МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет радіоелектроніки
Факультет Комп'ютерних наук
Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

другий (магістерський)

(рівень вищої освіти)

Дослідження методів створення сервісно-орієнтованих програмних систем у Azure

Виконав:
студент 2 курсу групи ІПЗм-21-1
Макєєв О.С.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення

Тип програми Освітньо-наукова

Керівник доц. Кравець Н.С.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. Кафедри _____ З.В. Дудар

2023 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121– Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри ____

(підпис)

«___»_____202_ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента Макєєва Олексія Сергійовича
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методів створення сервісно-орієнтованих програмних систем у Azure»

затверджена наказом університету від «29» 03.2023р. № 302Ст

2. Термін подання студентом роботи до екзаменаційної комісії «__» _____202__р.

3. Вихідні дані до роботи *вимоги до сервісно-орієнтованої програмної системи, атрибути якості програмної системи, хмарна платформа Azure, мова розробки Java, пояснювальна записка.*

4. Перелік питань, що потрібно опрацювати в роботі *мета роботи, аналіз предметної галузі і постановка задачі, аналіз вимог до програмного забезпечення, архітектура та проектування сервісно-орієнтованої системи, порівняння Azure сервісів.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної області дослідження	3 квітня 2023	Виконано
2	Аналіз вимог до програмної системи	4 квітня 2023	Виконано
3	Формування вимог ПЗ	6 квітня 2023	Виконано
4	Аналіз сервісів від Microsoft Azure	10 квітня 2023	Виконано
5	Проектування та розробка ПЗ	11 квітня 2023	Виконано
6	Написання пояснювальної записки	24 квітня 2023	Виконано
7	Перевірка записки керівником та норм контролером	3 травня 2023	Виконано
8	Оцінка роботи стороннім рецензентом, отримання відгуку від керівника роботи, попередній захист	4 травня 2023	Виконано
9	Занесення диплома в електронний архів	10 травня 2023	Виконано
10	Здача готової роботи	12 травня 2023	Виконано

Дата видачі завдання 29 березня 2023 р.

Студент _____ Макєєв О.С.

(підпис) (прізвище, ініціали)

Керівник роботи _____ доц. Кравець Н.С.

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до магістерської кваліфікаційної роботи: 95 с., 6 табл., 65 рис., 29 джерел, 4 додатки.

ВЕБ-СЕРВІСИ, ПРОГРАМИ-КОНТЕЙНЕРИ AZURE, СЕРВІС KUBERNETES, СЕРВІСНО-ОРІЄНТОВАНА ПРОГРАМНА СИСТЕМА, ХМАРИ AZURE, ХМАРНІ ТЕХНОЛОГІЇ, AZURE RED HAT OPENSHIFT, DOCKER, JAVA, REACT, SPRING, TYPESCRIPT

Метою дослідження є порівняльний аналіз методів створення сервісно-орієнтованих програмних систем на базі сервісів Azure, а предметом дослідження – програмне рішення, яке реалізоване за допомогою цих методів.

Об’єктом дослідження є процес розгортання створеної програмної системи на хмарній платформі Azure.

Методи розробки базуються на інструментах для створення веб-застосунку на мовах програмування Java, TypeScript використовуючи фреймворк Spring Framework та бібліотеку React.

В результаті магістерської кваліфікаційної роботи було досліджено методи створення сервісно-орієнтованих програмних систем, проаналізовані сервіси від Azure, які надають можливість розгортати сервіси програмної системи.

WEB SERVICES, AZURE CONTAINERS, KUBERNETES SERVICE, SERVICE-ORIENTED SOFTWARE SYSTEM, AZURE CLOUDS, CLOUD TECHNOLOGIES, AZURE RED HAT OPENSHIFT, DOCKER, JAVA, REACT, SPRING, TYPESCRIPT

The purpose of the study is a comparative analysis of the methods of creating service-oriented software systems based on Azure services, and the subject of the study is a software solution implemented using these methods.

The object of research is the process of deploying the created software system on the Azure cloud platform.

The development methods are based on tools for creating a web application in the programming languages Java, TypeScript using the Spring Framework and the React

library.

As a result of the master's qualification work, the methods of creating service-oriented software systems were investigated, the services from Azure were analyzed, which provide the opportunity to deploy the services of the software system

Умови публікації пояснювальної записки

Я,

Макєєв Олексій Сергійович

(прізвище, ім’я, по батькові)

студент(ка) групи ІПЗм-21-1здобувач вищої освіти на другому (магістерському)рівні

кафедра програмної інженерії,

(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему

Дослідження методів створення сервісно-орієнтованих програмним систем у Azure,

(назва роботи)

що буде представлена до ЕК для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

Велика кількість, на теперішній час, сервісно-орієнтованих програмних систем являє собою масштабні корпоративні системи, які мають проблеми з масштабуванням, підтримкою старого та додаванням нового функціоналу. Бізнесу, з кожним роком, доводиться витрачати все більше грошей, щоб підтримувати функціонал та шукати підходящих спеціалістів, які працюють або працювали зі старими технологіями. Кожен раз коли додається новий функціонал, це випробування для компанії та команди, щоб вкластися в певні рамки по часу реалізації та випуску даного функціоналу на ринок.

Сучасна розробка сервісно-орієнтованих програмних систем[14] супроводжується широким використанням хмарних технологій, які впливають на конкурентоспроможність компаній та їх продукту. Використання хмарних технологій надає можливість компаніям розширити клієнтські бази, через те що зменшиться затримка при використанні застосунку, буде менше випадкових помилок, які пов’язані з проблемою пам’яті, перенавантаженню системи та інше.

Перевагою хмарних сервісів є доступність в будь-який точці світу та дозволить бізнесу швидко масштабувати свої проекти та зменшить витрати на інфраструктуру та обслуговування.

Хмарні провайдери надають великий обсяг сервісів[11] для різних потреб: таких як хостинг, розгортання контейнерів, файлове сховище, бази даних тощо. Зокрема всі найпопулярніші хмарні провайдери пропонують кілька варіантів створення сервісно-орієнтованих програмних систем, включаючи як стандартні технології так і власні розробки.

Тому тема дослідницької магістерської кваліфікаційної роботи є актуальна в сучасному світі інформаційних технологій та допоможе бізнесу та науковій діяльності обрати потрібний метод по створенню сервісно-орієнтованих програмних систем та на основі експерименту обрати потрібну платформу від Azure[12] для своєї подальшої діяльності.

Магістерська кваліфікаційна робота повинна виконати такі завдання:

а) аналіз існуючих методів створення сервісно-орієнтованої програмної системи;

б) аналіз предметної галузі;

в) проектування програмної архітектури;

г) реалізація програмної архітектури;

д) аналіз вимог до програмної системи;

и) зробити експеримент на сервісах від Azure над реалізованим програмним застосунком.

Метою магістерської кваліфікаційної роботи є порівняльний аналіз методів створення сервісно-орієнтованих програмних систем на базі сервісів Azure.

Предметом дослідження є програмне рішення, яке реалізоване за допомогою цих методів.

Об’єктом дослідження є процес розгортання створеної програмної системи на хмарній платформі Azure.

Під час виконання кваліфікаційної роботи були використані теоритичні, емпірині та загально логічні методи дослідження.

Використовуючи теоретичні методи було проведено аналіз та проектування програмної системи та був застосований метод сходження від абстрактного до конкретного.

Використовуючи емпіричний метод було проведено експеримент над зробленою програмною системою, щоб дослідити сервіси від Azure за певними критеріями та отримані результати після експерименту.

Використовуючи загально логічні методи було проаналізовано предметну галузь, сервісно-орієнтовану архітектуру, аналоги по предметній області, існуючі методи та сервіси від Azure.

У даній магістерський роботі порівняні методи створення сервісно-орієнтованих програмних системи, а саме мікросервіси, RESTful веб-сервіси та сервіси на основі GraphQL.

Зроблений аналіз існуючих хмарних провайдерів від Microsoft, Amazon та Google та проаналізовано сучасні сервіси від Azure, які надають можливість розгортати проекти, а саме Azure Container Apps[2], Azure Kubernete Service[1] та Azure Red Hat OpenShift[3].

Розглянуто предметну область технологій реалізації сервісно-орієнтованої архітектури застосунків, запропоновано критерії для аналізу методів реалізації застосунків із такою архітектурою.

Спроектовано та розроблено програмне рішення для порівняння методів створення сервісно-орієнтованих застосунків на базі хмарної платформи Azure. Було спроектовано клієнтську та серверну частину. Предметна область програмної системи пов’язана з орендою транспорту, а саме самокатів, велосипедів та автомобілів.

Після виконання експериментальної частини, отримані результати можуть використатись у бізнесі при розробці нових продуктів та поступова міграція існуючих сервісів на хмарного провайдера Azure.

У вигляд тез було опубліковано магістерську кваліфікаційну роботу на Міжнародному молодіжному форумі «РАДІОЕЛЕКТРОНІКА І МОЛОДЬ В XXI СТОЛІТТІ» та подані результати до Міжнародного наукового журналу «Журнал комп’ютерних систем та інформаційних технологій».

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Загальні відомості

Для дослідження буде використовуватись сервісно-орієнтована архітектура[13] для програмного забезпечення для оренди транспорту, а саме самокатів, велосипедів та автомобілів.

У джерелах 26, 27 наводиться приклад використання та моделювання сервісно-орієтованої архітектури, що дає змогу проаналізувати сучасний стан та розвиток сервісно-орієнтованої архітектури.

Для розгортання сервісу будуть використовуватися хмарні технології, а саме публічна хмара Azure.

Хмарні технології це технології використання серверних ресурсів з одночасним запуском великої кількості віртуальних серверів незалежно один від одного. Перебої в роботі одного сервера не стосуватиметься інших серверів, що забезпечує загальну безперебійну роботу. Крім цього, хмарні технології дозволяють рівномірно регулювати та оплачувати лише використовувані ресурси, без переплат.

Публічна хмара надає інфраструктуру, яка призначена для вільного користування широкою публікою. Може перебувати у власності, управлінні та експлуатації комерційних, наукових та урядових організацій (або будь-якої їх комбінації). Фізично існує у юрисдикції власника – постачальника послуг.

Для дослідження було обрано хмарну платформу Microsoft Azure та її сервіси Azure Container Apps, Azure Kubernetes Service та Azure Red Hat OpenShift, які будуть використовуватись для порівняння.

Microsoft Azure – хмарна платформа, яка поєднує в собі як вирішення обчислювальної інфраструктури IaaS (сервери, сховища даних, мережі, операційні системи), так і набір інструментів і сервісів, що полегшують розробку і розгортання хмарних додатків (PaaS).

1.2 Аналіз існуючих аналогів по оренді транспорту

На даний час в Україні починається розвиток сервісів по оренді транспорту, а саме самокатів, велосипедів та автомобілів.

1.2.1 Аналог «BikeNow»

Було розглянуто аналог «BikeNow», як можна побачити на рисунку 1.1. Даний сервіс користується популярністю серед місцевого населення, яке хоче орендувати велосипед у місті Київ.

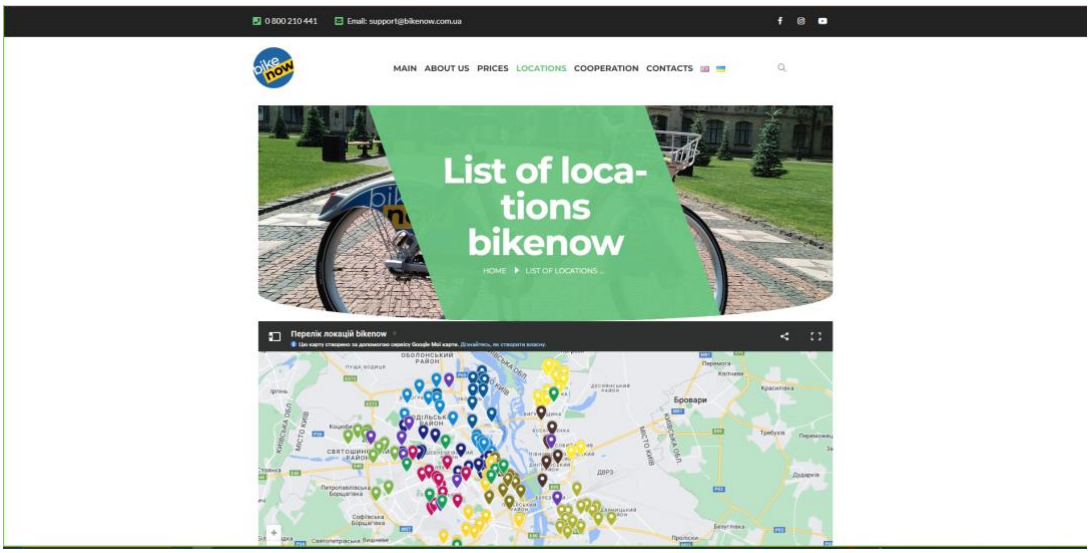

Рисунок 1.1 – Сервіс «BikeNow»[Рисунок виконаний самостійно*]

Даний сервіс дозволяє орендувати велосипеди та самокати в місті Києві. На карті веб-сайту даного сервісу відображено місця де знаходиться вільний транспорт. Також у даного сервісу є мобільний застосунок, який покриває недоліки веб-сайту, а саме користувацький профіль, гаманець, платіжні картки, абонементи та відображення найближчого транспорту від вашого місця розташування.

Недоліком даної програмної системи є в тому, що даний сервіс сфокусований на одному місті та не надає можливість масштабуватися в межах країни.

1.2.2 Аналог «Nars Cars»

Було розглянуто аналог «Nars Cars», як показано на рисунку 1.2, який надає можливість орендувати машини в таких містах, як Київ, Харків, Львів, Одеса, Дніпро, Полтава та інші міста України.

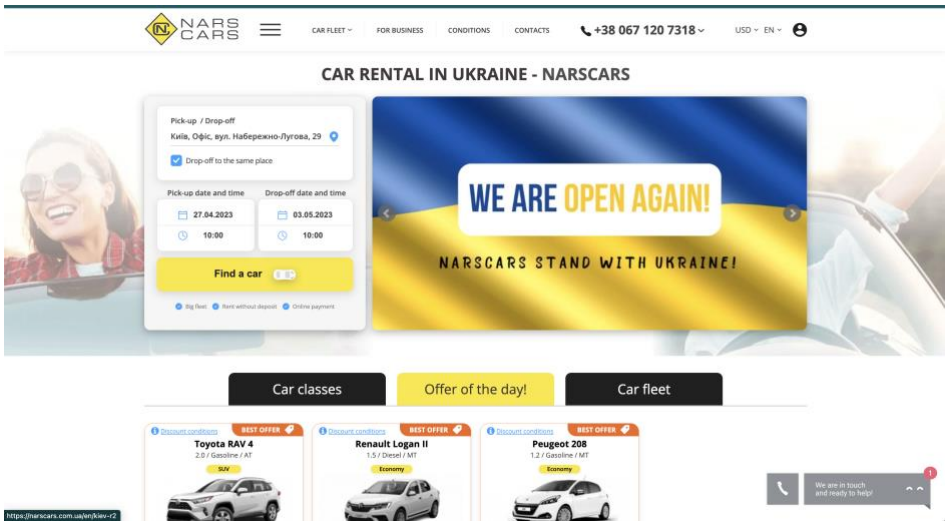

Рисунок 1.2 – Сервіс «Nars Cars» [Рисунок виконаний самостійно*]

Даний аналог надає можливість знайти вільні машини на потрібні дати та розраховує вартість транспорту. Має великий спектр автомобілів від економ до преміум.

Даний сервіс не має мобільного додатку де відображались би в зручному вигляді інформація про автомобілі та надавала змогу шукати вільний транспорт в межах свого міста чи області.

1.2.3 Аналог «Bolt електросамокати»

Було розглянуто аналог «Bolt електросамокати», веб-сайт застосунок показано на рисунку 1.3. Веб-застосунок лише містить інформацію про сервіс.

Даний сервіс надає можливість орендувати електросамокати в багатьох містах України використовуючи мобільний додаток.

Мобільний додаток надає можливість сканувати QR код та після чого орендувати електросамокат.

Також надає можливість знайти вільний самокат в додатку Bolt, використовуючи внутрішню карту міста.

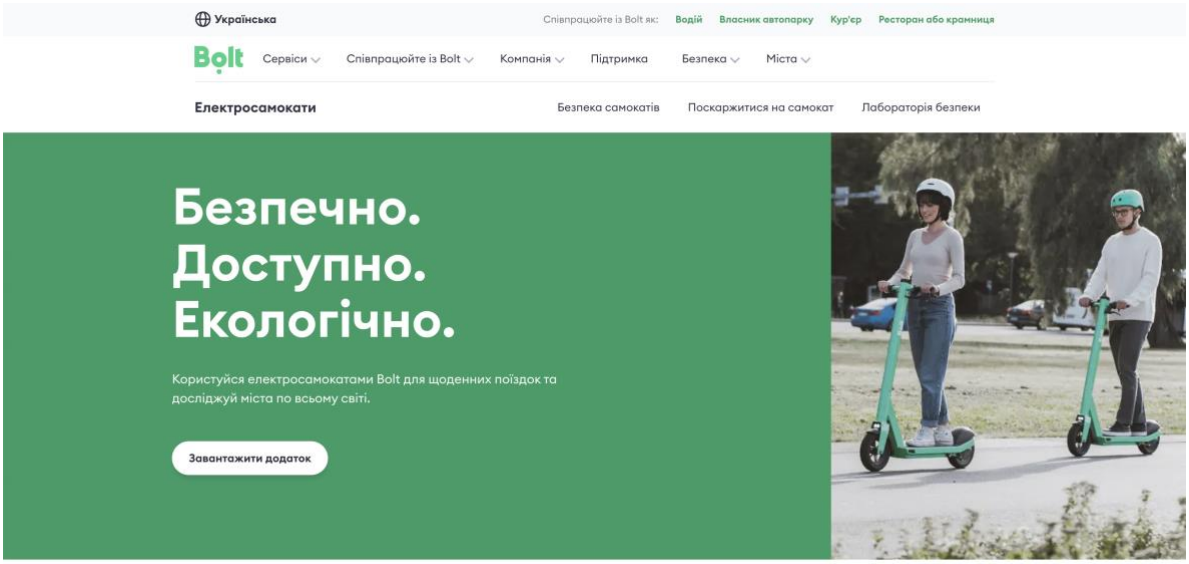

Рисунок 1.3 – Сервіс «Bolt електросамокати» [Рисунок виконаний самостійно*]

Щоб оплатити оренду самоката можна використовувати Google Pay, Apple Pay або кредитну картку.

1.3 Аналіз сервісно-орієнтованої архітектури

Дана архітектура – це архітектурний шаблон програмного забезпечення, який використовує модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабо пов’язаних замінних компонентів оснащених стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами.

Сервісно-орієнтована архітектура [7] не прив’язана до певної технології. Вона може бути реалізована з використанням широкого спектру технологій, включаючи такі технології, як REST, RPC, DCOM, CORBA або веб-сервіси.

Дана архітектура може включати такі елементи, як можна побачити на рисунку 1.4.

Дані елементи можуть бути слабопов’язані сервіси, що взаємодіють через строго певний інтерфейс між собою.

Сервіс містить окрему функціональну одиницю, доступну лише через формально визначений інтерфейс. Послуги, які надає сервіс, можуть бути свого роду «нанопідприємствами», які легко можуть реалізовуватись та вдосконалюватись. Також служби можуть бути «мегакорпораціями», побудованими як злагоджена робота підлеглих служб.

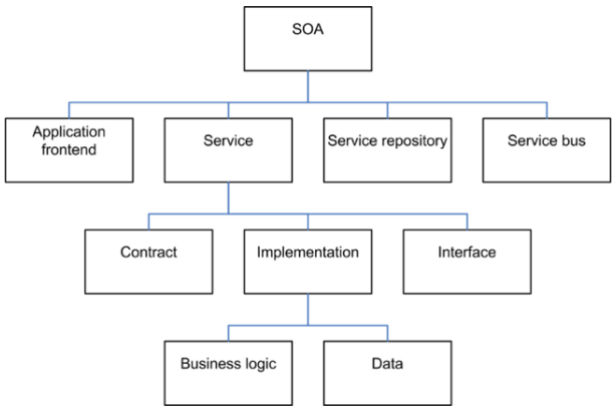

Рисунок 1.4 – Компоненти сервісно-орієнтованої архітектури[17]

1.4 Аналіз можливостей трьох найпопулярніших хмарних платформ по реалізації сервісно-орієнтованих систем

На сьогоднящній день окрім хмарної платформи Microsoft Azure, існує дві популярні хмарні платформи: Amazon Web Services(AWS) та Google Cloud Platform.

Дані платформи мають майже такий самий функціонал, але у всіх платформ різна інфраструктура до якої треба звикати.

В Amazon Web Services [9] центральної обчислювальної службою є сервіс Elastic Compute Cloud (EC2), як зображено на рисунку 1.5.

EC2 став головним синонімом для поняття «масштабовані обчислення на вимогу». Для того, щоб ще ретельніше планувати і знижувати витрати при запуску

проектів, компанія ввела нові подсервіси, такі як AWS Elastic Beanstalk, Amazon EC2 Container Service.

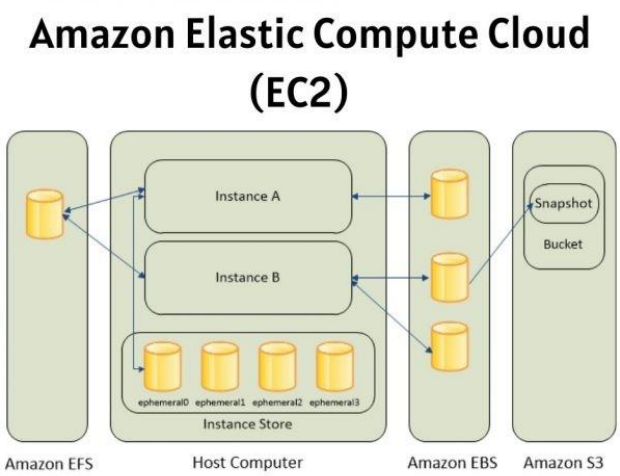

Рисунок 1.5 – Структура Elastic Compute Cloud[18]

Основа обчислювальних систем Microsoft Azure – це класичні віртуальні машини і високопродуктивні Virtual Machine Scale Sets, як зображено на рисунку 1.6. Клієнтські програми для Windows можуть бути розгорнуті за допомогою сервісу RemoteApp. Azure Virtual Machine включає 4 різних сімейства, 33 типу примірників, які ви можете розгорнути в різних регіонах. Але підтримка певної зони регіону поки не підтримується.

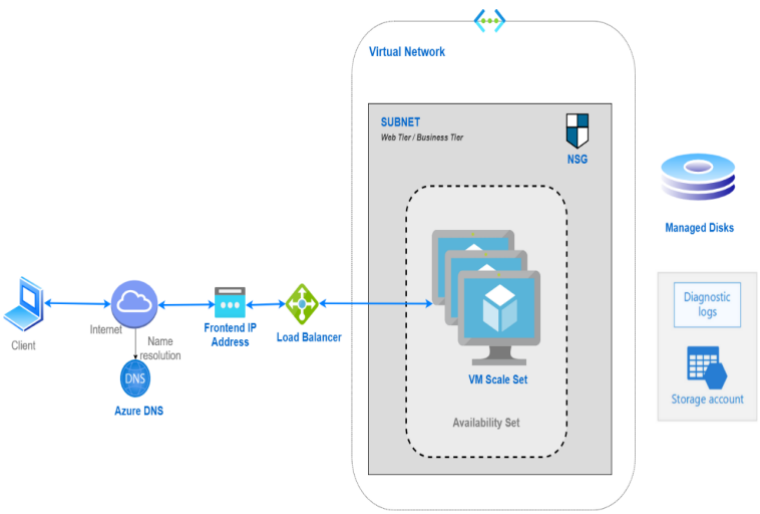

Рисунок 1.6 – Структура Virtual Machine Scale Sets[19]

Google Cloud Platform [8] використовує сервіс Compute Engine для обробки обчислювальних процесів. Одним з головних недоліків є ціноутворення, воно менш гнучке в порівнянні з AWS і Azure.

Compute Engine підтримує більшість основних хмарних послуг – розгортання контейнера, масштабованість і обробка даних.

Google Cloud підтримує 4 сімейств примірників, 18 різних типів примірників, а також забезпечує як регіональне розміщення, так і вибір зони.

На рисунку 1.7 зображено структуру Compute Engine.

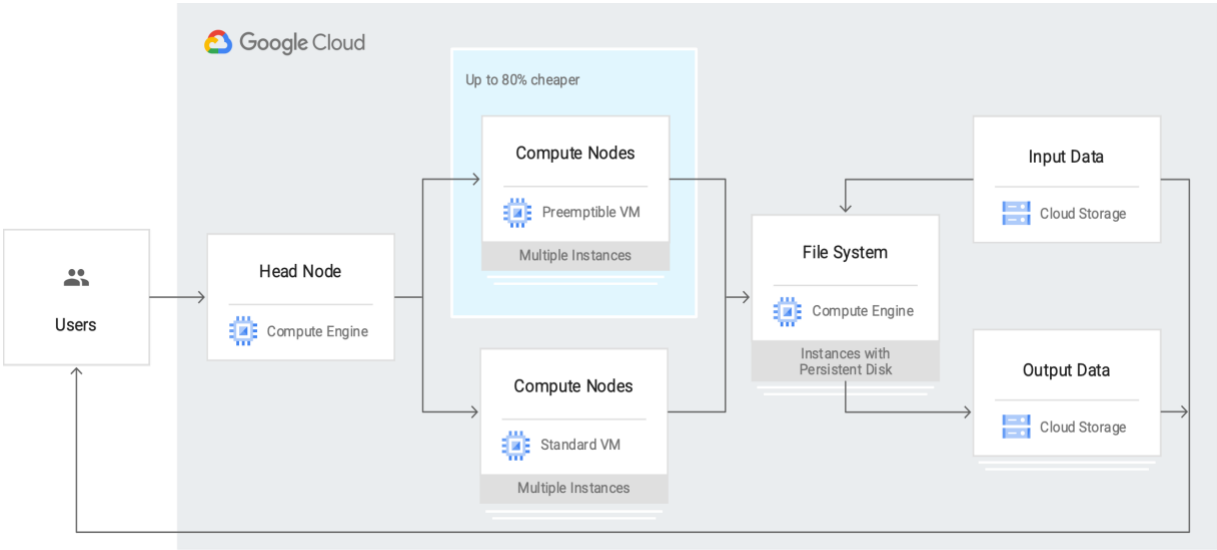

Рисунок 1.7 – Структура Compute Engine[20]

Якщо вибирати лідера, то AWS і Microsoft Azure зараз найбільш затребувані хмарні платформи. Пропоновані обчислювальні потужності у компаній перебувають практично на рівних рівнях.

2 ПОСТАНОВКА ЗАДАЧІ

В магістерській кваліфікаційній роботі потрібно спроектувати архітектуру та розробити прототип програмної системи для оренди транспорту. Дослідити методи створення сервісно-орієнтованої програмної системи використовуючи хмарного провайдера Azure та його сервіси, а саме:

а) Azure Container Apps;

б) Azure Kubernetes Service;

в) Azure Red Hat OpenShift.

Дослідити доцільність їх використання для даної програмної системи використовуючи наступні критерії для порівняння:

а) вартість і ціноутворення;

б) особливості та функціональність;

в) швидкість розгортання;

г) підтримка реєстрів контейнерів;

д) підтримка моніторингу та журналювання.

Використовувати Docker, щоб система працювала на всіх сервісах очікувано по функціоналу та щоб помістити модулі програмної системи до окремих Docker образів, які будуть розгортатись у Docker контейнерах, як зображено на рисунку 2.1.

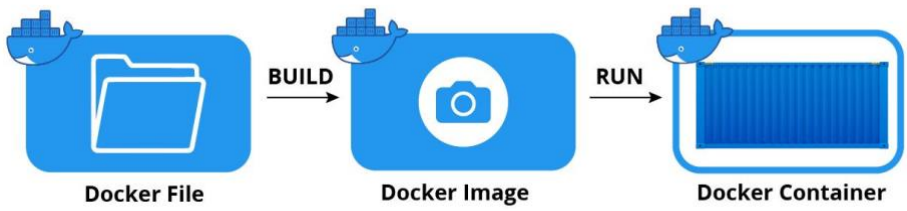

Рисунок 2.1 – Структура Docker процесу[21]

Для розгортання програмної системи у сервісах потрібно використовувати підхід CI/CD [6], як показано на рисунку 2.2 для того, щоб автоматизувати процеси розгортування, тестування, зборки проекту.

Для автоматизації буде використовуватись GitHub Actions, який дозволяє створити Action Pipeline.

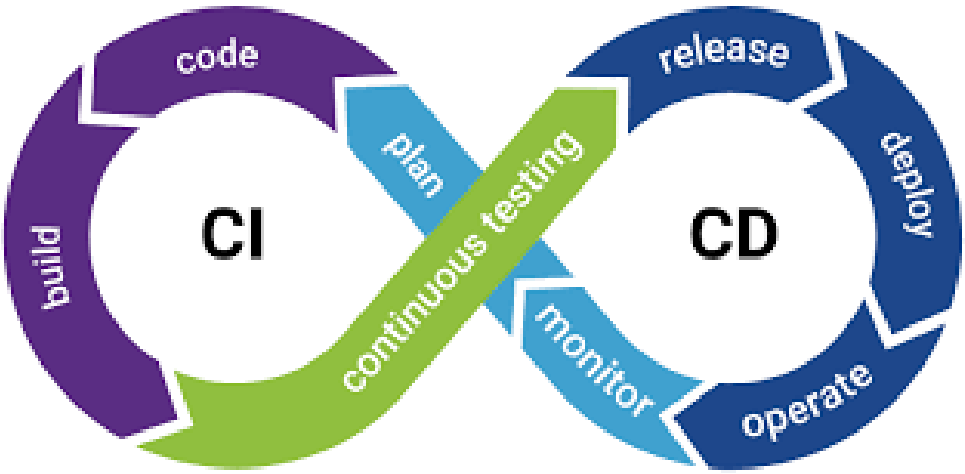

Рисунок 2.2 – CI/CD процес[22]

Використовувати GitHub для зберігання кодової бази в окремих репозиторіях для кожної частини.

3 АНАЛІЗ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Задачею даної магістерської кваліфікаційної роботи є проектування сервісно-орієнтованої програмної системи, яка буде надавати можливість орендувати транспорт та буде використовуватись для порівняння Azure сервісів.

Для проведення запланованих досліджень Azure сервісів необхідно побудувати прототип програмної системи використовуючи шаблони проектування, технології та певні архітектурні стилі.

Проектування системи починається з аналізу та формування функціональних та нефункціональних вимог до програмної системи.

3.1 Функціональні вимоги

Програмна система по оренді транспорту має відповідати вимогам та забезпечувати набір функцій. Прототип даної системи повинен включати наступний функціонал для проведення досліджень:

а) можливість зареєструвати нового користувача;

б) автентифікація та авторизація користувачів;

в) перегляд транспортів;

г) оренда транспорту;

д) подовження оренди;

е) фільтрування та пошук транспорту;

ж) поповнення віртуального балансу;

и) переключення міст для оренди транспорту;

і) наявність гнучкої системи груп користувачів;

к) модерація даних та прав доступу користувачів.

3.2 Нефункціональні вимоги

Програмна система повинна мати наступні нефункціональні вимоги:

а) система повинна підтримувати REST API для спілкування зі сторонніми сервісами;

б) модулі системи повинні знаходитись у Docker образах;

в) система повинна мати базу даних для зберігання інформації про транспорт та користувачів даної системи;

г) система повинна бути захищена від різних небезпек для веб-застосунків:

1) міжсайтове виконання сценаріїв/скриптів (XSS);

2) витік конфіденційних даних;

3) некоректна автентифікація;

4) небезпечна десеріалізація;

5) управління сеансами.

д) система повинна бути відказостійкою;

е) система повинна бути масштабуєма;

ж) система повинна включати українську та англійську локалізацію;

и) система не повинна призводити до втрати даних користувачів;

к) система повинна бути розрахована на 1 млн користувачів в місяць в межах однієї країни;

л) відклик системи під нагрузкою повинно бути до 10 секунд.

3.3 Вимоги до клієнтського застосунку

Для того щоб відобразити інформацію про транспорт, дані користувача потрібно мати користувацький інтерфейс.

Даний інтерфейс повинен містити наступний функціонал:

а) можливість зареєструватися;

б) можливість авторизуватися у системі;

в) можливість переглянути профіль користувача;

г) можливість перегляду та поповнення балансу;

д) можливість обрати транспорт для оренди;

ж) перегляд карти з неорендованим транспортом.

4 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ СТВОРЕННЯ СЕРВІСНО-ОРІЄНТОВАНОЇ ПРОГРАМНОЇ СИСТЕМИ

На теперішній час є декілька сучасних технологій для реалізації сервісно-орієнтованої архітктури:

а) мікросервісна архітекрура;

б) RESTful веб-сервіси;

в) GraphQL.

4.1 Мікросервісна архітектура

Мікросервісна архітектура – це конкретна реалізація сервіс-орієнтованої архітектури (SOA), яка зосереджена на розбитті великої монолітної системи на менші незалежні служби, які можна розробляти, розгортати та масштабувати незалежно, як показано на рисунку 4.1. Мікросервіси створені для слабкого зв’язку та зв’язку один з одним через API.

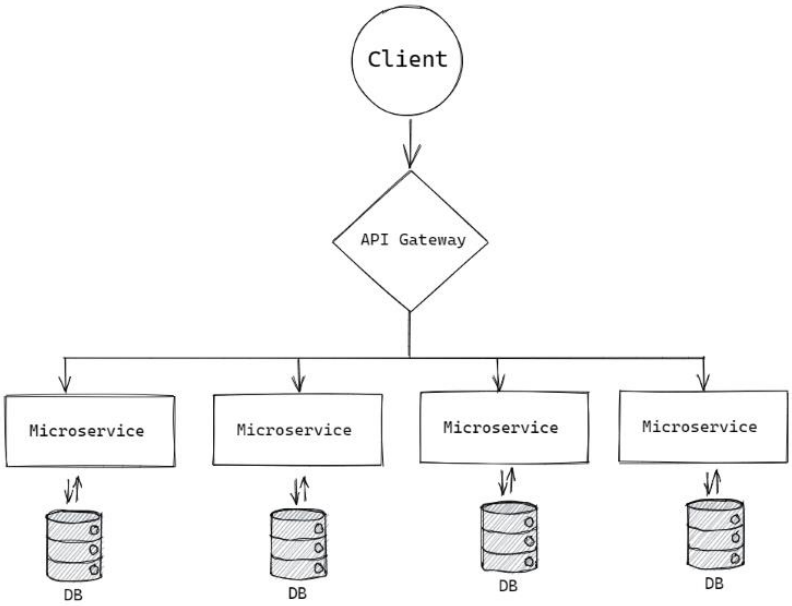

Рисунок 4.1 – Мікросервісна архітектура[27]

У сервісно-орієнтованій архітектурі мікросервіси можуть надавати низку переваг, зокрема:

а) гнучкість. Мікросервіси дозволяють швидше та частіше розгортати, оскільки окремі сервіси можна розробляти та розгортати незалежно один від одного;

б) масштабованість. Оскільки мікросервіси можна масштабувати незалежно від інших сервісів, що дає можливість справлятися зі змінами попиту на певні послуги, не впливаючи на решту системи;

в) відмовостійкість. Завдяки ізоляції кожної служби легше справлятися зі збоями в одній службі, не впливаючи на загальну системи.

Реалізація мікросервісної архітектури в сервісно-орієнтованій архітектурі також може мати свої недоліки, а саме:

а) складність. З більшою кількістю послуг стає проблематично керувати зв’язки між послугами, моніторингом і тестуванням;

б) узгодженість даних. Підтримання узгодженості даних в кількох службах може бути проблематичним завданням, оскільки кожна служба може мати своє сховище даних;

в) виявлення нових сервісів. В архітектурі мікросервісів інші сервіси повинні мати доступ до сервісів, керувати якими може бути складно.

4.2 RESTful веб-сервіси

Веб-сервіси RESTful відповідають цим обмеженням і розроблені як прості, легкі та масштабовані.

Representational State Transfer (REST) – це архітектурний стиль, який визначає набір обмежень для створення веб-служб.

Веб-сервіси RESTful використовують такі методи HTTP, як GET, POST, PUT і DELETE, щоб виконувати операції CRUD (створення, читання, оновлення, видалення) над ресурсами, як показано на рисунку 4.2.

У сервісно-орієнтованій архітектурі веб-сервіси RESTful можна використовувати для полегшення зв’язку між сервісами, забезпечуючи стандартний спосіб взаємодії сервісів один з одним.

Кожна служба може надавати RESTful API, який інші служби можуть використовувати для доступу до її функцій.

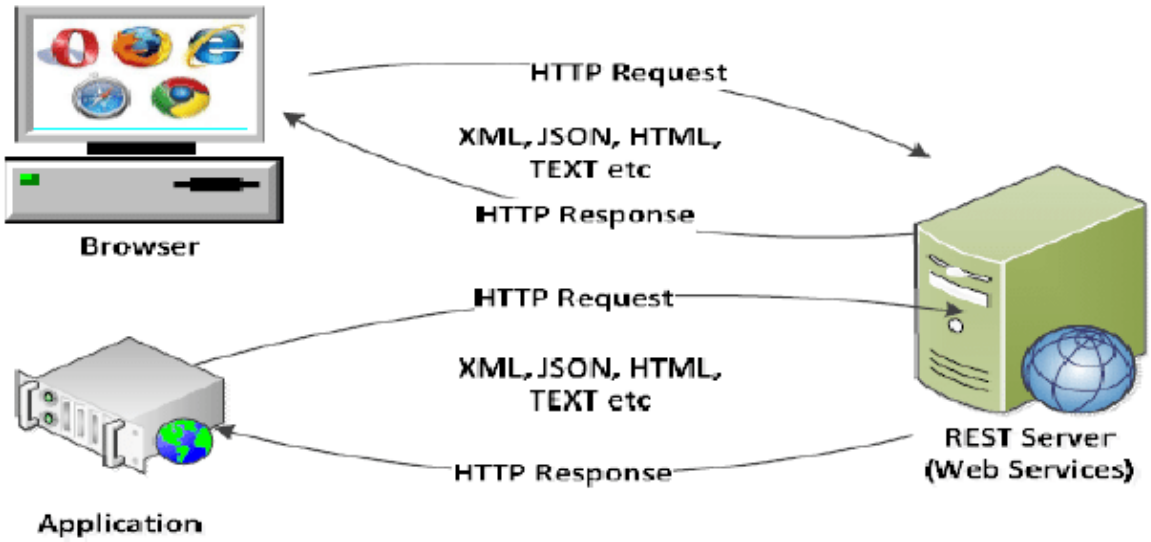

Рисунок 4.2 – RESTful веб-сервіси[24]

Переваги використання веб-служб RESTful у сервіс-орієнтованій архітектурі включають:

а) масштабованість. Веб-сервіси RESTful створені для масштабування, що дозволяє легко впоратися зі змінами попиту на певні послуги;

б) сумісність. Веб-сервіси RESTful використовують стандартні методи HTTP, доступ до них може отримати широкий спектр клієнтів і платформ;

в) простота. Веб-сервіси RESTful прості для розуміння та можуть бути реалізовані за допомогою простих легких фреймворків.

Реалізація RESTful веб-сервісів у сервісно-орієнтованій архітектурі також може мати свої недоліки, а саме:

а) нестандартизований API. Веб-сервіси RESTful можуть писатись без API документації, що може призвести до проблем, якщо щось зміниться в API, а кінцевий користувач про це не буде знати;

б) безпека. Веб-сервіси RESTful мають бути захищені від несанкціонованого доступу та атак, таких як впровадження SQL;

в) Проблема з продуктивністю. Якщо неправильно спроектувати базову архітектуру для RESTful веб-сервісів, то через це можуть бути проблеми продуктивністю.

4.3 GraphQL

GraphQL – це мова запитів і середовище виконання для API, розроблена Facebook у 2015 році.

GraphQL надає клієнтам більш гнучкий та ефективний спосіб, щоб запитувати дані від серверів, дозволяючи клієнтам точно вказувати, які дані їм потрібні, та зменшуючи кількість звернень до серверів, як зображено на рисунку 4.3.

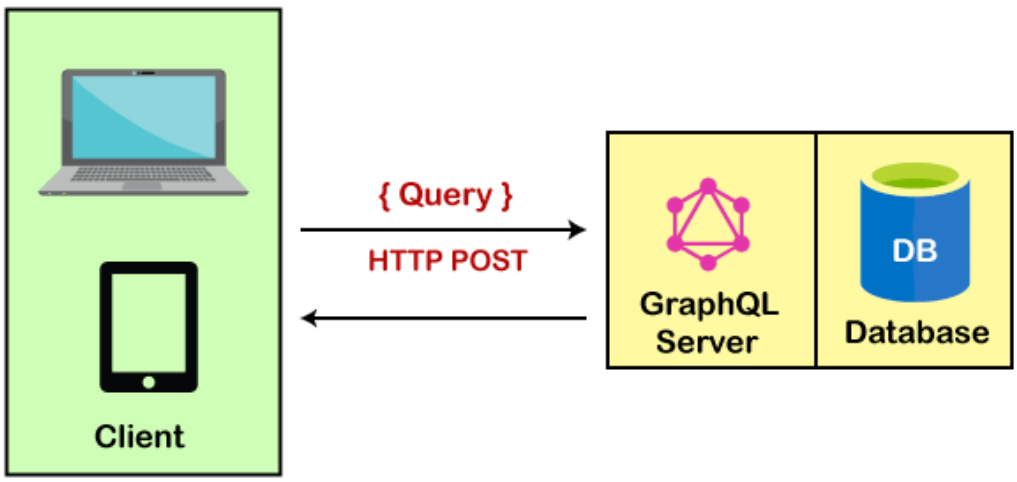

Рисунок 4.3 – GraphQL архітектура[23]

Реалізація сервісів за допомогою GraphQL для сервісно-орієнтованої архітектури можуть мати такі переваги, а саме:

а) гнучкий API. GraphQL можна використовувати для полегшення зв’язку між сервісами, надаючи єдину кінцеву точку API, яка агрегує дані з кількох сервісів;

б) роз’єднаність сервісів. Використання GraphQL у сервісно-орієнтованій архітектурі дозволяє більше роз’єднати сервіси. Оскільки кожен сервіс може

надавати власну схему GraphQL та резолвери. Сервіси можуть запитувати дані певного сервісу без знання подробиць внутрішньої реалізації даного сервісу. Це полегшить зміну сервісу або заміну окремих сервісів, не впливаючи на всю систему.

Реалізація сервісно-орієнтованої архітектури використовуючи GraphQL також може мати свої недоліки, а саме:

а) права власності на дані та контролю доступу. GraphQL вимагає ретельного розгляду прав власності на дані та контролю доступу;

б) проблеми з продуктивністю. Проблеми з продуктивністю виникають через збільшення кількості зворотних звернень до серверу;

в) додаткові затрати на конфігурацію схем GraphQL. Може знадобитись додатковий час для створення та підтримки схеми GraphQL і резолверів для кожного сервісу.

Для розробленої програмної системи «Orendar» було обрано RESTful веб-сервісів, оскільки веб-сервіси легко реалізувати за допомогою сучасних фреймворків, веб-сервіси можна використовувати в різних системах, оскільки вони підтримують методи HTTP, а веб-сервіси можуть підтримувати масштабованість. За допомогою веб-сервісів RESTful буде реалізована система прокату самокатів, велосипедів та автомобілів. Ця система буде використовуватися в службах Azure для аналізу методів створення сервіс-орієнтованого підходу.

5 ПРОЕКТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

5.1 Проектування архітектури програмної системи

Проектування архітектури для сервісно-орієнтованої програмної системи є складним і важливим процесом зі складною предметною областю. Враховуючи тематику програмної системи оренди транспорту, для бекенда веб-додатку була обрана багаторівнева архітектура, а для клієнтського додатку – Flux.

Сервер і клієнтська програма спілкуватимуться за допомогою REST API, а також використовуватиметься механізм авторизації. Реляційна база даних буде використовуватися для зберігання даних про транспорт, користувачів та інші предметні моделі.

Реляційна база даних буде використовуватися для зберігання даних про транспорт, користувачів та інші моделі об’єктів.

Усі сервіси, які будуть розроблені, зможуть створити образ Docker, щоб потім розмістити його в контейнерах Docker, як показано на рисунку 5.1.

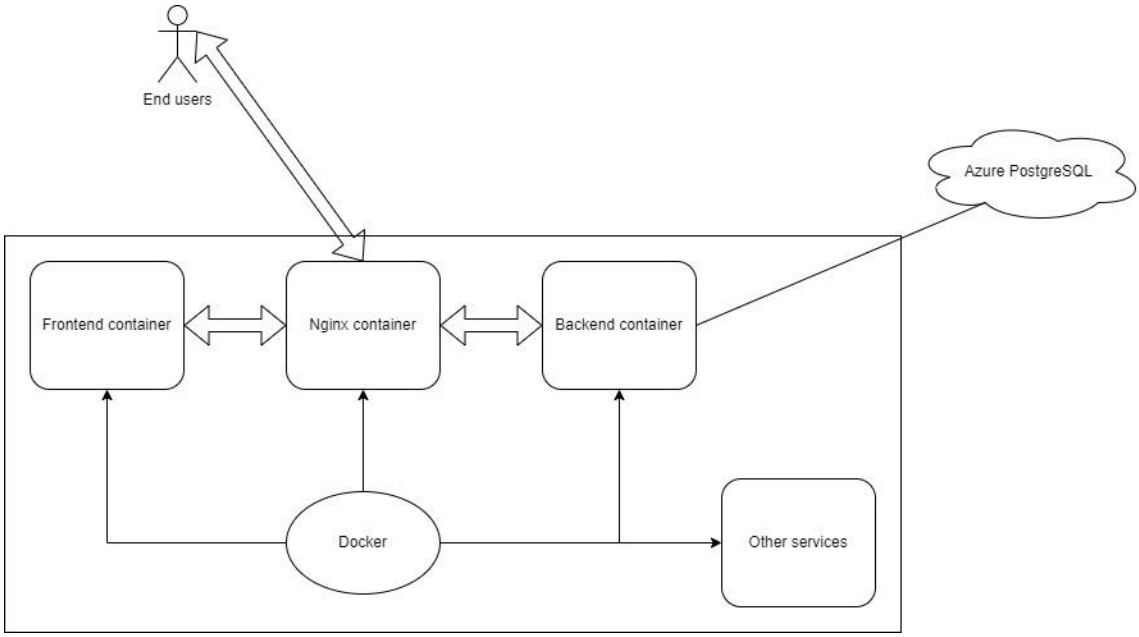

Рисунок 5.1 – Структура контейнерів Docker[Рисунок виконаний самостійно*]

Серверна та клієнтська частини будуть у контейнерах Docker, а Nginx[15] оброблятиме запити.

Nginx – це веб-сервер, який використовується як зворотний проксі, балансувальник навантаження, поштовий проксі та HTTP-кеш. Nginx перенаправить запит від кінцевого користувача до потрібного контейнера.

Створення образів Docker здійснюватиметься через Dockerfile.

Файл Docker – це текстовий документ, який містить усі команди, які користувач може викликати в командному рядку для створення образу Docker. Dockerfile міститиме основні файли для створення та налаштування проекту.

Зображення Docker можна створювати локально для локального тестування та за допомогою процесу CI/CD. Якщо це робиться через CI/CD, то в цьому випадку створення образів Docker має бути зроблено як останній крок, щоб перевірити сервіс на правильність виконання основних функцій.

Для проведення планових досліджень сервісів Azure необхідно побудувати прототип програмної системи з використанням шаблонів проектування, технологій і певних архітектурних стилів. Ця програмна система відповідатиме за прокат велосипедів, скутерів та автомобілів та надання їх у містах України.

5.2 Архітектура серверного застосунку

На стороні сервера буде використано архітектуру SOA, як показано на рисунок 5.2.

Архітектура містить 2 служби: основну службу та поштову службу. Ці служби спілкуються через шину обслуговування Azure за допомогою черги. Крім того, ці служби використовують ту саму базу даних, Azure PostgreSQL. Кожна служба відповідає за певну бізнес-логіку. Основна служба включає основну логіку для оренди транспортних засобів. Поштова служба відповідає за надсилання електронних листів користувачам для сповіщення користувачів, надсилання кодів підтвердження після реєстрації тощо.

Зв’язок між службами також використовує protobuf для стандартизації надсилання та отримання даних. Буфери протоколу (Protobuf) – це безкоштовний міжплатформний формат даних із відкритим кодом, який використовується для

серіалізації структурованих даних. Це корисно при розробці програм для спілкування один з одним через мережу або для зберігання даних.

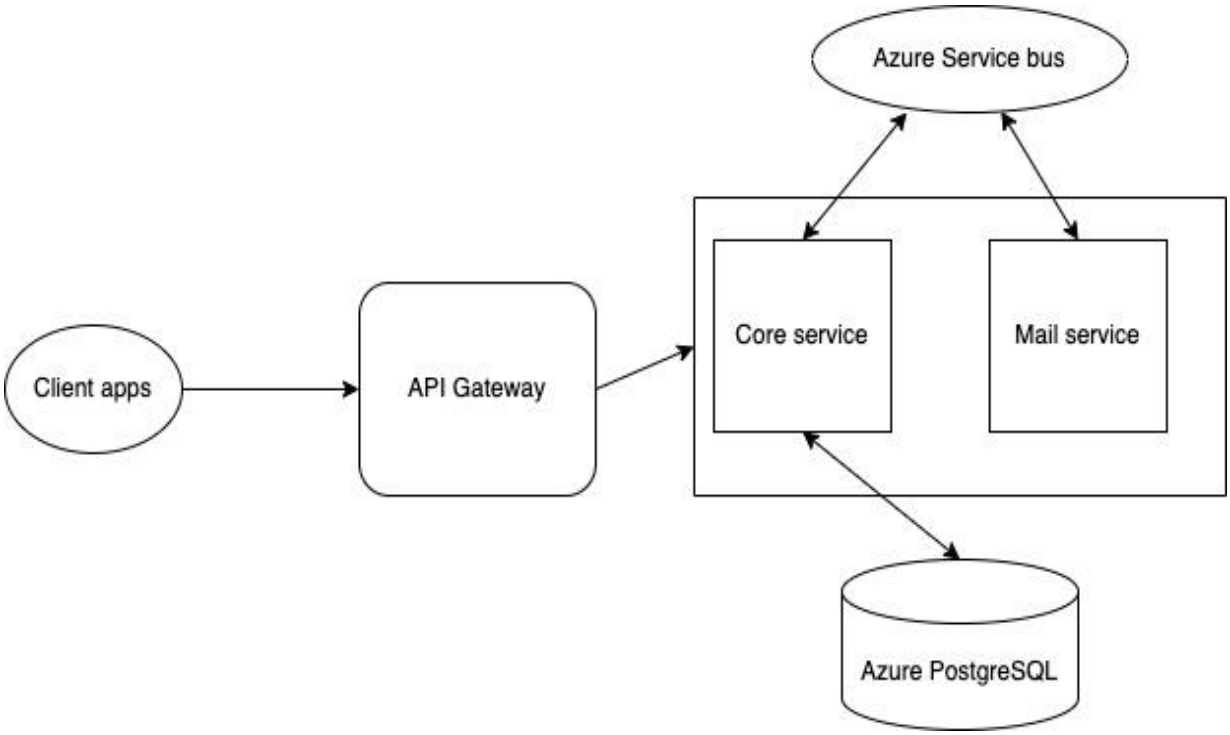

Рисунок 5.2 – Архітектура серверної частини[Рисунок виконаний самостійно*]

Крім того, зв’язок між службами та клієнтами використовує шлюз API, щоб приховати запити до служб і надати кінцевим користувачам зручний API.

Щоб розпочати розробку бізнес-логіки, спочатку необхідно спроектувати варіанти використання системи для різних користувачів за допомогою UML.

Під час розробки системи була створена діаграма UML, ця діаграма представляє ролі користувачів системи, а також інформацію про функціональність, доступну для кожної ролі, завдяки чому ви можете отримати уявлення про те, що кінцевий користувач може отримати.

На рисунку 5.3 показана діаграма прецедентів програмної системи оренди транспортних засобів.

Усі користувачі системи поділяються на дві основні ролі:

а) авторизований користувач;

б) неавторизований користувач.

Ця програмна система включає 2 ролі авторизованого користувача:

а) системний адміністратор;

б) авторизований користувач.

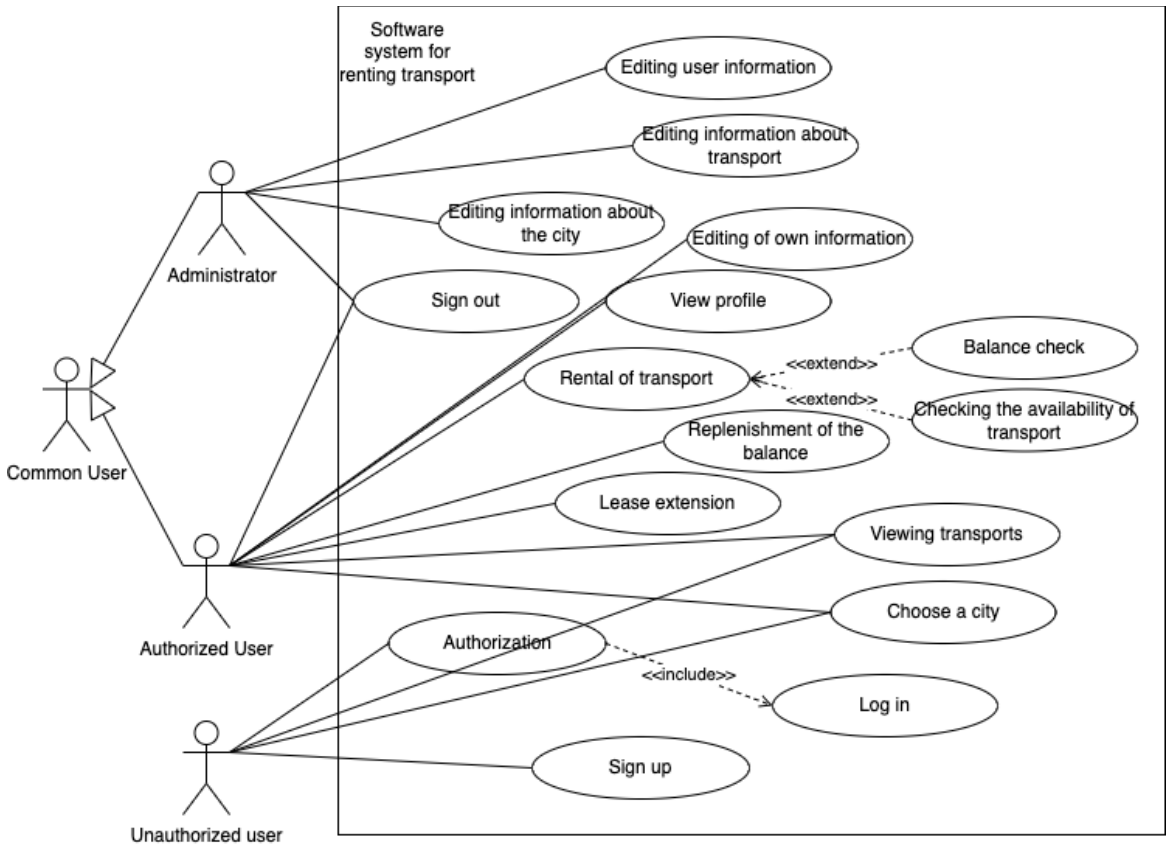

Рисунок 5.3 – Діаграма прецедентів[Рисунок виконаний самостійно*]

Кожна роль має свої обмеження щодо використання функціональності. Користувач, який має роль неавторизованого користувача, може авторизуватися в системі, тобто зареєструватися або авторизуватися в системі. Ви також можете переглянути транспорт і вибрати місто для подальшого вибору транспорту.

Користувач у ролі авторизованого користувача може переглядати транспорт, дивитися місто, орендувати автомобіль, переглядати свій профіль, редагувати свої дані та поповнювати баланс.

Користувач з роллю адміністратора керує даними користувача, транспортом і містами.

Для зберігання даних буде використано PostgreSQL[16]. База даних PostgreSQL сумісна з кількома основними мовами програмування та протоколами, включаючи C, C++, Go, Perl, Python, Java, .Net, Ruby, ODBC і Tcl. Це означає, що користувачі зможуть працювати мовою, яку вони знають найкраще, без ризику системних конфліктів. Для роботи з цією базою даних буде використовуватись базу даних Azure для сервера PostgreSQL. На рисунку 5.4 зображено ER-діаграму.

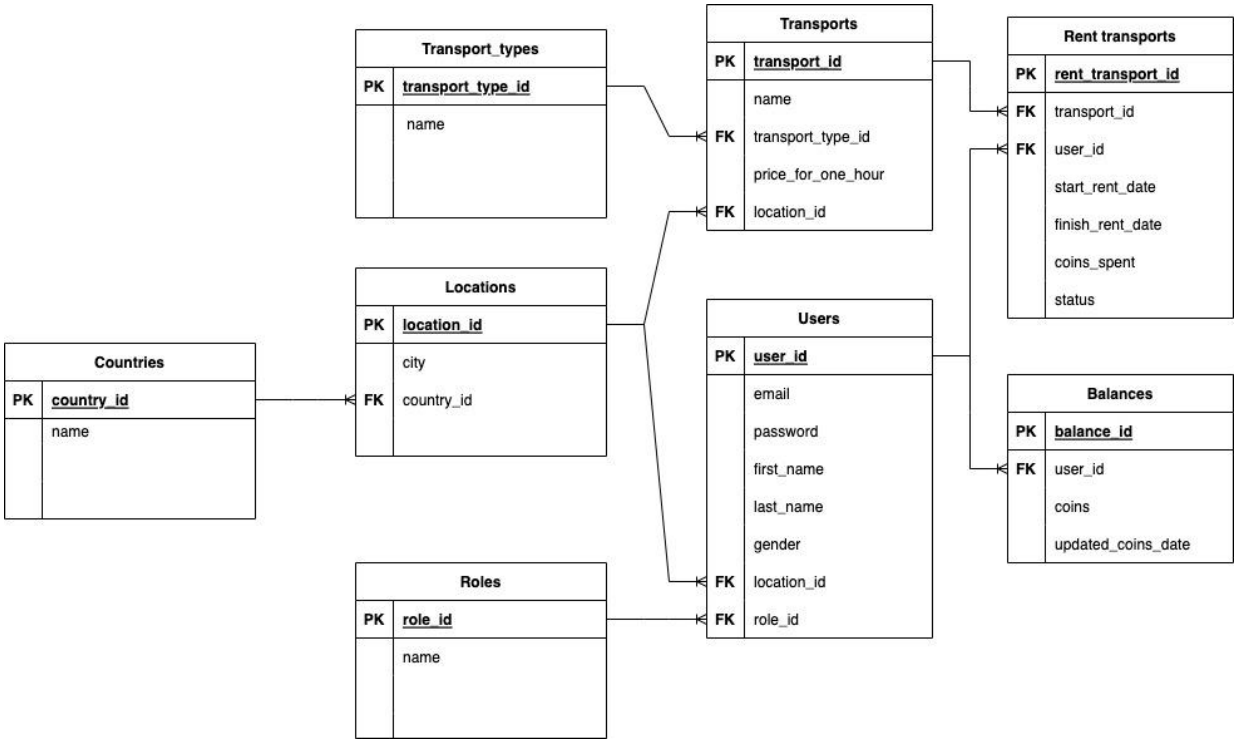

Рисунок 5.4 – ER-діаграма[Рисунок виконаний самостійно*]

Служби будуть побудовані на багаторівневій архітектурі, щоб розділити логіку. Одна з найбільших переваг багаторівневої архітектури полягає в тому, що до системи можна легко додати нові функції. Внесення змін до одного з рівнів системи жодним чином не вплине на модифікацію компонентів системи, якщо взаємодія відбувається через інтерфейси та ізоляцію моделі від інших компонентів.

Багаторівнева архітектура буде включати наступні рівні, як показано на рисунку 5.5. В даному випадку використовується чотирьох рівнена архітектура, яка включає наступні рівні:

а) controllers (API);

б) service layer;

в) domain layer;

г) data source layer.

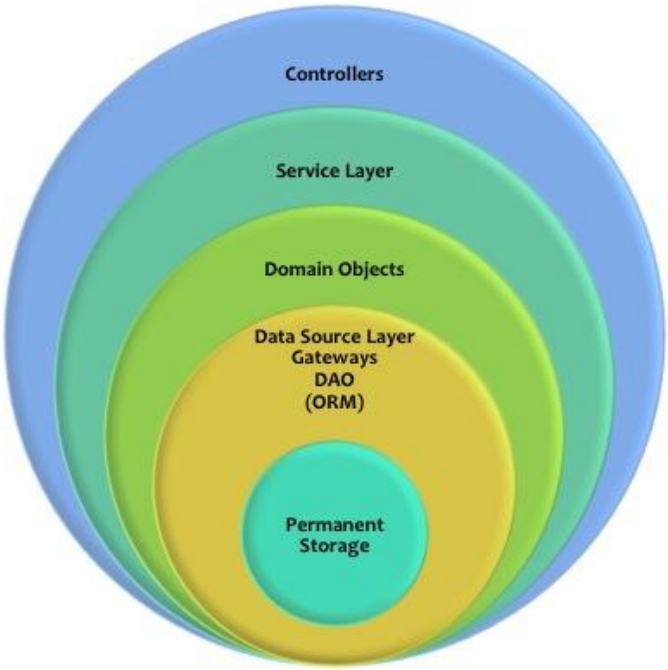

Рисунок 5.5 – Багаторівнена архітектура[Рисунок виконаний самостійно*]

Чим нижче рівень тим меншою інформацією він володіє. Наприклад, якщо брати domain рівень, то він лише знає про domain source рівень, а про service рівень він нічого не знає.

Рівень Controllers (API) надає можливість звертатися до серверної частини, використовуючи REST зв’язок, щоб отримати потрібну інформацію.

а) рівень Service відповідає за обробку серверної логіки.

б) рівень domain або рівень предметної області відповідає за уявлення понять прикладної предметної галузі, робочі стани, ділові регламенти;

в) рівень data source відповідає за роботу зі збереженням даних, спілкується з базою даних.

На діаграмі компонентів, як показано на рисунку 5.6, ви можете побачити частини кожної служби та те, як служби взаємодіють між собою.

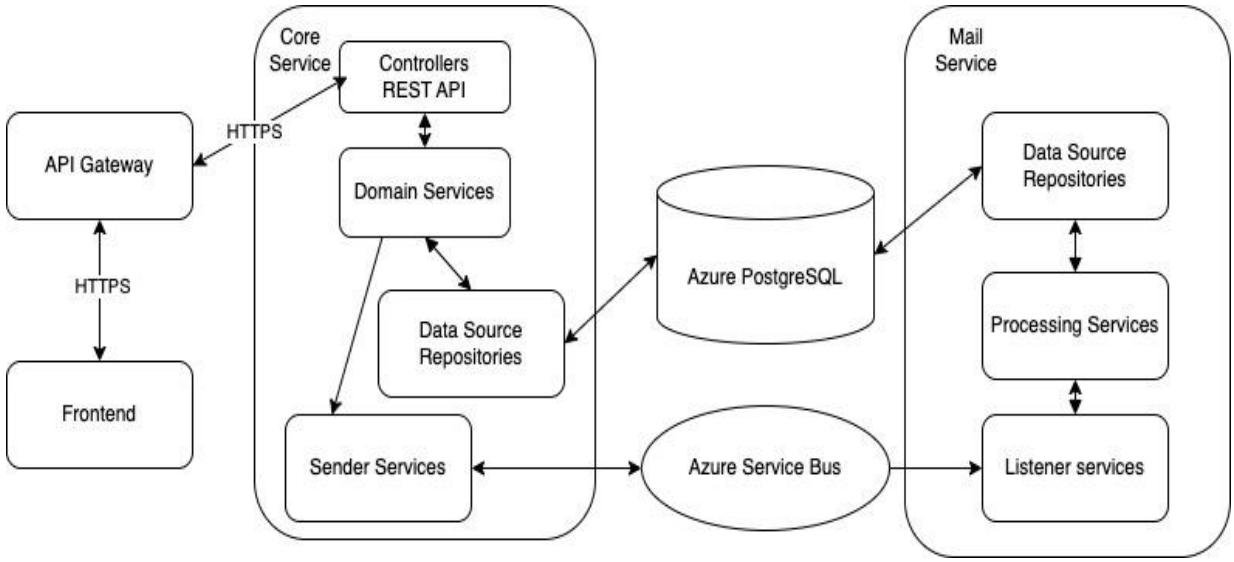

Рисунок 5.6 – Діаграма компонентів[Рисунок виконаний самостійно*]

Основна служба містить чотири рівні: контролери, служби домену, джерело даних і служби відправника. Служби домену можуть спілкуватися зі сховищами та службами відправників. Служби відправника надсилають повідомлення до шини обслуговування Azure, а інші отримують ці повідомлення. У цьому випадку поштова служба містить служби слухачів для отримання сповіщень від шини обслуговування Azure. Отримавши повідомлення, він починає їх обробку через служби обробки, відповідальні за надання певної логіки для надсилання пошти користувачам. Рівень джерела даних містить репозиторії та взаємодіє з Azure PostgreSQL.

Поштова служба містить три рівні: служби слухачів, служби обробки та джерела даних. Служби слухачів отримують повідомлення від шини обслуговування Azure і починають їх обробку за допомогою служб обробки. Служби обробки визначають шаблон для надсилання електронних листів. Це може бути підтвердження пароля, пропозиції на день/тиждень або нові оновлення.

Для реалізації основних і поштових сервісів було обрано Java та Spring Framework. Ця структура забезпечує швидкий розвиток для написання серверної частини системи.

Щоб максимально спростити зв’язок рівнів, буде використано принцип ін’єкції через Spring Framework, а саме модулі Spring Boot і Spring Core. Для роботи сервісів на різних пристроях з використанням необхідних версій бібліотек була обрана система автоматичного складання Gradle. Також, завдяки конфігурації Gradle, можна розбити структуру серверного додатку на окремі модулі, які відповідатимуть за окремі рівні архітектури.

5.3 Архітектура клієнтського застосунку

Для реалізації клієнтської частини було обрано мову програмування TypeScript та бібліотеку React. TypeScript забезпечить типізацію даних, що полегшить масштабування цієї програмної системи, а бібліотека React дозволить створювати компоненти, які використовуватимуться при реалізації архітектури Flux. Архітектура Flux включає такі компоненти, як Dispatcher, Store, React Views і Action Creators.

Архітектура Flux накладає обмеження на потік даних, зокрема, виключає можливість оновлення стану самих компонентів. Такий підхід робить потік даних передбачуваним і полегшує відстеження причин можливих помилок у програмному забезпеченні.

На рисунку 5.7 зображено Flux архітектуру, яка включає основні компоненти.

У Flux архітектури є наступні компоненти:

а) Actions – помічники, які передають дані в Dispatcher;

б) Dispatcher отримує ці дії і передає корисне навантаження зареєстрованим callback-ом;

в) Stores діють як контейнери для стану програми та логіки. Реальна робота програми відбувається у Stores. Stores, зареєстровані для прослуховування дій Dispatcher, відповідно і оновлюватимуть View;

г) Views – компоненти захоплюють стан Stores, а потім передають дочірнім компонентам.

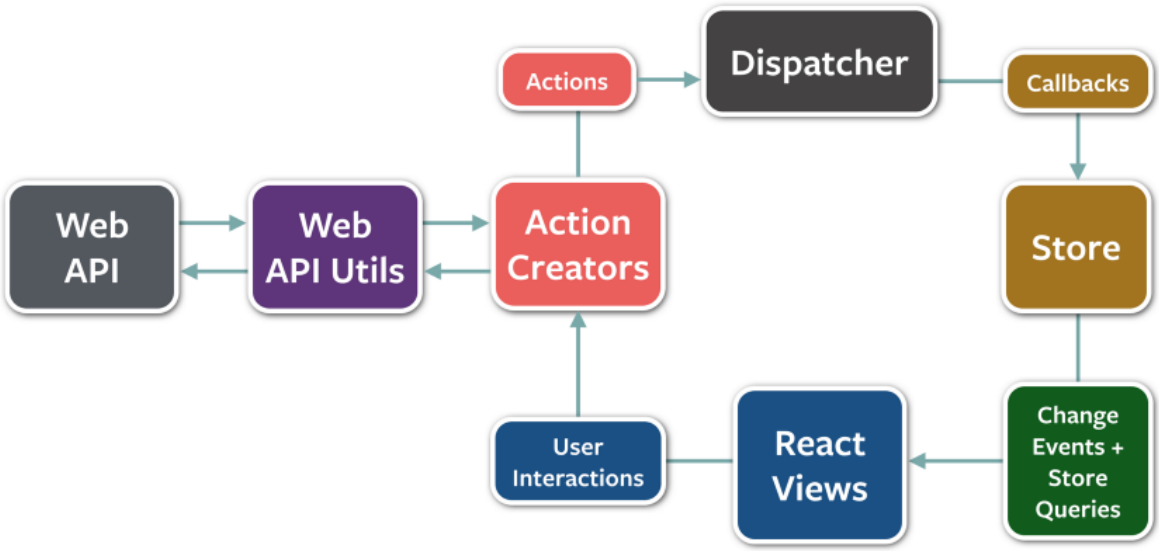

Рисунок 5.7 – Flux архітектура[29]

Flux пропонує статичну структуру передачі повідомлень, коли повідомлення отримує кожен компонент. Компонент вирішує, що робити із цим повідомленням. Це дозволяє обійти деякі архітектурні проблеми видавця-передплатника, пов’язані з порядком оповіщення компонентів при додаванні нових компонентів (проблеми масштабування), а також з додатковою складністю, пов’язаною з підпискою та відмовою від підписки протягом життєвого циклу компонентів, при якому можлива втрата значимих компонентів повідомлень.

6 АНАЛІЗ ФУНКЦІОНАЛЬНИХ МОЖЛИВОСТЕЙ СЕРВІСІВ ВІД MICROSOFT AZURE

Azure пропонує багато способів по розміщенню програмного коду використовуючи сервіси, які можна побачити на рисунку 6.1.

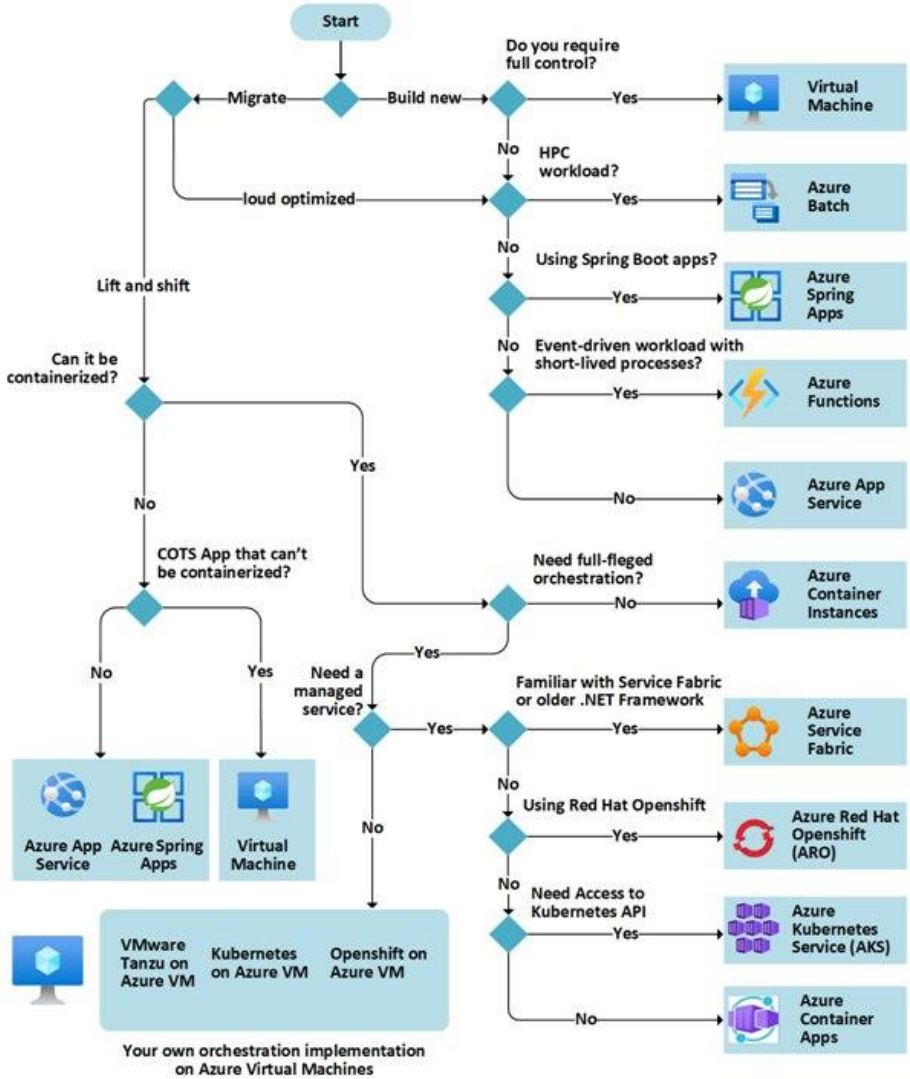

Рисунок 6.1 – Діаграма сервісів[25]

Для даного дослідження буде використовуватись наступні сервіси:

а) Azure Container Apps;

б) Azure Kubernetes Service;

в) Azure Red Hat OpenShift.

6.1 Аналіз функціональних можливостей Azure Container Apps

Azure Container Apps дає змогу запускати мікросервіси та контейнерні програми на безсерверній платформі.

На рисунку 6.1 зображено приклади сценаріїв застосування Azure Container Apps.

З цього можна побачити, що поширені способи використання Azure Container Apps є:

а) розгортання кінцевих точок API;

б) розміщення програм фонової обробки;

в) обробка event-driven процесінг;

г) запуск мікросервісів.

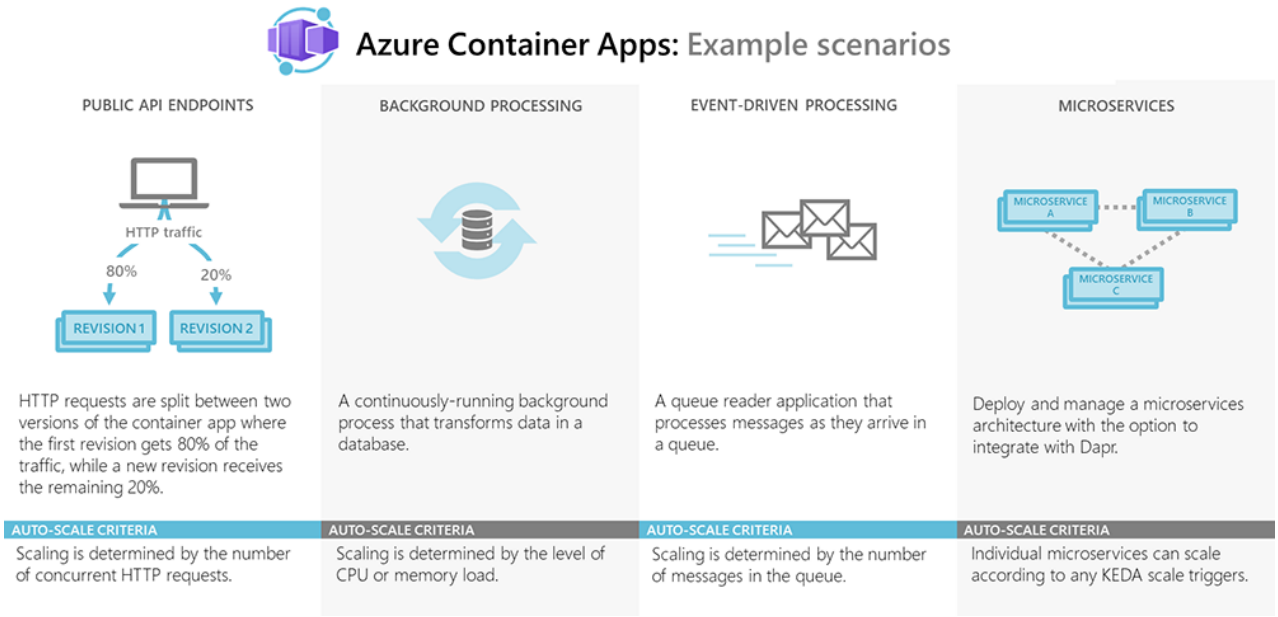

Рисунок 6.1 – Приклади сценаріїв застосування Azure Container Apps[2]

Програми, створені на основі Azure Container Apps, можуть динамічно масштабуватися на основі таких характеристик:

а) HTTP трафік;

б) event-driven процесінг;

в) завантаження ЦП або пам’яті.

Azure Container Apps дозволяє виконувати код програми, упакований у будь-який контейнер, і не залежить від середовища виконання чи моделі програмування.

Azure Container Apps керує автоматичним горизонтальним масштабуванням за допомогою набору декларативних правил масштабування. Коли програма-контейнер масштабується, нові екземпляри програми-контейнера створюються на вимогу. Ці екземпляри відомі як репліки. Коли ви вперше створюєте програму-контейнер, правило масштабу встановлюється на нуль.

6.2 Аналіз функціональних можливостей Azure Kubernetes Service

Azure Kubernetes Service (AKS) [5] – це служба оркестровки керованого контейнера на основі системи Kubernetes з відкритим кодом, яка доступна в публічній хмарі Microsoft Azure. Користувачі даного сервісу можуть використовувати AKS для обробки критично важливих функцій, таких як розгортання, масштабування та керування контейнерами Docker і додатками на основі контейнерів.

Kubernetes – це де-факто платформа з відкритим вихідним кодом для оркестровки контейнерів, але зазвичай вимагає багато накладних витрат на керування кластером. AKS допомагає керувати значною частиною накладних витрат, зменшуючи складність розгортання та завдань керування. AKS розроблено для користувачів та компаній, які хочуть створювати масштабовані програми за допомогою Docker і Kubernetes, використовуючи архітектуру Azure.

Кластер AKS можна створити за допомогою інтерфейсу командного рядка Azure (CLI), порталу Azure або Azure PowerShell. Користувачі також можуть створювати параметри розгортання на основі шаблонів за допомогою шаблонів Azure Resource Manager.

Основними перевагами AKS є гнучкість, автоматизація та зменшення витрат на керування для адміністраторів і розробників.

Наприклад, AKS автоматично налаштовує всі вузли Kubernetes, які контролюють і керують робочими вузлами під час процесу розгортання, і виконує низку інших завдань, включаючи інтеграцію Azure Active Directory (AD),

підключення до служб моніторингу та налаштування розширених мережевих функцій, таких як як маршрутизація програми HTTP.

Коли користувач створює кластер AKS, автоматично створюється та налаштовується площина керування (Control plane).

Площина керування – це керований ресурс Azure, до якого користувач не може отримати прямий доступ.

На рисунку 6.2 зображено представлення модуля Kubernetes, що інкапсулює контейнер, і те, як пакети збираються у вузли.

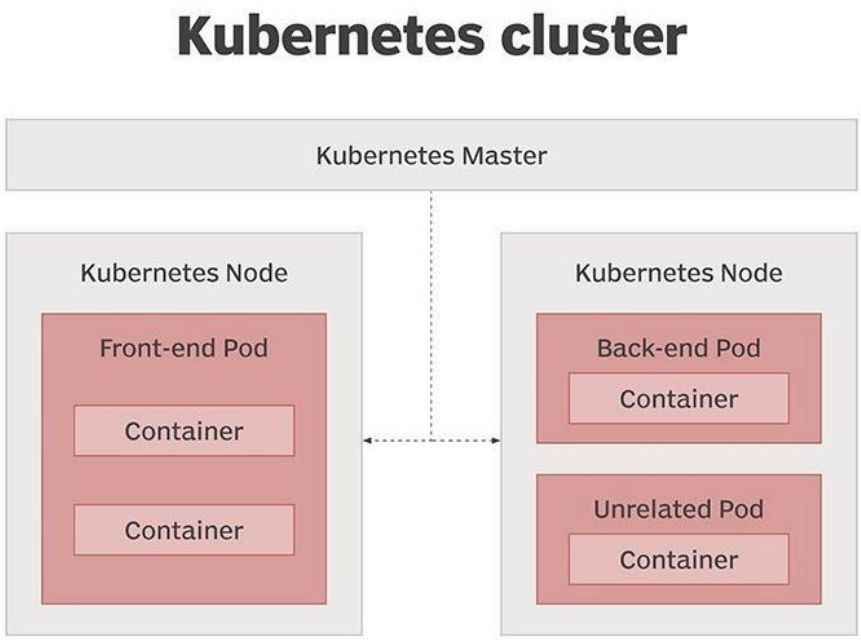

Рисунок 6.2 – Представлення модуля Kubernetes[5]

Кластер AKS має принаймні один вузол. Центральний процесор і пам’ять – це ресурси вузла, які використовуються для того, щоб допомогти вузлу функціонувати як частина кластера. Вузли зі схожою конфігурацією групуються в пули вузлів.

Розгортання AKS також охоплює дві групи ресурсів. Одна група – це лише ресурс служби Kubernetes, а інша – група ресурсів вузла. Група ресурсів вузла

містить усі ресурси інфраструктури, пов’язані з кластером. Для створення та керування іншими ресурсами Azure потрібен принципал служби або керований ідентифікатор.

6.3 Аналіз функціональних можливостей Azure Red Hat OpenShift

Azure Red Hat OpenShift [4] розширює Kubernetes, як показано на рисунку 6.3 зображено високорівненву архітектуру даного сервісу.

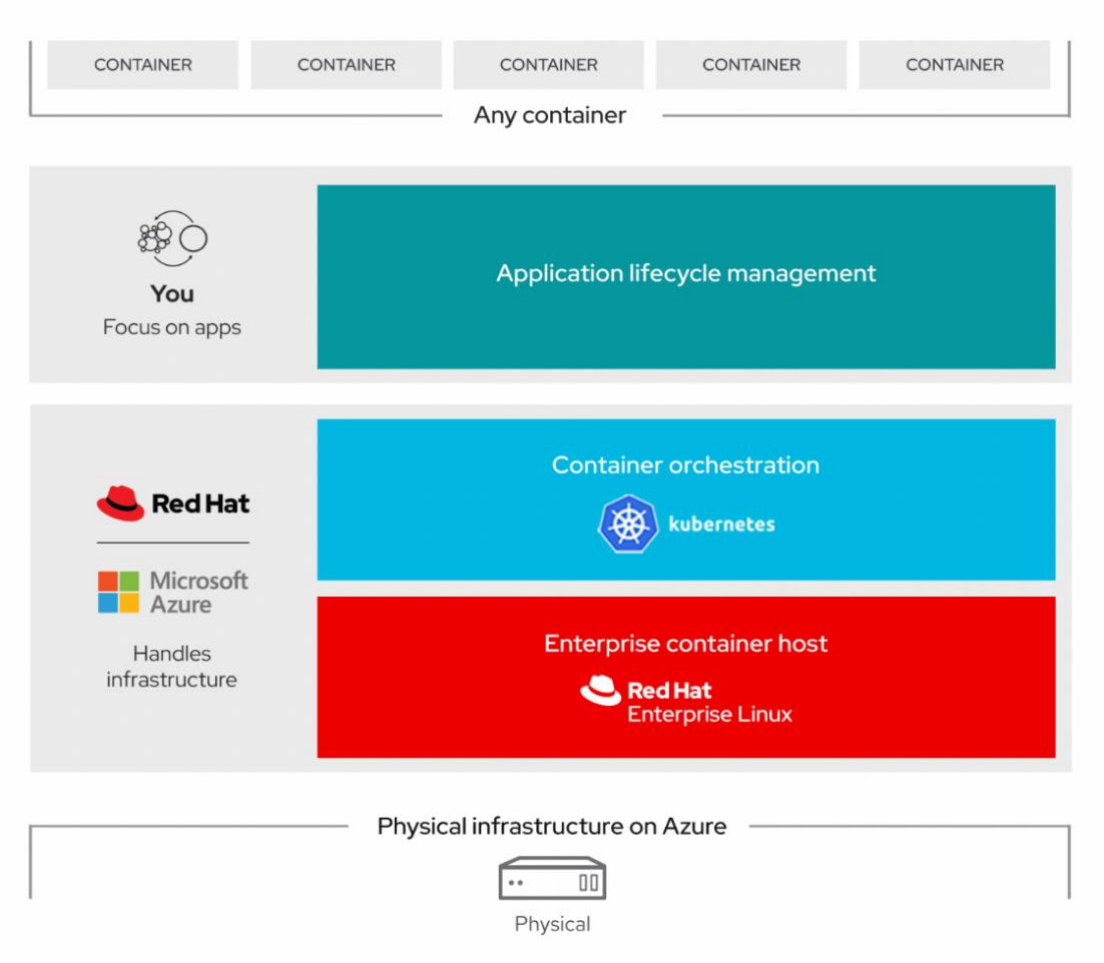

Рисунок 6.3 – Високорівнена архітектура Azure Red Hat OpenShift[3]

Запуск контейнерів у виробництві за допомогою Kubernetes потребує додаткових інструментів і ресурсів. Це часто включає в себе необхідність жонглювати реєстрами зображень, керуванням сховищами, мережевими

рішеннями, а також інструментами журналювання та моніторингу – усе це має бути версійним і тестованим разом.

Створення додатків на основі контейнерів вимагає ще більшої інтеграції з проміжним програмним забезпеченням, фреймворками, базами даних і інструментами CI/CD.

Azure Red Hat OpenShift об’єднує все це в єдину платформу, спрощуючи роботу IT-командам, водночас надаючи командам те, що їм потрібно для виконання. Окрім цього сервіс Microsoft Azure Red Hat OpenShift дозволяє розгортати повністю керовані кластери OpenShift.

Можна вибрати власний реєстр, мережеві рішення, рішення для зберігання та CI/CD або використовувати вбудовані рішення для автоматизованого керування вихідним кодом, збірки контейнерів і додатків, розгортання, масштабування, керування працездатністю тощо. Azure Red Hat OpenShift забезпечує інтегрований досвід входу через Azure Active Directory.

Вузли Azure Red Hat OpenShift виконуються на віртуальних машинах Azure, що дає можливість підключити сховища до вузлів та pod і оновити компоненти кластера.

7 РЕАЛІЗАЦІЯ ПРОГРАМНОЇ СИСТЕМИ

7.1 Реалізація головного сервісу

При реалізації головного сервісу використовувалась мова програмування Java, фреймворк Spring та база даних Azure PostgreSQL.

Базова архітектура для реалізації веб сервісу була обрана багатошарова, щоб розділити логіку на декілька шарів. Вийшло така кількість шарів:

а) web REST API;

б) business logic layer;

в) persistence layer.

Для реалізації Web REST API шару використовувались модулі Spring Web, Spring REST. Нижче можна побачити приклад коду для класу, який відповідає за обробку запитів для підтвердження коду, який надається, щоб активізувати акаунт.

```
@RestController
public class ConfirmCodeController {

      private final UserCodeService userCodeService;
      private final UserService userService;
      private final SenderService senderService;

      @GetMapping("/api/confirm/code")
      public ResponseEntity<?> confirmCode(@PathParam("value") int
value) {
      Code newCode = new Code(value);
      ConfirmCode confirmedCode =
userCodeService.confirmCode(newCode);
      if(confirmedCode.status().value().equals("expired")) {
      User foundUser =
userService.findUserByEmail(confirmedCode.email());
      User updatedUserStatus = User.builder()
      .id(foundUser.id())
      .role(foundUser.role())
      .email(foundUser.email())
      .firstName(foundUser.firstName())
      .lastName(foundUser.lastName())
      .gender(foundUser.gender())
      .location(foundUser.location())
      .status(new Status("ACTIVATE"))
      .build();
      User activatedUser =
userService.updateUserInformation(updatedUserStatus);
      return new
ResponseEntity<>(convertUserToUserResponse(activatedUser),
HttpStatus.OK);
      }
```

```
        throw new ValidationException("You cannot confirm the code");
        }
}
```

В даному коді використовується анотація RestController над класом, що надає можливість додавати нові REST запити через спеціальні анотації GetMapping, PostMapping, PutMapping та інш. Для кожної предметної області є ряд своїх REST запитів, як можна побачити з таблиці 1.

Таблиця 1 – REST запити головного сервісу[Таблиця виконана самостійно*]

Область	REST запити
Користувачі	1. POST /api/login 2. POST /api/register 3. PUT /api/users 4. GET /api/user, parameters: id чи email 5. GET /api/users
Балансовий акаунт	1. POST /api/balance/create 2. GET /api/balance 3. PUT /api/balance
Підтвердження коду	1. GET /api/confirm/code, parameter code 2. GET /api/code/recent, parameter email
Транспорт	1. GET /api/transports/all 2. POST /api/transports 3. PUT /api/transports 4. POST /api/rentTransport 5. GET /api/transports/rent
Локація	1. POST /api/locations 2. PUT /api/locations 3. GET /api/locations 4. GET /api/countries 5. POST /api/countries

Кінець таблиці 1[Таблиця виконана самостійно*]

Область	REST запити
Локація	6. PUT /api/countries

Для реалізації шару, який пов’язан з бізнес логікою використовувались гнучкі інтерфейси для кожного з сервісу. Гнучкі інтерфейси включають перечень методів, які буде або виконує сервіс. Гнучкий інтерфейс надає перелік API методів для використання певної логіки кінцевим користувачем. Кожен гнучкий інтерфейс повинен включати його реалізацію. Приклад коду гнучкого інтерфейсу.

```
Public interface SenderService {
    void sendMessage(Email email, ConfirmCode code);
}
```

Даний сервіс SenderService включає метод, який отримує два аргументи Email та ConfirmCode. Після отримання цих аргументів робить відправку повідомлення до Azure Service Bus, як можна побачити з коду реалізації.

```
@Service
@RequiredArgsConstructor
public class SenderServiceImpl implements SenderService {
    private static final Logger LOGGER =
LoggerFactory.getLogger(SenderServiceImpl.class);
    private final JmsTemplate jmsTemplate;

    @Override
    public void sendMessage(Email email, ConfirmCode code) {
        LOGGER.info("Message was sent to the queue. Message body: " +
email.toString());
        MailOuterClass.Mail mail = MailOuterClass.Mail.newBuilder()
                .setEmail(email.value())
                .setCode(String.valueOf(code.code().value()))
                .setType("confirm")
                .build();
        LOGGER.info("The message was prepared to sent to the queue.
Message body: " + mail.toString());

        jmsTemplate.convertAndSend("mail", mail.toByteArray());

        LOGGER.info("Message was sent to the queue.");
    }
}
```

В даному коді робиться відправка повідомлення до черги від Azure Service Bus через сервіс JmsTemplate, який був інтегрований як бібліотека. Клас

MailOuterClass.Mail був згенерований на основі proto файлу. Код proto файлу наведений нижче.

```
Syntax = “proto3”;
option java_package = “ua.nure.makieiev.proto”;

message Mail {
  string email = 1;
  string type = 2;
  string code = 3;
}
```

Для стандартизації моделей на сервісах використовується protobuf, як формат стандартизації даних.

Protobuf (Буфер протоколів) – це безкоштовний між платформний формат даних із відкритим кодом, який використовується для серіалізації структурованих даних. Це корисно при розробці програм для спілкування один з одним через мережу або для зберігання даних. Метод включає мову опису інтерфейсу, яка описує структуру деяких даних, і програму, яка генерує вихідний код з цього опису для створення або аналізу потоку байтів, який представляє структуровані дані.

Всі proto файли знаходяться в окремому репозиторії та підключаються до сервісів, як під модулі. Таким чином буде уникнуто дублювання коду та стандартизуємо моделі, які передаються між сервісами.

Для шару спілкування з базою даних використовується модуль Spring Data, щоб пришвидшити розробку програмної системи. Spring Data надає можливість пов’язати сутності з таблицями з бази даних та надати гнучкий інтерфейс, щоб формувати потрібні методи. Для цього використовується технологія ORM.

Приклад класу проекції на таблицю confirm_codes в базі даних.

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = “confirm_codes”)
@Entity
public class ConfirmCodeEntity {

    @Id
   @Column(name = “confirm_code_id”)
   private String uuid;
   @Column(name = “email”)
   private String email;
```

```
        @Column(name = "code")
        private int code;
        @Column(name = "status")
        private String status;
    }
```

В даному коді використовуються анотації, які мають назву таблиці, назву рядків в таблиці, та яке поле відповідає за ідентифікатор даної таблиці.

Наступним кроком було створено Repository для даної сутності, як показано нижче кодом.

```
    @Repository
    public interface ConfirmCodeRepository extends
JpaRepository<ConfirmCodeEntity, String> {
        Optional<ConfirmCodeEntity> findByCode(int code);
        Optional<ConfirmCodeEntity> findByEmail(String email);
    }
```

В даному інтерфейсі ConfirmCodeRepository створено 2 методи findByCode та findByEmail. Spring Data аналізує дані імена та формує SQL запит використовуючі власну реалізацію SQL будівельника. Також базовий інтерфейс JpaRepository надає базовий набір методів для взаємодії з таблицею, а саме додавання, редагування або видалення даних з таблиці.

7.2 Реалізація сервісу поштаря

Даний веб сервіс відповідає за відправку листів на електроні адреси користувачів. Сервіс може відправити інформацію про код після реєстрації або інформацію об оренді транспорту та інша інформація.

При реалізації сервісу поштаря використовувались ті самі технології, як для головного сервісу, а саме мова програмування Java, фреймворк Spring.

Даний сервіс слухає чергу та коли додається нове повідомлення, то він зчитує та аналізує до якого плану дане повідомлення відноситься. Для аналізу використовується стратегія повідомлень, як зображено в коді.

```
    @SpringBootApplication(scanBasePackages =
"ua.nure.orendar.mailer.mailservice")
    @EnableJms
    @Import(AppConfig.class)
    public class MailServiceApplication {
        private static final Logger LOGGER =
LoggerFactory.getLogger(MailServiceApplication.class);
        private static final String QUEUE_NAME = "mail";
        private final StrategyFactory strategyFactory;
```

```
        @Autowired
        public MailServiceApplication(StrategyFactory strategyFactory) {
            this.strategyFactory = strategyFactory;
        }
        public static void main(String[] args) {
            SpringApplication.run(MailServiceApplication.class, args);
        }
        @JmsListener(destination = QUEUE_NAME, subscription = "mail-
service", containerFactory = "jmsListenerContainerFactory")
        public void receiveMessage(byte[] bytes) throws
InvalidProtocolBufferException {
            LOGGER.info("Starting work with email strategy");
            MailOuterClass.Mail mail =
MailOuterClass.Mail.parseFrom(bytes);
            SendStrategy sendStrategy =
strategyFactory.findSendStrategy(mail.getType());
            LOGGER.info("Email information: {}", mail.toString());
            sendStrategy.send(mail.getEmail(),
Integer.parseInt(mail.getCode()));
        }
    }
```

В даному коді використовується стратегія відправки, яка обирається в залежності від типу повідомлення. Кожне повідомлення, яке обробляється повинно включати його тип. Також в даному коді можна побачити анотацію JmsListener, яка включає назву черги та ім'я підписки.

Якщо головний сервіс відправив в чергу код для активації акаунту після реєстрації, то обереться стратегія SendRegistrationCodeStrategy, яка відповідає за відправку коду на електрону адресу користувача. Код даної стратегії наведений нижче.

```
    @Service
    public class SendRegistrationCodeStrategy implements SendStrategy {

        private final JavaMailSender emailSender;

        @Autowired
        public SendRegistrationCodeStrategy(JavaMailSender emailSender) {
            this.emailSender = emailSender;
        }

        @Override
        public void send(String email, int code) {
            SimpleMailMessage message = new SimpleMailMessage();
            message.setFrom("orendar-ol@ukr.net");
            message.setText(String.format("Your confirmation code is %s.
Please, confirm your password", code));
            message.setSubject("Confirm registration on the orendar
service");
            message.setTo(email);
            emailSender.send(message);
```

```
        }
}
```

В даному коді, метод send приймає параметри, як email та code. Використовуючи сторонню бібліотеку JavaMailSender робиться відправка листа на електрону адресу користувача.

Після відправки повідомлення, можна побачити лист на пошті, як показано на рисунку 7.1.

Рисунок 7.1 – Отримання коду підтвердження[Рисунок виконаний самостійно*]

7.3 Реалізація клієнтської частини

Для реалізації клієнтської частини використовувалась мова програмування TypeScript та бібліотеки React, styled-components, material-ui та redux.

Бібліотека React надає можливість створювати компоненти клієнтського застосунку та перевикористовувати їх. Наприклад, це може бути меню, таблиці, форми та інш.

Бібліотека material-ui надає можливість використовувати набір готових компонентів, що дає прискорення у розробці клієнтської частини.

Бібліотека styled-components створює кастомні компоненти з CSS стилями, що замінює звичайні класи для HTML дому.

Бібліотека Redux надає можливість реалізувати Flux-архітектуру та маніпулювати з даними.

Клієнтський застосунок має декілька станів для різних користувачів.

Якщо користувачі неавторизовані, то вони лише можуть побачити перелік транспорту, як показано на рисунку 7.2.

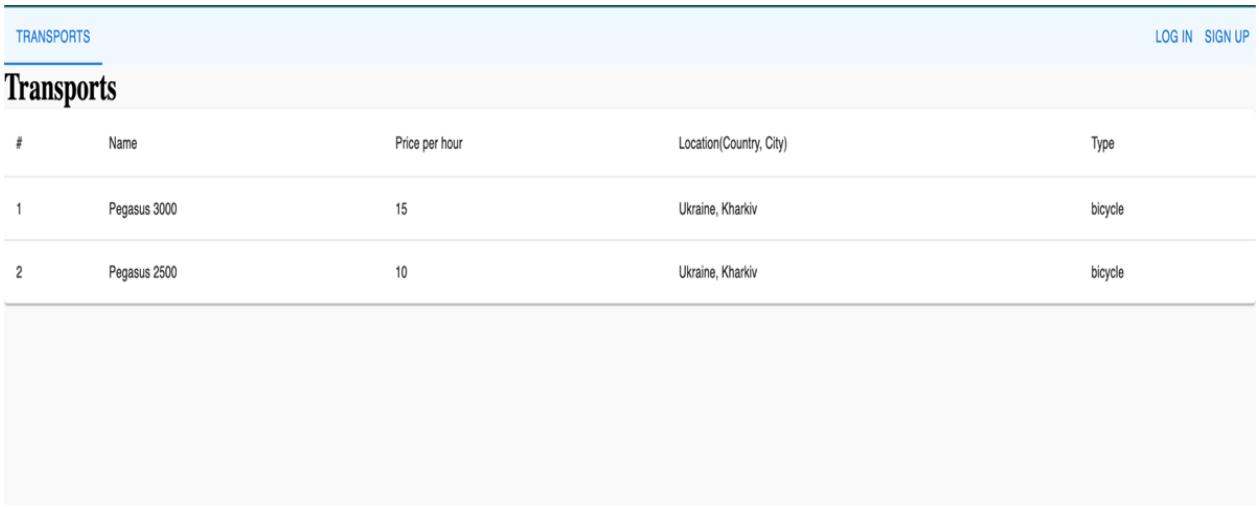

Рисунок 7.2 – Перелік транспорту[Рисунок виконаний самостійно*]

Якщо користувач має акаунт, то він може натиснути на кнопку Log In, щоб відкрити діалогове вікно для входу, як показано на рисунку 7.3.

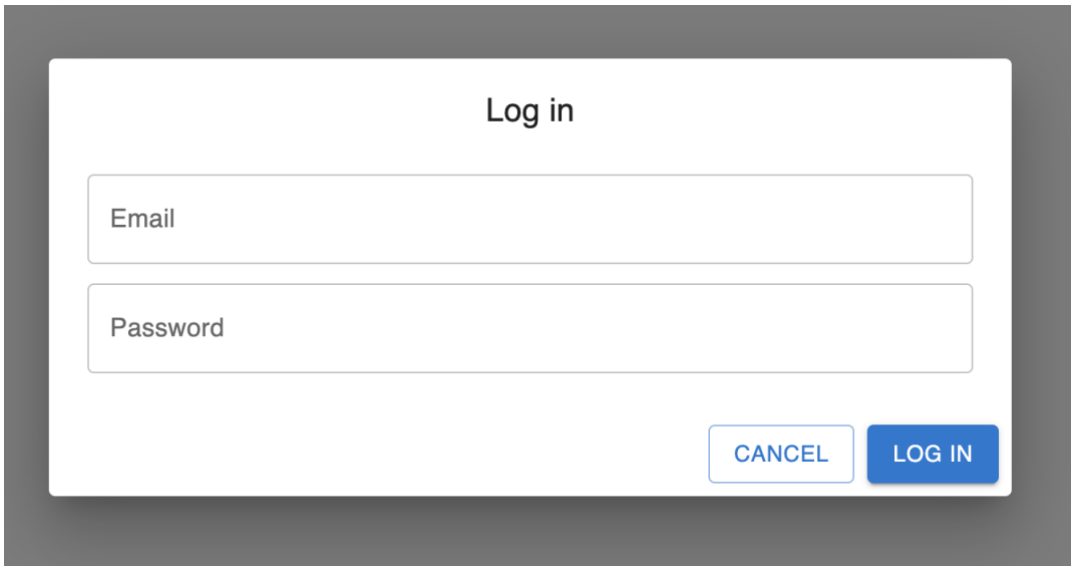

Рисунок 7.3 – Діалогове вікно для входу в систему[Рисунок виконаний самостійно*]

Після успішної аутентифікації, користувачу буде доступна вкладка з профілем де він може побачити свої дані про місце знаходження та інформацію про свій баланс, як показано на рисунку 7.4.

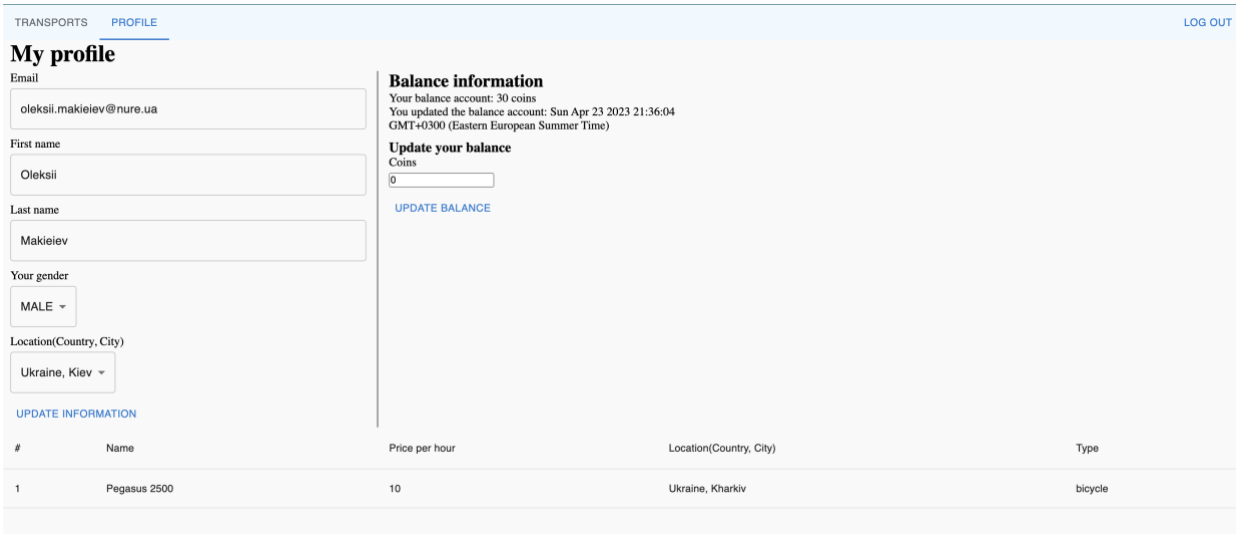

Рисунок 7.4 – Сторінка з профілем користувача[Рисунок виконаний самостійно*]

Якщо користувач тільки що зареєструвався та ще не підтвердив код, то йому дані функції будуть недоступні, замість цього він буде бачити поле для воду коду, як показано на рисунку 7.5.

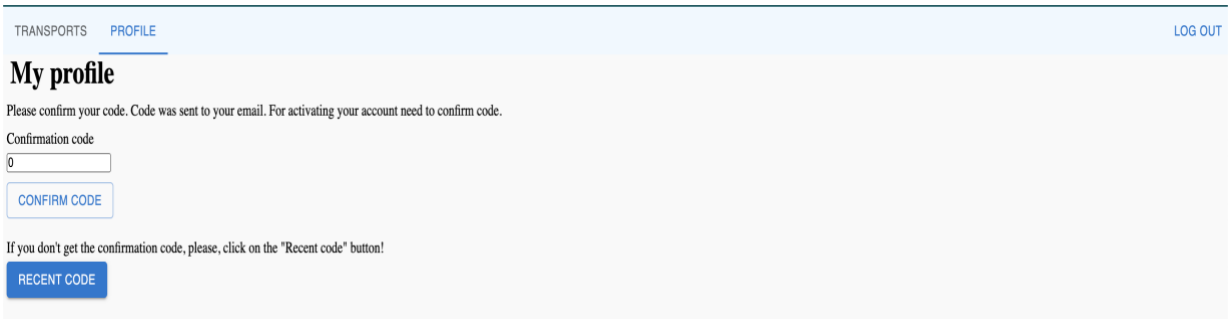

Рисунок 7.5 – Форма для підтвердження коду[Рисунок виконаний самостійно*]

Окрім цього користувачу доступна можливість ще раз надіслати собі код, якщо він не отримав при реєстрації або загубив серед інших листів на електронній адресі.

Після успішного активування акаунту та поповнення балансу, користувач може перейти до сторінки з транспортом та орендувати певний транспорт, як показано на рисунку 7.6.

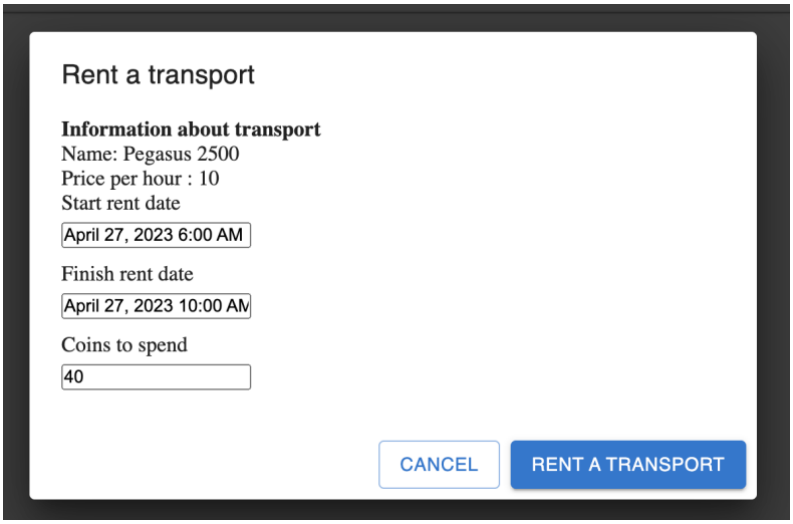

Рисунок 7.6 – Оренда транспорту[Рисунок виконаний самостійно*]

Після того, як користувач орендував транспорт він може побачити його в своєму власному профілі.

8 ЕКСПЕРИМЕНТ

Після реалізації програмної системи буде порівняно сервіси Azure за такими критеріями:

а) вартість і ціноутворення;

б) особливості та функціональність;

в) швидкість розгортання;

г) підтримка реєстрів контейнерів;

д) підтримка моніторингу та журналювання.

8.1 Вартість і ціноутворення

Для розрахунку було використано офіційний калькулятор цін – https://azure.microsoft.com/en-us/pricing/calculator/ і розраховували ціни за 1 місяць. Після дослідження було зроблено основні характеристики кожної служби та використали результати для створення таблиці 2 із вартістю та ціноутворенням для послуг Azure.

Таблиця 2 – Результати вартості та ціноутворення для послуг Azure[Таблиця виконана самостійно*]

	Azure Container Apps	Azure Kubernetes Services	Azure Red Hat OpenShift
vCPU	4	4	4
Memory	16 GB	16 GB	16 GB
Requests per month	60 million	-	-
Nodes	-	4	4
Temporary storage	-	150 GB	-
Master nodes	-	-	8 vCPU, 32 GB RAM
Price per month	771.04$	744.60$	2545.87$

8.2 Особливості та функціональність

Після дослідження було визначено деякі функції та функції для контейнерних програм Azure, служби Azure Kubernetes і Azure Red Hat OpenShift і представили їх у таблиці 3.

Таблиця 3 – Інформація про функціональність сервісів[Таблиця виконана самостійно*]

Назва сервісу	Перелік функцій
Azure Container Apps	1. Docker image support 2. Auto-scaling 3. Integration with various Azure services 4. Network Configuration 5. Security protection 6. Monitoring and logging 7. Support for multiple programming languages 8. Scaling services 9. Deployment speed 10. Versioning 11. Support for different types of containers 12. Cluster Management 13. Backup and Restore 14.Support for different operating systems
Azure Kubernetes Services	1. Optimized Kubernetes Management 2. Auto-scaling 3. Docker image support 4. Integration with Azure Monitor 5. Integration with Azure Active Directory 6. Security protection

Продовження таблиці 3[Таблиця виконана самостійно*]

Назва сервісу	Перелік функцій
Azure Kubernetes Services	7. Integration with Azure DevOps 8. Support for multiple programming languages 9. Scaling services 10. Deployment speed 11. Versioning 12. Support for different types of containers 13. Cluster Management 14. Backup and Restore 15. Automatic recovery 16. Remote access 17. Integration with Azure Policy 18. Flexibility 19. Cross-platform support 20. Integration with Azure Arc 21. Integration with Azure Policy for Kubernetes 22. Network management 23. Integration with Azure Private Link 24. Support for different types of accounts
Azure Red Hat OpenShift	1. Kubernetes-based container orchestration 2. Red Hat Enterprise Linux operating system 3. Integration with Azure services 4. Enterprise-grade security 5. High availability and disaster recovery options 6. Scalability and elasticity 7. Monitoring and logging 8. Resource utilization tracking and optimization 9. Automated container builds and deployments

Продовження таблиці 3[Таблиця виконана самостійно*]

Назва сервісу	Перелік функцій
Azure Red Hat OpenShift	10. Multi-cluster management 11. Application templates and deployment patterns 12. Developer tools and SDKs 13. Integrated development environment integration 14. Application lifecycle management 15. Automated testing and quality assurance 16. Continuous integration and delivery pipeline 17. Git integration and version control 18. Application scaling and load balancing 19. Networking and service mesh capabilities 20. Role-based access control 21. Compliance and audit logging 22. Integration with external identity providers 23. Customizable policies and quotas 24. Resource tagging and management 25. Secure image registry and distribution 26. Integration with third-party registries 27. Application portability across hybrid cloud environments 28. Kubernetes Operators for automated application management 29. Full-stack observability with Prometheus and Grafana 30. Distributed tracing with Jaeger 31. Logging with Elasticsearch, Fluentd, and Kibana 32. Integration with Azure DevOps for end-to-end software development 33. Container-native storage 34. Data persistence options 35. Serverless computing with Azure Functions

Кінець таблиці 3[Таблиця виконана самостійно*]

Назва сервісу	Перелік функцій
Azure Red Hat OpenShift	36. Artificial intelligence and machine learning services 37. Edge computing and IoT integration 38. Compatibility with Red Hat OpenShift ecosystem and marketplace 39. Flexible pricing and billing options 40. Enterprise-level support and service level agreements

8.3 Підтримка реєстрів контейнерів

Після дослідження було визначено реєстри контейнерів, які підтримують кожний з сервісів Azure Container Apps, Azure Kubernetes Service та Azure Red Hat OpenShift. Всього було знайдено 16 контейнерів, як показано в таблиці 4.

Таблиця 4 – Інформація про контейнера[Таблиця виконана самостійно*]

Ім’я сервісу	Azure Container Apps	Azure Kubernetes Services	Azure Red Hat OpenShift
Docker Hub	+	+	+
Azure Container Registry	+	+	+
GitHub Container Registry	+	+	+
Amazon Elastic Container Registry	+	+	+
Google Cloud Registry	+	+	+
Harbor Registry	+	+	+
Jfrog Container Registry	+	+	+
Quay.io	+	+	+
Red Hat Quay	+	+	+

Кінець таблиці 4[Таблиця виконана самостійно*]

Ім’я сервісу	Azure Container Apps	Azure Kubernetes Services	Azure Red Hat OpenShift
GitLab Container Registry	+	+	+
IBM Cloud Container Registry	+	+	+
Artifactory		+	+
Quay Enterprise		+	+
Vmware Harbor Registry		+	+
Oracle Cloud Infrastructure Registry			+
Azure Stack Hub Container Registry			+

8.4 Підтримка моніторингу та журналювання

Після дослідження було визначено функції моніторингу та журналювання для контейнерних програм Azure, служби Azure Kubernetes і Azure Red Hat OpenShift. Функції моніторингу та журналювання допомагають відстежувати та діагностувати стан програми для покращення продуктивності.

Монітор Azure для моніторингу та аналізу показників, журналів і трасування розподілених програм. Azure Log Analytics для збору, аналізу та візуалізації журналів із різних джерел у Azure, зокрема журналів контейнерів. Azure Application Insights для моніторингу та аналізу продуктивності програм, що працюють у контейнерах.

Інформаційна панель Kubernetes для візуалізації стану кластера Kubernetes та його компонентів. Azure Log Analytics для збору, аналізу та візуалізації журналів із різних джерел у Azure, зокрема журналів контейнерів і кластерів Kubernetes. Prometheus для збору, моніторингу та аналізу показників Kubernetes і контейнера.

Grafana для візуалізації даних моніторингу з Prometheus та інших джерел. Kibana для візуалізації журналів, зібраних за допомогою Elasticsearch і Logstash. Elasticsearch для зберігання та індексування журналів з різних джерел. Вільно збирає та пересилає журнали з контейнерів до Elasticsearch або іншого сховища. Jaeger для відстеження розподілених програм і виявлення проблем у взаємодії між компонентами програми.

Консоль OpenShift для візуалізації стану кластера та його компонентів. Istio для керування мережевим трафіком між компонентами програми та захисту від загроз безпеки між мікросервісами. Інші описи цих технологій ви можете прочитати в попередніх параграфах.

Отримані результати по моніторингу та журналюванню було представлено у вигляді таблиці 5 для показу кількості функцій моніторингу та журналювання, які підтримують служби Azure.

Таблиця 5 – Підтримка моніторингу та журналювання[Таблиця виконана самостійно*]

Ім’я сервісу	Azure Container Apps	Azure Kubernetes Services	Azure Red Hat OpenShift
Azure Monitor	+	+	+
Azure Log Analytics	+	+	+
Azure Application Insights	+		
Kubernetes Dashboard		+	
Prometheus		+	+
Grafana		+	+
Kibana		+	+
Elasticsearch		+	+
Fluentd		+	+
Jaeger		+	+
OpenShift Console			+

Кінець таблиці 5[Таблиця виконана самостійно*]

Ім’я сервісу	Azure Container Apps	Azure Kubernetes Services	Azure Red Hat OpenShift
Istio.OpenShift Console			+

8.5 Швидкість розгортання

Для порівняння швидкості розгортання було переміщенно систему в репозиторії GitHub, щоб розгорнути їх у різних службах.

Для розгортання буде використовуватись конвеєри для розгортання нових образів у контейнерних програмах Azure, службі Azure Kubernetes і Azure Red Hat OpenShift. Для більш точного результату було запущено розгортання 10 разів. У випадку Azure Red Hat OpenShift було використано Argo CD для налаштування процесорів GitOps, одним із яких є розгортання образу.

Argo CD – це декларативний інструмент безперервної доставки GitOps для Kubernetes.

Для створення зображення кожного сервісу було написано Dockerfile. Приклад коду даного файлу наведений нижче.

```
FROM gradle:7.4-jdk17 AS build
COPY –chown=gradle:gradle . /home/gradle/src
RUN chown -R gradle:gradle /home/gradle
USER gradle
WORKDIR /home/gradle/src
RUN git submodule update –init
RUN gradle build
FROM openjdk:17-jdk-slim
EXPOSE 8081
RUN mkdir /app
COPY –from=build /home/gradle/src/build/libs/orendar-backend-0.0.1-
SNAPSHOT.jar /app
ENTRYPOINT ["java", "-XX:+UnlockExperimentalVMOptions", "-
XshowSettings:vm", "-Djava.awt.headless=true", "-
Djenkins.install.runSetupWizard=false", "-
Djava.security.egd=file:/dev/./urandom","-jar","/app/orendar-backend-0.0.1-
SNAPSHOT.jar"]
```

В даному файлі можна побачити, що використовується Gradle версії 7.4 та Java 17. Після чого робиться підтягування під модуля, щоб отримати proto файли.

Дані файли потрібні для комунікації по протобаф. Також, контейнер буде мати порт 8081. Також в полі ENTRYPOINT написано додаткова конфігурація для даного зображення.

На рисунку 8.1 відображено процес розгортання Container Apps.

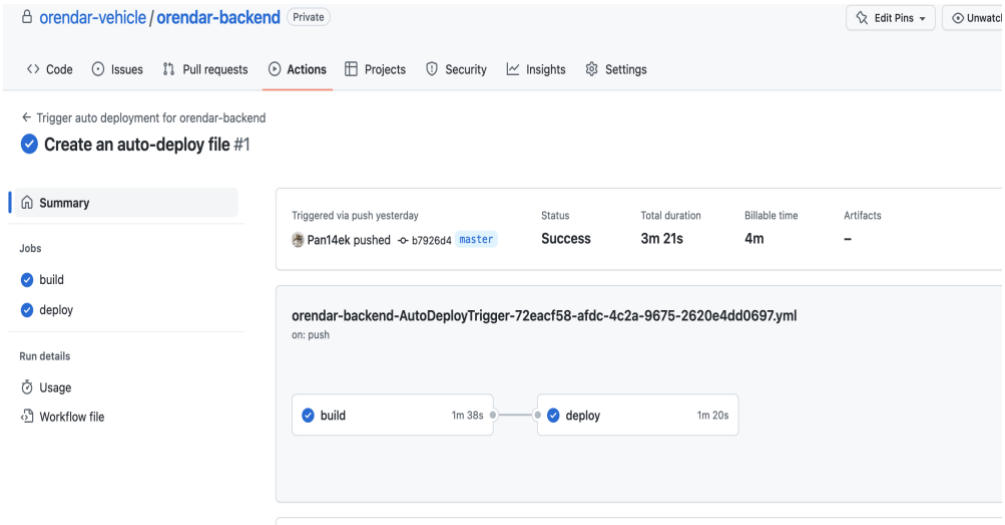

Рисунок 8.1 – Процес розгортання Container Apps[Рисунок виконаний самостійно*]

На рисунку 8.2 відображено процес розгортання для Azure Kubernetes Service.

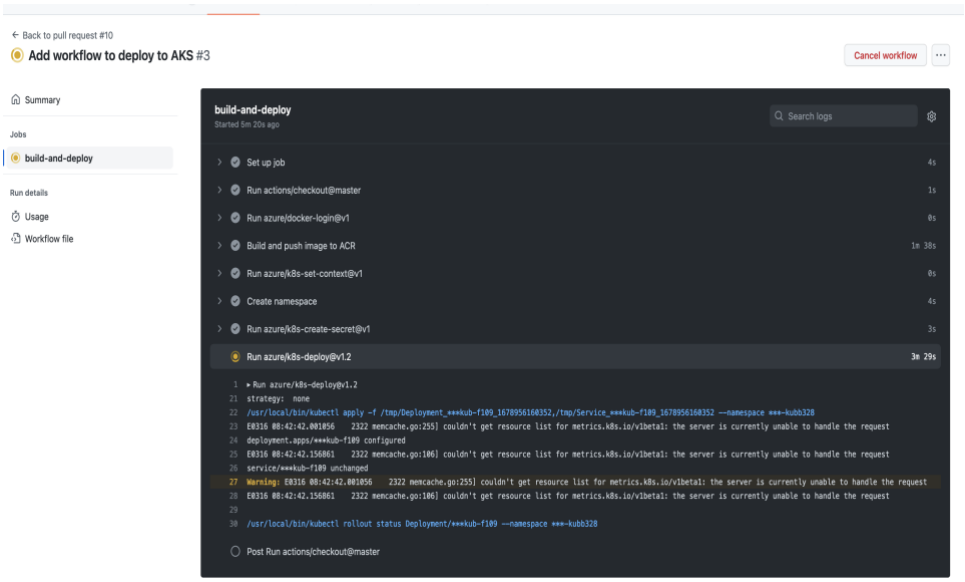

Рисунок 8.2 – Процес розгортання Azure Kubernetes Service[Рисунок виконаний самостійно*]

Кожен сервіс має аналогічний Dockerfile, щоб отримати власне зображення.

Використовуючи GitHub Actions було написано простий GitHub pipeline, щоб робити білд проекту та розгортати його на сервісі.

Після того як було зроблено певні зміни в окремій гілці та додали її до головної гілки, то автоматично запуститься GitHub Actions, щоб зробити білд проекту, а потім розгорнути його на певному сервісі.

Azure Red Hat OpenShift надає багато можливостей по розгортанню сервісів.

Можна робити це власноруч без написання GitHub Pipeline чи Jenkins Pipeline[10] або використовуючи сторонні сервіси, які інтегровані до платформи OpenShift. Наприклад, це може бути Argo CD, Jenkins та інші допоміжні служби.

Як раніше було згадано для Azure Red Hat OpenShift використовувався Argo CD, як показано на рисунку 8.3.

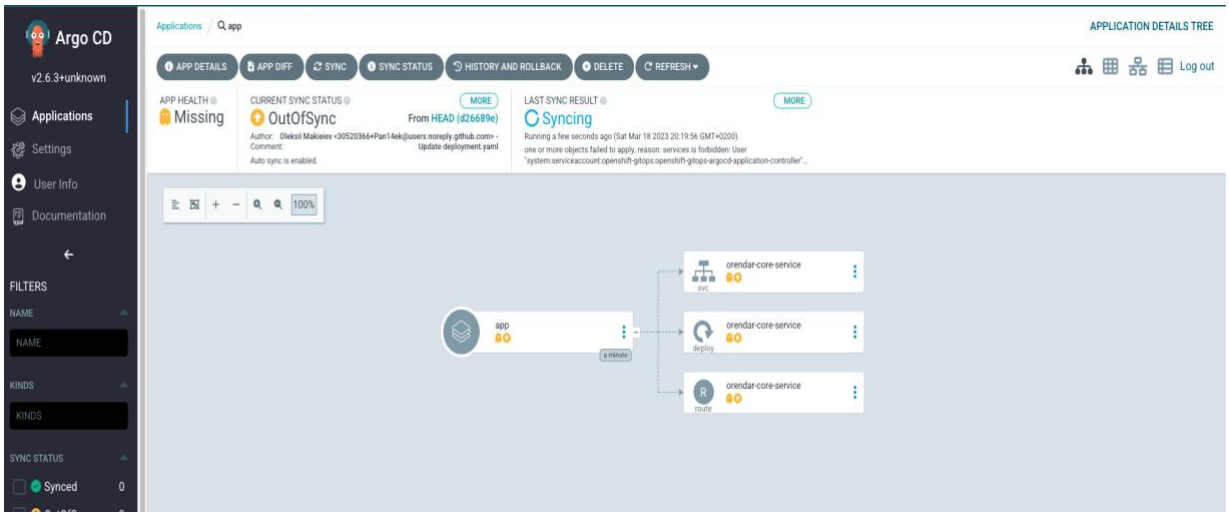

Рисунок 8.3 – Використання Argo CD для розгортання сервісів[Рисунок виконаний самостійно*]

Після того, як було розгорнуто сервіси 10 разів було отримано середні результати для кожної служби:

а) Azure Container Apps – 80 секунд;

б) Azure Kubernetes Service – 207 секунд;

в) Azure Red Hat OpenShift – 220 секунд.

За результатами вимірювання було створено діаграму для візуалізації отриманих результатів, як можна побачити на рисунку 8.4.

Azure Kubernetes Service та Azure Red Hat OpenShift мають майже схожі результати з відхиленням у 10-20 секунд. На 4-6 проміжку результати вийшли однаковими.

Якщо дивитись на результати Azure Container Apps, то можна побачити, що результати 2-4 мають зростаючий характер, що свідчить про те, що розгортання займало більше часу ніж зазвичай. Це може бути пов’язаним з тим, що сервер мав на той час менше ресурсів та був більш навантаженим, що спричинило таку затримку у часі. Якщо порівнювати інші результати, то вони мають відхилення 2-16 секунд.

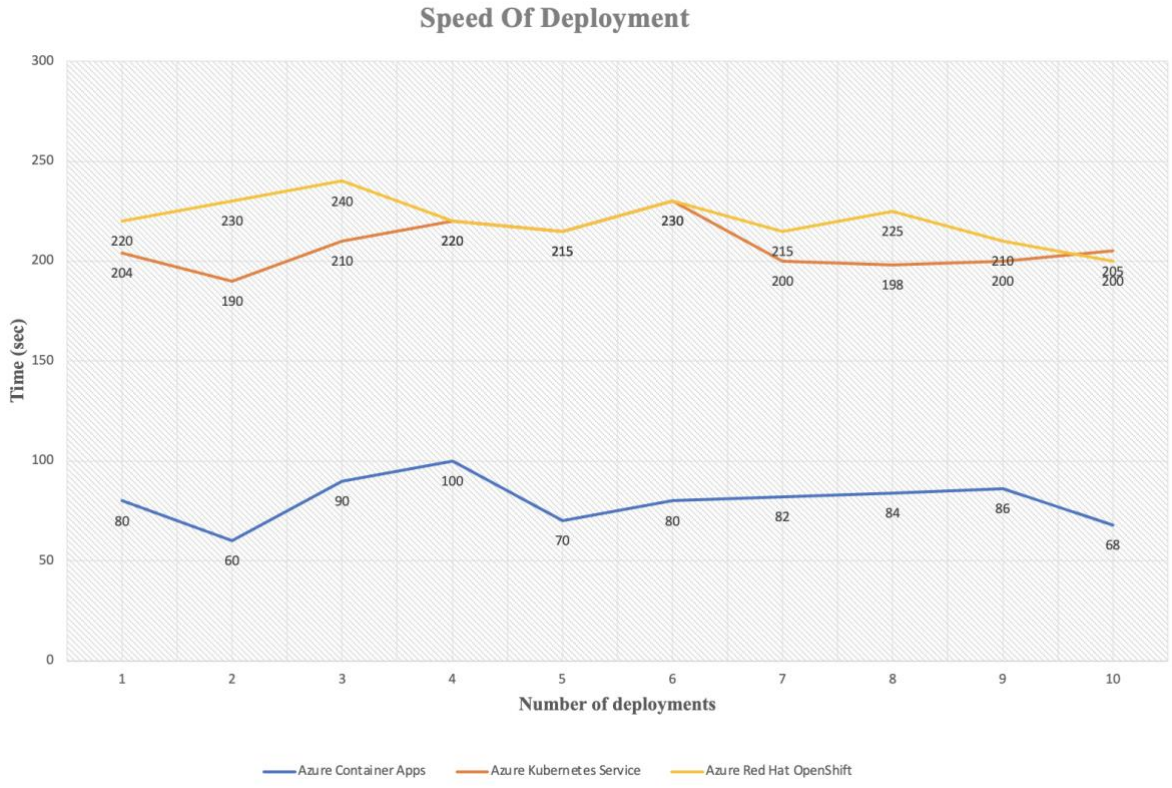

Рисунок 8.4 – Швидкість розгортання кожного сервісу[Рисунок виконаний самостійно*]

Після аналізу методів створення було створено таблицю з результатами, як показано в таблиці 6.

Таблиця 6 – Загальні результати по експерименту[Таблиця виконана самостійно*]

	Вартість і ціноутворення ($)	Особливості та функціональність (кількість)	Швидкість розгортання (sec)	Підтримка реєстрів контейнерів (кількість)	Підтримка моніторингу та журналювання (кількість)
Azure Container Apps	771.04	14	80	11	3
Azure Kubernetes Service	744.60	24	204	14	9
Azure Red Hat OpenShift	2545.87	40	220	16	10

Якщо підсумувати отримані результати по експерименту, то можна зробити такі висновки по використанню даних сервісів.

Якщо програмне забезпечення являю собою простим застосунком, яке включає клієнтську частину та лише один веб сервер, то для таких цілей може підійти Container Apps через те що він простий у використанні та має мінімальний базовий функціонал для контролювання та відслідковування змін, журналювання, навантаження та інше.

Якщо програмне забезпечення розвивається та створюються нові сервіси та дані сервіси треба об’єднувати та контролювати, то для цього може підійти Azure Kubernetes Service або Azure Red Hat OpenShift. Якщо потрібно лише автоматизація та зменшення витрат на керування для адміністраторів і розробників, то тут краще

використовувати AKS, бо він незатратний у використанні та надає майже ті самі можливості по журналюванню та використання певних реєстрів контейнерів.

Якщо програмне забезпечення має великий масштаб та треба контролювати та формувати звіти по журналюванню та відслідковувати навантаження та інші процесів, то тут краще використовувати Azure Red Hat OpenShift, який надає просунуті можливості по розгортанню сервісів, використання сторонніх сервісів, які допоможуть покращити та автоматизувати мануальні процеси.

У цьому дослідженні було використані наступні функції в Azure Container Apps для програмної системи: Docker image support, Network Configuration, Monitoring and logging, Support for multiple programming languages, Support for different types of containers.

У цьому дослідженні було використані наступні функції в Azure Kubernetes Service для програмної системи: Docker image support, Integration with Azure Monitor, Support for different types of containers, Support for multiple programming languages, Optimized Kubernetes Management, Cluster Management.

У цьому дослідженні було використані наступні функції в Azure Red Hat OpenShift для програмної системи: Git integration and version control, Monitoring and logging, Developer tools and SDKs, Application templates and deployment patterns, Continuous integration and delivery pipeline.

Для всіх трьох випадків використовувався реєстр контейнер Azure Container Registry.

Для розробленої програмної системи «Orendar» на перших етапах розвитку найкращим варіантом буде використання Azure Container Apps через низьку кількість сервісів та клієнтську частину. З подальшим розвитком програмної системи та збільшенням кількості сервісів треба мати можливість ними керувати, тому для цього кращим варіантом буде Azure Kubernetes Service. Виходячи з цін за місяць та великою кількості функціоналу, Azure Red Hat OpenShift буде надмірним у використанні для розробленої програмної системи, а це значить що програмна система не буде використовувати повний функціонал даної платформи.

ВИСНОВКИ

При написанні магістерської кваліфікаційної роботи було спроектовано та реалізовано сервісно-орієнтовану програмну систему. Дана програмна система включає серверну та клієнтську частину. Серверна частина включає в собі два RESTful веб сервіси: головний сервіс та сервіс для відправки листів на електроні адреси. Дані сервіси спілкуються між собою використовуючи Azure Service Bus по протоколу Protobuf. Також сервіси надають REST запити, щоб клієнтська частина могла отримувати потрібні дані для відображення.

За результатами дослідження предметної галузі було проаналізовано існуючі аналоги по оренді транспорту, сервісно-орієнтовані програмні системи та існуючі хмарні платформи.

Було проаналізовано існуючі методи по створенню сервісно-орієнтованої програмної системи, а саме:

а) мікросервісна архітекрура;

б) RESTful веб-сервіси;

в) GraphQL.

Також були наведені переваги та недоліки по кожній технології для створення сервісно-орієнтованої системи.

За результатами аналізу по методам створенням сервісно-орієнтованої програмної системи було обрано RESTful веб-сервіси, через те що веб-сервіси легко реалізувати за допомогою сучасних фреймворків, бібліотек. Також RESTful веб-сервіси підтримують методи HTTP та масштабованість.

У результаті проектування була отримана система для проведення експерименту на Azure сервісах, були розглянуті основні моменти по проектуванню для серверної та клієнтської частин, та загальної архітектури.

У результаті дослідження сервісів від Microsoft Azure було проаналізована такі сервіси, як Azure Container Apps, Azure Kubernetes Service, Azure Red Hat OpenShift за такими критеріями, як:

а) вартість і ціноутворення;

б) особливості та функціональність;

в) швидкість розгортання;

г) підтримка реєстрів контейнерів;

д) підтримка моніторингу та журналювання.

За отриманими результатами дослідження було зроблено висновки про доречність використання кожного Azure сервісу та при яких користувацьких потребах треба обирати той чи інший сервіс.

Отримані результати можуть бути використані в реальному житті при розробці нового сервісу, проекту або міграції з інших хмарних провайдерів на Azure.

Магістерська кваліфікаційна робота має перспективи на подальше дослідження даної області по хмарним технологіям та платформі Azure. Ще є ряд питань для подальшого дослідження:

а) дослідження складності при взаємодії великої кількості сервісів та як можна вирішити дане питання використовуючи сервіси від Azure;

б) дослідження масштабування сервісно-орієнтованої архітектури;

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Azure Kubernetes Service. / URL: https://azure.microsoft.com/en-us/products/kubernetes-service (дата звернення: 10.02.2023).

2. Azure Container Apps overview. / URL: https://learn.microsoft.com/uk-ua/azure/container-apps/overview (дата звернення: 10.02.2023).

3. Azure Red Hat OpenShift. / URL: https://azure.microsoft.com/en-us/products/openshift/ (дата звернення: 10.02.2023).

4. Introduction Azure Red Hat OpenShift. / URL: https://learn.microsoft.com/en-us/azure/openshift/intro-openshift (дата звернення: 10.03.2023).

5. Introduction Azure Kubernetes Service. / URL: https://www.techtarget.com/searchcloudcomputing/definition/Azure-Kubernetes-Service-AKS (дата звернення: 12.03.2023).

6. CI/CD for containers. / URL: https://learn.microsoft.com/en-us/azure/architecture/solution-ideas/articles/cicd-for-containers (дата звернення: 17.03.2023).

7. An Overview of Service-Oriented Architecture in Retail. URL: https://web.archive.org/web/20070701160401/http://msdn2.microsoft.com/en-us/library/bb264584.aspx (дата звернення: 17.03.2023).

8. Google Cloud Platform. / URL: https://cloud.google.com/ (дата звернення: 16.03.2023).

9. Amazon Web Services. / URL: https://aws.amazon.com/ (дата звернення: 16.03.2023).

10. Jenkins. / URL: https://www.jenkins.io/ (дата звернення: 16.03.2023).

11. Alkilani M., Kobziev V. Enhancing E-government Services by Using Cloud Computing //CEUR Workshop Proceedings. – 2019. – P.66-69.

12. Kravets N., Tseshkovskyi N., Liutova K. Containers and virtual machines in Microsoft Azure //Science, society, education: topical issues and development prospects. Abstracts of the 7th International scientific and practical conference. SPC “Sciconf. Com. Ua”. Kharkiv, Ukraine. – 2020. – P.278-281.

13. Ridvan S., Emine D. Application Development with Service Oriented Architecture. International Journal of Applied Mathematics Electronics and Computers. – 2019. – P.65-69.

14. Faten H. An Overview of Service Composition in Service Oriented Architecture. Modern Applied Science. – 2018. – P.172-179.

15. Тоттен Р. Основи Nginx: крок за кроком до володіння базовими можливостями Nginx в реальних додатках. Видавництво. Пакт.- 2019.- С.194.

16. Thomas, S. M., & Riggs, S. PostgreSQL 12 High Availability Cookbook: Over 100 recipes to design a highly available server with the advanced features of PostgreSQL. Packt Publishing, – 2020. – P.462.

17. Ж. Вуд, Ю. Бернабеу, Т. Грудінськи, Т. Мейер, К. Хеллер. Enterprise SOA. Prentice Hall. – 2005. – С.416.

18. Amazon Elastic Compute Cloud (EC2). / URL: https://info-savvy.com/introduction-of-amazon-elastic-compute-cloud-ec2/ (дата звернення: 27.04.2023).

19. Virtual Machine Scale Set Architecture. / URL: https://www.rupeshtiwari.com/what-is-azure-virtual-machine-scale-sets/ (дата звернення: 27.04.2023).

20. Compute Engine. / URL: https://cloud.google.com/compute (дата звернення: 27.04.2023).

21. A Beginner’s Guide to Understanding and Building Docker Images. / URL: https://jfrog.com/knowledge-base/a-beginners-guide-to-understanding-and-building-docker-images/ (дата звернення: 27.04.2023).

22. CI/CD. / URL: https://www.synopsys.com/glossary/what-is-cicd.html (дата звернення: 27.04.2023).

23. GraphQL Architecture. / URL: https://www.javatpoint.com/graphql-architecture (дата звернення: 27.04.2023).

24. Eiei T., Than N. A. Developing mobile application framework by using RESTful web service with JSON parser. Advances in Intelligent Systems and Computing. – 2015. – P.177-184.

25. Choose an Azure compute service. / URL: https://learn.microsoft.com/uk-ua/azure/architecture/guide/technology-choices/compute-decision-tree (дата звернення: 27.04.2023).

26. Vinaj R., Ravichandra S., Evaluation of SOA-Based Web Services and Microservices Architecture Using Complexity Metrics. SN Computer Science. – 2021. – P.1-10.

27. Naghmh N., Waidah I., Imran G., Behzad N., Mahadi B., Ab Razak Bin Che H. Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation. Information Systems. – 2020. – P.101491.

28. Building microservices in Go with Fiber . / URL: https://blog.logrocket.com/building-microservices-go-fiber/ (дата звернення: 01.05.2023)

29. An introduction to the Flux architectural pattern. / URL: https://www.freecodecamp.org/news/an-introduction-to-the-flux-architectural-pattern-674ea74775c9/ (дата звернення: 01.05.2023)

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

11. Alkilani M., Kobziev V. Enhancing E-government Services by Using Cloud Computing //CEUR Workshop Proceedings. – 2019. – P. 66-69.

12. Kravets N., Tseshkovskyi N., Liutova K. Containers and virtual machines in Microsoft Azure //Science, society, education: topical issues and development prospects. Abstracts of the 7th International scientific and practical conference. SPC “Sciconf. Com. ua”. Kharkiv, Ukraine. – 2020. – P. 278-281.