Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

комп'ютерно-інформаційних систем та програмної інженерії
(повна назва факультету)
комп'ютерних систем та мереж
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Файлове хмарне сховище на базі кластера Raspberry PI із застосуванням мікросервісної архітектури

Виконав(ла): студент(ка) 4 курсу, групи СІ-42
спеціальності 123 Комп'ютерна інженерія
(шифр і назва спеціальності)

(підпис) Сотник А.П. (прізвище та ініціали)

Керівник (підпис) Шингера Н.Я. (прізвище та ініціали)

Нормоконтроль (підпис) Тиш Є.В. (прізвище та ініціали)

Завідувач кафедри (підпис) Осухівська Г.М. (прізвище та ініціали)

Рецензент (підпис) Мудрик І.Я. (прізвище та ініціали)

Тернопіль
2024

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем та програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних систем та мереж
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Осухівська Г.М.
(підпис) (прізвище та ініціали)
« » 20___ р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 123 Комп'ютерна інженерія
(шифр і назва спеціальності)

студенту Сотнику Андрію Петровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Файлове хмарне сховище на базі кластера Raspberry PI із застосуванням мікросервісної архітектури

Керівник роботи Шингера Наталія Ярославівна, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «24» квітня 2024 року № 4/7-408

2. Термін подання студентом завершеної роботи 24 червня 2024 року

3. Вихідні дані до роботи вихідні дані: специфікація мікрокомп'ютера Raspberry Pi; документація по взаємодії з Kubernetes; особливості створення кластеру з балансуванням. навантаження; архітектура система клієнт-сервер; тип програмного забезпечення: Web-додаток

4. Зміст роботи (перелік питань, які потрібно розробити) 1. Аналіз технічного завдання: аналіз вимог до комп'ютерної системи; аналіз можливих рішень поставленого завдання. 2. Проектна частина: розробка загальної структури системи; обґрунтування вибору апаратного. забезпечення; обгрунтвання вибору програмного забезпечення; проектування програмної системи. 3. Практична частина: моделювання апаратної частини; розробка програмного забезпечення. 4. Обгрунтування роботи з точки зору безпеки життєдіяльності та охорони праці.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Мікросервісна архітектура. 2. Архітектура Kubernetes. 3. Діаграма варіантів використання. 4. Структура клієнтського додатку. 5. Дизайн API на рівні всієї системи.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
БЖД, основи ОП	Пилипець М.І., д.т.н., професор кафедри ТМ		

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Розробка і затвердження технічного завдання	01.02.24-09.02.2024	
2	Аналіз технічного завдання	05.02.24-11.02.24	
3.	Розробка загальної структури системи; обґрунтування вибору апаратного забезпечення; обгрунтування вибору програмного забезпечення	03.06.24-06.06.24	
4.	Проектування програмної системи	07.06.24-10.06.24	
5.	Виконання практичної частини	11.06.24-18.06.24	
6.	Виконання розділу «Безпека життєдіяльності та охорона праці.	18.06.24-19.06.24	
7.	Оформлення кваліфікаційної роботи	20.06.24-21.06.24	
9.	Попередній захист кваліфікаційної роботи	14.06.24	
10.	Захист кваліфікаційної роботи	24.06.24-28.06.24	

Студент _____ Сотник А.П.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Шингера Н.Я.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Файлове хмарне сховище на базі кластера Raspberry PI із застосуванням мікросервісної архітектури // Кваліфікаційна робота на здобуття освітнього рівня бакалавр // Сотник Андрій Петрович // ТНТУ, спеціальність 123 Комп'ютерна інженерія // Тернопіль, 2024 // с. - 71, рис. - 34, табл. - 7, аркушів А1 – 5, бібліогр. – 12.

Ключові слова: файлове хмарне сховище, кластер, Raspberry PI, мікросервісна архітектура.

У результаті виконання роботи було розроблене файлове сховище на базі кластеру Raspberry Pi з використанням мікросервісної архітектури. Основна мета розробки – створення надійної, масштабованої та ефективної системи для зберігання та перегляду файлів, яка може бути легко розширена та модифікована відповідно до змінних вимог.

Забезпечення узгодженості даних та надійності комунікації між мікросервісами було вирішено завдяки впровадженню механізмів обробки помилок та повторного виконання запитів, а також використанню брокера повідомлень RabbitMQ для асинхронної обробки завдань.

Завдяки використанню ефективних алгоритмів збереження та обробки даних, а також налаштуванню параметрів кластеру, вдалося досягти високих показників продуктивності при мінімальних витратах на обладнання.

Результатом виконання роботи стало створення надійної, масштабованої та ефективної системи, яка може бути легко адаптована до змінних вимог та умов експлуатації. Цей підхід відкриває широкі можливості для подальшого розвитку та інтеграції з іншими системами та сервісами, забезпечуючи високу гнучкість та адаптивність.

ABSTRACT

File cloud storage based on a Raspberry PI cluster using a microservice architecture // Кваліфікаційна робота на здобуття освітнього рівня бакалавр // Sotnyk Andrii Petrovych // TNTU, specialty 123 Computer engineering // Ternopil, 2024 // p. - 71, fig. - 34, tab. - 7, posters A1 – 5, refs. – 12.

Keywords: cloud file storage, cluster, Raspberry PI, microservices architecture.

After completing the qualification work, a microservice application for cloud file storage based on Raspberry Pi was developed. The primary objective of this project was to create a reliable, scalable, and efficient system for storing and manipulating files, designed to be easily extended and modified in response to evolving requirements.

Ensuring data consistency and communication reliability between microservices was achieved by implementing error handling mechanisms, retrying requests, and utilizing the RabbitMQ message broker for asynchronous task processing.

By employing efficient data storage and processing algorithms, as well as tuning cluster parameters, we achieved high performance metrics with minimal hardware costs.

The outcome of the project was the creation of a reliable, scalable, and efficient system that can easily adapt to changing requirements and operational conditions. This approach opens up broad possibilities for further development and integration with other systems and services, ensuring high flexibility and adaptability.

З М І С Т

ВСТУП

Сучасні технології зберігання даних та обробки інформації продовжують стрімко розвиватися, вимагаючи нових підходів для задоволення зростаючих потреб у масштабованості, надійності та ефективності. Одним із найперспективніших рішень у цій сфері є хмарні сховища, які надають можливість централізованого зберігання та обробки даних з доступом через Інтернет. У той же час, мікросервісна архітектура забезпечує гнучкість і модульність, дозволяючи розробляти складні системи, що легко масштабуються та адаптуються до нових вимог.

Ця робота присвячена розробці хмарного сховища на базі кластера Raspberry Pi з використанням мікросервісної архітектури. Raspberry Pi є доступним і популярним обчислювальним пристроєм, який відзначається низькою вартістю, низьким енергоспоживанням і достатньою продуктивністю для виконання багатьох задач. Використання Raspberry Pi для побудови кластера відкриває нові можливості для дослідження і розробки ефективних, економічно вигідних рішень для зберігання та обробки даних.

У цій роботі буде розглянуто концепцію побудови хмарного сховища з використанням мікросервісної архітектури, яка передбачає поділ системи на незалежні сервіси, що взаємодіють між собою через чітко визначені інтерфейси. Такий підхід дозволяє зменшити складність системи, покращити її масштабованість і підвищити надійність шляхом ізоляції відмов окремих компонентів.

Основні завдання, що ставляться в рамках цієї роботи, включають розробку архітектури системи, налаштування та конфігурацію кластера Raspberry Pi, створення та розгортання мікросервісів для управління даними, а також забезпечення їх взаємодії. Особлива увага приділяється питанням безпеки, резервного копіювання та відновлення даних.

РОЗДІЛ 1. АНАЛІЗ ТЕХНІЧНОГО ЗАВДАННЯ

1.1 Аналіз вимог до комп’ютерної системи

1.1.1 Огляд мікросервісної архітектури

Перш ніж перейти до аналізу вимог щодо розробки хмарного файлового сховища, потрібно проаналізувати переваги та недоліки використання мікросервісної архітектури, а також особливості використання мікрокомп’ютерного кластеру для розгортання програмної системи на його базі.

Мікросервісна архітектура – це підхід до розробки мережевого програмного забезпечення, де додаток складається з окремих незалежних сервісів, кожен з яких відповідає за конкретну функціональність. Кожен мікросервіс розробляється, тестується, розгортається та масштабується окремо від інших, що дозволяє збільшити гнучкість та спрощує обслуговування системи.

Комунікація між мікросервісами може бути реалізована різними способами в залежності від потреб системи. В загальному, існує поділ на 2 типи комунікації: синхронна та асинхронна.

Для синхронної комунікації, зазвичай, використовують текстові формати на основі HTTP або технологію RPC (Remote Procedure Call). У першому випадку, поширеним є використання форматів комунікації основаних на форматі JSON, іноді – XML. RPC технології можуть бути як текстовим (наприклад, JSON-RPC) та і двійковими.

Асинхронна комунікація передбачає наявність стороннього сервісу, який забезпечує реалізацію черги повідомлень, такого як Rabbit MQ або Apacha Kafka. Таким чином, мікросервіси можуть надсилати один одному дані не очікуючи отримання відповіді.

В порівнянні з традиційною монолітною архітектурою, яка передбачає наявність одного великого додатку, який може складатись з різних модулів, такий підхід має свої переваги та недоліки.

Основні переваги полягають в наступному:

- технологічна різнорідність: оскільки кожен з мікросервісів є окремим додатком, які взаємодіють між собою засобами відкритих протоколів, немає технічних обмежень стосовно використання технологій для реалізації сервісу. Оскільки існують технології, які підходять для реалізації певних задач краще за інші – це створює більший простір для прийняття рішень при проектуванні системи. Також це полегшує процес міграції на нові версії та впровадження нових технологій, оскільки немає необхідності змінювати всю систему разом;
- підвищена стійкість: в монолітних системах поширені ситуації, коли помилка в одному з модулів призводить до відмови всієї системи. Оскільки при впроваджені мікросервісної архітектури кожен сервіс є незалежним, при виникненні помилки відмовляє тільки частина системи, а не все одразу;
- масштабування: будь-який додаток завжди має як критичні у відношенні продуктивності модулі, так і ті, які не створюють значного навантаження. У випадку монолітної архітектури, коли повстає потреба масштабування, як правило потрібно робити репліку всього додатку, що потребує пропорційного збільшення серверних ресурсів. При використанні мікросервісної архітектури, достатньо масштабувати тільки той сервіс, який має проблеми з продуктивністю.

Однак також існує ряд недоліків, які потрібно брати до уваги обираючи мікросервісну архітектуру:

- складність управління: використання цього архітектурного підходу передбачає складнішу мережеву інфраструктуру, що накладає додаткові вимоги на логування та моніторинг;
- безпека: реалізація безпечної комунікації в розподіленій системі в загальному складніша, за рахунок того що потрібно передбачити різні рівні захисту для кожного з сервісів які можуть бути доступними з зовні мережі;
- ускладнене тестування: взаємопов'язані сервіси є більш складними у тестуванні, особливо це стосується інтеграційного та інших автоматизованих методів тестування;

1.1.2 Огляд концепції кластеру

Кластер – це група комп'ютерів, що працюють як єдина система. Застосування кластерів для розгортання системи є логічним кроком, враховуючи вибір мікросервісної архітектури.

Зазвичай, при створенні кластеру для розміщення мережевого додатку, можна розглядати дві опції: використання готових рішень від сторонніх провайдерів на базі хмарних віртуальних машин або використання кількох реальних серверів, об'єднаних у кластер.

Перший спосіб є простішим з точки зору керування системою, оскільки провайдери часто надають додаткові інструменти взаємодії з кластером, однак серед його недоліків є висока вартість та обмежений контроль над орендованим обладнанням. Створення кластеру на базі реального апаратного забезпечення усуває проблему неповного контролю, однак проблема вартості зберігається, а також додається необхідність в обслуговуванні серверів та мережевого обладнання.

Використання одноплатних мікрокомп'ютерів можна вважати компромісним рішенням: вони не дорогі та не потребують обслуговування. Недоліком є невелика обчислювальна потужність, однак кількох таких

пристроїв, об’єднаних разом, може бути достатньо для розгортання проектів не великих масштабів.

Одним з представників таких комп’ютерів є Raspberry Pi.

Raspberry Pi – це серія одноплатних комп’ютерів на базі ARM архітектури. В залежності від моделі, пристрій може бути обладнаний різними процесорами (від 1 до 4 ядер, з тактовою частотою до 1.5 ГГц), оперативною пам’яттю LPDDR4 від 1 до 8 ГБ, а також високошвидкісним Ethernet адаптером. Все це, а також хороші показники енергоефективності, роблять даний пристрій достатньо перспективним для побудови кластеру на його базі.

1.1.3 Аналіз вимог

При розробці такої системи як файлове сховище, основними не функціональними вимогами є безпека та надійність. Безпековий аспект полягає в реалізації надійного алгоритму авторизації та автентифікації, таким чином, щоб кожен користувач мав доступ виключно до тих файлів, які йому належать, або до тих, до яких йому було надано доступ іншими користувачами. Це включає впровадження шифрування даних, якими клієнт обмінюється з сервером на рівні протоколу HTTP, а також вибору механізму автентифікації який був би стійким до спроб втручання у процес. Також важливим аспектом безпеки є валідування усіх вхідних даних.

Надійність такої системи буде полягати у механізмах, які забезпечують зберігання файлу навіть у випадку відмови системи або виходу її з ладу. Оскільки будь-який тип накопичувача, особливо при інтенсивному використанні, може вийти з ладу частково або повністю, потрібно це наперед передбачити. Для того щоб забезпечити цілісність файлів, можна використовувати RAID-масиви, які будуть дзеркально записувати файли на кілька накопичувачів або багаторівневі системи резервного копіювання, для прикладу: мати окремі накопичувачі з резервними копіями та додатково зберігати копії на іншому сервері у хмарі в зашифрованому вигляді. Також,

додатковим засобом надійності може бути розподілений запис файлів між кількома накопичувачами, таким чином, якщо один з них вийде з ладу, втрачено буде тільки частина даних, яку можна буде відносно швидко відновити з копії.

Також важливою вимогою є можливість масштабування з метою додавання нових накопичувачів або засобів резервного копіювання.

Серед функціональних вимог системи можна зазначити наступні:

- авторизація та автентифікація: користувач повинен мати можливість зареєструватись, авторизуватись та підтвердити право на доступ до тих чи інших файлів;
- завантаження та скачування файлів: користувач повинен мати можливість завантажувати свої файли в рамках діючих для нього обмежень у обсязі, а також скачувати файли, що знаходяться у його власності без обмежень;
- управління файлами: користувач повинен мати можливість видаляти, переміщувати, перейменовувати файли у його власності;
- спільний доступ до файлів: користувач повинен мати можливість ділитись своїми файлами з іншими користувачами або шляхом генерації унікального посилання;
- сповіщення: користувач має отримувати сповіщення від системи засобами електронної пошти;
- адміністративний розділ: система повинна мати поділ на користувачів та адміністраторів;

Нефункціональні вимоги полягають у наступному:

- максимальний ліміт швидкості, на якій система повинна оперувати, не повинен перевищувати 1 Гбіт/с.;
- мінімальний ліміт швидкості, на якій система повинна оперувати, не повинен перевищувати 100 Мбіт/с.;

- час відгуку системи повинен складати менше 400 мс. при запиті сторінки; час відповіді при запиту файлу не повинен перевищувати 1с на файл розміром 10 МБ.;
- доступність: система повинна використовувати реплікацію на рівні мікросервісів та автоматично перезапускатись у разі відмови;
- шифрування: трафік між клієнтом та системою повинен бути зашифрований відповідно до протоколу HTTPS; резервні копії файлів повинні зберігатись у зашифрованому вигляді;

1.2 Аналіз можливих рішень поставленого завдання

При прийнятті рішення про спосіб реалізації завдання, доречно буде розпочати аналіз з апаратного забезпечення.

В загальному, існує кілька типів кластерів, які можна було б застосувати для вирішення поставленого завдання. Обчислювальний кластер – це тип кластеру, який дозволяє об’єднати ресурси всіх підключених до нього комп’ютерів та спрямувати їх для вирішення спільного завдання. Такий спосіб об’єднання часто зустрічається для кластерів на базі Raspberry Pi, однак для розгортання мікросервісів обчислювальний кластер не надає особливих переваг окрім збільшення продуктивності, але і це тільки за умови підтримки розподіленої роботи на рівні окремих сервісів. Для того, щоб скористатись перевагами обчислювального кластеру потрібно або розробляти окремі сервіси з врахуванням підтримки паралельної роботи, що суттєво ускладнить процес розробки, або встановлювати та налаштовувати додаткові інструменти для масштабування таким чином, щоб кілька реплік сервісу могли працювати у різних процесах.

Іншим типом кластеру є кластер спрямований на балансування навантаження. Така система використовується для розподілу вхідного трафіку між різними вузлами з метою оптимізації використання ресурсів. З точки зору мікросервісної архітектури, такі кластери є найкращим вибором,

оскільки програмне забезпечення, яке використовується для їхньої організації та налаштування, як правило, вже підтримує все необхідне для керування мікросервісним додатком.

Кластер складається з кількох пристроїв та програмного забезпечення яке використовується для встановлення зв’язку між ними. В залежності від обраного типу кластеру потрібно використовувати різне програмне забезпечення для його налаштування.

При використанні кластеру з балансуванням навантаження, таким програмним забезпеченням можуть бути такі рішення як Docker Swarm, Kubernetes, Apache Mesos та інші. Кожна з цих технологій має свої переваги та недоліки, однак найбільш розповсюдженою, а отже найбільш документованою, з них є Kubernetes.

Kubernetes - це система оркестрації контейнерів, розроблена для автоматизації розгортання, масштабування та управління контейнеризованими додатками. Вона призначена для роботи з кластерами різних розмірів та дозволяє легко додавати нові та видаляти існуючи вузли. Великою перевагою використання даної технології є те, що вона автоматично вирішує багато проблем які виникають при розробці мікросервісного рішення, а саме: вбудована можливість реплікації будь-якого з додатків, відслідковування статусу додатку та його перезапуск у разі відмови, автоматичне налаштування мережі та мережевої взаємодії між контейнерами.

У разі використання Kubernetes, до додатку висуваються певні вимоги, а саме необхідність у контейнеризації. Контейнером називають автономне, ізольоване оточення в межах якого працює програмне забезпечення. Перевагою використання контейнерів є можливість отримати ідентичну поведінку додатку у різних оточеннях, а також, при правильній контейнеризації, відсутність потреби в додаткових налаштування оточення для запуску того чи іншого програмного забезпечення. Для створення та керування контейнерами використовується спеціальне програмне забезпечення, таке як Docker або Podman.

При визначенні стратегії зберігання файлів у системі потрібно брати до уваги надійність та швидкість обраного типу накопичувачів, а також можливість масштабування шляхом додавання нових накопичувачів.

На поточному етапі розвитку технологій можна розглядати накопичувачі двох типів: механічно-магнітні (HDD) та твердотільні (SSD). Основні переваги магнітних дисків полягають у низькій ціні та вищій надійності, однак вони програють твердотільним накопичувачам у швидкості запису та зчитування. Особливим чином цей недолік проявляється при роботі з непослідовно-розміщеними дрібними файлами, що передбачається частим випадком в контексті хмарного сховища, яке, для оптимізації дискового простору, може зберігати файли одного користувача зовсім у різних місцях або навіть на різних дисках.

Оптимальним рішенням було б комбінувати використання обох типів накопичувачів. Використовувати SSD диски в якості основного сховища, з яким безпосередньо взаємодіє користувач, а HDD, об’єднані у RAID масиви, використовувати як засоби підвищеної надійності для зберігання резервних копій.

Оскільки при використанні Kubernetes складно передбачити на якому саме з вузлів буде запущено той чи інший сервіс, накопичувачі доречно робити доступними для всіх елементів кластеру через мережу. Цього можна досягнути використанням такого інструменту як samba, який дозволяє створювати мережеві директорії, таким чином накопичувачі можна буде підключати до довільного вузла системи.

При розробці програмного забезпечення потрібно брати до уваги обмеження та особливості апаратної платформи. Використання великих платформ, таких як Java або .Net Core, було б доцільно за наявності більшої кількості обчислювальних ресурсів або при розробці монолітного додатку, однак потреба у забезпеченні роботи виконавчого середовища (JVM або CLR) для кожного з мікросервісів створила б надмірне навантаження на Raspberry Pi. Сценарні мови програмування, такі як PHP, JavaScript або

Python не завжди є оптимальними з точку зору продуктивності, а відсутність сильної системи типів збільшує ймовірність виникнення помилок.

Для мікросервісного додатку, що призначений для роботи на обмеженій з точки зору ресурсів платформі, оптимальним вибором технологій можна вважати такі мови, які компілюються безпосередньо в машинний код, не мають додаткових залежностей у вигляді інтерпретатора або виконавчого середовища та використовують не більше обчислювальних ресурсів ніж потрібно для роботи самого мікросервіса. Таким чином, оскільки кожен такий процес потребує небагато ресурсів для свого функціонування, засобами Kubernetes можна балансувати навантаження між кількома одночасно працюючими репліками сервісу.

Мовами, які відповідають особливостям даної апаратної платформи, можна вважати Go, Rust, V, C++.

РОЗДІЛ 2. ПРОЕКТНА ЧАСТИНА

2.1 Розробка загальної структури системи

При розробці такого роду системи, з структурної точки зору можна окремо виділити фізичну структуру, яка передбачає конфігурацію мережі, кластеру, накопичувачів та іншого апаратного забезпечення, а також логічну, яка полягає у визначенні мікросервісів та встановленню зв'язків між ними.

З точки зору апаратного забезпечення, система буде складатись з кількох компонентів:

- мережевий маршрутизатор, який створить локальну мережу для вузлів кластеру а також буде забезпечувати керування доступом до системи з зовнішніх мереж;
- мікрокомп'ютери Raspberry Pi 4 у кількості 3шт., які разом утворюватимуть Kubernetes кластер;
- твердотільні накопичувачі, сумарним об'ємом не менш ніж 500 ГБ.;
- HDD накопичувач або RAID масив з кількох таких накопичувачів, сумарним об'ємом не меншим ніж об'єм твердотільних дисків;

Відповідно до вимог, максимальна швидкість, яку стабільно повинна забезпечувати мережа становить 1 ГБіт/с., а отже маршрутизатор повинен підтримувати стандарт GigE, а для з'єднання пристроїв у мережі повинен бути використаний кабель типу «вита пара», не менше ніж четвертої категорії у 4-парній конфігурації.

Загальну структуру апаратної частини у схематичному вигляді продемонстровано на рисунку 2.1.

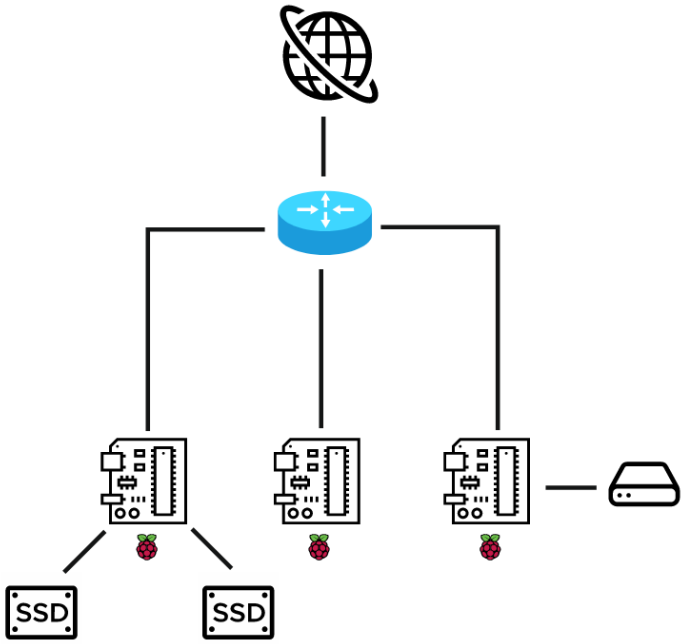

Рисунок 2.1 - Апаратна структура хмарного сховища

Для проектування програмної архітектури на базі мікросервісів, потрібно для початку виділити функціональність системи в окремі групи, кожну з яких в подальшому можна буде винести у окремий мікросервіс. На основі вимог, можна визначити наступні групи:

- сервіс керування користувачами, що включає в себе реєстрацію, авторизацію та аутентифікацію користувачів. До його складу також входить база даних, в якій будуть зберігатись зареєстровані користувачі;
- сервіс керування файлами, який надає можливість завантажувати, скачувати та редагувати файли. Цей сервіс відповідальний за роботу з основними накопичувачами, відслідковуванням користувацьких лімітів на використання дискового простору, оптимізацію використання дисків. Для його роботи також потрібна база даних, в якій буде зберігатись метаінформація про завантажені користувачами файли;
- API-шлюз, який буде виступати єдиною точкою комунікації клієнта та сервера. Цей сервіс відповідальний за реалізацію передавання

клієнтського запиту до відповідного сервісу всередині мережі, агрегацію запитів до різних сервісів, реалізацією захисту та аутентифікації;

- брокер повідомлень, що відповідальний за асинхронну комунікацію між мікросервісами. Побудований на основі RabbitMQ і додаткового драйвера, відповідального за публікацію повідомлень;
- сервіс повідомлень, що здійснює надсилання електронних листів користувачам;
- сервіс логування, який збирає записи про події в межах всієї системи;
- мовний сервіс, який реалізує можливість локалізації тексту;
- клієнтський додаток у вигляді SPA (Single Page Application), який надає можливість взаємодії з сховищем через браузер;

Така архітектурна модель передбачає, що клієнтський додаток буде взаємодіяти тільки з одним із сервісів, який спеціально для цього призначений, замість того, щоб мати можливість надсилати окремі запити до будь-якого з мікросервісів у системі. Такий спеціально виділений сервіс називається API-шлюзом.

Цей підхід вирішує одразу дві проблеми: в першу чергу проблема безпеки, яка була зазначена як один з недоліків мікросервісної архітектури. Замість того, щоб реалізовувати аутентифікацію на рівні кожного з окремих мікросервісів, можна зробити це тільки на рівні шлюзу, оскільки жоден запит не може пройти повз нього.

Інший недолік мікросервісів – це низька транзакційність операцій. Можуть існувати операції, для завершення яких потрібно зробити кілька запитів до різних мікросервісів. У випадку, коли архітектура розроблена без врахування API-шлюзу, контроль за послідовністю і результатами запитів полягає на клієнтський додаток, що збільшує складність розробки і може призводити до непередбачуваних помилок. API-шлюз, в свою чергу, дозволяє інкапсулювати такі операції таким чином, щоб для клієнтського додатку це

виглядало як один атомарний запит, а логіка агрегації запитів та верифікації їхніх результатів може бути реалізована на рівні API шлюзу.

Структурна схема програмної архітектури представлена на рисунку 2.2.

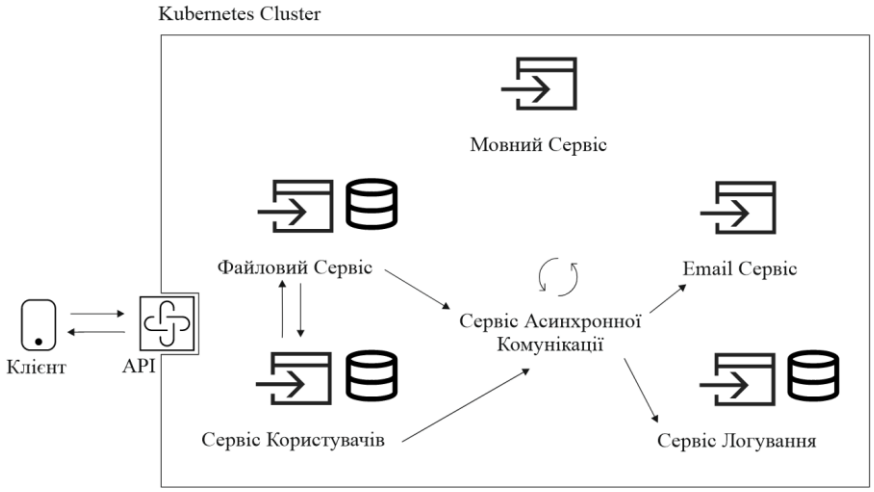

Рисунок 2.2 - Мікросервісна архітектура

2.2 Обґрунтування вибору апаратного забезпечення

Існує багато модифікацій мікрокомп’ютерів сімейства Raspberry Pi, тому вибір версії потрібно робити виходячи з вимог до системи. Всього існує 5 версії пристрою, кожна з яких має по дві модифікації: А та Б. Версії 1 та 2 не підлягають розгляду за рахунок застарілого оснащення. Версія 3 має достатній обчислювальний потенціал, однак недоліком є малий об’єм оперативної пам’яті та відсутність USB версії 3, що виключає можливість підключення швидкісних накопичувачів за цим протоколом. Версія 5 є найновішою і найбільш продуктивною, однак враховуючи налаштування кластеру, продуктивність однієї одиниці не так важлива як продуктивність всього кластеру, а також її вартість, використання останнього покоління Raspberry Pi є недоцільним для вирішення поставленого завдання. Результатом аналізу став вибір мікрокомп’ютера Raspberry Pi 4 (Рисунок 2.3) версії в модифікації Б з 2 ГБ оперативної пам’яті.

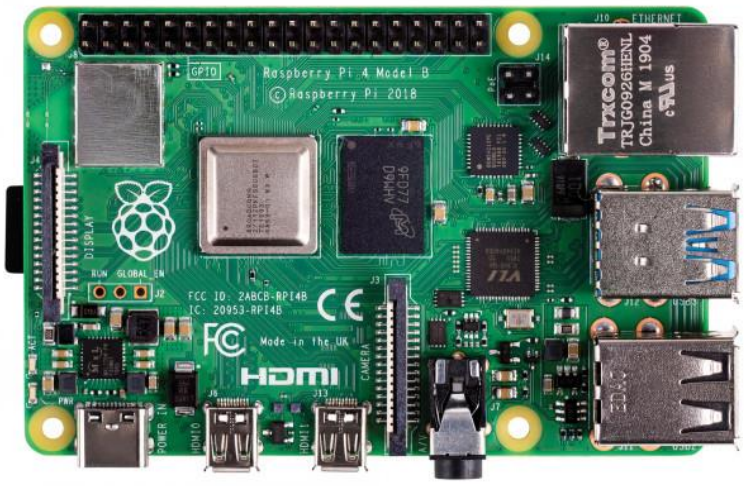

Рисунок 2.3 - Одноплатний комп’ютер Raspberry Pi 4B

Основними перевагами цієї моделі в порівнянні з попередніми є достатня кількість оперативної пам’яті, підтримка USB 3.0, що дозволить підключати нові накопичувачі без потреби вимикати при цьому пристрій, 64-бітний, 4 ядерний процесор на базі ARM архітектури з тактовою частотою 1.5 ГГц.

В якості мережевого маршрутизатора було обрано модель MikroTik hEX RB750Gr3 (Рисунок 2.4). Це напівпрофесійне мережеве обладнання, яке повністю задовольняє визначені раніше вимоги: швидкість обміну даними до 1000 Мбіт/с., наявність між мережевого екрану та різноманітні засоби захисту.

Рисунок 2.4 - Маршрутизатор MikroTik hEX RB750Gr

При виборі твердотільних накопичувачів увага приділялась таким параметрам як швидкість роботи при зчитування непослідовних файлів а також показник TBW (Terabytes To Write, гарантований ресурс накопичувача). На основі цих параметрів було обрано серверний накопичувач SK Hynix BC711 Pyrite (Рисунок 2.5) на 256 ГБ. Ресурс TBW для даного диску складає 400 терабайт, а максимальна швидкість зчитування файлів перевищує можливості інтерфейсів Raspberry Pi 4.

Рисунок 2.5 - SK Hynix BC711 Pyrite

Для підключення такого типу накопичувачів до Raspberry Pi 4 використовуються спеціальні док-станції, які виступають в ролі конвертерів між протоколами USB та PCI-E. В рамках реалізації даної роботи було обрано док станцію Maiwo 2*SATA M.2 SSD Key B/B+M USB 3.1 (Рисунок 2.6) за рахунок підтримки одночасного підключення двох накопичувачів.

Рисунок 2.6 - Док станція для M.2 накопичувачів

В якості бази для сховища резервних копій було обрано диск Western Digital Purple (Рисунок 2.7) об’ємом на 1 ТБ. HDD цієї моделі призначені для роботи в умовах постійного навантаження та підвищених температур.

Рисунок 2.7 - HDD Western Digital Purple

Як і у випадку з основними накопичувачами, для того щоб підключити резервний диск теж потрібний відповідний пристрій для підключення 3.5 дюймових дисків засобами USB. Для цього було обрано док-станцію від того ж виробника - Maiwo SSD/HDD 2.5/3.5" (Рисунок 2.8).

Рисунок 2.8 - Док станція Maiwo SSD/HDD 2.5/3.5"

2.3 Обґрунтування вибору програмного забезпечення

В якості програмного забезпечення для організації кластеру було обрано Kubernetes, а саме його оптимізовану для роботи в умовах обмежених обчислювальних ресурсів версію під назвою k3s.

Причина вибору Kubernetes полягає як в його розподіленій архітектурі, яка спрямована на організацію кластерів, так і тому, що Kubernetes, зокрема k3s, пропонує відмінну підтримку для мікросервісних архітектур. Він надає такі можливості, як автоматичне масштабування, відновлення після збоїв, управління трафіком і балансування навантаження.

На найвищому рівні, архітектура Kubernetes (Рисунок 2.9) складається з наступних елементів:

- node (вузел кластеру) – це окрема фізична або віртуальна машина, на яку встановлено Kubernetes і яка є складовою частиною кластеру. Кожен кластер має один контрольний вузол (Control pane), а також певну кількість дочірніх вузлів (Worker nodes). Контрольний вузол відповідає за керування кластером, а дочірні вузли використовуються для обробки вхідного трафіку на запуску контейнеризованих додатків;

- pod – це найменша одиниця розгортання та управління. Він складається з одного або кількох контейнерів, об’єднаних спільним мережевим та файловим простором. Кожна репліка додатку створюється у окремому pod’і;
- контейнер з додатком – розміщується всередині pod’у;

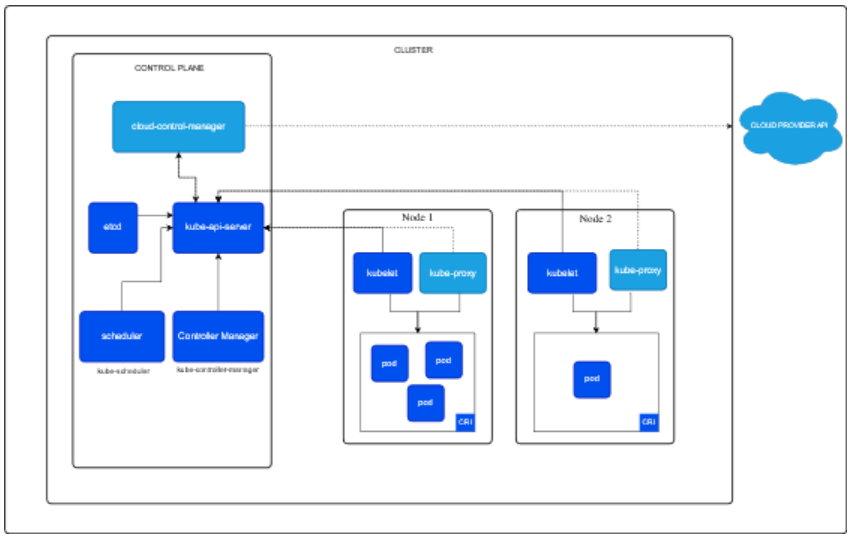

Рисунок 2.9 - Архітектура Kubernetes

Для розгортання в кластері Kubernetes, кожен з мікросервісів повинен бути упакований у свій контейнер. Для створення та застосування контейнерів зазвичай використовують інструмент під назвою Docker.

При виборі мов програмування та технологій для розробки мікросервісів основними критеріями були продуктивність та наявність достатньо розвиненої інфраструктури для вирішення поставленого завдання.

Було обрано наступні мови та технології: для розробки API-шлюзу, поштового сервісу, сервісу логування та в якості обгортки над Rabbit MQ мову програмування Golang та її фреймворк Fiber. Цей вибір було зроблено за рахунок можливостей фреймворку Fiber працювати в ролі зворотного проксі-серверу, а також через наявність зручного клієнту для взаємодії з RabbitMQ.

Для розробки сервісу керування користувачами та сервісу перекладів мову програмування V. Перевагами цієї технології є низький рівень споживання ресурсів та відсутність потреби у встановлення додаткових бібліотек для роботи з HTTP та базами даних.

Для файлового сервісу було вирішено застосувати мову програмування Rust та його фреймворк Warp. Основна перевага цієї мови полягає в безпечній роботі з пам’яттю та розвиненому механізму статичного аналізу, що знижує ймовірність виникнення помилок.

Всі мікросервіси, для яких передбачено використання бази даних, використовують систему керування базами даних PostgreSQL.

Клієнтське програмне забезпечення було вирішено реалізувати у вигляді веб-додатку, розробленого як Single Page Application. Для цієї задачі було обрано технологію Svelte за рахунок швидкості та простоти розробки, а також невисокій кількості абстракцій.

2.4 Проектування програмної системи

2.4.1 Модель варіантів використання хмарного сховища

Діаграма варіантів використання (use case diagram) є графічним представленням функціональності системи з точки зору кінцевого користувача. Вона описує взаємодію між користувачами (акторів) та системою через різні сценарії використання (варіанти використання). Ця діаграма допомагає формалізувати вимоги до системи та побудувати модель, що на високому рівні дозволяє визначити яким чином буде використовуватись система.

Для моделювання варіантів використання хмарного сховища було визначено перелік акторів (табл. 2.1) і використання (табл. 2.2.), після чого було складено модель (Рисунок 2.10).

Таблиця 2.1 – Опис акторів

Назва	Опис
Користувач	Користувач хмарного сховища
Адміністратор	Користувач, який має права адміністратор
Хмарне сховище	Сукупність мікросервісів

Таблиця 2.2 – Опис варіантів використання

Назва	Опис
Реєстрація	Функція створення нового користувача.
Авторизація	Функція входу користувачем у систему.
Завантаження файлу	Користувач має можливість завантажити файл.
Скачування файлу	Користувач має можливість скачати файл.
Пошук файлів	Користувач має можливість здійснювати пошук серед доступних йому файлів.
Копіювання файлу	Користувач має можливість створювати копії свого файлу.
Видалення файлу	Користувач має можливість видаляти свої файли.
Перейменування файлу	Користувач має можливість змінити ім’я своїх файлів.
Поширення файлів	Користувач має можливість поширити файли з іншими.
Підтвердження користувача	Адміністратор має можливість розглядати запити на реєстрацію та підтверджувати користувачів.
Розподіл дискового простору	Адміністратор має можливість визначати доступний для користувача об’єм на диску.

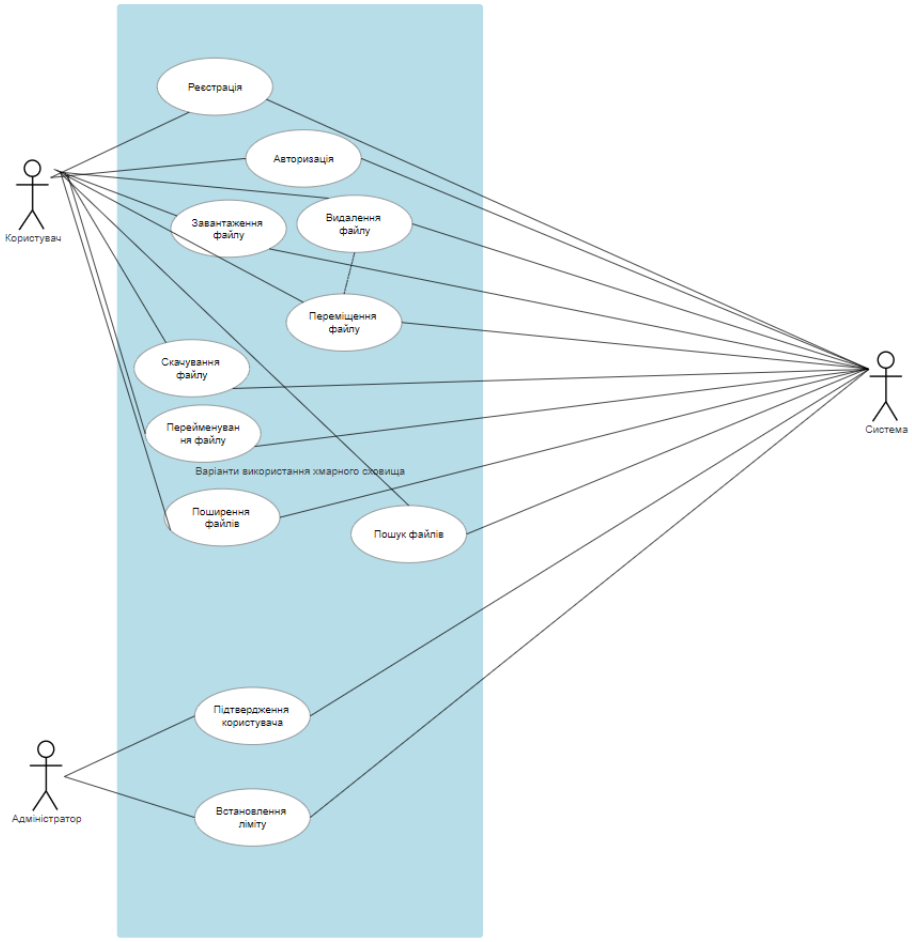

Рисунок 2.10 - Діаграма варіантів використання

2.4.2 Проектування клієнтського додатку

В якості клієнтського додатку було прийнято розробити SPA на базі бібліотеки Svelte. Single Page Application (SPA) - це веб-додаток, який завантажується як єдина веб-сторінка і динамічно оновлює свій контент у відповідь на взаємодії користувача, без необхідності перезавантаження всієї сторінки. Це забезпечує більш швидкий та інтерактивний досвід для користувачів, оскільки лише необхідні частини сторінки оновлюються. SPA зазвичай використовують AJAX для асинхронного завантаження даних з сервера.

Svelte - це сучасний фреймворк для розробки веб-додатків, який відрізняється від традиційних підходів тим, що виконує більшість своєї роботи під час компіляції, а не в браузері. Це означає, що Svelte перетворює компоненти в чистий JavaScript, який працює швидше і займає менше місця.

Однією з головних переваг Svelte є його простота у використанні та здатність створювати високопродуктивні додатки з мінімальним кодом.

При проектуванні клієнтського веб-додатку можна виділити наступні елементи:

- Router (маршрутизатор) – модуль, який відповідає за завантаження різних сторінок в залежності від введеної користувачем URL-адреси;
- Сторінки – компоненти глобального рівня, які завантажуються маршрутизатором та використовуються для композиції функціональних компонентів;
- Компоненти – окремі аспекти UI інтерфейсу та відповідної логіки, потрібної для їхньої роботи, інкапсульовані в окремі файли;
- API клієнт – сукупність засобів для взаємодії з серверною частиною додатку;
- Сховища даних (Stores) – глобально-доступні реактивні об’єкти, які використовуються для поширення спільного стану між компонентами;

На рисунку 2.11 представлена загальна структура та схема взаємодії між окремими компонентами клієнтської частини додатку.

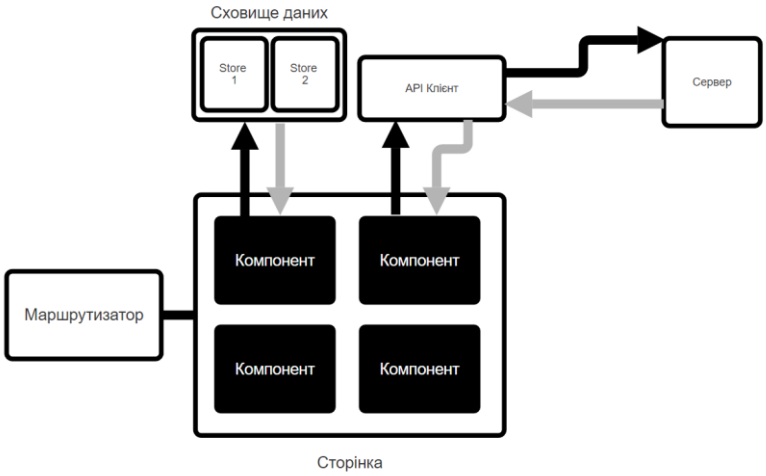

Рисунок 2.11 - Структура клієнтського додатку

Також, для клієнтського додатку потрібно розробити концепт UI/UX дизайну, який буде дотримано в процесі реалізації. Беручи до уваги сучасні тенденції, було вирішено розробити мінімалістичний дизайн у темних відтінках з використанням елементів векторної графіки. Дизайн однієї з сторінок додатку продемонстровано на рисунку 2.12.

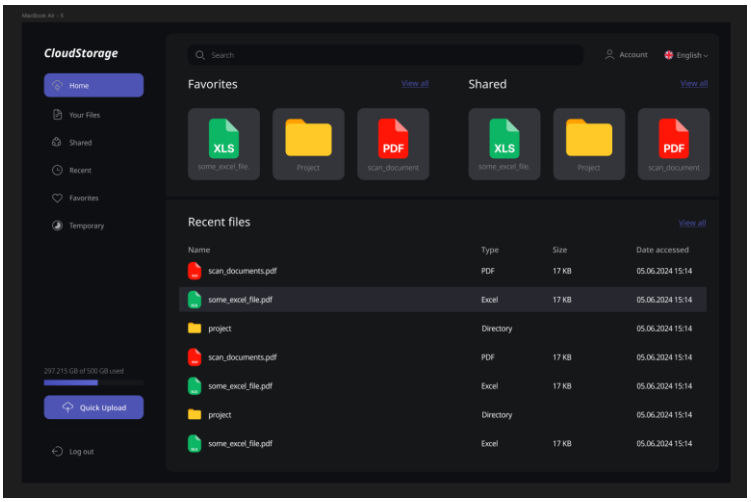

Рисунок 2.12 - Дизайн клієнтської частини

2.4.3 Визначення зв’язків та засобів комунікації між мікросервісами

Для комунікації між мікросервісами, а також між клієнтом на API-шлюзом, було вирішено використовувати REST. REST API (Representational State Transfer Application Programming Interface) — це архітектурний стиль побудови інтерфейсу для взаємодії клієнта і сервера через протокол HTTP. Він використовує стандартні HTTP методи для роботи з ресурсам, які представлені у вигляді URL-адрес.

Основною перевагою REST API є масовість підтримки на рівні різних технологій а також простота реалізації та використання. При застосуванні даного підходу тип дії, як правило, визначають вказанням HTTP методу, а дані, закодовані, як правило, в JSON або XML, передаються у тілі цього запиту.

Для деяких частин систем взаємодія засобами REST є неоптимальною: протокол HTTP є синхронним, а отже кожен запит триває до тих пір, доки не

прийде відповідь від сервера. Необхідність постійного очікування відповіді при між сервісній комунікації може збільшувати загальний час, необхідний для формування відповіді клієнту. Для вирішення цієї проблеми, комунікацію з деякими мікросервісами можна зробити асинхронним, використовуючи технологію обміну повідомленнями. RabbitMQ — це програмне забезпечення для черг повідомлень з відкритим вихідним кодом, яке реалізує протокол AMQP (Advanced Message Queuing Protocol). Воно дозволяє програмам обмінюватися повідомленнями і підтримує асинхронну комунікацію між компонентами програмного забезпечення. Принцип роботи RabbitMQ полягає в наступному: основними сутностями є виробник (producer), який створює та публікує повідомлення, споживач (consumer), який прослуховує чергу на предмет наявності нових повідомлень та є відповідальним за їх обробку. Після створення повідомлення воно надсилається у обмінник (exchange), який, в залежності від типу, передає його у відповідну чергу.

Клієнт буде взаємодіяти зі всіма мікросервісами через єдину точку зв'язку – мікросервіс під назвою API-шлюз. Він буде відповідальним за переадресацію запитів відповідним внутрішнім сервісам та забезпечення рівнів захисту для доступу до того чи іншого ресурсу. Ця комунікація теж буде відбуватись засобами REST API.

Для визначення способів комунікації між клієнтом та серверною частиною додатку, а також між окремими мікросервісами, необхідно визначити інтерфейси для кожного з них, а саме: сервісу користувачів (таблиця 2.3), сервіс перекладів (таблиця 2.4), сервіс файлів (таблиця 2.5), сервіс логування (таблиця 2.6), сервіс електронної пошти (таблиця 2.7). Всі ці ресурси поєднуються між собою API шлюзом

Таблиця 2.3 - API сервісу користувачів

Ресурс	Метод	Опис
/register	POST	Приймає такі дані як: ім’я користувача, прізвище, адресу електронної пошти та пароль. У разі коректності введених даних, створює нового користувача у статусі “неактивний”.
/login	POST	Приймає такі параметри як адресу електронної пошти та пароль. У разі, якщо пароль та електронна пошта введені правильно, а також акаунт користувача активовано, генерує пару JWT токенів для авторизації. Детальніше механізм авторизації буде описано вище.
/approve/:id	PUT	Приймає ідентифікатор користувача в якості параметру та змінює його статус на “активний”
/auth	GET	Зчитує HTTP заголовок під назвою “Authorization” на предмет наявності токену авторизації. Якщо токен присутній, перевіряє його коректність та, у випадку коректності токену, повертає інформацію про поточного користувача.
/refresh	POST	Зчитує HTTP заголовок під назвою Refresh на предмет наявності токену оновлення доступу. Якщо токен присутній та коректний, то відбувається його обмін на нову пару токенів.

Таблиця 2.4 - API сервісу перекладів

Ресурс	Метод	Опис
/get	GET	Повертає всі переклади для всіх мов
/get/:lang	GET	Повертає весь список перекладів для заданої мови.
/get/:lang/:term	GET	Повертає переклад конкретного рядку за ключем.

Таблиця 2.5 - API сервісу файлів

Ресурс	Метод	Опис
/assignQuote	POST	Приймає ідентифікатор користувача та об’єм дискового простору, який повинен бути йому доступний для запису.

Продовження табл. 2.5

/createFile	POST	Приймає метадані про файл: назву, розмір, дату створення, а також ідентифікатор користувача. Якщо файл такого розміру вписується у ліміт, то створює запис у базі даних про новий файл та повертає унікальний ідентифікатор файлу.
/uploadChunk/:uid	POST	Приймає сегмент файлу і його унікальний ідентифікатор, після чого записує сегмент на диск.
/finishUpoad/:uid	PATCH	Приймає ідентифікатор файлу та зберігає запис про те, що його запис завершено.
/delete/:uid	DELETE	Додає файл у чергу на видалення
/copy	POST	Приймає ідентифікатор файлу та шлях, куди потрібно зберегти копію.
/move	POST	Приймає ідентифікатор файлу та шлях, куди потрібно його перемістити
/downloadFile/:uid	GET	Надає файл у придатному для завантаження вигляді.
/zipDirectory	POST	Приймає шлях до директорії та додає її у чергу на архівування
/getFiles	GET	Повертає перелік всіх файлів для поточного користувача.
/getFiles/:path	GET	Повертає перелік всіх файлів для поточного користувача, які знаходяться за вказаним шляхом.
/getFile/:uid	GET	Повертає всю інформацію про запитуваний файл.
/backup	POST	Додає у чергу завдання на створення резервної копії дисків.

Таблиця 2.6 - API сервісу логування

Ресурс	Метод	Опис
/write	POST	Приймає два параметри: тип повідомлення а також саме повідомлення. Додає завдання у чергу для запису у файл.

Таблиця 2.7 - API сервісу Email

Ресурс	Метод	Опис
/send	POST	Приймає дані отримувача: ім’я та адресу електронної пошти, назву шаблону а також дані, якими потрібно заповнити шаблон. Додає завдання у чергу для надсилання електронного листа.

На рисунку 2.13 зображено загальну схему REST API ресурсів у системі. Деякі з них доступні з-зовні завдяки проксі серверу у вигляді API-шлюзу, а деякі повністю або частково доступні для використання тільки всередині мережі.

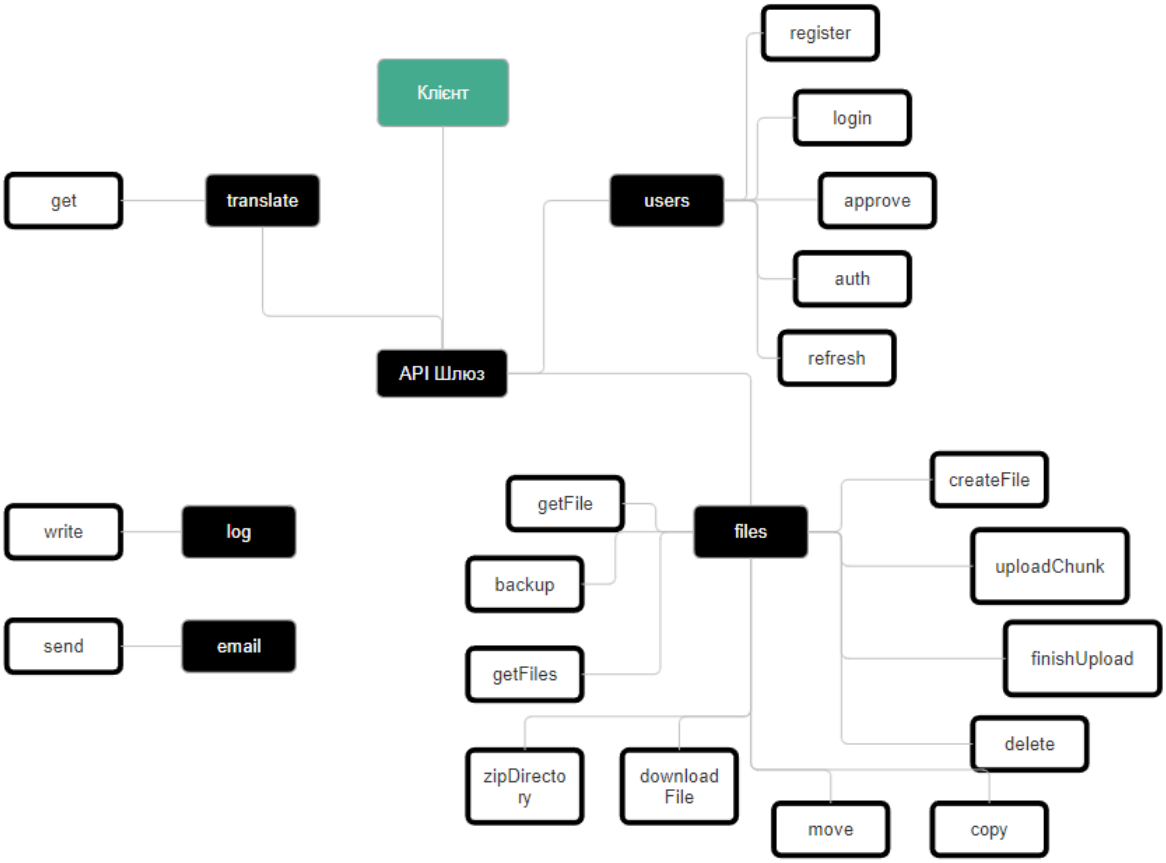

Рисунок 2.13 - Дизайн API на рівні всієї системи

2.4.4 Проектування бази даних

Для мікросервісних додатків характерне використання окремої бази даних для кожного з сервісів. Хоч це і збільшує споживання ресурсів та створює складнощі при потребі виконання складних запитів на пов'язаних таблицях – це дозволяє повністю відокремлювати мікросервіси один від одного і мати можливість безпечно вносити зміни у структуру таблиць без впливу на інші сервіси.

В рамках реалізації хмарного сховища наступні сервіси використовуватимуть бази даних: сервіс користувачів, сервіс файлів та сервіс перекладів.

База даних сервісу користувачів буде складатись з трьох таблиць (рисунок 2.14):

- users – таблиця з списком усіх користувачів;

- refresh_tokens – таблиця, яка містить у собі перелік всіх токенів, які можуть бути використані для обміну на нову пару;
- administrators – таблиця, яка містить записи про те, який користувач є адміністратором;

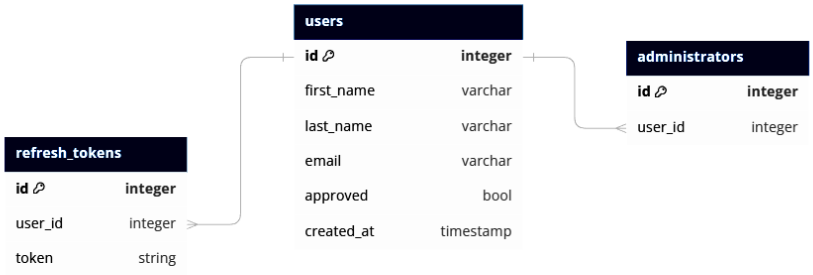

Рисунок 2.14 - Структура бази даних для сервісу користувачів

База даних сервісу перекладів була спроектована для зберігання даних про наявні мови та переклади в кількох нормалізованих таблицях. Цей підхід дозволить гнучко зберігати дані з можливістю додавання підтримки нових мов без потреби внесення змін у структуру таблиць.

Для цієї БД було визначено такі таблиці (рисунок 2.15):

- language – таблиця, у якій будуть зберігатись всі підтримувані мови;
- key – таблиця, у якій будуть зберігатись ключі, до яких будуть прив’язані переклади;
- translation – таблиця з перекладами;

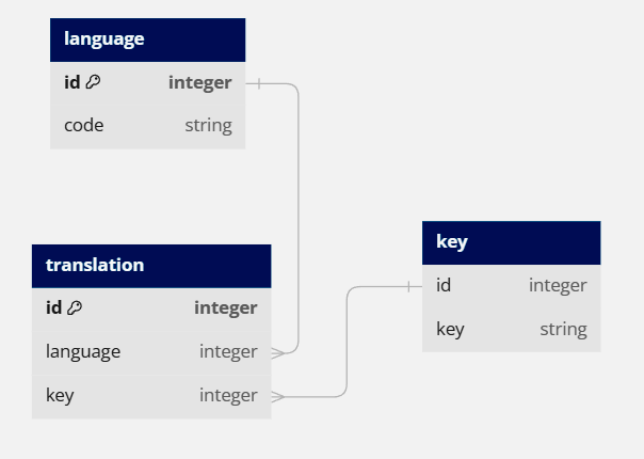

Рисунок 2.15 - Структура бази даних сервісу перекладів

Сервіс керування файлами повинен використовувати базу даних для ведення обліку використаного користувачем дискового простору та збереження коректної файлової ієрархії.

Для бази даних сервісу користувачів було виділено наступні таблиці:

- file – таблиця, в якій зберігаються метадані всіх завантажених файлів;
- directory – таблиця, в якій зберігаються метадані всіх завантажених директорій;
- user_quote – таблиця, в якій зазначено дисковий простір, доступний для використання для користувачів;
- temp_files – таблиця, в якій відслідковуються тимчасові файли, які буде видалено через певний час;
- shared_directory – таблиця, в якій зберігаються дані про те, які директорії були поширені з іншими користувачами;

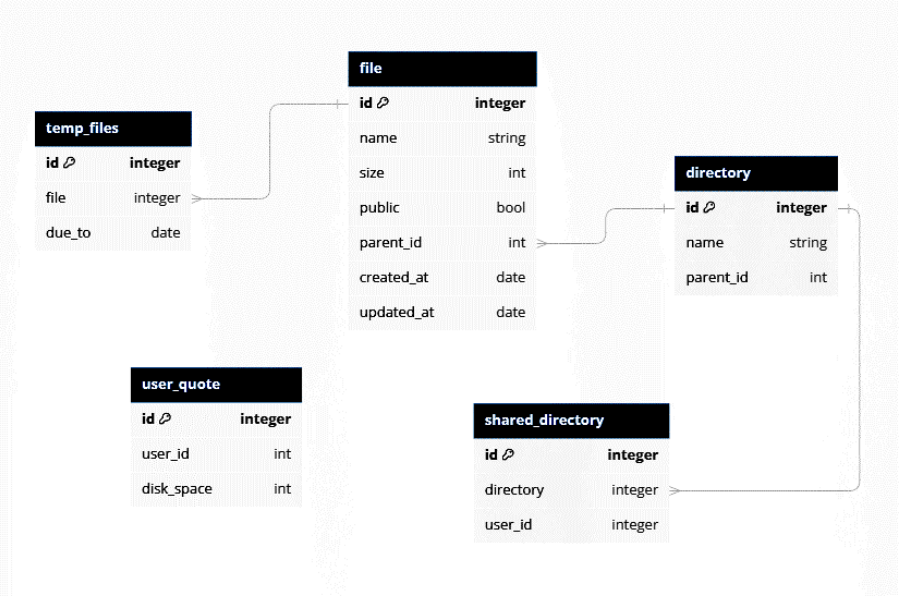

Рисунок 2.16 - Структура бази даних сервісу користувачів

РОЗДІЛ 3 ПРАКТИЧНА ЧАСТИНА

3.1 Моделювання апаратної частини

Для того, щоб створити оточення, наближене до реального, потрібно здійснити моделювання кластеру на базі Raspberry Pi. Цього можна досягнути наступним чином: використовуючи програмне забезпечення для керування віртуальними машинами, таке як VMWare, створити кілька віртуальних машин, встановити на них Raspberry Pi OS, налаштувати Kubernetes та створити кластер.

Цей процес складається з двох етапів, кожен з яких, в свою чергу, ділиться на кілька кроків. Перший етап: створення та конфігурація віртуальних машин та встановлення на них Raspberry PI OS. Другий – налаштування кластеру.

При створенні віртуальної машини необхідно виконати базові налаштування: обрати образ, з якого буде здійснено встановлення операційної системи, налаштувати кількість ядер процесора та ОЗП, налаштувати мережу. На рисунку 3.1 зображений один з етапів створення віртуальної машини.

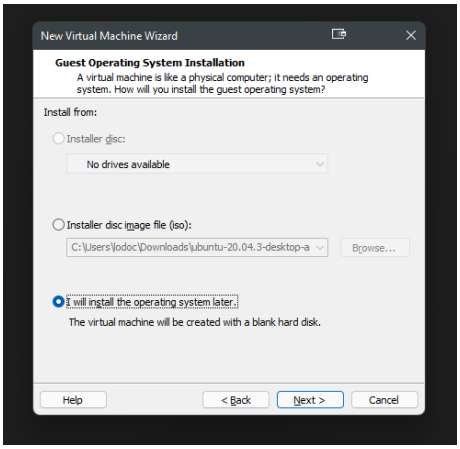

Рисунок 3.1 - Створення віртуальної машини

Після створення та першого запуску віртуальної машини, завантажується програма встановлення операційної системи (Рисунок 3.2). Потрібно обрати мову (Рисунок 3.3) та диск, на який буде встановлено операційну систему.

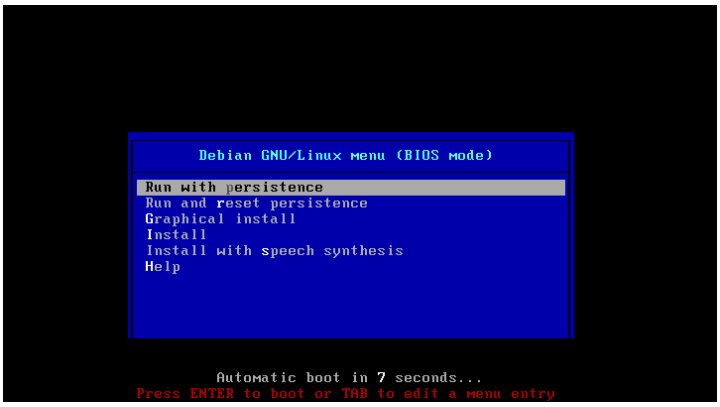

Рисунок 3.2 - Вікно вибору режиму встановлення

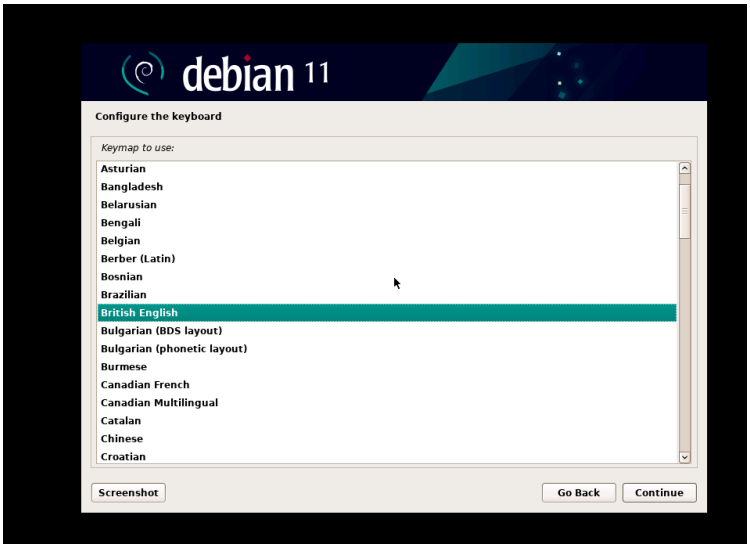

Рисунок 3.3 - Вибір мови

Після завершення базового налаштування розпочинається процес встановлення. В ході цього процесу програма скопіює необхідні файли на диск та налаштує завантажувач операційної системи GRUB. Цей процес займай кілька хвилин, після чого встановлення завершується і завантажується Raspberry Pi OS (Рисунок 3.4).

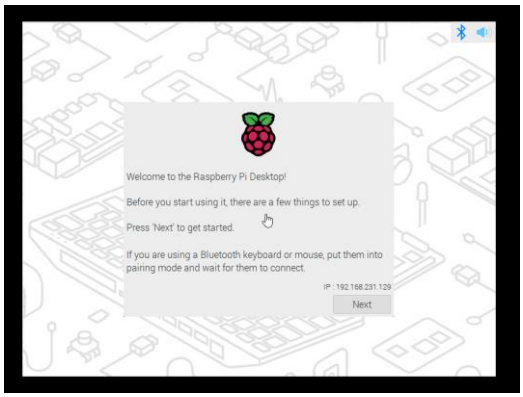

Рисунок 3.4 - Raspberry Pi OS

Цей процес потрібно повторити ще один раз: у кластері повинно бути принаймні 2 вузли. Після отримання двох налаштованих віртуальних машин з встановленою операційною системою, потрібно переконатись що вони знаходяться в одній мережі: для цього на одній з машин необхідно виконати команду ifconfig, щоб отримати її локальну IP адресу, а на іншій, за допомогою команди ping спробувати надіслати пакети з даними на цю адресу. Якщо прийде відповідь – значить машини мають спільну мережу. В даному випадку, оскільки обидві машини працюють в режимі NAT, вони мають доступ одна до одної, як видно на рисунку 3.5.

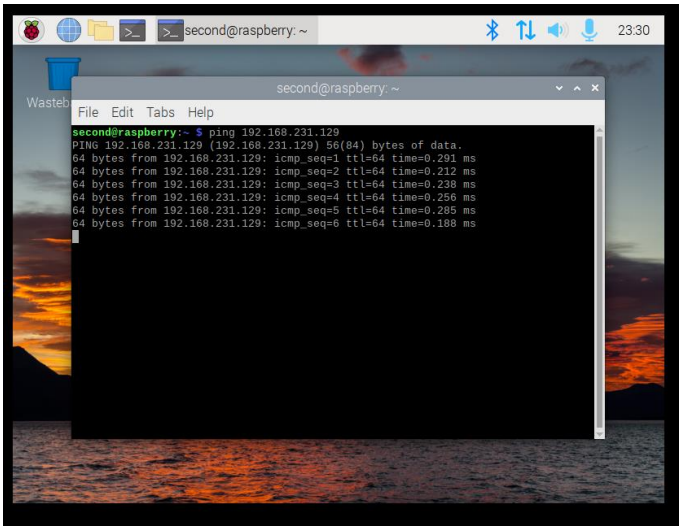

Рисунок 3.5 - Перевірка з’єднання

Наступним кроком буде налаштування кластеру Kubernetes. Як було визначено раніше, з метою оптимального використання ресурсів було вирішено використовувати збірку k3s. Спершу встановимо її на master вузол: для цього необхідно виконати наступну команду: curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="server --disable=traefik --flannel-backend=host-gw --tls-san=192.168.1.85 --bind-address=192.168.1.85 --advertise-address=192.168.1.85 --node-ip=192.168.1.85 --cluster-init" sh -s –

Після того, як master вузол налаштовано, потрібно перейти до встановлення k3s на worker вузлах. Для цього потрібно виконати команду curl -sfL https://get.k3s.io | K3S_URL=https://<ip master вузла> \

K3S_TOKEN="<токен> " sh -, де токен – це згенерований головним вузлом ключ, який використовується для підключення робочих вузлів.

3.2 Розробка програмного забезпечення

3.2.1 Розробка сервісу користувачів

Сервіс користувачів розроблено із застосуванням мови програмування V, вбудованого в неї фреймворку VWeb та ORM системи. Для початку потрібно було налаштувати базу даних. Оскільки система працює на базі кластеру Kubernetes, для налаштування бази даних використовувались yaml файли конфігурації. Приклад такого файлу наведено на рисунку 3.6.

```
apiVersion: v1
kind: Service
metadata:
  name: user-db
spec:
  selector:
    app: user-db
  ports:
    - protocol: TCP
      port: 5432
```

Рисунок 3.6 Лістинг yml файлу для оголошення сервісу

Для бази даних було створено наступні конфігурації:

- users-db-pv.yaml – файл, який оголошує так званий Persistent Volume – віртуальний дисковий розділ, який дозволяє pod'у не втрачати дані у випадку повторного створення контейнеру;
- users-db-claim.yaml – файл, який оголошує Persistent Volume Claim – об'єкт Kubernetes для керування доступом до Persistent Volume;
- users-db-config.yaml – файл, який містить конфігурації підключення до бази даних;
- users-db-deployment.yaml – файл, в якому оголошується нове розгортання. В ньому вказується який Docker образ повинен бути використаний для створення pod'ів, які порти повинні бути відкритими, скільки реплікацій додатку має бути створено, тощо;
- user-db-service.yaml – файл, в якому визначається, яким чином додаток буде взаємодіяти з зовнішньою та внутрішньою мережами;

Таким чином, для бази даних сервісу керування користувачами було виділено 3 ГБ дискового простору та відкрито порт 5432 для внутрішнього з'єднання.

Безпосередньо додаток сервісу користувачів також має 2 конфігураційних файли Kubernetes: deployment та service.

Розробка сервісу для користувачів почалася зі створення детальної моделі даних. Ці моделі є основою для ORM системи V, яка використовує їх для автоматичної генерації таблиць у базі даних. Моделі даних визначаються шляхом оголошення відповідних структур, що включають різноманітні атрибути та їх типи даних, відношення між об'єктами та інші важливі параметри. Завдяки цьому підходу, процес створення та підтримки бази даних стає більш ефективним і менш схильним до помилок, оскільки вся структура бази даних централізовано описується в одному місці та може бути легко модифікована у випадку необхідності. Всього було оголошено 3 таких

структури: Users (рис. 3.6), RefreshTokens (рис. 3.7) та Administrators (рис. 3.8).

```
@[table: 'users']
pub struct User {
pub:
	id int @[primary; sql: serial]
pub mut:
	email string @[unique]
	first_name string
	last_name string
	password string
	approved bool
  created_at    string    @[default:    'CURRENT_TIMESTAMP';
sql_type: 'TIMESTAMP']
}
```

Рисунок 3.6 Лістинг структури для користувачів

```
@[table: 'refresh_token']
pub struct RefreshToken {
	id int @[primary; sql: serial]
pub:
	user_id int
	token string @[unique]
}
```

Рисунок 3.7 Лістинг структури для refresh токенів

```
@[table: 'administrators']
pub struct Administrators {
	id int @[primary; sql: serial]
pub:
	user_id int
}
```

Рисунок 3.8 Лістинг структури для адміністраторів

Для створення REST API ресурсів в головному файлі програми створюється структура App, яка є похідною від vweb.Context. Далі, до структури App додаються reciver-функції – це такі функції, які прив'язані до конкретної структури та можуть здійснювати над нею операції. Функції, які використовуються для обробки HTTP запитів повинні мати додаткові додаткові атрибути з URI-шляхом, за яким вони будуть виконуватись, а також з HTTP методом. Приклад такої функції наведено на рисунку 3.9.

```
@['/register'; post]
fn (mut app App) create() vweb.Result {
	app.add_header("Access-Control-Allow-Origin", "*")

	mut user := json.decode(model.User, app.req.data) or {
		app.set_status(bad_request, "Bad request")
		return app.json(SimpleResponse{bad_request, "Check
your input data"})
	}

	if user.is_already_registered(app.db) {
		app.set_status(conflict, "Conflict")
		return   app.json(SimpleResponse{conflict,   "User
with this email already registered"})
	}

	if user.save(app.db) == 0 {
		app.set_status(internal_error, "Internal error")
		return     app.json(SimpleResponse{internal_error,
"Server error occured"})
	}

	app.set_status(created, "Created")
	return app.json(SimpleResponse{created, "Created"})
}
```

Рисунок 3.9 - Функція обробки реєстрації користувачів

Механізм авторизації було реалізовано на основі JWT. JSON Web Tokens (JWT) є розповсюдженим методом авторизації, що дозволяє безпечно передавати дані між клієнтом та сервером.

Користувач надсилає запит, який містить його адресу електронної пошти та пароль, і, у випадку якщо введені дані є коректними, генерується пара з двох токені:

- токен авторизації містить всі дані користувача та є дійсним протягом години;
- токен оновлення сесії, який містить ідентифікатор користувача та є дійсним тиждень;

Коли закінчується термін дії токен авторизації клієнт може надіслати токен оновлення сесії для того, щоб обміняти його на нову пару токенів. Такий підхід дозволяє збільшити загальну безпеку системи.

Логіка кожної з моделей інкапсулюється у вигляді reciver-функцій, приєднаних до структур.

3.2.2 Розробка мікросервісу керування файлами

При розробці мікросервісу керування файлами однією з головних задач є завантаження файлів. Простим підходом було б передавати весь файл у тілі одного запиту, однак при такій реалізації при завантаженні великих файлів може виникнути дві проблеми:

- обмеження максимального розміру тіла запиту зі сторони серверу не може бути нескінченно великим;
- те ж стосується і максимального часу, відведеного серверу на надання відповіді;

Для вирішення цієї проблеми було вирішено реалізувати наступний алгоритм завантаження файлів: спочатку на сервер надсилається запит на створення запису про наявність нового файлу. Після того, як сервер отримав дані про намір запису нового файлу, клієнт зчитує з файл сегменти по 16 МБ

та надсилає кожен з них окремими запитами. Сервер комбінує всі сегменти в один файл. На рисунку 3.10 представлено код роботи функції збереження сегмента.

```
    async fn save_file_segment(file_id: &str, data: &[u8]) ->
Result<(), warp::Rejection> {
        let file_path = format!("/tmp/{}", file_id);
        let mut file = OpenOptions::new()
            .create(true)
            .append(true)
            .open(file_path)
            .map_err(|e| {
                eprintln!("file open error: {}", e);
                warp::reject::reject()
            })?;

        file.write_all(data).map_err(|e| {
            eprintln!("file write error: {}", e);
            warp::reject::reject()
        })?;

        Ok(())
    }
```

Рисунок 3.10 - Лістинг функції збереження сегмента

Наступний важливий компонент даного мікросервісу – це функціональність отримання доступу до файлів. Були реалізовані наступні режими доступу до файлів: можливість перегляду вмісту каталогу та перегляд одного файлу. Одну з цих функцій приведено на рисунку 3.11.

```
    async fn handle_list_files(path: string) -> Result<impl
warp::Reply, warp::Rejection> {
        let files = fs::read_dir(path)
            .map_err(|e| {
                eprintln!("read dir error: {}", e);
                warp::reject::reject()
            })?
            .filter_map(|entry| entry.ok())
            .map(|entry|
entry.file_name().into_string().unwrap())
            .collect::<Vec<_>>();

        Ok(warp::reply::json(&files))
    }
```

Рисунок 3.11 - Лістинг функції перегляду вмісту каталогу

Також важливою функцією файлового сервісу є створення резервної копії всього диску за допомогою RabbitMQ. Це дозволяє організувати чергу завдань з резервного копіювання, які будуть виконуватись асинхронно. Для цього потрібно реалізувати:

- відправлення завдань (рис. 3.12): коли потрібно створити резервну копію, мікросервіс відправляє завдання у чергу RabbitMQ з вказаним шляхом до файлу або директорії.;
- споживач завдань (рис 3.13): споживач отримує завдання з черги і виконує резервне копіювання, забезпечуючи асинхронну обробку.

```
    async fn send_backup_task(file_path: &str, host: &str) ->
Result<(), Box<dyn std::error::Error>> {
        let conn = Connection::connect(host,
ConnectionProperties::default().with_tokio()).await?;
        let channel = conn.create_channel().await?;
        let payload = file_path.as_bytes();
        channel.basic_publish("", "backup_queue",
BasicProperties::default(), payload.to_vec(),
BasicPublishOptions::default()).await?;
        Ok(())
    }
```

Рисунок 3.12 Лістинг функції публікації повідомлень

```
    async fn consume_backup_tasks(host: &str) {
        let conn = Connection::connect(host,
ConnectionProperties::default().with_tokio()).await.unwrap();
        let channel = conn.create_channel().await.unwrap();
        let mut consumer = channel.basic_consume("backup_queue",
"my_consumer", BasicConsumeOptions::default(),
FieldTable::default()).await.unwrap();

        while let Some(delivery) = consumer.next().await {
            let delivery = delivery.unwrap();
            let file_path =
String::from_utf8(delivery.data).unwrap();
            // Логіка обробки завдання резервного копіювання

delivery.ack(BasicAckOptions::default()).await.unwrap();
        }
    }
```

Рисунок 3.13 - Лістинг функції споживача повідомлень

3.2.3 Розробка API-шлюзу

Мікросервіс API-шлюзу виступає в ролі проксі-серверу для запитів від клієнтського додатку, а також за перевірку авторизації. Для цього було

використано модуль proxy який входить до стандартного складу фреймворку fiber.

Для реалізації перевірки токену авторизації було створено middleware-функцію (рис. 3.14), яка виконується перед захищеними маршрутами та перевіряє актуальність токену та рівень доступу користувача.

```
func GetAuth(token string) (bool, UserResponse) {
	req, _ := http.NewRequest(http.MethodGet, fmt.Sprintf("%s%s", config.USER_SERVICE, USER_AUTH_ROUTE), nil)

	req.Header.Set(AUTHORIZATION, token)
	client := http.Client{
		Timeout: 30 * time.Second,
	}

	res, _ := client.Do(req)
	responseText, err := io.ReadAll(res.Body)
	var resp UserResponse
	json.Unmarshal(responseText, &resp)

	return true, resp
}
```

Рисунок 3.14 - Лістинг middleware-функції для перевірки авторизації

Для складніших операцій, які передбачають агрегацію кількох запитів до різних мікросервісів, заміть модуля proxy використовується HTTP net/http клієнт з стандартної бібліотеки Go, через який здійснюються надсилання запитів, після чого формується відповідь у JSON форматі на надсилається клієнту.

3.2.4 Розробка сервісу логування

Логування є критично важливим аспектом будь-якої програмної системи, забезпечуючи можливість моніторингу, налагодження та аналізу поведінки додатка. Розробка сервісу логування здійснювалась на мові програмування Go з використанням веб-фреймворку Fiber. Логи надсилаються через REST API, містять текст та тип логу, а задача на запис логу на диск публікується в RabbitMQ для асинхронної обробки.

Сервіс логування має тільки один ресурс /log (рис. 3.15), на який надсилаються запити наступного вмісту:

- тип повідомлення: в залежності від критичності проблеми логи можуть бути різного рівня: інформаційні, попередження, помилки, критичні помилки, тощо;
- текст повідомлення;

```
app.Post("/log", func(c *fiber.Ctx) error {
    logMessage := new(LogMessage)
    if err := c.BodyParser(logMessage); err != nil {
      return
      c.Status(fiber.StatusBadRequest).JSON(fiber.Map{"e
      rror": "cannot parse JSON"})
    }

    publishLogToQueue(logMessage)

      return
      c.Status(fiber.StatusAccepted).JSON(fiber.Map{"sta
      tus": "log received"})
})
```

Рисунок 3.15 - Лістинг функції обробки запиту

Після отримання повідомлення воно публікується в чергу для подальшої обробки та запису на диск.

3.2.5 Розробка сервісу Email

Перший крок у розробці сервісу — це створення REST API, яке буде приймати запити на відправку email. Користувачі або інші сервіси можуть надсилати HTTP POST запити, що містять необхідну інформацію для відправки листа, таку як адреса одержувача, тема та тіло повідомлення. Далі отриимані дані будуть перевірятись на коректність та публікуватись у чергу RabbitMQ.

Основні етапи життєвого циклу:

- отримання HTTP запиту;
- валідація отриманих даних;
- публікації завдання у чергу;
- асинхронна обробка завдання з черги;

Таким чином, мікросервіс якому потрібно надіслати повідомлення не повинен буде чекати доки SMTP сервер отримає та обробить запит, за це відповідає споживач повідомлень. Спрощений код функції споживача представлено на рисунку 3.16.

3.2.6 Розробка клієнтської частини

Для розробки клієнтської частини додатку було використано фреймворк Svelte. Першим кроком є ініціалізація проекту, для цього необхідно виконати команду npx degit sveltejs/template svelte-app, після чого розпочнеться інтерактивний процес налаштування додатку. Також, додатково потрібно встановити маршрутизатор – бібліотеку, яка надає можливість завантажувати різні сторінка в залежності від переданих URL параметрів. Для цього виконується команда npm i -D svelte-routing.

Розробка розпочинається з створення визначення основних сторінок та створення для них маршрутів.

```
func consumer() {
     conn, _ := amqp.Dial(AMP_HOST)
    defer conn.Close()

    ch, _ := conn.Channel()
    defer ch.Close()

    q, _ := ch.QueueDeclare("emails", true, false, false,
false, nil)

    msgs, _ := ch.Consume(q.Name, "", true, false, false,
false,nil)
    forever := make(chan bool)

    go func() {
        for d := range msgs {
            emailReq := EmailRequest{}
            if err := json.Unmarshal(d.Body, &emailReq); err
!= nil {
                log.Fatalf("Failed to unmarshal email
request: %v", err)
            }
            sendEmail(emailReq)
        }
    }()

    <-forever
}
```

Рисунок 3.16 - Лістинг функції споживача повідомлень

Було створено такі сторінки:

- сторінка реєстрації;
- сторінка авторизації;

- домашня сторінка;
- сторінка перегляду всіх файлів;
- загальна сторінка перегляду своїх файлів;
- сторінка перегляду поширених файлів;
- сторінка перегляду нещодавно завантажених файлів;
- сторінка з переліком файлів, відмічених як «Улюблені»;

Кожна сторінка є окремим компонентом svelte. Такий компонент – це файл з розширенням .svelte який містить в собі HTML розмітку, CSS стилі та логіку компоненту на JavaScript.

В рамках розробки клієнтської частини було розроблено такі основні компоненти:

- форма авторизації: містить в собі поля для введення електронної адреси та паролю;
- форма реєстрації: містить набір полів, потрібних для створення нового користувача;
- перемикач мови: змінює конфігурацію поточної мови;
- меню: містить перелік пунктів меню в панелі керування користувача;
- переглядач файлів: завантажує з серверу перелік файлів які містяться у директорії;
- компонент пошуку: виконує пошук файлів у поточній директорії;

Також, для повноцінного функціонування клієнтського додатку було розроблено кілька сервісних класів та функції. Однією з таких функцій є простий валідатор, який можна використовувати для перевірки відповідності даних певним правилам. Як правило, ця функція застосовується для перевірки коректності введених у форми даних. Код валідатору представлено на рисунку 3.17.

```
import { __ } from "./transalte"

export const validators = {
  minLength: length => value => {
    return value.length >= length ? "" : (length == 1 ?
__("Field can not be empty.") : (__("Minimum length is ") +
length.toString() + __(" characters.")))
  },
  maxLength: length => value => {
    return value.length <= length ? "" : (__("Maximum length
is ") + length.toString() + __(" characters."))
  },
  sameAs: (compare, valueName, asName) => value => {
    return value == compare ? "" : (valueName + __(" should
be the same as ") + asName)
  }
}

export const validate = (rules, value) => {
  return rules.map(r => r(value)).filter(msg => (msg != null
&& msg != ''))
}
```

Рисунок 3.17 - Лістинг функцій валідування даних

Також необхідно було розробити обгортку над HTTP клієнтом для врахування особливостей авторизації: якщо сервер повертає код 401 або 403, що свідчить про відсутність доступу до ресурсу, є ймовірність що справа у тому, що закінчився термін дії токену авторизації. Тому, в такий випадках є необхідність робити спробу обміну refresh токену та повторити запит.

Окрім того, визначений раніше алгоритм завантаження файлів потрібно підтримувати на стороні клієнтського додатку.

Також потрібно взяти до уваги алгоритм завантаження файлів сегментами, визначений раніше. Веб браузери реалізують File API, який

дозволяє працювати з файлами та директоріями. Кожен BLOB (Binary Large Object) у JavaScript має метод .slice() який дозволяє отримати сегмент файлу заданого розміру. На рисунку 3.18 представлено функцію, яка ділить файл на сегменти та надсилає кожен з них через мережу.

```
async function uploadFile(file, uid, chunkSize) {
    const totalChunks = Math.ceil(file.size / chunkSize);
    for (let i = 0; i < totalChunks; i++) {
        const start = i * chunkSize, end = Math.min(start
+ chunkSize, file.size), chunk = file.slice(start, end);

        const formData = new FormData();
        formData.append('chunk', chunk);
        formData.append('chunkNumber', i);

        const response = await
axios.post(`${BACKEND_HOST}${uploadChunk}/${uid}`, formData, {
            headers: {
                'Content-Type': 'multipart/form-data'
            }
        });
    }
}
```

Рисунок 3.18 - Лістинг коду функції поділу файлу на сегменти

3.3 Тестування

Тестування є критичним етапом розробки будь-якого програмного забезпечення, особливо коли мова йде про мікросервісну архітектуру, яка має складну структуру і залежності між компонентами. У цьому розділі ми розглянемо процес тестування файлового сховища, розробленого на базі кластеру Raspberry Pi, з використанням мікросервісної архітектури. Основними аспектами тестування будуть функціональне тестування,

інтеграційне тестування, тестування навантаження та продуктивності, а також тестування надійності та відмовостійкості.

Функціональне тестування зосереджується на перевірці коректної роботи кожного мікросервісу відповідно до вимог.

Тестові випадки для завантаження файлів:

- тестовий випадок 1: завантажити невеликий файл (менше 16 МБ) і перевірити, чи зберігається він правильно;
- тестовий випадок 2: завантажити файл середнього розміру (кілька сегментів) і перевірити, чи всі сегменти зберігаються і об'єднуються правильно;
- тестовий випадок 3: завантажити великий файл (кілька десятків сегментів) і переконатися у відсутності втрат даних та коректному об'єднанні сегментів;

Тестування скачування файлів:

- тестовий випадок 1: запросити скачування раніше завантаженого файлу і перевірити його цілісність;
- тестовий випадок 2: завантажити кілька файлів одночасно і перевірити коректність всіх файлів;

Інтеграційне тестування перевіряє взаємодію між різними мікросервісами і компонентами системи. Тестові випадки для інтеграційного тестування:

- завантажити файл і переконатися, що відповідні логи були створені і записані;
- відправити email і перевірити, що лог про відправку був створений і записаний;
- завантажити файл, перевірити його наявність і цілісність, а потім завантажити його назад і порівняти з оригіналом;

Тестування надійності та відмовостійкості перевіряє, як система реагує на збої та як швидко відновлюється після них. Тестові випадки:

- імітувати втрату з'єднання з мережею під час завантаження файлу і перевірити, чи система правильно обробляє збої та чи може продовжити завантаження після відновлення з'єднання;
- зупинити один з мікросервісів (наприклад, сервіс логування) і перевірити, чи система продовжує функціонувати і чи відновлює роботу після перезапуску сервісу;
- імітувати збій під час завантаження файлу і перевірити, чи система може відновити процес з місця збою;

Процес тестування мікросервісної архітектури файлового сховища на базі кластеру Raspberry Pi включає в себе різні аспекти: функціональне, інтеграційне, навантажувальне, продуктивне та тестування надійності. Використання автоматизованих інструментів допомагає зробити тестування більш ефективним і детальним, забезпечуючи високу якість і надійність системи. Тестування дозволяє виявити і виправити потенційні проблеми до того, як система буде впроваджена у продуктивне середовище, забезпечуючи стабільну і безперебійну роботу файлового сховища.

РОЗДІЛ 4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Природне середовище та його забруднення

Забруднення навколишнього природного середовища негативно позначається на здоров'ї. Забруднене атмосферне повітря може стати джерелом проникнення в організм шкідливих речовин через органи дихання. Забруднена вода може містити хвороботворні мікроорганізми й небезпечні для здоров'я речовини. Забруднені грунт і грунтові води погіршують якість сільськогосподарських продуктів харчування.

Людина здавна розглядає навколишнє природне середовище як джерело сировинних запасів (ресурсів), необхідних для задоволення своїх потреб. При цьому велика частка узятих від природи ресурсів повертається назад у вигляді відходів. Основна частина цих відходів і забруднень утворюється в містах. Усі види транспорту сильно забруднюють атмосферу вихлопними газами, що містять речовини, шкідливі для здоров'я людини. В результаті життєдіяльності людей утворюється багато промислових і побутових відходів. Від звалищ, розташованих поблизу міст, на велику відстань поширюється неприємний запах. На звалищах розмножується велика кількість мух, мишей і щурів, які є переносниками різних хвороботворних бактерій. Діяльність людини призводить до постійного забруднення навколишнього природного середовища: атмосферного повітря, природних вод і ґрунтів. До основних джерел забруднення повітря відносяться підприємства паливно-енергетичного комплексу, транспорт і промислові підприємства. Основні забруднювачі природних вод – нафта і нафтопродукти, які надходять у воду в результаті природних виходів нафти в районах її залягання, нафтовидобутку, транспортування, переробки та використання в якості палива та промислової

сировини. Забруднення водного середовища відбувається при надходженні у водойми рідини, що стікає з оброблених хімікатами сільськогосподарських та лісових земель, і при скиданні у водойми відходів підприємств. Все це погіршує санітарно-гігієнічні показники якості води. Основними забруднювачами ґрунтів є метали та їх сполуки, радіоактивні елементи, а також добрива і пестициди.

Під впливом навколишнього середовища в організмі людини можуть відбуватися зміни (мутації), які передаються у спадок. Постійне погіршення навколишнього середовища в кінцевому рахунку може призвести до зниження захисних властивостей організму, який перестане опиратися різним захворюванням.

Бурхливий розвиток інформаційних технологій також вніс значну частку у забруднення навколишнього середовища.

Важко уявити сьогодення без комп'ютерів, телевізорів та іншої електронної техніки. Інформаційні й телекомунікаційні технології стали способом життя людства, запорукою нового циклу розвитку цивілізації та планети. Інформаційні технології сьогодні є екологічнішими за більшість видів активної людської діяльності, проте їх ще не можна назвати справді екологічними. Ефективність інформаційних мереж прямо залежить від кількості користувачів, тобто від кількості комп’ютерів, під’єднаних до мережі. На кожен функціонуючий комп’ютер (який використовується в середньому протягом 4 років) припадає 1,5 новостворених комп’ютери. А близько третини комп’ютерів ніколи не буває продано взагалі – через швидкість, з якою вони втрачають технологічну актуальність.

До складу електронних пристроїв входять дуже токсичні з’єднання. Потрапляючи в навколишнє середовище, вони створюють серйозну небезпеку для здоров’я та життя людей. 22% ртуті, що видобувається щороку в усьому світі, йде саме на потреби електронної промисловості. Зокрема, цей хімічний елемент міститься в мобільних телефонах. Кадмій, який є канцерогеном, використовують практично в усіх

напівпровідникових пристроях. Свинець, особливо токсичний для нервової системи, міститься в акумуляторах і екранах моніторів. З часом захисні покриття електронних пристроїв розкладаються і у навколишнє середовище виділяється діоксин та інші високотоксичні з'єднання.

Стурбованість суспільства проблемами екології, а також жорсткіші закони із захисту навколишнього середовища змушують великих виробників електронного обладнання створювати спеціальні мережі, які займаються збором техніки, що вийшла з обігу, а також підприємства з її утилізації. Крім того при виробництві такого устаткування максимально збільшується частка матеріалів, придатних для переробки.

Вся оргтехніка містить органічні складові (пластик різних видів, матеріали на основі полівінілхлориду, фенолформальдегіду) і метали: дорогоцінні (золото, срібло); кольорові (алюміній, мідь); небезпечні (кадмій, свинець, цинк, нікель). Тому при списанні та утилізації обладнання керівнику необхідно керуватися законодавством в області охорони навколишнього середовища. Утилізація оргтехніки та комп'ютерів – це процес, який виконують в кілька етапів. Перший – списання обладнання безпосередньо з балансу підприємства. Другий – розбирання техніки і сортування отриманих матеріалів. Якщо деталі здатні служити вихідною сировиною, то їх відправляють на очищення. Так як до складу комп'ютерної техніки входять метали, то певним етапом утилізації персональних комп'ютерів є те, що за допомогою спеціальних технологій з них можна видобути частку дорогоцінних, корисних та рідкісних металів.

Одне з нововведень для утилізації друкованих плат придумали співробітники з Національної фізичної лабораторії Великобританії. Вони продемонстрували можливості спеціального розчину, дія якого зумовлює відшарування електронних компонентів.

Практично жодне підприємство не зможе самостійно утилізувати комп'ютери та оргтехніку, так як цей процес вимагає сучасного обладнання

та специфічних знань. Тому довірити таку роботу можна тільки професіоналам, які мають великий досвід у даній сфері.

Проблема утилізації використаних та морально застарілих комп'ютерів, периферійного обладнання, стає гострішою з кожним роком. Обсяги виробництва продуктів інформаційно-телекомунікаційних технологій та частота їх заміни на нові моделі примушують компанії замислюватись над проблемою біодеградації. Успіхи в цій галузі допоможуть, серед іншого, компаніям-виробникам зменшити податки, котрі вони сплачують зараз за утилізацію застарілих моделей. Останнє тим більше важливо, оскільки робить екологізацію економічно вигідною, тож спрямовує у цю сферу дедалі більше зусиль дослідників та довгострокових капіталовкладень. Таким чином, подальший розвиток інформаційних технологій не збільшить, а, навпаки, зменшить техногенне навантаження на довкілля. Отже, розвиток і вдосконалення сучасних інформаційних технологій повинно слугувати не тільки для створення максимально комфортних умов життя людини, але й для досягнення безвідходного процесу утилізації відпрацьованої техніки, не завдаючи шкоди навколишньому середовищу.

4.2 Правила безпеки при роботі на персональному комп'ютері

До роботи на персональному комп'ютері допускають осіб, які пройшли інструктажі з питань охорони праці та пожежної безпеки.

Користувач зобов'язаний:

- виконувати правила внутрішнього трудового розпорядку;
- не допускати на своє робоче місце сторонніх осіб;
- не виконувати вказівок, які суперечать правилам охорони праці та пожежної безпеки;
- знати правила надання домедичної допомоги;
- знати розташування та вміти користуватись первинними засобами пожежогасіння;

- вміти працювати з ПК.

Основні небезпечні та шкідливі виробничі фактори, що можуть впливати на користувача:

- підвищений рівень статичної електрики;
- нерівномірність розподілу яскравості в полі зору;
- підвищена яскравість світлового зображення;
- ураження електричним струмом;
- напруга зору та уваги;
- тривалі статичні навантаження.

Основним обладнанням робочого місця є ПК або ноутбук, монітор, клавіатура, маніпулятор, робочий стіл, стілець (крісло).

При розміщенні елементів робочого місця слід враховувати:

- робочу позу користувача;
- простір для розміщення користувача;
- можливість огляду елементів робочого місця;
- можливість огляду простору поза межами робочого місця;
- можливість робити записи, розміщувати на робочому столі документацію та матеріали, які використовує користувач.

Розміщення елементів робочого місця не повинно заважати рухам та переміщенню для експлуатування ПК.

ПК потрібно встановлювати на рівній твердій поверхні (столі). Не дозволено встановлювати ПК та оргтехніку на хитких підставках чи на похилій поверхні. Не встановлювати ПК впритул до стіни, перегородки. Не допускати загородження вентиляційних отворів ПК сторонніми предметами.

Розетка біля ПК має бути в доступному місці, щоб в аварійних випадках можна було своєчасно його відімкнути. Не рекомендовано використовувати подовжувачі. Під час переміщення ПК, периферійних пристроїв витягти вилку живлення з розетки. Не допускати ушкодження чи модифікування шнура живлення. Заборонено ставити важкі речі на шнур

живлення, тягнути чи надмірно перегинати його, скручувати та перев'язувати шнур живлення вузлом.

Під'єднувати ПК до електромережі лише за допомогою справних штепсельних з'єднань та електророзеток заводського виробництва.

Штепсельні з'єднання та електророзетки повинні мати спеціальні контакти для під'єднання нульового захисного провідника. Їхня конструкція має забезпечувати з'єднання нульового захисного провідника раніше, ніж з'єднання фазового та нульового робочого провідників. Порядок роз'єднань при вимкненні має бути зворотнім.

Заборонено під'єднувати електрообладнання до звичайної двошнурової електромережі.

Перед початком роботи працівник повинен:

- оглянути робоче місце і впевнитись, що на ньому немає сторонніх предметів, все обладнання і блоки ПК з'єднані із системним блоком з'єднувальними шнурами;

- перевірити надійність встановлення апаратури на робочому столі. Монітор не повинен стояти на краю стола. Повернути монітор так, щоб було зручно дивитися на екран — під прямим кутом (а не збоку) і трохи згори вниз; при цьому екран має бути трохи нахиленим — нижній край ближче до користувача;

- перевірити загальний стан апаратури, справність електропроводки, з'єднувальних шнурів, штепсельних вилок, розеток, заземлення захисного екрана;

- вставити вилку в розетку і впевнитися, що вона міцно тримається;

- заборонено вставляти і виймати вилку мокрими руками;

- за потреби приєднати до комп'ютера необхідну апаратуру. Усі кабелі, що з'єднують системний блок з іншими пристроями, під'єднувати та від'єднувати лише при вимкненому комп'ютері;

- про всі виявлені несправності інформувати керівника робіт і не братися до роботи, доки їх не буде усунено.

Вимоги безпеки під час виконання роботи.

Під час роботи на ПК:

- стійко встановити клавіатуру на робочому столі, не допускаючи її хитання, водночас передбачити можливість її поворотів та переміщень;
- якщо в конструкції клавіатури не передбачено простору для упору долонь, клавіатуру розміщують на відстані не менше 100 мм від краю столу в оптимальній зоні моніторного поля;
- під час роботи на клавіатурі сидіти рівно, не напружуватися;
- щоб зменшити несприятливе навантаження на користувача при роботі з комп'ютерною мишею (вимушена поза, необхідність постійно контролювати якість дій), забезпечити велику вільну поверхню столу для переміщення комп'ютерної миші та зручного упору ліктьового суглоба;
- періодично при вимкненому комп'ютері прибирати пил із поверхонь апаратури спеціальними серветками.

Вимоги безпеки при роботі з ПК та оргтехнікою:

- вмикати і вимикати комп'ютер, ноутбук та іншу оргтехніку тільки вимикачами, забороняється вимикати шляхом витягуванням вилки з розетки.

- забороняється знімати захисні пристрої з обладнання і працювати без них;

- не допускати до комп'ютера та оргтехніки сторонніх осіб, які не беруть участі в роботі;

- забороняється переміщати та переносити системний блок, монітор, принтер, будь-яке обладнання, яке знаходиться під напругою;

- забороняється під час роботи пити будь-які напої, приймати їжу;

- забороняється будь-яке фізичне втручання у пристрій комп'ютера, принтера, сканера, ксерокса під час їх роботи.

- забороняється залишати включене обладнання без нагляду;

- забороняється класти предмети на комп'ютерне обладнання, монітори, екрани та оргтехніку;

- суворо виконувати загальні вимоги з електро- та пожежної безпеки;

- під час усунення застрягання паперу на ксероксі чи принтері, задля уникнення ураження електрострумом, необхідно відключити обладнання від електромережі. Необхідно також вимикати обладнання від мережі при тривалому простої;

- самостійно розбирати та проводити ремонт електронної та електронно-механічної частини комп'ютера, периферійних пристроїв, оргтехніки категорично забороняється. Ці роботи може виконувати тільки спеціаліст або інженер з технічного обслуговуваннюя комп'ютерної техніки.

- комп'ютер, будь-які його периферійні пристрої, оргтехніку використо-вувати у суворій відповідності до експлуатаційної документації до них;

- під час виконання роботи необхідно бути уважним і не відволікатись;

- про всі виявлені несправності та збої в роботі апаратури необхідно повідомити безпосередньо інженера з обслуговування комп'ютерної техніки або завідувача ДНЗ.

Тривалість безперервної роботи за ПК не має перевищувати 2 год. Після цього необхідно зробити 15-хвилинну перерву. Якщо виник зоровий дискомфорт або інші неприємні відчуття, необхідно зробити коротку перерву.

Для зниження нервово-емоційного напруження, стомлення зорового аналізатора, поліпшення мозкового кровообігу, подолання несприятливих наслідків гіподинамії, запобігання втомі доцільно під час декількох перерв виконувати комплекс вправ.

Вимоги безпеки після закінчення роботи:

- вимкнути комп'ютер, ноутбук, телевізор, плазмову панель, LCD-екран, принтер, ксерокс, сканер, колонки та іншу оргтехніку від електромережі, для чого необхідно вимкнути тумблери, а потім акуратно витягнути штепсельні вилки з розетки;

- протерти зовнішню поверхню комп'ютера чистою вологою тканиною. При цьому не допускати використання розчинників, одеколону, препаратів в аерозольній упаковці;

- прибрати робоче місце;

- ретельно провітрити приміщення з персональним комп'ютером та іншою оргтехнікою.

ВИСНОВКИ

У даній роботі було здійснено розробку файлового сховища на базі кластеру Raspberry Pi з використанням мікросервісної архітектури. Основною метою розробки було створення надійної, масштабованої та ефективної системи для зберігання та перегляду файлів, що може бути легко розширена та модифікована відповідно до змінних вимог.

Розробка файлового сховища на базі кластеру Raspberry Pi з використанням мікросервісної архітектури також супроводжувалась певними викликами. Одним із них було забезпечення узгодженості даних та надійності комунікації між мікросервісами. Це було вирішено завдяки впровадженню механізмів обробки помилок та повторного виконання запитів, а також використанню брокера повідомлень RabbitMQ для асинхронної обробки завдань.

Іншим важливим аспектом була оптимізація продуктивності та використання ресурсів. Завдяки використанню ефективних алгоритмів збереження та обробки даних, а також налаштуванню параметрів кластеру, вдалося досягти високих показників продуктивності при мінімальних витратах на обладнання.

Подальший розвиток даної системи може включати інтеграцію з іншими сервісами та додатками, що дозволить розширити її функціональні можливості. Додавання підтримки для додаткових протоколів зберігання та обробки даних, таких як NFS або S3, дозволить забезпечити більшу гнучкість та сумісність з іншими системами.

Завдяки використанню сучасних технологій та архітектурних рішень вдалося створити надійну, масштабовану та ефективну систему, яка може бути легко адаптована до змінних вимог та умов експлуатації. Цей підхід відкриває широкі можливості для подальшого розвитку та інтеграції з іншими системами та сервісами, забезпечуючи високу гнучкість та адаптивність.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Осухівська Г.М., Тиш Є.В., Луцик Н.С., Паламар А.М. Методичні вказівки до виконання кваліфікаційних робіт здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 123 «Комп'ютерна інженерія" усіх форм навчання. Тернопіль, ТНТУ, 2022. 28 с.
2. Харченко О., Яцишин В. Розробка та керування вимогами до програмного забезпечення на основі моделі якості. Вісник ТДТУ. Тернопіль, 2099. Т.14. №1. С.201-207.
3. Yatsyshyn V., Pastukh O., Pflfmfr A., Zharovsryy R. Technology of relational database management system performance evalution during computer systems design. Scientific Jornal of TNTU, Ternopil, Ukraine, 2023. Vol.109, No 1. P. 54-56.
4. Raspberry Pi Computer Boards. URL: https://www.okdo.com/c/pi-shop/theraspberry-pi/ (дата звернення 15.05.2022 р.).
5. Жидецький В.Ц. Охорона праці користувачів комп'ютерів. Львів : Афіша. 2000. 176 с.
6. Building Microservices: Designing Fine-Grained Systems 1st Edition, Sam Newman, 280, 2015, O'Reilly.
7. Елізабет Робсон. Head First. Програмування на JavaScript/ Елізабет Робсон, Ерік Фрімен. 2022. 672 с.
8. Code Complete (Developer Best Practices) 2nd Edition, Steve McConnell. 916. 2015.
9. Мережа розробників Mozilla URL: https://developer.mozilla.org/ (дата звернення 10.06.2024).
10. Офіційна документація kubernetes URL: https://kubernetes.io /docs/home/ (дата звернення 10.06.2024).
11. Офіційна документація vlang URL: https://docs.vlang.io/introduction.html (дата звернення 10.06.2024).

12. Офіційна документація rust-lang URL: https://doc.rust-lang.org/book/ (дата звернення 10.06.2024).

13. Офіційна документація svelte URL: https://svelte.dev/docs/ (дата звернення 10.06.2024).

14.Мартін Роберт. Чистий код/ Мартін С.Р. Фабула. 2019. 368 с.

15. Мартін Роберт. Чиста архітектура/ Мартін С.Р. Фабула. 2019. 368 с.

16. Бєлов С.В., Ільницька А.В., Казько А.Ф. Безпека життєдіяльності: Підручник для вузів. Вища школа. 2000. 138 с.

17. НПАОП 0.00-7.15-18 «Про затвердження Вимог щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями».

Додаток А

Технічне завдання

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії

Кафедра комп'ютерних систем та мереж

«Затверджую»

Завідувач кафедри КС

_____ Осухівська Г.М.

«___» _____ 2024 р.

ФАЙЛОВЕ ХМАРНЕ СХОВИЩЕ НА БАЗІ КЛАСТЕРА RASPBERRY PI ІЗ ЗАСТОСУВАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

ТЕХНІЧНЕ ЗАВДАННЯ

на 9 арк.

Вид робіт Кваліфікаційна робота

На здобуття освітнього ступеня «бакалавр»

Спеціальність 123 Комп'ютерна інженерія

«УЗГОДЖЕНО»

Керівник кваліфікаційної роботи

_____ к.т.н., доц. Шингера Н.Я.

«___»_____ 2024 р.

ВИКОНАВЕЦЬ

Студент групи СІ-42

_____ Сотник А.П.

«___»_____ 2024 р.

Тернопіль 2024

1 Загальні відомості

1.1 Повна назва та її умовне позначення

Повна назва кваліфікаційної роботи: «Файлове хмарне сховище на базі кластера Raspberry PI із застосуванням мікросервісної архітектури».

Умовне позначення кваліфікаційної роботи: КС КРБ 123.132.00.00

1.2 Виконавець

Студент групи СІ-42 факультету комп’ютерно-інформаційних систем і програмної інженерії кафедри комп’ютерних систем та мереж Тернопільського національного технічного університету імені Івана Пулюя Сотник Андрій Петрович.

1.3 Підстава для виконання роботи

Підставою для виконання кваліфікаційної роботи є наказ по університету (№4/7-408 від 24.04.2024 р.).

1.4 Планові терміни початку та завершення роботи

Плановий термін початку виконання кваліфікаційної роботи – 01.02.2024 р.

Плановий термін завершення виконання кваліфікаційної роботи – 24.06.2024 р.

1.5 Порядок оформлення та пред’явлення результатів

Порядок оформлення пояснювальної записки та графічного матеріалу здійснюється відповідно до чинних норм та правил ІСО, ГОСТ, ЕСКД, ЕСПД та ДСТУ.

Пред’явлення проміжних результатів роботи з виконання кваліфікаційної роботи здійснюється відповідно до графіку, затвердженого керівником роботи.

Попередній захист кваліфікаційної роботи відбувається при готовності роботи на 90%, наявності пояснювальної записки та графічного матеріалу.

Пред’явлення результатів кваліфікаційної роботи відбувається шляхом захисту на відповідному засіданні ЕК, ілюстрацією основних досягнень та за допомогою графічного матеріалу.

2 Призначення і цілі створення системи

2.1. Призначення системи

Система призначена для збереження файлів у розподіленому хмарному середовищі.

2.2 Мета створення системи

Розробка гнучкої і розширюваної платформи для зберігання файлів.

2.3 Характеристика об’єкту

2.3.1 Основні задачі та функції об’єкту

До основних задач, які має виконувати файлове хмарне сховище на базі кластера Raspberry PI із застосуванням мікросервісної архітектури належать безпечне та надійне зберігання файлів та можливість роботи з ними.

3 Вимоги до системи

3.1 Вимоги до системи в цілому

При розробці такої системи як файлове сховище, основними не функціональними вимогами є безпека та надійність. Безпековий аспект полягає в реалізації надійного алгоритму авторизації та автентифікації, таким чином, щоб кожен користувач мав доступ виключно до тих файлів, які йому належать, або до тих, до яких йому було надано доступ іншими користувачами. Надійність такої системи буде полягати у механізмах, які забезпечують зберігання файлу навіть у випадку відмови системи або виходу її з ладу. Також важливою вимогою є можливість масштабування з метою додавання нових накопичувачів або засобів резервного копіювання.

3.1.1 Вимоги до структури та функціонування системи

Передбачається застосування таких структурних компонентів як:

- мережевий маршрутизатор, який створить локальну мережу для вузлів кластеру а також буде забезпечувати керування доступом до системи з зовнішніх мереж;
- мікрокомп’ютери Raspberry Pi 4 у кількості 3шт., які разом утворюватимуть Kubernetes кластер;
- твердотільні накопичувачі, сумарним об’ємом не менш ніж 500 ГБ.;
- HDD накопичувач або RAID масив з кількох таких накопичувачів, сумарним об’ємом не меншим ніж об’єм твердотільних дисків.

3.1.2 Вимоги до способів та засобів зв’язку між компонентами системи

Максимальна швидкість, яку стабільно повинна забезпечувати мережа становить 1 ГБіт/с., а отже маршрутизатор повинен підтримувати стандарт GigE, а для з’єднання пристроїв у мережі повинен бути використаний кабель типу «вита пара», не менше ніж четвертої категорії у 4-парній конфігурації. На програмному рівні передбачено використання RESTARTPI для синхронної та RabbitMQ – для асинхронної. Крім того:

- максимальний ліміт швидкості, на якій система повинна оперувати, не повинен перевищувати 1 Гбіт/с.;
- мінімальний ліміт швидкості, на якій система повинна оперувати, не повинен перевищувати 100 Мбіт/с.;
- час відгуку системи повинен складати менше 400 мс. при запиті сторінки; час відповіді при запиту файлу не повинен перевищувати 1с на файл розміром 10 МБ.;
- доступність: система повинна використовувати реплікацію на рівні мікросервісів та автоматично перезапускатись у разі відмови;
- шифрування: трафік між клієнтом та системою повинен бути зашифрований відповідно до протоколу HTTPS; резервні копії файлів повинні зберігатись у зашифрованому вигляді;

Вимоги до діагностування системи

3.1.3 Перспективи розвитку, модернізація системи

Подальший розвиток даної системи може включати інтеграцію з іншими сервісами та додатками, що дозволить розширити її функціональні можливості. Додавання підтримки для додаткових протоколів зберігання та обробки даних, таких як NFS або S3, дозволить забезпечити більшу гнучкість та сумісність з

іншими системами. Модульність архітектури забезпечує ефективне впровадження змін.

3.1.4 Вимоги до надійності системи

Надійність такої системи буде полягати у механізмах, які забезпечують зберігання файлу навіть у випадку відмови системи або виходу її з ладу. Оскільки будь-який тип накопичувача, особливо при інтенсивному використанні, може вийти з ладу частково або повністю, потрібно це наперед передбачити. Для того щоб забезпечити цілісність файлів, можна використовувати RAID-масиви, які будуть дзеркально записувати файли на кілька накопичувачів або багаторівневі системи резервного копіювання, для прикладу: мати окремі накопичувачі з резервними копіями та додатково зберігати копії на іншому сервері у хмарі в зашифрованому вигляді. Також, додатковим засобом надійності може бути розподілений запис файлів між кількома накопичувачами, таким чином, якщо один з них вийде з ладу, втрачено буде тільки частина даних, яку можна буде відносно швидко відновити з копії.

3.1.5. Вимоги до функцій та задач, які виконує система

Основні функціональні вимоги:

- авторизація та автентифікація: користувач повинен мати можливість зареєструватись, авторизуватись та підтвердити право на доступ до тих чи інших файлів;
- завантаження та скачування файлів: користувач повинен мати можливість завантажувати свої файли в рамках діючих для нього обмежень у обсязі, а також скачувати файли, що знаходяться у його власності без обмежень;

- управління файлами: користувач повинен мати можливість видаляти, переміщувати, перейменовувати файли у його власності;
- спільний доступ до файлів: користувач повинен мати можливість ділитись своїми файлами з іншими користувачами або шляхом генерації унікального посилання;
- сповіщення: користувач має отримувати сповіщення від системи засобами електронної пошти;
- адміністративний розділ: система повинна мати поділ на користувачів та адміністраторів.

3.1.6 Вимоги до апаратного забезпечення

- мережевий маршрутизатор, який створить локальну мережу для вузлів кластеру а також буде забезпечувати керування доступом до системи з зовнішніх мереж;
- мікрокомп’ютери Raspberry Pi 4 у кількості 3шт., які разом утворюватимуть Kubernetes кластер;
- твердотільні накопичувачі, сумарним об’ємом не менш ніж 500 ГБ.;
- HDD накопичувач або RAID масив з кількох таких накопичувачів, сумарним об’ємом не меншим, ніж об’єм твердотільних дисків;

3.1.7 Вимоги до програмного забезпечення

До мов програмування:

- малий розмір бінарного файлу;
- мінімальні кількість залежностей.

До K3S:

- низьке споживання ресурсу;
- швидкість компіляції.

До програмного забезпечення для формування кластера – швидке введення в експлуатацію нового вузла.

4 Вимоги до документації

Документація повинна відповідати вимогам ЄСКД та ДСТУ.

Комплект документації повинен складатись з:

- пояснювальної зписки;
- графічного матеріалу:

1. Мікросервісна архітектура.
2. Архітектура Kubernetes.
3. Діаграма варіантів використання.
4. Структура клієнтського додатку.
5. Дизайн API на рівні всієї системи.

*Примітка: У комплект документації можуть вноситися зміни та доповнення в процесі розробки.

5 Стадії та етапи проектування

Таблиця 1 – Стадії та етапи виконання кваліфікаційної роботи бакалавра

№ з/п	Назва етапів роботи	Термін виконання етапів роботи
1.	Розробка і затвердження технічного завдання	01.02.24-09.02.2024

Продовження табл. 1

2	Аналіз технічного завдання	05.02.24-11.02.24
3.	Розробка загальної структури системи; обґрунтування вибору апаратного забезпечення; обгрунтування вибору програмного забезпечення	03.06.24-06.06.24
4.	Проектування програмної системи	07.06.24-10.06.24
5.	Виконання практичної частини	11.06.24-18.06.24
6.	Виконання розділу «Безпека життєдіяльності та охорона праці.	18.06.24-19.06.24
7.	Оформлення кваліфікаційної роботи	20.06.24-21.06.24
9.	Попередній захист кваліфікаційної роботи	14.06.24
10.	Захист кваліфікаційної роботи	24.06.24-28.06.24

6 Додаткові умови виконання кваліфікаційної роботи

Під час виконання кваліфікаційної роботи у дане технічне завдання можуть вноситись зміни і доповнення