ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Гринішак Сергій Володимирович

УДК 004.02

Оптимізація алгоритму масштабування хмарної системи з застосуванням кроссервісної архітектури

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації магістра

Нормоконтроль

_____к.т.н. Мануляк І.З.
(підпис, дата, розшифрування підпису)

Студент

_____ Гринішак С.В.
(підпис, дата, розшифрування підпису)

Допускається до захисту
Завідувач кафедри

_____к.т.н. Пашкевич О.П.
(підпис, дата, розшифрування підпису)

Керівник роботи :

_____ к.т.н. Ващишак С.П.
(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2021

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

Освітній ступінь: «магістр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____

« ___ » _____ 2021 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Гринішаку Сергію Володимировичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи
Оптимізація алгоритму масштабування хмарної системи з застосуванням кроссервісної архітектури

керівник роботи:
Ващишак Сергій Петрович, кандидат технічних наук

затверджена наказом вищого навчального закладу від « 30 »
вересня 2021 року
11/1
№ НВ

2. Термін подання студентом роботи 03.12.2021

3. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)
1. Огляд та аналіз існуючих програмних рішенньдля забезпечення відмовостійкості хмарних систем
2. Методи реалізації системи

3. Опис програмної реалізації

4. Дата видачі завдання 22.02.2021

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи (проекту)	Термін виконання етапів роботи	Примітка

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

Вихідні дані роботи:

Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Сторінка	Опис графічного матеріалу	Сторінка	Опис графічного матеріалу

АНОТАЦІЯ

Тема кваліфікаційної роботи – “Оптимізація алгоритму масштабування хмарної системи з застосуванням кроссервісної архітектури”.

Робота полягає у визначенні способів підвищення надійності та стійкості розподілених хмарних систем на прикладі інтерактивної веб–системи для розширення словникового запасу іноземної мови.

Для досягнення поставленої задачі були сформульовані наступні завдання дослідження, що визначили логіку дослідження та його структуру:

– проаналізувати існуючі підходи до проектування розподілених хмарних систем;

– проаналізувати існуючі засоби для розгортки та забезпечення відмовостійкості хмарних систем;

– створити систему для вивчення нових слів, що спроектована для роботи у хмарній інфраструктурі та використовує переваги і мінімізує недоліки хмарного середовища;

– забезпечити засоби для автоматичного масштабування критичними для системи метриками.

значення одержаних результатів роботи полягає в розробці розподіленої хмарної системи, що дозволяє вивчати слова іноземної мови, може бути розгорнута у Kubernetes кластері та динамічно адаптується під поточне навантаження системи за допомогою автоматичного масштабування екземплярів кожного з сервісів

Ключові слова: ХМАРНА ІНФРАСТРУКТУРА, МІКРОСЕРВІСИ, REST*, KUBERNETES, МАСШТАБУВАННЯ, ВІДМОВОСТІЙКІСТЬ*

SUMMARY

The topic of the qualification work is "Optimization of the cloud system scaling algorithm using cross-service architecture".

The work is to identify ways to increase the reliability and resilience of distributed cloud systems on the example of an interactive web system to expand the vocabulary of a foreign language.

To achieve this goal, the following research objectives were formulated, which determined the logic of the study and its structure:

- analyze existing approaches to the design of distributed cloud systems;

- analyze existing tools for scanning and ensuring the resilience of cloud systems;

- create a system for learning new words, designed to work in the cloud infrastructure and uses the advantages and minimizes the disadvantages of the cloud environment;

- provide tools for automatic scaling with critical metrics for the system.

The value of the results is the development of a distributed cloud system that allows you to learn foreign language words, can be deployed in the Kubernetes cluster and dynamically adapts to the current system load by automatically scaling instances of each service

Keywords: CLOUD INFRASTRUCTURE, MICROSERVICE, REST, KUBERNETES, SCALE, FAILURE RESISTANCE.

ЗМІСТ

РОЗДІЛ 1. ОГЛЯД ТА АНАЛІЗ ІСНУЮЧИХ ПРОГРАМНИХ РІШЕНЬ ДЛЯ ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ ХМАРНИХ СИСТЕМ 10
1.1 Балансування навантаження 12
1.2 Масштабування і висока доступність 17
1.3 Аналіз шаблонів розробки розподілених відмовостійких систем 18
1.3.1 Балансування на стороні клієнта (Client Side Load Balancing) 19
1.3.2 Реєстр сервісів (Service Discovery) 20
1.3.3 API Gateway 22
1.3.4 Circuit Breaker 23
1.3.5 Комбінація шаблонів 23
1.4 Аналіз програмних рішень для забезпечення відмовостійкості 24
1.4.1 Apache Mesos 25
1.4.2 Kubernetes 26
1.4.3 Docker Swarm 27
1.4.4 Порівняння оркестраторів 28
Висновки до розділу 1 30
РОЗДІЛ 2. МЕТОДИ РЕАЛІЗАЦІЇ СИСТЕМИ 32
2.1 Середовище розробки та мова програмування 32
2.2 Інструменти для збірки проекту 35
2.3 Засоби розгортки програмного продукту 36
2.4 Технології для розробки інтерфейсу 38
2.4.1 Фреймворк Angular 6 38
2.4.2 Мова TypeScript 39
2.4.3 Структура інтерфейсу 40
2.5 Технології для розробки серверної частини 41
2.6 Взаємодія між клієнтом і сервером 41
2.7 Масштабування баз даних 45
2.7.1 Партиціювання (partitioning) 45
2.7.2 Реплікація (replication) 45
2.7.3 Фрагментація (sharding) 46

2.8 Автоматичне відновлення даних 47
Висновки до розділу 2 48
РОЗДІЛ 3. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ 49
3.1 Реалізація мікросервісних підходів 49
3.2 Реєстр сервісів 49
3.3 Сервер конфігурації 50
3.4 Взаємодія сервісів один з одним 51
3.5 Точка доступу для повернення метрики послуги 53
3.6 Робота з базою даних 53
3.6.1 Опис таблиць бази даних 53
3.6.2 Об'єктно– реляційне відображення даних у системі 57
3.7 Розгортання системи 59
3.8 Автомасштабування за довільними метриками 63
3.9 Аутентифікація користувача 68
3.10 Робота із зображеннями 70
3.11 Генерація вимови слів 72
3.12 Переклад слів 72
3.13 Тестування серверної частини 73
3.14 Документація з API 74
3.15 Веб– клієнт 79
3.16 Взаємодія компонентів системи 82
3.17 Практика вимови 83
3.18 Слідкуйте за прогресом у вивченні слів 83
3.19 Взаємодія користувача с сервісом 84
Висновки до розділу 3 90
ВИСНОВКИ 91
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 92

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД– База даних

SQL– Structured query language — мова структурованих запитів

СУБД– Система керування базами даних

MVC– Model-View-Controller — Модель–вигляд–контролер

HTML– HyperText Markup Language — Мова розмітки гіпертекстових документів

IDE– Integrated development environment — Інтегроване середовище розробки

UI– User Interface — Інтерфейс користувача

JSON– JavaScript Object Notation

ORM –Object-relational mapping — Об’єктно-реляційна проекція

CRUD –Create read update delete (створення, зчитування, зміна і видалення)

ВСТУП

Актуальність теми. Хмарна інфраструктура дуже популярна і широко використовується багатьма компаніями тому що дозволяє економити як на обслуговуванні і персоналі, так і на інфраструктурі. Але, оскільки віртуальні машини у хмарній інфраструктурі не надійні, системи повинні бути спроектовані таким чином, щоб мінімізувати простої та перерви на обслуговування. Це зумовлює актуальність розробки розподілених хмарних систем, що максимально ефективно використовують хмарне середовище.

Мета дослідження полягає у визначенні способів підвищення надійності та стійкості розподілених хмарних систем на прикладі інтерактивної веб– системи для розширення словникового запасу іноземної мови.

Для досягнення поставленої задачі були сформульовані наступні **завдання дослідження**, що визначили логіку дослідження та його структуру:

– проаналізувати існуючі підходи до проектування розподілених хмарних систем;

– проаналізувати існуючі засоби для розгортки та забезпечення відмовостійкості хмарних систем;

– створити систему для вивчення нових слів, що спроектована для роботи у хмарній інфраструктурі та використовує переваги і мінімізує недоліки хмарного середовища;

– забезпечити засоби для автоматичного масштабування критичними для системи метриками.

Об’єктом дослідження є розподілені системи та засоби автоматичного масштабування вузлів кластера.

Предметом дослідження є алгоритми для автоматичного масштабування у розподілених системах.

Методи дослідження. Розв’язання поставлених задач виконувались засобами комп’ютерного моделювання, зокрема з використанням наступних методів:

– реалізація клієнта, що генерує навантаження на систему та імітує поведінку реального користувача;

– аналіз метрик кожного з сервісів для виявлення проблем;

– аналіз запитів що виконуються довше за все за допомогою використання розподіленого трасування запитів.

Наукова новизна одержаних результатів. Найбільш суттєвими науковими результатами магістерської дисертації є удосконалення алгоритму автоматичного горизонтального масштабування у оркестраторі Kubernetes за рахунок застосування інформації про довжину черги, що дозволило збільшити ефективність зас тосування обчислювальних ресурсів у системі.

Практичне значення одержаних результатів роботи полягає в розробці розподіленої хмарної системи, що дозволяє вивчати слова іноземної мови, може бути розгорнута у Kubernetes кластері та динамічно адаптується під поточне навантаження системи за допомогою автоматичного масштабування екземплярів кожного з сервісів

Структура й обсяг дипломної роботи.

Магістерська дисертація складається зі вступу, трьох розділів, висновку, переліку посилань з 40 найменувань. Повний обсяг магістерської дисертації складає 92 сторінок, з яких перелік посилань займає 3 сторінок

РОЗДІЛ 1. ОГЛЯД ТА АНАЛІЗ ІСНУЮЧИХ ПРОГРАМНИХ РІШЕНЬ ДЛЯ ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ ХМАРНИХ СИСТЕМ

Метою дипломної роботи є розробка кроссервісної відмовостійкої хмарної системи, забезпечення та оптимізацію автоматичного масштабування сервісів системи залежно від навантаження та провести аналіз інструментів для забезпечення відмовостійкості. Дана система є веб– застосунком, її завдання полягає в розширенні словникового запасу іноземних мов, тобто вивчення нових та повторенні вже вивчених слів іноземної мови.

В розробці та побудові сервісу, що мають надмірні навантаження, зтикаються з двома значущими проблемами це: надійність системи та маштабованість. Зазвичай системам присутнє нерівномірне навантаження протягом певних проміжків часу (рисунок 1.1). Розробка та проектування системи повинна проводитись таким чином, так щоб надмірне навантаження (тимчасові всплески) не змогли вплинути на надійність системи.

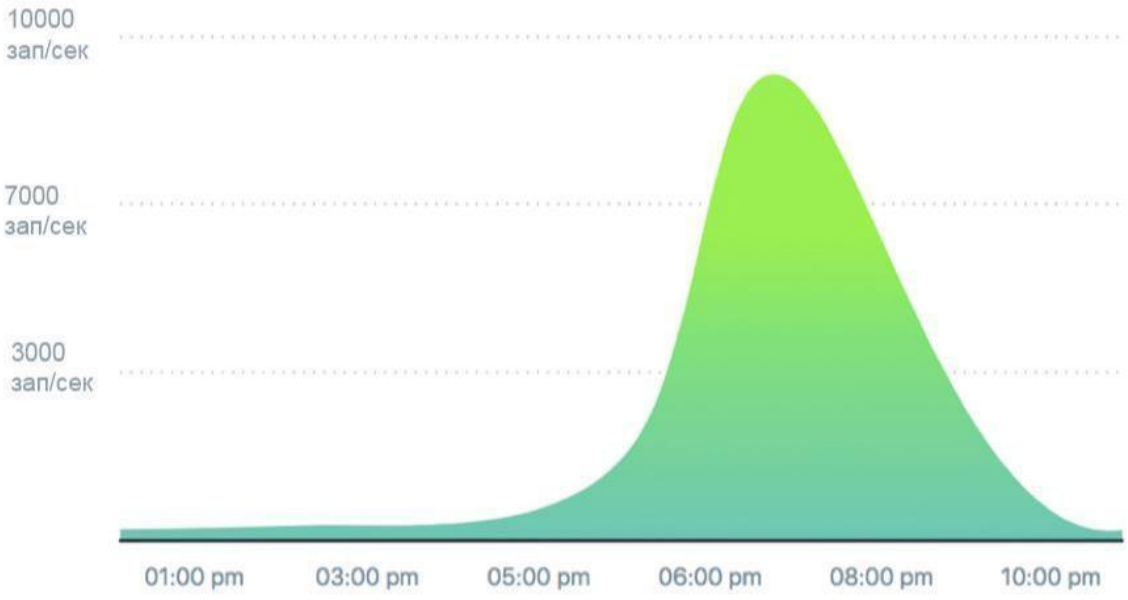

Рисунок 1.1 – Розподіл кількості запитів залежно від часу доби

Програмний продукт повинен забезпечити наступні можливості:

– єдиний шлюз для виконання всіх запитів;

– реєстрація екземплярів сервісів у загальному реєстр;

– аналіз розподілених запитів та часу їх виконання;

– автоматичне масштабування сервісів за рівнем використання CPU або оперативної пам’яті;

– автоматичне масштабування сервісів залежно від довжини черги запитів;

– розгортка системи у хмарному середовищі;

– балансування навантаження на стороні кожного клієнта;

– повтор запитів у випадку помилки;

– дострокове переривання запитів у випадку відмови сервісу або довгого часу відповіді;

– точки для отримання інформації про стан кожного сервісу;

– точки для отримання інформації про поточні метрики кожного сервісу;

– інтеграція зі сторонніми хмарними сервісами;

– перегляд зареєстрованих екземплярів кожного сервісу;

– контроль стану кожного екземпляру сервісу та перезапуск у випадку невдачі перевірки життєвого стану;

При побудові надійних виробничих систем мінімізація простоїв і перерв в обслуговуванні часто є головним пріоритетом. Незалежно від того, наскільки надійні системи та програмне забезпечення, виникають проблеми, які можуть призвести до збоїв програмного забезпечення або сервера.

Використання інфраструктури високої доступності – важлива стратегія зменшення впливу різних подій на поведінку системи. Високодоступні системи можуть автоматично відновлюватись із сервера або компонента.

Одна з цілей високої доступності – усунути єдині точки відмови в інфраструктурі. Єдина точка відмови – це компонент інфраструктури, який може вимкнути всю систему, якщо він стане недоступним. Таким чином, будь–який компонент, необхідний для правильного функціонування системи, не має надмірності і вважається єдиною точкою відмови.

Для забезпечення високої доступності практично необхідно ретельно врахувати різні компоненти. Висока доступність залежить, не тільки від програмного забезпечення, а також від таких факторів, як:

– навколишнє середовище: якщо всі сервери розташовані в одній географічній області, умови навколишнього середовища, такі як землетруси або повені, можуть зруйнувати всю систему. Резервні сервери у різних центрах обробки даних та географічних регіонах підвищують надійність;

– апаратне забезпечення: сервери з високою доступністю повинні бути стійкими до відключень живлення та збоїв обладнання, включаючи жорсткі диски та мережеві інтерфейси;

– програмне забезпечення: весь програмний стек, включаючи операційну систему та саму програму, повинен бути підготовлений до обробки несподіваних помилок, які, наприклад, можуть вимагати перезапуску системи;

– дані: втрата та неузгодженість даних можуть бути викликані кількома факторами та не обмежуються збоями жорсткого диска. Системи високої доступності повинні забезпечувати безпеку даних у разі збою;

– мережа: незаплановані збої у роботі мережі – ще одна точка відмови для високодоступних систем. Важливо мати стратегію резервної мережі у разі будь– яких збоїв.

Проаналізуємо підходи щодо побудови стійких систем на прикладному рівні хмарних систем.

1.1 Балансування навантаження

Балансування навантаження є одним з ключовою частиною стійких інфраструктур, яка зазвичай використовується для покращення продуктивності та надійності веб–сайтів, програм, баз даних та інших служб шляхом розподілу навантаження на декількох серверах.

Веб– інфраструктура без балансування навантаження може виглядати приблизно так:

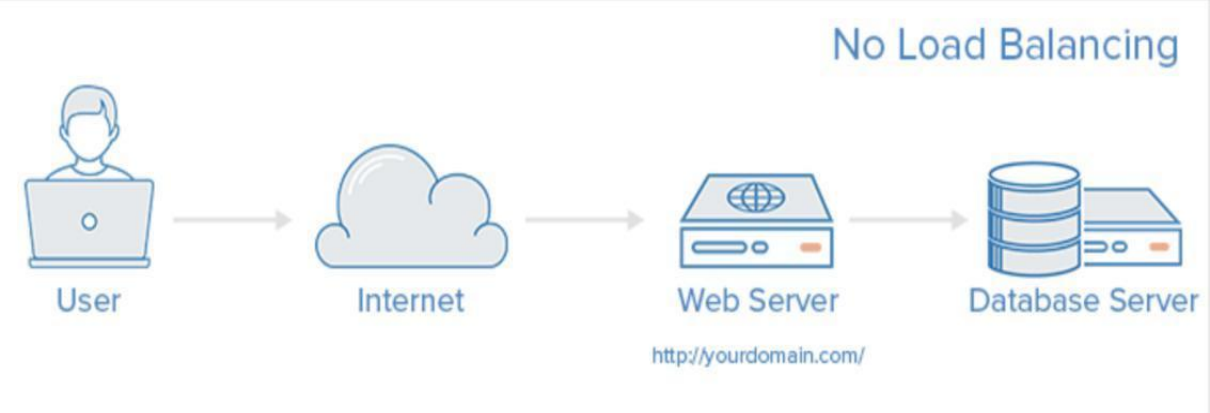

Рисунок 1.1 – Інфраструктура без балансування навантаження

У цьому прикладі користувач підключається безпосередньо до веб–сервера на сайті yourdomain.com.

Якщо цей єдиний веб зможе отримати доступ до веб–сайту.

Крім того, якщо багато користувачів намагаються одночасно отримати доступ до сервера і він не в змозі впоратися з завантаженням, то користувачі можуть довго чекати відповіді або взагалі не зможуть з’єднатися.

Цю єдину точку збою можна модернізувати та покращити шляхом введення балансування навантаження та, принаймні, одного додаткового веб–сервера (рисунок 1.2).

Як правило, всі сервери посилатимуть однакові відповіді, щоб користувачі отримували правильну відповідь незалежно від того, який сервер відповідає.

У наведеному прикладі користувач доступається до балансувальника навантаження, який пересилає запит користувача на один з серверів, який потім безпосередньо відповідає на запит користувача.

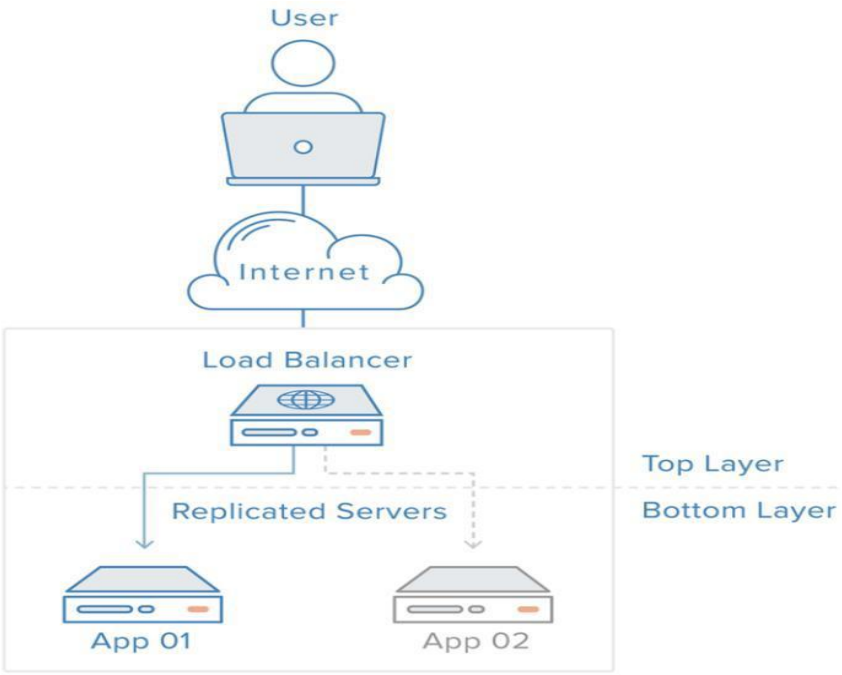

Рисунок 1.2 – Інфраструктура з балансуванням навантаження

Для усунення єдиних точок відмови, кожен рівень системи повинен бути готовим до надмірності. Наприклад, візьмемо інфраструктуру, що складається з двох ідентичних, надлишкових веб– серверів з балансуванням навантаження.

Трафік, що створюють клієнти, буде рівномірно розподілятися між веб–серверами, однак, якщо один з серверів відмовить, балансування навантаження переспрямовуватиме весь трафік на решту доступних серверів.

У цьому випадку рівень веб–серверів не є однією точкою збою, оскільки:

– маються надлишкові компоненти для одного й того ж сервісу;

– механізм, що знаходиться перед рівнем веб–сервісів (балансування навантаження) здатний виявити збої в компонентах і адаптувати його поведінку для своєчасного відновлення.

За описаним сценарієм, що не є рідкістю в реальному житті, рівень балансування навантаження сам по собі залишається єдиною точкою відмови.

Однак усунення єдиної точки відмови, що залишилася, може бути складним завданням;

Навіть якщо можна легко налаштувати додатковий балансувальник навантаження, щоб досягти надмірності, не існує очевидної точки над балансувальниками навантаження для виявлення та відновлення збою.

Тільки надлишковість не може гарантувати високу доступність. Потрібно встановити механізм для виявлення збоїв та виконання дій, коли один з компонентів інфраструктури стає недоступним.

Виявлення та відновлення збоїв для надлишкових систем можна реалізувати, використовуючи підхід «зверху донизу»: верхній шар стає відповідальним за відстеження та контроль збоїв шару, що знаходиться під ним. У попередньому прикладі сценарій балансування навантаження є верхнім шаром. Якщо один з веб–серверів (нижній рівень) стає недоступним, балансування навантаження зупинить переадресацію запитів для цього конкретного сервера.

Цей підхід має обмеження: в інфраструктурі буде місце, де верхній шар або неіснуючий, або недоступний, як у випадку з балансуванням навантаження. Створення сервісу виявлення несправностей для балансування навантаження на зовнішньому сервері просто створить нову точку відмови.

З таким сценарієм необхідний розподілений підхід. Кілька резервних вузлів повинні бути з’єднані разом як кластер, де кожен вузол повинен бути здатним однаково виявляти та відновлювати несправності (рисунок 1.3).

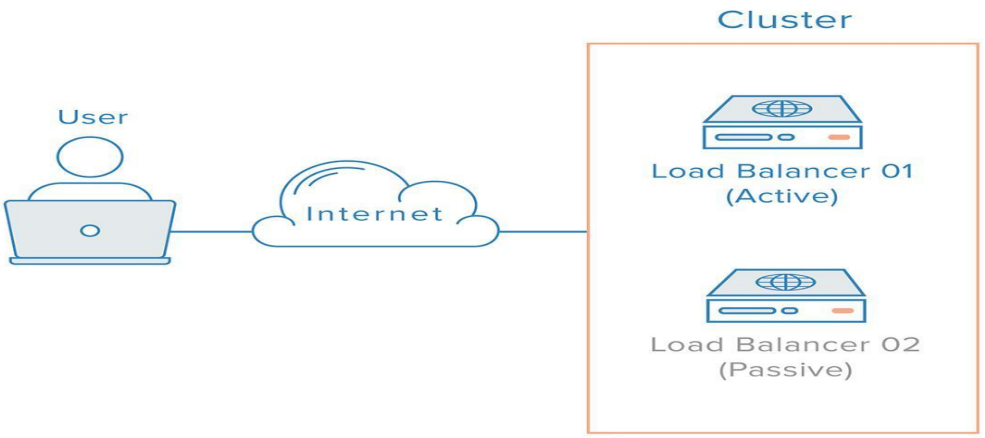

Рисунок 1.3 – Кластер балансувальників навантаження

Однак у разі балансування навантаження виникає додаткова складність через роботу DNS–серверів. Відновлення після збою балансувальника навантаження зазвичай означає переключення на резервний балансувальник навантаження, що означає, що вам потрібно змінити DNS, щоб вказати ім'я домену на IP– адресу балансувальника навантаження. Розповсюдження таких змін через Інтернет може зайняти багато часу, що може призвести до серйозних збоїв у роботі системи.

Одне з можливих рішень – по черзі використовувати балансування навантаження DNS round–robin (по черзі). Проте такий підхід ненадійний, оскільки призводить до балансування на стороні клієнта. Більш стійким та надійним рішенням є використання систем, що дозволяють гнучке переміщення IP (рисунок 1.4). Ця функція називається плаваючим IP. Ці функції надаються Amazon Web Services (AWS), Digital Ocean та багатьма іншими постачальниками.

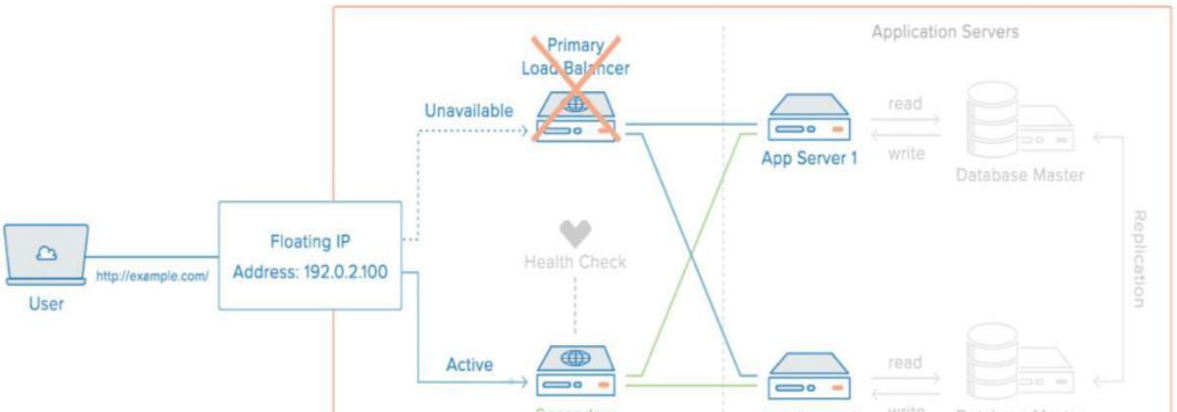

Рисунок 1.4 – Балансування за допомогою Floating IP

Зміна IP–адреси в міру необхідності вирішує проблеми розповсюдження та кешування, властиві змінам DNS за рахунок надання статичного IP– адреса, який можна нескладно відновити за потреби. Доменне ім'я може залишатися пов'язаним з тією ж IP–адресою, в той час як сама IP–адреса переміщається між серверами.

Кожен рівень високодоступної системи матиме різні потреби у програмному забезпеченні та конфігурації. Однак на рівні програмних додатків балансувальники навантаження є важливою частиною програмного забезпечення для забезпечення високої доступності.

1.2 Масштабування і висока доступність

Масштабованість можна розглядати з кількох сторін, але наша основна мета – надання хмарних ресурсів у міру за потреби.

Масштабованість (як знизу вгору, так і зверху вниз) означає більшу продуктивність для користувача та підвищену складність для розробника хмарних систем.

Масштабованість має бути забезпечена не тільки для самої системи (функціональне масштабування), але і для її пропускної спроможності (масштабування навантаження).

Іншою ключовою характеристикою хмарних систем є географічний розподіл даних (географічна масштабованість), що дозволяє розміщувати дані та ресурси максимально близько до користувача завдяки набору хмарних центрів зберігання (шляхом міграції). Хмарна інфраструктура також має маштабуватися всередині.

Розмір серверів та сховища можуть змінюватись, та це не впливає на користувачів.

Система завжди повинна мати можливість відповісти на запит користувача. Враховуючи час простою мережі, помилку користувача та інші обставини, може бути дуже складно виконати цю умову надійним та детермінованим способом.

1.3 Аналіз шаблонів розробки розподілених відмовостійких систем

Для масштабування та надійності систем треба розбивати всі сервіси на маленькі частини та дублювати функціональність (рисунок 1.8). Такий тип архітектури називається мікросервісним. Ця архітектура полягає в тому, що система будується з декількох маленьких сервісів, кожен з яких добре виконує свою частину роботи. Для комунікації один з одним можуть виконуватися як синхронні REST запити, так і асинхронна обробка та відправка повідомлень за допомогою черг.

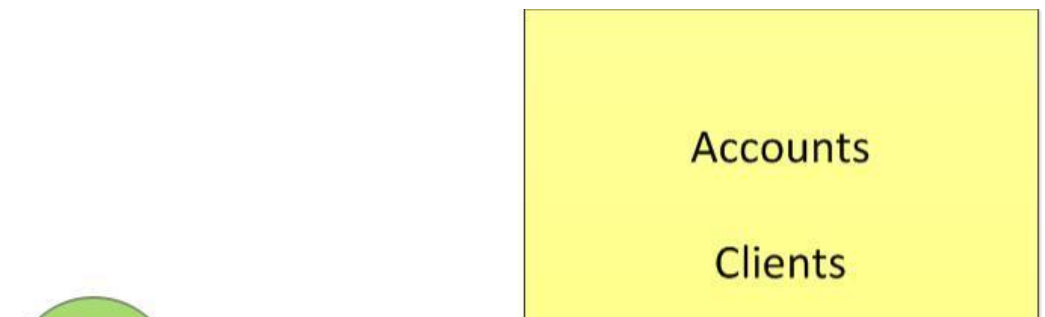

Рисунок 1.8 – Монолітна система

Система мікросервісів має розподілений характер. Отже, систему необхідно поділити на логічні служби (рисунок 1.9).

Система мікросервісів поділяє функціональні блоки у окремі сервіси і масштабується за допомогою багаторазового запуску необхідних функцій на декількох серверах.

Розподілені сервіси забезпечують можливість самостійного масштабування. Кожну службу можна розгорнути окремо, збільшуючи кількість екземплярів служби, які потребують більшої обчислювальної потужності (наприклад, для підтримки більшої кількості запитів).

Це називається горизонтальним масштабуванням екземплярів сервісів.

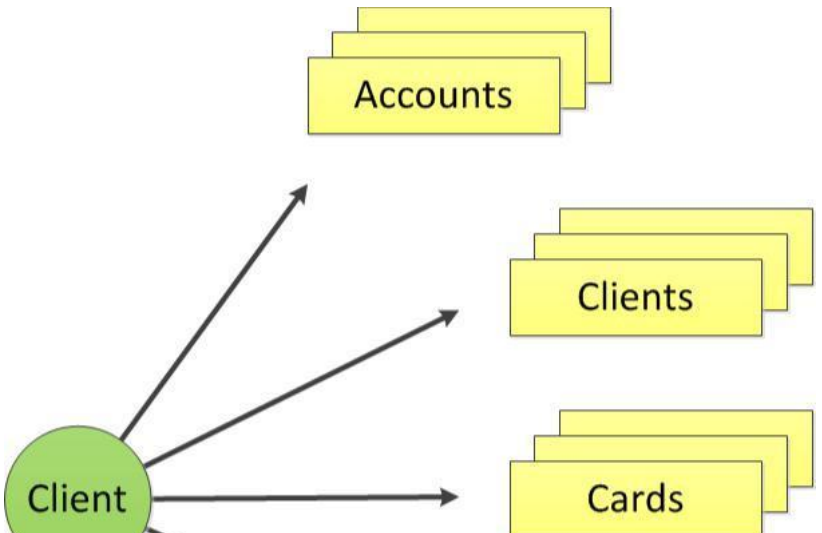

Рисунок 1.9 – Система, яка розподілена на сервіси

1.3.1 Балансування на стороні клієнта (Client Side Load Balancing)

Ми можемо застосувати модель балансування навантаження на стороні клієнта. Ця застосування вже знайомого балансувальника навантаження, але не на високому рівні для вхідних запитів, а програмно під час виконання запитів між вашими службами. Коли клієнт хоче надіслати запит до сервісів, він може вибрати будь– який екземпляр із списку доступних екземплярів сервісу.

Це дозволяє рівномірно розподіляти навантаження від клієнтів по всіх екземплярах служби. Будь– який сервіс у мережі може бути клієнтом.

Зазвичай клієнт робить запити та отримує відповіді, а служба може бути клієнтом іншої служби та робити запити до неї, і це можна робити каскадом (рис. 1.10).

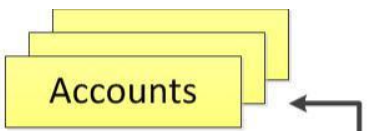

Рисунок 1.10 – Каскадні запити

1.3.2 Реєстр сервісів (Service Discovery)

Загалом все, що обговорювалося вище, може бути найбільш поширеним методом реалізації мікросервісної архітектури. Однак у цьому випадку виникає проблема. Як замовник отримує інформацію про доступні екземпляри інших сервісів? Для цього можна використовувати конфігураційні файли, але це додає ще більшої складності, оскільки конфігурацію необхідно оновлювати після кожної зміни топології. Використання шаблону (Service Discovery) вирішує проблему. Цей шаблон є окремим сервісом, який діє як адресна книга (реєстр) для інших сервісів (рисунок 1.11). Коли сервіс запускається, вона повідомляє адресу (наприклад, хост та порт) до реєстру та надсилає повідомлення (наприклад, кожні 30 секунд) поки він працює. Таким чином, реєстр знає які сервіси ще доступні.

Крім того, цей шаблон забезпечує гнучку масштабованість. Таким чином, сервіси можна масштабувати під час роботи системи, динамічно додаючи або зменшуючи кількість екземплярів при потребі. Коли клієнт хоче відправити запит до сервісу, єдине, що йому потрібно знати – це ім'я, яке використовується для реєстрації служби у реєстрі.

Таким чином, клієнт запитує у реєстрі список доступних екземплярів служби, а потім виконує прямий запит до вибраного екземпляра сервісу. Екземпляр зазвичай вибирається з використанням логіки балансування навантаження.

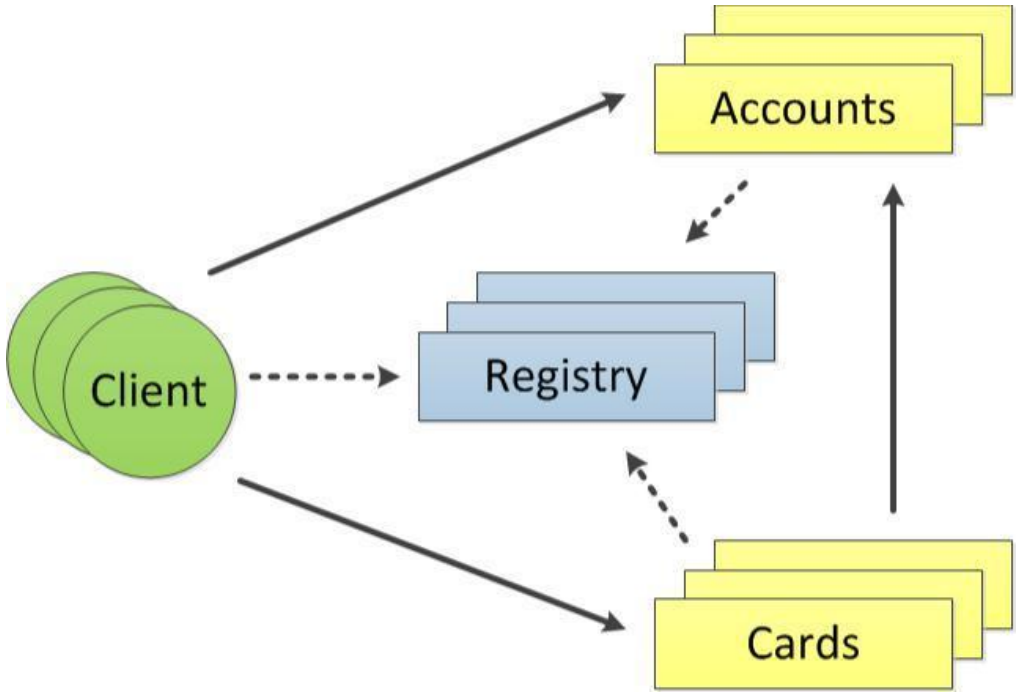

Рисунок 1.11– Робота з реєстром сервісів

1.3.3 API Gateway

Ви можете помітити, що ця архітектура вимагає, щоб клієнти знали, де знаходиться реєстр і як інтегруватися з ним. Це проста практика для мікросервісних систем. Проте зовнішні клієнти не повинні нічого знати про

внутрішню архітектуру системи. Архітектура мікросервісів використовує шаблон шлюзу API (API Gateway), яка є єдиною точкою доступу до системи зовнішніх систем (рис. 1.12).

Використання шлюзу маскує властиву системі складність для зовнішніх клієнтів.

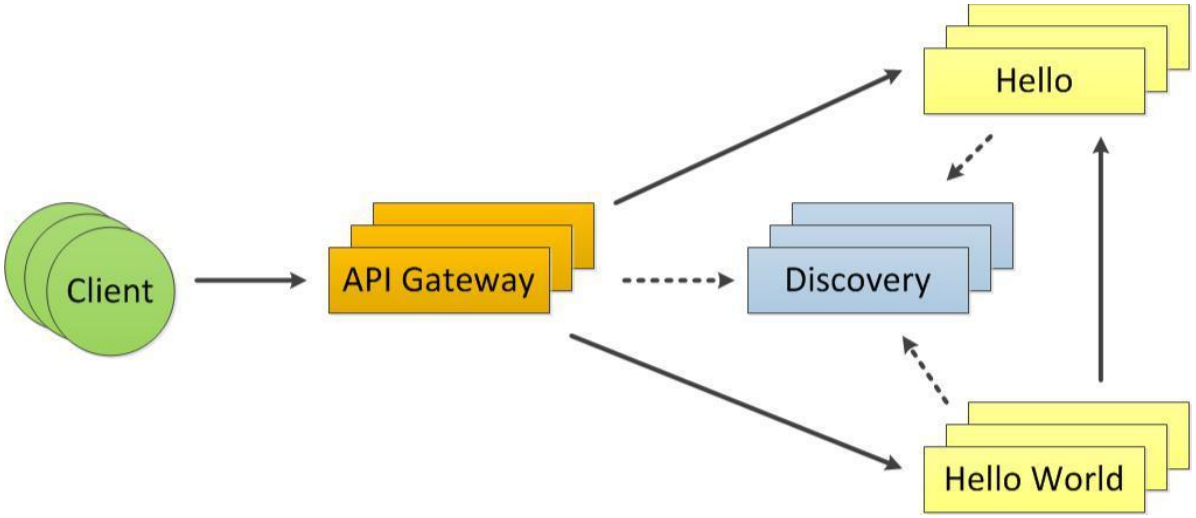

Рисунок 1.12 – API шлюз

Шлюзи API додають функціональність за допомогою існуючих мікросервісів. Шлюзи шукають у реєстрах список доступних сервісів та отримують до них доступ за допомогою балансування навантаження. Є два типи шлюзів: проксі та логічні.

Проксі–шлюзи просто переадресовують виклик служби на потрібний сервіс, а логічні шлюзи можуть робити кілька запитів служб, щоб надати своїм клієнтам інший API.

1.3.4 Circuit Breaker

Якщо система складається з багатьох сервісів і одна з кінцевих сервісів починає давати збій, можуть виникати каскадні переривання через тайм– аут

(рисунок 1.13). Таким чином, клієнт може почекати кілька секунд, доки запит не буде скасовано через таймаути всіх каскадних запитів.

Для прискорення реакції системи використовується шаблон проектування Автоматичний вимикач (Circuit Breaker), який навмисно заздалегідь перериває запити, які перервуться по тайм–ауті.

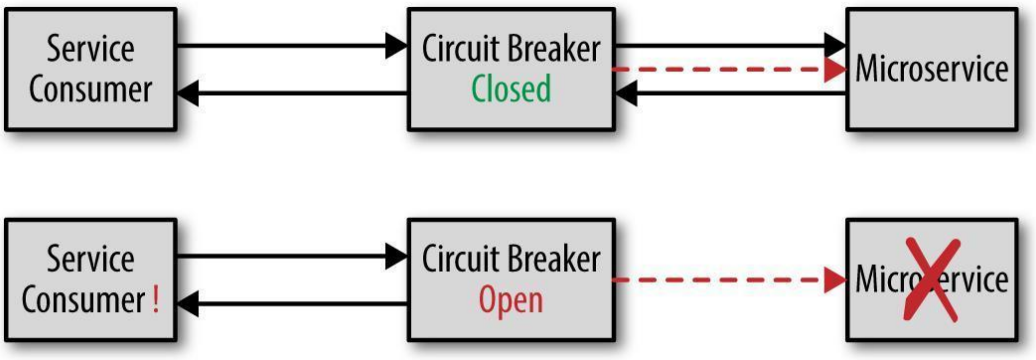

Рисунок 1.13– Шаблон проектування Circuit Breaker

1.3.5 Комбінація шаблонів

Це базові шаблони проектування, без яких не існує справжньої системи мікросервісів

.Комбінація цих моделей дозволяє створювати великі відмовостійкі системи (рис. 1.14).

Також існує ще багато інших різних шаблонів, які спрямовані на покращення відмовостійкості системи та спрощенні її конфігурації.

Для відновлення інфраструктури після збою використовуються системи, здатні генерувати нові екземпляри сервісів.

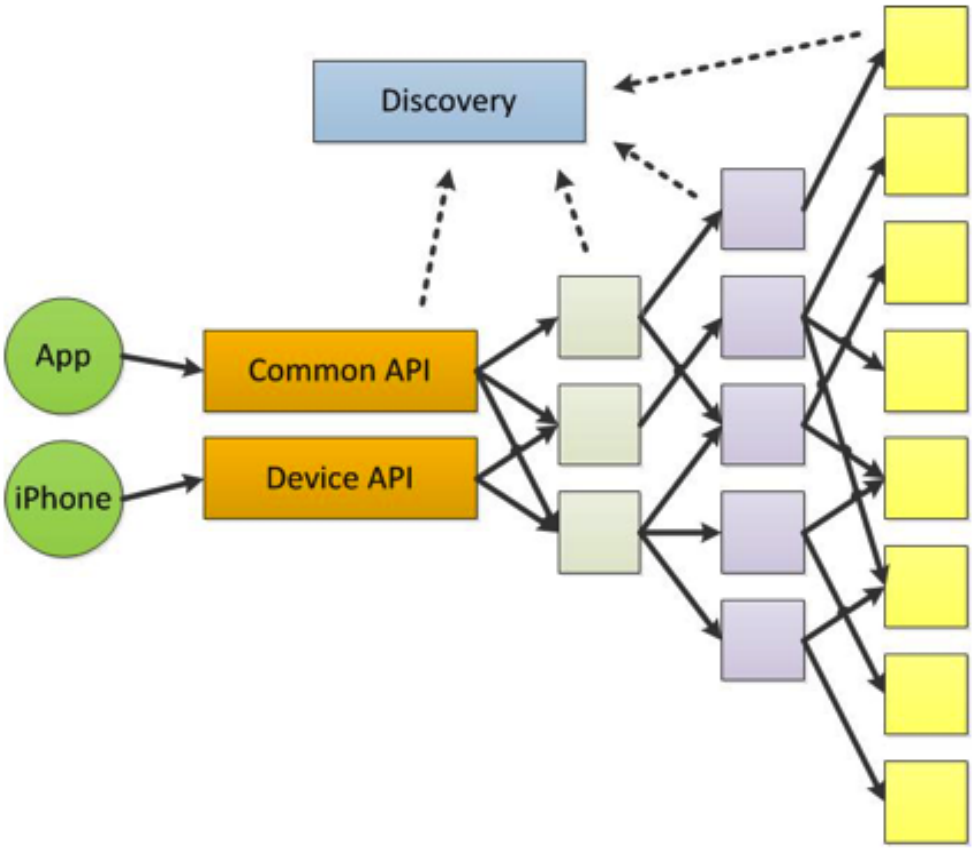

Рисунок 1.14 – Мікросервісна система

1.4 Аналіз програмних рішень для забезпечення відмовостійкості

Давайте розглянемо, як архітектура мікросервісів допомагає досягти живучості та відновлюваності всієї системи. Якщо один із екземплярів служби виходить з ладу, він перестає надсилати запити службі реєстру, яка, у свою чергу, відзначає її як непрацюючу та видаляє її зі списку служб. В цей час, коли інші служби запитують про цей сервіс, сервіс реєстру повертає інформацію лише про робочі екземпляри служби. Отже, при достатньому рівні надмірності відмова деяких екземплярів сервісу ніяк не вплине на роботу системи.

Усі попередні засоби гарантують живучість системи, але безпосередньо не впливають на відновлюваність. Для відновлення інфраструктури після збою використовуються системи, здатні генерувати нові екземпляри сервісів. До

таких систем належать Apache Mesos, Kubernetes та інші. Як правило, ці системи можуть відстежувати стан усіх екземплярів служби відповідно до частоти відгуку, рівня загруженості ЦП і пам'яті, а також запускати нові екземпляри необхідних служб або перезапускати екземпляри, які більше не відповідають. Таким чином, побудована інфраструктура може бути динамічно перебудована та адаптована до навантаження.

1.4.1 Apache Mesos

Apache Mesos Orchestrator – це диспетчер кластера, який забезпечує ефективну ізоляцію та спільне використання ресурсів між розподіленими застосунками. Mesos має відкритий вихідний код та спочатку був розроблений Каліфорнійським університетом у Берклі. Він знаходиться між прикладним рівнем та рівнем операційної системи, щоб спростити розгортання та управління у великому кластерному середовищі.

Mesos використовує можливості ядер сучасних операційних систем для ізоляції ЦП, пам'яті, введення– виведення, файлової системи, розташування стійки і т. д. (Рисунок 1.15).

Велика ідея – полягає у створенні великої кількості гетерогенних ресурсів. Mesos є розподіленим дворівневим механізмом планування під назвою Resource Proposals. Mesos вирішує, скільки ресурсів пропонувати кожному фреймворку, тоді як у фреймворк вирішує, які ресурси приймати і які обчислення на них виконувати.

Це тонкий шар розподілу ресурсів, який забезпечує точний обмін ресурсами між різними комп'ютерними системами кластера та надає системам загальний інтерфейс доступу до ресурсів кластера.

Багато сучасних робочих застосунків і фреймворків можуть працювати на Mesos, включаючи Hadoop, Ruby on Rails, Storm, JBoss Data Grid, MPI, Spark та Node.js, а також різні веб– сервери, бази даних та сервери додатків.

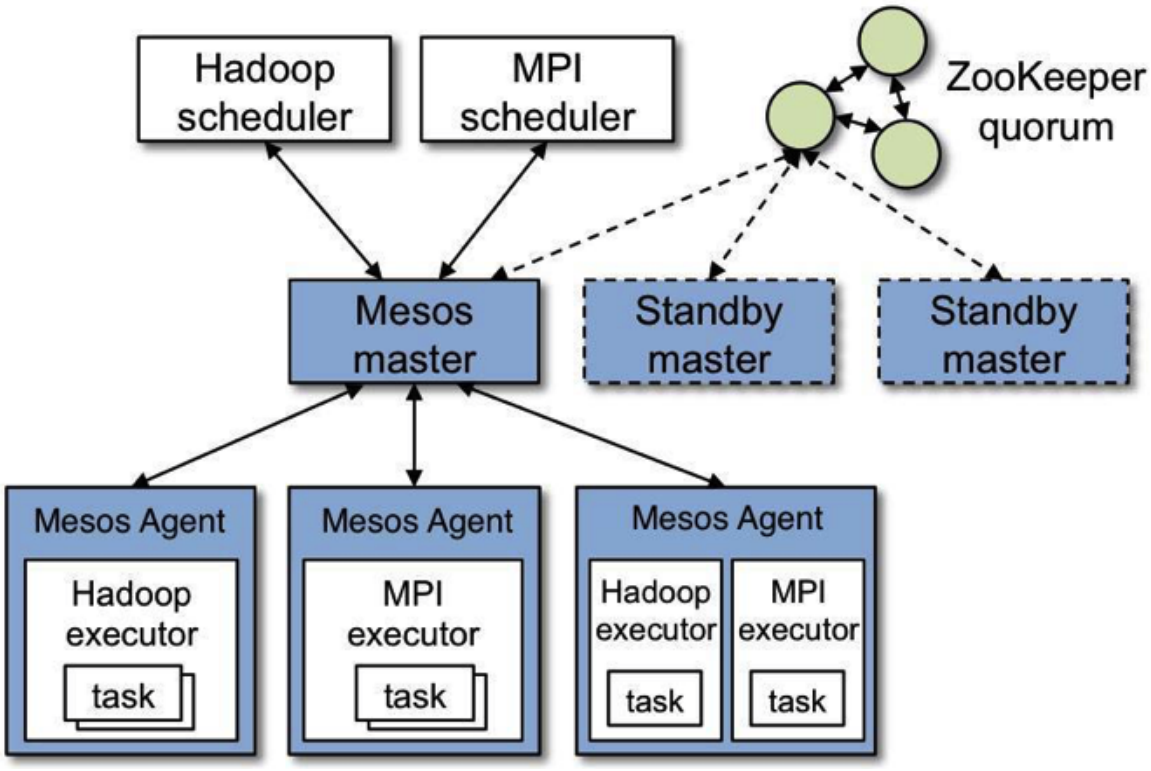

Рисунок 1.15 – Архітектура Apache Mesos

Для автоматичного масштабування Mesos може застосовувати показники використання CPU та оперативної пам’яті. Окрім того, Mesos дозволяє інтегрувати свої модулі для збору користувацьких метрик.

1.4.2 Kubernetes

Kubernetes Orchestrator – це система з відкритим вихідним кодом для автоматизації розгортання, масштабування та управління контейнерними додатками (рис. 1.16). Об'єднює контейнери, які складають додаток у логічні

блоки для полегшення управління та відкриття. Основні характеристики включають:

– надати розробникам програм потужний інструмент для оркестрування контейнерів Docker без взаємодії з базовою інфраструктурою;

– надати стандартний інтерфейс та примітиви для розгортання додатків та API– інтерфейсів у хмарах;

– створення систем на основі модульного API, що дозволяє клієнтам інтегрувати системи на основі базової технології Kubernetes.

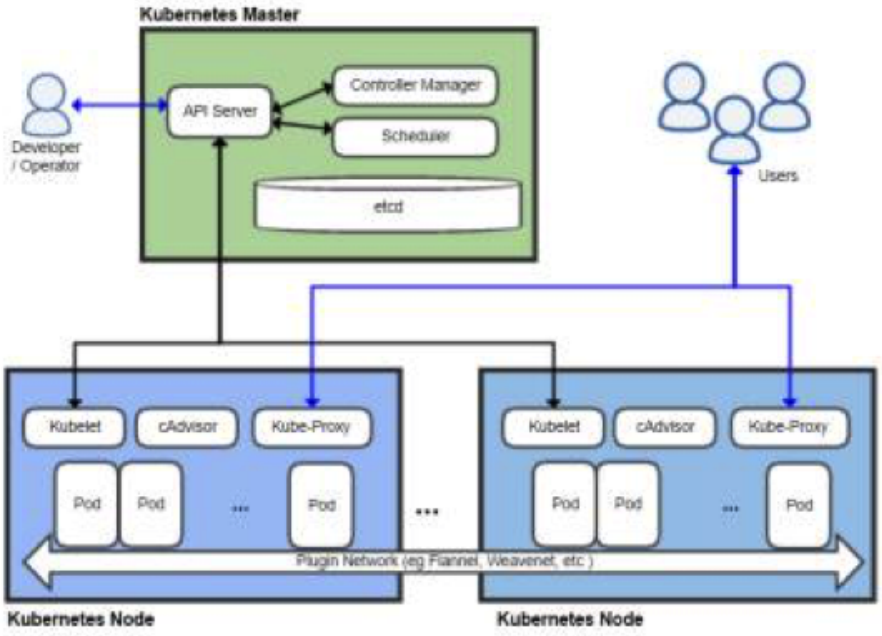

Рисунок 1.16 – Архітектура Kubernetes

Kubernetes дуже привабливий для розробників додатків, тому що він знизив їх залежність від інфраструктури та операційних груп. Основні переваги Kubernetes – надати розробникам додатків потужні інструменти для керування контейнерами Docker. Існує безліч ініціатив щодо масштабування проекту для відповідності більшій кількості робочих навантажень (наприклад, аналітика та державні служби обробки даних). Ці ініціативи все ще знаходяться на ранній стадії.

1.4.3 Docker Swarm

Як платформа Docker змінив спосіб доставки програмного забезпечення на сервери. Docker Swarm – це контейнерний фреймворк з відкритим вихідним кодом та є власним оркестратором кластерів для Docker.

Будь–яке програмне забезпечення, сервіс чи інструмент, що працює з контейнерами Docker, однаково добре працює зі Swarm. Крім того, Swarm використовує ту ж утиліту командного рядка, що й Docker Swarm перетворює набір хостів Docker на один віртуальний хост (рисунок 1.17). Swarm особливо корисний, коли простота розгортання дуже важлива.

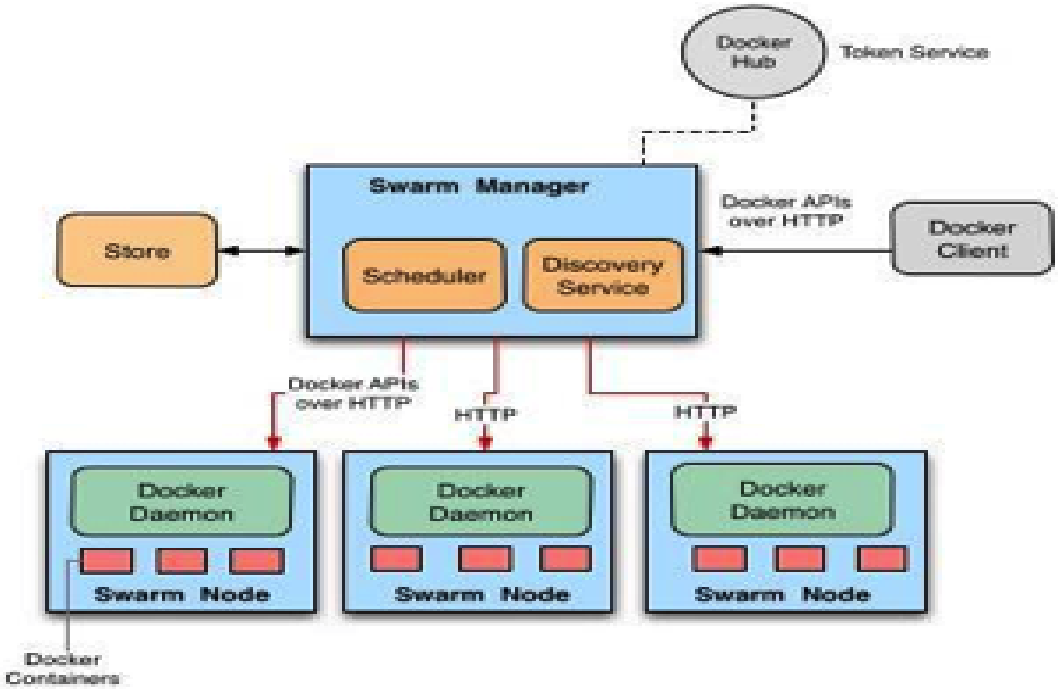

Рисунок 1.17 – Архітектура Docker Swarm

1.4.4 Порівняння оркестраторів

Порівняємо оркестратори за наступними критеріями:

– декларація програми;

– мережа;

– масштабованість;

– балансування навантаження;

– висока доступність.

Декларація програми

Apache Mesos – програму можна розгорнути під керівництвом планувальника. Для простих завдань використовується планувальник Maraphon, але для складніших завдань потрібно створити окремий планувальник для кожної служби;

Kubernetes–програма може бути розгорнута в Kubernetes за допомогою комбінації сервісів (або мікросервісів), розгортання та компонентів.

Docker Swarm – додатки можна розгортати як мікросервіси або служби у кластерній групі в Docker Swarm. Файли YAML можна використовувати для визначення декількох контейнерів.

Мережа

Apache Mesos – не створює віртуальних мереж та працює на мережному рівні між фізичними або віртуальними машинами, на яких встановлені агенти;

Kubernetes – мережна модель – це однорівнева мережа, яка дозволяє всім модулям взаємодіяти один з одним. Мережеві правила визначають, як вони взаємодіють одне з одним;

Docker Swarm – реалізує внутрішню віртуальну мережу, де кожен контейнер може бути підключений до певної мережі, а самі вони можуть бути підключені один до одного.

Користувачі можуть самі шифрувати трафік даних контейнера, створивши накладену мережу;

Масштабованість

Apache Mesos – повідомляє про вільні ресурси планувальникам, яким необхідно самостійно реалізувати логіку масштабування.

Kubernetes – для розподілених систем Kubernetes – це система загального призначення. Це складна система, оскільки вона забезпечує надійні гарантії працездатності кластера та уніфікований набір API.

Docker Swarm: у порівнянні з Kubernetes він може розгортати контейнер набагато швидше, що дозволяє швидше реагувати на попит. Але в той же час у ньому менше механізмів керування процесом розгорнення.

Висока доступність

Apache Mesos – з урахуванням підтримки масштабатора, він підтримує роботу кластера, але полягається на планувальник.

Kubernetes – всі екземпляри сервісів розподілені по вузлах для забезпечення високої доступності та стійкості до помилок. Служба балансування навантаження виявляє пошкоджені екземпляри та перезапускає їх. Таким чином підтримується висока доступність.

Docker Swarm – оскільки сервіси можуть дублюватись, Docker Swarm також забезпечує високу доступність. Вузли диспетчера Swarm відповідають за весь кластер та керують ресурсами робочих вузлів. У порівнянні з Kubernetes у ньому менше інструментів для моніторингу життя сервісів.

Балансування навантаження

Apache Mesos – не масштабує навантаження, оскільки служби обмінюються даними без втручання Mesos.

Kubernetes – модулі доступні через службу, яку можна використовувати для балансування навантаження у кластері. Це досягається тим, що кожну взаємодію зі службою контролює оркестратор.

Docker Swarm – надає внутрішній DNS, який можна використовувати для розповсюдження запитів на ім'я служби. Служби можуть призначатись автоматично або запускатися на вказаних користувачем портах. Варіантів

налаштування балансувальника навантаження менше, оскільки для балансування навантаження використовується лише DNS.

Висновки до розділу 1

Задача побудови відмовостійкої хмарної системи полягає в тому, що система повинна надійно працювати в умовах хмарної інфраструктури та гарантувати толерантність до помилок різного рівня. Система розробляється з можливістю масштабування за метриками користувача, так і з врахуванням метрики ресурсів .

Для реалізації стійкої до відмови хмарної системи потрібна безліч інструментів і підходів. Кожен компонент системи повинен мати можливість обробляти надлишкові ресурси, розташовані на різних фізичних пристроях, щоб він міг працювати навіть у разі відмови обладнання.

Система повинна підтримувати масштабованість, що дозволяє за необхідності змінювати кількість ресурсів, що використовуються.

Для оркестрації всі служби використовують спеціалізоване програмне забезпечення, яке забезпечує додатковий рівень абстракції порівняно з обладнанням у кластері. Apache Mesos – дуже гнучкий інструмент, що дозволяє реалізувати велику кількість різних функцій, але дуже низькорівневий і вимагає написання додаткових планувальників і фреймворків. Він працює для будь– якої програми, яка може бути або контейнерами Docker або звичайними завданнями, що виконуються на машинах за розкладом. Kubernetes та Docker Swarm працюють тільки з контейнерами, але їх легко налаштувати та використовувати. Kubernetes є більш гнучким і складним у використанні, тоді як Docker Swarm пропонує просте рішення, яке швидко налаштовується.

РОЗДІЛ 2. МЕТОДИ РЕАЛІЗАЦІЇ СИСТЕМИ

2.1 Середовище розробки та мова програмування

Основним середовищем розробки було IntelliJ IDEA Ultimate. Для запуску програми використовувався контейнерний інструмент Docker.

Для реалізації серверної частини використовувалися Java 10 та Spring Framework. Веб– клієнт реалізований за допомогою Typescript, фреймворку Angular та інструментів верстки HTML та CSS.

Реалізацію аутентифікації побудовано з використанням Javascript Web Token (JWT).

Створення великих продуктів розробки практично неможливе без використання спеціальних інтегрованих середовищ розробки (IDE), які спрощують роботу над проектом. Крім можливостей звичайного текстового редактора, IDE спрощують роботу з системами контролю версій, зв'язок із базою даних, інтеграцію з фреймворками та інструментами для побудови та тестування систем.

Існує три основні IDE для Java: IntelliJ IDEA, NetBeans та Eclipse. NetBeans та Eclipse безкоштовні та підтримують майже всі основні платформи та бібліотеки.

IntelliJ IDEA – це платне середовище, яке, крім основних функцій інших IDE, стабільніше, має кращі пошукові системи в дизайні, широкий спектр додаткових високоякісних плагінів і зручні методи рефакторингу коду. Студенти можуть отримати IntelliJ IDEA абсолютно безкоштовно.

Тому весь код написано в інтегрованому середовищі розробки IntelliJ IDEA для різних мов програмування JetBrains.

Основні особливості IntelliJ IDEA:

– розумне автозаповнення, інструменти аналізу якості коду, проста навігація, покращена за допомогою рефакторингу та форматування для Java, Groovy, Scala, HTML, CSS, JavaScript, CoffeeScript, ActionScript, LESS, XML та багатьох інших мов;

– підтримка всіх популярних фреймворків та платформ, включаючи Java EE, Spring Framework, Grails, Play Framework, GWT, Struts, Node.js, AngularJS, Android;

– інтеграція з серверами додатків, включаючи Tomcat, TomEE, GlassFish, JBoss, WebLogic, WebSphere, Geronimo, Resin, Jetty та Virgo;

– інструменти для роботи з базами даних та файлами SQL, включаючи зручний клієнт та редактор схеми бази даних;

– інтеграція із системами управління комерційними версіями Perforce, Team Foundation Server, ClearCase, Visual SourceSafe;

– інструменти тестування та аналізу покриття коду, включаючи підтримку всіх популярних фреймворків тестування.

IntelliJ IDEA включає модуль візуального дизайну для програм із графічним інтерфейсом.

Інтерфейс Swing UI Designer, редактор XML, редактор регулярних виразів, система перевірки коду, система управління завданнями та надбудови для імпорту та експорту проектів з Eclipse. Інструменти інтеграції доступні з системами відстеження помилок JIRA, Trac, Redmine, Pivotal Tracker, GitHub, YouTrack та Lighthouse.

– інструменти тестування та аналізу покриття коду, включаючи підтримку всіх популярних фреймворків тестування;

Однією з переваг мікросервісної архітектури є свобода вибору бажаної мови яка використовується при створенні програм і навіть дозволяє комбінувати декілька мов у різних сервісах між собою. Але є ще більш

підходящі мови для вирішення конкретних завдань. Для розробки сервісу була обрана мова програмування Java з низки причин.

Java – це мова програмування загального призначення, яка слідує парадигмі об'єктно– орієнтоване програмування та «Написаний час може працювати де завгодно». Java дуже популярна у веб–додатках та корпоративних додатках.

Java – це не тільки мова програмування, а й екосистема інструментів, які:охоплює практично все, що вам знадобиться для програмування. Включає в себе: Java Development Kit – це комплект для розробки Java. З його допомогою ви можете писати, компілювати та запускати Java– код.

Давайте подивимося на основні переваги Java:

Java включає об'єктно– орієнтоване програмування – концепцію, в якій ви визначаєте як тип даних та його структуру, а й усі функції, застосовується до цього. Таким чином, структура даних стає об'єктом, який можна вдається створювати відносини між різними об'єктами. Коли ви використовуєте процедурний підхід, у вас є лише можливість писати інструкції, визначати змінні чи функція. ООП дозволяє групувати ці об'єкти разом із допомогою контексту, позначте кожен із них і звертайтеся до нього як до окремого об'єкта.

У чому основні переваги ООП:

– за допомогою ООП ви можете повторно використовувати об'єкти в інших програмах;

– ООП передбачає помилки, тому що об'єкти приховують інформацію, до якої доступу не повинно бути;

– ООП більш ефективно організує структуру програм, зокрема великий;

– ООП полегшує модернізацію та підтримку застарілого коду;

Java створювалася як мова розподіленого програмування: в ньому є вбудований механізм для обміну даними та додатками між різними комп'ютерами, що підвищує продуктивність та ефективність роботи. Інші

мови повинні використовувати зовнішній API для розподіленого обміну даними. Java має цю інтегровану технологію. Методологія розподілених обчислень, специфічна Java, називається віддаленим викликом методу (RMI). RMI дозволяє використовувати всі переваги Java: безпеку, незалежність від платформи та об'єктно– орієнтоване програмування для розподіленої обробки. Крім того, Java також підтримує програмування сокетів та методологію розповсюдження CORBA для програмного забезпечення обміну об'єктами між програмами, написаними різними мовами.

2.2 Інструменти для збірки проекту

Системи на основі Java можуть бути побудовані за допомогою двох популярних систем: Maven та Gradle. Maven – інструмент, що найчастіше використовується, і має безліч функцій, але використовує XML для налаштування проектів, що робить всю конфігурацію досить великою та їх важко читати. Крім того, щоб створювати складніші алгоритми складання проектів, вам потрібно написати безліч конфігурацій та плагінів.

Для складання проекту використовувався Gradle – інструмент складання з упором на автоматизацію складання та підтримка розробки різними мовами програмування. Gradle пропонує гнучку модель, яка може підтримувати весь цикл розробки, від збирання та пакування коду до публікації веб– сайтів. Gradle був розроблений для забезпечення автоматизації складання кількома мовами та платформами, включаючи Java, Scala, Android, C/C++ та Groovy і тісно інтегрований із засобами розробки та сервери безперервної інтеграції, включаючи Eclipse, IntelliJ та Jenkins. Нижче наведено список творців функцій Gradle:

– в основі Gradle лежить предметна мова (DSL), що розширюється, на основі Groovy. Gradle виводить декларативну побудову проекту на новий

рівень, надаючи декларативні мовні елементи, які ви можете застосовувати на власний розсуд.

Ці елементи також забезпечують підтримку створення угод для проектів Java , Groovy, Web та Scala. Крім того, ця декларативна мова є розширеною. Ви можете додавати власні мовні елементи або покращувати існуючі, надаючи короткі чисті збірки, які легко підтримувати;

– незважаючи на новаторський підхід, Gradle полегшує перехід з інших систем збирання проектів, таких як Maven та Ant;

– скрипти збору Gradle написані на Groovy, а не XML. Це дозволяє використовувати повноцінну динамічну мову для опису ваших колекцій, що забезпечує безпрецедентну гнучкість і стислість для всіх скриптів;

– вам не потрібно встановлювати її самостійно, щоб використовувати Gradle. Gradle дозволяє використовувати оболонку, що настроюється, яка скорочує кількість необхідних залежностей і спрощує створення служб безперервної інтеграції (CI).

2.3 Засоби розгортки програмного продукту

В даний час існує два основних способи розгортання вашої системи: робити це за допомогою різних скриптів або використовувати сучасні інструменти контейнеризації. На відміну від першого методу, контейнери ізолюють послуги один від одного і полегшують процес повторного розгортання. Тому для проекту було обрано метод розгортання із використанням контейнерів.

Docker – провідна програмна платформа для роботи з контейнерами. Розробники використовують Docker для усунення несправностей у середовищі під час спільної роботи над кодом з колегами. Адміністратори використовують

Docker для запуску та керування програмами, які працюють разом в окремих контейнерах для кращої сумісності. Компанії використовують Docker для створення гнучких методів розповсюдження програмного забезпечення, щоб надавати нові можливості швидше, надійніше та безпечніше для розгортань як Linux, так і Windows Server.

З контейнерами все, що потрібно для запуску певної частини вашої програми, упаковано в ізольований контейнер. На відміну від віртуальних машин, на контейнерах не працює повноцінна операційна система, для роботи програмного забезпечення потрібні лише бібліотеки та налаштування.

Це дозволяє створювати ефективні, легкі, автономні системи та забезпечує узгоджене операційне середовище незалежно від того, де система розгорнута. Docker автоматизує повторювані завдання з налаштування та налаштування середовища розробки, щоб розробники могли зосередитися на найважливішому: розробці програмного забезпечення (Рис 2.1)..

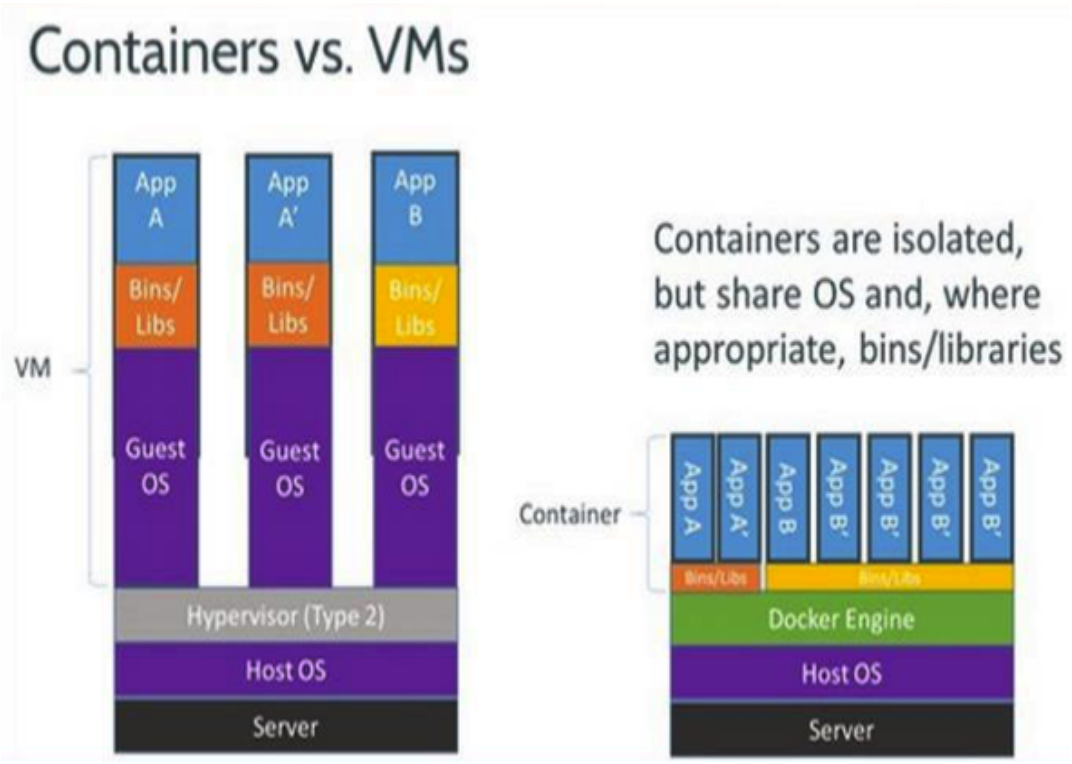

Рисунок 2.1 – Порівняння віртуальної машини та контейнера

Розробникам що користуються Docker не потрібно встановлювати та налаштовувати складні бази даних та турбуватися про перемикання між несумісними версіями середовища виконання. Коли програма перетворюється на образ Docker, складність переноситься в контейнери, які легко збираються, які спільно використовуються командою розробників і працюють у різних середовищах. З кодом, який постачається з файлом Dockerfile, який містить інформацію про складання образу, простіше працювати.

Залежно завантажуються в упаковані образи Docker, і будь–хто може створити та запустити програму за лічені хвилини.

2.4 Технології для розробки інтерфейсу

Для розробки інтерфейсу можна використовувати один із багатьох фреймворків JavaScript: Vue, React, Angular та інші. Для цього програмного продукту було обрано Angular 6, оскільки він дозволяє писати веб– клієнтів на друкує мовою TypeScript та забезпечує інтелектуальну структуру дизайну, що дозволяє клієнтам писати швидше, а потім легко додавати нові функції без значних змін в інших частинах програми.

Крім того, для створення клієнтської частини системи використовувалися такі технології, як HTML5, CSS3 та Bootstrap 4, що автоматизує створення інтерфейсу та полегшує доступ до різних частин веб–сайту.

2.4.1 Фреймворк Angular 6

Оновлений Angular 6 визначає структуру проекту, включає безліч вбудованих блоків, що охоплюють найпопулярніші сценарії веб–розробки.

Також оновлений Angular 6 задає фреймворк проекту, включає безліч вбудованих блоків, що охоплюють найбільш популярні сценарії при створенні веб– програми. Також існує чіткий поділ ролей окремих елементів:

Сервіси – це елементи, які використовують дані API або поділяють стан між кількома компонентами;

Компоненти – це будівельні блоки інтерфейсу користувача, що використовує служби;

Вони можуть бути вкладені за допомогою селекторів структурних директив.

Директиви – поділені на структурні та атрибутні директиви. Структурні директиви (такі як * NgFor) впливають на DOM (об'єктну модель документа), тоді як директиви атрибутів є частиною елементів і керують їх стилем та станом;

Трубки використовуються для форматування того, як дані відображаються в області перегляду.

Пайпи (Pipes) – використовуються для форматування того, як відображаються дані у вікні перегляду;

Модулі (Modules) – експортоспроможні блоки додатку, які ізолюють компоненти, директиви, пайпи, сервіси і маршрути разом;

2.4.2 Мова TypeScript

Фреймворк написаний на TypeScript (TS), це мова– надбудова над JavaScript, розроблена компанією Microsoft, яка здатна компілюватись у чистий JavaScript (рисунок 2.2)

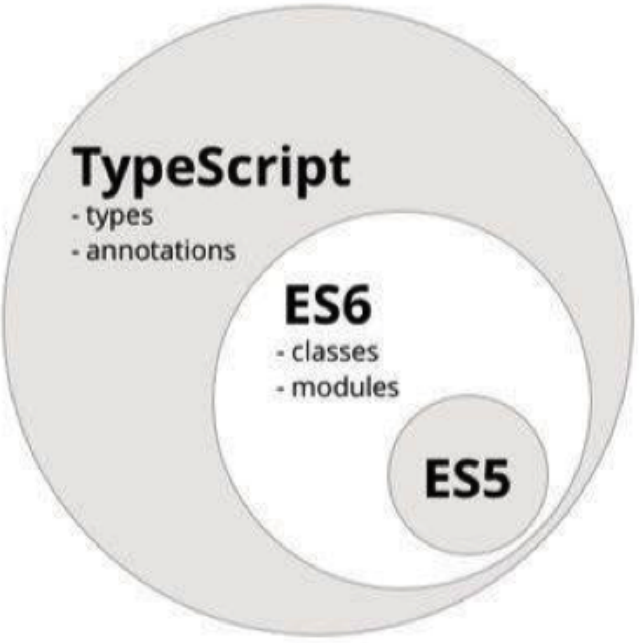

Рисунок 2.2 – Можливості TypeScript у порівнянні з JavaScript

У мові представлені нові типи, нові структури даних та більш об'єктно–орієнтована функціональність, що полегшує читання коду в порівнянні з чистим JavaScript. Спочатку синтаксис TypeScript був навмисно схожий на C#. Однак нещодавно TypeScript використовував стрілку (=>) та інший синтаксис, аналогічний ES6, що робить TS більш схожим на ES6. Зворотною стороною використання TypeScript є додаткова складність під час налаштування програми.

2.4.3 Структура інтерфейсу

Angular 6 використовує мову розмітки гіпертексту (HTML), мову розмітки документів, для побудови структури інтерфейсу. Мова розмітки для гіпертекстових документів HTML дозволяє визначати різні типи елементів, які забезпечують функціональність документа: фрагменти тексту із заданими параметрами форматування, списки, таблиці, зображення, гіперпосилання

тощо. буд. Елементи HTML оголошуються з допомогою команд розмітки, званих тегами. Усі теги HTML, виявлені у тексті документа, інтерпретуються браузером під час відображення документа.

Для стилізації розмітки використовуються каскадні таблиці стилів CSS – спосіб опису зовнішнього вигляду документа, написаного мовою розмітки. CSS використовується для встановлення кольорів, шрифтів, макету та інших аспектів представлення документа.

Основна мета розробки CSS полягала в тому, щоб відокремити контент, написаний HTML або іншою мовою розмітки, від представлення стилю документа. Такий поділ може підвищити доступність документа, дати більшу гнучкість та контроль над його зовнішнім виглядом, а також зменшити складність та повторення структурованого вмісту. Крім того, CSS дозволяє представляти один і той же документ у різних стилях.

Ви можете спростити реалізацію адаптивного дизайну за допомогою Bootstrap.

Bootstrap – це найпопулярніший набір інструментів HTML, CSS та JS для розробки адаптивних веб– сайтів та веб– застосунків, призначених для роботи на мобільних пристроях. Цей проект спрощує розробку адаптивних веб– сайтів та веб– додатків.

2.5 Технології для розробки серверної частини

Зазвичай для розробки систем Java використовується Java EE (Java Enterprise) або Spring Framework. Хоча Spring Framework не надавала будь–якої конкретної моделі програмування, вона набула широкого поширення Java в основному як альтернатива і заміна моделі Enterprise JavaBeans. Spring Framework дає розробникам Java велику свободу проектування та надає добре документовані та прості у використанні інструменти для вирішення проблем, що виникають при створенні програм промислового масштабу.

Spring Framework – це платформа розробки Java, яка забезпечує наскрізну підтримку інфраструктури для розробки систем. Spring Framework управляє інфраструктурою залежностей, щоб розробник міг зосередитись на бізнес– логіці системи.

Програми Java зазвичай складаються з об'єктів, які працюють для виконання певної логіки програми. Таким чином, об'єкти у додатку залежать один від одного.

Хоча платформа Java надає безліч функцій розробки додатків, їй не вистачає ресурсів для організації основних будівельних блоків в єдине ціле, залишаючи це на розсуд архітекторів та розробників. Розробник може використовувати шаблони проектування, такі як Factory, Abstract Factory, Builder, Decorator та Service Locator, для створення різних класів та об'єктів, що становлять програму. Однак ці моделі просто називають передовими методами розробки програмного забезпечення, описуючи, що вони роблять, де їх застосовувати, проблеми, які вони вирішують, і таке інше. Шаблони – це формалізовані передові практики, які можна застосувати у програмі.

2.6 Взаємодія між клієнтом і сервером

Принцип взаємодії передачі репрезентативного стану (REST) був застосований до взаємодії між клієнтом та сервером. Це стиль проектування розподілених систем із обмеженнями. Центральна абстракція у REST – це ресурс. І основні обмеження такі:

– модель клієнт– сервер;
– взаємодія без громадянства;
– логічний інтерфейс.

Коли ви заходите на будь– який сайт, браузер (клієнт) надсилає GET– запит на сервер. Сервер приймає запит і надсилає відповідь. Браузер приймає та відображає отриману інформацію.

У моделі клієнт– сервер сервер надає деякі послуги або ресурси, які клієнти одержують, надсилаючи запити. Причому клієнт може бути будь– ким:

Програма Android, браузер або банкоматом (рисунок. 2.3).

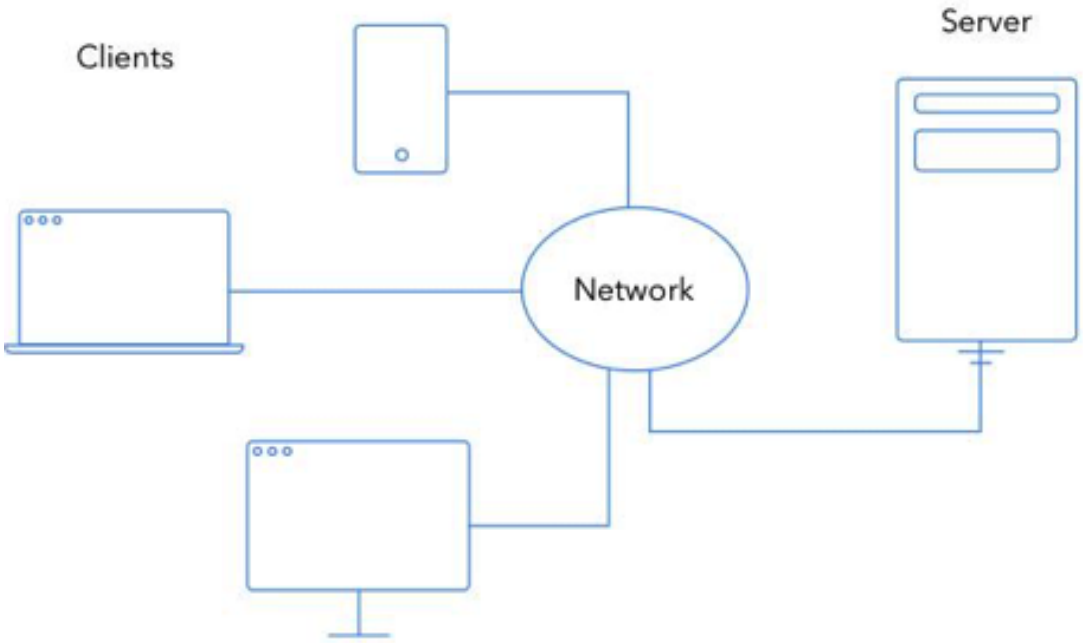

Рисунок 2.3 – Взаємодія між сервером і клієнтами

Перевага цього принципу полягає в тому, що він підтримує розподіл інтересів. Завдяки цьому сервер може бути однією машиною, а клієнти можуть бути різними.

В архітектурі REST серверу не потрібно зберігати будь– яку інформацію про стан операції. Сесії мають зберігатися у клієнта. Це означає, що якщо сервер отримує два різні запити від одного і того ж клієнта, вони не повинні заважати один одному. Отже, клієнт повинен надіслати всю інформацію, необхідну для негайного виконання дії (рисунок 2.4)

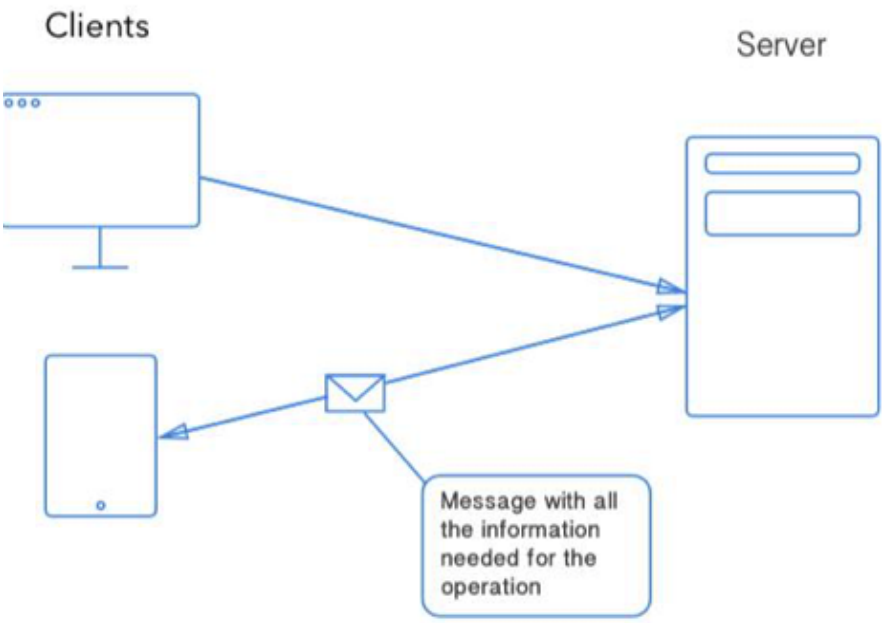

Рисунок 2.4 – Передача необхідної інформації

Такий підхід економить багато часу та ресурсів та спрощує масштабування сервера.

Не повинно бути функції входу/виходу. Натомість має бути роль, яка на основі даних авторизації повертає користувачеві токен, який він повинен додавати до кожного запиту.

Ресурс – це уявлення віртуального об'єкта (наприклад, зображення), реального об'єкта (покупця) чи набору об'єктів. Загалом, функція може бути будь– чим, розробник API вирішує, що вона матиме функцію.

При написанні RESTful API для служби вивчення слів ресурсом можуть бути слова, набори слів, користувачі, режими навчання та багато іншого.

У REST кожен ресурс має бути унікальним. Тому їм надається ідентифікатор. Для слів / words / 1 та / words / 2 це два різні слова від ID 1 та 2. Ресурси також можуть бути вкладеними. Наприклад, URI слова 2 у першому наборі слів виглядатиме так

/ sets / 1 / words / 2.

Більшість запитів до API будуть на виконання CRUD – операцій.Можна провести аналогію між SQL та HTTP.

Таблиця 2.1 – CRUD операції для SQL та HTTP

Операція	SQL	HTTP
Створення	INSERT	POST
Читання	SELECT	GET
Оновлення	UPDATE	PUT/PATCH
Видалення	DELETE	DELETE

Приклади використання API:

– щоб отримати слово 2, клієнт повинен виконати GET /words/2;

– щоб створити нове слово, клієнт повинен виконати POST /words.

Узагальнюючи огляд методів реалізації, середовище розробки було

Використовувалася IntelliJ IDEA, Gradle – для складання проекту, а Docker – для розгортання системи.

Інтерфейс був написаний на TypeScript з використанням Angular 6, а інтерфейс створено з використанням Bootstrap .

2.7 Масштабування баз даних

Класична схема роботи програми з базою даних включає екземпляр сервісу і екземпляр бази даних. Така схема роботи з базою даних не допускає

масштабованості та робить базу даних єдиною точкою відмови системи. Масштабованість даних ґрунтується на тому ж принципі, що й масштабованість веб– додатків: це поділ даних на групи та їх призначення на окремі сервери. Є дві основні стратегії: реплікація та сегментування(шардінг).

Описані нижче схеми масштабування можна застосовувати як до реляційних баз даних, так і до архівів NOSQL. Зрозуміло, що всі бази даних та архіви мають свою специфікацію, тому ми розглянемо лише основні напрямки.

2.7.1 Партиціювання (partitioning)

Партиціювання складається з поділу таблиць, що містять велику кількість записів, на логічні частини на основі критеріїв, вибраних адміністратором. Партиціювання таблиць розділять весь обсяг операцій обробки даних на кілька незалежних та виконуваних потоків, що значно прискорює роботу СУБД. Для правильного налаштування параметрів поділу необхідно, щоб у кожному потоці була приблизно однакова кількість записів.

Наприклад, на сайтах новин має сенс розбивати записи за датою публікації, тому що останні новини на кілька порядків затребувані і частіше доводиться працювати з ними, а не з усім архівом за роки існування ресурс новин.

2.7.2 Реплікація (replication)

Реплікація – це синхронне або асинхронне копіювання даних між декількома серверами (рисунок 2.5).

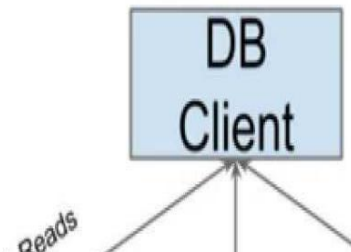

Рисунок 2.5 – Реплікація бази даних

Керуючі сервери називаються провідними (master), а керовані – слейвами(slave). Майстри використовуються для зміни даних, а слейви – для читання. У класичній схемі реплікації зазвичай використовується один майстер і кілька слейвів, оскільки у більшості веб– проектів операцій читання значно більше, ніж операцій запису. Однак у складнішій схемі реплікації може бути кілька майстрів.

Наприклад, створення декількох додаткових slave– серверів дозволяє зняти навантаження з головного сервера і підвищити загальну продуктивність системи, а також ви можете організувати підпорядковані пристрої для конкретних ресурсомістких завдань і таким чином, наприклад, спростити підготовку серйозних аналітичних звітів – використовуваний для цих цілей slave може бути завантажений на 100%, але це не вплине на роботу інших користувачів програми.

2.7.3 Фрагментація (sharding)

Фрагментація – це метод, що дозволяє розподіляти дані між різними фізичними серверами.

Процес фрагментації включає розподіл даних між окремими фрагментами на основі деякого ключа фрагментації (рисунок 2.6).

Об'єкти, пов'язані з тим самим значенням ключа блоку, групуються в набір даних за заданим ключем, і цей набір даних зберігається у фізичному блоці. Це значно спрощує обробку даних.

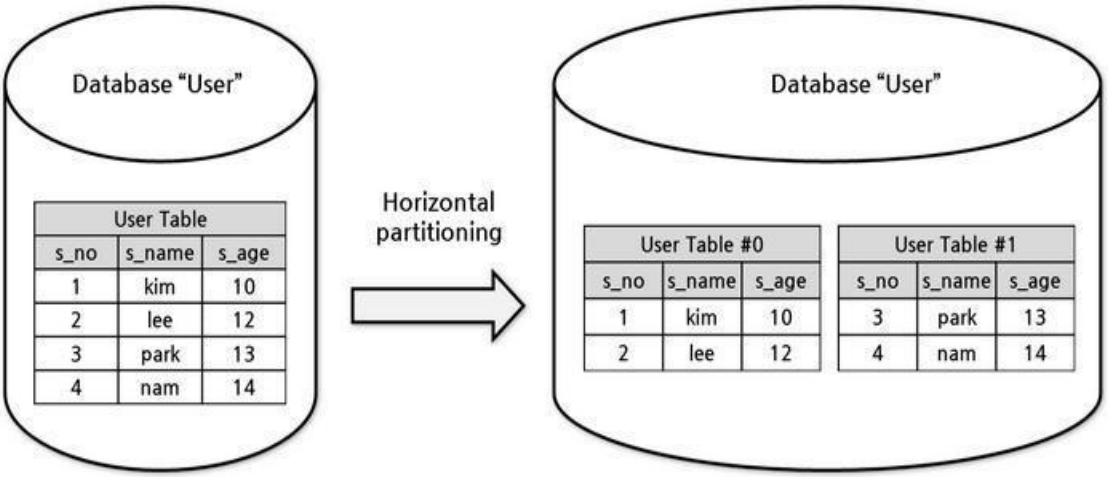

Рисунок 2.6 – Фрагментація бази даних

Наприклад, у таких системах, як соціальні мережі, ключем до фрагментації може бути ідентифікатор користувача, тому всі дані користувача будуть зберігатися і оброблятися на одному сервері, а не збиратися частинами з декількох.

2.8 Автоматичне відновлення даних

Комбінація описаних стратегій масштабування може підвищити надійність та відмовостійкість систем. Після об'єднання описаних стратегій всі дані розбиваються на невеликі частини, що повторюються, і переносяться на різні сервери (рисунок 1.7).

У разі відмови одного із серверів можна запустити новий сервер бази даних і передати втрачені фрагменти даних до нової служби бази даних. Більшість сучасних баз даних підтримують автоматичну фрагментацію, реплікацію та відновлення стану після відмов.

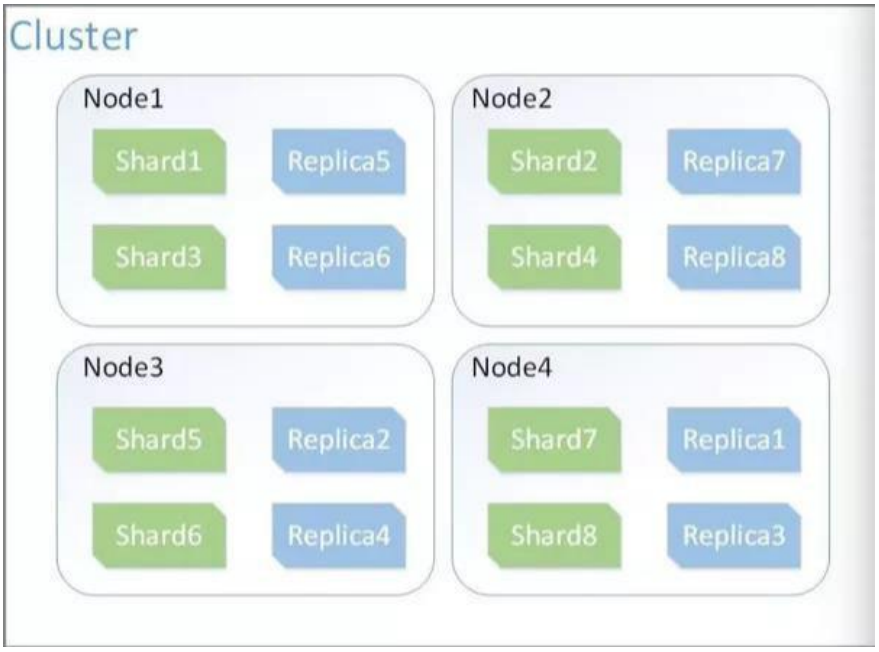

Рисунок 2.7 – Розміщення фрагментів по серверам

У разі відмови одного із серверів можна запустити новий сервер бази даних і передати втрачені фрагменти даних до нової служби бази даних. Більшість сучасних баз даних підтримують автоматичну фрагментацію, реплікацію та відновлення стану після відмов

Висновки до розділу 2

Для розробки клієнтської частини був використаний фреймворк Angular 6, а для розробки серверної частини був використаний фреймворк Spring 5. Взаємодія між клієнтом і сервером відбувається за допомогою REST, що дозволяє виконувати запити через HTTP.

Для розгортання системи був використаний Docker, що дозволило для кожного сервісу створити окремий образ та розгортати сервіс разом з усіма необхідними залежностями у будь–якому середовищі.

РОЗДІЛ 3. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Реалізація мікросервісних підходів

Основна розробка програмного продукту поділяється на 2 частини: написання серверу та веб клієнта для браузера. Сервер написаний на мові Java 10 з використанням Spring Framework. Клієнтська частина написана на мові Typescript з використанням фреймворку Angular 6 та платформи Node для запуску веб– серверу.

Веб клієнт взаємодіє з сервером за допомогою REST, що дозволяє зменшити зв’язок між обома компонентами до мінімуму.

Мікросервісна архітектура складається з багатьох маленьких сервісів, кожен з яких динамічно змінювати кількість своїх екземплярів. Це обумовлює необхідність реалізації підходів, що дозволяють системі продовжувати працювати у такому динамічному середовищі. До цих підходів відносяться: реєстр с ервісів, наявність єдиного серверу конфігурації, балансування навантаження, механізми повторів та переривання запитів. Для реалізації підходів були використані бібліотеки з проекту Spring Cloud: Eureka, Zuul, Hystrix, Ribbon, Config Server.

3.2 Реєстр сервісів

Використання Eureka дозволило створити сервіс, що зберігає реєстр всіх сервісів та їх екземплярів. Якщо з’являється новий сервіс, екземпляр сервісу, або навпаки, екземпляр сервісу перестає відправляти повідомлення про свій стан – дані в реєстрі оновлюються.

Окрім того, Eureka надає інтерфейс адміністратора для перегляду інформації про всі зареєстровані сервіси.

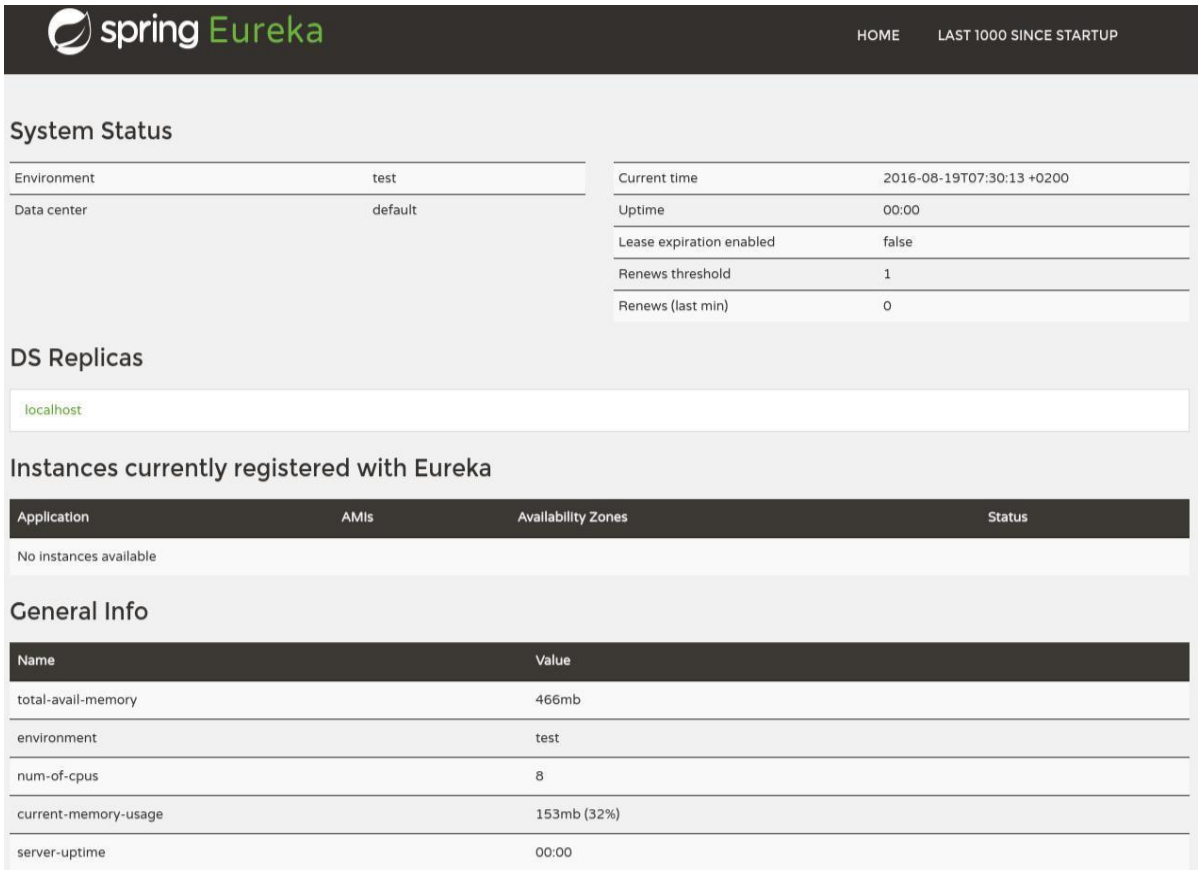

Рисунок 3.5 – Реєстр сервісів Eureka

Для взаємодії з Eureka– сервером кожен клієнт має використовувати eureka– client та сконфігурувати місцезнаходження серверу. Таким чином, кожні 30 секунд всі клієнти повідомляють інформацію про свій стан до сервера. Якщо якість екземпляри сервісів не повідомили про свій стан, то вони вважаються вимкненими.

3.3 Сервер конфігурації

Оскільки кожна служба може мати велику кількість екземплярів, і може виникнути ситуація, коли конфігурацію необхідно оновити у всіх екземплярах

служби, для вирішення цієї проблеми можна використовувати один сервер конфігурації. Так, кожен екземпляр служби повинен перейти на сервер конфігурації та отримати всі необхідні настройки.

Щоб використовувати сервер конфігурації, необхідно використовувати бібліотеку config– server– client і налаштувати розташування сервера. Шлях до сервера може бути як статичним, так і динамічним, використовуючи ім'я сервера та реєстр служб.

3.4 Взаємодія сервісів один з одним

При доступі до служби спочатку потрібно з'ясувати, де знаходяться екземпляри цієї служби. Для цього використовувався описаний реєстр сервісів Eureka. На запит на реєстр надається інформація про місцезнаходження всіх примірників служби. Оскільки кожна служба може мати кілька екземплярів, дуже важливо рівномірно розподілити навантаження по всіх екземплярах служби.

Для цього на стороні клієнта було застосовано балансування навантаження. Крім того, HTTP– запити можуть бути відкинуті через мережу, тому реалізований механізм повторної спроби невдалих запитів. Бібліотека Ribbon використовувалася для балансування завантаження на стороні клієнта та повторних запитів.

Може виникнути ситуація, коли одна із служб нижнього рівня перестала відповідати. Таким чином, замовник чекатиме завершення запиту кожної з послуг на шляху до запитаної та отримуватиме помилки про тривалий час очікування. Таким чином, користувач довго спостерігатиме за виконанням запиту.

Для поліпшення взаємодії із системою реалізовано підхід, що дозволяє зупиняти невдалі запити, які можна здогадатися заздалегідь. Коли точка

доступу повертає помилки на 90% запитів протягом однієї хвилини, вона закривається клієнтом на одну хвилину. Протягом цієї хвилини 1% запитів надсилається до точки доступу. Якщо цей 1% запит продовжує отримувати помилки, точка доступу залишиться закритою. Але якщо цей 1% запит буде успішним, точка доступу відкриється знову.

Схема роботи вимикача навантаження показана (рис 3.6.) Для цього підходу використовувалася бібліотека Hystrix.

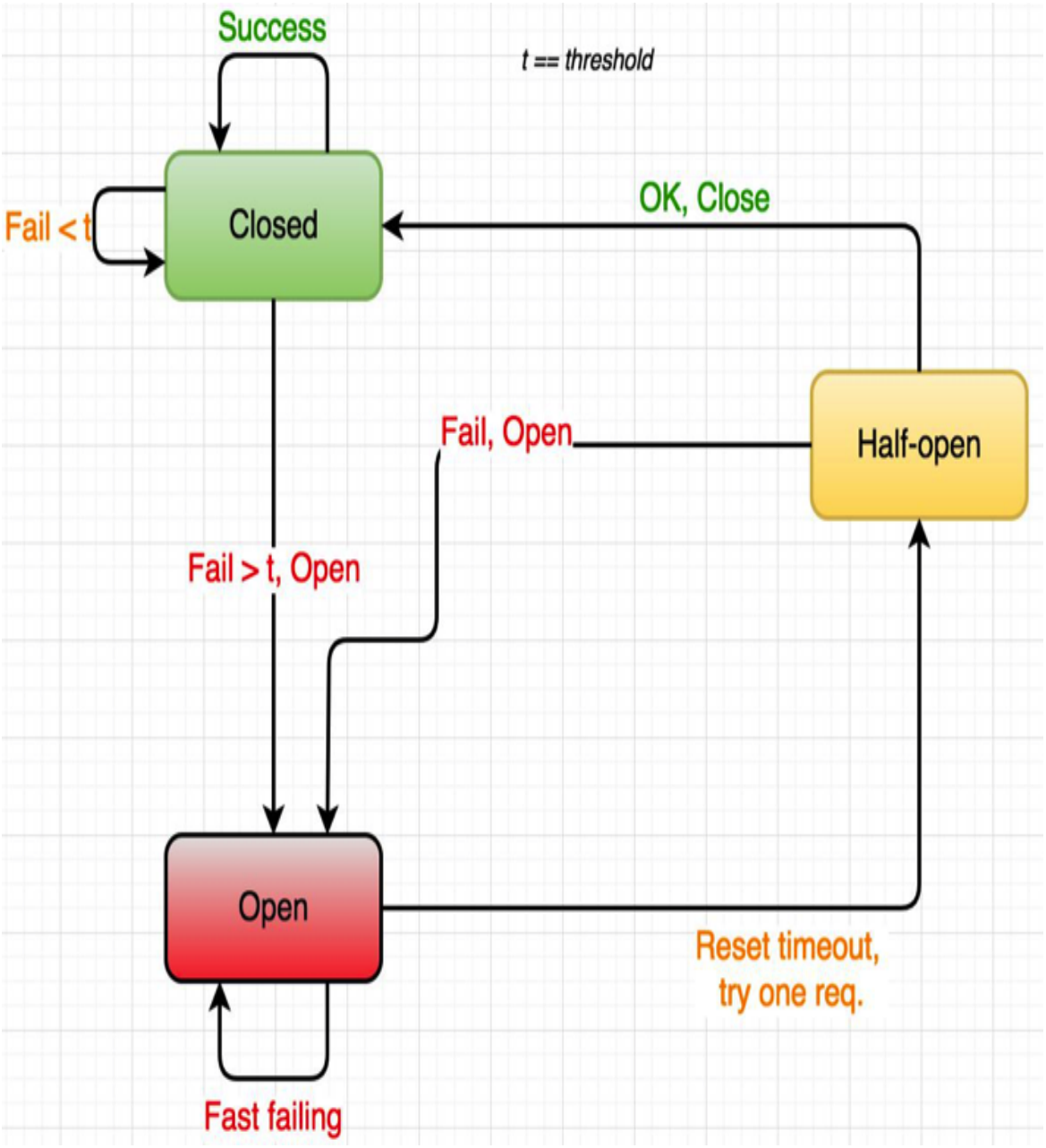

Рисунок 3.6 – Діаграма роботи Hystrix

Ведення журналу служби та необхідність балансування навантаження можуть ускладнити роботу сторонніх клієнтів із системою, тому для спрощення загальнодоступного API використовувався шлюз API.

Запити від усіх сторонніх клієнтів проходять через цей шлюз, і шлюз вже зв'язується з реєстром служб та балансує навантаження в екземплярі служби.

Бібліотека Zuul використовувалася для реалізації API шлюзу.

3.5 Точка доступу для повернення метрики послуги

Бібліотека Spring Actuator використовувалася для реалізації точки доступу, яка повертає інформацію про показники обслуговування.

Стандартна реалізація повертає базу такі показники як загальна кількість запитів. Щоб додати довільні метрики обслуговування, вам необхідно створити Spring Bean, наприклад, Gauge або Counter, де gauge повертає поточне значення деякої метрики, такої як довжина черги, а лічильник підраховує кількість певних подій і характеризує загальне значення для весь період експлуатації сервісу, наприклад кількість помилок. при обробці певних запитів.

Оскільки для автоматичного масштабування необхідно буде застосувати метрику, що характеризує поточне значення довжини черги повідомлень, яку необхідно обробити, було створено метрику типу Gauge.

Для отримання поточного значення довжини черги використовувався RabbitAdmin, який дозволяє отримати цільову інформацію щодо конкретної черги.

3.6 Робота з базою даних

Використовується реляційна база даних, що складається з 11 таблиць, що створюють єдиний інформаційний простір для зберігання даних та доступу до них.

3.6.1 Опис таблиць бази даних

Група таблиць «Слово», «Зображення слів» та «Переклади слів» (рисунок 4.3) використовується для зберігання слів, доданих до системи. У цій групі основна таблиця – «Слово», а дві інші – допоміжні та містять посилання на зображення та переклади для кожного слова.

Поділ на три таблиці дозволяє встановити відношення «один до багатьох», коли кожне слово може мати багато зображень або багато перекладів. Такий підхід дозволяє зберігати загальні дані слів під час використання сервісу. Тому користувачам надається список альтернативних перекладів та зображень.

Всі інші таблиці утворюють другу групу, яка поєднує дані про кожного користувача (рис. 3.7).

Таблиця «Користувач» містить всю необхідну інформацію про користувача, його адресу, логіну та паролі. Таблиця ролей містить імена існуючих ролей.

Так як у кожного користувача може бути кілька ролей і одна одна й та сама роль може мати кілька користувачів, тоді було створено додаткову таблицю «Роль користувача» для реалізації відношення «багато до багатьох» для підтримки цього зв'язку.

На даний момент у системі лише одна роль: КОРИСТУВАЧ. Але розроблена архітектура дозволяє додати нову роль, не порушуючи принцип Open– Closed.

Наприклад, ви можете додати роль ADMIN до системи і дозволити користувачам з цією роллю створювати загальнодоступні навчальні набори слів, доступні всім користувачам служби.

Рисунок 3.7 – Таблиці для збереження бази слів

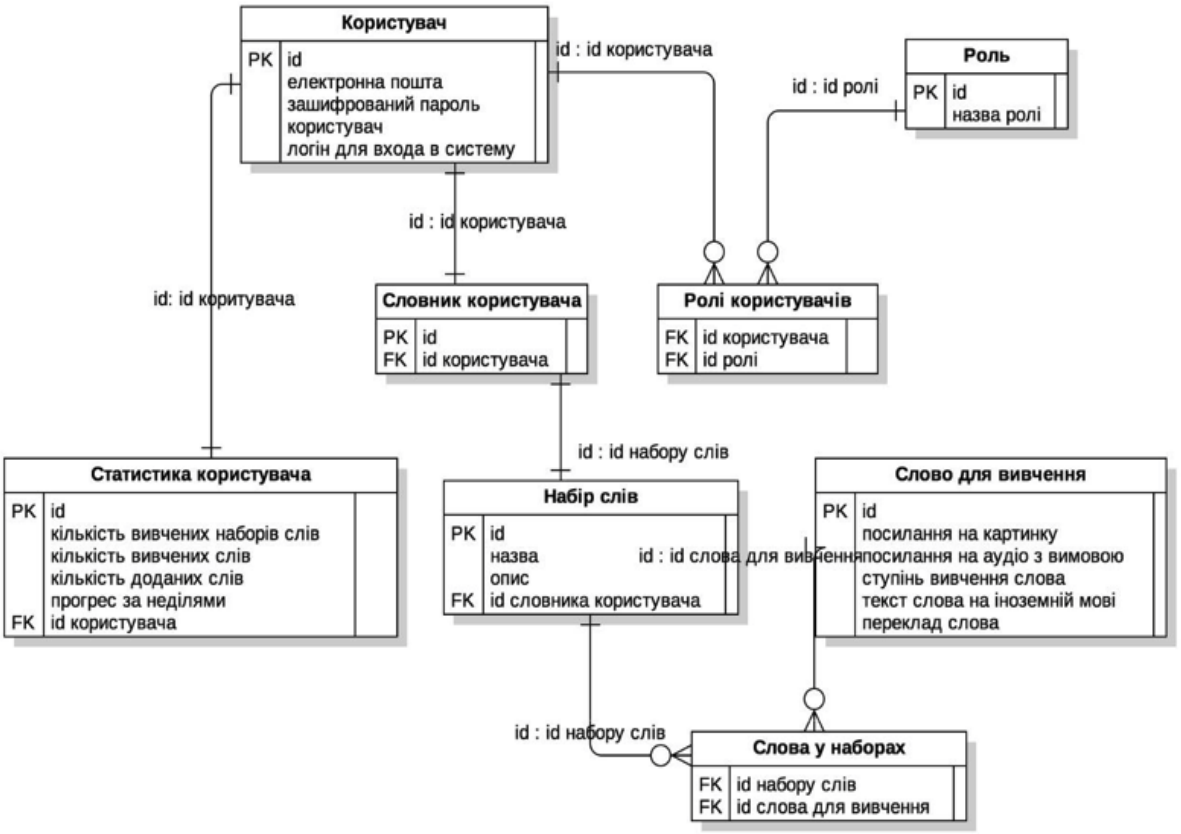

Рисунок 3.8 – Таблиці для збереження даних про користувача

Таблиця «Слово вивчення» відбиває суть слова під час тренування. Ця таблиця містить конкретне значення слова, переклад та етап вивчення слова. Ці слова можуть бути в різних наборах, які описані у таблиці «Набір слів». Оскільки кожне слово може бути додано до безлічі наборів, а набір містить багато слів, було застосовано додаткову таблицю «Слова у наборах» для нормалізації відношення «багато хто до багатьох».

Набір слів пов'язують із користувачем через таблицю «Словник користувача».

Це дозволяє розширювати систему та додавати до неї нові сутності.

Розглянемо докладніше структуру основних таблиць бази даних системи (таблиці 3.1 – 3.5). Для унікальних ідентифікаторів використовується тип bigint, це дозволяє уникнути ситуації, коли в базі даних більше об'єктів, ніж може помістится в 4– байтовому int.

Таблиця 3.1. – Структура таблиці «Користувач»

Ім’я поля	Тип і розмір поля	опис поля
id	bigint	первинний ключ
email	varchar(255)	електронний адрес користувача
name	varchar(255)	Ім’я користувача
password	varchar(255)	пароль користувача у зашифрованому вигляді
username	varchar(255)	логін користувача для входу в систему

Таблиця 3.2. – Структура таблиці «Слово»

Ім’я поля	Тип і розмір поля	опис поля
id	bigint	первинний ключ

sound	varchar(255)	посилання на аудіо файл з вимовою слова
text	varchar(255)	текст слова на іноземній мові

Таблиця 3.3. – Структура таблиці «Слово для вивчення»

Ім’я поля	Тип і розмір поля	Опис поля
id	bigint	первинний ключ
image	varchar(255)	Посилання на зображення для полегшення вивчення слова
sound	varchar(255)	посилання на аудіо файл з вимовою слова
stage	varchar(255)	рівень вивчення слова
text	varchar(255)	текст слова на іноземній мовві
translation	varchar(255)	Переклад слова

Таблиця 3.4. – Структура таблиці «Набір слів»

Ім’я поля	Тип і розмір поля	Опис поля
id	bigint	первинний ключ
description	varchar(255)	опис набору слів
name	varchar(255)	назва набору слів
user_dictionary_id	bigint	id запису з таблиці USER_DICTIONARY

Таблиця 3.5. – Структура таблиці «Роль»

Ім’я поля	Тип і розмір поля	опис поля

id	bigint	первинний ключ
role_type	varchar(255)	назва ролі користувача

3.6.2 Об'єктно– реляційне відображення даних у системі

Для роботи з базою даних використовувався Java Persistence API (JPA), що дозволяє описувати моделі за допомогою єдиного інтерфейсу JPA – це специфікація

Java EE та Java SE, що описують систему для управління збереженням об'єктів Java у таблицях реляційної бази даних у зручній формі. Сама Java не містить реалізації JPA, проте існує безліч реалізацій цієї специфікації від різних компаній.

Java Persistence API — стандартизований інтерфейс для Java ORM фреймворків. Є частиною EJB 3 та J2EE 5, хоча може використовуватись незалежно від них. Виник через популярність вільного ORM фреймворку Hibernate, та бажання мати незалежний від конкретної реалізації стандарт.

Такий підхід до роботи з базою даних називається об'єктно– реляційним відображенням (ORM), який є технологією програмування, яка пов'язує бази даних з концепціями об'єктно– орієнтованих мов програмування.

JPA – не єдиний спосіб зберігання об'єктів бази даних, але він є одним із найпопулярніших у світі Java.

Використання ORM дозволяє відволіктися від бази даних, яку ви використовуєте, і довіряти постачальнику JPA, який може працювати з різними реляційними базами даних.

Таким чином, при розробці ви можете використовувати легку базу даних (наприклад H2), яка створюватиметься одночасно із запуском служби, а для роботи з великою кількістю користувачів використовуйте більш надійні та швидкі бази даних, такі як PostgreSQL.

Рівень роботи з базою складається із зв'язаних моделей (сутностей), що відбивають логічні сутності системи.

Модель – це постійний об'єкт домену, що зберігається в базі даних.

Кожна базова модель має бути анотована анотацією @Entity, а зв'язки між моделями мають бути організовані за допомогою допоміжних анотацій.

Для посилання один на одного потрібно застосувати інструкцію «@OneToOneMapping».

Анотації @ManyToOneMapping і @OneToManyMapping застосовуються, коли кілька моделей належать до однієї моделі.

Щоб зазначити зв'язок «багатьом до багатьох», використовується інструкція @ManyToManyMapping, тобто. коли багато моделей відносяться до багатьох інших моделей. І тут JPA створює допоміжну таблицю для нормалізації відносин між таблицями.

При розробці було застосовано Domain Driven Design, тобто з ієрархії доменної області системи (рисунок 3.9) було створено структуру бази даних.

Це посилання описує лише відносини між моделями системи. Провайдер JPA піклується про структуру таблиць та відносини між ними. Абстрактний клас використовується, щоб гарантувати, що немає необхідності описувати повторювані поля в кожній моделі.

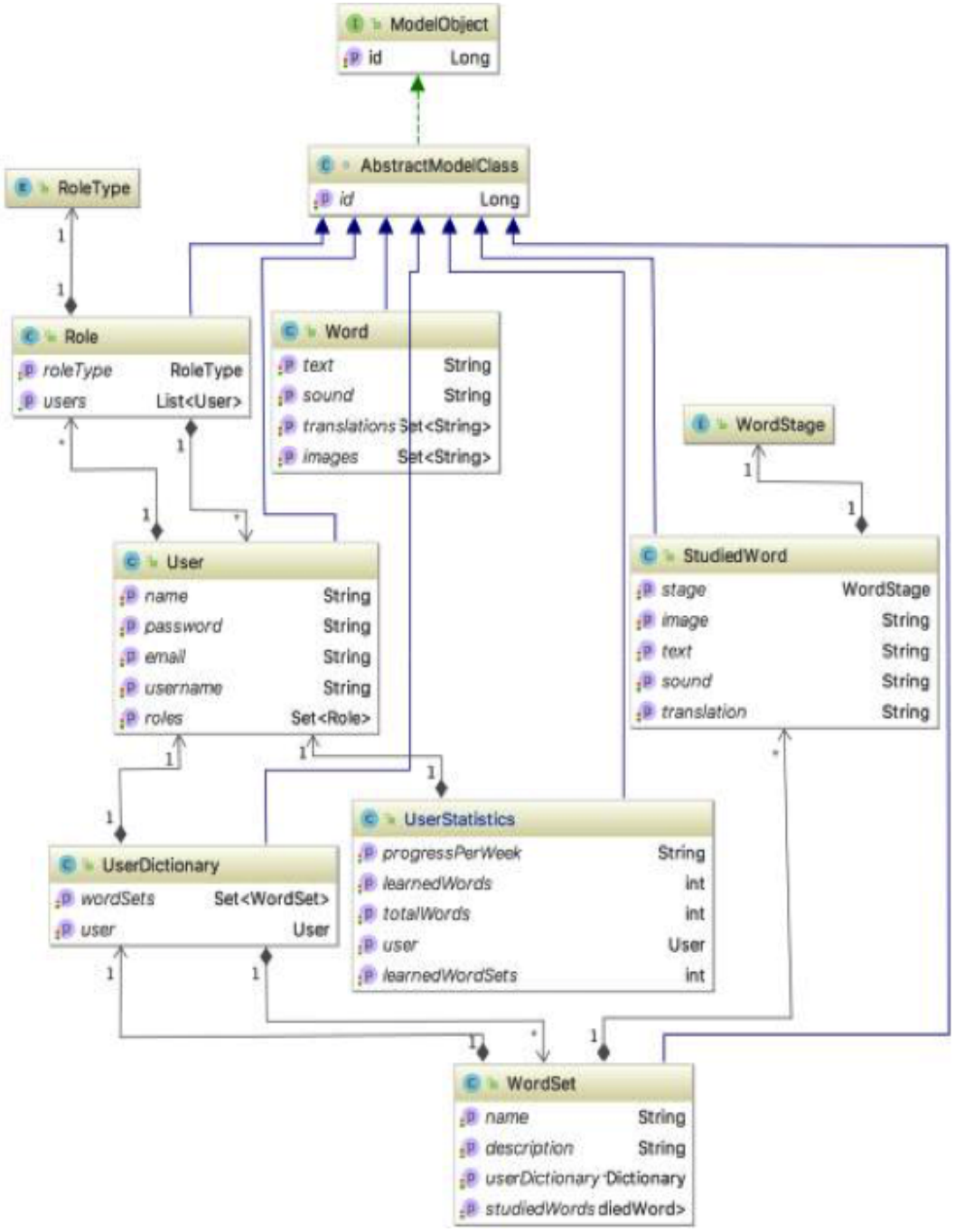

Рисунок 3.9 – Ієрархія моделі

3.7 Розгортання системи

Використання Docker значно спростило процес розгортання служб. Dockerfile описує кроки створення контейнера. Директива FROM дозволяє вказати базовий образ контейнера, який описує дистрибутив Linux, кроки

встановлення додаткових утиліт і конфігурацію контейнера. Базовий контейнер openjdk10: slim використовувався для сервера Java. Мінімалістичний дистрибутив Alpine із встановленою Java Runtime Environment (JRE). Базовим чином, обраним для веб– клієнта, є node: 7, дистрибутив Ubuntu із встановленим nodejs версії 7.

Також у кожному конфігураційному файлі вам необхідно описати кроки для запуску служби (Рисунок 3.10, Рисунок 3.11).

```
FROM frolvlad/alpine-oraclejdk8:slim
VOLUME /tmp
ADD glossary.jar app.jar
RUN sh -c 'touch /app.jar'
ENV JAVA_OPTS=""
ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar" ]
```

Рисунок 3.10 – Dockerfile для Java– серверу

```
FROM node:7
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY package.json /usr/src/app
RUN npm install
COPY . /usr/src/app
EXPOSE 4200
CMD ["npm", "start"]
```

Рисунок 3.11 – Dockerfile для веб– клієнту

Щоб побудувати контейнер, потрібно запустити команду «docker build», вказати ім'я контейнера і шлях до вхідних файлів. Потім Docker зберігає контейнер і додає його до локального репозиторію. Приклад команди для збирання контейнера: «docker build – t hrunishak/glossary– webui: dev».

Щоб переглянути локальні контейнери, просто запустіть команду «dockerimages» (рис.3.12).

```
❯ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED
 hrunishak/glossary-webui   dev                 39758902438e        6 days ago
 hrunishak/glossary         0.0.2-SNAPSHOT      3f492377c943        2 weeks ago
node                        6                   70c1274808eb        3 weeks ago
node                        7                   2ca756a6578b        3 weeks ago
```

Рисунок 3.12 – Перегляд списку контейнерів

Для сервера створення контейнера можна автоматизувати, створивши окреме завдання Gradle. Для цього потрібно використовувати плагін Docker та описати завдання. Плагін застосовується за допомогою команди «застосувати плагін». Щоб створити окреме завдання, ви повинні вказати ім'я програми, шлях до файлу Docker, залежність від інших завдань та підготовчі кроки перед створенням контейнера (рис. 3.13).

```
apply plugin: "docker"

task buildDocker(type: Docker, dependsOn: build) {
    push = true
    applicationName = jar.baseName
    dockerfile = file('src/main/docker/Dockerfile')
    doFirst {
        copy {
            from jar
            into stageDir
        }
```

Рисунок 3.13 – Конфігурація Gradle для збірки Docker контейнера

Після встановлення Gradle можна створити контейнер за допомогою однієї простої команди: «gradlew web: buildDocker». Ця команда складається з назви проекту та назви завдання.

Щоб запустити будь– який контейнер, використовуйте команду «docker run», вкажіть ім'я контейнера та порти, які потрібно відкрити. Приклад команди для запуску сервера: «docker run – p 8080: 8080 – t solomkinmv/glossary».

Щоб запустити контейнер на іншому комп'ютері, необхідно завантажити зібраний образ контейнера. Для спрощення передачі зображень використовувався Dockerhub – спеціальний репозиторій зображень на серверах Docker. Зображення вивантажується за аналогією з серверами Git, за допомогою команди «docker push» та вказівка імені зображення. Потім на будь– якому іншому комп'ютері ви можете написати «docker run <ім'я образу», і Docker завантажить образ і запустить новий контейнер.

Оскільки для запуску всієї системи необхідно одночасно запустити кілька контейнерів та прописати певні параметри, для автоматизації цього процесу використовувався Kubernetes Orchestrator – технологія запуску кількох контейнерів для організації спільної роботи. Переваги Kubernetes включають спрощення запуску безлічі контейнерів та налагодження взаємодії між ними. Файли yaml описують всі служби, які потрібно розгорнути, порти для зв'язку та додаткові параметри середовища (рисунок 3.14).

```
version: '2' # specify docker-compose version

# Define the services/containers to be run
services:
  webui: # name of the first service
    image: hrunishak/glossary-webui:dev # specify the directory of the Dockerfile
    ports:
      - "4200:4200" # specify port forwarding

  server: #name of the second service
    environment:
      - SPRING_PROFILES_ACTIVE=dev,s3
    image: hrunishak/glossary:0.0.2-SNAPSHOT # specify the directory of the Dockerfile
    ports:
      - "8080:8080" #specify ports forwarding
```

Рисунок 3.14 – Файл– конфігурації для запуску сервісів системи

Після цього завантаження системи прописуєм «kubelet create– f». Ця команда відображає настроєні середовища, запускає всі служби та відкриває порти, щоб вони могли обмінюватись даними.

Розгортання системи зводиться до виклику команди в кластері Kubernetes, який налаштований для запуску всього необхідного для повноцінної роботи. Цей підхід є провідним у галузі та використовується великими компаніями для запуску своїх продуктів.

3.8 Автомасштабування за довільними метриками

Kubernetes Orchestrator має механізм масштабування, який дозволяє змінювати кількість екземплярів кожної служби.

Масштабування дозволяє вам робити всі запити клієнтів у періоди пікового навантаження, але виникає проблема пунктуальності.

Неможливо постійно передбачати зміни навантаження на систему, тому слід відстежувати показники системи і діяти своєчасно.

Зазвичай, для надійності система працює з надмірними ресурсами, щоб витримувати непередбачуване навантаження, але такий підхід використовує ресурси віртуальних машин і потребує додаткових грошей.

Використання черги дозволяє автоматизувати горизонтальне масштабування та, таким чином, оптимізувати витрати та покращити швидкодію системи.

Щоб вирішити проблему автоматизації, можна використовувати Kubernetes Orchestrator та його Horizontal Pod Autoscaler (HPA).

HPA реалізований як цикл моніторингу, який періодично опитує метрики ресурсів щодо ключових метрик, таких як використання ЦП / ОЗУ, а також надає прикладний програмний інтерфейс з відкритим кодом для реалізації специфічних для системи метрик (рисунок 3.15).

За замовчуванням Kubernetes підтримує автоматичне масштабування залежно від використання ЦП або ОЗП, але ці показники не підходять для всіх систем. Іноді для більшої точності слід використовувати інші показники.

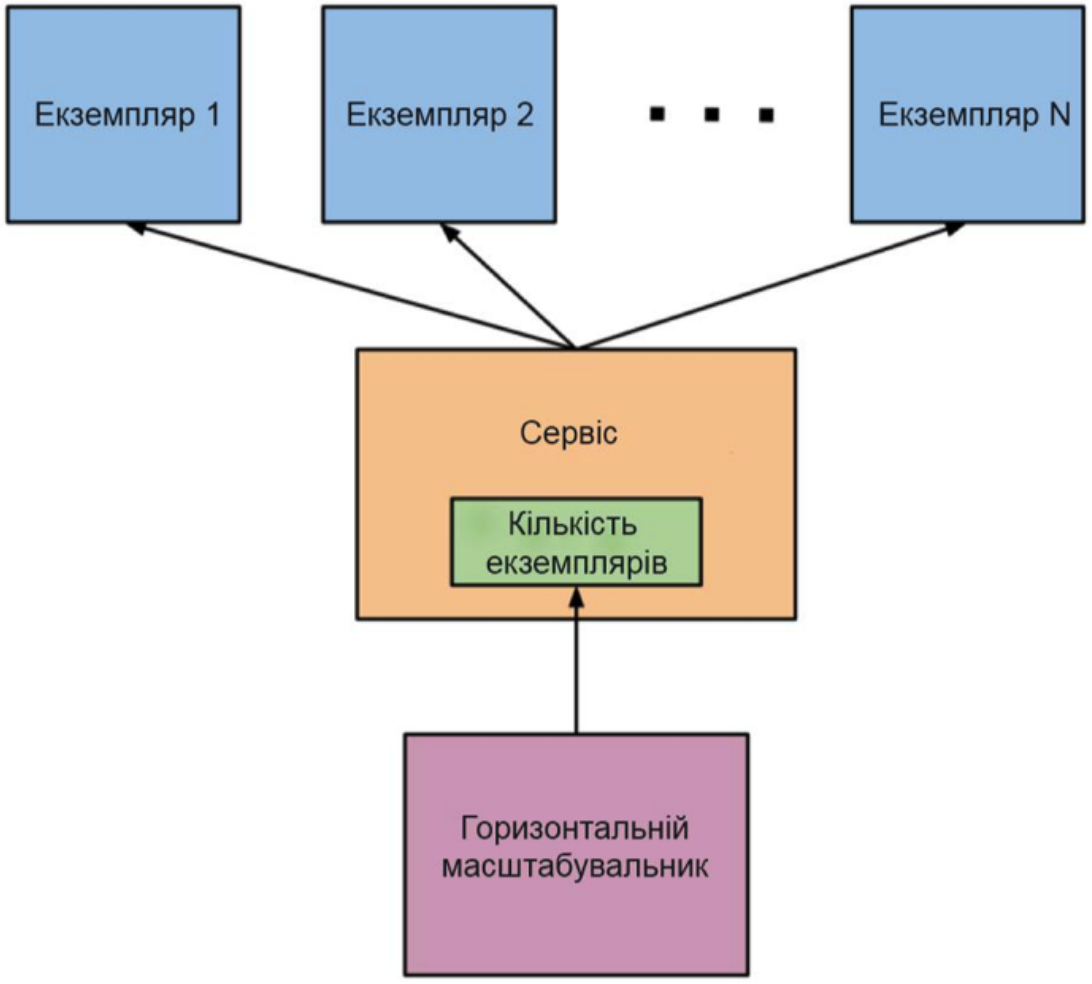

Рисунок 3.15 – Діаграма роботи горизонтального масштабувальника

Для оптимізації автоматичного масштабування довжина черги ідеальна як індикатор необхідності масштабування. Чим більше необроблених повідомлень перебуває у черзі, тим більше потрібно створити нових екземплярів служби. Якщо черга майже порожня, кількість послуг можна знову зменшити.

Так, наприклад, при використанні Spring Framework ви можете застосувати Spring Actuator, що дає базові метрики про систему. Подробиці реалізації точки доступу, що повертає метрики користувача, були описані в попередньому розділі.

На кожній ітерації контролер вимагає рівень використання зазначених ресурсів в описі кожного скалера (рис. 3.16). Потім, якщо встановлено цільовий рівень використання ресурсів, контролер обчислює значення метрики кожного екземпляра служби і порівнює його з цільовим рівнем, виходячи з якого вирішує збільшити чи зменшити масштаб.

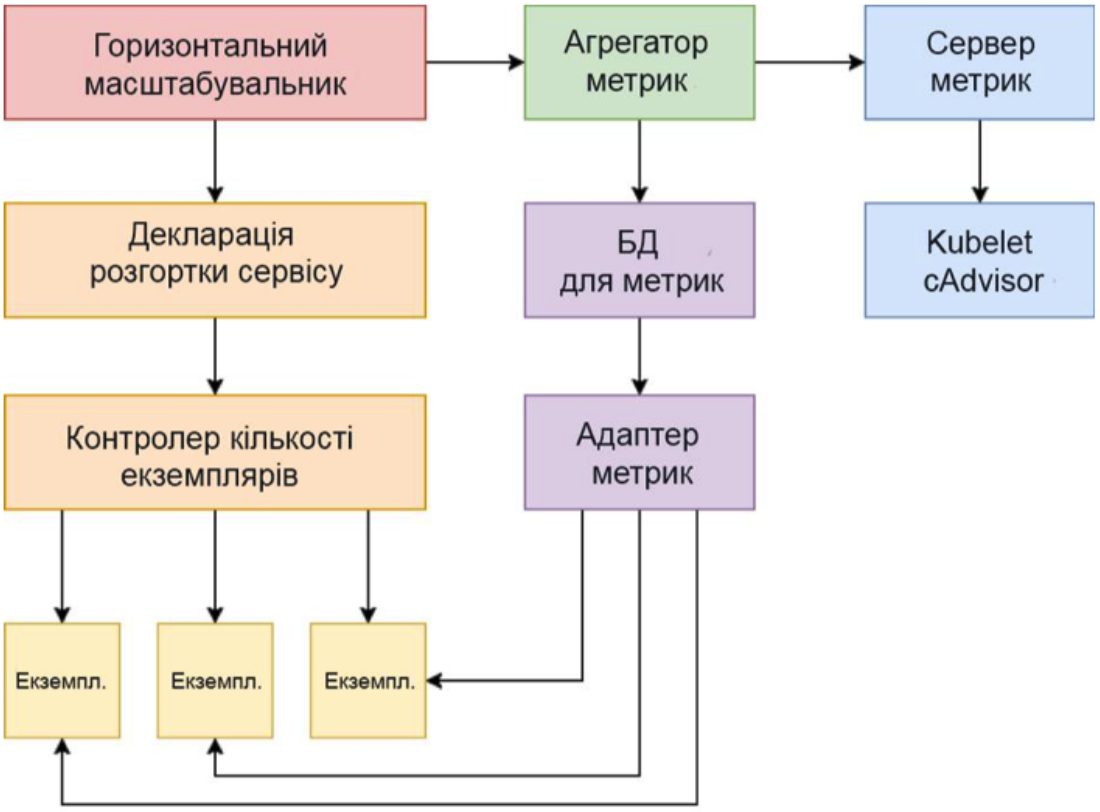

Рисунок 3.16 – Діаграма взаємодії компонентів для збору та збереження метрик

Для вирішення проблеми спочатку потрібно перекласти розгортання системи для кластера Kubernetes. Щоб застосувати горизонтальне масштабування в Kubernetes, необхідно, щоб кожна служба, яка потребує масштабування, реалізовувала HTTP API та повертала необхідні метрики. Це можна зробити вручну за допомогою будь– якої мови програмування або спеціалізованих бібліотек.

Так, наприклад, при використанні Spring Framework ви можете застосувати Spring Actuator, що дає базові метрики про систему. Подробиці реалізації точки доступу, що повертає метрики користувача, були описані в попередньому розділі.

Для обробки метрик обслуговування використовувався сервер метрик, який є агрегатором усіх метриків із кластера.

Сервер метрик збирає дані про використання процесора та оперативної пам'яті кожного вузла, витягуючи дані за допомогою kubernetes.summary_api, який є ефективним API для передачі даних з Kubelet/cAdvisor на сервер метрик (рис. 3.17).

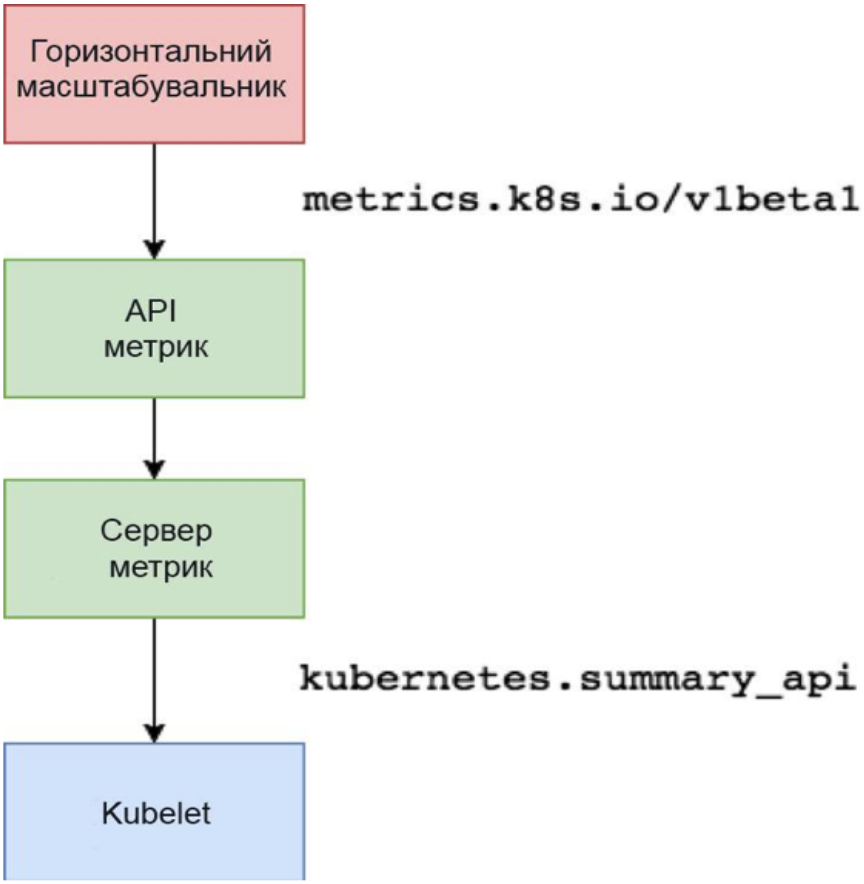

Рисунок 3.17 – Збір метрик з кожного екземпляра сервісу за допомогою Kubelet

Для запуску сервера метрик ми використовуємо налаштування з офіційного репозиторію, який запускається за допомогою командного рядка та консольної утиліти kubectl.

Для автоматичного масштабування HPA був розгорнутий у Kubernetes і вказував мінімальну та максимальну кількість екземплярів, ім'я метрики та цільове середнє значення (рис. 3.18).

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: spring-boot-hpa
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: backend
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Pods
    pods:
      metricName: messages
      targetAverageValue: 10
```

Рисунок 3.18 – Налаштування для масштабування за метрикою та цільовим значенням

У цьому прикладі вказується мінімальна та максимальна кількість екземплярів служби, а також середнє значення метрики повідомлення залежно від того, яке масштабування відбувається.

Після створення HPA кількість екземплярів служби повинна дорівнювати мінімальній кількості, вказаній у файлі конфігурації. У цьому випадку 2. Щоб переглянути події, які призвели до масштабування, виконайте таку команду: kubectl describe hpa.

Якщо ви завантажите систему, ви помітите, що кількість екземплярів служби збільшилася (рис. 3.19). Це означає, що автомасштабування Kubernetes працює. За допомогою Horizontal Pod Autoscaler можна створювати системи, які можуть динамічно адаптуватися до поточного поточного навантаження системи. Такий підхід не тільки збільшує стабільність систем, але й дозволяє заощаджувати ресурси на серверах, що дозволяє мінімізувати кількість

необхідних ресурсів за мінімального навантаження. Використання показників, що настроюються, підвищує точність автомасштабування і надає безліч варіантів налаштування розподіленої системи.

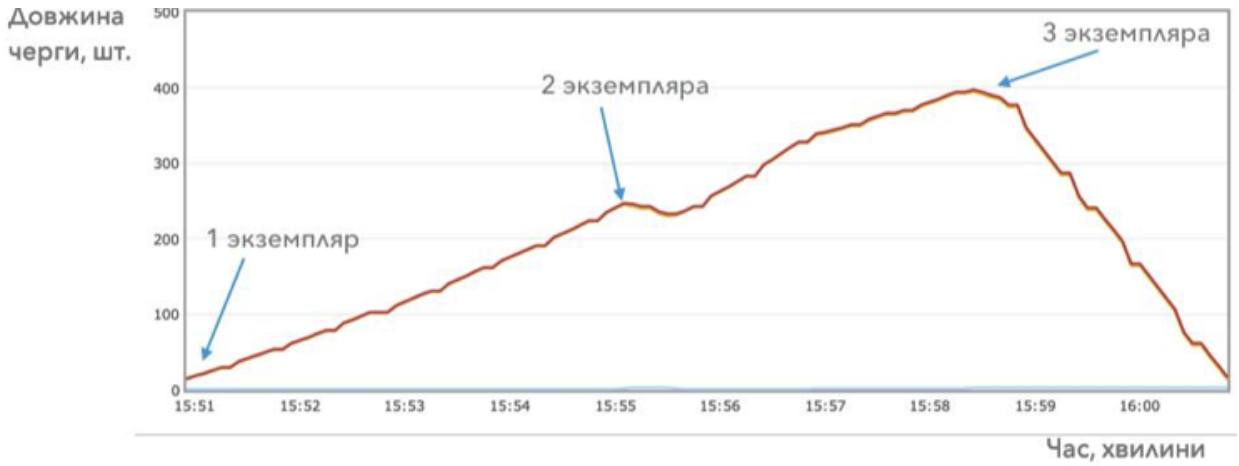

Рисунок 3.19 – Автоматичне масштабування при збільшенні черги

3.9 Аутентифікація користувача

Так як у кожного користувача свої слова для навчання є певний прогрес і статистика, без механізму автентифікації обійтися неможливо. Для вирішення цієї проблеми було застосовано Spring Security , яка надає базові методи захисту веб– служби.

Для аутентифікації Spring Security пропонує базову аутентифікацію та файли cookie. Обидва ці методи швидко розвиваються, але вони мають кілька недоліків. Аутентифікація за допомогою файлів cookie змушує сеанс зберігатися на сервері та адаптований для роботи з браузерами, у той час як більшість сучасних веб– сервісів мають безліч мобільних та веб– клієнтів, а базова автентифікація змушує вас надсилати ім'я користувача та пароль при кожному запиті, який просто неприйнятно з погляду безпеки. Для усунення цих недоліків було вирішено застосувати більше сучасний метод аутентифікації на основі JSON Web Token, скорочено JWT

Послідовність роботи з JWT наступна: клієнт відправляє на сервер логін та пароль, сервер створює JWT та повертає його клієнту, наступного разу, коли клієнт застосовує токен до запиту, сервер перевіряє підпис та аутентифікує клієнта на основі отриманих даних та повертає результат (рисунок 3.20) .

Токен JWT сам по собі може містити всю необхідну інформацію для аутентифікації, це дозволяє позбавитися сеансу та непотрібних перевірок на наявність користувача в базі або його ролі. Крім того, JWT – це універсальний інструмент, що дозволяє працювати як із браузерами, так і з мобільними пристроями.

Оскільки Spring Security не надає реалізацію аутентифікації JWT, нам потрібно було додати її, щоб максимально ефективно інтегруватися з поведінкою Spring Security.

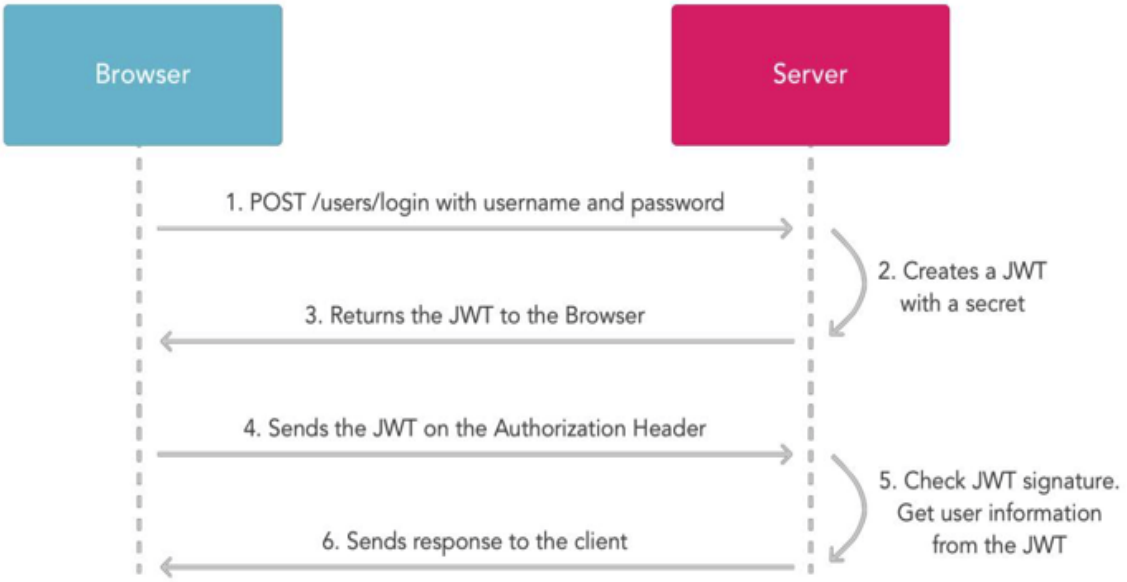

Рисунок 3.20 – Автентифікація на основі токенів

Залежність io.jsonwebtoken: jwt була додана для створення, перевірки та дешифрування токенів. Spring Security надає механізм фільтрації запитів, який дозволяє обробляти, аутентифікувати чи відхиляти запит для виклику

відповідного контролера. Тому для підтримки JWT був створений новий фільтр для обробки запиту та отримання свого токена заголовка, а потім викликає постачальника аутентифікації та передає йому токен. Провайдер аутентифікації реалізує інтерфейс AuthenticationProvider, перевіряє підпис, видаляє всю корисну інформацію з токена, таку як ім'я користувача, роль (користувач або адміністратор) та термін дії токена. Результатом роботи провайдера є об'єкт Authentication, що містить всю інформацію про користувача та його роль, необхідну системою. Для обробки ситуацій, коли аутентифікація не вдалася, був створений обробник помилок, який реалізує інтерфейс AuthenticationFailureHandler, вловлює винятки Java та генерує зручні HTTP– відповіді зі статусом UNAUTHORIZED та кодом помилки.

Інтеграція Spring Security та аутентифікація через фільтр запитів дозволили нам чітко розрізняти компоненти, що відповідають за безпеку та бізнес– логіку. Будь– які зміни автентифікації не вплинуть на базову логіку служби. Результат реалізації JWT дозволив нам знизити навантаження на базу даних та побудувати службу REST без збереження стану, яка спростить масштабування зі збільшенням кількості користувачів.

3.10 Робота із зображеннями

Робота із зображеннями складається з двох основних елементів: пошук зображення за словом та завантаження користувачем власних зображень.

Для пошуку зображень використовувався відкритий API веб– сайту Flickr, який спеціалізується на роботі із зображеннями. Використовуючи цей API, можна зробити запит, який поверне певну кількість посилань на зображення певного розміру. Таким чином, було реалізовано сервіс, що повертає список посилань на невеликі зображення. Клієнт може запросити сервер і отримати всю необхідну інформацію для пошукового запиту. Такий

рівень абстракції означає, що клієнту не потрібно турбуватися про деталі пошукової системи та джерело зображень.

Завантажені зображення повинні повертати посилання на збережене зображення, щоб клієнту не потрібно було знати місце збереження та деталі побудови посилань. Для збереження зображень реалізовано 2 сервіси, один з яких зберігається на файловій системі, а друга – на зовнішній Amazon S3 .

Обидві служби реалізують той самий інтерфейс і призначені для роботи з різними профілями. Таким чином, якщо сервер запущений із профілем s3, зображення зберігаються зовнішньою службою. В іншому випадку, всі зображення будуть збережені у файловій системі в папці, вказаній у налаштуваннях сервісу.

Amazon S3 – це служба зберігання статичних файлів (рис. 3.21).

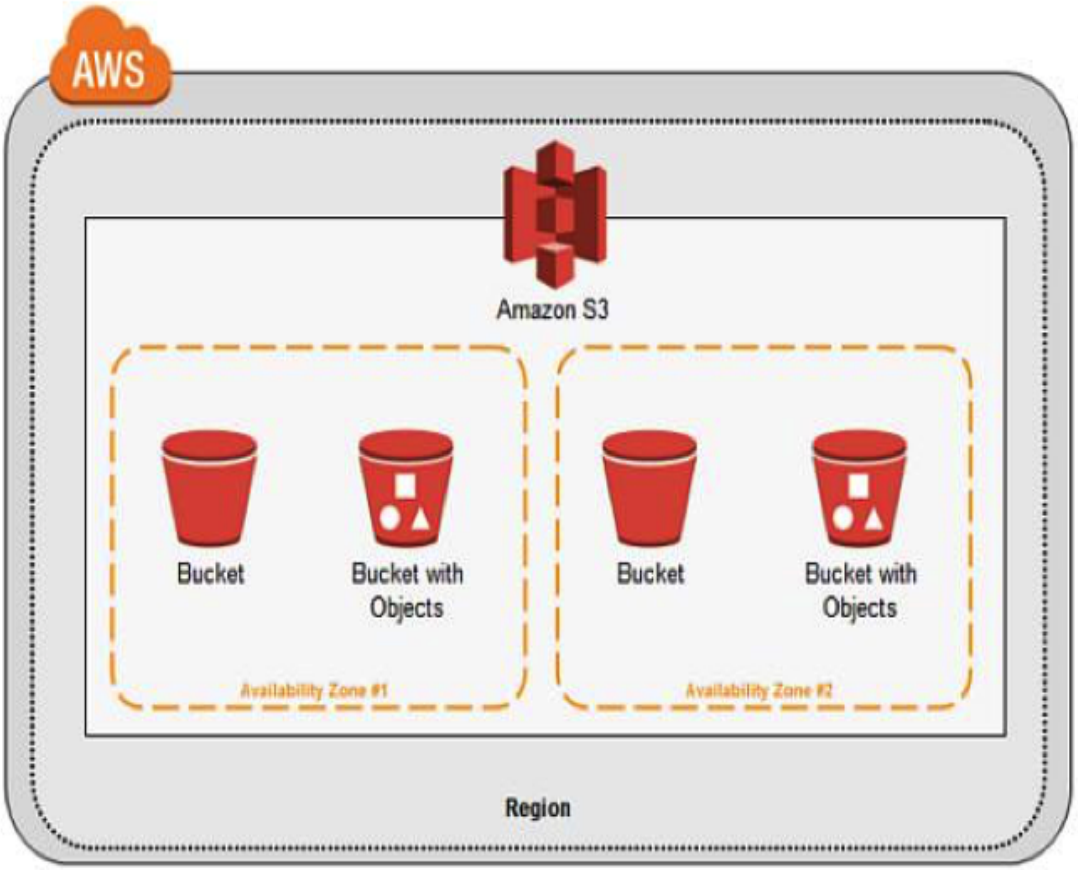

Рисунок 3.21 – Архітектура Amazon S3

Він ідеально підходить для зберігання даних у вигляді зображень та аудіофайлів. Для цього було створено дві групи для зберігання даних: Глосарій зображень та Глосарій вимови.

3.11 Генерація вимови слів

Для створення вимови слів використовувався сервіс Amazon Polly. Ця служба приймає текст, який вимагає вимови та ряду параметрів, таких як тип голосу, швидкість і гучність вимови та багато іншого. Результатом цього API є потік вхідних даних в аудіофайл (рисунок 3.22).

Amazon SDK для Java виключає прямий API та низько рівневу потокову передачу введення. За допомогою бібліотеки ви можете покластися на розроблену вами абстракцію, створити об'єкт AmazonPolly за допомогою вбудованого Builder, створити запит синтезу вимови та викликати метод synthesizeSpeech.

В даному додатку реалізовано два методи зберігання аудіо файлів.Один записує дані в файлову систему, наступний на amazon S3.

Рисунок 3.22 – Взаємодія з Amazon Polly

3.12 Переклад слів

API Яндекс Перекладача використовувався для автоматичного перекладу слів, що полегшувало додавання слів користувачем. Сервіс Яндекс дозволяє робити безкоштовні запити з певними обмеженнями: результат повертає лише один варіант перекладу. Для отримання перекладу надішліть запит та вкажіть ключ із мовою перекладу (рис. 3.23). Де key – це ключ, отриманий з сайту Яндекс Перекладача, ui – ідентифікатор, з якого і якою мовою виконується переклад. Взаємодія із зовнішньою службою здійснюється у відповідному класі, який додає шар абстракції над перекладом слів.

```
https://translate.yandex.net/api/v1.5/tr.json/getLangs ?
key=<API key>
 & [ui=<language code>]
 & [callback=<name of the callback function>]
```

Рисунок 3.23 – Yandex Translate API

3.13 Тестування серверної частини

Без тестування неможливо бути впевненим, що продукт працює правильно. У цьому продукті акцент був зроблений на інтеграційне тестування, оскільки воно може підтвердити правильність роботи всієї системи, на відміну від модульного тестування, де кожен невеликий компонент тестується окремо, що не гарантує правильної роботи всієї системи.

Інтеграційне тестування набагато складніше за власну організацію, оскільки кожен тест повинен піднімати весь сервіс, включаючи базу даних. Як наслідок, інтеграційні тести виконуються набагато повільніше. Для тестування використовувалися Spring Boot Tests, що спрощує запуск усієї служби, запуск

полегшеної бази даних, що зберігає дані в пам'яті, та спеціальні методи створення запитів і перевірки відповідей сервера.

Для тестів було створено абстрактний клас MockMvcBase, який має анотацію @SpringBootTest для запуску тестів, ініціалізує тестове середовище MockMvc та оголошує метод, який створює обробник запиту для додавання заголовка автентифікації до тестів. Потім кожен тестовий клас випливає з MockMvcBase, описує, які дані слід додавати перед кожним тестом і моделює реальну роботу сервісу.

Отже, щоб зробити запит GET від імені користувача, ви повинні виконати «mockMvc.perform (get(«/api/words/{wordId}»,word.getId()). With(userToken()))». Метод perform().) Приймає інструкції про те, який запит виконати, а метод with() приймає додаткові програми обробки програм. Метод userToken () створений вручну та додає дані для автентифікації. Вказаний рядок запускає всю систему і обробляє запит, якби це був справжній запуск. Шаблон проектування Buildera використовується перевірки результатів, тобто. викликається метод andExpect () і очікувана умова виконується.

Умови які можуть провіритись:

– код відповіді (200, 201, 404, 500 та ін.);

– тип відповіді (text/plain, application/json та ін.);

– очікуване тіло відповіді.

3.14 Документація з API

Оскільки сервер і клієнт – це абсолютно різні програми, документація серверного API була створена для опису взаємодії, що дозволило задокументувати серверний контракт і дотримуватися його клієнта.

Зазвичай найбільша проблема з документацією полягає в тому, що вона не є актуальною. Інакше кажучи, договір взаємодії зазвичай змінюється, але документація залишається старою, тобто неактуальною.

Для створення документації використовувався Spring REST Docs, що вирішує проблему застарілої документації, оскільки немає окремого місця для опису API . Вся документація пишеться одразу після тестування. Якщо в документації пропущене поле описано, результат чи навпаки щось описує, отже система не проходить цей тест.

Документація описується у форматі моделі Builder і пишеться відразу після методів andExpect(). Spring REST Docs дозволяє описувати такі елементи:

– заголовки HTTP;
– параметри HTTP– запиту;
– параметри URL– шляху;
– поля відповіді;
– приклад запиту.

Після запуску тестів створюються файли, що описують один аспект запиту. Щоб створити єдину сторінку документації, вам необхідно створити індексний файл у форматі AsciiDoc (adoc) у каталозі src/docs/. У цьому файлі можна написати текст і вставити згенеровані фрагменти (рис. 3.24).

Крім того, файл підтримує спеціальний синтаксис для форматування тексту та створення таблиць.

```
[[resource-practices-quizzes]]
=== Get practice quiz
A `GET` request generates quiz for the specified word set.

include::{snippets}/practice-controller-test/get-practice-quiz/response-fields.adoc[]

==== Headers

include::{snippets}/practice-controller-test/get-practice-quiz/request-headers.adoc[]

==== Request parameters

include::{snippets}/practice-controller-test/get-practice-quiz/request-parameters.adoc[]

==== Example request

include::{snippets}/practice-controller-test/get-practice-quiz/curl-request.adoc[]

==== Example response

include::{snippets}/practice-controller-test/get-practice-quiz/http-response.adoc[]

==== Links

include::{snippets}/practice-controller-test/get-practice-quiz/links.adoc[]
```

Рисунок 3.24 – Шаблон документації

Фрагменти документації форматуються та вставляються під час налаштування системи.

У шаблоні документації це дозволяє відображати останню версію HTML– документації під час роботи служби (рис. 3.25).

Публічний доступ до документації для сервера налаштував «/Docs/index.html» із конфігурацією Gradle (рис. 3.26) та налаштуваннями Spring для надання статичних даних за допомогою вбудованого адаптера веб– конфігурації «WebMvcConfigurerAdapter».

```
jar {
    baseName = "glossary"
    version = null
    dependsOn asciidoctor
    from ("${asciidoctor.outputDir}/html5") {
        into 'static/docs'
    }
}
```

Рисунок 3.25 – Конфігурація Gradle для відображення документації

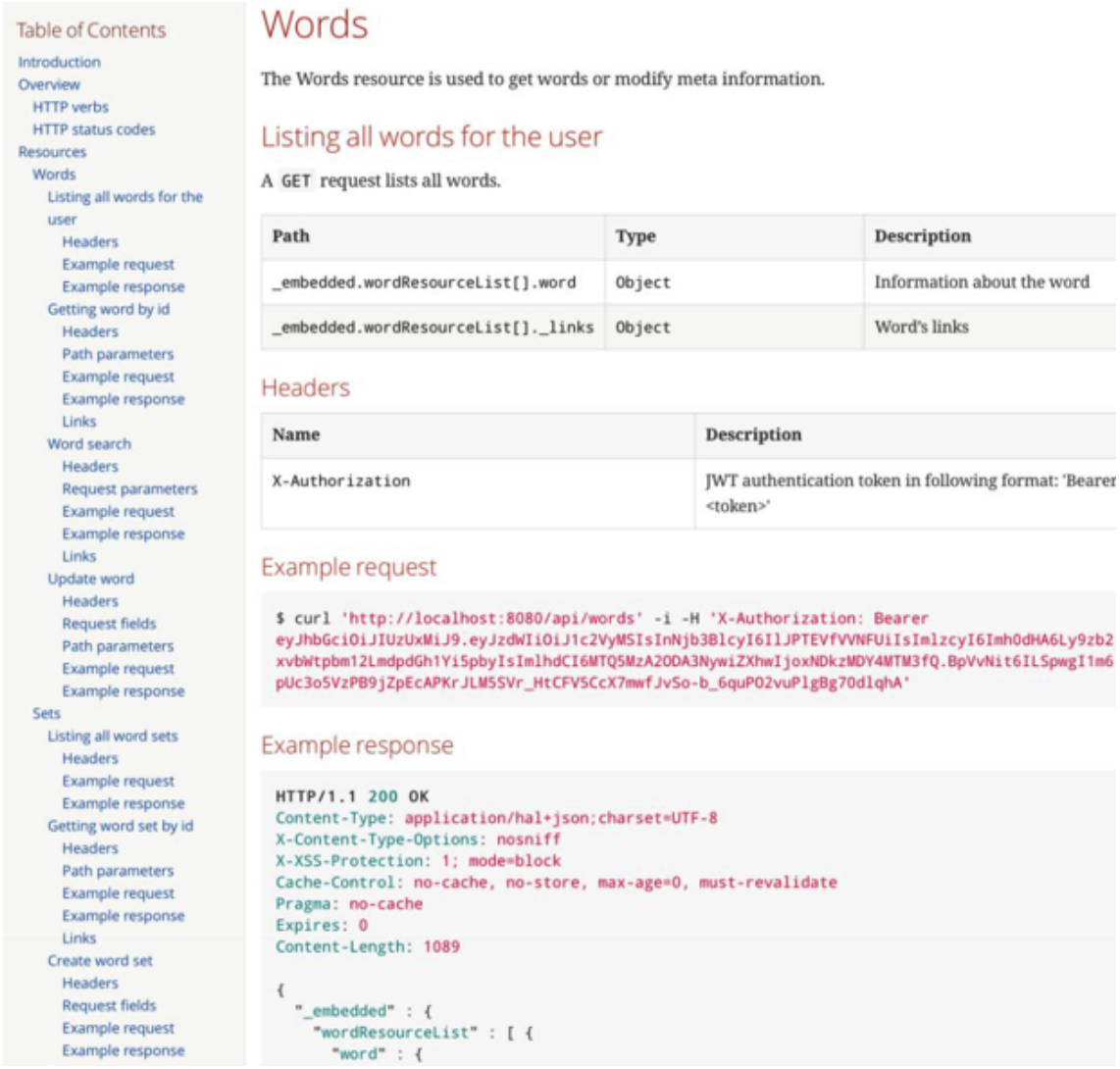

Рисунок 3.26 – Сторінка з документацією

Створена документація має вбудовану панель навігації, підсвічування синтаксису, автоматичне форматування JSON та підтримку мобільних пристроїв. Окрім того стилі документації підтримують адаптивний дизайн, тобто відображення на пристроях з невеликим екраном. Відформатований файл із документацією може або розповсюджуватися самою службою, або розміщуватись як звичайний HTML– файл.

Шаблонний код від loombook.

У Java прийнято створювати прості класи зберігання даних (JavaBeans). Усі поля класу повинні бути закритими, і для отримання або зміни значень ви повинні створити методи з префіксом get або set. Наприклад для поля

Необхідно створити «користувацькі» методи getUser() та setUser() _0_. Зазвичай, ці методи забезпечують прямий доступ до поля класу, але цей підхід забезпечує необхідну інкапсуляцію даних. Ви завжди можете додати логіку цих методів, не переписуючи код. Цей підхід має недоліки – великий обсяг шаблонного коду, який ускладнює читання (рис. 3.27). Для вирішення цієї проблеми використовувалася бібліотека Lombok, яка генерує цей шаблонний код під час компіляції .

Щоб зберігати об'єкти в хешованих структурах даних, вам необхідно перезаписати методи equals() та hashCode().

Найчастіше визначення цих методів цілком шаблонно. Приблизно така сама ситуація і з методом toString().

Для генерації всіх цих шаблонних методів достатньо додати інструкції @EqualsAndHashCode і @ToString.

Оскільки ця група анотацій використовується дуже часто, бібліотека надає анотацію @Data, яка об'єднує цю групу. Загалом Lombok надає безліч різних анотацій для генерації найбільш поширеного шаблонного коду.

Також генерація відбувається на етапі компіляції, також для підтримки Loombook потрібно встановити спец плагін .

Принцип роботи Ломбока є досить простим. Розробник використовує інструкції, щоб вказати, які елементи створювати.

Щоб створити гетери та сетери, вам потрібно помістити наступні два оператори над класом: @ Getter, @ Setter. Тоді замість сорока рядків коду залишиться лише десять (рис. 3.28).

```
public class User {
    private String name;
    private String surname;
    private String username;
    private String password;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

Рисунок 3.27 – Вигляд класу java без використання Loombook

```
@Getter
@Setter
public class User {
    private String name;
    private String surname;
    private String username;
    private String password;
}
```

Рисунок 3.28 – Вигляд класу без використання шаблонного коду

3.15 Веб– клієнт

Веб–клієнт побудований за допомогою фреймворку Angular 6, який дозволяє розділяти класи на різні рівні абстракції: служби та компоненти.

Для роботи з кожним ресурсом створено окрему службу інкапсуляції.

у собі всі необхідні знання частини зовнішньої системи. Так, наприклад, всі аспекти роботи зі словами були реалізовані в WordService, а процедура розпізнавання мови реалізована в окремому SpeechRecognitionService.

Angular 6 використовує принцип Inversion Of Control, який унеможливлює ручне створення класів і переносить цю відповідальність на окремі файли конфігурації. Для впровадження залежностей необхідно виконати такі кроки: відзначити службу з анотацією @Injectable(), додати службу до розділу «постачальник» опису модуля та створити конструктор, який приймає цю службу в існуючій службі.

На етапі запуску сервісу Angular 6 вручну внесе компоненти всі необхідні сервіси.

Компонент відповідає за керування певною частиною шаблону HTML. Для створення компонента необхідно відзначити будь– який клас інструкцією @Component і вказати шлях до файлу шаблону в параметрі інструкції templateUrl. Вибір компонентів базується на поточному шляху у браузері. Шлях та необхідний компонент налаштовуються в окремому модулі маршрутизації (рис. 3.29).

Завдяки такому поділу кожен сервісний елемент має сильну внутрішню узгодженість та низьку узгодженість з іншими компонентами.

Це спрощує процес написання та підтримки існуючого коду.

Кожна служба повинна мати доступ до відповідного ресурсу на сервері, але явна вказівка адреси сервера може призвести до великих проблем при розгортанні системи в різних середовищах. Для вирішення цієї проблеми було застосовано проксі. Кожна служба не пов'язується із сервером безпосередньо, але вказує відносну адресу необхідного API, наприклад: «/api/images». Таким чином, запит буде направлений до того ж сервісу, звідки він був відправлений. Щоб перенаправити запит на інший сервер, необхідно створити файл proxy.conf.json і помістити його в корінь проекту.

У файлі необхідно встановити умови перенаправлення та адреси цільового сервера (рис 3.30).

```
const routes: Routes = [
  {path: '', component: DictionaryComponent},
  {path: 'set/:id', component: WordSetComponent},
  {path: 'practice', component: PracticeComponent},
...
];

@NgModule({
  imports: [
    RouterModule.forChild(routes)
  ], exports: [
    RouterModule
  ]
})
export class DictionaryRoutingModule {
}
```

Рисунок 3.29 – Налаштування маршрутизації та компонентів

За допомогою проксі настроюється одне місце, де вказується адреса сервера API. Це полегшує подальшу роботу з клієнтом.

```
{
  "/api": {
    "target": "http://localhost:8080"
    "secure": false,
    "changeOrigin": true,
    "logLevel": "info"
  }
}
```

Рисунок 3.30 – Налаштування проксі

Логіка веб– клієнта поділена на різні компоненти та прив'язана до різних URL– адрес. Таким чином легко розширити функціональність веб– клієнта без

значних витрат часу. Крім того, для більшої ефективності використовується відкладена ініціалізація компонентів, пов'язаних із вивченням слів (рис. 3.31).

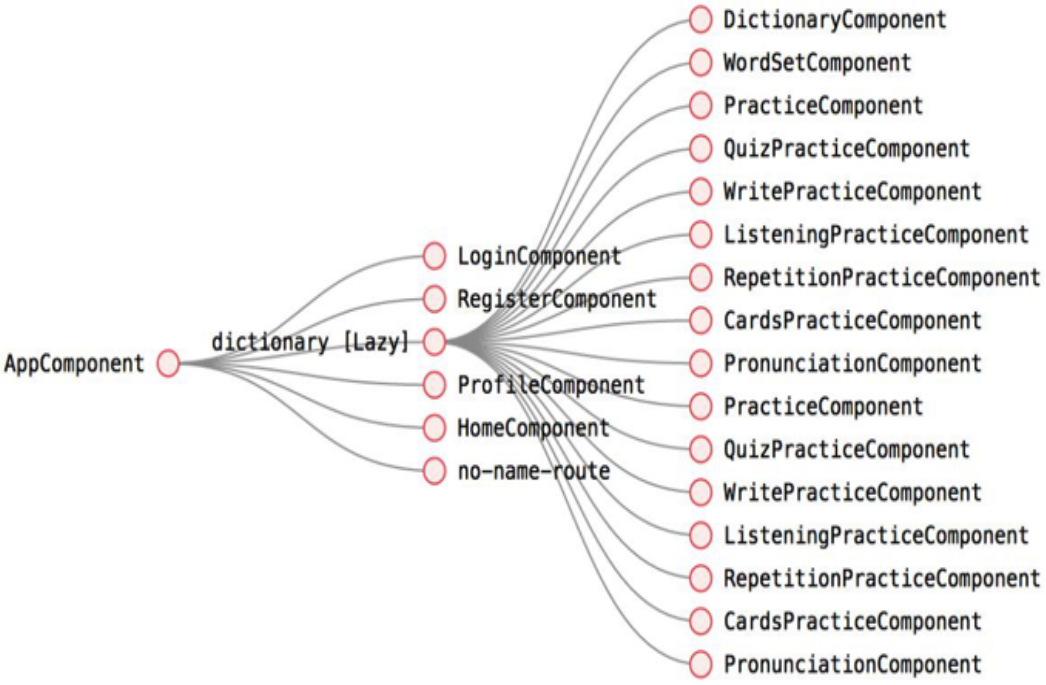

Рисунок 3.31 – Ієрархія компонентів

Таким чином, користувач, який не увійшов до свого профілю, не витрачатиме час на ініціалізацію компонентів, які йому недоступні.

Такий спосіб ініціалізації компонентів можливий завдяки використанню в проекті Angular 6 та розбиття коду на безліч компонентів та модулів, кожен з яких має свою частину відповідальності.

3.16 Взаємодія компонентів системи

Програмний продукт складається з кількох сервісів, кожен із яких виконує окрему частину роботи (рисунок 3.32). Сервер взаємодіє із Amazon S3 для зберігання статичних даних.

Це відбувається, коли користувач завантажує нове зображення або зберігає вимову нового слова.

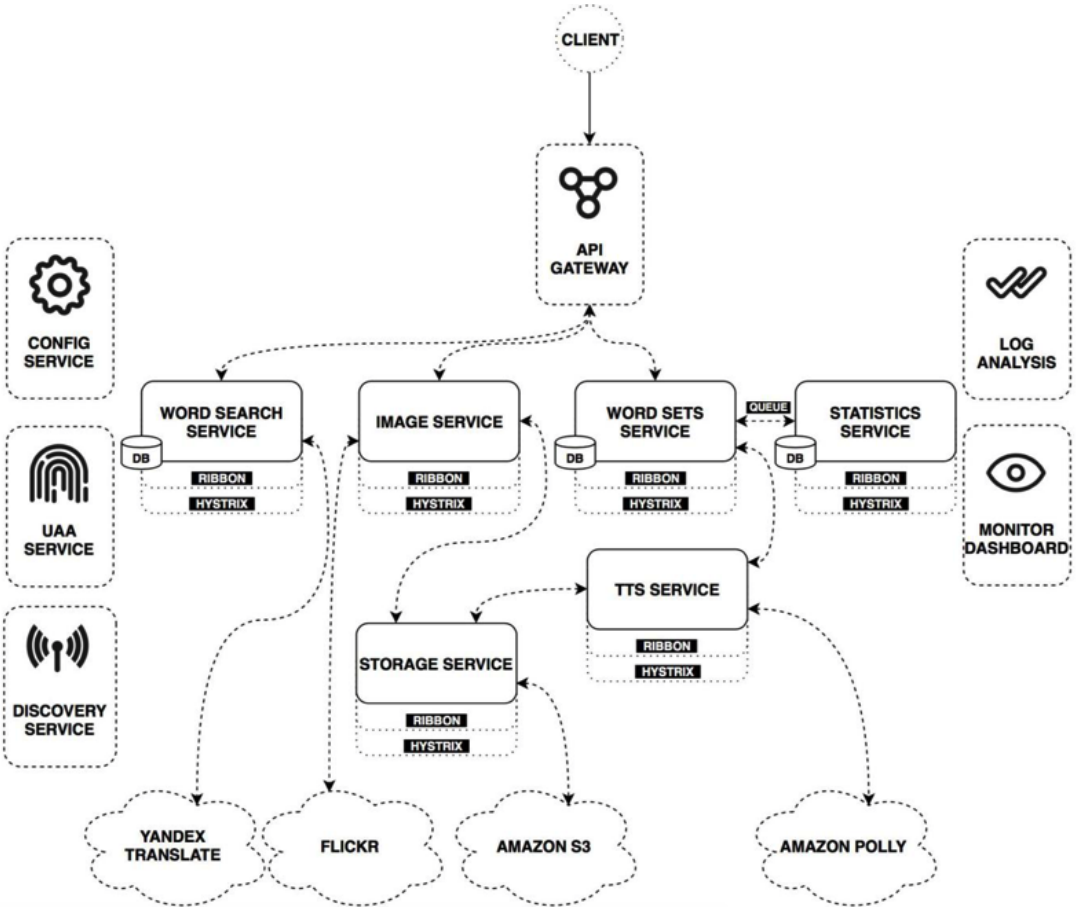

Рисунок 3.32 – Взаємодія між сервісами

Взаємодія з Яндекс Перекладачем відбувається, коли сервер уперше зберігає слово.

Результат перекладу додається до переліку слова. Коли додається нове слово, сервер надсилає до Amazon Polly запит на створення вимови слова. Поки що Flickr викликається щоразу для пошуку зображень, коли користувач додає слово до свого набору слів.

Усі запити надсилаються на сервер, тому клієнти не знають існування зовнішніх служб. Якщо потрібно змінити сервіс перекладу або пошуку зображень, клієнту нічого змінювати не потрібно.

Якщо клієнту потрібне посилання на статичні ресурси, сервер може повертати прямі посилання на зовнішній ресурс, але це ніяк не впливає на клієнта.

3.17 Практика вимови

Навчання вимови здійснюється так: сервіс правильно вимовляє слово, після чого користувач натискає клавішу для прослуховування і намагається повторити слово. Якщо слово вимовляється правильно, служіння переходить на нове слово. В іншому випадку служба показує, що вона розпізнала, і продовжує прослуховування. Якщо користувач не може правильно вимовити слово, він може натиснути клавішу, щоб перейти до наступного слова. Служба розпізнає вимову користувача за допомогою _0_ Web Speech API, реалізованого в більшості сучасних браузерів.

3.18 Слідкуйте за прогресом у вивченні слів

Кожне слово має три етапи вивчення: невивчене, вивчене у процесі навчання. При генерації практичних завдань сервіс пропонує спочатку вивчити невчені слова, а вже потім вже учні.

Після кожного тренування клієнт надсилає результати відповідей на сервер. Якщо відповідь правильна, сервер переходить до етапу вивчення слова на наступному рівні, інакше він встановлює початковий рівень. Вивчені слова сервер не пропонує практики. Ці слова можна повторювати лише у режимі повтору. Вивчені слова відображаються для користувача одне за одним, і користувач повинен вибрати, запам'ятовувати це слово чи ні. Якщо ви не пам'ятаєте, то сервер перекладає слово початковому рівні навчання.

Цей підхід заснований на системі Лейтнера, яка поділяє інформацію на групи відповідно до рівня навчання. Перша група містить інформацію, яку важко засвоїти або яка є новою, тоді як остання група містить інформацію, яка майже засвоєна.

У розробленому програмному продукті реалізовано всі функціональні вимоги. Для полегшення виконання всіх вимог використовувалися такі сторонні послуги: Amazon S3, Amazon Polly, Yandex Translate і Flickr. Високий рівень підтримки продукту досягається за рахунок використання інтеграційних тестів та документації публічного інтерфейсу на стороні сервера.

3.19 Взаємодія користувача с сервісом

Коли користувач зайде на наш сервіс перше що він побачить це буде вікно з авторизацією користувача (рисунок 3.33).

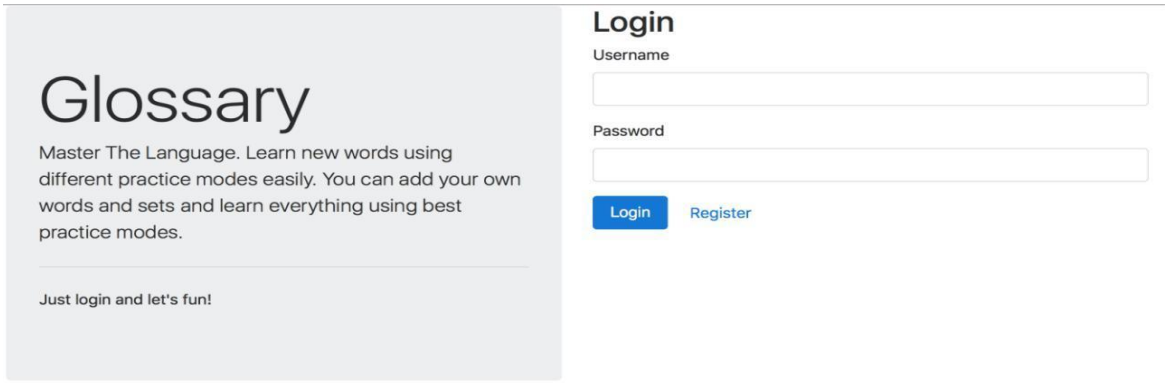

Рисунок 3.33 – Вікно входу у сервіс

Якщо користувач не зареєстрований в системі зразу з головної сторінки можна перейти на сторінку реєстрації нового користувача.

Рисунок 3.34 – Вікно реєстрації нового користувача

Коли користувач авторизується він зможе побачити статистику: кількість вивчених та доданих слів (рис. 3.35).

Користувач може переглядати список наборів слів також добавляти нові.Набори слів динамічно переміщуються при зміні їх кількості та розташовуються у вигляді сітки, ці набори також можна видаляти та редактувати.

Рисунок 3.35 – Коротка статистика

У вікні перегляду слів також відображається інформація про всі слова з набору (рис. 3.36).

Поруч із кожним словом є зображення та статус вивчення. Клацніть по значку звуку ввімкне звукову вимову слова.

У цьому вікні ви можете додавати слова. Для цього вам просто потрібно запровадити слово іноземною мовою.

Користувач може вибрати один із запропонованих перекладів або ввести свою версію вручну.

Після цього користувач може вибрати зображення зі списку запропонованих або завантажити .

Вибір зображень ділиться на 2 етапи: вибір із запропонованих зображень або завантаження власного. Як варіант, можна додати слово без зображення. Це ускладнить процес вивчення нових слів.

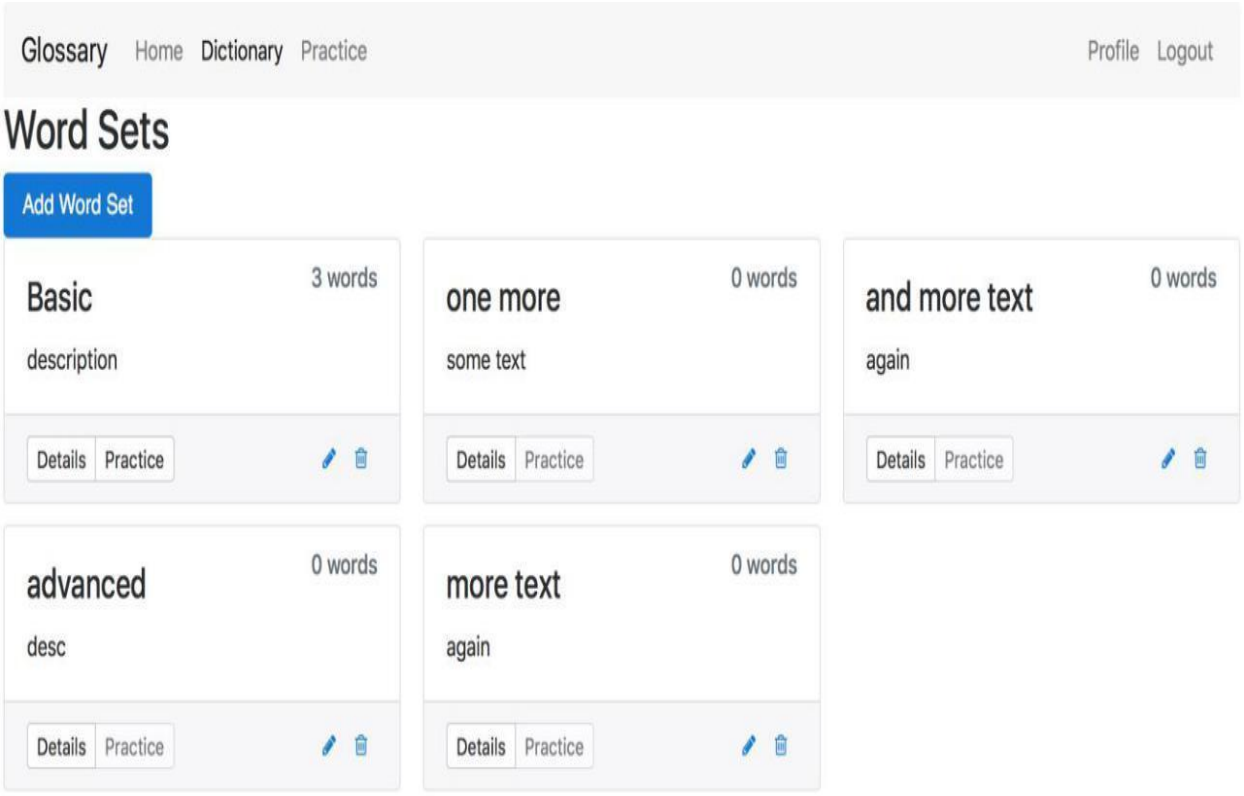

Рисунок 3.36 – Список наборів слів

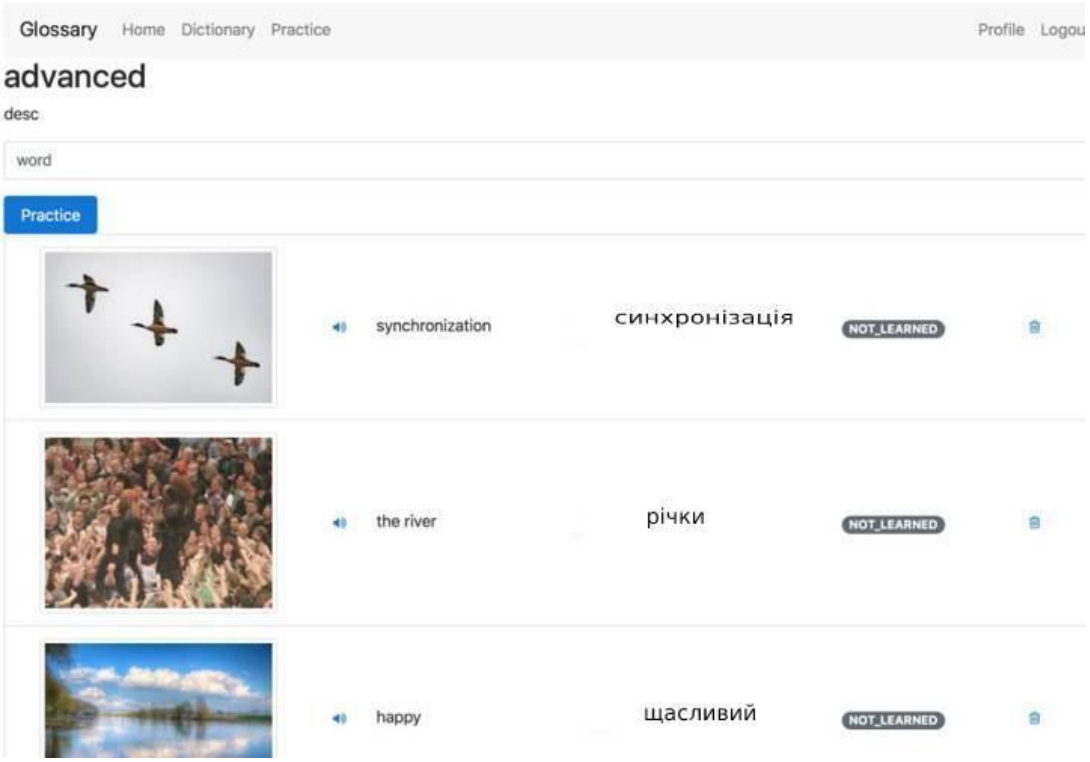

Рисунок 3.37 – Набір слів

Після додавання нових слів користувач може перейти в режими практики (рис. 3.39). Реалізовано 6 тренувальних режимів:

– стандартні тести із вибором правильної відповіді;

– режим із написанням відповідей вручну;

– слухання;

– навчання вимові;

– літери зі словами та їх переклад на інший бік;

– режим повтору, у якому випадатимуть лише вивчені слова.

Для деяких режимів практики ви можете вибрати напрямок тестових питань.

Іншими словами, завдання буде полягати в тому, щоб вибрати правильний переклад або навпаки, залежно від перекладу, вибрати правильне слово.

Існує спеціальна кнопка для повернення до набору слів. Ви також можете змінити словниковий запас на тій самій сторінці.

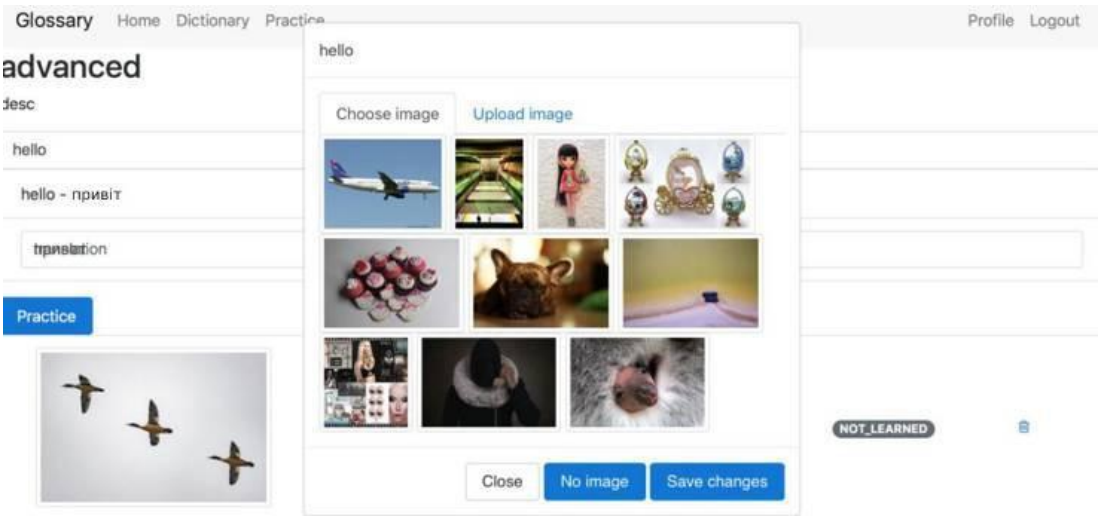

Рисунок 3.38 – Вибір зображення

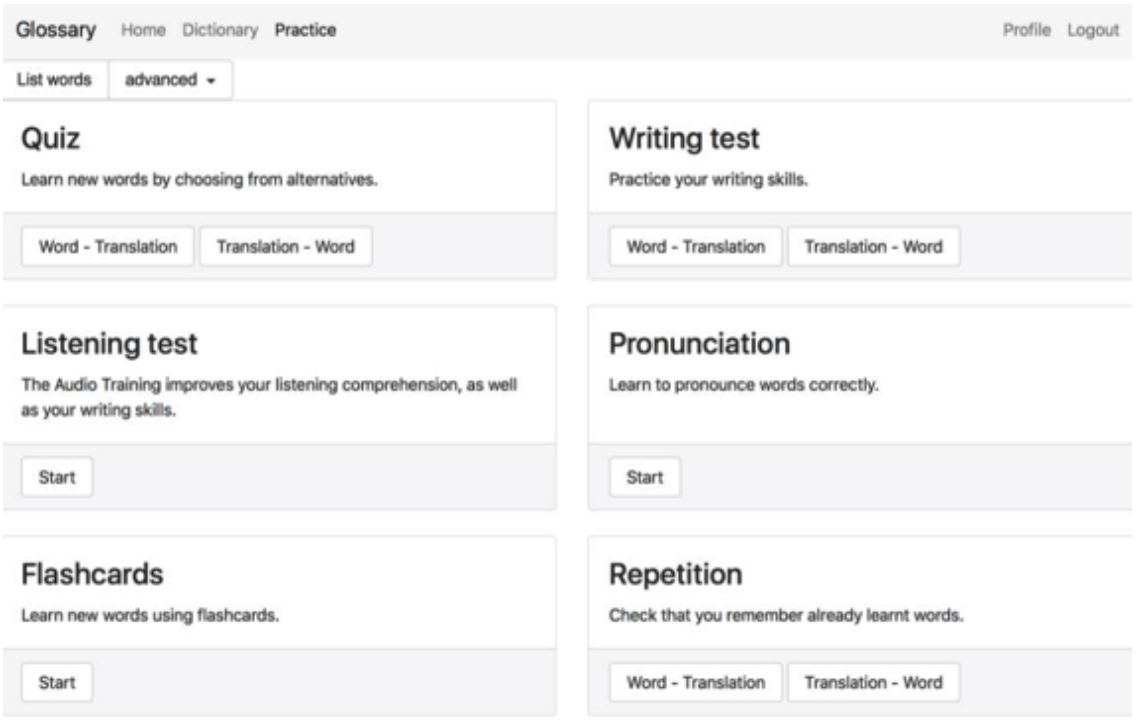

Рисунок 3.39 – Режими практики

Користувач може переглянути інформацію про свій профіль на окремій сторінці (рис. 3.40).

Також є окремий блок для зміни налаштувань, де можна змінити ім'я користувача, адресу електронної пошти та пароль (рисунок 3.41).

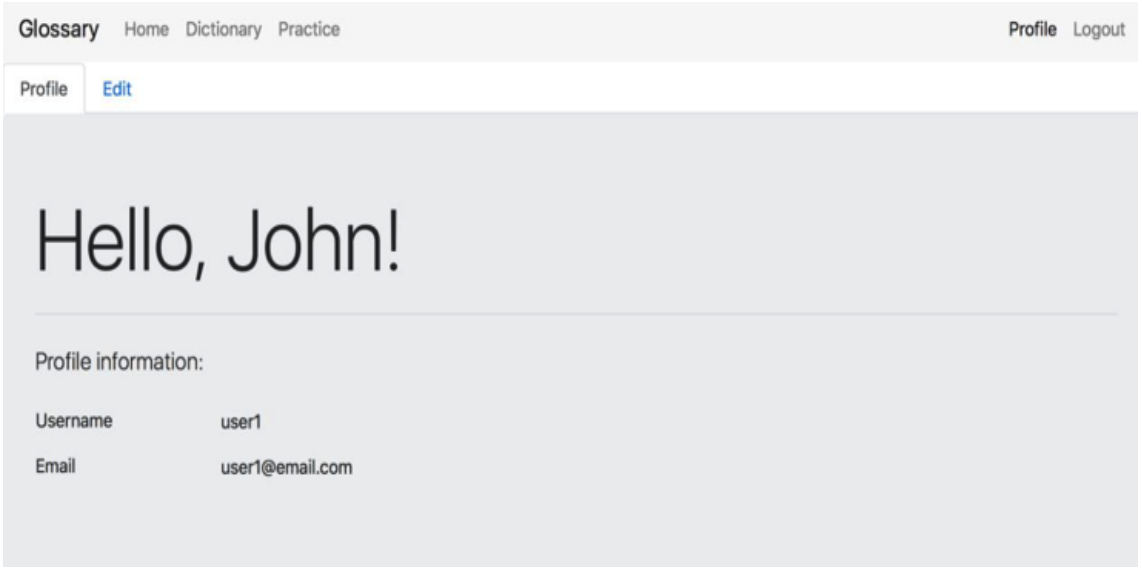

Рисунок 3.40 – Сторінка профіля

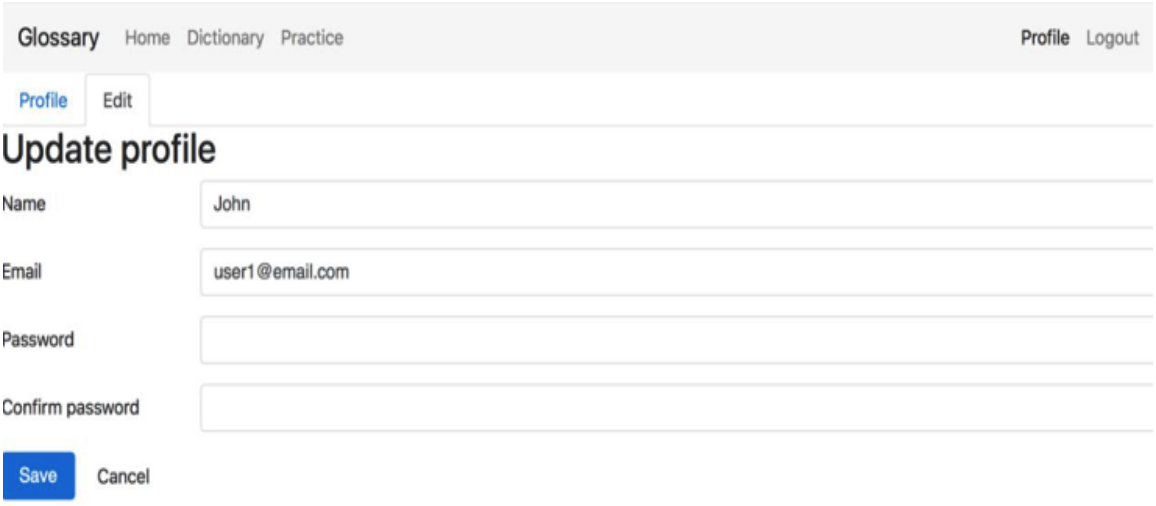

Рисунок 3.41 – Редагування профіля

Для виходу з системи потрібно натиснути на посилання «Logout» у правому верхньому кутку.

Після цього вся інформація про користувача видаляється з локального сховища браузера і сервіс перенаправляє на сторінку входу в систему.

Оскільки інтерфейс веб– клієнта побудований за допомогою Bootstrap 5, то він автоматично має адаптивний дизайн для коректного відображення на мобільних пристроях.

Висновки до розділу 3

В системі були реалізовані підходи мікросервісної архітектури, такі як єдиний шлюз для API, повтори невдалих запитів, єдиний реєстр всіх сервісів та переривач запитів.

Взаємодія з базою даних була реалізована через Java Persistence API, що дозволяє зберігати об’єкти у реляційній базі даних. Для автентифікації та авторизації був використаний Spring Security, а сама автентифікація відбувається за допомогою передачі JWT токенів, що дозволяють перевірити справжність клієнта без додаткових звернень в базу даних.

Для розгортання системи був застосований Kubernetes, що дозволяє відстежувати кожен з екземплярів сервісів та автоматично масштабувати сервіси за потребою.

ВИСНОВКИ

Була описана архітектура хмарних інфраструктур, основні категорії стійкості систем та їх види метрик. Були проаналізовані основні підходи до побудови стійкої хмарної інфраструктури.

В результаті дипломної роботи була побудована відмовостійка хмарна система, що надійно працює в умовах хмарної інфраструктури та гарантує толерантність до помилок різного рівня. Розроблена система має можливості автоматичного масштабування як за метриками ресурсів, так і за метриками користувача. Мікросервісна архітектура дозволила точно налаштувати продуктивність системи залежно від навантаження на кожен з сервісів системи. Застосування мікросервісної архітектури дало простір для масштабування, гнучкості та відмовостійкості, але привнесло додаткові навантаження на сервери, ускладнило розробку та тестування.

Використання Kubernetes дозволило відстежувати кожен з екземплярів сервісів та автоматично масштабувати сервіси за потребою. Реалізація автоматичного масштабування дозволила збудувати систему, що підлаштовуються під поточне навантаження. Цей підхід не тільки підвищив стабільність системи, але й дозволив зберегти ресурси, що дозволило мінімізувати обсяг необхідних ресурсів, коли рівень навантаження мінімальний. Використання користувацьких показників покращило точність

автоматичного масштабування та надало широкий спектр можливостей налаштування розподіленої системи.

Клієнт реалізовано у вигляді веб– застосунку, що може бути доступний з будь–якого браузеру. Застосунок має адаптивний дизайн, що дозволяє користуватися ним як з великих настільних комп'ютерів, так і з мобільних телефонів. За результатами виконання інтеграційних та ручних тестів підтверджена коректність отриманих результатів, отже система відповідає поставленим вимогам.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. MDN web docs – Вступ в JavaScript: веб-сайт. URL: https://developer.mozilla.org/JavaScript/Guide/Introduction (дата звернення: 12.06.2021).
2. MDN web docs - Основи CSS веб-сайт. URL: https://developer.mozilla.org/uk/docs/Learn/CSS_basics. (дата звернення: 12.06.2021).
3. Mongo DB // веб-сайт. URL: https://www.mongodb.com/. (дата звернення: 10.10.2021).
4. Wikipedia. MVC // веб-сайт. URL: https://uk.wikipedia.org/wiki/модель-вид-контроллер. (дата звернення: 10.05.2021).
5. Wikipedia. UML // веб-сайт. URL: https://uk.wikipedia.org/wiki/Unified_Modeling_Language. (дата звернення: 10.10.2021).
6. Wikipedia. Сценарій використання // веб-сайт. URL: https://uk.wikipedia.org/wiki/Сценарійвикористання. (дата звернення: 10.05.2021).

7. Ашманов И.С., Продвижение сайта в поисковых системах / И.С. Ашманов, А.А. Иванов. – М.: Вильямс, 2007. – 304 с. 2.
8. Гото К. Веб-редизайн / К. Гото, Э. Котлер – М.: Символ-Плюс, 2006. – 416 с.
9. Гото Келлі Веб-редизайн, 2-е видання / Котлер Емілі. – СПб.: Символ Плюс, 206. – 416 с.
10. Завадський І. О. Основи баз даних / І. О. Завадський. – Київ: ПП І.О. Завадський, 2011. – 192 с.
11. Кантор Марри. Управление программными проектами. Издательство: Вильямс, 2002 г. — 176 стр.
12. Петлюшкин А.В., HTML в Web-дизайне. – СПб.: БВХ-Петербург, 2004. – 400 с.: ил.
13. Постановка задачи при проектировании сайта. – веб-сайт. URL: http://site-manager.ru/webmasters/lib/110/. (дата звернення: 12.08.2021).
14. Програмування по-українськи - А що таке HTML? веб-сайт. URL: http://programming.in.ua/web-design/html/73-html-introdution.html. (дата звернення: 13.07.2021).
15. Прототип сайту – як він допомагає оцінювати і розробляти сайти веб-сайт. URL: http://evergreens.com.ua/ua/articles/prototype-site.html. (дата звернення: 10.07.2021).
16. Севостьянов И.О., Поисковая оптимизация. Практическое руководство по продвижению сайта в Интернете / И.О. Севостьянов. – СПб.: Питер, 2010. – 240 с. 9.
17. Современный учебник Javascript Введение в – AJAX веб-сайт. URL: [https://learn.javascript.ru/ajax-intro. (дата звернення: 22.10.2021).
18. Типи сайтів // веб-сайт. URL: https://internetdevels.ua/blog/most-common-types-of-websites . (дата звернення: 10.10.2021).

19. Уэйншенк С. Интуитивный веб-дизайн / С. Уэйншенк – М.:, Эксмо, 2011. – 160 с.
20. Фрейн Б. HTML5 и CSS3. Разработка сайтов для любых браузеров и устройств / Бен Фрейн – СПб.: Питер, 2014. – 304 с.
21. Ken Pugh Interface Oriented DesignWith Patterns. / Publisher: Pragmatic Bookshelf, July 2006 – 240p.
22. Eric Elliott Programming JavaScript ApplicationsRobust Web Architecture With Node, HTML5, and Modern JS Libraries. / Publisher: O'Reilly Media, February 2013. – 300p.
23. API в веб-приложениях веб-сайт. URL: https://mkdev.me/posts/chto-takoe-api-v-veb-prilozheniyah-i-zachem-on-nuzhen. (дата звернення: 03.10.2021).
24. Working with Web API : веб-сайт. URL: https://launchschool.com/books/working_with_apis (дата звернення: 04.10.2021).
25. PSR-7: HTTP message interfaces: веб-сайт. URL: http://www.php-fig.org/psr/psr-7/. (дата звернення: 17.10.2021).
26. SOAP и REST/ Л. Черняк. – 2003. – веб-сайт. URL: http://www.osp.ru/os/2003/09/183376/ (дата звернення: 10.07.2021).
27. Интеграция приложний в WWW: веб-сайт. URL: http://www.4stud.info/networking/web-services.html (дата звернення: 19.07.2021).
28. API веб-сайт. URL: http://progschool.ru/products/webapi/ (дата звернення: 25.09.2021).
29. Learn API testing веб-сайт. URL: http://www.guru99.com/api-testing.html (дата звернення: 14.10.2021).
30. SOAP-vs-REST_веб-сайт.URL: http://www.alliancetek.com/Blog/post/2012/10/12/SOAP-vs-REST.aspx (дата звернення: 10.07.2021).

31. OpenLibraryBooks API веб-сайт. URL: http://openlibrary.org/dev/docs/api/books (дата звернення: 11.09.2021).
32. Webservices.API.веб-сайт.URL: http://research.cs.wisc.edu/htcondor/manual/v7.6/4_5Applicat ion_Program.html (дата звернення: 27.09.2021).
33. Kevin MakiceTwitter API: Up and RunningLearn How to Build Applications with the Twitter API / Publisher: O'Reilly Media, March 2009. – 416p.
34. James McGovernJava Web Services Architecture / James McGovern, Sameer Tyagi, Michael Stevens, Sunil Mathew / Publisher: Elsevier, May 2003. – 832p.
35. Stephen T. The Art of Software ArchitectureDesign Methods and Techniques / Publisher: WileyReleased, April 2003. – 336p.
36. H2 Database Engine веб-сайт.URL: http://www.h2database.com/html/main.html (дата звернення: 15.10.2021).
37. JWT веб-сайт.URL:https://jwt.io (дата звернення: 13.09.2021).
38. Amazon Web Services веб-сайт.URL: https://aws.amazon.com (дата звернення: 19.11.2021).
39. Gradle Build Tool веб-сайт.URL:https://gradle.org (дата звернення: 22.10.2021).
40. Docker [Електронний ресурс] — Режим доступу: https://docker.com (дата звернення: 26.10.2021).