервіс виконує свою окрему функцію і взаємодіє з іншими для обміну даними. Існує два основних способи комунікації між мікросервісами: синхронний (REST, RPC) та асинхронний обмін повідомленнями (ActiveMQ, RabbitMQ, Kafka).

Підхід, коли один мікросервіс надсилає запит іншому через HTTP або інший протокол і очікує на відповідь називають синхронним, оскільки мікросервіс, що викликає запит, залишається заблокованим, поки не отримає

відповідь.

Проаналізуємо переваги та недоліки синхронного підходу, використовуючи, як приклад архітектурний стиль REST (рис. 2.2). Він базується на принципах взаємодії між клієнтом і сервером через протокол HTTP. Одним із основних принципів архітектури REST є те, що кожен елемент системи представлений як ресурс. Це означає, що кожна сутність, з якою працює система, будь то дані або об'єкти, представлена як ресурс. Кожен ресурс (наприклад, користувач, продукт або сторінка) має унікальний ідентифікатор (URI), що дозволяє доступ до нього через HTTP-запити.

Ресурсами виступають сутності, відомі сервісу, такі як пристрої або дані вимірювань. Під час обробки запиту сервер може створювати представлення ресурсу, зручне для клієнта. Кожен запит клієнта до сервера є незалежним, що означає, що сервер не зберігає інформацію про попередні запити. Уся необхідна інформація, включаючи ідентифікацію клієнта або сесію, передається в кожному запиті. REST також підтримує кешування відповідей, що може суттєво покращити продуктивність системи. Якщо відповідь на запит не змінюється часто, її можна кешувати для швидшого доступу. Найпоширенішими форматами для передачі даних є JSON і XML.

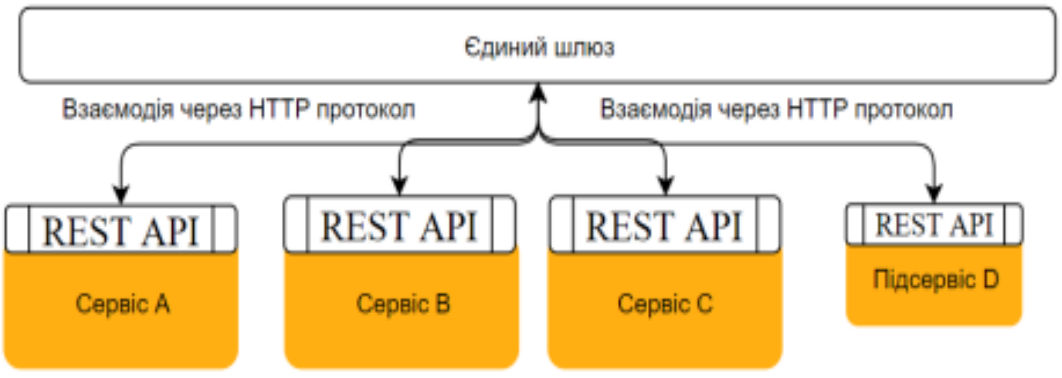

Рисунок 2.2 – Схема комунікації сервісів у REST-підході

Протокол HTTP є основою архітектури REST, оскільки надає стандартні методи, які відповідають її принципам: GET — для отримання даних, POST — для створення нового ресурсу, PUT — для оновлення наявного ресурсу, а DELETE — для його видалення.

REST є популярним підходом для створення API в сучасних веб-додатках завдяки своїй простоті та ефективності, що дозволяє легко інтегруватися з різними системами та сервісами.

Однією з основних переваг синхронного підходу є те, що серверна частина не залежить від типу клієнта. Використовуючи один і той самий інтерфейс, з сервісом можуть взаємодіяти різні клієнти: веб-додатки, мобільні додатки, датчики для роботи з фізичними об'єктами, а також інші мікросервіси в межах системи. Однак до недоліків можна віднести збільшення затримок через очікування відповіді. Крім того, якщо мікросервіс, до якого надсилається запит, недоступний, весь процес може бути заблокований.

Другий підхід до комунікації в розподілених системах — це асинхронний обмін повідомленнями. У цьому методі відправник і отримувач не взаємодіють одночасно. Замість цього повідомлення передаються через проміжні елементи, такі як черги або брокери повідомлень, що дозволяє кожному сервісу працювати незалежно. Мікросервіси можуть надсилати повідомлення в чергу або шину подій без очікування негайної відповіді, а отримувач обробляє їх у зручний для нього час. Це знижує залежність від доступності інших сервісів у реальному часі. Основними компонентами цього підходу є можливість мікросервісів генерувати події та механізми для їхнього визначення іншими сервісами. Брокери повідомлень, такі як RabbitMQ, Kafka або ActiveMQ, зазвичай допомагають вирішувати обидва ці завдання.

Брокери повідомлень виступають посередниками між генераторами (producers) та споживачами (consumers), забезпечуючи надійну доставку й зберігання даних. Генератори надають програмний інтерфейс для публікації

подій або повідомлень у чергу, яка є окремим сервером для приймання, зберігання й передачі повідомлень споживачам. Споживачі отримують повідомлення про певні події, які були створені генераторами. До черги можуть надходити повідомлення від багатьох генераторів, і також багато споживачів можуть отримувати одне й те саме повідомлення. Це дозволяє реалізувати схему «один до багатьох» в обміні повідомленнями. Наприклад, подія про створення нового користувача може бути одночасно отримана сервісами пристроїв та вимірювань, але розглядається як одна подія в системі.

ActiveMQ — це популярний відкритий брокер повідомлень, розроблений компанією Apache. Він реалізує специфікацію Java Message Service (JMS) і підтримує асинхронну комунікацію між мікросервісами. ActiveMQ забезпечує простий і зручний API для публікації та підписки на повідомлення, підтримує кілька протоколів для обміну повідомленнями, таких як AMQP, STOMP, MQTT і OpenWire. ActiveMQ (рис. 2.3) дозволяє організовувати обробку повідомлень як в режимі черг (queue), так і в режимі публікацій/підписок (topic). Брокер підтримує надійну доставку повідомлень, зокрема механізми підтвердження та повторної доставки. ActiveMQ можна масштабувати, використовуючи кластеризацію. Має зручний веб-інтерфейс для моніторингу та управління брокером.

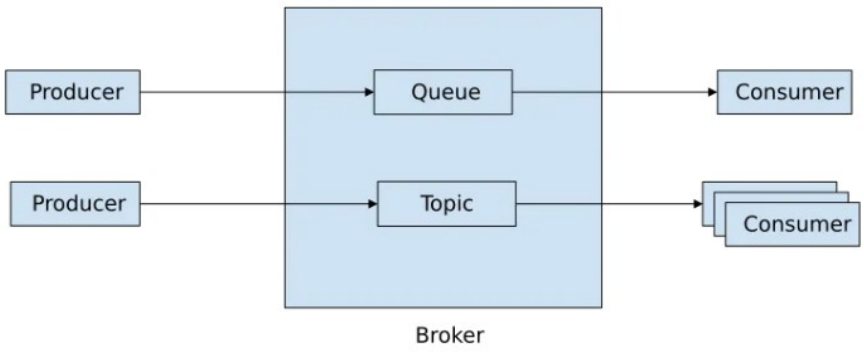

Рисунок 2.3 – Схема роботи брокеру повідомлень ActiveMQ

ActiveMQ підходить для сценаріїв, де важлива простота використання, підтримка JMS і інтеграція з різними протоколами. Він ідеальний для традиційних корпоративних додатків, де потрібна надійна доставка повідомлень

RabbitMQ — це універсальний брокер повідомлень, який підтримує протоколи MQTT, AMQP та STOMP. Він підходить для високопродуктивних сценаріїв, таких як обробка онлайн-платежів, виконання фонових завдань та обмін повідомленнями між мікросервісами. RabbitMQ забезпечує високу доступність та стійкість до відмов. RabbitMQ (рис. 2.4) дозволяє реалізовувати складні механізми маршрутизації повідомлень через обміни (exchanges) та черги (queues), забезпечує механізми підтвердження отримання повідомлень, зберігання повідомлень на диску та повторну доставку в разі збою. RabbitMQ має клієнтські бібліотеки для багатьох мов, таких як Java, Python, Ruby, .NET, PHP та інші

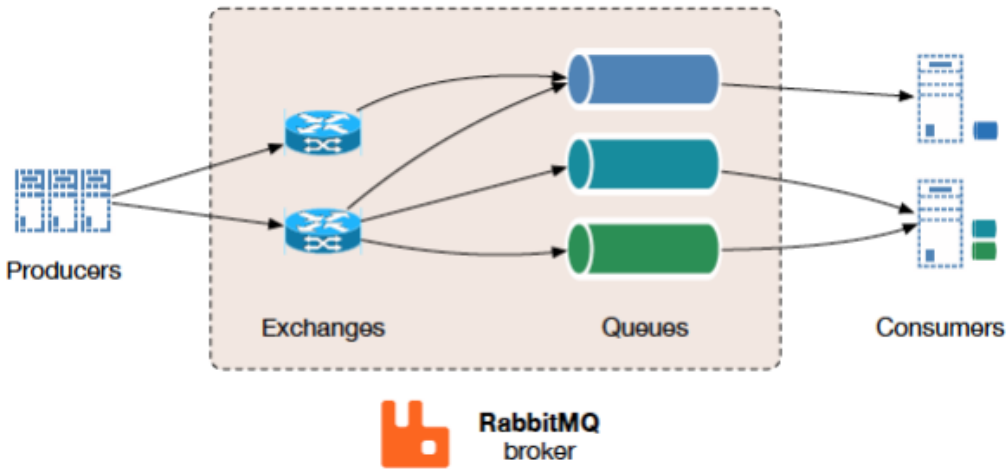

Рисунок 2.4 – Схема роботи брокеру повідомлень RabbitMQ

Суть роботи RabbitMQ полягає в забезпеченні ефективної передачі повідомлень між різними компонентами програмного забезпечення через систему черг і обмінів (exchanges). Відправники (producers) надсилають повідомлення до обмінів, а споживачі (consumer) отримують ці повідомлення

з черг. RabbitMQ підтримує модель публікації/підписки, що дозволяє багатьом споживачам підписуватися на одне й те саме повідомлення. Повідомлення, що надходять до обмінів, зберігаються у чергах до тих пір, поки їх не заберуть споживачі.

Черга в RabbitMQ обмежена пам'яттю хоста та доступним дисковим простором, виступаючи в ролі великого буфера для повідомлень. У цій системі багато відправників (producers) можуть надсилати повідомлення в одну чергу, а багато споживачів (consumers) можуть намагатися отримувати дані з цієї черги. Споживач — це програма, яка чекає на отримання повідомлень з черги та обробляє їх відповідно до визначеної логіки.

Таке налаштування дозволяє реалізувати асинхронну обробку, де виробники можуть надсилати повідомлення без затримок, навіть якщо споживачі ще не готові їх обробити. Якщо черга заповнена, RabbitMQ може застосовувати різні механізми управління потоком, щоб уникнути перевантаження системи, включаючи обмеження на кількість збережених повідомлень або механізми повідомлення виробників про перевантаження.

RabbitMQ підходить для простих сценаріїв використання черг повідомлень, особливо при невеликому обсязі трафіку даних. Він має певні переваги, такі як підтримка пріоритетних черг і гнучкі параметри маршрутизації, що дозволяє ефективно управляти обробкою повідомлень.

Однак для роботи з великими обсягами даних і високою пропускною спроможністю краще використовувати Apache Kafka. Kafka забезпечує кращу продуктивність у сценаріях з масивними даними, ідеально підходить для потокової обробки даних та сценаріїв, пов'язаних з аналітикою оскільки спроектований для обробки величезної кількості повідомлень з високою швидкістю. Kafka забезпечує високу пропускну здатність (мільйони повідомлень на секунду), підтримує потокове оброблення, резервне копіювання черг і гарантує високу доступність. Крім того, для Kafka важливий порядок доставки повідомлень.

Apache Kafka — це розподілений брокер повідомлень, призначений для

обробки великих обсягів даних у режимі реального часу [10].

Kafka забезпечує високу продуктивність при обробці великих обсягів повідомлень завдяки своїй архітектурі, яка базується на поділах (partitions) та розподілі даних. Вона використовує механізм реплікації, де кожен топік має свої копії (репліки), що дозволяє уникнути втрати даних. Це забезпечує доступність системи навіть у разі збою одного або кількох вузлів.

У Kafka повідомлення зберігаються на дисках у вигляді журналів, що дозволяє зберігати дані тривалий час і забезпечує можливість їх повторного оброблення. Kafka має горизонтальну масштабованість, що дозволяє додавати нові вузли для підвищення потужності системи без значних витрат. Крім того, вона підтримує інтеграцію з системами для обробки потоків, такими як Apache Flink і Apache Spark, що дає змогу обробляти дані в реальному часі. Kafka створює теми (topics), до яких можуть підписуватися споживачі, що забезпечує потужний механізм обміну повідомленнями в рамках системи.

Принцип роботи системи Apache Kafka (рис. 2.5) полягає в ефективному обміні та зберіганні повідомлень між різними компонентами системи через архітектуру, що складається з генераторів (producers), споживачів (consumers), брокерів (brokers) та топіків (topics).

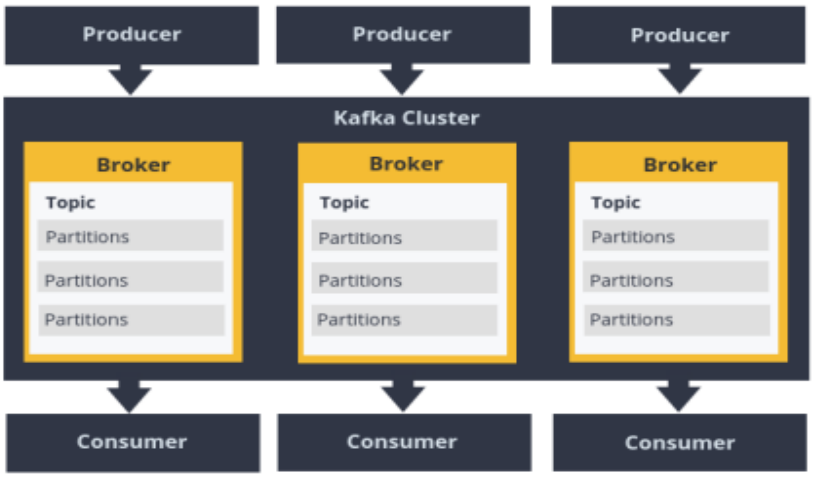

Рисунок 2.5 – Схема роботи шини повідомлень Kafka

Користувачі або адміністратори системи створюють топіки, які є логічними структурами для зберігання повідомлень. Кожен топік складається з одного або кількох поділів (partitions), що дозволяє розподілити навантаження. Виробники (producers) надсилають повідомлення до конкретного топіка. Кожне повідомлення може містити дані у різних форматах (наприклад, JSON, XML) і має унікальний ідентифікатор. Повідомлення зберігаються на брокерах у вигляді журналу. Це дозволяє зберігати дані тривалий час і забезпечує можливість повторного зчитування. Для забезпечення надійності дані в Kafka реплікуються на кілька брокерів. Споживачі (consumers) підписуються на один або кілька топіків і отримують повідомлення, щойно вони стають доступними на брокерах. Споживачі можуть обробляти дані в реальному часі або зчитувати їх у міру необхідності.

2.4. Роль API Gateway у мікросервісній архітектурі

Для забезпечення взаємодії між мікросервісами та зовнішніми системами можна використовувати API Gateway, який допомагає централізувати управління доступом та оптимізувати трафік [11].

API Gateway — це компонент, через який зовнішні клієнти взаємодіють з мікросервісами. Він функціонує як єдина точка доступу для всіх мікросервісів, маршрутизуючи запити до відповідного сервісу. Це дозволяє спростити процес взаємодії між клієнтами та мікросервісами, підвищуючи ефективність та безпеку системи.

API Gateway спрямовує запити до відповідних мікросервісів на основі URL, методів HTTP та інших параметрів. Це дозволяє клієнтам взаємодіяти з декількома сервісами через єдину точку входу.

API Gateway може перевіряти облікові дані користувачів, забезпечуючи автентифікацію запитів, контролювати кількість запитів, які надходять до сервісів, запобігаючи перевантаженню та зловживанням.

API Gateway може кешувати відповіді на запити, що зменшує навантаження на мікросервіси та покращує швидкість відгуку.

А також може вести журнали запитів та відповідей, що полегшує моніторинг роботи системи та виявлення проблем.

Переваги використання API Gateway:

– клієнти взаємодіють з множинними мікросервісами через один API, що зменшує складність;

– централізована автентифікація та авторизація;

– дозволяє легко додавати нові сервіси або змінювати існуючі без зміни клієнтського коду.

2.5. Інструменти розгортання мікросервісів

Розгортання мікросервісів зазвичай відбувається в кілька етапів, кожен з яких виконує певні функції та забезпечує успішну інтеграцію та роботу сервісів.

Вибір технології передбачає:

– вибір мови, яка найкраще підходить для певних мікросервісів, наприклад, Java, Go, Python, Node.js;

– використання відповідних фреймворків, таких як Spring Boot (Java), Express (Node.js), Flask (Python), що спрощують розробку мікросервісів;

– вибір баз даних (SQL або NoSQL) для зберігання даних мікросервісів. Наприклад, MySQL, PostgreSQL, MongoDB, Cassandra;

– інструменти для CI/CD: автоматизація тестування та розгортання через інструменти, такі як Jenkins, GitLab CI, CircleCI;

– контейнеризація**:** пакування мікросервісів у контейнери за допомогою Docker;

– оркестрація**:** використання Kubernetes, Docker Swarm для управління контейнеризованими мікросервісами, їх масштабування і балансування навантаження.

2.5.1. Вибір мови програмування для мікросервісної системи

Вибір мови для створення мікросервісів залежить від багатьох чинників, таких як специфіка завдання, продуктивність, доступність бібліотек, зручність у підтримці, та досвід команди. Розглянемо популярні варіанти.

Python — це високорівнева мова програмування, що відома своєю простотою, гнучкістю та великим набором бібліотек. Python активно застосовується в різних галузях, зокрема у веб-розробці, аналізі даних, автоматизації, штучному інтелекті та машинному навчанні.

Python відомий своєю читабельністю, що дозволяє розробникам швидше писати код та підтримувати його. Він має велику кількість вбудованих модулів, які покривають базові потреби розробників, підтримує кілька стилів програмування, включаючи процедурний, об’єктно-орієнтований та функціональний. Python працює на більшості операційних систем, таких як Windows, macOS, Linux.

Python забезпечує високий рівень модульності та можливість повторного використання коду завдяки підтримці модулів і пакетів. Інтерпретатор і стандартне API Python доступні безкоштовно як у вихідному, так і двійковому вигляді для основних платформ і можуть вільно розповсюджуватися. Відсутність етапу компіляції робить цикл розробки та налагодження швидким. Код на Python легкий у підтримці, і більшість його бібліотек сумісні на таких платформах, як UNIX, Windows, і macOS. Інтерактивний режим дозволяє зручно тестувати й налагоджувати код фрагментами, а підтримка широкого спектра апаратних платформ із уніфікованим інтерфейсом полегшує портативність.

Python також надає можливість додавати модулі низького рівня, які допомагають підвищити продуктивність коду, а інтеграція з основними базами даних (SQL і NoSQL) розширює можливості застосування Python для роботи з даними.

Python є ефективним вибором для мікросервісної архітектури завдяки підтримці фреймворків, які спрощують розробку та масштабування мікросервісів. Фреймворки, такі як Django і Flask, надають інструменти для швидкого створення RESTful API, керування запитами, маршрутизації та інтеграції з базами даних, що є ключовими аспектами мікросервісної архітектури. Інші фреймворки, наприклад, FastAPI, спеціалізуються на високій продуктивності та асинхронній обробці запитів, що забезпечує ефективну роботу сервісів при великій кількості одночасних запитів.

Крім цього, Python має багатий набір бібліотек і інструментів для реалізації оркестрації контейнерів (зокрема, через Docker), а також підтримує інтеграцію з інструментами моніторингу та CI/CD, що робить його універсальним вибором для побудови надійної мікросервісної системи.

Java - це об'єктно-орієнтована мова програмування, що використовується для створення багатьох типів програмного забезпечення, від десктопних до серверних додатків. Java підходить для великих корпоративних систем, мобільних додатків (Android) та веб-додатків (через Java EE).

Java є одним із найбільш популярних виборів для мікросервісної архітектури, особливо для складних корпоративних систем, завдяки своїй надійності, стабільності та широкому набору інструментів. Основні фреймворки, такі як Spring Boot і Micronaut, створені для розробки мікросервісів і надають простий спосіб реалізації RESTful API, обробки запитів і керування транзакціями. Spring Boot особливо цінується за модульність і зручні інструменти для впровадження залежності, автоматичного налаштування та підтримки великої екосистеми модулів і бібліотек.

Java також має відмінні засоби для масштабування, обробки багатопоточності та асинхронних запитів. Вона оптимальна для систем з високими вимогами до продуктивності та відмовостійкості. Платформа JVM дозволяє використовувати Java у середовищах з великим навантаженням і

легко інтегруватися з контейнерами, такими як Docker, і системами оркестрації на основі Kubernetes.

JavaScript — це динамічна типізована, однопоточна інтерпретована мова програмування, що використовується переважно для веб-розробки. Вона дозволяє створювати інтерактивні елементи на веб-сторінках. Завдяки своїй асинхронній природі JavaScript може обробляти запити без блокування основного потоку виконання. Мова підтримує об'єктно-орієнтований підхід, що дозволяє організувати код у зрозумілі модулі та компоненти. JavaScript також широко використовується на серверній стороні через платформу Node.js, можливо використовувати одну мову на всіх рівнях стека [12].

JavaScript є інтерпретованою мовою програмування, тобто код виконується безпосередньо інтерпретатором, а не компілюється в машинний код перед виконанням. Це дозволяє розробникам писати та тестувати код швидше, оскільки немає необхідності проходити етап компіляції. Інтерпретація JavaScript відбувається в браузері (на стороні клієнта) або на сервері (з використанням Node.js). Завдяки своїй інтерпретованій природі JavaScript також підтримує динамічну типізацію, що дозволяє змінювати типи змінних під час виконання програми [13].

JavaScript – це однопоточна мова, тобто вона виконує код у єдиному потоці. Це означає, що JavaScript обробляє лише одну операцію за раз, виконуючи задачі по черзі. Однак для роботи з асинхронними операціями, такими як запити до сервера, таймери або взаємодія з файлами, JavaScript використовує подієвий цикл і механізм зворотних викликів (callback), обіцянки (Promise) та async/await, що дозволяє працювати з багатьма задачами одночасно на вигляд. Незважаючи на однопоточність, подієвий цикл JavaScript забезпечує ефективне управління асинхронними операціями.

Java та JavaScript мають суттєві відмінності:

– Java використовує статичну типізацію, що забезпечує перевірку типів під час компіляції, підвищуючи стабільність та надійність коду. JavaScript, навпаки, є динамічною типізованою мовою, що дозволяє більш

гнучку роботу з типами, але вимагає обережного підходу для уникнення помилок;

– Java створена для великих корпоративних застосунків і підтримує складні середовища з багатьма різними компонентами та інтеграціями. JavaScript зазвичай застосовується для розподілених систем та взаємодії з веб-інтерфейсами;

– Java переважно обмежена серверною стороною. JavaScript же працює як на клієнті, так і на сервері (з використанням Node.js), що робить його універсальним для веб-розробки;

– Java побудована на об’єктно-орієнтованій парадигмі (ООП), що забезпечує модульність. JavaScript дозволяє комбінувати ООП і функціональний стиль програмування, що додає гнучкості та розширює можливості для написання різноманітних типів програм.

JavaScript і Python мають різні сильні сторони:

– Python популярний у менших проектах і наукових дослідженнях, тоді як серверний JavaScript (Node.js) добре підходить для розподілених і масштабованих систем;

– серверний JavaScript підтримує багатопоточність та асинхронність завдяки подіям та потокам, тоді як Python працює однопоточно, але підтримує асинхронність через бібліотеки;

– JavaScript дозволяє писати в функціональному стилі (що корисно для розподілених обчислень). Python підтримує процедурну та об’єктно-орієнтовану парадигми, але менш орієнтований на функціональний стиль;

– Python має потужні бібліотеки для обробки та аналізу даних (наприклад, Pandas, NumPy), JavaScript ж має багату екосистему бібліотек для клієнтської та серверної веб-розробки, включаючи такі фреймворки, як Express, Angular, та React.

Завдяки цим перевагам JavaScript стає все більш популярним вибором для розробників, які створюють мікросервіси, особливо в середовищах, де потрібна швидка та ефективна обробка запитів.

2.5.2. Вибір відповідних фреймворків

Angular, React і Vue — одні з найпопулярніших інструментів JavaScript для розробки складних, інтерактивних і сучасних інтерфейсів користувача. Вони допомагають ефективно створювати масштабовані веб-застосунки, що швидко реагують на дії користувачів. За допомогою додаткових бібліотек, таких як React Native, Ionic (з підтримкою Angular або React) і NativeScript, можна також розробляти кросплатформні мобільні додатки, використовуючи Angular, React і Vue.

Серед цих фреймворків немає абсолютного лідера — кожен має свої переваги та обмеження, які можуть бути більш або менш відповідними в залежності від задачі та специфіки команди розробників. Вибір найкращого інструменту значною мірою визначається поставленою метою та вимогами до проекту.

Angular — повноцінний фреймворк від Google для побудови складних веб-застосунків, пропонує двостороннє зв’язування даних та велику кількість інструментів для побудови масштабованих програм.

React є бібліотекою для побудови інтерфейсів користувача від Facebook, відомий своєю компонентною архітектурою, що дозволяє легко створювати інтерактивні елементи [14].

Vue - легкий фреймворк з гнучкою архітектурою, який дозволяє легко інтегрувати його в поточні проекти або використовувати для створення нових застосунків.

З усіх трьох, Angular розглядають як повноцінний фреймворк, іноді навіть називають його “платформою”, оскільки він містить інструменти для широкого контролю інтерфейсу користувача. Angular забезпечує зручність роботи з введенням даних, валідацією форм, маршрутизацією, управлінням станом, запитами через Ajax HTTP та реалізацією PWA.

Angular надає також офіційний CLI, що значно полегшує створення та керування проектами, оновлення їх залежність, розгортання тощо. Ключова

мета Angular — створення багаторазових компонентів користувацького інтерфейсу, які можна поєднувати для побудови повноцінного інтерфейсу з набору компонентів.

React, на відміну від Angular, є бібліотекою, що спеціалізується на редагуванні елементів у DOM та ефективному їх управлінні. Основна мета React полягає у створенні користувацьких інтерфейсів за допомогою компонентів. Бібліотека надає всі необхідні інструменти для маніпуляцій з елементами інтерфейсу. Проте, у React відсутня вбудована підтримка валідації форм. Бібліотека не має вбудованого маршрутизатора для зміни та рендеру різних компонентів залежно від змін у URL, а також не включає власного Http-клієнта. React забезпечує управління внутрішнім станом компонентів і передачу стану між компонентами, але для реалізації додаткових функцій, таких як маршрутизація та валідація форм, потрібно використовувати сторонні бібліотеки.

Vue — це бібліотека, яка поєднує в собі особливості як React, так і Angular. Vue включає в себе вбудоване управління станом і маршрутизатор, але не пропонує підтримки валідації форм і функціональних можливостей для роботи з HTTP.

Як і Angular, так і React, ядро Vue фокусується на створенні інтерфейсів користувача шляхом поєднання повторно використовуваних компонентів. Однак Vue надає більше можливостей, ніж React, але трохи менше, ніж Angular. Вибір між цими бібліотеками залежить від конкретного проекту та особистих уподобань.

Деякі функції, такі як управління станом і маршрутизація, є необхідними в більшості проектів, незалежно від їхнього масштабу. Angular і Vue мають повну вбудовану підтримку для цих операцій, тоді як React має лише базову підтримку управління станом і не має вбудованої маршрутизації.

Vue має легку інтеграцію, простоту використання та гнучкість, але його популярність менша в порівнянні з Angular і React, а також має меншу

екосистему.

Простота React, висока продуктивність завдяки Virtual DOM і акцент на компонентах та візуалізації інтерфейсу є значними перевагами. Однак, з недоліків — потрібно додавати бібліотеки для маршрутизації та управління станом. React має активну спільноту, яка розробляє додаткові бібліотеки, такі як React Router, Redux та Formik. Це дозволяє React зосередитися на наданні найкращої бібліотеки для рендерингу інтерфейсу, в той час як спільнота забезпечує розширення для його функціональності.

Angular, у свою чергу, має вбудовані рішення для маршрутизації та управління станом, проте його структура є складнішою.

Angular був розроблений Google, React — Facebook, а Vue — це проект з відкритим кодом, створений спільнотою. Всі ці інструменти є компонентними бібліотеками та фреймворками: Angular є повноцінним фреймворком, тоді як Vue та React — звичайні бібліотеки.

Angular вимагає використання TypeScript, поєднуючи HTML з логікою на TypeScript. React використовує JavaScript з синтаксисом "JSX", який є комбінацією HTML та JavaScript. Vue, в свою чергу, розробляється на основі нативного JavaScript та поєднує HTML з JavaScript.

Щодо навчання, Vue та React є найлегшими для освоєння, в той час як Angular має найвищу криву навчання через вимоги до TypeScript та обмеження фреймворку. Усі три технології забезпечують чудову продуктивність роботи додатків, проте React вважається найшвидшим серед них. React є найпопулярнішою бібліотекою для створення користувацьких інтерфейсів, маючи найбільшу спільноту розробників, тоді як Angular та Vue менш популярні.

2.5.3. Вибір баз даних для мікросервісів

У системах автоматизованого документообігу вибір бази даних надзвичайно важливий, адже вона визначає, як зберігатимуться, оброблятимуться та управлятимуться великі обсяги документів, що мають різні типи даних і часто змінюються.

Реляційні бази даних (SQL) (PostgreSQL, MySQL, Oracle Database) підтримують ACID-транзакції, що забезпечує цілісність даних, особливо під час роботи з документами, де важлива історія змін, контроль доступу, управління версіями. Добре підходять для структурованих даних і дозволяють розділити таблиці за типами документів, користувачів, запитів.

Недоліки - менша гнучкість у зберіганні неструктурованих даних. Для зберігання великих файлів або вкладених структур може знадобитися додаткова обробка або зовнішні рішення.

Документоорієнтовані бази даних (NoSQL) (MongoDB, Couchbase) - придатні для зберігання документів у вигляді JSON або BSON, що дозволяє гнучко працювати з різними структурами даних. Легко масштабуються горизонтально, тому можуть витримувати велике навантаження від численних запитів.

Недоліки - обмежена підтримка транзакцій у багатьох NoSQL-рішеннях, хоча MongoDB має підтримку ACID для певних випадків, що може бути прийнятним для документаційних систем.

Для систем автоматизованого документообігу часто підходить комбінований підхід:

– використання реляційної бази (PostgreSQL) для зберігання структурованих метаданих документів та зв'язків між документами і користувачами;

– використання документоорієнтованої бази даних (MongoDB) для зберігання самих документів або вкладених структур, що дозволить зберігати неструктуровані дані з гнучкістю та масштабованістю.

PostgreSQL — це потужна, надійна реляційна база даних з відкритим кодом, яка підтримує стандарти SQL і відома своєю стабільністю та широким набором функцій. Вона використовується для зберігання, обробки та управління великими обсягами даних, забезпечуючи при цьому високу продуктивність транзакцій і підтримку складних запитів [15].

PostgreSQL гарантує цілісність і захищеність даних завдяки підтримці транзакційних принципів ACID, що дуже важливо для систем, які обробляють критичні дані. PostgreSQL дозволяє створювати власні функції, типи даних, індекси, та інтегрувати додаткові модулі. Крім того підтримує зберігання документів у форматі JSON, який є оптимізованим для пошуку.

В PostgreSQL підтримуються паралельні запити, реплікації для налаштування резервних копій та створювання багаторівневої архітектуру для розподілених систем. PostgreSQL підтримує різноманітні види індексів (B-tree, GiST, GIN, BRIN та ін.).

Крім основного SQL-синтаксису, PostgreSQL підтримує кілька мов для створення та виконання збережених функцій і процедур: Python, Perl, Tcl, JavaScript. Підтримка кількох мов у PostgreSQL надає розробникам гнучкість у виборі відповідного інструменту для специфічних задач, спрощує складну обробку даних і дозволяє легше інтегрувати PostgreSQL з іншими системами.

Переваги PostgreSQL:

– висока продуктивність та здатність працювати з великими обсягами даних;

– безпека і надійність, підтримка ролей і прав доступу до даних;

– велика кількість додатків, модулів та інструментів для інтеграції;

– підтримка різноманітних типів даних. Унікальність PostgreSQL полягає в його здатності працювати з географічними, часовими, текстовими та бінарними даними, а також створювати власні типи даних.

Недоліки PostgreSQL:

– вища крива навчання у порівнянні з деякими іншими базами даних;

– складніше налаштування та адміністрування для великих і

розподілених систем;

– обмежена підтримка горизонтального масштабування у порівнянні з деякими NoSQL-базами даних, хоча це компенсується засобами реплікації і шардингу за допомогою додаткових рішень.

PostgreSQL добре підходить для системи документообігу та управління контентом (CMS), фінансової системи, де потрібна надійна підтримка транзакцій, системи аналітики та обробки великих даних.

MongoDB — це документноорієнтована NoSQL база даних, яка розроблена для зберігання великих обсягів даних і відрізняється від традиційних реляційних баз даних своєю структурою та підходом до управління даними. Вона була створена в середині 2000-х років і зберігає дані у форматі документів BSON (Binary JSON). MongoDB є безкоштовною та має відкритий вихідний код, що робить її популярним вибором для багатьох розробників [16].

Документи у MongoDB можуть мати різні структури в межах однієї колекції, що спрощує зберігання і обробку неструктурованих даних. Дозволяє легко модифікувати структуру даних без складних міграцій, підтримує горизонтальне масштабування для роботи з великими обсягами даних і розподілення навантаження на кілька серверів.

MongoDB добре підходить для додатків, які потребують високої швидкості запису і обробки великих обсягів неструктурованих даних.

Підтримує різноманітні мови, включаючи Python, Java, Node.js, C#, що спрощує інтеграцію з різними сервісами та фреймворками. Підтримує реплікацію даних для забезпечення високої доступності, що дозволяє створювати резервні копії даних і швидко відновлювати їх у разі відмови сервера.

Переваги MongoDB:

– підтримка неструктурованих і напівструктурованих даних;
– простота масштабування і балансування навантаження;
– швидка обробка даних і висока швидкість запису;

– гнучкість структури даних для додатків, що швидко змінюються.

Недоліки MongoDB:

– відсутність підтримки транзакцій у складних запитах (до версії 4.0), що може ускладнювати роботу з критично важливими даними;

– не завжди підходить для складної аналітики, яка потребує зв’язків між багатьма таблицями.

MongoDB добре підходить для веб-застосунків, які потребують роботи з великими обсягами неструктурованих даних, наприклад, для додатків з великою кількістю користувачів, логів подій, систем IoT або для автоматизованих процесів, що часто змінюються.

2.5.4. Основи безперервної інтеграції (CI)

Безперервна інтеграція (Continuous Integration, CI) — це підхід у розробці програмного забезпечення, який передбачає регулярне злиття коду від усіх розробників у спільне сховище кілька разів на день. Основною метою CI є швидке виявлення помилок та підтримка стабільності основного коду, що дозволяє команді працювати над розробкою паралельно і швидше виявляти конфлікти чи помилки.

Основні етапи CI:

– автоматичне тестування (після кожного коміту запускається набір автоматизованих тестів, які допомагають перевірити функціональність нових змін і визначити, чи не спричинили вони помилок);

– автоматичне збирання (код збирається в єдине середовище для забезпечення сумісності та правильного функціонування всіх компонентів);

– перевірка якості коду (виконується аналіз якості коду, його стилю та можливих вразливостей);

– повідомлення команди (система CI надсилає розробникам повідомлення про успіх або помилку, щоб вони могли оперативно виправити будь-які недоліки).

CI-сервер розпізнає зміни в нововведених сервісах і запускає серію перевірок для перевірки інтеграції з наявним кодом. Він перевіряє належність компіляції та проходження всіх необхідних тестів, щоб підтвердити працездатність нового коду. Успішно зібраний і протестований код зберігається у вигляді артефактів — це полегшує подальше розгортання, особливо коли йдеться про масштабування. Артефакт створюється одноразово для конкретної версії і використовується для всіх розгортань цієї версії. Артефакти зберігаються в репозиторії CI-системи для доступу до них у майбутньому.

У CI-підході для мікросервісів кожен мікросервіс має окреме сховище коду, що використовується для збірок лише цього сервісу. При зміні коду окремого сервісу автоматично запускається тестування лише цього сервісу. Однак, у випадках, коли зміни стосуються кількох сервісів, можуть виникнути труднощі. Щоб забезпечити безперебійну інтеграцію, необхідно запускати збірки і тестування кожного сервісу паралельно, забезпечити обмін станами між сервісами, щоб уникнути конфліктів.

Таким чином, CI-система знижує час на інтеграцію і тестування, мінімізує ризики помилок і забезпечує швидку адаптацію змін у системі.

Переваги підходу безперервної інтеграції:

– швидке виявлення помилок (регулярна інтеграція дозволяє виявляти проблеми на ранніх етапах, що знижує ризики накопичення помилок у кодовій базі);

– стабільність основного коду (постійне тестування та збірка коду підтримують стабільність головної гілки);

– автоматизація (завдяки CI значна частина перевірок автоматизується, зменшує потребу в ручному тестуванні);

– покращення якості коду (CI сприяє дотриманню стандартів якості та забезпечує високу якість програмного продукту).

2.5.5. Визначення контейнеризації

Кількість сервісів у програмній системі може бути значною, і кожен сервіс може використовувати різні технології, що створює виклики в управлінні середовищами. Для спрощення процесу створення та підтримки таких різноманітних середовищ використовується технологія контейнерів.

Контейнери дозволяють ізолювати кожен сервіс зі всіма необхідними залежностями, створюючи стандартизовані середовища для його запуску. Це допомагає уникати конфліктів між сервісами, спрощує їх розгортання та масштабування. Контейнеризація стала основною тенденцією в розробці програмного забезпечення, виступаючи як альтернатива або доповнення до віртуалізації. Контейнери використовують спільне ядро операційної системи, що дозволяє їм бути більш легкими та ефективними, порівняно з віртуальними машинами. У контейнері можна запускати будь-які програми разом із усіма необхідними залежностями, бібліотеками та конфігураційними файлами. Це дозволяє забезпечити стабільність і консистентність роботи додатка незалежно від середовища, в якому він запускається.

Пакет програмного забезпечення або «контейнер» здатний працювати на будь-якій платформі чи в хмарі, що забезпечує його портативність. Контейнеризація дозволяє розробникам створювати та розгортати програми швидше і безпечніше, оскільки додатки ізольовані один від одного та можуть працювати незалежно. Існуючі програми можуть бути перепаковані в контейнери для більш ефективного використання обчислювальних ресурсів. Відмова одного контейнера не впливає на роботу інших контейнерів.

Одним із найвідоміших інструментів, що підтримують контейнеризацію, є Docker [17, 18].

Docker надає можливість легко створювати, розгортати та управляти контейнерами, забезпечуючи ізоляцію середовищ. Контейнери використовуються в різних середовищах, включаючи локальні машини, сервери та хмарні платформи.

Docker дозволяє упакувати кожен мікросервіс разом із його залежностями (бібліотеками, фреймворками та налаштуваннями) в окремий контейнер. Це гарантує, що мікросервіс працює однаково незалежно від середовища, де він розгорнутий (локально, на сервері або в хмарі). Кожен мікросервіс запускається в окремому контейнері, що дозволяє незалежно управляти версіями, ресурсами і життєвими циклами сервісів. Оскільки всі залежності містяться в контейнері, немає конфліктів між бібліотеками або версіями різних мікросервісів. Для масштабування системи можна легко збільшувати кількість контейнерів із певним мікросервісом залежно від навантаження. Це особливо зручно при використанні оркестраційних інструментів, таких як Kubernetes.

Завдяки Docker, команди можуть працювати незалежно одна від одної, створюючи, тестуючи і розгортаючи мікросервіси. Це сприяє розвитку підходу CI/CD (Continuous Integration / Continuous Deployment).

Архітектура Docker складається з кількох основних компонентів (рис.2.6):

1. Docker клієнт — основний інтерфейс для взаємодії з Docker. для відправки запитів демону Docker клієнт використовує:
    a) командний рядок або інші інструменти;
    b) REST API, UNIX-сокети або мережевий інтерфейс.
2. Docker демон (Docker Daemon) відповідає за управління контейнерами, образами, мережами та сховищами. Демон отримує команди від клієнта та виконує їх, забезпечуючи створення, запуск і зупинку контейнерів.
3. Docker образи (Images) — статичні шаблони для створення контейнерів. Образи містять інструкції для запуску програми, операційної системи та залежностей. Вони можуть зберігатися локально або у віддалених репозиторіях, таких як Docker Hub.
4. Docker контейнери (Containers) — це робочі екземпляри образів. Вони ізольовані середовища, що запускаються на основі образу, і містять усі

необхідні залежності для виконання програми. Кожен контейнер має власний файловий простір, мережу та обмеження по ресурсах, проте використовує спільне ядро ОС з іншими контейнерами.

5. Docker реєстри (Registries) — це місця для зберігання та розповсюдження образів. Вони полегшують процес спільної роботи та управління версіями програм. Найпоширенішим публічним реєстром є Docker Hub, який використовується за замовчуванням для пошуку та завантаження зображень. За допомогою команди docker pull можна завантажити образ з реєстру, а за допомогою docker push — відправити свій образ в реєстр.

6. Docker мережі (Networks) надає можливість створення ізольованих віртуальних мереж, що дозволяють контейнерам взаємодіяти між собою або з зовнішнім світом. Мережі можна налаштовувати для різних сценаріїв використання, наприклад, для забезпечення безпеки та ізоляції.

7. Docker томи (Volumes) — це механізм зберігання даних, який дозволяє контейнерам зберігати та спільно використовувати дані. Вони зберігаються поза життєвим циклом контейнера, тому дані не втрачаються після його завершення.

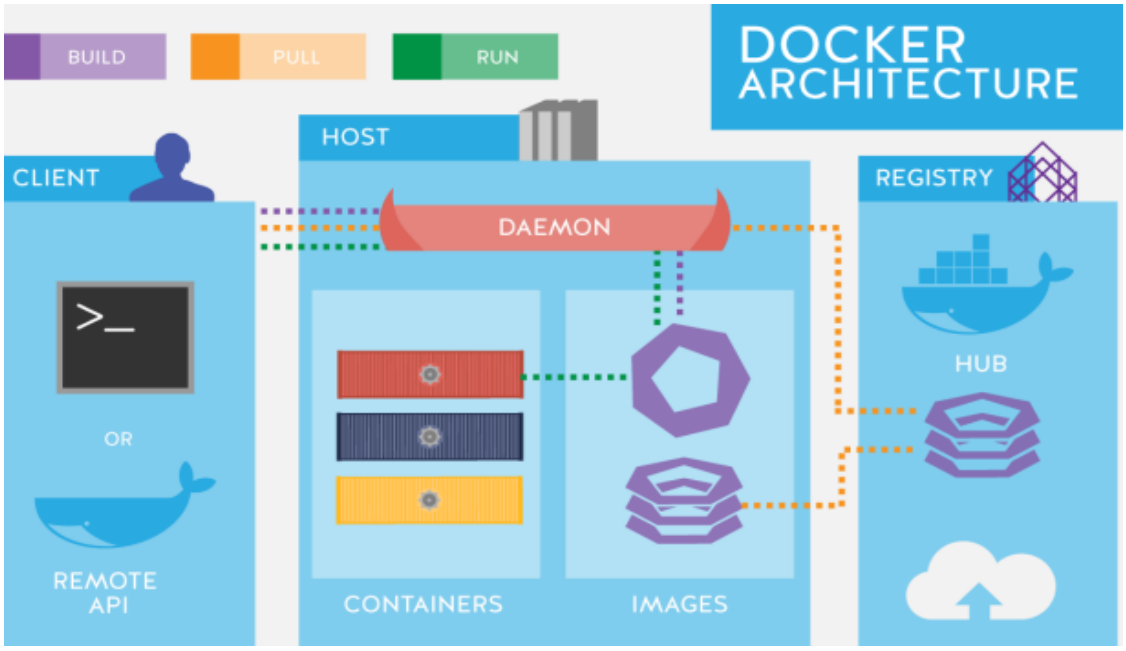

Рисунок 2.6 – Архітектура системи Docker

Визначимо основні функції та властивості Docker контейнера:

– контейнер містить усередині все необхідне для роботи програми — операційну систему, бібліотеки, залежності та код. За межами контейнера цей вміст ізольований від інших процесів;

– контейнер можна легко перенести та запускати на будь-якій платформі — на локальному комп'ютері, на пристрої колеги або на серверах хмарного провайдера без змін у конфігурації;

– контейнер взаємодіє із зовнішнім світом через відкриті порти, які дозволяють, наприклад, доступ через браузер, також можна налаштувати взаємодію з контейнером через командний рядок;

– контейнер створюється на основі образу, який містить інструкції в Dockerfile, операційну систему, бібліотеки та програмний код. Цей образ можна завантажити з віддаленого репозиторію та використовувати для створення контейнера в будь-якому середовищі.

2.5.6. Визначення оркестрації контейнерів

Оркестрація контейнерів — це процес автоматизованого управління, розгортання, масштабування і підтримки контейнеризованих додатків у кластерних середовищах. Вона допомагає ефективно координувати контейнери, щоб забезпечити стабільність і надійність сервісів на великих інфраструктурах.

Оркестрація контейнерів вирішує такі завдання:

– автоматичне розподілення контейнерів між вузлами;

– балансування навантаження;

– забезпечення відмовостійкості та самовідновлення;

– централізоване управління конфігураціями та секретами.

Оркестрація допомагає компаніям підтримувати безперервність бізнес-процесів, автоматизуючи ключові аспекти життєвого циклу контейнерів, особливо в мікросервісній архітектурі.

Платформа оркестрації контейнерів — це система, яка дозволяє автоматизувати управління контейнерами на великих розподілених кластерах.

Розвиток платформ для оркестрації контейнерів почався з потреби компаній у масштабованих і автоматизованих інструментах для керування контейнеризованими додатками. Спочатку для управління контейнерами використовували Docker, який запропонував простий спосіб створення та запуску контейнерів. Однак, із зростанням числа контейнерів і складністю інфраструктур, виникла потреба в платформах, здатних автоматизувати завдання розгортання та управління контейнерами на масштабних кластерах.

Docker Swarm, перша оркестраційна платформа від Docker, надавала основні функції для об’єднання контейнерів у кластери, їх розподілення між вузлами, але мала обмежені можливості для складних розгортань.

Архітектура Docker Swarm (рис. 2.7) складається з кількох основних компонентів:

1) менеджери (managers) відповідають за управління кластерами, розподілення завдань, та підтримку стійкості. Один з менеджерів є лідером (leader), який приймає рішення про розподіл навантаження і конфігурації; решта виконують роль резервних менеджерів (followers), що переймають керування у разі відмови лідера;

2) робочі ноди (workers) приймають завдання від менеджерів і запускають необхідні контейнери, передають звіти про стан запущених контейнерів менеджерам, щоб ті могли оновлювати інформацію про кластер. Робочі ноди не приймають рішень про розподіл сервісів і не управляють кластером, вони лише виконують завдання, призначені менеджерами;

3) сервіси (services) — це абстракція для контейнерів, яка дозволяє керувати і масштабувати їх у кластері. Вони визначають конфігурацію для контейнерів (наприклад, образ контейнера, порти, що необхідно відкрити, обмеження по ресурсах) та виконують автоматичне масштабування, щоб задовольнити потреби у навантаженні. Один сервіс може містити кілька

екземплярів (реплік) контейнера;

4) задачі (tasks) - це одиничні елементи роботи, які виконуються у вигляді контейнерів. Задача призначена конкретному вузлу залишається на ньому до завершення. Якщо ж завдання перестає виконуватися або вузол виходить з ладу, диспетчер переназначить нову копію цього завдання на інший доступний вузол.

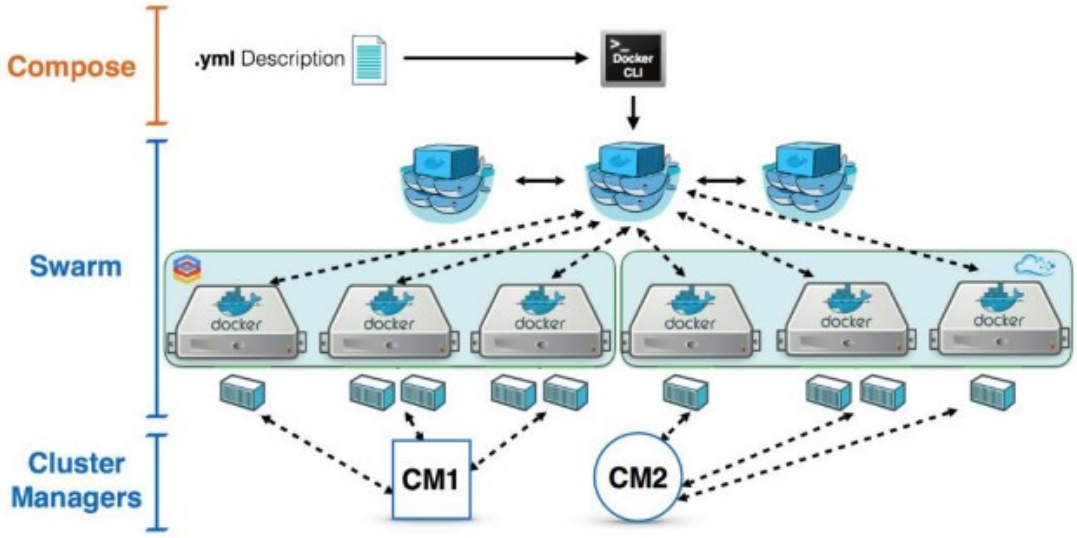

Рисунок 2.7 – Архітектура Docker Swarm

Із плюсів Docker Swarm можна відзначити легкість у використанні та швидке освоєння. Із мінусів – вузький функціонал, який не має таких значних можливостей відмовостійкості як у Kubernetes.

Docker Swarm — зручний інструмент оркестрації для невеликих і середніх проектів, особливо якщо основною вимогою є простота розгортання.

Поява Kubernetes суттєво змінило підхід до оркестрації. Kubernetes запропонував багатий набір функцій для автоматизації розгортання, масштабування, балансування навантаження, а також для управління життєвим циклом контейнерів у складних системах. Kubernetes і його екосистема значно розширилися завдяки розвитку таких проектів, як Helm

(інструмент для управління пакетами), Prometheus (для моніторингу) та Istio (для управління мережею сервісів). Також з’явились сервіси з підтримкою multi-cloud (можливість працювати одночасно в кількох хмарних середовищах).

Сьогодні Kubernetes є домінуючою платформою для оркестрації контейнерів. Вона широко використовується як основа для хмарно-нативних архітектур і мікросервісів, а сучасні рішення розширюють її функціонал для забезпечення безпеки, моніторингу, та керування мережею сервісів [19].

Kubernetes керує запуском Docker-контейнерів на численних хостах, забезпечуючи розміщення та реплікацію контейнерів у великих масштабах. Він автоматично контролює стан додатків і може перепланувати їх роботу у разі збою обладнання. Kubernetes абстрагує апаратну інфраструктуру, представляючи її як єдиний великий обчислювальний ресурс, що дозволяє розгортати й запускати програмні компоненти на рівні всього центру обробки даних.

Система обирає сервери для розміщення компонентів, розгортає їх та надає можливість легко знаходити й взаємодіяти з іншими компонентами програми. Kubernetes надає розробникам просту платформу для розгортання та запуску додатків, не вимагаючи знань про деталі роботи тисяч програм, що працюють на обладнанні. Сьогодні Kubernetes є визнаним стандартом для запуску розподілених систем, як у хмарному середовищі, так і локально.

Архітектура Kubernetes (рис. 2.8) складається з основних компонентів, що розміщуються на двох рівнях: рівні управління (control plane) та робочих вузлів (worker nodes).

1. Рівень управління (control plane) відповідає за загальне керування та координацію всіх елементів у кластері Kubernetes. Основні компоненти рівня управління:

– API-сервер (API server) - основний компонент для взаємодії користувачів і компонентів із кластером. Він приймає запити до Kubernetes і розподіляє їх між іншими сервісами через REST API;

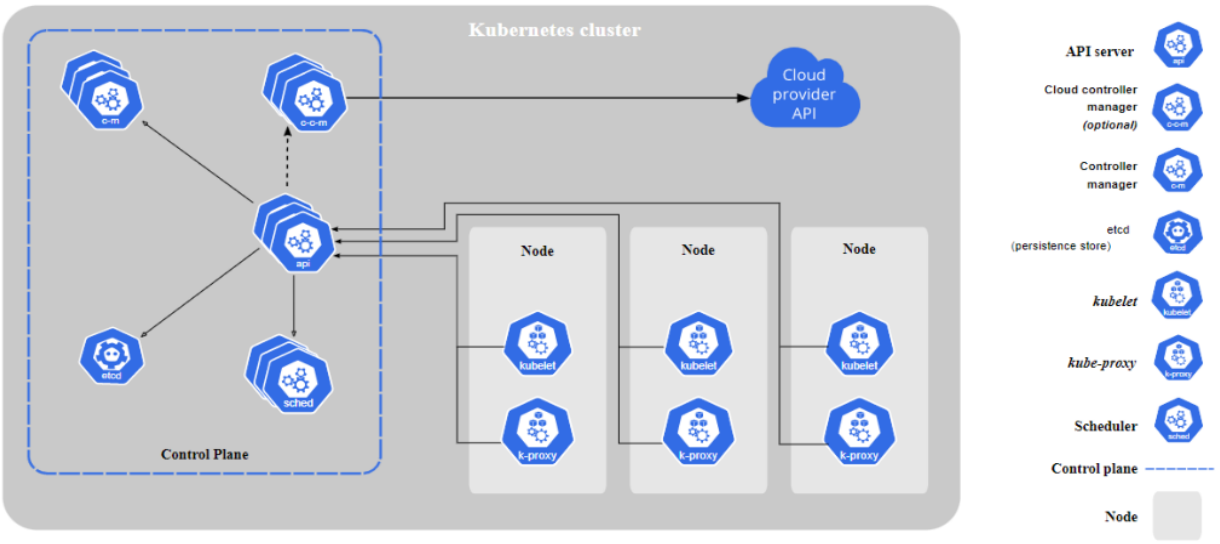

Рисунок 2.8 – Архітектура Kubernetes

– сховище (etcd) ключ-значення, яке зберігає всю конфігураційну інформацію про стан кластера. Це центральна база даних для всіх конфігурацій та налаштувань Kubernetes. Відповідає за забезпечення надійного і розподіленого зберігання даних;

– планувальник (scheduler) вибирає робочий вузол для запуску нових подів (груп контейнерів), враховуючи їхні вимоги до ресурсів та наявні ресурси на вузлах;

– менеджер контролерів (controller manager) запускає різні контролери, які відповідають за моніторинг стану кластеру та автоматичне управління ресурсами (наприклад, ReplicaSet, Node Controller).

2. Робочі вузли (worker nodes) — це сервери, на яких розгортаються додатки та здійснюється виконання контейнерів. Основні компоненти робочих вузлів:

– кублет (Kubelet) — агент, який працює на кожному робочому вузли й відповідає за управління станом контейнерів. Кублет отримує завдання від API-сервера й запускає контейнери за допомогою контейнерного середовища (наприклад, Docker).

– kube proxy — мережева служба, що відповідає за маршрутизацію та балансування навантаження запитів на поди. Kube Proxy допомагає в

налаштуванні правил доступу до подів і управлінні трафіком між ними.

– контейнерний рушій (container runtime) — виконує контейнери в кожному поді. Kubernetes підтримує кілька контейнерних середовищ, наприклад Docker або containerd.

3. Додаткові компоненти:

– pods - основна одиниця розгортання в Kubernetes, яка може містити один або кілька контейнерів;

– namespace - логічне розділення ресурсів для організації та управління великими кластерами;

– services створюють стійкі точки доступу до наборів подів, дозволяючи їм залишатися доступними навіть при зміні IP-адрес.

Kubernetes архітектура дозволяє надійно масштабувати додатки, забезпечує автоматичне відновлення у разі збоїв та полегшує розгортання і управління розподіленими системами.

Розгортання мікросервісів — це складний, але структурований процес, який вимагає ретельного планування, розробки, тестування та моніторингу. Кожен етап є важливим для забезпечення надійності та ефективності системи в цілому.

РОЗДІЛ 3
РОЗРОБКА СИСТЕМИ АВТОМАТИЗОВАНОГО ДОКУМЕНТООБІГУ З ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

3.1. Технологічний стек для системи автоматизованого документообігу

Для розробки технологічного стеку системи автоматизованого документообігу з використанням мікросервісів у банківській сфері, необхідно вибрати рішення, які найкраще підходять для кожного сервісу. Розроблена система дозволить управляти документами, автоматизувати робочі процеси, керувати користувачами. Технологічний стек повинен базуватися на вимогах до безпеки, масштабованості, швидкодії та надійності.

Система автоматизованого документообігу для банку на базі мікросервісної архітектури складається з кількох основних компонентів, кожен з яких виконує свої функції і відповідає за окрему частину процесу обробки документів:

1. Інтерфейс користувача:

a) панель користувача для співробітників банку, які завантажують, переглядають, редагують і підписують документи;

b) панель адміністратора - інтерфейс для адміністраторів системи, з можливістю керування правами доступу, налаштування мікросервісів і моніторингу стану системи;

c) модуль для клієнтів, якщо потрібно забезпечити клієнтам доступ до певних документів (наприклад, виписок, угод).

2. API Gateway - спрямовує запити клієнтів і користувачів до потрібних мікросервісів, забезпечує єдину точку доступу до мікросервісів.

3. Сервіси управління документами (завантаження, обробка, підпис документів, історія).

4. Сервіс управління ролями та доступом (визначає права доступу співробітників і клієнтів до документів, Аутентифікація та авторизація

користувачів через OAuth2).

5. Сервіс повідомлень і черг (сповіщає користувачів про важливі події, черга повідомлень асинхронного обміну інформацією між сервісами Apache Kafka).

6. Система зберігання документів (сховище) (об’єктне сховище для документів, NoSQL база даних для зберігання метаданих про документи, кешування (Redis) для тимчасового зберігання запитів або документів, які часто використовуються).

7. Моніторинг і логування (Prometheus для збору метрик з мікросервісів, Grafana для візуалізації продуктивності, ELK Stack для централізованого збору та аналізу логів).

8. Система управління робочими процесами (автоматизує основні етапи обробки документів).

9. Забезпечення безпеки (шифрування даних, контроль доступу, антивірусная перевірка).

Визначимо оптимальні рішення стеку технологій для створення мікросервісів системи автоматизованого документообігу у банківській сферів.

3.1.1. Frontend (Клієнтська частина)

Фротенд охоплює всі елементи інтерфейсу, з якими користувачі можуть взаємодіяти: кнопки, форми, текстові поля, зображення, меню та інші елементи дизайну, зберігає та оновлює дані, що відображаються на екрані.

Для клієнтської частини веб-додатків традиційно використовується JavaScript (разом з HTML і CSS).

Фронтенд будується за допомогою таких технологій, як:

1. HTML (HyperText Markup Language) — це мова розмітки, яка визначає основну структуру та зміст вебсторінки, організовуючи елементи, такі як заголовки, абзаци, посилання, зображення та таблиці.

2. CSS (Cascading Style Sheets) — мова стилів, яка дозволяє форматувати зовнішній вигляд HTML-елементів: змінювати кольори, шрифти, вирівнювання, відступи тощо, а також робити сайт адаптивним для різних пристроїв.

3. JavaScript — мова програмування, що додає динаміку та інтерактивність до сторінки, дозволяючи оновлювати контент, обробляти події (кліки, натискання клавіш), маніпулювати елементами DOM та взаємодіяти з сервером. Вона дозволяє розробникам маніпулювати HTML та CSS, забезпечуючи динамічний контент.

4. Фреймворки - React, яка спрощує створення масштабованих і динамічних інтерфейсів. React — це популярна бібліотека JavaScript, створена Facebook для побудови інтерфейсів користувача (UI), які забезпечують швидке і чуйне взаємодію з користувачами та відображення даних у реальному часі. React дозволяє розробникам створювати багатосторінкові веб-додатки, використовуючи компонентний підхід, що робить код більш структурованим і зручним для підтримки. Він також використовує JSX, синтаксис, який дозволяє писати HTML-подібний код у JavaScript.

5. UI бібліотека Bootstrap — колекція готових стилів та компонентів, які дозволяють швидко будувати інтерфейси, додаючи до них сучасний вигляд, що узгоджується з принципами зручності та адаптивності.

6. Додаткова бібліотека (Redux) — для управління станом додатка і виконання API-запитів. JavaScript підтримує безліч бібліотек, які доповнюють його функціональність. Бібліотека Redux часто використовується з React для централізованого зберігання стану програми.

Технологічний стек для системи автоматизованого документообігу в банківській системі клієнтської частини:

- мова програмування**:** JavaScript;
- фреймворки**:** React;
- бібліотека для управління станом: Redux;

– маршрутизація: React Router;

– CSS-фреймворки: Bootstrap.

3.1.2. Backend (Серверна частина)

Для серверної частини веб-додатків можна використовувати кілька мов, але для JavaScript - орієнтованого проекту підходить Node.js, як платформа з використанням JavaScript. Це дозволяє писати серверний код тією ж мовою, що і для клієнтської частини, спрощуючи розробку.

Основні компоненти для побудови серверної частини з використанням Node.js:

– платформа Node.js — забезпечує можливість запуску JavaScript на сервері. Вона відома своєю ефективною обробкою великої кількості паралельних запитів завдяки асинхронній обробці і подієво-орієнтованій архітектурі;

– мова JavaScript — рідна мова для Node.js, з підтримкою модульності, асинхронності та об’єктно-орієнтованих підходів;

– фреймворки Express.js — це легкий і гнучкий фреймворк для веб-розробки на основі платформи Node.js, який широко використовується для створення серверної частини додатків, зокрема мікросервісів. Він спрощує налаштування HTTP-серверів, управління маршрутизацією, обробку запитів і відповідей, а також інтеграцію з іншими інструментами та базами даних;

– бази даних:

• MongoDB (разом з Mongoose для ODM) — документно-орієнтована NoSQL база даних, зручно інтегрується з JavaScript і підходить для JSON-подібних даних. Підходить для зберігання неструктурованих документів і швидкого доступу до даних;

• PostgreSQL — реляційна база даних з підтримкою JSON, для зберігання структурованих даних і підтримки транзакцій;

– інші інструменти:

• Redis — це система керування базами даних, яка зберігає дані в пам'яті (in-memory data store) та працює як сховище типу NoSQL**.** Redis часто використовується в архітектурі мікросервісів для забезпечення високої швидкості доступу до даних, що робить його ідеальним для кешування черг повідомлень та обробки сеансів користувачів у реальному часі;

• Docker — для контейнеризації сервісів, що забезпечує портативність та гнучкість розгортання;

• Kubernetes – для автоматизації розгортання, масштабування і управління контейнеризованими додатками.

Технологічний стек для системи автоматизованого документообігу в банківській системі серверної частини:

- платформа**:** Node.js;
- мова програмування**:** JavaScript;
- фреймворки**:** Express.js;
- бази даних:
  - PostgreSQL;
  - MongoDB;
- кешування**:** Redis;
- контейнеризація: Docker;
- оркестрація: Kubernetes.

3.1.3. Комунікація між мікросервісами

Для комунікації між мікросервісами в системі автоматизованого документообігу в банківській системі вибір технологічного стеку повинен забезпечувати надійність, безпеку, а також низьку затримку передачі даних.

Якщо потрібні негайні відповіді та існує потреба в централізованому управлінні доступом до сервісів (для запитів, пов’язаних із взаємодією з користувачем), краще застосовувати синхронну комунікацію з API Gateway.

Якщо система потребує обробки великих обсягів даних та повинна

бути стійкою до відмов, рекомендується асинхронна комунікація через повідомлення з використанням системи черг, наприклад, Kafka або RabbitMQ.

Основні компоненти стеку для комунікації між мікросервісами:

– API Gateway — використовується як єдина точка входу для всіх зовнішніх і внутрішніх запитів. API Gateway маршрутизує запити до відповідних мікросервісів, обробляє автентифікацію, кешування і обмеження трафіку, що спрощує взаємодію з користувачами та клієнтськими додатками: Kong або AWS API Gateway – для управління API та маршрутизації запитів. Kong працює локально та в хмарі, підтримує гібридні середовища. AWS API Gateway повністю керований хмарний сервіс від Amazon Web Services.

– Синхронна комунікація (REST, gRPC):

– REST API — один із найпоширеніших протоколів для обміну даними між сервісами. Використовує HTTP для запитів і відповідає JSON або XML-даними. REST простий у реалізації, але не підходить для дуже швидкої взаємодії між сервісами, де потрібна низька затримка;

– gRPC — фреймворк для швидкої синхронної комунікації, який використовує HTTP і протоколи для стиснення даних. Він забезпечує високу продуктивність і підтримує різні мови програмування;

– Асинхронна комунікація (повідомлення, події):

– повідомлення через брокери повідомлень (RabbitMQ, Apache Kafka) — асинхронна комунікація, де мікросервіси надсилають та отримують повідомлення через черги. RabbitMQ підходить для обміну повідомленнями, а Kafka для обробки великих потоків даних в реальному часі.

– події (Event-Driven Architecture) — підхід, коли один мікросервіс «створює подію», а інші сервіси «підписуються» на неї. Цей підхід полегшує масштабування та додає гнучкості в роботі з даними.

Залежно від конкретних вимог системи, підхід до комунікації може бути комбінованим, об'єднуючи переваги різних методів для досягнення оптимальної продуктивності і надійності:

– запити даних у реальному часі — використовують REST або gRPC;

– постійний потік даних або подій — використовують Apache Kafka для обміну даними в реальному часі;

– обмін короткими асинхронними повідомленнями — RabbitMQ для швидкого обміну повідомленнями між сервісами;

– відстеження змін в одній частині системи — Event-Driven Architecture дозволяє створювати і обробляти події в різних частинах системи.

Технологічний стек для комунікації між мікросервісами:

– API Gateway: Kong [20]

– брокера повідомлень: RabbitMQ, Apache Kafka

– протоколи: REST; gRPC.

3.1.4. Безпека

Використання надійного технологічного стеку в частині безпеки для систем автоматизованого документообігу в банківських установах є критично важливим для захисту конфіденційності, забезпечення доступності даних і захисту від кібератак.

1. Аутентифікація та авторизація:

- OAuth 2.0 — для керування аутентифікацією та авторизацією користувачів, забезпечує безпечний обмін правами доступу. OAuth 2.0 - це протокол авторизації, який дозволяє стороннім застосункам отримувати обмежений доступ до ресурсів користувача, не передаючи при цьому їх облікові дані (наприклад, ім’я користувача та пароль). Цей стандарт використовується для управління доступом у різних веб-додатках, мобільних додатках і API.

- JWT (JSON Web Tokens) — це відкритий стандарт (RFC 7519) для безпечної передачі даних між клієнтом і сервером у формі об'єкта JSON, а також використовується для зберігання інформації про сеанс. Ідеально

підходить для мікросервісних архітектур. JWT дозволяє передавати дані у компактному форматі, який легко передавати через URL, HTTP заголовки або в JavaScript об'єкти. Використання JWT в поєднанні з протоколом OAuth 2.0 може суттєво підвищити безпеку системи.

2. Шифрування:

- TLS (Transport Layer Security) — забезпечення шифрування для комунікацій через мережу, включаючи HTTPS для веб-інтерфейсу та шифрування API;
- AES (Advanced Encryption Standard) — для шифрування даних, що зберігаються, зокрема документів і файлів.

Технологічний стек для безпеки в автоматизованій системі документообігу:

– аутентифікація та авторизація: OAuth 2.0, JWT;
– шифрування TLS, AES.

3.1.5. Моніторинг і логування

Моніторинг системи автоматизованого документообігу є важливим для забезпечення стабільної роботи, швидкого виявлення проблем та підтримки високої продуктивності, логування для виявлення помилок, аналізу поведінки системи та підтримки безпеки.

1. Моніторинг:

• Prometheus — це система моніторингу та збору метрик з відкритим вихідним кодом, розроблена для надійного збору та зберігання інформації про стан і продуктивність різних систем. Вона використовує модель даних, основану на часових рядах, що дозволяє зберігати метрики у вигляді пар «метрика» + «мітка». Prometheus підтримує запити через свій власний мова запитів (PromQL). Також підтримує автоматичне виявлення сервісів, що дозволяє йому адаптуватися до змін у середовищі, зокрема в контейнеризованих середовищах, таких як Kubernetes. Prometheus зберігає

дані у своїй власній базі даних, оптимізованій для швидкого запису та читання даних. Prometheus може використовуватися для збору метрик продуктивності та стану кожного мікросервісу в системі, підходить для збору статистики про виконання запитів, використання ресурсів та стан системи.

- Grafana — це платформа для візуалізації даних та моніторингу, яка дозволяє створювати інтерактивні дашборди та графіки для аналізу різноманітних метрик. Вона часто використовується у комбінації з системами збору даних, такими як Prometheus. Grafana підтримує безліч типів візуалізацій, таких як графіки, діаграми, таблиці, гістограми та багато інших. Це дозволяє користувачам адаптувати дашборди під свої потреби. Grafana підключається до різних джерел даних, включаючи Prometheus, MySQL, PostgreSQL та інші. Дашборди Grafana інтерактивні: користувачі можуть фільтрувати дані, налаштовувати часові інтервали, змінювати параметри запитів. Grafana дозволяє налаштовувати алерти на основі даних з джерел, що допомагає відслідковувати аномалії та отримувати сповіщення про критичні події в реальному часі. Вона має простий і зрозумілий інтерфейс користувача. Grafana використовується для візуалізації метрик продуктивності мікросервісів, таких як час відгуку, кількість запитів, навантаження на сервери тощо. З її допомогою легко аналізувати дані про обробку документів, зміни в системі, а також візуалізувати статистику використання. Візуалізація алертів і метрик допомагає виявляти потенційні проблеми на ранніх стадіях, що підвищує надійність системи. Grafana є невід'ємним інструментом для візуалізації даних у системі автоматизованого документообігу в банківських системах. Її можливості інтеграції з різними джерелами даних, а також гнучкість у налаштуванні дашбордів дозволяють командам оперативно відслідковувати стан системи, приймати зважені рішення та покращувати продуктивність.

2. Логування

ELK Stack — це набір потужних інструментів для збору, аналізу та візуалізації даних логів у реальному часі. Він складається з трьох основних

компонентів:

• Elasticsearch це потужний розподілений інструмент для пошуку та аналізу даних з великою швидкістю. Elasticsearch забезпечує повнотекстовий пошук, фільтрацію, агрегацію та аналітику в реальному часі. Він оптимізований для роботи з великими обсягами даних і підтримує різноманітні типи запитів.

• Logstash це інструмент для збору, обробки та передачі логів і даних з різних джерел. Logstash може отримувати дані з різних форматів (лог-файли, метрики, API) та трансформувати їх перед збереженням у Elasticsearch. Підтримує численні плагіни для інтеграції з різними джерелами та форматами.

• Kibana - це веб-інтерфейс для візуалізації даних, які зберігаються в Elasticsearch. Kibana дозволяє створювати дашборди, графіки, діаграми та інші візуалізації для аналізу даних. Інтерфейс користувача дозволяє зручно працювати з даними та отримувати аналітичну інформацію.
Використання ELK Stack у системі автоматизованого документообігу:

ELK Stack є важливим інструментом для моніторингу та логування в системах автоматизованого документообігу. Його можливості збору, аналізу та візуалізації даних дозволяють ефективно управляти інформацією, виявляти проблеми та підвищувати продуктивність системи. Застосування ELK Stack у банківських системах забезпечує високий рівень надійності та безпеки обробки документів.

Технологічний стек моніторингу та логування в автоматизованій системі документообігу:

– моніторинг стану системи: Prometheus, Grafana;
– ведення логів: ELK Stack.

3.1.6. Тестування

Використання різноманітних інструментів та фреймворків для тестування в системі автоматизованого документообігу забезпечить високу якість продукту, швидке виявлення помилок та відповідність вимогам безпеки. Технологічний стек тестування повинен враховувати специфіку банківської системи, забезпечуючи надійність і ефективність усіх етапів розробки [21].

1. Фреймворки: Jest — це популярний фреймворк для тестування JavaScript, розроблений компанією Facebook. Він є особливо зручним для тестування веб-додатків, побудованих на базі React, але також може використовуватися для тестування Node.js та інших JavaScript-додатків.

Jest має простий та інтуїтивно зрозумілий API. Він автоматично виявляє файли тестів, використовуючи специфічні імена (наприклад, файли з розширенням .test.js або .spec.js). Також підтримує тестування асинхронних функцій, що дозволяє писати тести для кодів, які виконуються з затримкою або у фоновому режимі. Jest підтримує тестування знімків, що дозволяє зберігати результати рендерингу компонентів і порівнювати їх у наступних запусках тестів. Має вбудовану можливість генерувати звіти про покриття коду, що дозволяє розробникам бачити, які частини коду були протестовані, а які — ні. Jest автоматично виконує тести в паралельних потоках, що скорочує час тестування.

Jest є потужним інструментом для тестування JavaScript-додатків, який забезпечує високу продуктивність і зручність використання. Його можливості, такі як автоматичне мокування, тестування знімків та звітність про покриття коду, роблять його відмінним вибором для системи автоматизованого документообігу в банківській сфері, де надійність та якість програмного забезпечення мають критичне значення.

2. Бібліотеки: Supertest — це популярна бібліотека для тестування HTTP-запитів у Node.js. Вона часто використовується разом із фреймворком

для тестування Jest, щоб забезпечити простий і зрозумілий спосіб перевірки API. Supertest дозволяє розробникам перевіряти коректність оброблених HTTP-запитів і відповідей повернутих з серверу. Бібліотека підтримує всі основні HTTP-методи, такі як GET, POST, PUT, DELETE, PATCH і інші, що робить її універсальним інструментом для тестування RESTful API. Supertest підтримує асинхронні запити.

3.2. Архітектура системи автоматизованого документообігу з використанням мікросервісів

Для візуалізація архітектури технічного стеку системи автоматизованого документообігу в банківській системі використаємо блок-схему, яка ілюструє різні компоненти, їх взаємозв'язки та технології, що застосовуються в кожному з цих компонентів (рис. 3.1).

Загальна структура схеми:

1. Клієнтська частина:
    - React (Redux) - ітерфейс користувача;
    - HTML/CSS (з Bootstrap) - розмітка та стилізація.
2. API Gateway:
    - Kong - управління запитами до мікросервісів.
3. Мікросервіси (Backend):
    - сервіс управління документами:
        - технології: Node.js (Express.js), JavaSpring;
    - сервіс електронного підпису:
        - технології: Node.js (Express.js), JavaSpring;
    - сервіс сповіщень:
        - технології: Node.js (з використанням RabbitMQ, Kafka);
    - сервіс аналітики та моніторингу:
        - технології: Prometheus, Grafana, ELK Stack.
4. Бази даних:

- PostgreSQL: зберігання метаданих документів;
- MongoDB: зберігання документів.

5. Безпека:
   - OAuth 2.0: авторизація та аутентифікація користувачів;
   - JWT: токени для безпечного доступу до ресурсів;
   - TLS, AES: шифрування.

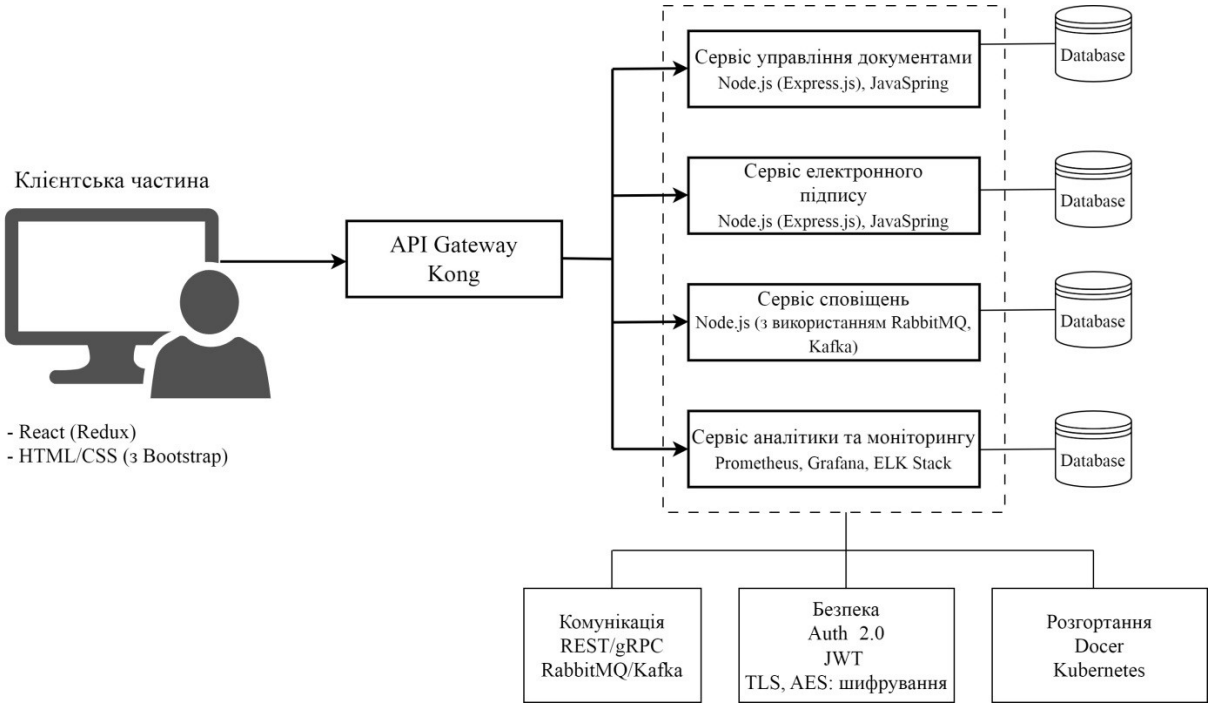

Рисунок 3.1 – Архітектура системи автоматизованого документообігу в банківській системі з використанням мікросервісів

3.3. Програмна реалізація системи автоматизованого документообігу

Розглянемо програмну реалізацію системи автоматизованого документообігу з використанням мікросервісів.

1. Сервіс управління документами. Цей мікросервіс надає основний функціонал для управління документами. Він дозволяє створювати нові документи та отримувати існуючі за їх ідентифікатором. Маршрути (/documents) підтримують операції створення документів за допомогою методу POST і отримання документів за допомогою GET. У реальному

застосуванні сервіс взаємодіятиме з базою даних, наприклад, PostgreSQL, для збереження та витягування даних документів. Цей варіант сервісу також дозволяє створювати і отримувати документи. Контролер DocumentController реалізує аналогічний до Node.js функціонал, де запити POST додають нові документи, а GET дозволяє отримати документ за його ідентифікатором. Spring Boot забезпечує структуру та засоби для інтеграції з базою даних для роботи з документами. Лістинги коду сервісу управління документами наведено на рис. 3.2 та 3.3.

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
app.use(bodyParser.json());

// Приклад маршруту для створення документа
app.post('/documents', (req, res) => {
    const documentData = req.body;
    // Логіка збереження документу у базу даних PostgreSQL
    res.status(201).send({ message: 'Документ створено', data: documentData });
});

// Приклад маршруту для отримання документа за ідентифікатором
app.get('/documents/:id', (req, res) => {
    const documentId = req.params.id;
    // Логіка отримання документу з бази даних PostgreSQL
    res.send({ id: documentId, data: {} }); // Повернення даних документу
});

app.listen(3001, () => {
    console.log('Document management service running on port 3001');
});
```

Рисунок 3.2 – Лістинг коду сервісу управління документами (app.js)

```
import org.springframework.web.bind.annotation.*;
import java.util.HashMap;
import java.util.Map;

@RestController
@RequestMapping("/documents")
public class DocumentController {

    @PostMapping
    public Map<String, String> createDocument(@RequestBody Map<String, Object> documentData) {
        // Логіка збереження документу у базу даних
        Map<String, String> response = new HashMap<>();
        response.put("message", "Документ створено");
        return response;
    }

    @GetMapping("/{id}")
    public Map<String, Object> getDocument(@PathVariable String id) {
        // Логіка отримання документу з бази даних
        Map<String, Object> document = new HashMap<>();
        document.put("id", id);
        document.put("data", new HashMap<>());
        return document;
    }
}
```

Рисунок 3.3 – Лістинг коду сервісу управління документами (DocumentController.java)

2. Сервіс електронного підпису. Сервіс електронного підпису приймає текст документу, генерує його криптографічний підпис і повертає разом з оригінальним документом. Підпис формується за допомогою хешування алгоритмом SHA-256, що дозволяє забезпечити унікальність підпису для кожного документа, гарантуючи його автентичність і цілісність. Цей варіант сервісу використовує Java та Spring Boot для генерування підписів. При надходженні запиту з документом, контролер SignatureController обчислює його хеш за допомогою SHA-256, формуючи унікальний підпис для кожного документа. Це допомагає забезпечити захист даних у процесі обміну документами. Лістинги коду сервісу електронного підпису наведено на рис. 3.4 та 3.5.

```
const express = require('express');
const crypto = require('crypto');
const app = express();
app.use(express.json());

app.post('/sign', (req, res) => {
    const { document } = req.body;
    const signature = crypto.createHash('sha256').update(document).digest('hex');
    res.send({ document, signature });
});

app.listen(3002, () => {
    console.log('Signature service running on port 3002');
});
```

Рисунок 3.4 – Лістинг коду сервісу електронного підпису (signatureService.js)

```
import org.springframework.web.bind.annotation.*;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.HashMap;
import java.util.Map;

@RestController
@RequestMapping("/sign")
public class SignatureController {

    @PostMapping
    public Map<String, String> signDocument(@RequestBody Map<String, String> request) throws NoSuchAlgorithmException {
        String document = request.get("document");
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] hash = md.digest(document.getBytes());
        StringBuilder hexString = new StringBuilder();
        for (byte b : hash) {
            hexString.append(Integer.toHexString(0xFF & b));
        }
        Map<String, String> response = new HashMap<>();
        response.put("signature", hexString.toString());
        return response;
    }
}
```

Рисунок 3.5 – Лістинг коду сервісу електронного підпису (SignatureController.java)

3. Сервіс сповіщень. Сервіс сповіщень використовує RabbitMQ як брокера повідомлень для обробки сповіщень. Основна функція sendNotification встановлює з'єднання з RabbitMQ, створює чергу notifications, куди надсилає сповіщення, а потім закриває з'єднання. Цей сервіс обробляє вхідні повідомлення (/notify) і надсилає їх до черги для подальшої обробки або доставки користувачам.

```
const amqp = require('amqplib');
const express = require('express');
const app = express();

async function sendNotification(message) {
    const connection = await amqp.connect('amqp://localhost');
    const channel = await connection.createChannel();
    const queue = 'notifications';

    await channel.assertQueue(queue, { durable: false });
    channel.sendToQueue(queue, Buffer.from(message));
    console.log("Sent:", message);

    setTimeout(() => {
        connection.close();
    }, 500);
}

app.post('/notify', (req, res) => {
    const { message } = req.body;
    sendNotification(message).then(() => {
        res.send({ status: 'Notification sent', message });
    });
});

app.listen(3003, () => {
    console.log('Notification service running on port 3003');
});
```

Рисунок 3.6 – Лістинг коду сервісу сповіщень (notificationService.js)

4. Сервіс аналітики та моніторингу. Конфігурація на основі Docker Compose забезпечує спільну роботу Prometheus та Grafana. Prometheus збирає метрики з додатків або системи (в цьому випадку через експортер node-exporter), а Grafana візуалізує ці метрики для моніторингу в реальному часі. Файл docker-compose.yml описує контейнери для обох сервісів і налаштовує порти: Prometheus доступний на 9090, а Grafana на 3000.

```yaml
version: '3'

services:
  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"
    depends_on:
      - prometheus
```

Рисунок 3.7 – Лістинг коду сервісу аналітики та моніторингу (docker-compose.yml)

Конфігураційний файл Prometheus (prometheus.yml) описує, як Prometheus збирає метрики. У ньому встановлено інтервал збору даних (scrape_interval) та цільове налаштування для збирання метрик з localhost:9100. Це дозволяє Prometheus відстежувати показники продуктивності і доступності сервісів.

```yaml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'node-exporter'
    static_configs:
      - targets: ['localhost:9100']
```

Рисунок 3.8 – Лістинг коду конфігураційного файлу (prometheus.yml)

5. Конфігурація безпеки. Сервіс аутентифікації реалізований з використанням JSON Web Tokens (JWT), що забезпечує безпеку і зручну авторизацію. При логіні користувач отримує токен доступу, який може бути використаний для подальших запитів до захищених ресурсів. Секретний ключ (supersecretkey) використовується для шифрування токенів, що забезпечує їхню безпечність. Запити до захищених ресурсів (/protected) перевіряються на наявність та коректність токена, що дозволяє надавати

доступ лише авторизованим користувачам.

```
const express = require('express');
const jwt = require('jsonwebtoken');
const app = express();
app.use(express.json());

const secretKey = 'supersecretkey';

app.post('/login', (req, res) => {
    const { username } = req.body;
    const token = jwt.sign({ username }, secretKey, { expiresIn: '1h' });
    res.send({ token });
});

app.get('/protected', (req, res) => {
    const token = req.headers['authorization'];
    if (token) {
        jwt.verify(token, secretKey, (err, decoded) => {
            if (err) return res.sendStatus(403);
            res.send({ message: 'Access granted', user: decoded });
        });
    } else {
        res.sendStatus(401);
    }
});

app.listen(3004, () => {
    console.log('Authentication service running on port 3004');
});
```

Рисунок 3.9 – Лістинг коду файлу (authService.js)

Ця система мікросервісів забезпечує гнучке управління документами, зручність використання електронного підпису, відправку сповіщень, моніторинг продуктивності та безпеку доступу до сервісів.

ВИСНОВКИ

У кваліфікаційній роботі було досліджено питання можливості застосування мікросервісної архітектури для побудови системи автоматизованого документообігу в банківській сфері. З’ясовано, що мікросервісний підхід суттєво підвищує гнучкість, масштабованість і надійність систем автоматизованого документообігу, дозволяючи ефективно відповідати на вимоги сучасних банківських установ, такі як збільшення обсягів даних, зростання транзакційної активності, потреби у високій доступності та стабільності.

Використання мікросервісів забезпечує незалежне розгортання та оновлення компонентів системи, що спрощує процеси підтримки та розвитку. Це особливо актуально для великих організацій, де збої та простої систем можуть призвести до значних фінансових втрат та негативного впливу на репутацію. Також, завдяки розподілу системи на окремі сервіси, можна легко інтегрувати нові функціональні модулі або адаптувати систему до змін у законодавчих чи регуляторних вимогах.

Система електронного документообігу на основі мікросервісів показала високий потенціал щодо забезпечення ефективного управління документами, автоматизації процесів їх обробки, а також захисту і контролю доступу. Розглянута архітектура використовує сучасні технології для кожного з мікросервісів: управління документами, підпис документів, сповіщення, аналітика та моніторинг. Завдяки цьому забезпечується високий рівень продуктивності, захищеність даних, а також можливість масштабування системи відповідно до зростаючих вимог.

Практична значущість дослідження полягає у розробці рекомендацій та принципів побудови систем автоматизованого документообігу, що можуть бути використані як у банківському секторі, так і в інших галузях з високими вимогами до обробки та зберігання великих обсягів даних. Перспективи подальших досліджень можуть охоплювати оптимізацію взаємодії між

мікросервісами, покращення безпеки та захищеності даних, а також інтеграцію з іншими фінансовими системами.

Загалом, проведене дослідження підтверджує, що мікросервісна архітектура є ефективним та надійним підходом для реалізації сучасних систем автоматизованого документообігу в банківській сфері, сприяючи підвищенню продуктивності та зменшенню ризиків втрати інформації.

ПЕРЕЛІК ПОСИЛАНЬ

1. Карпенко М. Ю. Системи електронного документообігу : конспект лекцій для студентів усіх форм навчання першого (бакалаврського) рівня вищої освіти спеціальності 122 – Комп’ютерні науки; Харківський національний університет міського господарства ім. О. М. Бекетова. Харків : ХНУМГ ім. О. М. Бекетова, 2021. – 68 с.

2. Грицяк Н.В. Електронний документообіг та захист інформації: навчальний посібник. Київ : НАДУ, 2015. – 84 с.

3. Золотарьова І. О., Бутова Р. К. Автоматизація документообігу. Навчальний посібник / І. О. Золотарьова, Р. К. Бутова. – Харків: Вид. ХНЕУ, 2008. – 169 с.

4. Microservices vs Monolithic. [Електроний ресурс]. – Режим доступу: https://ncube.com/blog/microservices-vs-monolithic-which-architecturesuits-best-for-your-project.

5. Мікросервіси. [Електроний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/%D0%9C%D1%96%D0%BA%D1%80%D0%BE%D1%81%D0%B5%D1%80%D0%B2%D1%96%D1%81%D0%B8.

6. Клапчук Р. Г. Харченко В. С. Монолітні веб-сервіси та мікросервіси: порівняння та вибір. Радіоелектронні і комп’ютерні системи. - 2017. - № 1. - С. 51–56. - Режим доступу: http://nbuv.gov.ua/UJRN/recs_2017_1_8.

7. Чапля О. Ю., Клим Г. І. Мікросервісна архітектура для кіберфізичних систем. Вісник Херсонського національного технічного університету, № 2(89), 2024. – С. 242-250.

8. Лавріщева К.М. Програмна інженерія.–Київ, 2008. – 319 с.

9. Табунщик Г. В., Каплієнко Т. І., Петрова О. А. Проектування та моделювання програмного забезпечення сучасних інформаційних систем. − Запоріжжя : Дике Поле, 2016. – 250 с.

10. Apache Kafka. [Електроний ресурс]. – Режим доступу:

https://kafka.apache.org/.

11. Amazon API Gateway. [Електроний ресурс]. – Режим доступу: https://aws.amazon.com/api-gateway/?nc1=h_ls

12. JavaSpring. [Електроний ресурс]. – Режим доступу:https://spring.io/.

13. Node.js. [Електроний ресурс]. – Режим доступу: https://nodejs.org/en.

14. React JavaScript-бібліотека для створення користувацьких інтерфейсів. [Електроний ресурс]. – Режим доступу: https://uk.legacy.reactjs.org/.

15. PostgreSQL: The World's Most Advanced Open Source Relational Database. [Електроний ресурс]. – Режим доступу: https://www.postgresql.org/.

16. MongoDB: The Developer Data Platform. [Електроний ресурс]. – Режим доступу: https://www.mongodb.com/.

17. Docker: Accelerated Container Application Development. [Електроний ресурс]. – Режим доступу: https://www.docker.com/.

18. Docker. [Електроний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Docker.

19. Kubernetes. [Електроний ресурс]. – Режим доступу: https://kubernetes.io/.

20. Kong. [Електроний ресурс]. – Режим доступу: https://konghq.com/.

21. Авраменко А.С., Авраменко В.С., Косенюк Г.В. Тестування програмного забезпечення. Навчальний посібник. Черкаси: ЧНУ імені Богдана Хмельницького, 2017. – 284 с.