КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота
на здобуття ступеня бакалавра

за спеціальністю 121 Інженерія програмного забезпечення

на тему:

РОЗРОБКА ВОЛОНТЕРСЬКОГО ВЕБ ЗАСТОСУНКУ СИСТЕМИ МАСОВОГО ОБСЛУГОВУВАННЯ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

Виконав студент 4-го курсу
Максим КОВАЛЬЧУК _____
(підпис)

Науковий керівник:
асистент, кандидат фіз.-мат. наук
Костянтин ЖЕРЕБ _____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем
« 25 » травня 2022 р.,
Протокол №10
Завідувач кафедри
Олександр ПРОВОТАР _____
(підпис)

Київ – 2022

РЕФЕРАТ

Обсяг роботи: 29 сторінок, 15 використаних джерел, 9 рисунків.

Ключові слова: МІКРОСЕРВІС, МОНОЛІТ, СИСТЕМА МАСОВОГО ОБСЛУГОВУВАННЯ, СЕРВІСНО-ОРІЄНТНОВНА АРХІТЕКТУРА, DOCKER, KAFKA, TYPESCRIPT, NODE.JS, NEST.JS, ANGULAR, POSTGRESQL, REDIS.

Об’єктом роботи є дослідження систем масового обслуговування з використанням мікросервісів.

Предметом роботи є створення веб застосунку з мікросервісною архітектурою мовою TypeScript (Node.js) з використанням Kafka (як брокера повідомлень), PostgreSQL (як бази даних), Redis (як кешу) та популярних фреймворків – Nest.js та Angular.

Інструментом розробки обрано VSCode – сучасний редактор програмного коду, зручний для крос-платформної розробки веб-застосунків. Редактор містить дебагер, розширення для системи контролю версій Git, форматування та підсвічування коду. Мова програмування в середовищі розробки – TypeScript.

Результатом роботи є аналіз переваг і недоліків запропонованих архітектурних підходів, ключових можливостей допоміжних програм та технологій, використаних у роботі. Практична частина являє собою клієнт-серверний застосунок, взаємодія з яким можлива через веб інтерфейс будь-якого сучасного браузера.

ЗМІСТ

ВСТУП

Оцінка сучасного стану роботи. Під час роботи з сучасними технологіями розробки програмних систем виникає вибір стосовно архітектури, мови програмування, бібліотек. Існують різні підходи до побудови архітектури застосунків, такі як моноліт та мікросервіси. При мікросервісному підході єдиний застосунок проектується як сукупність невеликих, самодостатніх, незалежних, слабо зв'язаних сервісів, що обмінюються інформацією між собою. У кваліфікаційній роботі розглядаються конкретні приклади застосування мікросервісів, які дозволяють оптимально вирішувати задачі розробки програмних застосунків. Далі буде розглянуто реалізацію системи масового обслуговування з використанням мікросервісної архітектури.

Актуальність роботи та підстави для її виконання. Використання і популярність мікросервісів пояснюється можливістю гнучких змін під час розробки ПЗ. До гнучких змін належить можливість оптимального керування витратами ресурсів порівняно з монолітами, коли система будується, як єдине ціле. У кваліфікаційній роботі розглядаються конкретні приклади застосування мікросервісів, які дозволяють оптимально вирішувати задачі розробки програмних застосунків.

Мета й завдання роботи. Проектування та розробка веб-орієнтованої системи масового обслуговування на основі мікросервісної архітектури. Для досягнення цієї мети поставлено такі завдання:

- Аналіз переваг і недоліків мікросервісів
- Аналіз існуючих підходів до побудови мікросервісної архітектури
- Формування вимог до функціоналу системи масового обслуговування
- Вибір технологій для реалізації
- Проектування архітектури
- Розробка прототипу системи масового обслуговування

Об’єкт, методи та засоби дослідження. Об’єктом роботи є дослідження систем масового обслуговування з використанням мікросервісів.

Розробці програмного засобу передував аналіз переваг і недоліків архітектурних підходів до побудови програмних застосунків, ключових можливостей допоміжних програм та технологій, використаних у роботі. Практична частина являє собою клієнт-серверний застосунок, взаємодія з яким можлива через веб інтерфейс будь-якого сучасного браузера.

Інструментом розробки обрано VSCode – сучасний редактор програмного коду, зручний для крос-платформної розробки веб-застосунків. Редактор містить дебагер, розширення для системи контролю версій Git, форматування та підсвічування коду. Мова програмування в середовищі розробки – TypeScript.

Можливі сфери застосування. Розроблена система масового обслуговування може використовуватись волонтерами для зв’язку з людьми, які потребують допомоги. Запропоновану мікросервісну архітектуру та патерни проектування можна використовувати в подальшому, як шаблон для розробки мікросервісних веб застосунків.

РОЗДІЛ 1. ЗАГАЛЬНИЙ ОГЛЯД МІКРОСЕРВІСІВ

1.1 ІСТОРІЯ ВИНИКНЕННЯ МІКРОСЕРВІСІВ

Історія та походження мікросервісів – це постійні покращення для забезпечення кращого зв'язку між різними платформами, більшої простоти та зручніших для користувача систем. Вперше термін "мікро-веб-сервіси" використав Пітер Роджерс у 2005 році під час своєї презентації з хмарних обчислень [1]. Роджерс виступав проти традиційного мислення в рамках монолітної архітектури та запропонував програмні компоненти, що підтримують мікро-веб-сервіси. У презентації він створив функціональну модель мікросервісів, яка стала реальністю. Він описав, як добре організована платформа мікро-веб-сервісів «застосовує базові архітектурні принципи веб-сервісів та REST-сервісів разом з Unix-подібним плануванням та конвеєрами, щоб забезпечити високу гнучкість та підвищену простоту в сервіс-орієнтованих архітектурах».

Мікросервісна архітектура — це архітектура, при якій єдиний додаток проектується як сукупність невеликих, самодостатніх, незалежних, слабо пов’язаних сервісів, які комунікують через синхронні протоколи, такі як HTTP/REST, або асинхронні, такі як AMQP [2]. Мікросервіси являють собою один із підходів до реалізації сервісно-орієнтованих архітектур (SOA), що використовуються для побудови програмних систем, які розгортаються незалежно одна від одної. Самі по собі мікросервіси можуть бути написані на різних мовах і використовувати різні технології.

Загалом, мікросервіси є першою реалізацією SOA, яка пішла після впровадження DevOps [3] і стає все більш популярною в проектах, які розробляються з використанням Agile [4] методологій. Цей підхід довів свою високу ефективність, особливо для великих корпоративних додатків, які розробляються незалежними групами розробників. У лютому 2020 року звіт про дослідження ринку хмарних мікросервісів [5] передбачав, що розмір глобального

ринку архітектури мікросервісів зросте на 21,37% з 2019 по 2026 рік і досягне 3,1 мільярда доларів до 2026 року.

1.2 ПЕРЕВАГИ ТА НЕДОЛІКИ МІКРОСЕРВІСІВ

До переваг мікросервсіної архітектури можна віднести:

1. Забезпечує безперервну доставку та розгортання великих складних програм.
2. Покращена ремонтопридатність – кожна служба відносно невелика, тому її легше зрозуміти та змінити.
3. Краща тестованість – служби менші та швидше тестуються.
4. Краще розгортання – служби можна розгортати незалежно одна від одної. Кожна команда може розробляти, тестувати, розгортати та масштабувати свої послуги незалежно від усіх інших команд.
5. Покращена ізоляція несправностей. Наприклад, якщо в одній службі є витік пам’яті, це вплине лише на цю службу. Інші служби продовжуватимуть обробляти запити. Для порівняння, один компонент монолітної архітектури, що погано працює, може вивести з ладу всю систему.
6. Усуває будь-які довгострокові зобов’язання щодо стека технологій. При розробці нового сервісу можна вибрати новий технологічний стек. Аналогічно, вносячи серйозні зміни в існуючий сервіс, можна переписати його, використовуючи новий технологічний стек.

До недоліків можна віднести:

1. Розробники повинні впровадити механізм міжсервісного зв’язку та впоратися з частковими збоями.
2. Реалізувати запити, які охоплюють декілька служб, складніше.
3. Тестувати взаємодію між службами складніше.
4. Реалізація запитів, які охоплюють декілька служб, вимагає ретельної координації між розподіленими командами розробників.

1.3 ВИКОРИСТАННЯ МІКРОСЕРВІСІВ ЯК НАОЧНОГО ПРИКЛАДУ СИСТЕМИ МАСОВОГО ОБСЛУГОВУВАННЯ

Систему масового обслуговування [6] можна описати як систему, що має об'єкт обслуговування, до якого прибувають певні одиниці (загалом звані «клієнтами») для обслуговування; всякий раз, коли в системі є більше одиниць, ніж прилади обслуговування можуть обробляти одночасно, виникає черга (або черга очікування). Підрозділи очікування приймають свою чергу на обслуговування за заздалегідь встановленим правилом, а після обслуговування залишають систему. Таким чином, вхід до системи складається з клієнтів, які вимагають обслуговування, а на виході – обслуговувані клієнти.

Систему масового обслуговування зазвичай характеризують такими термінами:

1. Процес введення

   Нехай клієнти прибувають у моменти t0, t1, t2,…; тоді час між прибуттям становить: u(r) = t(r) - t(r-1), r = 1, 2, 3 ...

   Випадкові величини u(r), загалом, вважаються статистично незалежними, а їх розподіл ймовірностей A(u) називається розподілом часу між прибуттям або, просто, розподілом прибуття або розподілом вхідних даних. Клієнти можуть надходити з нескінченного джерела, як у випадку телефонних дзвінків і як передбачається в багатьох дослідженнях черг, або з кінцевого джерела. Вони можуть надходити поодинці або групами фіксованих або різних розмірів. Система масового обслуговування може мати верхню межу кількості, яку можна допустити в систему, як у випадку кінцевого простору очікування.

2. Дисципліна черги

   Це можна описати як правило, що визначає формування черги або черг і спосіб, яким клієнт або клієнти вибираються для обслуговування з тих, хто чекає.

   Найпоширеніша дисципліна черги – «першим прийшов – першим обслужений», згідно з якою підрозділи надходять на службу в порядку їх

прибуття. Інші можливості – це випадковий вибір для обслуговування, правило пріоритету або навіть правило «останній прийшов, перший обслужений». У разі пріоритету може бути два класи, а саме: пріоритетний і непріоритетний, або може бути кілька класів пріоритету, що представляють різні рівні пріоритету. Крім того, може існувати випереджувальна пріоритетна дисципліна, згідно з якою підрозділ з нижчим пріоритетом виводиться з експлуатації щоразу, коли прибуває блок з вищим пріоритетом, при цьому обслуговування блоку з випередженням відновлюється лише тоді, коли в системі немає одиниць з вищим пріоритетом. В протилежність цьому є невипереджувальне правило пріоритету, або правило пріоритету, за яким пріоритети враховуються лише на початку служби, а після початку обслуговування продовжується до завершення. У цей заголовок прийнято включати явища черги, такі як відмову та відмову, що відображають поведінку клієнтів, які чекають.

Кажуть, що клієнти заперечуються, коли, дивлячись на розмір черги, а потім оцінюючи час, який їм, можливо, доведеться чекати перед обслуговуванням, вони не приєднуються до черги. Кажуть, що після приєднання до черги клієнти відмовляються, якщо їм не терпиться чекати, і залишають чергу до початку обслуговування.

3. Механізм обслуговування.

Час, який проходить під час обслуговування підрозділу, називається часом його обслуговування. Вважається, що час обслуговування v1, v2, v3,… послідовних одиниць не залежить один від одного та від вхідного розподілу, а їх розподіл ймовірностей B(v) називається розподілом часу обслуговування або, коротко, розподілом обслуговування. . Специфікація механізму обслуговування включає кількість серверів. Таким чином, розрізняють односерверні та багатосерверні системи. У системі масового обслуговування послуги можуть надаватися партіями фіксованих або різних розмірів.

Система масового обслуговування потрібна в кожній галузі, від роздрібної торгівлі до освіти, і ця потреба стає все сильніше, оскільки відвідувачі прагнуть більшого комфорту.

Управління чергами зосереджується на досвіді клієнтів, але цінність системи черг не обмежується вирішенням черг. Це допомагає суттєво зменшити час очікування та обслуговування клієнтів, підвищити ефективність обслуговування та персоналу, збільшуючи тим самим дохід.

РОЗДІЛ 2. ОПИС ОБРАНИХ ТЕХНОЛОГІЙ

2.1 ANGULAR FRONTEND FRAMEWORK

Angular – це фреймворк для побудови клієнтських додатків за допомогою HTML, JavaScript та TypeScript.

Angular складається з декількох бібліотек, деякі з них основні, а деякі – необов'язкові. Основними складовими частин програми Angular є NgModules, які забезпечують контекст компіляції компонентів. NgModules збирають відповідний код у функціональні набори. У додатку завжди є принаймні кореневий модуль AppModule, який, як правило, містить у собі ще багато інших функціональних модулів.

Компоненти визначають представлення даних, що представляють собою набори елементів екрану, які можна вибирати та змінювати відповідно до логіки та даних конкретної програми.

Компоненти використовують сервіси, які надають певні функціональні можливості, безпосередньо не пов'язані з переглядами. Постачальники послуг можуть бути введені в компоненти як залежності, що робить ваш код модульним, багаторазовим та ефективним.

Angular – додатки є модульними, а Angular має власну модульну систему, що називається Angular – модулі або NgModules.

На рис.1 можна побачити внутрішню будову Angular фреймворку. Компоненти і сервіси - це просто класи, в яких є декоратори, які позначають їх тип та надають метадані, які вказують Angular, як ними користуватися.

Метадані класу компонентів асоціюють його з шаблоном, який визначає представлення. Шаблон поєднує звичайний HTML з директивами та розміткою, що дозволяють Angular змінювати HTML, перш ніж відображати його для користувача.

Метадані для класу обслуговування надають інформацію, яку Angular потребує, щоб зробити її доступною для компонентів через введення залежності DI (Dependency Injection) [7].

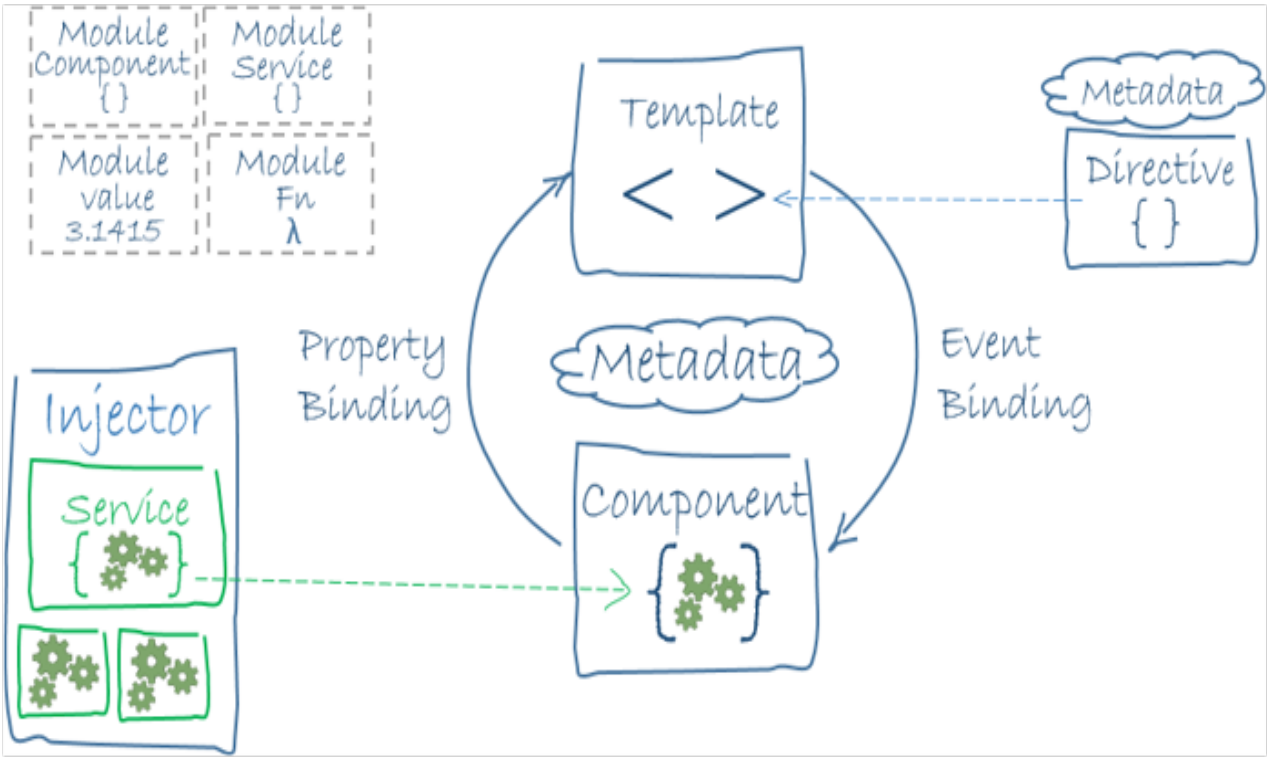

рис.1 Внутрішня будова Angular

Компоненти програми зазвичай визначають багато представлень, розташованих ієрархічно. Angular надає послугу маршрутизатора, щоб допомогти визначати шляхи навігації серед представлень. Маршрутизатор забезпечує складні навігаційні можливості в браузері.

До переваг Angular можна віднести:

1. Використовується разом з TypeScript. Має вбудовану підтримку для цього.
2. Детальна документація, що дозволяє розробнику отримати всю необхідну інформацію, не вдаючись до допомоги його колег.
3. Одностороння прив'язка даних, яка забезпечує виняткову поведінку додатка, що зводить до мінімуму ризик можливих помилок.
4. Структура і архітектура, спеціально створені для великої масштабованості проекту.

2.2 NEST.JS BACKEND FRAMEWORK

Nest (NestJS) – це платформа для створення ефективних масштабованих серверних додатків на Node.js. Всередині він використовує прогресивний JavaScript, побудований на TypeScript і повністю підтримує його (але при цьому дозволяє розробникам писати код і на чистому JavaScript) і поєднує в собі елементи ООП (об'єктно–орієнтоване програмування), FP (функціональне програмування) і FRP (функціонально–реактивне програмування).

Під капотом Nest використовує надійні фреймворки HTTP-серверів, такі, як наприклад Express [8] (за замовчуванням) або Fastify.

Nest.js забезпечує рівень абстракції вище загальних фреймворків Node.js (наприклад, Express), але також надає їх API безпосередньо розробнику. Це дає розробникам свободу використовувати безліч сторонніх модулів, доступних для базової платформи.

В останні роки, завдяки Node.js, JavaScript став «мовою спілкування» в Інтернеті як для зовнішніх, так і для серверних додатків. Це призвело до появи великої кількості проектів, які підвищують продуктивність розробників і дозволяють створювати швидкі, тестовані і розгортаємі програми. Однак, хоча для Node.js (і серверного JavaScript) існує безліч чудових бібліотек і допоміжних інструментів, жоден з них повністю не вирішує основну проблему – архітектурну.

Nest.js надає готову архітектуру додатків, яка дозволяє розробникам і командам створювати добре тестовані, масштабовані, модульні і слабо пов'язані застосунки.

Архітектура в значній мірі була натхненна Angular. Так само в основі фреймворку лежать декоратори, модульна структура, DI, сервіси та контролери.

Nest.js має вбудовану інтеграцію для усіх популярних брокерів повідомлень, що робить його чудовим варіантом при розробці застосунків з мікросервісною архітектурою.

2.3 APACHE KAFKA BROKER

Apache Kafka – це проект з відкритим вихідним кодом для розподіленої системи обміну повідомленнями публікації-підписки, переосмислений як розподілений журнал фіксації. Написаний на мовах програмування Java та Scala.

На рис.2 зображено внутрішню будову брокеру Kafka. Вона зберігає повідомлення в розділах, які розділені і реплікуються між декількома брокерами в кластері. Виробники (Producers) розсилають повідомлення за темами, які читають споживачі (Consumers).

Споживачі можуть бути згруповані в групу споживачів. Кожен споживач в групі буде читати повідомлення з унікальної підмножини розділів в кожній темі (topic), на яку вони підписані. Кожне повідомлення доставляється одному споживачеві в групі, і всі повідомлення з одним і тим же ключем приходять одному і тому ж споживачеві.

Kafka незалежна від мови – виробники і споживачі використовують бінарний протокол для взаємодії з кластером Kafka.

Повідомлення є байтовими масивами (найбільш поширеними форматами є String, JSON і Avro [9]). Якщо у повідомлення є ключ, Kafka гарантує, що всі повідомлення з одним і тим же ключем знаходяться в одному розділі (partition).

Kafka не відслідковує, які повідомлення були прочитані кожним споживачем. Вона зберігає всі повідомлення протягом обмеженого часу, і відповідальність за відстеження їх місця розташування по кожній темі, тобто зміщення (offset), лежить на самих споживачах.

Kafka має високу продуктивність. Вона забезпечує високу пропускну здатність як для публікації, так і для підписки, використовуючи дискові структури, які здатні забезпечити постійний рівень продуктивності, навіть коли йдеться мова про багато терабайт збережених повідомлень.

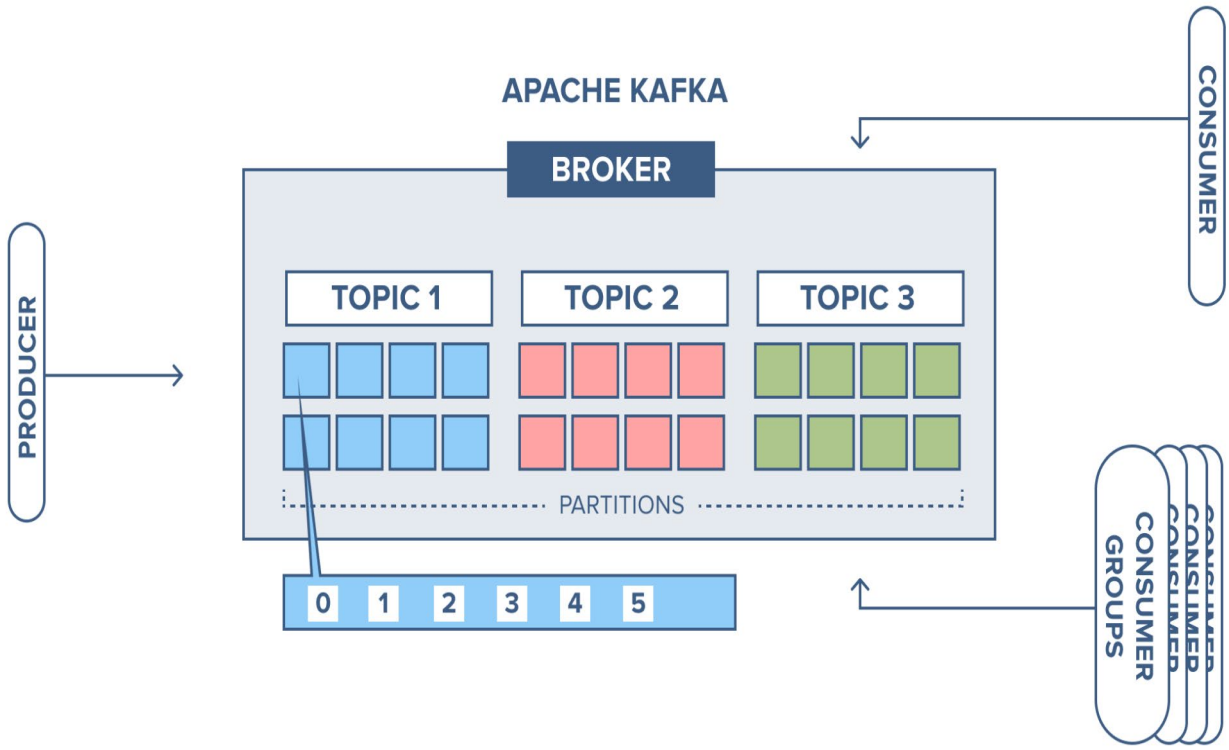

рис.2 Внутрішня будова брокеру Kafka

До переваг Kafka можна віднести:

1. Гарна масштабованість. Kafka - це розподілена система, яку можна швидко та легко масштабувати, не вимагаючи простою. Вона здатна обробляти багато терабайт даних, не витрачаючи на це зайвих витрат.
2. Kafka зберігає повідомлення на дисках, що забезпечує реплікацію всередині кластера. Це створює довговічну та надійну систему обміну повідомленнями.
3. Kafka копіює дані і може підтримувати одразу декількох абонентів. Крім того, вона автоматично балансує споживачів у разі несправності. Це означає, що вона надійніша, ніж подібні доступні служби обміну повідомленнями.

Kafka є одним з найбільш швидкодійних брокерів повідомлень. На рис.3 можна побачити, що в порівнянні з іншими популярними брокерами RabbitMQ та Pulsar, Kafka забезпечує набагато кращу пропускну здатність при мінімальних затримках.

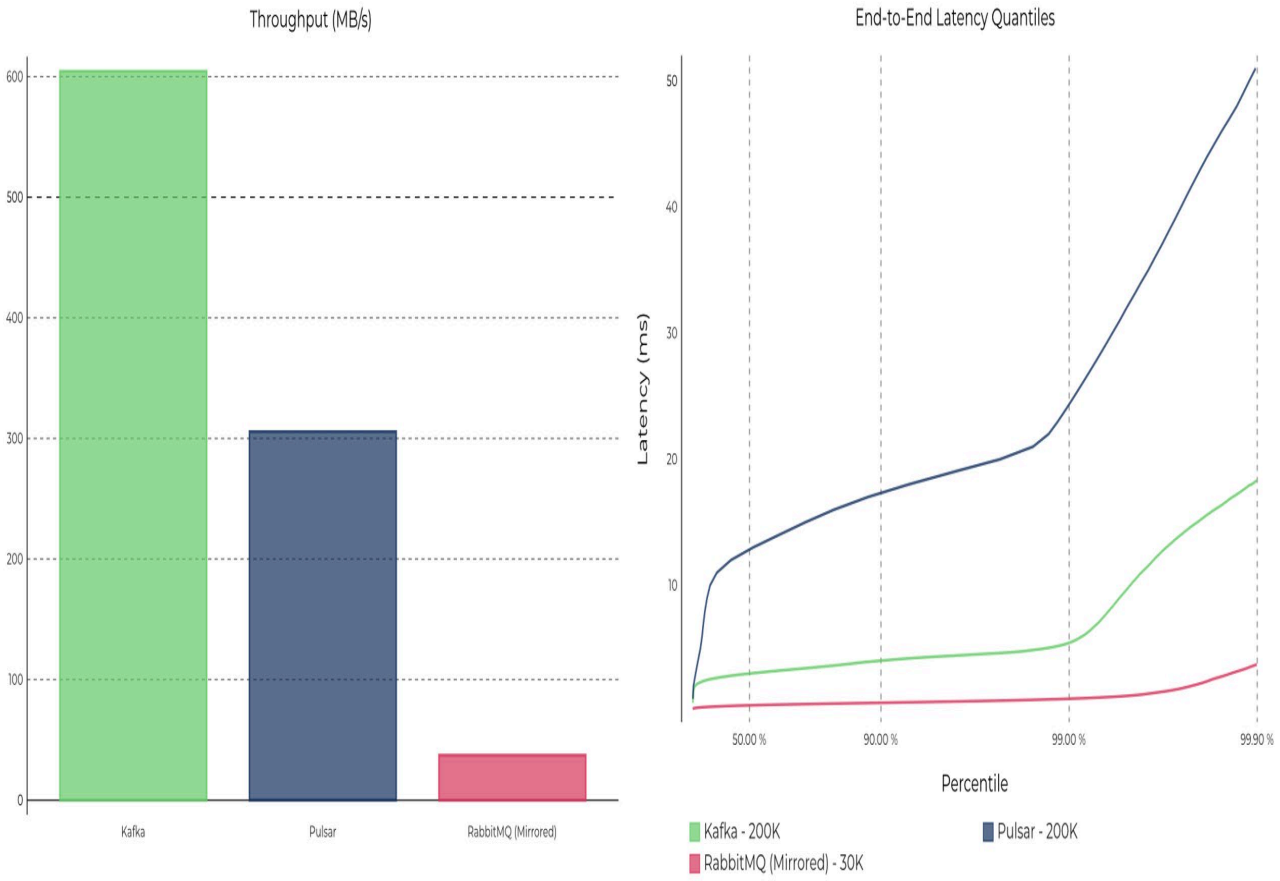

рис.3 Тест пропускної здібності брокерів

2.4 POSTGRESQL DATABASE

PostgreSQL – це потужна система об’єктно-реляційних баз даних з відкритим вихідним кодом, яка використовує та розширює мову SQL у поєднанні з багатьма функціями, які безпечно зберігають і масштабують робочі навантаження даних. Витоки PostgreSQL сягають 1986 року в рамках проекту POSTGRES в Каліфорнійському університеті в Берклі і має понад 30 років активної розробки на базовій платформі.

PostgreSQL заслужив репутацію завдяки своїй перевіреній архітектурі, надійності, цілісності даних, надійному набору функцій, розширюваності та спільноті відкритих вихідних кодів, які стоять за програмним забезпеченням, щоб постійно надавати продуктивні та інноваційні рішення. PostgreSQL працює на всіх основних операційних системах, має ACID-сумісність [10] і має власні додаткові доповнення, такі як популярний розширювач геопросторової бази

даних PostGIS. Не дивно, що PostgreSQL став реляційною базою даних з відкритим кодом, яку вибирають багато людей та організацій.

PostgreSQL постачається з багатьма функціями, які допомагають розробникам створювати програми, адміністраторам захищати цілісність даних і створювати відмовостійкі середовища, а також допомагають розробникам керувати даними незалежно від того, наскільки великий чи малий набір даних. Окрім того, що PostgreSQL є базою даних з безкоштовним і відкритим кодом, він добре розширюваний. Наприклад, можна визначати власні типи даних, створювати власні функції, писати код з різних мов програмування без перекомпіляції бази даних.

PostgreSQL підтримує лише один механізм зберігання даних, але розробляє новий механізм під назвою zheap, який призначений для оновлень на місці та повторного використання простору, щоб допомогти контролювати розмір бази даних.

PostgreSQL забезпечує високу швидкість як для читання, так і для запису. Він забезпечує паралельність без блокування читання/запису, що дозволяє PostgreSQL обробляти та коригувати декілька потоків читання та запису одночасно.

Загалом, масштабованість і висока продуктивність PostgreSQL робить його цінним у великих, складних проектах баз даних, у яких важлива висока швидкість читання та запису. При належній архітектурі PostgreSQL не має обмежень продуктивності.

2.5 REDIS CACHE

Redis — це сховище з відкритим вихідним кодом, сховище структури даних у пам’яті, яке використовується як база даних, кеш, посередник повідомлень і потокова система. Redis надає такі структури даних, як рядки, хеші, списки, набори, відсортовані набори із запитами діапазону, растрові зображення, гіперлогові журнали, геопросторові індекси та потоки. Redis має вбудовану реплікацію, скріпти Lua, транзакції та різні рівні збереження на диску,

а також забезпечує високу доступність через Redis Sentinel та автоматичне розбиття на розділи за допомогою Redis Cluster.

Щоб досягти найвищої продуктивності, Redis працює з набором даних у пам’яті. Залежно від вашого варіанту використання, Redis може зберігати ваші дані або періодично виписуючи набір даних на диск, або додаючи кожну команду до журнала на диску. Ви також можете вимкнути збереження, якщо вам просто потрібен багатофункціональний мережевий кеш у пам’яті.

Redis підтримує асинхронну реплікацію, швидку неблокуючу синхронізацію та автоматичне повторне підключення з частковою ресинхронізацією при розділенні мережі. Ви можете використовувати Redis з більшості мов програмування.

Redis написаний на ANSI C і працює на більшості систем POSIX, таких як Linux, *BSD і Mac OS X, без зовнішніх залежностей. Linux та OS X — це дві операційні системи, де Redis розробляється та тестується найбільше.

Варіанти використання Redis:

1. Сховище даних в режимі реального часу

   Універсальні структури даних Redis в пам’яті дозволяють створювати інфраструктуру даних для застосунків реального часу, які вимагають низької затримки та високої пропускної здатності.
2. Кешування та зберігання сесій

   Швидкість Redis робить його ідеальним для кешування запитів до бази даних, складних обчислень, викликів API та стану сеансу.
3. Потокове передавання та обмін повідомленнями

   Потокові типи даних забезпечують високу швидкість прийому інформації, обміну повідомленнями, пошуку подій і сповіщень.

Список відомих компаній, які наразі використовують Redis:

- Twitter
- Github
- Snapchat

- Craigslist
- StackOverflow

2.6 DOCKER CONTAINERIZATION

Docker — це програмна платформа, яка дозволяє швидко створювати, тестувати та розгортати програми. Docker упаковує програмне забезпечення в стандартизовані блоки, які називаються контейнерами, і які містять все необхідне для роботи програмного забезпечення, включаючи бібліотеки, системні інструменти, код і час виконання. Використовуючи Docker, можна швидко розгортати та масштабувати програми в будь-якому середовищі та знати, що код працюватиме правильно.

Docker є операційною системою для контейнерів. Подібно до того, як віртуальна машина віртуалізує (позбавляє від необхідності безпосереднього керування) обладнання сервера, контейнери віртуалізують операційну систему сервера. Docker встановлюється на кожному сервері і надає прості команди, які можна використовувати для створення, запуску або зупинки контейнерів.

Docker можна використовувати як основний будівельний блок для створення сучасних програм і платформ. Docker дозволяє легко створювати та запускати архітектури розподілених мікросервісів, розгортати програмний код за допомогою стандартизованих безперервних конвеєрів інтеграції та доставки, створювати високомасштабовані системи обробки даних та повністю керовані великі платформи.

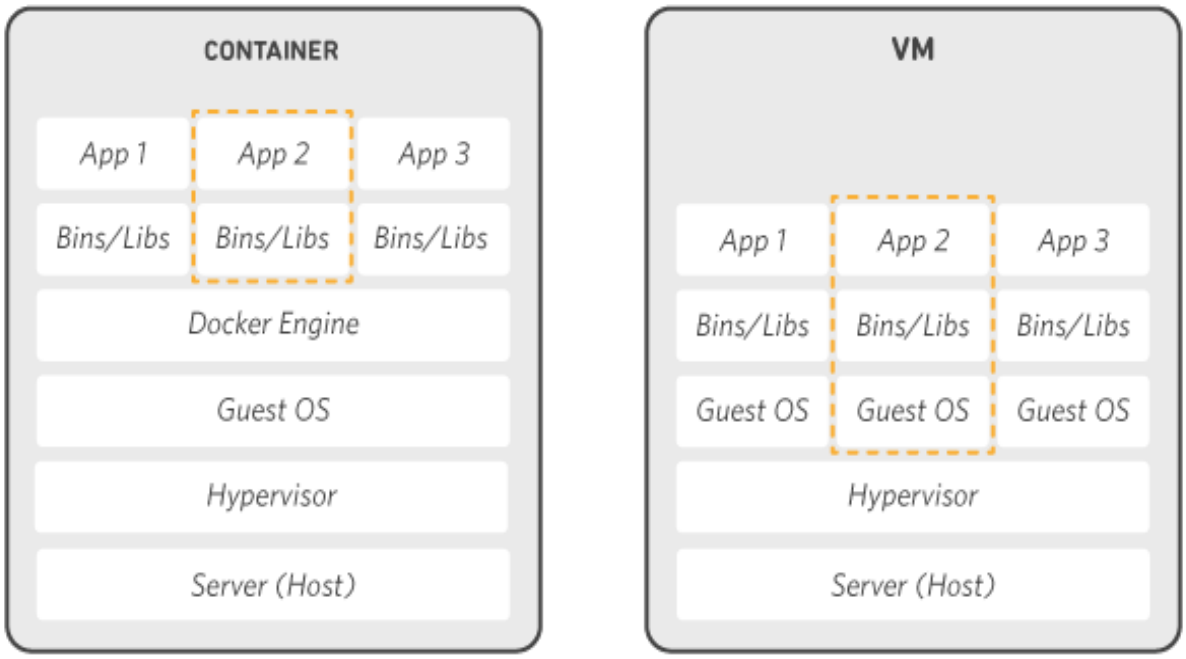

рис.4 Порівняння docker та virtual machine

Переваги Docker:

1. Невеликі контейнерні програми спрощують розгортання, виявлення проблем і відкат для усунення.
2. Застосунки на основі Docker можна плавно перенести з локальних машин розробки на виробниче розгортання на хмарному провайдері.
3. Контейнери Docker полегшують запуск більшої кількості коду на кожному сервері, оптимізуючи використання та заощаджуючи гроші.
4. Користувачі Docker в середньому поставляють програмне забезпечення в 7 разів частіше, ніж користувачі, які не використовують Docker. Docker дає змогу надсилати ізольовані послуги так часто, як це необхідно.

РОЗДІЛ 3. ПРОЕКТУВАННЯ ТА РОЗРОБКА ЗАСТОСУНКУ

3.1 ФОРМУВАННЯ ВИМОГ ДО ФУНКЦІОНАЛУ ЗАСТОСУНКУ

Стоїть задача реалізувати систему масового обслуговування з використанням мікросервісів, для того, щоб об’єднати волонтерів та людей, які потребують допомоги.

В застосунку треба реалізувати наступне:

1. Логіку авторизації (регистрації та логіну, за номером телефону та паролем).
2. Можливість переглядати всі вільні запити в системі (для волонтерів).
3. Можливість взяти запит в опрацювання і переглядати свої поточні запити (для волонтерів).
4. Можливість створити новий запит (для звичайних користувачів).
5. Можливість переглядати всі свої запити і змінювати ті, які знаходяться в опрацюванні (для звичайних користувачів).

3.2 ПРОЕКТУВАННЯ ЗАГАЛЬНОЇ АРХІТЕКТУРИ

На рис. 5 можна побачити архітектурну діаграму системи і те, яким саме чином сервіси комунікують між собою.

Загалом, застосунок представлений 4 мікросервісами:

1. `vp-web-ui` - frontend сервіс, де реалізовано графічний інтерфейс. Отримує дані по HTTP з сервісу `vp-api-gateway`. До інших сервісів доступу не має.
2. `vp-api-gateway` - єдиний backend сервіс, до якого клієнт має доступ. Виступає в ролі проксі для делегування задач іншим backend сервісам. Саме тут зосереджена вся проміжна логіка обробки запитів (валідація, авторизація і т.д.). Не має власної бази даних. Отримує всі потрібні дані з `vp-request-service` та `vp-user-service` в синхронній моделі через Kafka протокол. Має доступ до Redis кешу, звідки дістає авторизаційні сесії користувачів.

3. `vp-request-service` - внутрішній backend сервіс, який відповідає за всі CRUD [11] операції, пов’язані з волонтерськими запитами. Має власну окрему PostgreSQL базу даних. Отримує асинхронні Kafka події з сервісу `vp-user-service`, для синхронизації та консистентності даних.
4. `vp-user-service` - внутрішній backend сервіс, який відповідає за всі CRUD операції, пов’язані з користувачами системи. Має власну окрему PostgreSQL базу даних. Відправляє асинхронні Kafka події до сервісу `vp-request-service`, для синхронизації та консистентності даних. Має доступ до Redis кешу, куди додає авторизаційні сесії користувачів.

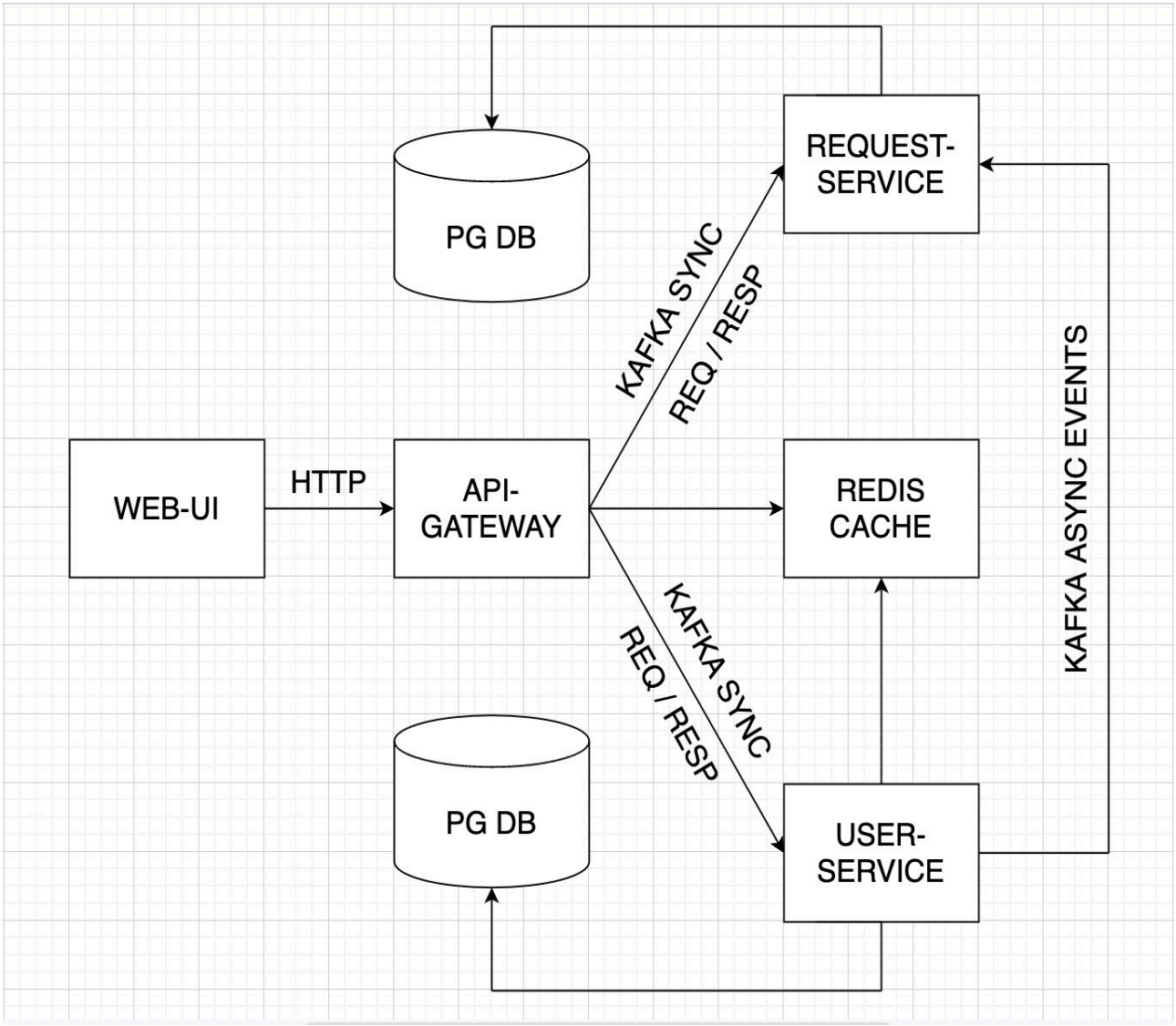

рис.5 Архітектурна діаграма застосунку

Під час розробки системи було використано наступні патерни мікросервісної архітектури:

1. API Gateway Pattern [12]

   Проблема: Як клієнти програми на основі мікросервісів мають отримувати доступ до даних окремих служб?

   Рішення: Реалізувати сервіс, який буде єдиною точкою входу для всіх клієнтів.

   Переваги:

   - Ізолює клієнтів від того, як програма поділяється на мікросервіси
   - Перекладає «стандартний» загальнодоступний протокол API на будь-які протоколи, які використовуються всередині
   - Спрощує роботу клієнта шляхом переміщення логіки виклику кількох служб від клієнта до сервісу Gateway API

2. Request-Reply Integration Pattern [13]

   Проблема: Коли сервіс надсилає повідомлення, як він може отримати відповідь від одержувача ?

   Рішення: Надіслати пару повідомлень "Запит-Відповідь", кожне на своєму окремому каналі.

   Переваги:

   - Можливість реалізувати синхронну модель request-response в нативно асинхронних системах
   - Краща можливість масштабування, аніж в синхронних системах

3. Event Sourcing Pattern [14]

   Проблема: Як надійно/атомарно оновлювати базу даних та асинхронно надсилати повідомлення/події ?

Рішення: Джерело подій зберігає стан об’єкта як послідовність подій, що змінюють стан. Щоразу, коли змінюється стан об’єкта, до списку подій додається нова подія, яка надсилається всім залежним сервісам.

Переваги:

- Дає можливість надійно публікувати події і підтримувати консистентність [15] даних, коли змінюється стан об’єкту
- Забезпечує надійний журнал аудиту змін, внесених до об’єкта
- Бізнес-логіка на основі джерела подій складається з слабо пов’язаних частин, які обмінюються подіями

3.3 ОПИС СТВОРЕНОЇ СИСТЕМИ

Для волонтерів в застосунку було розроблено:

- Сторінку для перегляду всіх актуальних запитів в системі, з можливістю взяти запит в обробку чи відмовитись від нього. Інтерфейс цієї сторінки зображено на рис.6

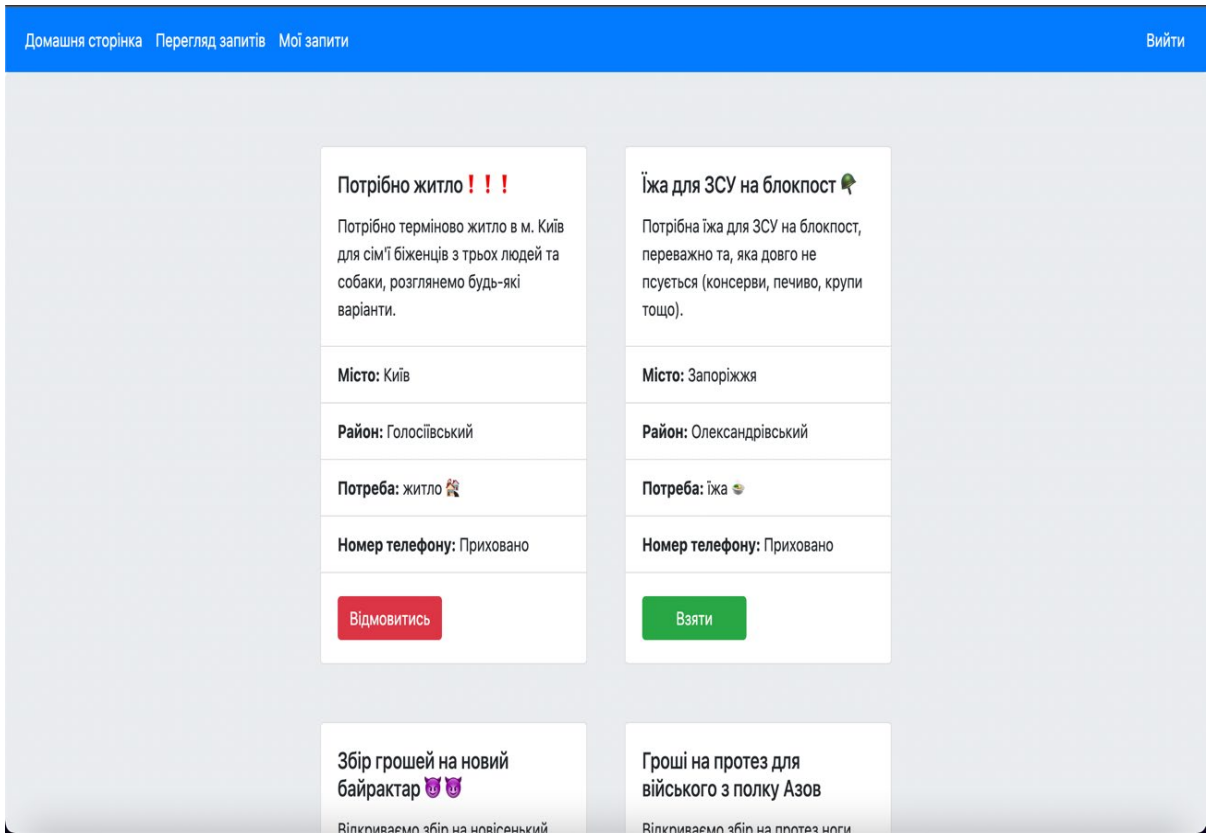

рис.6 Сторінка волонтера для перегляду запитів в системі

- Сторінку для перегляду запитів, які наразі находяться в опрацюванні

волонтером. Є можливість завершити виконання запиту (відмітити його як виконаний) чи відмовитись. Інтерфейс цієї сторінки зображено на рис.7

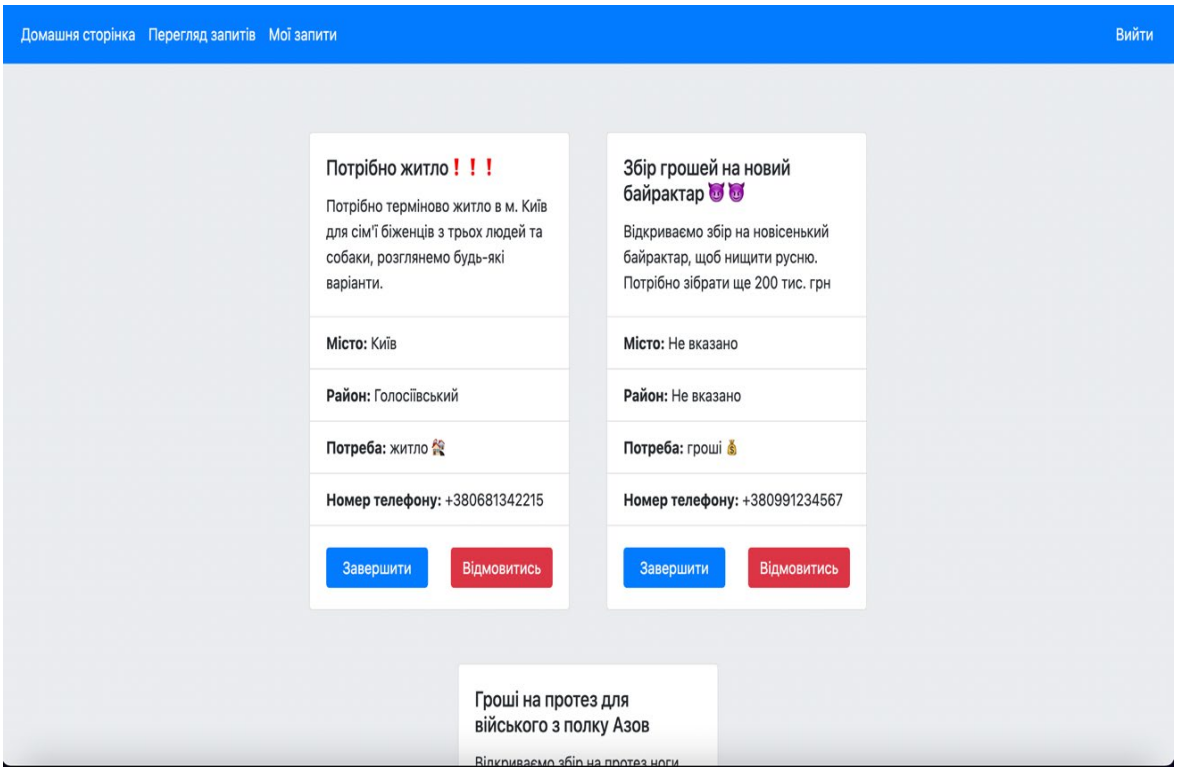

рис.7 Сторінка волонтера для перегляду запитів, які наразі знаходяться на опрацюванні

Для користувачів в застосунку було розроблено:

- Сторінку з можливістю сформувати новий запит з певною нуждою. Ця сторінка являє собою форму, в якій треба надати усю необхідну інформацію для волонтера. Інтерфейс цієї сторінки зображено на рис. 8
- Сторінку для перегляду своїх створених запитів, з можливістю перегляду їх статусу та редагування запитів, які ще не виконані. Інтерфейс цієї сторінки зображено на рис. 9

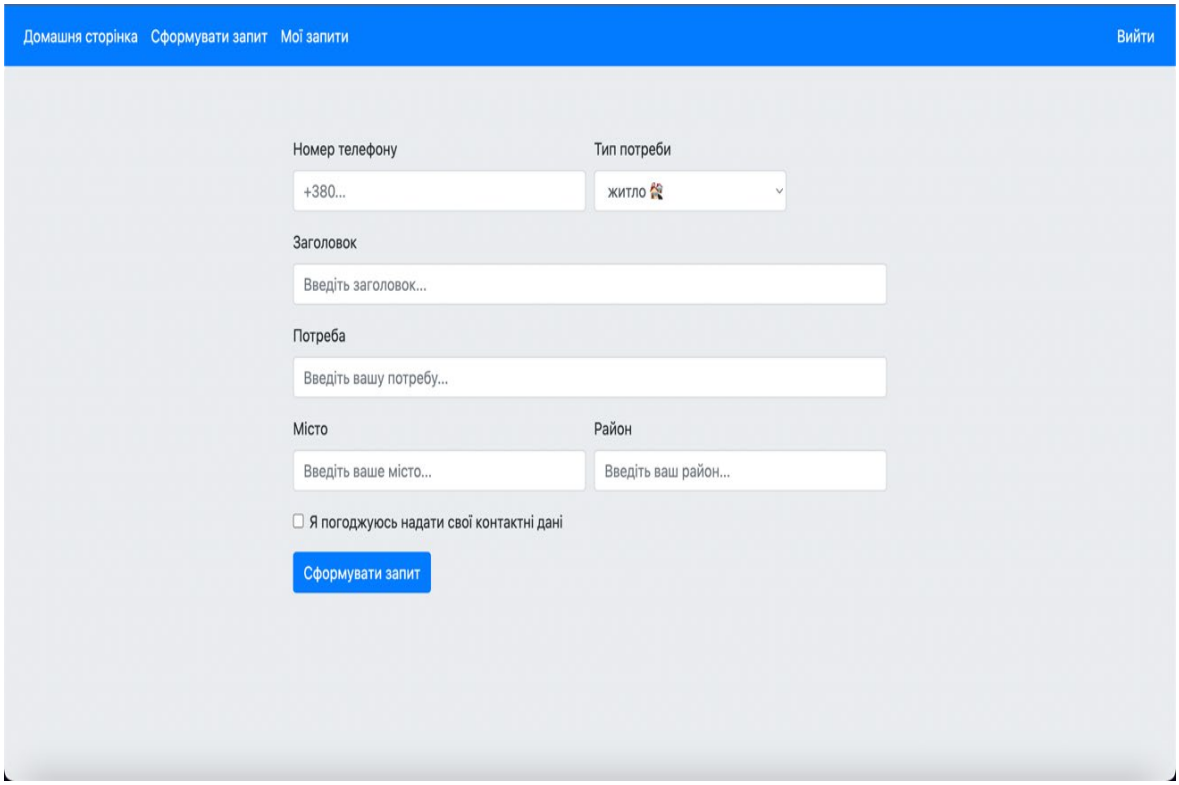

рис.8 Сторінка користувача для формування нового запита

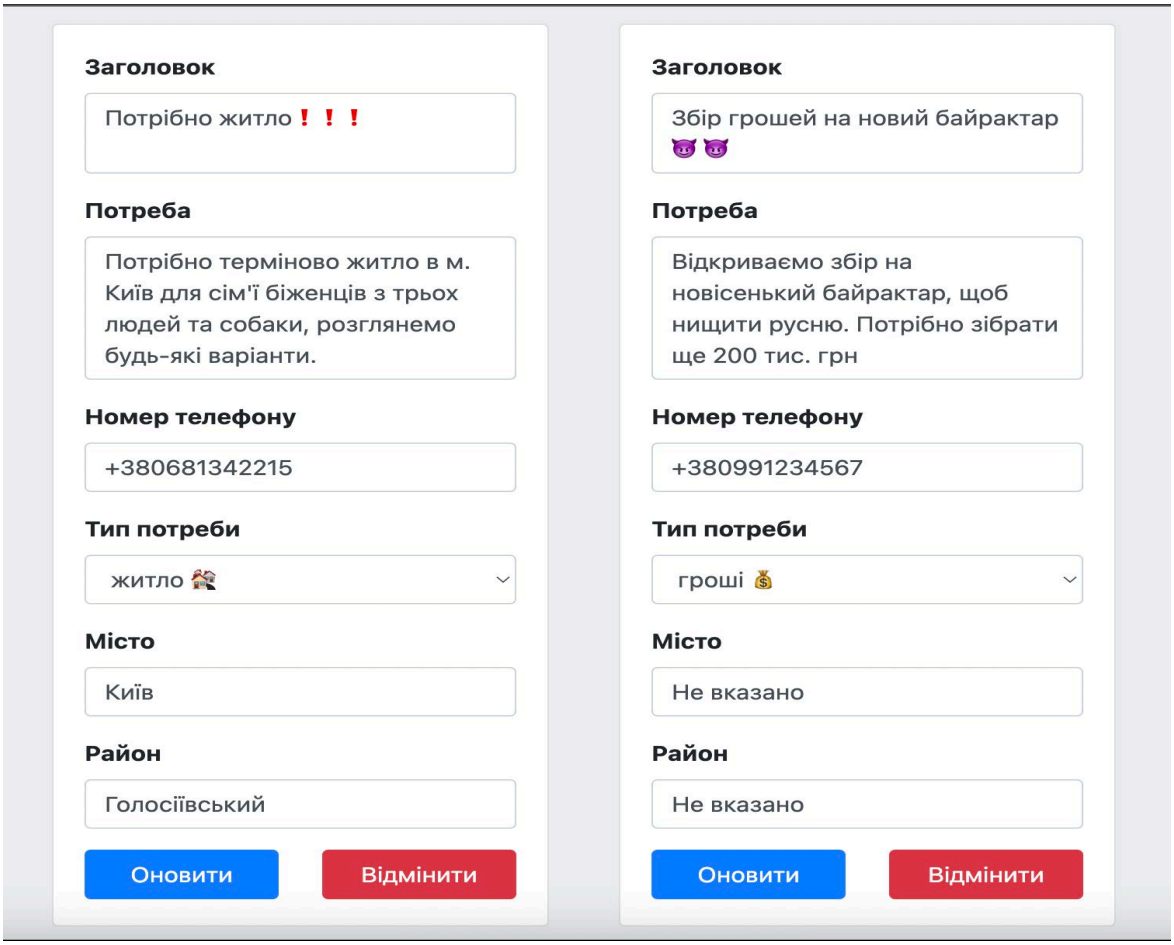

Рис.9 Сторінка користувача для перегляду своїх запитів

ВИСНОВКИ

У кваліфікаційній роботі було розглянуто існуючі підходи для побудови мікросервісної архітектури програмних застосунків, проаналізовано основні переваги та недоліки мікросервісів, досліджено системи масового обслуговування та можливі варіанти їх практичного використання.

В результаті роботи було розроблено веб-орієнтовну систему, з використанням мікросервісної взаємодії. Було обрано та аргументовано вибір архітектури та технологій для реалізації системи, розглянуто сучасні патерни проектування, які було використано під час розробки.

Застосунок представляє собою чотири повноцінних та незалежних сервіси клієнтської та серверної частини. У системі впроваджено можливість: авторизації, реєстрації, формувати та редагувати запити, а також брати їх на опрацювання . У роботі покроково описано, яким чином розроблялось програмне рішення та з яких частин воно складається.

У майбутньому планується покращувати застосунок, а саме: інтеграція з Kubernetes для автоматичного масштабування та оркестрації мікросервісів, автоматичне розподілення запитів серед волонтерів шляхом вибору найбільш релевантних для конкретного волонтера по заданим критеріям, розгортання в клауді та інтеграція з хмарними сервісами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Перша згадка про мікросервіси [Електронний ресурс]. – Режим доступу до ресурсу: https://www.sumerge.com/what-is-microservices-architecture
2. AMQP [Електронний ресурс]. – Режим доступу до ресурсу: https://www.cloudamqp.com/blog/what-is-amqp-and-why-is-it-used-in-rabbitmq.html
3. DevOps [Електронний ресурс]. – Режим доступу до ресурсу: https://tproger.ru/curriculum/devops
4. Джефф Сазерленд. Scrum. Навчись робити вдвічі більше за менший час / Джефф Сазерленд – К.: «Клуб Сімейного Дозвілля», 2022. – 25 с.
5. Головні тренди хмарних обчислень за 2020 рік [Електронний ресурс]. – Режим доступу до ресурсу: https://denovo.ua/blog/metamorfozi-hmar-golovni-trendi-2020-73
6. Литвинов А.Л. Теорія систем масового обслуговування: навч. посібник / А.Л. Литвинов; Харків. нац. ун-т міськ. госп-ва ім. О.М. Бекетова. – Харків: ХНУМГ ім. О.М. Бекетова, 2018. – 141 с.
7. DI (Dependency Injection) [Електронний ресурс]. – Режим доступу до ресурсу: https://www.martinfowler.com/articles/injection.html
8. Express [Електронний ресурс]. – Режим доступу до ресурсу: https://www.npmjs.com/package/express
9. Avro [Електронний ресурс]. – Режим доступу до ресурсу: https://avro.apache.org
10. Kalen Delaney. Microsoft SQL Server Internals. / Kalen Delaney, Craig Freedman. – К.: «Microsoft Press», 2013. – 110 с.
11. CRUD [Електронний ресурс]. – Режим доступу до ресурсу: https://stackify.com/what-are-crud-operations
12. Кріс Річардсон. Мікросервіси. Патерни розробки і рефакторінга / Кріс Річардсон. – К.: «Пітер», 2019. – 259 с.

13. Gregor Hohpe. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / Gregor Hohpe, Bobby Woolf. – К.: «Addison-Wesley Professional», 2011. – с. 145
14. Event Sourcing Pattern [Електронний ресурс]. - Режим доступу до ресурсу: https://microservices.io/patterns/data/event-sourcing.html
15. Consistency [Електронний ресурс]. — Режим доступу до ресурсу: https://www.scylladb.com/glossary/database-consistency