Міністерство освіти і науки України
Харківський національний університет радіоелектроніки
Факультет _Інформаційних радіотехнологій і технічного захисту інформації____
(повна назва)
Кафедра _____Радіотехнологій інформаційно-комунікаційних систем_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____другий (магістерський)_____

ПРОЕКТУВАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ СИСТЕМИ ОБЛІКУ ПАЦІЄНТІВ МЕДИЧНОЇ ЛАБОРАТОРІЇ
(тема)

Виконав:
студент II курсу, групи АПСм -22-1
Братищенко Т.С.
(прізвище, ініціали)

Спеціальність
126 Інформаційні системи та технології
(код і повна назва спеціальності)

Тип програми
освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Архітектурне проєктування інформаційних систем
(повна назва освітньої програми)

Керівник доц. Сайківська Л. Ф.
(посада, прізвище, ініціали)

Допускається до захисту
В.о. зав. кафедри РТІКС _____ Зарудний О.А.
(підпис) (прізвище, ініціали)

2024 р.

Не містить відомостей, заборонених
до відкритого публікування

Керівник _____ Братищенко Т.С.

Студент _____ Сайківська Л.Ф.

Харківський національний університет радіоелектроніки

Факультет_Інформаційних радіо технологій і технічного захисту інформації
Кафедра_Радіотехнологій інформаційно-комунікаційних систем
Рівень вищої освіти_____другий (магістерський)_____
Спеціальність_____126 Інформаційні системи та технології
(код і повна назва)
Освітня програма Архітектурне проєктування інформаційних систем
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)
«____»_____ 2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту_____Братищенку Тарасу Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи " Проектування мікросервісної архітектури системи обліку пацієнтів медичної лабораторії "_____
затверджена наказом університету від _03_11____2023 р. №_1295Ст_____

2. Термін подання студентом роботи до екзаменаційної комісії _10_січня_ 2024 р.

3. Вихідні дані до роботи: _ побудувати інформаційну систему за методологією мікросервісної архітектури та брокера повідомлень
_____

4. Перелік питань, що потрібно опрацювати в роботі _____
Вступ
1. Аналіз проблеми обліку пацієнтів медичної лабораторії
2.Аналіз методів проектування мікросервісної архітектури
3.Проектування мікросервісної архітектури системи обліку
4.Імплементація системи обліку пацієнтів
Висновки
_____

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)

Слайди презентації

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
	Ознайомлення зі завданням. Уточнення ТЗ	04.10.2023	вик.
	Підбір літератури за темою роботи	18.10.2023-30.10.2023	вик.
	Написання першого, другого розділів	03.11.2023-01.12.2023	вик.
	Написання третього, четвертого розділів	03.12.2023-30.12.2023	вик.
	Оформлення записки	10.12.2023-15.12.2023	вик.
	Оформлення презентаційного матеріалу, підготовка	15.12.2023 – 10.01.2024	вик
	до захисту у ЕК		

Дата видачі завдання 4 жовтня 2023 р.

Студент _____ Братищенко Т.С.
(підпис) (прізвище, ініціали)

Керівник роботи _____ доц. Сайківська Л.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Кваліфікаційна робота магістра містить 62 сторінки тексту, 12 рисунків, 11 джерел посилання та 2 додатки.

МІКРОСЕРВІСНА АРХІТЕКТУРА, БАЗИ ДАНИХ, РОЗПОДІЛЕННЯ НАВАНТАЖЕННЯ, СИСТЕМА ОБЛІКУ

Предмет проектування – Архітектура системи обліку пацієнтів медичної лабораторії

Мета роботи - опис та розробка мікросервісної архітектури для системи обліку пацієнтів в медичній лабораторії з метою покращення ефективності та функціональності цієї системи.

Основні цілі включають наступне:

1. Покращення доступності медичних даних
2. Підвищення швидкості обробки заявок
3. Забезпечення безпеки та конфіденційності
4. Можливість масштабування та розвитку
5. Зменшення залежності від одного центрального компонента
6. Зменшення витрат та підвищення продуктивності

Загальна мета полягає в створенні високоефективної, масштабованої та безпечної системи для обліку пацієнтів в медичній лабораторії, що відповідає вимогам сучасної медицинської практики та сприяє покращенню обслуговування пацієнтів.

ABSTRACT

The master's thesis contains 62 pages of text, 12 figures, 11 references and 2 appendixes.

MICROSERVICE ARCHITECTURE, DATABASES, LOAD BALANCING, ACCOUNTING SYSTEM

Design subject - Architecture of the patient accounting system of a medical laboratory

The purpose of the work is to describe and develop a microservice architecture for a patient accounting system in a medical laboratory in order to improve the efficiency and functionality of this system.

The main objectives include the following:

1. Improving the availability of medical data
2. Increase the speed of application processing
3. Ensuring security and confidentiality
4. Ability to scale and develop
5. Reducing dependence on one central component
6. Reduced costs and increased productivity

The overall goal is to create a highly efficient, scalable, and secure system for patient accounting in a medical laboratory that meets the requirements of modern medical practice and contributes to the improvement of patient care.

ЗМІСТ

ВСТУП

Сучасна медицина стикається з високими вимогами щодо обліку та збереження медичної інформації пацієнтів, а також потребує ефективних інструментів для її обробки та аналізу. Мета роботи проекті полягає в тому, щоб відповісти на виклики суспільства та покращити процеси в медичних установах шляхом впровадження мікросервісної архітектури для системи обліку пацієнтів в медичній лабораторії.

Проект спрямований на впровадження ефективних інструментів обробки та аналізу даних пацієнтів. У сучасному світі, де технології активно впливають на розвиток медичної науки, виникає необхідність у нових підходах до структурування та управління цією інформацією.

Мікросервісна архітектура вирішує ці завдання. Замість великих, монолітних систем, мікросервіси дозволяють розділити функціональність на компактні, автономні модулі, що полегшує розгортання, масштабування та підтримку системи в цілому. Цей підхід створює гнучку та динамічну інфраструктуру, здатну адаптуватися до швидкозмінюваних вимог сучасного медичного середовища.

Відзначаючи цільові переваги та перспективи мікросервісної архітектури для систем обліку пацієнтів, даний проект пропонує аналіз та проектування такої системи з використанням методологій розробки мікросервісної архітектури. Завдяки цьому, проект має за мету внести вклад у розвиток сфери медичних технологій та поліпшення надання медичних послуг для пацієнтів.

1 АНАЛІЗ ПРОБЛЕМИ ОБЛІКУ ПАЦІЄНТІВ МЕДИЧНОЇ ЛАБОРАТОРІЇ

1.1 Огляд сучасної системи обліку пацієнтів в медичній лабораторії

На сьогоднішній день медичні лабораторії грають важливу роль у наданні медичних послуг та діагностиці різних захворювань. Ці установи здійснюють обширний облік медичних даних, включаючи інформацію про пацієнтів, результати аналізів, призначення, історію лікування та багато іншої важливої інформації.

Однак багато сучасних систем обліку пацієнтів мають ряд обмежень і недоліків. Серед них:

Обмежена інтеграція: багато зі сучасних систем обліку пацієнтів працюють в ізоляції від інших медичних систем, що ускладнює обмін даними та унеможливлює їхню швидку та ефективну інтеграцію з іншими медичними установами та системами.

Складність доступу до даних: зазвичай доступ до медичних даних обмежений, адже збереження та обробка цих даних пов'язана з рядом законодавчих та етичних обмежень. Це може ускладнювати обмін інформацією та отримання доступу до неї для авторизованих користувачів.

Неефективність та затримки: традиційні системи можуть бути повільними та неефективними в обробці даних та видачі результатів аналізів, що може призводити до затримок у діагностиці та лікуванні пацієнтів.

Брак стандартизації даних: дані, що зберігаються в різних системах, часто не відповідають єдиної стандартизованій специфікації, що ускладнює їхню обробку та обмін між різними установами.

Огляд сучасної системи обліку пацієнтів в медичній лабораторії підкреслює необхідність вдосконалення та модернізації існуючих підходів до обробки медичних даних. Використання мікросервісної архітектури разом із специфікацією FHIR та сучасними технологіями дозволить подолати ці

обмеження та покращити ефективність та доступність обліку пацієнтів у медичних лабораторіях.

1.2 Визначення потреб стейкхолдерів

У процесі аналізу проекту, було важливо визначити потреби користувачів та стейкхолдерів системи обліку пацієнтів в медичній лабораторії. Це допомагає забезпечити те, що розроблена система відповідає їхнім потребам та очікуванням. Основні стейкхолдери та їх потреби включають:

Лікарі та медичний персонал мають потребу в швидкому доступі до медичної інформації пацієнтів, результатів аналізів, а також можливості внесення та оновлення медичних даних. Вони також очікують зручного інтерфейсу та можливості миттєвого спілкування з іншими членами медичного персоналу.

Пацієнти мають потребу в зручному доступі до своїх медичних даних, реєстрації на аналізи та отриманні результатів досліджень. Важливо, щоб система була легкою у використанні та забезпечувала конфіденційність їхніх особистих даних.

Адміністратори медичної лабораторії мають потребу в ефективному управлінні ресурсами, контролі над доступом до даних та можливості моніторингу та аналізу роботи системи.

Управління медичної лабораторії має потребу в звітах та аналітиці діяльності лабораторії, можливості планування та координації роботи персоналу, а також взаємодії з іншими медичними установами та органами регулювання.

Стейкхолдери з питань безпеки даних: це можуть бути представники органів, що регулюють обробку медичних даних, а також аудитори, які перевіряють відповідність системи стандартам безпеки та конфіденційності даних. Вони мають потребу в забезпеченні безпеки та відповідності до вимог законодавства.

Визначення цих потреб дозволяє забезпечити, що розроблена система повинна відповідати вимогам та очікуванням різних категорій користувачів та стейкхолдерів, а також покращити якість обслуговування пацієнтів та робочий процес медичних установ.

1.3 Аналіз та оцінка технічних та функціональних вимог до системи

Для успішного проектування та реалізації мікросервісної архітектури системи обліку пацієнтів в медичній лабораторії важливо детально проаналізувати та оцінити технічні та функціональні вимоги. Цей процес допомагає визначити, як система має функціонувати, які можливості вона має надавати та які технічні аспекти необхідно враховувати. Основні вимоги включають наступне:

Збереження та обробка медичних даних: система повинна надійно зберігати медичні дані пацієнтів, включаючи аналізи, історію лікування, діагнози та інші важливі дані. Дані мають бути доступні для обробки та аналізу.

Інтеграція з іншими медичними системами: система повинна бути здатна інтегруватися з іншими медичними системами, такими як Електронні Медичні Документи (EMR), системи призначень, системи реєстрації пацієнтів та інші, для забезпечення надійного обміну даними.

Автентифікація та авторизація: система повинна забезпечувати безпечний доступ до даних, використовуючи механізми автентифікації та авторизації для ідентифікації та контролю доступу користувачів.

Масштабованість: система повинна бути легко масштабованою, щоб забезпечити швидку обробку даних та підтримувати рост використання.

Швидкість та продуктивність: система повинна працювати швидко та ефективно, надавати результати аналізів пацієнтів у найкоротший термін та забезпечувати високу продуктивність.

Захист даних та конфіденційність: система повинна гарантувати безпеку та конфіденційність медичних даних, забезпечуючи відповідність до вимог законодавства щодо обробки особистих даних в медицині.

Можливість аналізу та звітності: система повинна надавати інструменти для аналізу даних та створення звітів для лікарів, адміністраторів та інших стейкхолдерів.

Можливість розширення та підтримки: система повинна бути легко розширюваною та підтримувати оновлення та виправлення помилок.

Аналіз та оцінка цих технічних та функціональних вимог допомагає визначити необхідність вибору конкретних технологій, архітектури та стратегій розробки для вдалого втілення проекту та забезпечення задоволення потреб користувачів та стейкхолдерів.

1.4 Огляд існуючих рішень

У контексті систем обліку пацієнтів у медичних лабораторіях широко використовується монолітна архітектура. Основним принципом цієї архітектури є об'єднання всіх компонентів системи в один нероздільний блок. Такий підхід передбачає існування єдиної централізованої бази даних, що містить інформацію про пацієнтів, їх медичну історію та інші пов'язані дані. Монолітна система обліку пацієнтів включає в себе єдиний додаток, що виконує всі функції, пов'язані з обліком пацієнтів. Вона також може включати різні інтегровані модулі, такі як модуль прийому пацієнтів, лабораторний модуль та фінансовий модуль, об'єднані в рамках цього додатку.

Щодо переваг, монолітна архітектура має простоту у розробці та розгортанні, а також одну центральну точку контролю за базою даних, що сприяє виявленню та усуненню помилок.

Монолітні системи обліку пацієнтів у медичних лабораторіях, хоча і мають свої переваги, але також супроводжуються рядом значущих недоліків, які можуть впливати на їх ефективність та можливість адаптації:

Складність масштабування: монолітні системи часто зіткнюються з викликом масштабування, оскільки збільшення обсягу даних або трафіку може вимагати глобальних змін у всій системі. Це може призводити до збільшених витрат та складнощів у внесенні змін.

Ризик "одного пункту відмови": у випадку великого, комплексного монолітного додатку, існує ризик, що відмова в одному з його компонентів може призвести до відмови всієї системи. Це створює значний ризик для надійності та доступності системи.

Затримки в розробці та впровадженні нових функцій: внесення змін або додавання нових функцій в монолітні системи може бути складним та затриманим процесом. Це пов'язано з тим, що будь-яка зміна може мати далекосяжні наслідки і вимагати великого обсягу тестування та перевірок.

Витрати на інфраструктуру: з великою кількістю функціональності в єдиному додатку, інфраструктура монолітних систем може вимагати значних ресурсів. Це може призводити до неправомірно великого розміру серверів та інших витрат на обслуговування.

Обмежена гнучкість та швидкість вдосконалення: через монолітну структуру, впровадження інновацій та вдосконалень може бути обмеженим, оскільки це вимагає змін в цілій системі. Внаслідок цього система може бути менш гнучкою та повільною у вдосконаленні порівняно з більш сучасними архітектурними підходами.

Урахування цих недоліків важливо при розгляді архітектурних рішень для систем обліку пацієнтів, особливо в умовах зростаючих вимог до швидкості, гнучкості та масштабованості в медичній сфері. Тенденцією в розробці систем обліку пацієнтів є перехід від монолітної архітектури до мікросервісної архітектури, більш гнучкої та піддатливої до масштабування. Такий підхід дозволяє краще враховувати ці критичні вимоги, що є критичним у сучасному середовищі медичних лабораторій.

1.5 Постановка задачі

У сучасних умовах охорони здоров'я, створення системи обліку пацієнтів для медичної лабораторії, базованої на мікросервісній архітектурі, є актуальним завданням. Високий обсяг медичної інформації, стрімкий розвиток технологій та постійні зміни вимог до доступності та обробки даних створюють потребу в нових, більш ефективних підходах до систем обліку пацієнтів.

Враховуючи вказані тенденції, параметри системи обліку пацієнтів медичної лабораторії, розробленої на основі мікросервісної архітектури, визначаються такими як масштабованість, гнучкість та модульність. Важливим аспектом є забезпечення ефективного масштабування системи для вирішення зростаючих обсягів обробки даних та числа користувачів. Також, система повинна відзначатися високою надійністю та можливістю швидкого відновлення після відмов для забезпечення безперебійності обслуговування пацієнтів. Забезпечення безпеки та ефективного керування доступом до медичної інформації вважається важливим врахуванням.

Однак, така система повинна також забезпечувати швидку обробку даних та відповіді на запити, що стає ключовим аспектом для задоволення потреб користувачів та медичного персоналу. Крім того, інтеграція з іншими медичними системами, такими як Інформаційні системи лікарень та Електронні медичні картки, є невід'ємною частиною функціоналу системи для забезпечення єдиного електронного звіту про стан пацієнтів.

Отже, створення системи обліку пацієнтів медичної лабораторії, заснованої на мікросервісній архітектурі, враховує вищезгадані параметри для досягнення оптимальної ефективності, надійності та гнучкості в умовах сучасних вимог до охорони здоров'я.

2 АНАЛІЗ МЕТОДІВ ПРОЕКТУВАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1 Опис процесу проектування мікросервісної архітектури

В основі підходу проектування мікросервісної архітектури лежить розбиття системи на невеликі автономні компоненти, які працюють разом, надаючи гнучкість та масштабованість програмному забезпеченню. Мікросервісна архітектура є сучасним підходом до розробки програмного забезпечення, що відрізняється від традиційних монолітних структур [1]. Цей метод розглядає програму як набір невеликих та автономних сервісів, які взаємодіють між собою, надаючи більшу гнучкість та швидкість в розробці та масштабуванні програмних продуктів.

На шляху до ефективної мікросервісної архітектури, кожен етап грає важливу роль у створенні стійкої та ефективної системи. Від аналізу вимог і стратегій планування до масштабування, стабільності та безпеки - ці аспекти не лише формують технічний фундамент, але і визначають успіх у реальних умовах експлуатації.

Процес проектування мікросервісної архітектури розпочинається з детального аналізу вимог до системи. Розробники вивчають бізнес-потреби та ідентифікують функції кожного мікросервісу, а також взаємодії між ними. Цей етап є ключовим для подальшої побудови оптимальної архітектури, оскільки правильне визначення вимог визначає успішність системи.

На другому етапі проектування мікросервісної архітектури відбувається детальний аналіз вимог до системи. Розробники вивчають функціональні та нефункціональні вимоги, враховуючи специфічні потреби бізнесу та користувачів. Зазначення чітких вимог є ключовим етапом, оскільки від цього залежить подальше планування та розподіл мікросервісів.

Для ефективного розподілу функцій між мікросервісами використовуються стратегії планування. Це включає в себе визначення границь

кожного сервісу, його відповідальності та інтерфейсів. Важливо також уникати зайвої залежності між сервісами, щоб забезпечити їхню автономність. Детальний аналіз вимог та розробка стратегій планування становлять основу для подальшого створення масштабованої та гнучкої мікросервісної архітектури.

Після аналізу вимог та стратегій планування, наступним кроком є визначення структури мікросервісів. Кожен сервіс повинен бути самодостатнім, виконуючи конкретну функцію або послугу. Важливо правильно розділити функціональність, щоб сервіси були невеликими та максимально автономними.

Визначення границь між мікросервісами та їх взаємодія визначаються на цьому етапі. Забезпечення чіткості та зручності комунікації між сервісами важливо для забезпечення ефективної роботи системи в цілому. Кожен мікросервіс може розглядатись як окремий компонент, який може бути незалежно розвиватись, масштабоватись та оновлюватись.

Визначення структури мікросервісів є стратегічним етапом, оскільки від цього залежить подальше розгортання системи та забезпечення її легкості управління. (Рис. 2.1)

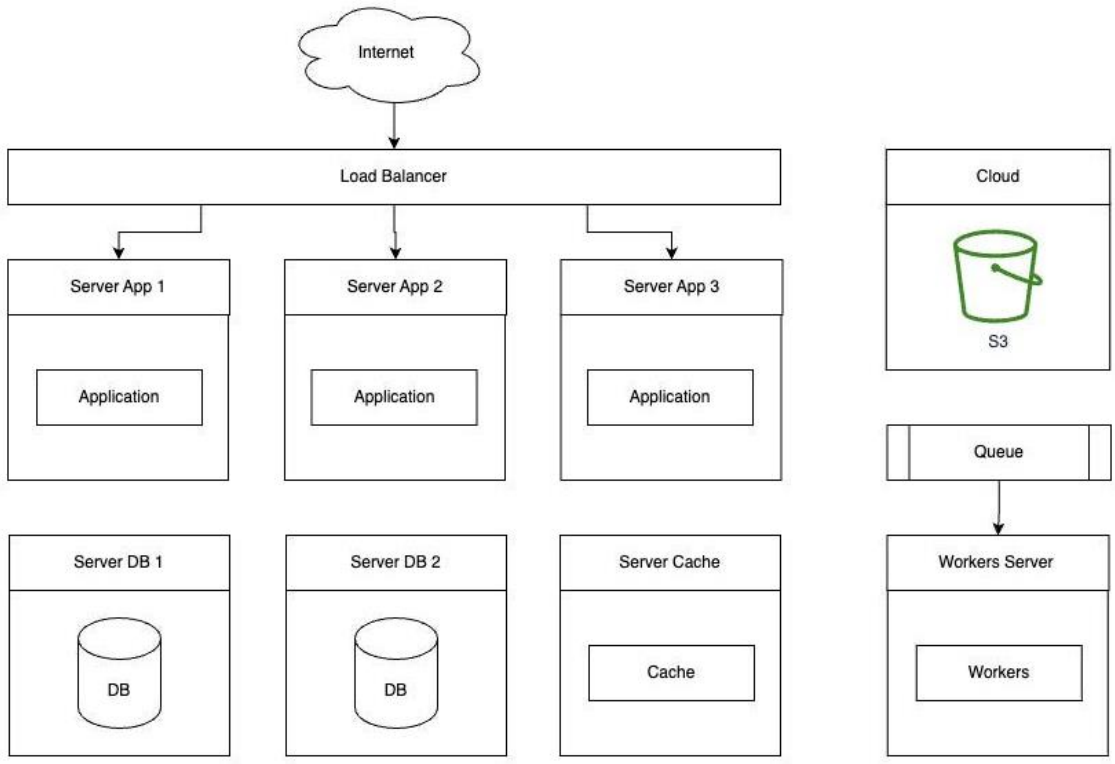


Рисунок 2.1 - Структура мікросервісів [2]

Важливим аспектом мікросервісної архітектури є забезпечення ефективної взаємодії між окремими сервісами. Для цього використовуються різні механізми комунікації та протоколи. RESTful API, міжслужбова шина чи пряма синхронна взаємодія - кожен варіант має свої переваги та обмеження. (Рис. 2.2)

Важливо розробити чіткі інтерфейси для взаємодії між сервісами, щоб забезпечити прозорість та стабільність системи. Використання стандартів та документація для API допомагають розробникам легко розуміти, як взаємодіяти з кожним сервісом.

Процес взаємодії є ключовим в архітектурі, оскільки від цього залежить загальна ефективність та надійність системи.

Не менш важливим аспектом є масштабування мікросервісної архітектури: це дозволяє системі ефективно працювати при збільшенні обсягів роботи. Кожен мікросервіс може бути масштабований незалежно в залежності від його завдань та навантаження.

Важливо визначити стратегії масштабування, такі як вертикальне та горизонтальне масштабування. Вертикальне масштабування включає в себе збільшення ресурсів для окремого сервісу, тоді як горизонтальне масштабування передбачає додавання нових екземплярів сервісу. (Рис. 2.3)

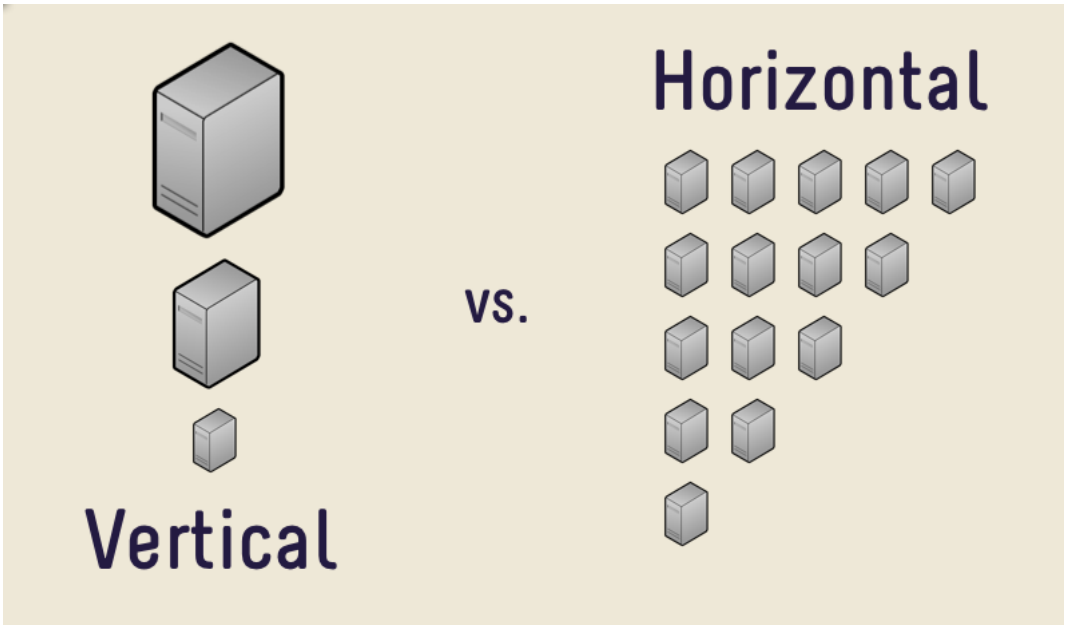


Рисунок 2.2 - Вертикальне та горизонтальне масштабування [3]

Застосування сучасних технологій контейнеризації, таких як Docker, та оркестраційних систем, таких як Kubernetes, робить процес масштабування більш простим та ефективним.

Забезпечення стабільності мікросервісної архітектури - це важливий етап в процесі проектування. Врахування високих навантажень, виявлення та усунення проблем вчасно є ключовими аспектами.

Використання моніторингу, журналювання та інструментів трасування дозволяє оперативно виявляти та вирішувати проблеми у роботі сервісів. Важливо також мати механізми відновлення та автоматичного виправлення помилок для забезпечення безперебійної роботи системи.

Приділення уваги аспектам стабільності включає в себе планування резервних копій, забезпечення резервних рішень та планування регулярних тестувань стійкості системи до великого обсягу трафіку.

Виділення аспектів безпеки є важливою частиною проектування мікросервісної архітектури. Забезпечення захисту даних, виявлення та відповідь на потенційні загрози та використання ефективних методів аутентифікації та авторизації - це невід'ємні елементи безпеки. Важливо застосовувати шифрування для передачі чутливої інформації між мікросервісами та здійснювати регулярні аудити безпеки для виявлення та усунення можливих уразливостей.

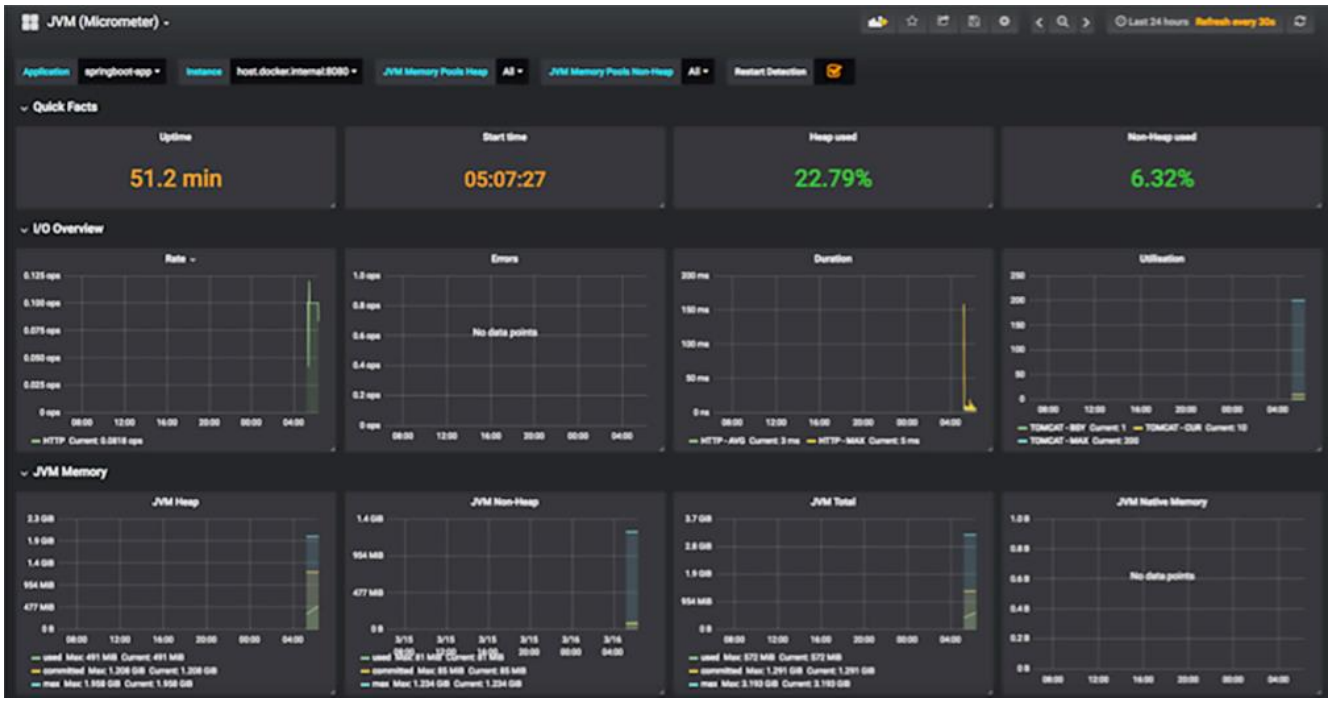
Імплементація практик DevSecOps [4], яка включає безпеку у всі етапи розробки, робить процес забезпечення безпеки більш ефективним та інтегрованим.

Тестування в мікросервісній архітектурі є критичним етапом для забезпечення високої якості та надійності системи. Кожен сервіс повинен бути протестований на функціональність, взаємодію та витривалість Автоматизоване тестування, таке як юніт-тести, інтеграційне та завдання витривалість допомагають виявляти помилки на ранніх етапах розробки та забезпечують швидкий цикл виправлення. Тестування безпеки, включаючи тестування на проникнення, є важливим компонентом для виявлення потенційних

уразливостей та забезпечення безпеки системи. Ефективне управління версіями є важливим аспектом мікросервісної архітектури. Зміни в коді кожного сервісу повинні супроводжуватися чітким управлінням версіями, щоб уникнути конфліктів та забезпечити сумісність.

Використання систем контролю версій, таких як Git, дозволяє відслідковувати зміни, робити злиття та легко відкатуватися до попередніх версій в разі потреби Застосування семантичного версіювання та визначення стабільних API-інтерфейсів допомагає розробникам та іншим сервісам адаптуватися до змін без проблем. Забезпечення доступності, продуктивності та виявлення проблем в реальному часі дозволяє швидко реагувати на можливі неполадки та забезпечити стабільну роботу системи.

Використання інструментів моніторингу, таких як Prometheus чи Grafana,



дозволяє відслідковувати ключові метрики, а також сповіщати про події та аномалії.

Логування та аналіз журналів допомагає в ідентифікації причин проблем та відновленні нормального функціонування системи. Етап розгортання є критичним моментом у життєвому циклі мікросервісної архітектури.

Рисунок 2.3 - Інтерфейс інструменту Grafana [5]

Забезпечення ефективного та безперебійного розгортання нових версій сервісів допомагає швидше впроваджувати зміни та оновлення.

Використання інструментів автоматизації розгортання, таких як Docker Compose чи Kubernetes, спрощує управління контейнерами та оркестрацію. Стратегії розгортання, такі як blue-green та canary deployments, дозволяють впроваджувати новий функціонал без впливу на загальну стабільність системи.

Останній етап - це етап підтримки та майбутнього розвитку мікросервісної архітектури. Підтримка включає в себе постійний моніторинг, аналіз та вирішення проблем, а також підтримку безперебійної роботи системи.

Щоб забезпечити майбутній розвиток, розробники мають вивчати відгуки користувачів та слідкувати за новими технологіями. Регулярне оновлення мікросервісів та їхніх залежностей дозволяє системі залишатися сучасною та безпечною.

2.2 Оцінка ефективності мікросервісної архітектури

Система на базі мікросервісів може легко збільшувати чи зменшувати обсяги ресурсів для кожного сервісу окремо, що сприяє оптимізації та ефективному використанню інфраструктури.

Оцінка ефективності починається з усвідомлення та визначення, наскільки добре мікросервісна архітектура може використовувати ці переваги для досягнення більшої продуктивності та швидкодії. Хоча мікросервісна архітектура надає численні переваги, її ефективність часто стикається з рядом викликів. Одним із найзначущих є питання безпеки. Розбиття системи на невеликі компоненти може створити додаткові точки для атак та збільшити складність управління доступом.

Забезпечення консистентності та надійності в умовах розподіленості також може бути складним завданням. Переходячи від монолітної архітектури до мікросервісів, необхідно сфокусувати увагу на механізми синхронізації та взаємодії між сервісами. Збільшення кількості сервісів може призвести до

складності управління та розгортанням: кожен мікросервіс має власну кодову базу, залежності та інфраструктуру, що може ускладнити розуміння та моніторинг. (Рис. 2.4)

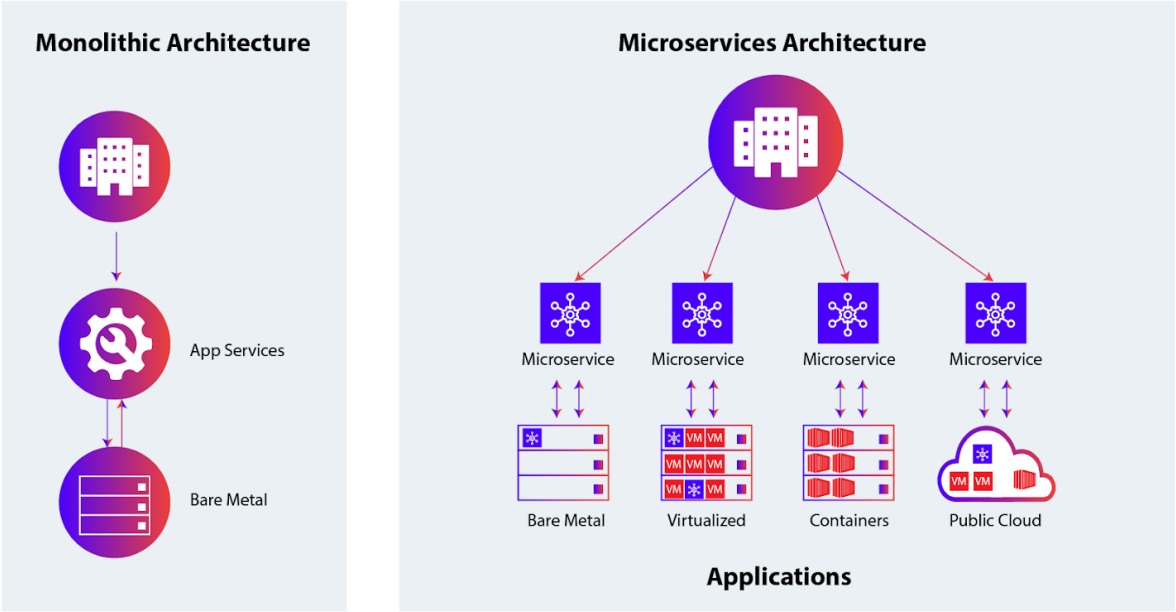


Рисунок 2.4 - Порівняння монолітної та мікросервісної архітектури

Ефективність мікросервісної архітектури пов'язана з розробленням ефективних інструментів для керування цією складністю. Системи управління контейнерами, такі як Kubernetes, можуть полегшити деплоймент та керування сервісами. Важливо також ретельно планувати структуру та залежності між сервісами, щоб забезпечити легку масштабованість та підтримку.

Ефективність мікросервісної архітектури також пов'язана з швидкістю вдосконалення та впровадження змін. Однією з головних переваг мікросервісного підходу є можливість незалежного вдосконалення окремих сервісів, що прискорює процес розробки та випуску нових функцій. Затримки у внесенні змін, зумовлені великими монолітними системами, у мікросервісах значно зменшуються. Однак важливо розглядати, наскільки добре інфраструктура підтримує автоматизовані тести, постійну інтеграцію та постійну доставку (CI/CD), щоб забезпечити стабільність та впевненість у внесенні змін.

Ефективне використання мікросервісної архітектури пов'язане із здатністю системи масштабуватися зростанням обсягів роботи. Кожен сервіс може бути масштабований незалежно, що дозволяє оптимізувати використання ресурсів. З іншого боку, масштабованість може також призвести до високих витрат. Збільшення кількості сервісів може призвести до збільшення обсягів трафіку та зростання витрат на інфраструктуру.

Надійність системи є ключовим аспектом ефективності мікросервісної архітектури. Розподілена природа системи ставить завдання перед розробниками щодо забезпечення стійкості та надійності кожного окремого сервісу. Забезпечення механізмів відновлення після виникнення помилок та визначення стратегій обробки винятків є важливою складовою забезпечення надійності системи. Системи моніторингу та логування допомагають вчасно виявляти та усувати неполадки.

Важливим аспектом ефективності мікросервісної архітектури є здатність постійно моніторити та аналізувати продуктивність кожного окремого сервісу. Відслідковування ключових метрик, таких як час відповіді, завантаження та використання ресурсів, дозволяє вчасно виявляти проблеми ефективності та продуктивності. Системи моніторингу, такі як Prometheus або Grafana, забезпечують можливість відображення та аналізу метрик у реальному часі. Аналіз продуктивності допомагає визначити слабкі місця системи та оптимізувати її роботу. Важливим аспектом в оцінці ефективності мікросервісної архітектури є здатність ефективно управляти версіями сервісів та проводити розгортання нового функціоналу. Завдяки незалежності кожного сервісу, можливо оновлювати лише конкретний компонент без впливу на інші частини системи.

Стратегії розгортання, такі як "canary deployments" чи "blue-green deployments", є важливими для забезпечення безперебійного впровадження нового функціоналу. Підтримка системи в умовах мікросервісної архітектури вимагає особливої уваги до моніторингу умов експлуатації кожного сервісу. Надійна система моніторингу дозволяє виявляти аномалії та узгоджувати дії для

забезпечення стабільності. Засоби моніторингу повинні включати в себе логування та трейсинг для швидкого виявлення та вирішення проблем. Оперативна реакція на події та аналіз логів є ключовим елементом в підтримці роботи мікросервісів

Останній, але не менш важливий аспект - це інтеграція та взаємодія між мікросервісами. Ефективність мікросервісної архітектури визначається не лише незалежністю окремих сервісів, але й їхньою здатністю взаємодіяти. Забезпечення стандартів комунікації, таких як REST або MQTT, дозволяє ефективно обмінюватися даними між сервісами. Розробка чітких контрактів і використання API Gateway допомагають управляти цією взаємодією.

2.3 Стратегії керування даними в мікросервісах

В мікросервісній архітектурі ефективне керування даними грає ключову роль у забезпеченні функціональності та продуктивності системи. Однією з основних стратегій є розподілене зберігання даних на рівні окремих мікросервісів. Кожен сервіс відповідає за свою область даних, що сприяє автономності та гнучкості. Розділення баз даних на рівні сервісів дозволяє незалежно розвивати та масштабувати кожен компонент системи. Це полегшує роботу розробникам та забезпечує високий рівень ізоляції даних між сервісами. Однак, при цьому важливо розробити ефективний механізм обміну даними між сервісами для забезпечення координації та консистентності. (Рис. 2.5)

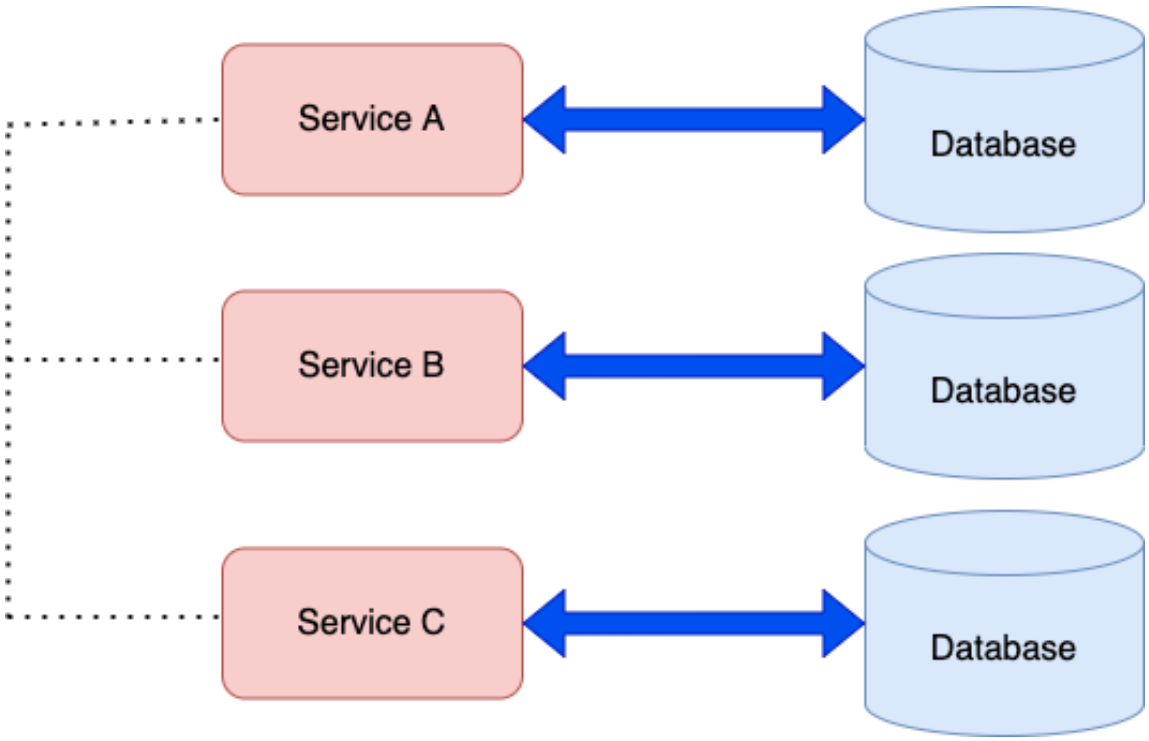


Рисунок 2.5 - Позподілене зберігання даних на рівні окремих мікросервісів

Ще однією стратегією керування даними в мікросервісах є централізоване зберігання. У цьому підході, окремі мікросервіси можуть використовувати спільні бази даних або централізовані засоби для зберігання даних. Це спрощує обмін інформацією між сервісами та дозволяє забезпечити єдність даних.Централізоване керування даними може бути особливо ефективним для областей, де важлива консистентність та інтеграція інформації. Однак цей підхід може призвести до залежності сервісів один від одного та ускладнити розгортання та масштабування системи.

Обидві стратегії - розподілене зберігання даних та централізоване керування - мають свої переваги та недоліки. Розподілене зберігання дозволяє досягти високої автономності сервісів, але може призвести до проблем у взаємодії та консистентності даних. Централізоване керування даними забезпечує єдність інформації та спрощує управління даними, але стає центральним пунктом збоїв та обмежує гнучкість в розвитку окремих сервісів. Модульність та гнучкість розподіленого зберігання, поряд із єдністю даних централізованого підходу, можуть бути поєднані в комбінованих стратегіях, таких як гібридні системи.

У реальних проектах часто використовують комбіновані стратегії керування даними для забезпечення балансу між автономією сервісів та єдністю даних. Один із підходів - розподілене зберігання для основних сутностей та централізоване керування для об'єднання даних на рівні, необхідному для інтеграції. Цей підхід дозволяє сервісам мати свої власні бази даних для швидкого доступу та збереження консистентності у своєму контексті, а водночас об'єднує ключові дані в централізовану систему для інтегрованої обробки та звітності. Ці комбіновані стратегії можуть ефективно вирішувати проблеми, пов'язані із завданням балансу між автономністю та спільністю в мікросервісній архітектурі. Їхній вибір залежить від конкретних вимог та характеристик системи.

Додатковою стратегією керування даними є використання асинхронних методів синхронізації. У цьому підході, сервіси можуть асинхронно спілкуватися та обмінюватися даними через події чи черги повідомлень. Це знижує пряму залежність між сервісами та робить їх менш чутливими до змін у схемі бази даних. Асинхронна синхронізація особливо корисна в умовах великої кількості мікросервісів, де збільшується ймовірність одночасних змін даних. Завдяки цьому підходу, система може бути більш ефективною, зменшуючи блокування та забезпечуючи гнучкість обміну даними. Ще однією важливою аспект стратегії керування даними є забезпечення безпеки та конфіденційності інформації. Використання шифрування для збереження та передачі даних між мікросервісами є ефективним заходом забезпечення безпеки. Це особливо важливо в галузі охорони здоров'я та обробки особистих даних. Ці стратегії варто розглядати при розробці мікросервісних систем, оскільки вони допомагають уникнути проблем, пов'язаних із залежністю та безпекою даних, і водночас забезпечують гнучкість та ефективність в обміні інформацією.

Ще однією важливою частиною стратегії керування даними є управління змінами в схемі бази даних. У мікросервісах, де кожен сервіс може розвиватися автономно, важливо мати ефективний механізм версіонування та міграції схеми даних. Використання механізмів автоматичної міграції баз даних, а також

стратегій версіонування API, дозволяє уникнути проблем, пов'язаних із несумісністю схем та забезпечити плавний розвиток системи. Це особливо актуально у сценаріях, де різні частини системи можуть розгортатися незалежно.

Ефективна стратегія керування даними також включає в себе моніторинг та журналювання подій. Системи моніторингу можуть допомагати вчасно виявляти проблеми зі збереженням даних, а журналювання подій дозволяє відслідковувати зміни та дії над даними для вирішення конфліктів та відновлення консистентності. Ці аспекти стратегії важливі для забезпечення надійності та доступності даних, а також для реагування на події та зміни в реальному часі.

Важливою частиною стратегії керування даними є резервне копіювання та відновлення. Забезпечення регулярного створення резервних копій даних допомагає уникнути втрати інформації в разі збоїв, випадкового видалення або інших подій, що можуть вплинути на цілісність даних. Стратегія резервного копіювання також повинна включати тестування процесів відновлення, щоб впевнитися, що дані можна ефективно відновити в робочому стані в найкоротший термін. Для оптимізації продуктивності та масштабованості можна використовувати стратегії шардування. Горизонтальне шардування включає розподіл даних між різними сервісами або базами даних, в той час як вертикальне шардування передбачає розподіл колонок або полів даних між різними сутностями.Ці стратегії дозволяють розподіляти навантаження та збільшувати швидкодію деяких частин системи, що може бути особливо важливим у великих та високовитратних мікросервісних системах.

Ефективне використання кешування може значно покращити продуктивність системи, зменшуючи час відповіді на запитання до бази даних. Кешування може бути реалізоване на різних рівнях, включаючи рівень бази даних, рівень сервісів та рівень клієнта. Оптимізація запитань до баз даних включає в себе розробку ефективних SQL-запитів, використання індексів та інших методів, щоб забезпечити швидкий доступ до необхідних даних. Це особливо важливо в мікросервісних системах, де кожен сервіс може мати свою

базу даних. Інструменти моніторингу та аналізу даних дозволяють вчасно виявляти проблеми продуктивності, визначати патерни використання та робити інформовані рішення для оптимізації системи. Моніторинг допомагає виявляти слабкі місця та аналізувати поведінку даних у реальному часі. Введення стратегій для спільного використання даних та синергії між сервісами може підвищити ефективність системи. Використання спільних сховищ даних, а також розробка інтерфейсів для обміну та інтеграції інформації можуть сприяти гармонійній роботі мікросервісів.

Інфраструктура як код (IaC) стає все більш важливою стратегією для керування середовищем та ресурсами, необхідними для роботи мікросервісів. Використання IaC дозволяє визначати та управляти інфраструктурними ресурсами через код, що полегшує автоматизацію розгортання та забезпечує консистентність середовищ мікросервісів.

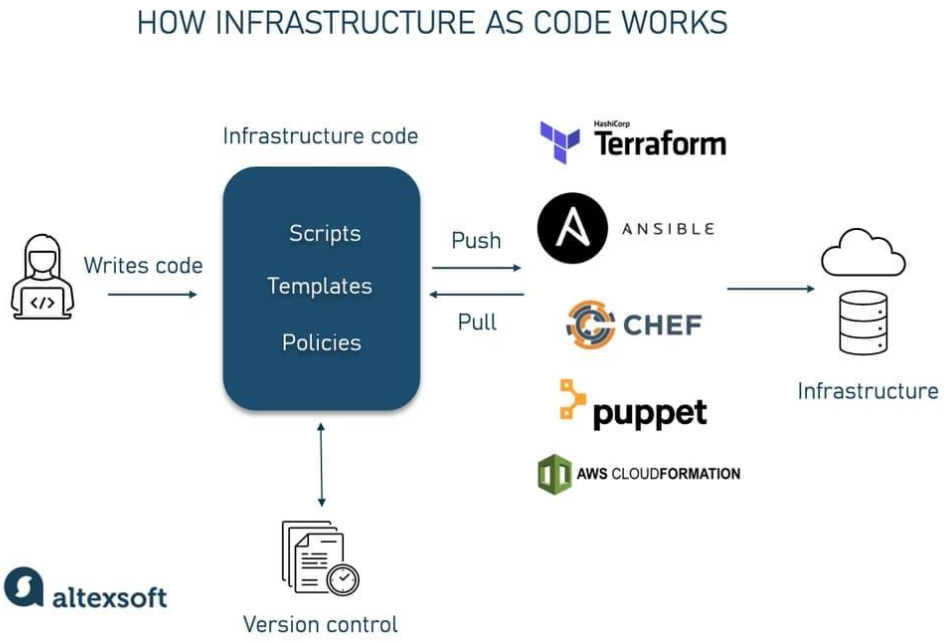


Рисунок 2.6 - Інфраструктура як код

IaC впроваджує парадигму, де конфігурації інфраструктури визначаються за допомогою коду, а не традиційними ручними методами або скриптами. Це привносить низку переваг для керування мікросервісною архітектурою.

1. Автоматизація розгортання: IaC дозволяє розгортати та налаштовувати інфраструктуру шляхом визначення коду, що спрощує та прискорює процес розгортання мікросервісів. Це особливо важливо в сучасних розвиткових середовищах, де вимоги до швидкості впровадження нових функцій є критичними.

2. Консистентність середовищ: завдяки IaC можна забезпечити консистентність середовищ у різних етапах розробки та в різних тестових середовищах. Це допомагає уникнути проблем, пов'язаних із різницею конфігурацій між середовищами розробки, тестування та виробництва.

3. Легше масштабування: IaC дозволяє швидко та ефективно масштабувати інфраструктуру, визначаючи потрібні зміни в коді. Це особливо важливо для мікросервісів, які можуть зазнавати змін у вимогах до обсягу роботи.

4. Версіонування та аудит: використання коду для опису інфраструктури дозволяє зберігати версії конфігурацій у системах контролю версій. Це дозволяє відслідковувати зміни, проводити аудит та відновлювати попередні версії, що є важливими для забезпечення безпеки та стабільності.

5. Інтеграція з CI/CD: IaC добре поєднується з Continuous Integration та Continuous Deployment, спрощуючи автоматизацію тестування, збірки та розгортання мікросервісів у виробничі середовища.

6. Покращення керування змінами: зміни в інфраструктурі можна вносити за допомогою коду, що сприяє кращому керуванню змінами та зменшує ризик помилок, які можуть виникнути при ручному втручанні.

2.4 Безпека та захист даних в мікросервісах

Безпека є невід'ємною частиною розробки та експлуатації мікросервісних систем. Забезпечення надійного захисту даних та мікросервісних архітектур передбачає використання різноманітних стратегій та інструментів.

1. Аутентифікація та авторизація: ефективна аутентифікація та авторизація є ключовими аспектами безпеки мікросервісних систем. Використання механізмів, таких як токени JWT [6] (JSON Web Tokens), дозволяє надійно ідентифікувати та визначати права доступу до різних сервісів.

2. Захист комунікацій: шифрування та захист трафіку між мікросервісами є обов'язковим для уникнення перехоплення та несанкціонованого доступу до данних. Використання протоколів HTTPS та TLS дозволяє створювати зашифровані тунелі для безпечної передачі інформації.

3. Моніторинг та аудит: системи моніторингу та аудиту грають ключову роль у виявленні та вирішенні потенційних загроз безпеці. Автоматизовані системи відслідковування дозволяють виявляти аномалії та проводити детальний аналіз подій для подальшого аудиту.

4. Керування ключами та сертифікатами: ефективне керування ключами та сертифікатами є важливою частиною стратегії захисту даних в мікросервісних системах. Регулярна зміна ключів та використання безпечних протоколів допомагає уникнути зламів.

5. Захист від кросс-сайтового скриптування (XSS) та внедрення коду: мікросервіси повинні бути захищені від загроз, таких як XSS атаки та внедрення коду. використання валідації даних та регулярне навчання команди є важливими елементами безпеки.

6. Регулярні оновлення та патчі: актуальність програмного забезпечення та системних компонентів є критичною для безпеки мікросервісів. Регулярні оновлення та вчасні патчі дозволяють уникнути використання вразливостей та забезпечити стійкість системи до потенційних атак.

7. Обмеження за принципом найменшого дозволеного: використання принципу найменшого дозволеного забезпечує те, що кожен мікросервіс має лише ті права доступу, які необхідні для виконання своїх функцій. Це зменшує ризик несанкціонованого доступу у разі компрометації.

8. Захист від відмови в обслуговуванні (DDoS): мікросервіси повинні мати механізми захисту від DDoS-атак, які можуть перешкоджати нормальному

функціонуванню системи. Використання CDN, розподілення трафіку та автоматизовані системи виявлення DDoS допомагають у збереженні доступності.

9. Захист від витоку інформації: механізми, спрямовані на запобігання витоку конфіденційної інформації, є критичними для безпеки мікросервісів. Шифрування даних у спокої та під час передачі, а також контроль доступу до конфіденційних ресурсів, гарантують найвищий рівень захисту.

10. Постійне тестування на захист: регулярне проведення тестів на безпеку є важливим етапом в життєвому циклі мікросервісів. Застосування пенетраційних тестів, сканування вразливостей та аналіз коду дозволяє виявляти та усувати потенційні проблеми безпеки.

У сучасному цифровому ландшафті, де мікросервіси стають основним елементом архітектури додатків, безпека та захист даних виступають на перший план. Правильна стратегія безпеки в мікросервісах необхідна для забезпечення конфіденційності, цілісності та доступності інформації.

Інтеграція механізмів аутентифікації та авторизації забезпечує контроль над доступом до мікросервісів, зменшуючи ризик несанкціонованого доступу. Захист комунікацій за допомогою шифрування та застосування протоколів безпеки є критичним у запобіганні перехопленню та модифікації даних у транзиті. Моніторинг та аудит є необхідними для виявлення аномалій та швидкого реагування на потенційні загрози безпеки. Регулярні оновлення, ефективне керування ключами та інші стратегії забезпечують стійкість системи до еволюції загроз. Захист від DDoS-атак, витоку інформації та постійне тестування на захист є невід'ємними складовими безпеки мікросервісних систем. Постійне вдосконалення та впровадження передових практик допомагає забезпечити високий рівень безпеки та довіри в екосистемі мікросервісів.

Таким чином, ефективна стратегія безпеки в мікросервісах вимагає комплексного підходу, поєднуючи технічні рішення та проактивний моніторинг для забезпечення безпечного та стабільного функціонування системи.

3 ПРОЕКТУВАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ СИСТЕМИ ОБЛІКУ

3.1 Визначення ключових компонентів нової архітектури

3.1.1 Стандарт для обміну медичними даними.

FHIR [7] (Fast Healthcare Interoperability Resources), є відкритим стандартом для обміну медичною інформацією в охороні здоров'я. Розроблений Health Level Seven International (HL7), FHIR виступає альтернативою традиційним стандартам обміну даними, пропонуючи модерний, гнучкий та легкий підхід до роботи з медичною інформацією.

Основною метою FHIR є полегшення взаємодії між різними системами у сфері охорони здоров'я. Це досягається за допомогою визначення стандартів обміну даними за допомогою ресурсів, які представляють собою конкретні аспекти медичної інформації (наприклад, пацієнти, лікарські препарати, діагнози тощо). Кожен ресурс у FHIR має унікальний ідентифікатор та визначається за допомогою стандартів відкритого програмного забезпечення та веб-технологій, таких як RESTful API.

Однією з ключових переваг FHIR є його простота та гнучкість. Він використовує сучасні підходи до створення API, такі як JSON та XML, що робить його легким для використання та розширення. Більше того, FHIR активно використовується для розвитку мобільних додатків, додатків для збору даних та інших інноваційних рішень в галузі охорони здоров'я.

У контексті проектування мікросервісної архітектури для системи охорони здоров'я, використання FHIR стає ключовим фактором для забезпечення ефективного обміну та управління медичною інформацією. Стандарт дозволяє створювати інтероперабельні, легкі для розширення та модерні системи, які відповідають сучасним вимогам у галузі охорони здоров'я. FHIR виступає не тільки як стандарт обміну даними в охороні здоров'я, але й як ключовий фактор у проектуванні мікросервісної архітектури, спрямованої на вдосконалення

обробки та обміну медичною інформацією. FHIR впливає на визначення ключових компонентів у контексті нового проекту.

Ресурси FHIR як ключові компоненти: ресурси, визначені у FHIR, стають основними компонентами мікросервісної архітектури системи обліку пацієнтів медичної лабораторії. Кожен ресурс (пацієнти, лікарські препарати, процедури тощо) перетворюється в окремий мікросервіс, що дозволяє їм бути автономними та відповідати конкретним функціональним вимогам.

RESTful API для взаємодії: основуючись на принципах RESTful API, мікросервіси взаємодіють між собою та іншими системами за допомогою стандартизованого протоколу обміну даними. Це забезпечує простоту інтеграції та ефективний обмін інформацією.

Гнучкість та розширюваність: FHIR дозволяє додавати власні елементи та розширювати стандартні ресурси, що надає гнучкість у розробці та збереженні власних функціональних можливостей. Мікросервіси можуть бути легко розширені, щоб відповідати унікальним потребам системи.

Спрощення розгортання та масштабованість: мікросервіси, засновані на FHIR, сприяють спрощенню розгортання та масштабованості. Кожен мікросервіс може бути розгорнутий незалежно, а масштабування можливе тільки для конкретних компонентів, які потребують більше ресурсів.

Безпека та захист даних: засновані на сучасних принципах безпеки, мікросервіси FHIR використовують шифрування та механізми аутентифікації для забезпечення конфіденційності та цілісності медичних даних.

Отже, FHIR стає фундаментом для створення високопродуктивних, гнучких та легко розширюваних мікросервісів у сфері охорони здоров'я, визначаючи ключові компоненти для нової архітектури. Це сприяє створенню інноваційних рішень, які відповідають вимогам сучасної медичної галузі.

3.1.2 Серверна частина мікросервісів.

Обрання правильного бекенд-фреймворку – важливе рішення для успішного розвитку та функціонування системи. Використання NestJS [8] для

реалізації бекенду мікросервісів приводить до ряду переваг, що визначають його перевагу в контексті проектування та розгортання високопродуктивних та масштабованих систем. Нижче розглянуті основні аспекти, які визначають обрання NestJS як ключового компонента архітектури мікросервісів.

Тенденції уніфікації та спрощення розробки: за останні роки спостерігається тенденція до уніфікації розробки за допомогою TypeScript, що надає переваги типізації та великої розширюваності. NestJS, як фреймворк, базується на TypeScript, визначаючи його як оптимальний вибір для сучасного бекенду.

Міцна архітектура та підтримка мікросервісів: NestJS відзначається архітектурою, заснованою на концепціях модульності та підтримки мікросервісної архітектури. Його структура дозволяє легко організовувати та масштабувати окремі мікросервіси.

Ефективне управління залежностями: завдяки використанню dependency injection та вбудованої системи управління залежностями, NestJS спрощує і покращує управління компонентами та забезпечує ефективну інтеграцію з іншими сервісами в системі.

Типізація та TypeScript: NestJS базується на TypeScript, який привносить в процес розробки сильну типізацію. Це не тільки робить код більш стійким до помилок, але і сприяє покращенню інтеграції та підтримки коду.

Підтримка websocket та real-time функціоналу: вбудована підтримка WebSocket у NestJS відкриває нові можливості для реалізації реального часу та інтерактивних функцій в мікросервісах. Це особливо важливо для систем, де важлива швидкість та актуальність даних.

Активна спільнота та підтримка: NestJS користується великою та активною спільнотою розробників. Це не лише гарантує наявність багатьох корисних розширень та плагінів, але і забезпечує актуальність фреймворку та вчасні виправлення потенційних проблем.

Розширені можливості аутентифікації та авторизації: NestJS надає ефективні засоби для реалізації аутентифікації та авторизації в мікросервісах.

Розширені засоби, такі як готові модулі Passport, спрощують інтеграцію з різними стратегіями безпеки.

Висока продуктивність та швидкодія: використання TypeScript, підтримка асинхронного програмування та оптимізація ресурсів роблять NestJS відмінним вибором для високопродуктивних систем.

Моніторинг та журналювання: NestJS дозволяє ефективно впроваджувати засоби моніторингу та журналювання для відстеження роботи мікросервісів. Це важливо для аналізу продуктивності та вчасного виявлення проблем.

NestJS виявляється особливо зручним у поєднанні з Kubernetes, надаючи високий рівень зручності та ефективності у процесі розгортання та управління мікросервісною архітектурою.

Контейнеризація з Docker: NestJS легко контейнеризується з використанням Docker, що дозволяє ізолювати кожен мікросервіс та його залежності у власному середовищі. Це створює єдиний, переносимий образ, готовий для розгортання.

Гнучкість та конфігурація: NestJS дозволяє легко налаштовувати та конфігурувати сервіси для різних середовищ. Засоби конфігурації, такі як dotenv, інтегруються з легкістю, дозволяючи налаштовувати мікросервіси під конкретні потреби Kubernetes.

Масштабованість та оптимізація ресурсів: завдяки вбудованій підтримці масштабування та оптимізації ресурсів, NestJS легко інтегрується з можливостями Kubernetes щодо автоматичного масштабування сервісів, а також ефективного розподілу ресурсів.

Інтеграція з Kubernetes API: NestJS має пакети та бібліотеки для зручної інтеграції з Kubernetes API. Це дозволяє автоматизувати такі завдання, як реєстрація сервісів, обмін конфігураційними даними та взаємодія з Kubernetes для управління контейнерами.

Rolling updates та canary deployments: Kubernetes надає зручні інструменти для впровадження rolling updates та canary deployments. NestJS, у свою чергу,

забезпечує гнучкість та безпеку впровадження нових версій мікросервісів, дозволяючи плавно оновлювати або тестувати новий функціонал.

Service discovery та load balancing: NestJS легко інтегрується з механізмами service discovery та load balancing Kubernetes. Це дозволяє ефективно виявляти та маршрутизувати трафік між різними інстанціями мікросервісів.

Monitoring та logging: Kubernetes надає інструменти для моніторингу та збору логів. NestJS може легко інтегруватися з різними системами моніторингу, забезпечуючи детальну статистику та розуміння продуктивності мікросервісів.

Все це робить NestJS інструментом для реалізації мікросервісної архітектури та подальшого ефективного деплою за допомогою Kubernetes.

3.1.3 Брокер повідомлень для міжсервісної комунікації.

За визначенням мікросервісної архітектури як базису для створення системи обліку пацієнтів у медичній лабораторії, виникає необхідність ретельного розгляду ключових аспектів інфраструктури, спрямованих на забезпечення ефективного та надійного взаємодії між різними сервісами. Важливий вибір, який впливає на взаємодію компонентів системи, є обрання брокера повідомлень.

RabbitMQ [9] - високоефективний інструмент для забезпечення надійного та ефективного обміну інформацією між мікросервісами. Обрання RabbitMQ в якості брокера повідомлень виявляється необхідною складовою у структурі мікросервісної архітектури, забезпечуючи ефективний та надійний обмін інформацією між компонентами системи. Основні переваги RabbitMQ, які визначили його вибір, детально розглядаються в контексті декількох ключових аспектів.

Асинхронність та розподіленість: RabbitMQ дозволяє будувати розподілені системи з асинхронною взаємодією. Замість прямої взаємодії між сервісами, можливо відправляти повідомлення через брокер, дозволяючи сервісам працювати асинхронно та реагувати на події у відокремленому середовищі.

Гнучкість та легкість інтеграції: RabbitMQ підтримує різні протоколи, такі як AMQP та MQTT, що робить його гнучким для інтеграції з різними

технологіями. Ця гнучкість дозволяє легко впроваджувати нові сервіси та розширювати функціонал системи.

Надійність та опції забезпечення якості служб: з можливістю визначення політик обробки повідомлень, RabbitMQ дозволяє забезпечувати надійну доставку та обробку повідомлень. Опції переадресації, повторної відправки та управління помилками забезпечують високу якість обслуговування.

Масштабованість та висока продуктивність: здатність RabbitMQ масштабуватися горизонтально дозволяє збільшувати обробку повідомлень при зростанні навантаження. Його архітектура розрахована на високий обсяг та швидкість обробки повідомлень, що важливо для даної системи.

Можливості маршрутизації та фільтрації: RabbitMQ дозволяє налаштовувати маршрутизацію та фільтрацію повідомлень, що дозволяє точно направляти та обробляти повідомлення відповідно до потреб конкретних сервісів.

Однією з ключових переваг RabbitMQ є його архітектурна стійкість та здатність до забезпечення високого рівня надійності та продуктивності у мікросервісних архітектурах. Ці аспекти є критичними для забезпечення ефективної обробки повідомлень та забезпечення неперервної доступності сервісів.

Розподілена архітектура: RabbitMQ базується на розподіленій архітектурі, що сприяє паралельному обробці повідомлень. Кожен з вузлів системи може працювати незалежно, забезпечуючи таким чином гнучкість та високу доступність у разі збоїв чи навантаження.

Модель publish-subscribe: основним принципом взаємодії в RabbitMQ є модель publish-subscribe, яка дозволяє ефективно передавати повідомлення між виробниками (publishers) та споживачами (subscribers). Це полегшує реалізацію асинхронної взаємодії між сервісами та підвищує швидкість обміну інформацією.

Гнучка система керування кількістю з'єднань: RabbitMQ керує кількістю з'єднань та каналів, що дозволяє ефективно використовувати ресурси системи.

Кожен канал представляє собою окремий потік для обміну повідомленнями, що сприяє ефективності та управлінню ресурсами.

Система маршрутизації та фільтрації: RabbitMQ володіє потужною системою маршрутизації, що дозволяє точно направляти повідомлення відповідно до ключів обміну, обмінів та багатьох інших параметрів. Це надає гнучкість та точність в управлінні потоком даних.

Забезпечення надійності повідомлень: система підтвердження доставки (acknowledgment) в RabbitMQ забезпечує надійність обміну повідомленнями. Кожен споживач може підтверджувати отримання повідомлення, забезпечуючи високий рівень доставки.

Масштабованість за допомогою кластеризації: RabbitMQ дозволяє легко масштабуватися шляхом створення кластерів. Кластеризація забезпечує резервування та обробку повідомлень навіть у випадку збоїв в окремих вузлах.

Захист від потенційних збоїв: за допомогою різноманітних політик маршрутизації та обробки помилок, RabbitMQ може автоматично перенаправляти повідомлення та управляти ситуаціями нездатності до доставки.

У мікросервісній архітектурі системи обліку пацієнтів, використання topic exchange в RabbitMQ виявляється надзвичайно корисним для ефективної обробки та маршрутизації повідомлень між сервісами.

Topic exchange дозволяє визначити ключові слова (теми) для кожного повідомлення. При відправці повідомлення, виробник може вказати ключові слова, які визначають, до яких черг чи обмінів повинно бути направлено повідомлення. Споживачі ж можуть створювати свої черги, підписуючись на конкретні теми, тим самим отримуючи тільки ті повідомлення, які їх цікавлять. У контексті даного проекту, можна використовувати topic exchange для різних цілей, наприклад:

Розподіл завдань між сервісами, які обробляють різні аспекти обліку пацієнтів (прийом лікаря, аналізи, виписка рецептів тощо).

Сповіщення про стан пацієнтів, такі як результати аналізів, терміни прийому, зміни у медичних записах.

Topic exchange дозволяє нам гнучко налаштовувати комунікацію між сервісами, забезпечуючи точну та ефективну передачу інформації.

dead letter queue (черга для мертвих повідомлень) в RabbitMQ виявляється важливою складовою системи, де необхідно ефективно обробляти неправильні або невдалі повідомлення. Dead letter queue дозволяє визначити чергу, в яку повідомлення будуть переміщені, якщо вони не можуть бути оброблені або доставлені з певною кількістю спроб. Це може бути корисно у випадках, коли повідомлення не вдається обробити через помилки в коді чи невідповідність формату. В даній системі, dead letter queue може використовуватися для обробки ситуацій, де повідомлення не можуть бути коректно оброблені сервісами. Наприклад, при отриманні повідомлень з невірною структурою або з невірними даними, це повідомлення може бути перенаправлене до dead letter queue для подальшого аналізу та виправлення. Dead letter queue – ефективний інструмент для вдосконалення обробки помилок та забезпечення надійності мікросервісної архітектури.

Підсумовуючи, dead letter queue – механізм для обробки неуспішних повідомлень, що підвищуючи надійність та стійкість системи до помилок. Використання RabbitMQ в даному проекті надає нам потужний інструментарій для розробки ефективних та розширюваних мікросервісів, створюючи збалансовану та надійну архітектуру системи обліку пацієнтів.

3.1.4 База даних для кешування даних.

У цьому розділі розглянемо важливий аспект інфраструктури системи обліку пацієнтів – використання Redis [10] як елемента мікросервісної архітектури. Обрання Redis зумовлено рядом функціональних можливостей, які він пропонує та його визнаною ефективністю в розподіленому середовищі.

Швидкодія та кешування: Redis відрізняється своєю винятковою швидкодією, що робить його хорошим рішенням для реалізації механізмів кешування. Використання Redis дозволяє зберігати результати запитів та швидко їх витягувати, полегшуючи доступ до даних та покращуючи продуктивність системи.

Забезпечення асинхронної взаємодії: Redis підтримує механізми публікації-підписки, що дозволяє реалізувати асинхронну взаємодію між різними сервісами. Це стає важливим компонентом для забезпечення швидкодії та ефективності комунікації між різними частинами системи.

Підтримка горизонтального масштабування: Redis дозволяє горизонтальне масштабування, забезпечуючи стабільну продуктивність системи навіть при збільшенні обсягу даних та навантаження. Це важливо для системи обліку пацієнтів, яка може зростати та змінюватися з часом.

Ефективне управління сесіями та станом: у випадках, де необхідно зберігати стан або інформацію про сесії користувачів, Redis виступає ефективним засобом забезпечення цих потреб. Його можливості управління строками, списками та іншими типами даних дають можливість створювати структури для зберігання необхідної інформації.

Підтримка транзакцій: Redis підтримує транзакції, що робить його відмінним вибором для сценаріїв, де необхідно забезпечити консистентність та атомарність операцій. Можливість виконання групи операцій як одного цілого сприяє цілісності даних в системі.

Обрання Redis в якості ключового компонента мікросервісної архітектури базується на цих перевагах та сприятиме покращенню ефективності, надійності та гнучкості системи обліку пацієнтів.

У контексті мікросервісної архітектури системи обліку пацієнтів, Redis відіграє ключову роль в зберіганні та обробці проміжних даних, що виникають під час виконання сервісів-воркерів. Цей механізм дозволяє ефективно керувати тимчасовою інформацією та покращує загальну продуктивність системи.

Збереження проміжних результатів: Redis служить як місце для тимчасового збереження проміжних результатів обробки даних. Сервіс-воркери можуть зберігати та зчитувати дані у реальному часі, користуючись гнучкістю та швидкістю Redis.

Забезпечення атомарності та синхронізації: використання Redis дозволяє забезпечити атомарні операції та синхронізацію даних між різними етапами

обробки. Такий підхід покращує консистентність проміжних даних та гарантує їх коректність на кожному етапі обробки.

Масштабованість та висока швидкодія: Redis, завдяки своїм властивостям масштабованості та високої швидкодії, ідеально підходить для обробки великої кількості проміжних даних. Здатність до роботи з розподіленими даними та реалізація кешування дозволяють ефективно оптимізувати процес обробки.

Зменшення навантаження на основну базу даних: використання Redis дозволяє значно зменшити навантаження на основну базу даних, яка призначена для зберігання стійкої та остаточної інформації. Проміжні дані обробки, які можуть змінюватися часто, не накладають додаткового тиску на основну систему зберігання.

Забезпечення високої доступності: Redis забезпечує високу доступність та надійність, що робить його стабільним інструментом для збереження проміжних даних. Його механізми реплікації та забезпечення консистентності дозволяють уникати втрати даних навіть у випадку збоїв системи.

Загалом, Redis виступає важливим елементом інфраструктури для оптимізації обробки даних у мікросервісній архітектурі, дозволяючи зберігати, обробляти та синхронізувати проміжні дані ефективно та надійно

3.1.5 База даних для сталих даних.

Ще одна основна частина системи обліку пацієнтів – база даних для збереження сталих даних. В якості бази даних я обрав MongoDB [11]. MongoDB – це документоорієнтована база даних, яка здатна зберігати та обробляти дані у форматі JSON-подібних документів. У контексті мікросервісної архітектури системи обліку пацієнтів, використання MongoDB визначається його гнучкістю, масштабованістю та здатністю працювати з різноманітними структурами даних.

Документоорієнтований підхід: MongoDB працює з документами у форматі BSON (Binary JSON). Кожен документ представляє собою набір пар ключ-значення та може містити вкладені документи або масиви. Цей підхід надає гнучкість у зберіганні та організації даних.

Масштабованість: MongoDB підтримує горизонтальне масштабування, що дозволяє розподілити дані між різними серверами та додавати нові сервери при збільшенні обсягу даних. Це важливо для системи обліку пацієнтів, яка може рости та змінюватися з часом.

Індексація та запити: MongoDB забезпечує широкий спектр можливостей індексації, що дозволяє оптимізувати швидкодію запитів. У мікросервісній архітектурі, де ефективність доступу до даних є ключовою, це робить MongoDB привабливим вибором.

Підтримка геоданих: MongoDB надає можливість зберігання та оптимізованого використання геоданих, що може бути корисним для систем обліку пацієнтів, де важливо враховувати просторові аспекти, наприклад, розташування пацієнтів чи медичних закладів.

Гнучка схема та легкість змін: MongoDB не вимагає строго визначеної схеми даних, що дозволяє легко вносити зміни у структуру документів. Це зручно у випадках, коли система піддається регулярним змінам чи оновленням.

MongoDB, обрана для інтеграції в систему обліку пацієнтів медичної лабораторії, привносить кілька ключових переваг. Ця документоорієнтована база даних дозволяє зручно зберігати та моделювати медичну інформацію у форматі JSON-подібних документів. Важливою особливістю є гнучкість у визначенні структури даних, що дозволяє ефективно враховувати різні вимоги щодо зберігання медичних даних.

MongoDB стає центральною точкою для збереження зв'язків між пацієнтами та лікарями, де використання вкладених документів чи ідентифікаторів надає можливість швидко отримувати пов'язану інформацію. Це стає ключовим фактором для ефективної роботи системи, особливо при необхідності вивчення історії лікування та взаємодії пацієнтів з лікарями.

Застосування агрегаційних фреймворків у MongoDB дозволяє проводити різноманітний аналіз даних, що є важливим для отримання статистики та виявлення паттернів у медичних записах. Це може сприяти вдосконаленню лікування та вивченню трендів у медичній сфері. MongoDB також відзначається

зручністю інтеграції з іншими мікросервісами та зовнішніми системами завдяки гнучкості роботи з JSON-подібними документами. Це забезпечує спільний доступ до медичної інформації та сприяє взаємодії різних компонентів системи. Висока швидкодія та можливість горизонтального масштабування роблять MongoDB оптимальним вибором для ефективного управління медичними даними у мікросервісній архітектурі системи обліку пацієнтів медичної лабораторії.

Ще однією важливою аспектом є спрощення роботи зі структурою даних завдяки вбудованій гнучкості MongoDB. Медичні дані можуть бути представлені у формі, яка найкращим чином відповідає конкретним потребам системи, що дозволяє ефективно управляти різноманітністю інформації. MongoDB використовується для забезпечення зв'язності між великою кількістю записів про пацієнтів та відповідальними лікарями. Це дозволяє легко відстежувати інформацію про медичний супровід та виявляти відповідальних за конкретних пацієнтів лікарів. Можливості агрегації в MongoDB використовуються для проведення аналітичних запитів та отримання детальної статистики щодо характеристик пацієнтів та їхнього стану здоров'я. Це забезпечує можливість оперативного втручання та адаптації лікувальних стратегій.

Загальна концепція використання MongoDB в системі обліку пацієнтів полягає в створенні розширеної, гнучкої та ефективної платформи для зберігання та обробки медичної інформації. MongoDB стає ключовим елементом успішної реалізації мікросервісної архітектури, забезпечуючи високий рівень швидкодії, зручний аналіз та інтеграцію з іншими компонентами системи.

Важливим етапом у виборі MongoDB була її можливість взаємодії з системами моніторингу та аналізу здоров'я пацієнтів. MongoDB пропонує зручний механізм для інтеграції з різноманітними інструментами аналітики, дозволяючи швидко та ефективно обробляти велику кількість даних. Крім того, відмінною особливістю MongoDB є підтримка геоданих, що є важливим аспектом в медичній сфері. Це дозволяє враховувати просторові аспекти, такі як

розташування лікарень, доступність до медичних закладів та інші фактори, які можуть впливати на обслуговування пацієнтів.

MongoDB використовується для збереження логів та аудиту доступу до медичної інформації, що сприяє забезпеченню високого рівня безпеки даних пацієнтів. Засоби автентифікації та авторизації MongoDB гарантують, що лише уповноважений персонал має доступ до конфіденційної медичної інформації. Таким чином, MongoDB обрана для ролі бази даних у мікросервісній архітектурі системи обліку пацієнтів медичної лабораторії не лише через свою технічну потужність, але й завдяки специфічним можливостям, які вона пропонує для розвитку та покращення медичних сервісів.

Узагальнюючи, використання MongoDB у мікросервісній архітектурі системи обліку пацієнтів дозволяє забезпечити гнучкість, масштабованість та ефективне управління даними, що є ключовим для успішної інтеграції та функціонування системи.

3.2 Розробка схем та діаграм, що відображають взаємодію мікросервісів

В ході даного етапу дослідження мною були вирішені ключові завдання, пов'язані з розробкою архітектури, а саме визначено стратегію взаємодії сервісів, обрано необхідні технології та створено структурні схеми системи.

Протягом етапу еволюції архітектури до її остаточного стану відбулися численні ітерації та поліпшення. Початкова ітерація архітектури (див. Рис. 3.1) включала сервіси diagnosticReport, patient та specimen як репрезентативний приклад (на остаточному етапі передбачалося наявність наближено сотні подібних сервісів).

Потік даних виглядає наступним чином:

1. Api шлюз отримує запит і надсилає його до відповідного сервісу
2. Сервіс (наприклад, пацієнт) зберігає тіло запиту в mongoDB, робить запит до сервісу воркера для створення нової задачі

3. Сервіс воркера зберігає задачу в БД, надсилає повідомлення через RabbitMQ до сервісу валідації та надсилає id з документу mongoDB
4. Сервіс (наприклад, пацієнт) отримує ідентифікатор задачі та надсилає об'єкт з job_id, subject_id, статусом, тощо у відповідь до api шлюзу
5. Api шлюз надсилає цю відповідь назад тому, хто виконав початковий запит

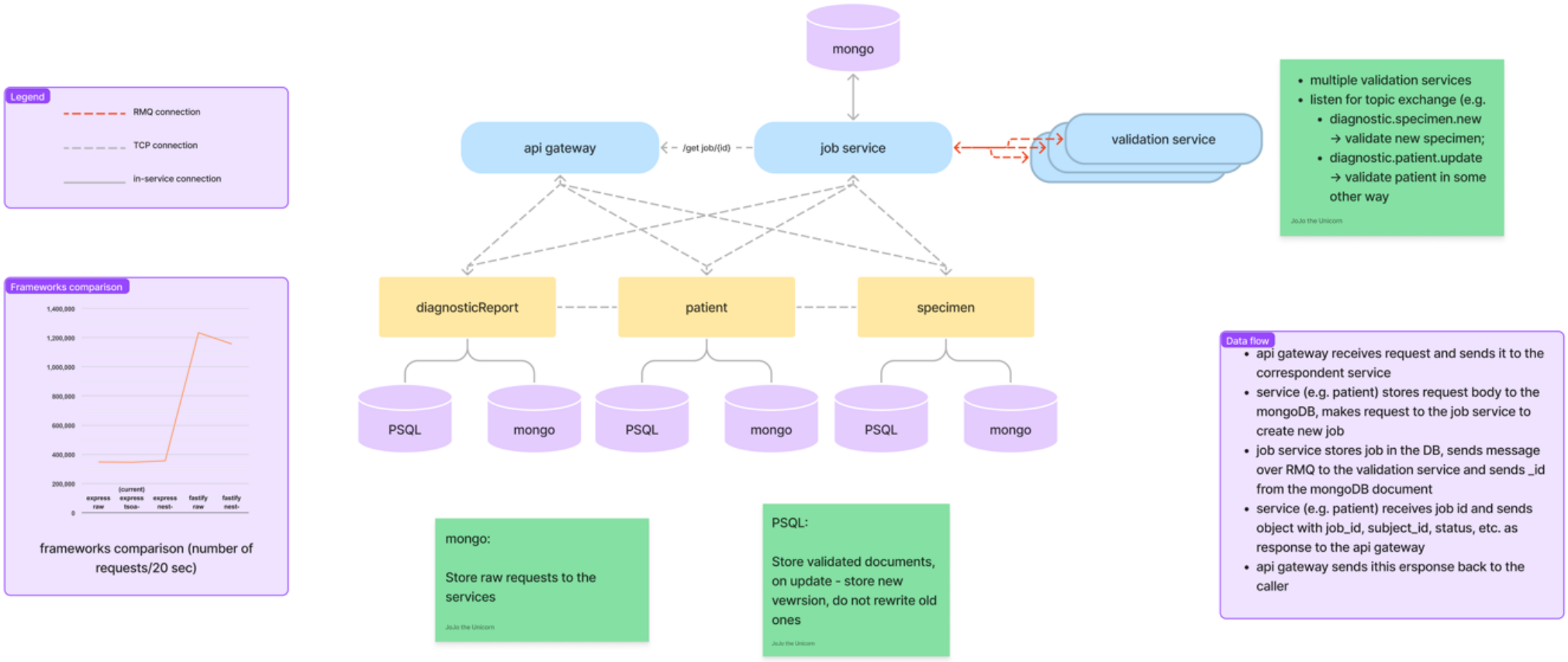


Рисунок 3.1 - Перша ітерація архітектури

Однією з негативних сторін такої архітектури є високий рівень навантаження на кожен індивідуальний мікросервіс. Замість того, щоб просто включати в себе сервер для обробки інформації, кожен сервіс надмірно обтяжений, маючи по дві активні бази даних. Це призводить до значного збільшення витрат на підтримку системи, оскільки потребується додатковий час та ресурси для ефективного управління та синхронізації подвійних баз даних у кожному мікросервісі. Така подвійна навантаженість ризикує ускладненням процесів та вимагає додаткових зусиль для забезпечення стабільності та продуктивності системи.

У покращеній ітерації архітектури (зображено на Рис. 3.2), визначається більш простий вигляд, який спрямований на ефективне використання ресурсів. Нововведення включає в себе перехід від концепції двох баз даних у кожному окремому сервісі до використання двох баз даних для всієї системи, з метою оптимізації ресурсів та вдосконалення економічної ефективності.

У цьому варіанті розвитку, я обрав стратегію об'єднання баз даних, спрямовану на зменшення накладних витрат та покращення адаптивності системи. Важливим аспектом є використання двох типів баз даних: MongoDB для тимчасового зберігання даних та PostgreSQL для постійного зберігання. Цей підхід дозволяє оптимально використовувати кожен тип бази даних для відповідної категорії даних, сприяючи більш ефективному управлінню інформацією та підвищенню продуктивності системи в цілому.

Потік даних виглядає наступним чином:

1. Діагностичний сервіс отримує виклики клієнтського api, обробляє мінімальну перевірку типів, зберігає тіло запиту у відповідній колекції mongoDB, робить запит до сервісу job через TCP для перевірки документа
2. Робочий сервіс генерує новий документ у таблиці `jobs` і відповідає з цим ідентифікатором діагностичному сервісу, з кодом статусу 202 та відповідним об'єктом завдання

Рисунок 3.2 - Друга ітерація архітектури

У вищезазначеному покращеному варіанті архітектури, впроваджено стратегію об'єднання баз даних, що спрямована на ефективне використання ресурсів та забезпечення економічної ефективності. Однак, при такому підході, можуть виникнути деякі виклики та потенційні проблеми:

Відходження від принципів мікросервісної архітектури: об'єднання баз даних для всієї системи може суперечити основним принципам мікросервісної архітектури, таким як незалежність та автономія сервісів. Це може призвести до складнощів у масштабованості та розвитку окремих сервісів.

Ризик однопотоковості: використання двох баз даних для всієї системи може призвести до ризику створення одного потоку з обмеженим доступом, що може стати вузьким кінцем для всіх сервісів та призвести до заворушень та затримок.

Складність управління консистентністю: перехід до такого підходу може ускладнити підтримку консистентності даних між різними типами баз даних. Відсутність однорідності може вплинути на цілісність даних та призвести до потенційних проблем.

З урахуванням цих викликів, важливо здійснювати уважну експлуатацію та регулярну оцінку ефективності обраного підходу, забезпечуючи баланс між простотою реалізації та дотриманням ключових принципів мікросервісної архітектури.

У третьому етапі розвитку архітектури системи було проведено ще більш глибоке спрощення, що включало в себе відмову від сервісів валідації та перенесення всієї функціональності на робочі сервіси (див. Рис. 3.3). Gateway API приймає запит від клієнта, реєструє весь запит в базі даних у формі, в якій він був отриманий, і генерує унікальний ідентифікатор job_id для подальшого використання при доступі до інформації щодо процесу обробки даних робочим сервісом. Після цього Gateway API відправляє job_id, разом з ідентифікатором запису в базі даних, через RabbitMQ. Кожен окремий робочий сервіс очікує на отримання інформації за допомогою AMQP протоколу RabbitMQ. Після отримання необхідних даних робочий сервіс проводить обробку та виконання

відповідних функцій, забезпечуючи ефективну взаємодію всіх компонентів системи.

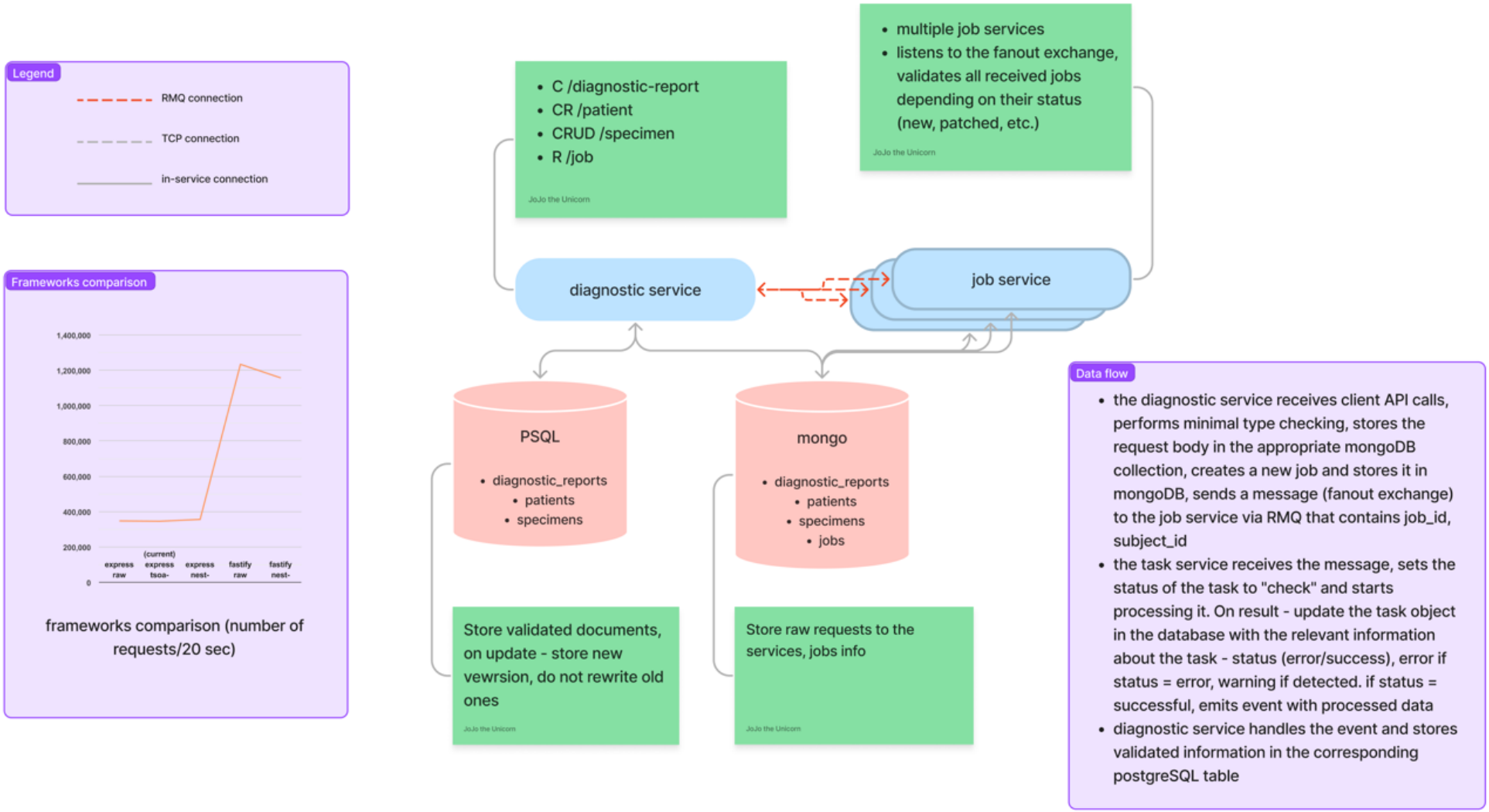


Рисунок 3.3 - Третя ітерація архітектури

З часом стали зрозумілими явні недоліки такої архітектури:

Брак безпеки: Описана модель не включає аспекти безпеки, такі як захист від несанкціонованого доступу чи забезпечення конфіденційності даних.

Відсутність обробки помилок: Опис не враховує можливих сценаріїв помилок або виключень, які можуть виникнути під час взаємодії. Розробка механізмів обробки помилок та відновлення системи є ключовим аспектом для стабільності.

Тестування та валідація: Опис не включає аспекти тестування, які є важливим етапом в розробці мікросервісних систем. Важливо врахувати стратегії тестування та валідації для забезпечення якості та надійності системи.

Моніторинг та аналітика: Не враховано питання моніторингу та аналізу даних для виявлення проблем та вдосконалення продуктивності системи. Додаткові інструменти для моніторингу та аналізу допоможуть забезпечити ефективне управління системою.

Четверта ітерація архітектурного вдосконалення призначалася для стратегічної оптимізації процесу передачі даних від API Gateway до робочого сервісу. Ключовою метою цього етапу було удосконалення ефективності потоку інформації, що забезпечує подальшу обробку документів в базі даних сталих документів. Дана модифікація спрямовувалася на вдосконалення процесу отримання актуальної інформації про оброблені документи, підвищуючи швидкість та ефективність взаємодії між різними компонентами системи. Застосування четвертої версії архітектури виокремлювалося як стратегічний крок у напрямку оптимізації та удосконалення системних процесів для досягнення більш високої продуктивності та відзначалося як важливий крок у напрямку оптимізації ефективності системи в цілому. (Рис. 3.4)

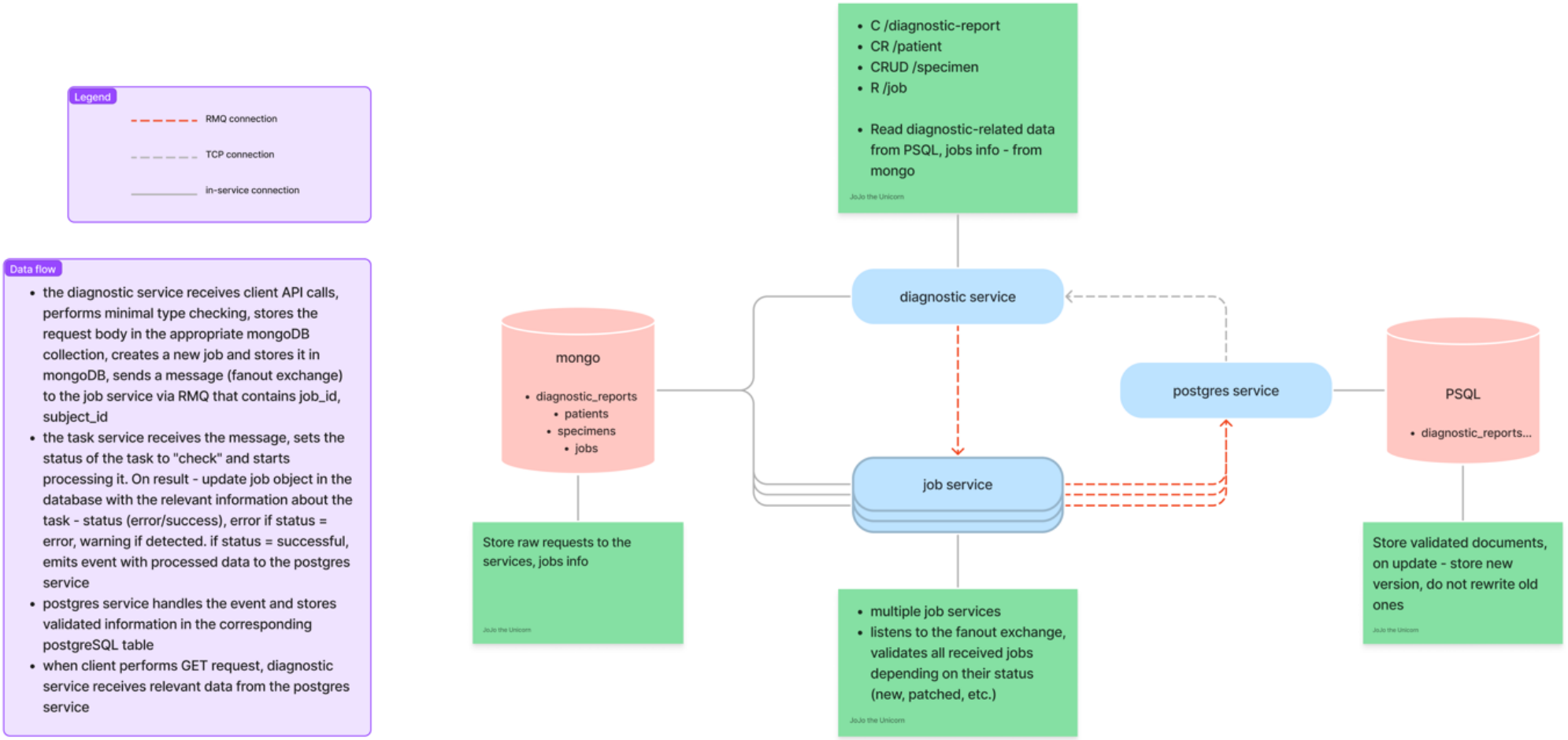


Рисунок 3.4 - Четверта ітерація архітектури

Потік даних в цій ітерації архітектури:

1. API gateway отримує виклики клієнтського API, виконує мінімальну перевірку типів, зберігає тіло запиту у відповідній колекції mongoDB, створює нове завдання, надсилає повідомлення (fanout exchange) робочій службі через RabbitMQ, що містить job_id, subject_id
2. Робочий сервіс отримує повідомлення, встановлює статус завдання в "перевірка" і починає його обробку. По результату - оновлює об'єкт job в базі даних з відповідною інформацією про завдання - статус (помилка/успіх), помилка, якщо статус = помилка, попередження, якщо виявлено. якщо статус = успіх, надсилає подію з обробленими даними до сервісу postgres
3. Сервіс postgres обробляє подію та зберігає перевірену інформацію у відповідній таблиці PostgresSQL
4. Коли клієнт виконує GET запит, діагностичний сервіс отримує відповідні дані від сервісу postgres

Попри на перший погляд оптимальний потік даних в ході розробки проявився недолік використання об’єднаних баз даних для усіх робочих сервісів, Api gateway і postgres сервісу:

1. Коплексність та залежність: використання єдиної бази даних може призводити до виникнення великої кількості схем та таблиць, які обслуговують всі мікросервіси. Це може призвести до складності в управлінні та розгортанні змін, оскільки вони можуть впливати на всю систему.
2. Масштабованість: при великому обсязі даних та високому завантаженні одна об'єднана база може стати обмеженням для масштабованості. Розширення бази даних може бути складним завданням, а необхідність масштабувати всю систему може впливати на продуктивність.
3. Низька ізоляція: всі мікросервіси, які використовують єдину базу даних, можуть взаємодіяти один з одним, впливаючи на стан та дані інших

сервісів. Це може призводити до проблем з ізоляцією та конфліктами даних.

4. Блокування розвитку: об'єднана база даних може стати обмеженням для швидкості та гнучкості розробки. Зміни в одному мікросервісі можуть впливати на всю базу, ускладнюючи процес впровадження нових функцій.
5. Безпека: об'єднана база даних може збільшити ризик щодо безпеки. Якщо один сервіс компрометовано, це може вплинути на всю базу даних, погіршуючи ризик конфіденційності та цілісності даних.
6. Велика кількість відносин: з ростом кількості мікросервісів може зростати і кількість взаємозалежних відносин між таблицями. Це може зробити структуру бази даних складною та важкою для розуміння.
7. Відсутність функціональності для конкретних потреб сервісів: кожен мікросервіс може вимагати специфічних можливостей бази даних, і використання одного рішення для всіх може вести до втрати або обмеження функціональності для конкретних потреб окремих сервісів.

П'ята ітерація архітектури, завершена у кінцевій стадії, виправила недоліки попередньої версії шляхом відновлення принципу "один сервіс – одна база даних". Цей стратегічний крок виявився ключовим для поліпшення ряду аспектів системи. Зокрема, відновлення цього принципу допомогло значно покращити масштабованість, підняло рівень ізоляції між сервісами, зменшило взаємну залежність між ними та підняло рівень безпеки системи. Також важливо відзначити, що ця стратегія сприяла значному зменшенню кількості відносин між таблицями, свідченням чого є графічне відображення цих змін на рис. 3.5. Ця розвинута версія архітектури не лише усунула виявлені недоліки, але й впровадила значні покращення, враховуючи ключові аспекти масштабованості, безпеки та структурної організації баз даних.

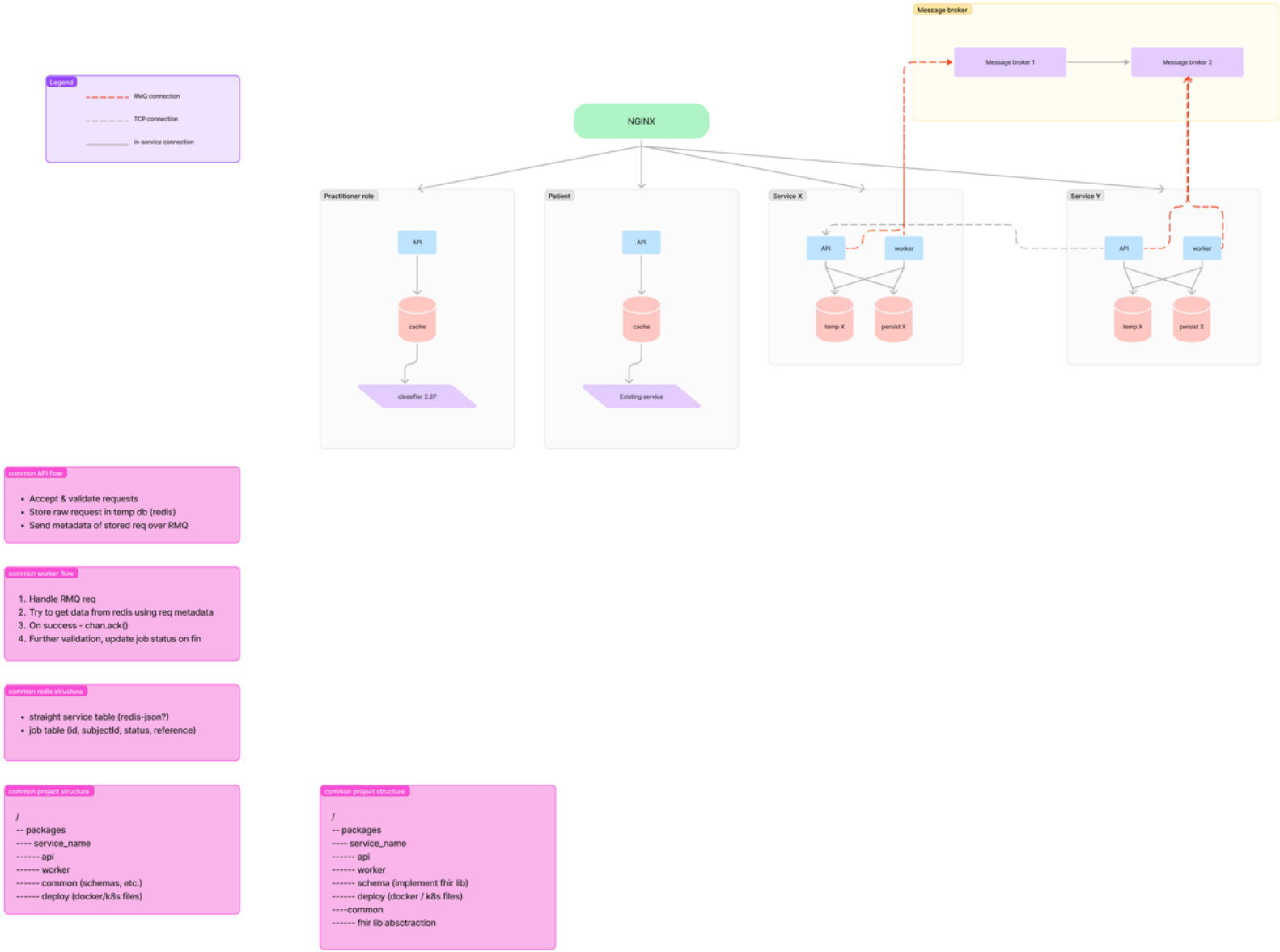


Рисунок 3.5 - Кінцева версія архітектури

4 ІМПЛЕМЕНТАЦІЯ СИСТЕМИ ОБЛІКУ ПАЦІЄНТІВ

Розгляд імплементації зроблений на прикладі сервісу «specimen», що використовується для керування та відстеження зразків аналізів, які беруться у пацієнтів. Структура кожного подібного сервісу включає три основні директорії, що розподілені таким чином для зручності процесу розробки. (Рис 4.1)

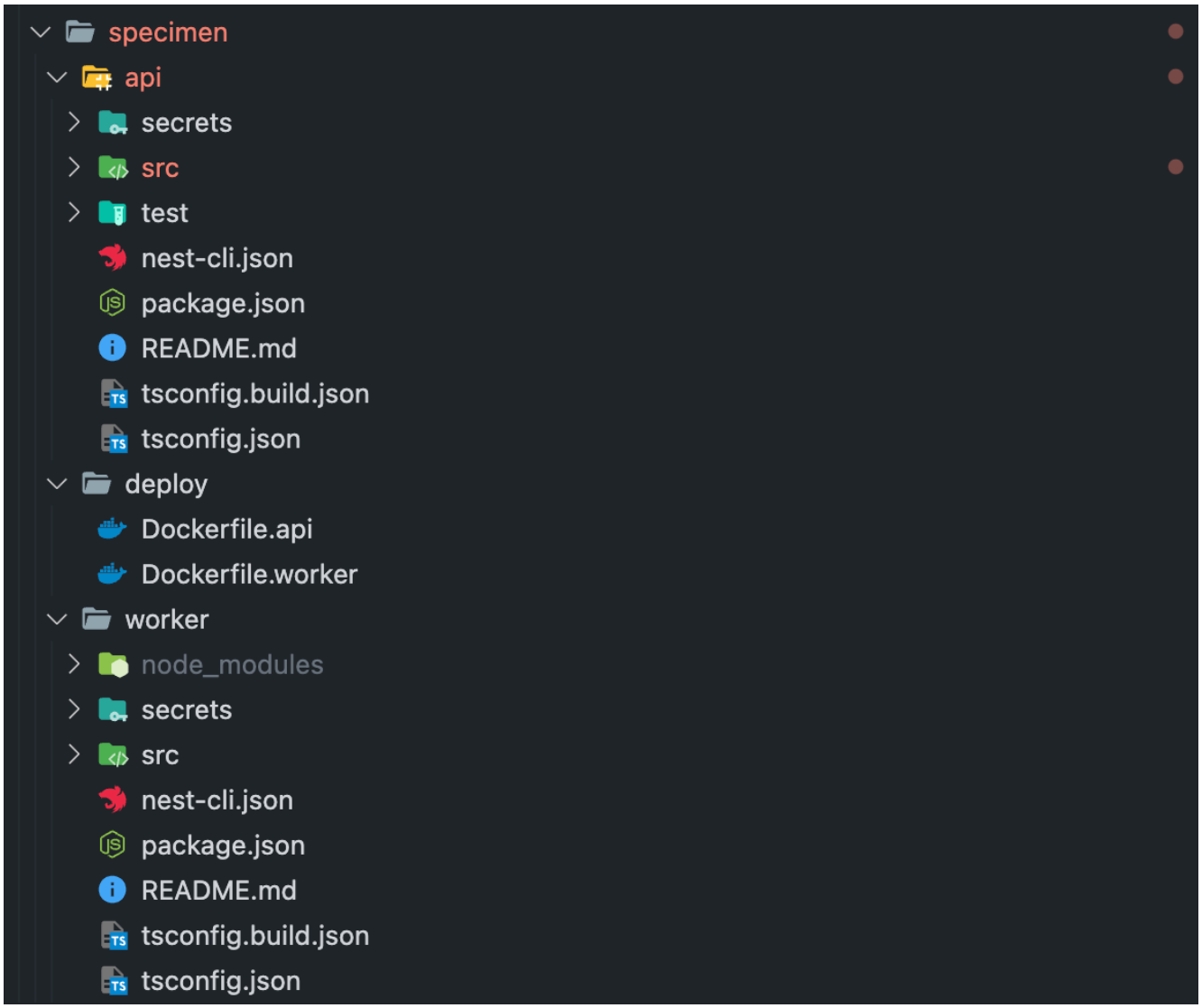


Рисунок 4.1 - Структура файлів сервісу

В директорії api зберігається сервіс, який оброблює основну комунікацію, в директорії worker – допоміжний сервіс, що відповідає за основну обробку інформації. Наприклад, фактичний аналіз та обробку медичних зразків, виконання обчислювальних операцій, тощо. Таке розподілення дозволяє

максимально оптимізувати продуктивність системи шляхом розділення сервісів на менші частини, що дозволяє за необхідності додати нові репліки сервісів що зменшать навантаження на інші сервіси. В директорії deploy знаходяться файли що допомагають при розгортанні сервісу.

Сервіс в директорії api має таку структуру:

- Файл main.ts, точка входу в додаток;
- Файл specimen.module.ts, вхід в основну частину фреймворку Nest.js;
- Файл specimen.controller.ts, що відповідає за обробку запитів до сервісів;
- Файл specimen.service.ts, що розвантажує обробку даних з контролеру на себе;
- Файл specimen.entity.ts, що відповідає за організацію моделі бази даних, з котрою працює specimen.service.ts.

Файл main.ts містить в собі основну фунцію bootstrap що «розгортає» модуль. В процесі конфігурації модуля вказуються кастомний адаптер, логер, логіка валідації запитів і хост та порт котрий прослуховує сервер.

Лістинг функції bootstrap:

```
async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    SpecimenModule,
    new FastifyAdapter({
      bodyLimit: 30 * 1024 * 1024 // Default Limit set to 30MB
    })
  )
  app.useLogger(app.get(WINSTON_MODULE_NEST_PROVIDER))
  app.useGlobalPipes(new ValidationPipe())
  await app.listen(3026, '0.0.0.0')
}
bootstrap()
```

Файл specimen.module.ts містить один клас SpecimenModule, який за допомогою декораторів typescript сконфігурований для можливості використання підключення до основної бази даних, бази даних redis, RabbitMQ. Спільні модулі із кастомних бібліотек для взаємодії між api і робочим сервісом,

логер і окремо – з використанням dependency injection вказані контролер і сервіс що відповідає за цей контролер.

Лістинг класу SpecimenModule

```
@Module({
 imports: [
  TypeOrmModule.forRootAsync({
   useFactory: () => ({
    type: 'postgres',
    host: psqlConfig.host,
    port: psqlConfig.port,
    username: psqlConfig.username,
    password: psqlConfig.password,
    database: psqlConfig.database,
    entities: [ResourceEntity, JobEntity, ResourceHistoryEntity],
    synchronize: false
   })
  }),
  RedisRepoModule.forRoot({
   db: redisConfig.database,
   host: redisConfig.host,
   port: redisConfig.port
  }),
  RmqModule.registerAsync({
   config: { ...rmqConfig },
   queueName: Queue.Specimen
  }),
  JobRepoModule,
  ResourceRepoModule,
  LoggerModule.forRoot({ serviceName: 'specimen_api' }) // use winston logger for logging
 ],
 controllers: [SpecimenController],
 providers: [SpecimenService]
})
export class SpecimenModule implements NestModule {
 // use middleware logger
 configure(consumer: MiddlewareConsumer) {
  consumer.apply(LoggerMiddleware).forRoutes('*')
 }
}
```

Файл specimen.controller.ts містить лише один клас SpecimenController, котрий наслідує один спільний клас ResourceController, що використовує інші

контролери. ResourceController має основну конфігурацію, що наслідує методологію CRUD (Create, Read, Update, Delete).

Лістинг класу SpecimenController:

```
@Controller('specimen')
export class SpecimenController extends ResourceController<ISpecimen>() {
  constructor(service: SpecimenService) {
    super(service)
  }
}
```

Лістинг класу ResourceController:

```
@Controller()
  class ResourceController {
    constructor(public readonly service: BaseService<T>) {}

    @Post()
    @HttpCode(HttpStatus.ACCEPTED)
    async create(
      @Body() resource: T,
      @Headers('x-original-url') url: string
    ): Promise<JobEntity | Error> {
      return await this.service.create(resource)
    }

    @Get(':id')
    async get(@Param('id', ParseUUIDPipe) id: string): Promise<T | undefined> {
      return await this.service.get(id)
    }

    @Get('_exists/:id')
    async exists(@Param('id', ParseUUIDPipe) id: string): Promise<boolean> {
      return await this.service.exists(id)
    }

    @Get(':id/_history/:version')
    async history(
      @Param('id', ParseUUIDPipe) id: string,
      @Param('version', ParseIntPipe) version: number
    ): Promise<T | undefined> {
      return await this.service.history(id, version)
    }

    @Put(':id')
    @HttpCode(HttpStatus.ACCEPTED)
```

```
  async update(
    @Param('id', ParseUUIDPipe) id: string,
    @Body() resource: T
  ): Promise<JobEntity | Error> {
    return await this.service.update(id, resource)
  }

  @Delete(':id')
  async delete(@Param('id', ParseUUIDPipe) id: string): Promise<void> {
    return await this.service.delete(id)
  }
}
```

Файл specimen.service.ts містить один клас SpecimenService, котрий наслідує один спільний клас BaseService, що використовують інші сервіси. BaseService має конфігурацію, що імплементує відповідні методі класу ResourceController.

Лістинг класу SpecimenService:

```
@Injectable()
export class SpecimenService extends BaseService<ISpecimen> {
  constructor(
    specimenRepo: ResourceRepository<SpecimenEntity>,
    historyRepo: ResourceHistoryRepository<SpecimenHistoryEntity>,
    redis: RedisRepository,
    rabbit: RmqService,
    jobRepo: JobRepository,
    @Inject(WINSTON_MODULE_PROVIDER) logger: LoggerService
  ) {
    super(specimenRepo, historyRepo, redis, rabbit, jobRepo, logger)
  }
}
```

Лістигнг класу BaseService:

```
export abstract class BaseService<T extends CommonResource> {
  constructor(
    private readonly genericRepository: ResourceRepository<ResourceEntity<T>>,
    private readonly historyRepository: ResourceHistoryRepository<
      ResourceHistoryEntity<T>
    >,
    private readonly redisRepo: RedisRepository,
    private readonly rabbitRmq: RmqService,
    private readonly jobRepo: JobRepository,
    @Inject(WINSTON_MODULE_PROVIDER) private readonly logger: LoggerService
```

```
) {}

async create(entity: T): Promise<JobEntity | Error> {
  this.logger.log('debug', `Creating new resource`)
  try {
    const id = v4()
    await this.redisRepo.set(id, entity)

    const job = { id }

    this.jobRepo.insertOne({ ...job, id } as JobEntity & BasePayload)

    this.rabbitRmq.emit(Pattern.Resource_Create, job)

    return job as JobEntity
  } catch (error) {
    throw new BadGatewayException(error)
  }
}

async get(id: string): Promise<T | undefined> {
  this.logger.log('debug', `Getting resource with id: ${id}`)
  const value = await this.genericRepository.findOneById(id)

  //check if entity exists
  if (value === null) throw new NotFoundException()

  return value?.payload
}

async history(id: string, version: number): Promise<T | undefined> {
  this.logger.log('debug', `Getting resource with id: ${id} and version: ${version}`)
  const value = await this.historyRepository.findOneByVersionInfo({
    resource_id: id,
    version
  })

  //check if entity exists
  if (value === null) throw new NotFoundException()

  return value?.payload
}

async exists(id: string): Promise<boolean> {
  return this.genericRepository.exists(id)
}
```

```
  async update(id: string, resource: T): Promise<JobEntity | Error> {
    this.logger.log('debug', `Updating resource with id: ${id}`)
    await this.redisRepo.set(id, resource)

    const job = { id } as JobEntity & BasePayload

    await this.jobRepo.insertOne(job)

    this.rabbitRmq.emit(Pattern.Resource_Update, job)

    return { id } as JobEntity
  }

  async delete(id: string): Promise<void> {
    this.logger.log('debug', `Deleting resource with id: ${id}`)
    //check if entity exists
    await this.checkIfEntityExists(id)

    try {
      this.genericRepository.deleteOneById(id)
    } catch (error) {
      throw new BadGatewayException(error)
    }
  }

  private async checkIfEntityExists(id: string): Promise<boolean> {
    const entity = await this.genericRepository.findOneById(id)

    if (entity === null) throw new NotFoundException()

    return true
  }
}
```

ВИСНОВОК

У кваліфікаційній роботі у рамках огляду та аналізу проблеми в системах обліку пацієнтів у медичних установах виявлено ключові аспекти, що визначають необхідність розробки нового підходу. Нинішні інформаційні системи часто стикаються з викликами, такими як обмежена масштабованість, низька ізоляція, та неефективна взаємодія між різними компонентами. Ці недоліки можуть призводити до порушень безпеки та неефективного використання ресурсів.

Аналіз літератури підтверджує, що мікросервісна архітектура виявляється перспективною стратегією для оптимізації функціонування систем у сфері охорони здоров'я. Вона надає можливість створювати високомасштабовані, гнучкі та високопродуктивні інформаційні системи, що можуть ефективно взаємодіяти з іншими елементами екосистеми.

Вибір методології та технічних засобів у досліджені визначив успішність впровадження мікросервісної архітектури в систему обліку пацієнтів. Один із стратегічних рішень стосується використання стандарту FHIR для визначення структури даних та інтерфейсів сервісів, що надає гнучкість та стандартизований підхід до обміну медичною інформацією між сервісами, сприяючи їхній взаємодії та інтеграції.

У розробці бекенду використано NestJS, вибір якого обгрунтований його модульністю, зручністю для тестування та можливістю розширення. Крім того, NestJS сприяє застосуванню принципів Dependency Injection та забезпечує структурованість коду. В якості брокера повідомлень обрано RabbitMQ, що забезпечує надійну передачу повідомлень між сервісами. Це дозволяє забезпечити асинхронну взаємодію між сервісами та підтримує принципи локальної обробки подій.

В цілому, вплив запропонована розробка має покращити якість медичних послуг та сприяти розвитку сучасного медичного обліку. Представлене моделювання мікросервісної архітектури дозволить покращити роботу системи,

забезпечивши швидкий обмін та доступ до медичної інформації. Оптимізація масштабованості та гнучкості системи стали ключовими факторами для забезпечення стабільності та адаптивності до змін у медичній сфері.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Robert C. Martin Clean Architecture / M. Robert, 2018 – 400p.
2. Рисунок «Структура мікросервісів» - URL: https://medium.com/@IvanZmerzlyi/microservice-architecture-f8a382291ff4
3. Рисунок «Вертикальне та горизонтальне масштабування» - URL: https://www.researchgate.net/publication/348540041_A_Distributed_Object_Model_for_the_Java
4. Офіційна сторінка Microsoft DevSecOps [Електронний ресурс] – URL: https://www.microsoft.com/en-us/security/business/security-101/what-is-devsecops (дата звернення: 01.12.2023)
5. Рисунок «Інтерфейс інструменту Grafana» - URL: https://medium.com/@gopalkrushnapattanaik/monitoring-spring-boot-services-using-micrometer-prometheus-grafana-30106774372d
6. Офіційна сторінка JWT [Електронний ресурс] – URL: https://jwt.io/introduction (дата звернення: 01.12.2023)
7. Офіційна сторінка FHIR [Електронний ресурс] – URL: https://www.hl7.org/fhir/ (дата звернення: 01.12.2023)
8. Офіційна сторінка NestJS [Електронний ресурс] – URL: https://nestjs.com / (дата звернення: 06.12.2023)
9. Офіційна сторінка RabbitMQ [Електронний ресурс] – URL: https://www.rabbitmq.com / (дата звернення: 06.12.2023)
10. Офіційна сторінка Redis [Електронний ресурс] – URL: https://redis.io (дата звернення: 10.12.2023)
11. Офіційна сторінка MongoDB [Електронний ресурс] – URL: https://www.mongodb.com/ (дата звернення: 09.12.2023)