МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

Кваліфікаційна робота магістра

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ОПТИМІЗАЦІЇ ДОСТУПНОСТІ ТА ШВИДКОДІЇ СЕРВЕРНИХ ДОДАТКІВ

Здобувач освіти гр. ІНм. 12ан/2у Станіслав ШУЛЬГА

Науковий керівник Оксана ШОВКОПЛЯС
к.ф.-м.н.

В.о.завідувач кафедри Ігор ШЕЛЕХОВ
доц., к.т.н.

Суми 2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ЕЛЕКТРОНІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Затверджую_____
В.о. зав. кафедри Шелехов І.В.
«_____» _____2022 р.

ЗАВДАННЯ
до кваліфікаційної роботи

здобувача вищої освіти другого курсу магістратури, групи ІН.м-12ан спеціальності «122 – Комп'ютерні науки» денної форми навчання Шульги Станіслава Олексійовича.

Тема: «Інформаційна технологія оптимізації доступності та швидкодії серверних додатків»

Затверджена наказом по СумДУ
№_____ від _____2022 р.

Зміст пояснювальної записки: 1) Огляд технологій розробки та існуючих рішень. 2) Формування мети та постановки задачі, визначення етапів реалізації проєкту. 3) Вибір технологій та інструментів, що необхідні для розробки, огляд та аналіз баз даних. 4) Проєктування бази даних. 5) Проектування системи. 6) Розробка веб-додатку. 7) Тестування готового програмного продукту та аналіз результатів.

Дата видачі завдання «_____»_____ 2022 р.

Керівник роботи _____ О. А Шовкопляс

Завдання прийняв до виконання _____ С. О Шульга

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Термін виконання проєкту (роботи)	Примітка
1.	Огляд технологій розробки та аналіз існуючих рішень		
2.	Формування мети та постановка задачі		
3.	Вибір технологій та інструментів для розробки		
4.	Проектування бази даних		
5.	Розробка веб-сервісу		
6.	Тестування готового програмного продукту		
7.	Оформлення пояснювальної записки до кваліфікаційної магістерської роботи		

Студент–дипломник _____
(підпис)

Керівник проєкту _____

РЕФЕРАТ

Записка: 71 стор., 21 рис., 1 додаток, 28 джерел.

Об’єкт дослідження – комп’ютерні методи проєктування REST API, що може швидко та легко горизонтально масштабуватися, та витримувати великі навантаження.

Мета роботи – створити веб сервіс, API сайту з можливістю взаємодії з ним, що буде мати зазначений функціонал з використанням Python та фреймворку FastAPI.

Методи дослідження – методи оптимізації запитів при роботі з різними базами даних, кешування, кодовою базою, а саме, алгоритм використання асинхронного програмування.

Результати – створений веб-сервіс, API сайту з можливістю взаємодії з ним, що буде мати зазначений функціонал з використанням Python та фреймворку FastAPI.

REST API, Python, FastAPI, Redis, кешування, PostgreSQL, Docker, Pydantic

Зміст

ВСТУП

Постійний розвиток онлайн бізнесу, підкріплений розвитком інформаційних технологій спричиняє майже експоненціальний ріст [1] кількості даних різних видів: дані про користувачів, логи, метрики тощо. Їх треба зберігати та аналізувати, а це в свою чергу задає нові стандарти до швидкості опрацювання цих даних та створює потребу в оптимальному використанні наявних ресурсів, таких як сервери.

Для того, щоб швидко та ефективно обробляти за зберігати дані різних типів використовуються різні типи баз даних. Термін «база даних» є доволі обширним, тому що будь-який формат збереження даних, у якому є структура, може бути названим базою даних, наприклад excel таблиця або csv файл. Однак, найбільшою популярністю зараз користуються реляційні бази даних у випадку коли дані чітко структуровані, та нереляційні – коли дані не мають чіткої структури або їх не треба зберігати довгий час. Реляційна модель даних була введена Е. Ф. Коддом у 1970 році. На даний час це найбільш широко використовувана модель даних.

Реляційна модель стала основою для:

- Дослідження теорії даних/зв’язків/обмежень.
- Чисельних методологій проєктування баз даних.
- Стандартної мови доступу до баз даних, яка називається мовою структурованих запитів (SQL).
- Майже всіх сучасних комерційних систем управління базами даних [2].

Збільшення кількості даних у реляційній базі даних призводить до погіршення швидкодії запитів. Для покращення швидкодії запитів в таких базах даних є багато способів оптимізації, але використання індексів є одним з найбільш ефективних. Іншим підходом до оптимізації роботи з даними може бути використання другорядної бази даних, такої як Redis, що може бути використана для кешування. Таким чином запит буде зроблений спочатку до кешу, а тільки потім, якщо немає результату, то запит буде надісланим до

основної реляційної бази. Через те, що більшість операцій в Redis мають алгоритмічну складність O(1) [3], запити будуть відпрацьовувати швидше ніж такі ж запити, але надіслані до реляційної бази даних, тому що їхня алгоритмічна складність O(n) [4].

Дана робота присвячена проблемі оптимізації запитів до баз даних, а саме необхідності аналізу типових запитів, можливості оптимізації за допомогою додавання індексів, а також можливості кешування їхніх результатів в Redis.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Методи оптимізації коду: асинхронне програмування

Починаючи з 1995 року кількість користувачів в інтернеті неспинно росла. Влітку 2022 їхня кількість приблизно дорівнювала 5 мільярдам 470 мільйонам [5].

Зокрема, в дослідженні “A Minute on the Internet in 2021” наводяться дані про те, що за кожну хвилину на сервіс Youtube завантажується близько 500 годин відео, майже 700 тисяч історій публікуються у Instagram та 1.6 мільярда доларів витрачається [6]. Всі види інтернет-активностей генерують певну кількість даних, яку зберігають та аналізують компанії. Згідно з дослідженням statista (рис. 1.1), у 2022 році сукупна кількість інформації у всьому інтернету становить 97 зетабайт, що дорівнює 97000000 петабайт, та ця цифра буде зростати [7].

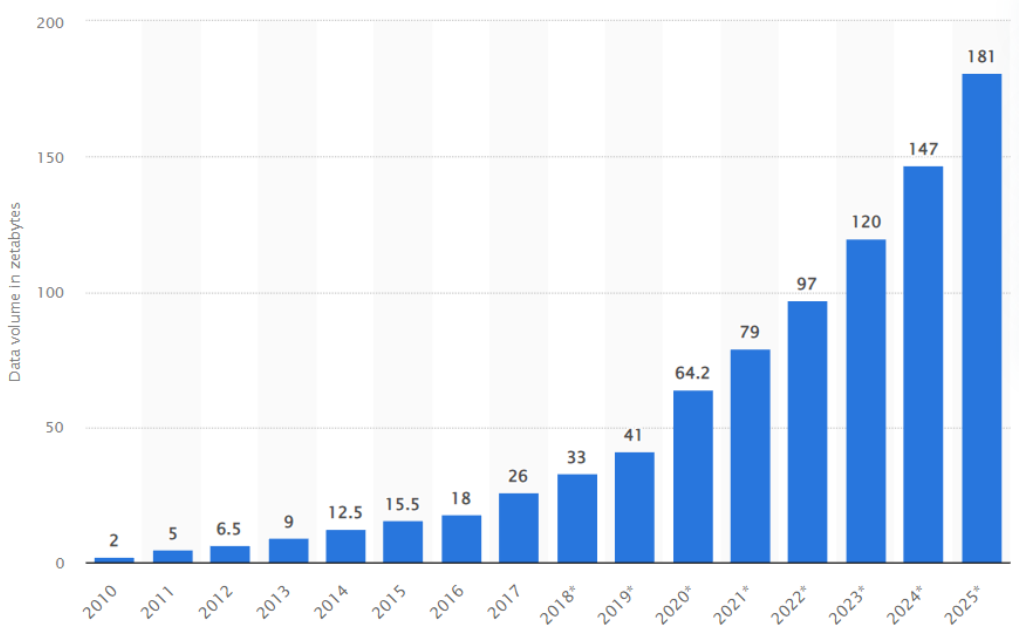

Рисунок 1.1 – Динаміка росту кількості даних в інтернеті [7]

Збільшення кількості даних в цілому в інтернеті створює декілька ключових проблем з точки зору коду, серед яких:

- Необхідність додавання ресурсів до сервера для обслуговування більшої кількості клієнтів одночасно.
- Необхідність змінення архітектури під використання потоків або асинхронності.

Проблема додавання ресурсів до серверу має 2 ключових рішення.

Вертикальне масштабування – це додавання ресурсів до одного сервера, таких як ресурси процесора, оперативної пам’яті тощо, для того щоб сервер одночасно обслуговувати більше кількість користувачів. Цей варіант не є ідеальним, тому що кількість ресурсів для додавання є обмеженою, наприклад, не можна добавити більше 1 процесора для сервера. Незважаючи на це, багато компаній використовуюсь саме цей варіант, тому що він простий і не вимагає змін у коді [8].

Горизонтальне масштабування – додавання копій серверів, потужність кожного з яких не обов’язково є дуже високою, таким чином, що навантаження балансується та розподіляється між ними [8]. Цей варіант часто використовується великими компаніями, тому що кількість копій не обмежена, тож фактично, можна масштабуватися скільки завгодно. Кожна окрема копія сервісу може навіть не знати, що є інші копії, тому що кожна з них працює без покладання на інші копії (рис. 1.2).

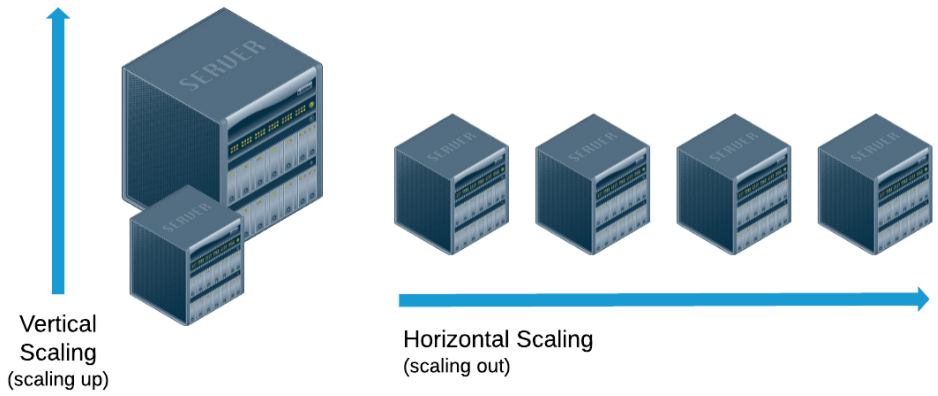

Рисунок 1.2 – Різниця між вертикальним та горизонтальним масштабуванням [8]

Щоб горизонтально масштабувати сервіси, зазвичай використовуються контейнери.

Контейнеризація – це пакування програмного коду лише з бібліотеками операційної системи (ОС) та залежностями, необхідними для запуску коду, для створення єдиного легкого виконуваного файлу, який називається контейнером, що стабільно працює на будь-якій інфраструктурі. Більш портативні та ресурсоефективні, ніж віртуальні машини (ВМ), контейнери стали де-факто обчислювальними одиницями сучасних хмарних додатків. Контейнери часто називають «легкими», що означає, що вони використовують ядро операційної системи машини і не вимагають накладних витрат, пов'язаних з підключенням операційної системи в кожному додатку. Контейнери за своєю суттю менші за обсягом, ніж віртуальні машини, і вимагають менше часу для запуску, що дозволяє набагато більшій кількості контейнерів працювати на тій же обчислювальній потужності, що і одна віртуальна машина (рис. 1.3). Це сприяє підвищенню ефективності роботи серверів і, в свою чергу, знижує витрати на сервери та ліцензування [12].

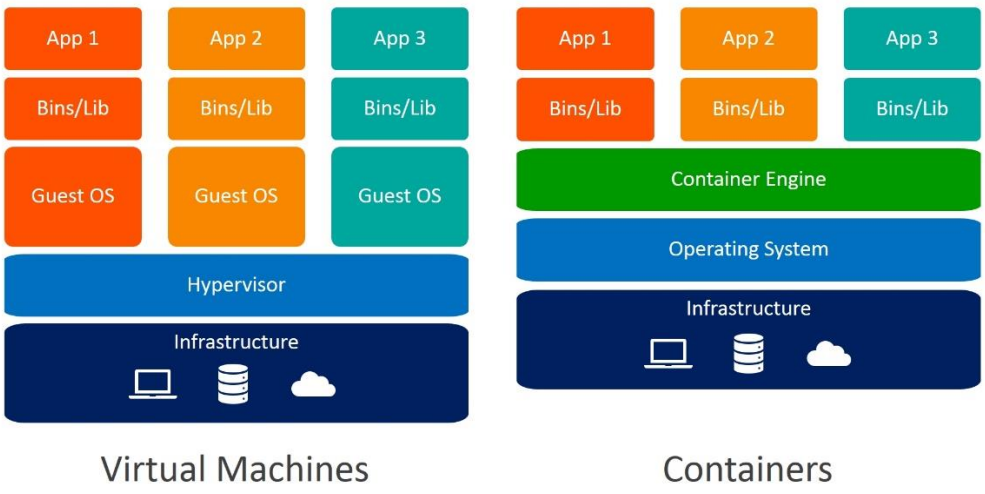

Рисунок 1.3 – різниця між контейнерами та віртуальними машинами [12]

Можливим рішенням другої проблеми може бути перехід до асинхронного стилю написання коду, замість багатопоточності і ось чому:

1. Дуже важко написати код, який є безпечним для потоків. З асинхронним кодом ви точно знаєте, де код буде переходити від однієї задачі до іншої, і тому умови перегонів набагато складніше створити.
2. Потоки споживають значну кількість даних, оскільки кожен потік повинен мати свій власний стек. В асинхронному коді весь код використовує один і той же стек, і стек залишається невеликим завдяки постійному розгортанню стека між завданнями.
3. Потоки є структурами операційної системи і, отже, вимагають більше пам'яті для підтримки платформою. З асинхронними завданнями такої проблеми немає.

Деякі процеси прив’язані до процесора: вони складаються з серії інструкцій, які повинні виконуватися одна за одною до тих пір, поки не буде обчислено результат. Весь час, поки вони працюють, вони повністю використовують можливості комп'ютера. Інші процеси, однак, прив'язані до вводу-виводу: вони витрачають багато часу на надсилання та отримання даних від зовнішніх пристроїв або процесів, а отже, часто повинні розпочати операцію, а потім дочекатися її завершення, перш ніж продовжити. Під час очікування вони не роблять дуже багато. Коли програма виконує код з прив'язкою до вводу-виводу, досить часто процесор витрачає багато часу на те, щоб взагалі нічого не робити, тому що єдине, що зараз робиться, - це очікування чогось в іншому місці [9].

Саме під час очікування програма може робити інші речі, які не залежать від вводу-виводу, тож їх не треба чекати. Тож асинхронність у програмуванні дозволяє структурувати код таким чином, що коли код стикається з очікуванням, процесор не буде простоювати в очікуванні відповіді, а буде займатися іншими задачами. Йдеться не про використання декількох ядер, а про більш ефективне використання одного ядра.

Більшість мов програмування мають методи, які слідують так званій «підпрограмній» моделі виклику. У цій моделі виклику кожного разу, коли викликається функція, виконання переміщується на початок цієї функції, потім продовжується до тих пір, поки не досягне кінця цієї функції (або оператора повернення), після чого виконання повертається в точку, що знаходиться безпосередньо після виклику функції, будь-які наступні виклики функції є незалежними викликами, які починаються знову на початку.

Однак існує альтернативна модель виконання коду, яка називається «корутинною» моделлю виклику. У цій моделі виклику існує новий спосіб для методу (який називається підпрограмою) передати виконання назад тому, хто його викликає: замість повернення він може «віддати» управління. Коли підпрограма «віддає» управління, виконання повертається до точки, з якої вона була викликана, але наступні виклики підпрограми не починаються з самого початку, а продовжуються з того місця, де виконання зупинилося востаннє (рис. 1.4). Таким чином, управління може переходити між кодом виклику та кодом підпрограми, як показано на наступній діаграмі [9]:

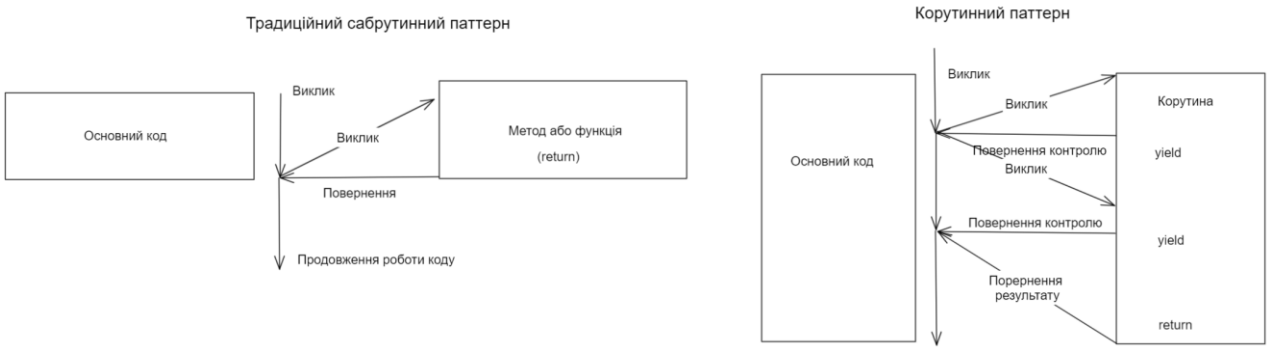

Рисунок 1.4 – Різниця між підпрограмним та корутинним моделями у програмуванні

Гарним прикладом асинхронності в реальному житті є шахова виставка 1992 року за участі Джудіт Полгар. Коли вона проводить шахову виставку, в якій грає з кількома гравцями-аматорами. Вона має два способи проведення виставки: синхронно і асинхронно.

Припущення:

- 24 суперники
- Джудіт робить кожен хід за 5 секунд
- Суперники витрачають по 55 секунд на хід
- Партія в середньому складається з 30 парних ходів (всього 60 ходів)

Синхронна версія: Джудіт грає одну партію за раз, ніколи не дві одночасно, поки гра не буде завершена. Кожна гра триває (55 + 5) * 30 = 1800 секунд, або 30 хвилин. Вся виставка займає 24 * 30 = 720 хвилин, або 12 годин.

Асинхронна версія: Джудіт переходить від столу до столу, роблячи по одному ходу за кожним столом. Під час очікування вона залишає стіл і дає можливість супернику зробити наступний хід. Один хід на всіх 24 іграх займає у Джудіт 24 * 5 = 120 секунд, або 2 хвилини. Вся виставка тепер скорочується до 120 * 30 = 3600 секунд, або всього 1 годину [10, 11].

Якщо повернутися до технічної частини, то Джудіт це процесор, суперники – це задачі, що треба зробити, час, що суперники витрачають на хід – очікування вводу-воду.

Іншим важливим концептом в асинхронному програмуванні є об'єкт, який називається цикл обробки подій (Event Loop) (рис. 1.5). Це свого роду менеджер задач, який має список задач, які має виконати процесор. В будь-який момент часу може бути лише одна активна задача, тому що процесор може робити тільки одну річ одночасно, а всі інші будуть призупинені. Асинхронний код виконується так само як і синхронний до того як процесору треба буде чекати, замість очікування задача віддасть контроль циклу обробки подій, а він у свою чергу вибере іншу задачу для процесора.

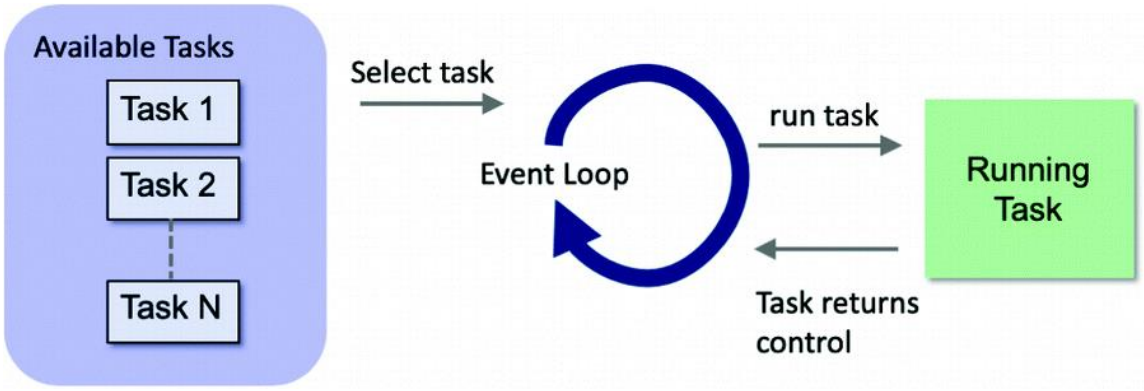

Рисунок 1.5 – Алгоритм вибору задач циклом обробки подій [9]

1.2 Методи оптимізації запитів у реляційних баз даних

Зростання кількості даних в інтернеті також створює декілька ключових проблем і для баз даних, серед яких:

- Проблема зберігання великої кількості даних
- Проблема аналізу великих даних

Вирішенням проблеми зберігання великої кількості даних є впровадження політики зберігання даних – retention policy. Компанія визначає періоди зберігання для кожного типу даних, а після його завершення видаляє ці дані. Наприклад, логи зберігаються 3 тижні, після чого видаляються, дані про неактивних користувачів – 1 рік.

Стосовно проблеми аналізу великих даних, є декілька можливих рішень. Перехід до іншого типу бази даних, а відповідно моделі даних. Цей варіант дуже важно реалізувати, тому що треба правильно мігрувати дані с одної бази в іншу, а це не завжди можливо через те, що вони підтримують різні моделі даних, наприклад буде важно або неможливо без втрат перенести дані з реляційної бази даних в нереляційну чи навпаки.

Ще одним способом пришвидшення аналізу великих даних є використання декількох різних баз даних для різних типів даних, а також створення реплік баз даних. Так, наприклад, логи будуть знаходитися в

Elasticsearch, неструктуровані дані будуть знаходитися в Google Cloud Firestore або MongoDB, а основні структуровані бізнес дані вже будуть в реляційній базі, такій як Oracle чи PostgreSQL. Реплікація баз даних (рис. 1.6) значить, що є основна база даних, в яку здані записуються (або видаляються) і у неї є копії, які підтягують зміні з основної, саме з цих копій дані читаються, тому що операція читання є більш ресурсозатратною ніж запис.

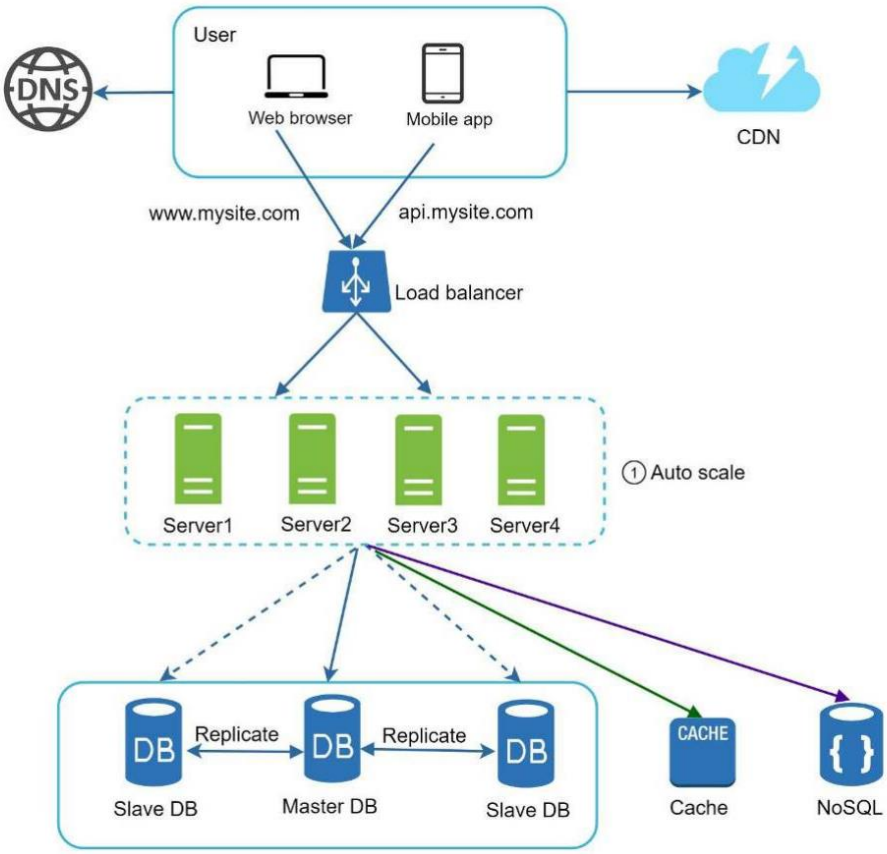

Рисунок 1.6 – Використання різних баз даних та їхніх реплік

Створення індексів до таблиць – хороший програмний спосіб прискорення запитів.

Індекс – це структура даних, яка допомагає зменшити час пошуку запитуваних даних. Індекси досягають цього за рахунок додаткових витрат на зберігання, пам'ять і підтримання її в актуальному стані (повільніший запис), що дозволяє нам уникнути перевірки кожного рядка таблиці. Перед тим, як

добавляти індекси на колонки, треба спершу провести аналіз і зрозуміти, по яких полях частіше всього відбувається пошук.

При виконанні запиту на база даних перевірить, чи існує індекс для стовпця (стовпців), який (які) запитується (запитуються). Якщо припустити, що стовпець має індекс, база даних повинна буде вирішити, чи є сенс використовувати індекс для пошуку шуканих значень, оскільки існують деякі сценарії, коли використання індексу бази даних є менш ефективним, а більш ефективним є просто сканування всієї таблиці.

Існує багато видів індексів, основні з них:

- B-дерево індекс – це самобалансуюче дерево, яке зберігає відсортовані дані і дозволяє здійснювати пошук, вставку, видалення та послідовний доступ за логарифмічним часом. Планувальник запитів PostgreSQL розгляне можливість використання індексу B-дерева щоразу, коли стовпці індексу беруть участь у порівнянні, яке використовує один з наступних операторів: < , <=, >=, > BETWEEN, IN, IS NOT NULL, IS NULL [14]
- Хеш індекс – можуть обробляти тільки просте порівняння на рівність (=). Це означає, що кожного разу, коли індексований стовпець бере участь у порівнянні за допомогою оператора рівності (=), планувальник запитів буде розглядати можливість використання хеш-індексу [14].
- Унікальний індекс – створюється коли колонка має обмеження на унікальні значення
- Первинний індекс – створюється на основі первинного ключа таблиці. Ці первинні ключі є унікальними для кожного запису і містять відношення 1:1 між записами. Оскільки первинні ключі зберігаються у відсортованому порядку, виконання операції пошуку є досить ефективним.

1.3 Кешування даних як спосіб пришвидшення обробки запитів

В обчислювальній техніці кеш - це високошвидкісний рівень зберігання даних, який зберігає підмножину даних, зазвичай тимчасових за своєю природою, таким чином, щоб майбутні запити на ці дані обслуговувалися швидше, ніж це можливо при доступі до первинного місця зберігання даних. Кешування дозволяє ефективно повторно використовувати раніше отримані або обчислені дані. Як саме працює кешування?

Дані в кеші, як правило, зберігаються в апаратних засобах швидкого доступу, таких як оперативна пам'ять, а також можуть використовуватися в поєднанні з програмним компонентом. Основна мета кешу - підвищити продуктивність пошуку даних за рахунок зменшення потреби в доступі до базового більш повільного рівня зберігання (рис. 1.7).

В обмін на швидкість кеш зазвичай зберігає підмножину даних тимчасово, на відміну від баз даних, дані в яких зазвичай є повними і довготривалими [15].

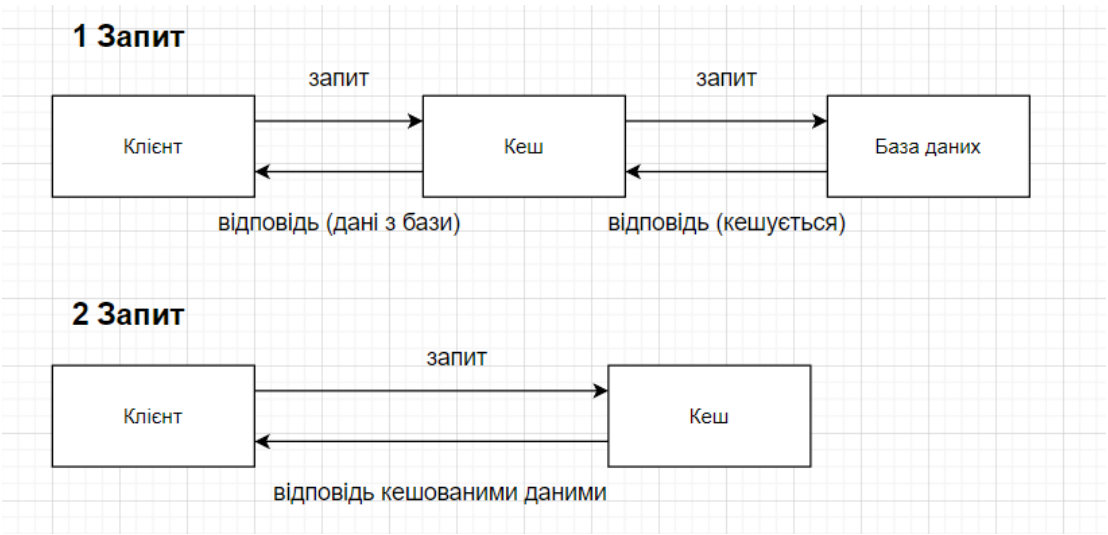

Рисунок 1.7 – Схема роботи кешу

Переваги використання кешування [15]:

- Підвищення продуктивності додатків – оскільки пам'ять працює на порядки швидше, ніж диск (HDD або SSD), читання даних з

кешу в пам'яті відбувається надзвичайно швидко (субмілісекунди). Цей значно швидший доступ до даних покращує загальну продуктивність програми.

- Зниження вартості бази даних – один екземпляр кешу може забезпечити сотні тисяч IOPS (операцій вводу-виводу в секунду), потенційно замінюючи кілька екземплярів бази даних, тим самим знижуючи загальну вартість. Це особливо важливо, якщо основна база даних стягує плату за пропускну здатність. У таких випадках економія коштів може становити десятки відсотків.
- Передбачувана продуктивність – поширеною проблемою в сучасних додатках є робота з піковими навантаженнями під час використання додатків. Приклади включають соціальні додатки під час Суперкубка або в день виборів, веб-сайти електронної комерції під час Чорної п'ятниці тощо. Підвищене навантаження на базу даних призводить до збільшення затримок при отриманні даних, що робить загальну продуктивність програми непередбачуваною. Використовуючи високопродуктивний кеш в пам'яті, цю проблему можна вирішити

Гарним рішенням для кешування є база даних типу ключ-значення – Redis. Він має високу продуктивність, 140500 запитів в секунду [16]. Така висока продуктивність обумовлена тим, що більшість операцій, таких додавання даних, отримання, видалення мають алгоритмічну складність O(1) [18]. Також, редіс є дуже універсальним рішенням, тому що він має багато структур даних, що можуть використовуватися в різних випадках. Через це, багато великих технологічних компаній використовують його, серед них: GitHub, Twitter, Pinterest та багато інших [17].

1.4 Мета роботи

Метою дипломної роботи є розробка сервісу Chop-chop з використанням асинхронного фреймворку для досягнення найкращої продуктивності, а також аналіз типових запитів до бази даних та додавання відповідних індексів, а також використання кешу для досягнення найвищої швидкості обробки запитів.

2 ІНФОРМАЦІЙНИЙ ОГЛЯД ПРОГРАМ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ

2.1 Обрання програмних методів реалізації

Є багато методів реалізації поставленої задачі, мови такі як Java, JavaScript, C# добре підходять для розборки серверного коду, але мовою програмування був обраний Python.

Java використовує об'єктно-орієнтоване програмування (ООП) - концепцію кодування, в якій ви не тільки визначаєте тип даних і їх структуру, але і набір функцій, що застосовуються до них. Таким чином, ваша структура даних стає об'єктом, яким тепер можна маніпулювати для створення взаємозв'язків між різними об'єктами. На відміну від іншого підходу - процедурного програмування, де ви повинні слідувати послідовності інструкцій, використовуючи змінні та функції, ООП дозволяє групувати ці змінні та функції за контекстом, таким чином маркуючи їх та звертаючись до функцій в контексті кожного конкретного об'єкту. Корпоративні додатки є найбільшою перевагою Java. Це почалося ще в 90-х роках, коли організації почали шукати надійні інструменти програмування, які не були мовою C. Java підтримує безліч бібліотек - будівельних блоків будь-якої корпоративної системи - які допомагають розробникам створювати будь-яку функцію, яка може знадобитися компанії. Величезний пул талантів також допомагає - Java є мовою, що використовується для вступу до комп'ютерного програмування в більшості шкіл та університетів. Крім того, її інтеграційні можливості вражають, оскільки більшість хостинг-провайдерів підтримують Java. І останнє, але не менш важливе, Java порівняно дешева в обслуговуванні, оскільки вам не потрібно залежати від конкретної апаратної інфраструктури і ви можете запускати свої сервери на будь-якому типі машин, які у вас можуть бути. Іншим плюсом Java є кросплатформеність (Write Once Run Anywhere)

Write Once Run Anywhere (WORA) - популярна фраза в програмуванні, введена компанією Sun Microsystems для опису крос-платформних можливостей мови Java. Це означало, що можна створити програму на Java, скажімо, під Windows, скомпілювати її в байт-код і запустити додаток на будь-якій іншій платформі, що підтримує віртуальну машину Java (JVM). У цьому випадку JVM слугує рівнем абстракції між кодом та апаратним забезпеченням. Всі основні операційні системи, включаючи Windows, Mac OS і Linux, підтримують JVM. І якщо ви не пишете програму, яка в основному покладається на специфічні для платформи функції та користувальницький інтерфейс, ви можете поділитися - можливо, не всім - але великим шматком байт-коду.

Одним з вагових недоліків Java є платна комерційна ліцензія. Нещодавно компанія Oracle оголосила, що з 2019 року почне стягувати плату за використання Java Standard Edition 8 для «ділового, комерційного або виробничого» використання. Щоб отримувати всі нові оновлення та виправлення помилок, потрібно буде платити за кількістю користувачів або за процесор. На сьогоднішній день поточна версія Java є безкоштовною та доступною для розповсюдження для обчислень загального призначення. Щоб підготуватися до змін, кожна компанія повинна оцінити, скільки Java вона використовує, і шукати альтернативну технологію, якщо підвищення ціни обіцяє бути занадто болючим. Іншим вагомим недоліком є багатослівний і складний код. Коли код багатослівний, це означає, що в ньому використовується занадто багато слів. Хоча це може здатися перевагою, коли ви намагаєтеся зрозуміти мову, довгі, надмірно складні речення роблять код менш читабельним і менш придатним для сканування. Намагаючись імітувати англійську мову, багато мов високого рівня створюють занадто багато шуму. Java, створена для того, щоб пом'якшити неприступну C++, змушує програмістів друкувати саме те, що вони мають на увазі, що робить мову більш прозорою для неспеціалістів, але, на жаль, менш компактною [25].

Стосовно мови JavaScript. Популярність є одним з ключових плюсів. Це одна з найпоширеніших мов, яка використовується у веб-розробці. Вона є основною частиною кожного функціонуючого веб-сайту. JavaScript вважається дуже потужним інструментом, який використовують навіть найбільші веб-сайти у світі, наприклад, Amazon та Google. Завдяки його постійно зростаючій популярності, вивчити цю мову онлайн за допомогою різноманітних курсів стало простіше, ніж будь-коли.

Найціннішою особливістю JavaScript є його універсальність. Існує незліченна кількість методів, за допомогою яких його можна впровадити на вашому веб-сайті. Він може не тільки створювати і завершувати фронтенд веб-сайту, але також може мати справу з бекендом (завдяки Node.js), оскільки він добре працює з MongoDB і MySQL [26]. JavaScript є однією з найпростіших мов для вивчення, особливо для веб-розробки. Вона була розроблена таким чином, щоб її було зручно вивчати та впроваджувати для веб-розробників. Це допомагає веб-компаніям заощаджувати значні кошти на розробці, оскільки складніші мови мають менше розробників, а отже, вимагають більшого бюджету. Основним недоліком цієї мови є безпека. Оскільки мова є клієнтською, доступ до цього коду дуже простий. Це означає, що будь-хто може побачити і скопіювати код, який реалізований за веб-сторінкою. Це робить мову значно небезпечнішою [26].

Python – широкопрофільна мова програмування, використовуючи його можна робити багато чого: аналіз даних, розробляти мобільні додатки, писати серверний код, автоматизувати щоденні завдання [21]. Основною цього перевагою є те, то що на ньому розробник може дуже швидко писати через його зрозумілість та простоту. Python є масштабованим, і на ринку є багато додатків, які це доводять. Instagram та Pinterest є одними з найпопулярніших додатків, які успішно отримують мільйони запитів від користувачів.

Instagram, Pinterest та Facebook є одними з найпопулярніших додатків у світі створеними за допомогою цієї мови. Python підтримує різні стилі та парадигми програмування, що робить його дуже гарним вибором майже для будь-якої задачі [27]. Також, Python має великий набір бібліотек і містить код для різних цілей, таких як генерація документації, регулярні вирази, веб-браузери, модульне тестування, CGI, бази даних, маніпулювання зображеннями тощо. Таким чином, усувається необхідність написання повного коду вручну.

У Python є багато фреймфорків, що використовуються для створення веб додатків, найпопулярнішими з них є: Django, Flask, FastAPI, Tornado, Aiohttp [20]. Різни фреймоворки покривають різні потреби в програмуванні, але найбільшою різницею є підтримка асинхронності, швидкість написання коду, а також швидкодія самого коду. Django та Flask є синхронними фреймворками, в той час як інші перелічені – асинхронними. FastAPI – це сучасний, високопродуктивний веб-фреймворк для побудови API мовою Python на основі стандартних підказок типів. Він має наступні ключові особливості:

- Швидкий у виконанні: Пропонує дуже високу продуктивність, на рівні з NodeJS і Go, завдяки Starlette і pydantic.
- Швидко кодується: Дозволяє значно збільшити швидкість розробки.
- Зменшена кількість помилок: Зменшує ймовірність помилок, спричинених людиною.
- Інтуїтивно зрозумілий: Пропонує чудову підтримку редактора, з завершенням скрізь і меншим часом налагодження.
- Простота: Він розроблений таким чином, щоб бути нескладним у використанні та вивченні, щоб ви могли витрачати менше часу на читання документації. Короткий: мінімізує дублювання коду.
- Надійний: надає готовий до виробництва код з автоматичною інтерактивною документацією.

- Заснований на стандартах: Заснований на відкритих стандартах для API, OpenAPI та JSON Schema.

Фреймворк призначений для оптимізації роботи розробника, щоб вони могли писати простий код для створення готових до виробництва API з використанням найкращих практик за замовчуванням [22]. Серед наведених фреймворків FastAPI був обраний для виконання задачі через фактор швидкодії та можливості обробки великої кількості запитів одночасно.

Серед баз даних є багато хороших варіанів, серед реляційних це: Oracle, MySQL, PostgreSQL, SQLServer. Серед нереляційних: MongoDB, Google Cloud Firestore, Elasticsearch та багато інших.

Як перевірена і надзвичайно гнучка система управління базами даних, Postgres використовується в численних галузях і сценаріях. Об'єктно-реляційна база даних є першокласною основою для безпечної роботи найрізноманітніших додатків. Наприклад, проект з відкритим вихідним кодом є ідеальним рішенням для програмного забезпечення інтернет-банкінгу, завдяки інтегрованій концепції транзакцій та підтримці MVCC (багатоверсійний контроль паралелізму: процедура для ефективного виконання конкуруючого доступу). Програми аналізу, такі як Matlab або R, також добре працюють з базою даних, саме тому PostgreSQL часто використовується в поєднанні з цими програмами. У поєднанні з розширенням PostGIS (яке надає сотні функцій для роботи з геоданими), Postgres також вражає, коли мова йде про роботу з просторовими та географічними даними.

PostgreSQL також користується попитом як рішення для веб-проектів: Об'єктно-реляційна система працює з різними сучасними фреймворками, такими як Django, Node.js або Ruby on Rails, і підтримує класичні веб-мови, такі як PHP. Підтримка синхронної та асинхронної реплікації також дозволяє легко розподіляти збережені дані між декількома серверами для забезпечення високої відмовостійкості та мінімального часу доступу до критично важливих даних [28].

База PostgreSQL була обрана основною для цієї інформаційної системи, а Redis – як додаткова, а саме як кеш.

Щодо засобів контейнеризації, є дві ключові опції: Docker та Podman. Docker - популярна платформа для створення, розгортання та управління контейнерами. Контейнери Docker дозволяють розробникам застосовувати системно-діагностичний підхід до розгортання програмного забезпечення. Оскільки Docker запускає одні й ті ж докер-контейнери на будь-якій ОС, контейнерні додатки є кросплатформенними.

Podman – бездемоновий, безкореневий контейнерний движок, розроблений компанією RedHat, створений як альтернатива Docker. Модульна конструкція дозволяє Podman використовувати окремі компоненти системи лише за необхідності. Його безкореневий підхід до управління контейнерами дозволяє запускати контейнери користувачам, які не мають прав root [19]. Docker був обраний через більшу популярність на ринку та підтримку спільноти.

2.2 Опис обраних методів

FastAPI – фреймворк Python створений розробником на ім’я Sebastián Ramírez у 2018 році. Основними перевагами цього фреймворку є дуже висока продуктивність, на рівні з NodeJS і Go, він є одним з найшвидших фрейморків мови Python [22]. Також, його перевагами над іншими фреймворками є слідування стандартам OpenAPI та JSON Schema, а також підтримка підказок типів, що робить архітектуру проєктів більш продуманою, тому що розробнику треба думати про типи даних.

Pydantic – бібліотека для валідації даних у запитах та роботи з внутрішніми налаштуваннями додатку. Ця бібліотека гарно працює з FastAPI, тому що також робить акцент на типізації коду. Вона також має високу продуктивність в порівнянні з іншими бібліотеками схожого типу [23].

PostgreSQL – реляційна база даних, з відкритим кодом написана на мові C. Тільки PostgreSQL забезпечує продуктивність і функції корпоративного класу серед існуючих СУБД з відкритим вихідним кодом і безмежними можливостями розвитку.

Redis – це сховище структур даних з відкритим вихідним кодом (ліцензія BSD), яке використовується в якості бази даних, кешу, брокера повідомлень та механізму потокової передачі даних. Redis надає такі структури даних, як рядки, хеші, списки, множини, відсортовані множини з діапазонними запитами, растрові зображення, гіперлоги, геопросторові індекси та потоки. Redis має вбудовані засоби реплікації, Lua-скриптування, LRU-виселення, транзакції та різні рівні персистентності на диску, а також забезпечує високу доступність за допомогою Redis Sentinel та автоматичне розбиття на розділи за допомогою Redis Cluster. Для досягнення максимальної продуктивності Redis працює з набором даних в пам'яті. Залежно від використання, Redis може зберігати ваші дані або шляхом періодичного скидання набору даних на диск, або шляхом додавання кожної команди до журналу на диску. Ви також можете відключити збереження, якщо вам просто потрібен багатофункціональний, мережевий, кеш в пам'яті. [24].

Docker – інструмент для контейнеризації коду. Простіше кажучи, контейнер – це ізольований процес на вашому комп'ютері, який ізольований від усіх інших процесів на хост-машині. Ця ізоляція використовує простори імен ядра і cgroups, функції, які були в Linux протягом тривалого часу. Docker працював над тим, щоб зробити ці можливості доступними і простими у використанні. При запуску контейнера використовується ізольована файлова система. Ця користувацька файлова система забезпечується образом контейнера. Оскільки образ містить файлову систему контейнера, він повинен містити все необхідне для запуску програми - всі залежності, конфігурації, скрипти, двійкові файли тощо. Образ також містить іншу конфігурацію контейнера, таку як змінні оточення, команду запуску за замовчуванням та

інші метадані. Образи н а основі Linux Alpine є дуже легкими в розгортуванні та займають дуже мало пам’яті, наприклад образ Python-alpine займає всього близько 50 мегабайт, а вага redis-alpine – 28 мегабайт. При коректному використанні виборі інструментів можна досягти гарного результату, фінальний образ, включаючий усі залежності буде важити 470 мегебайт, при тому що використання образу slim buster вага буде складати 1.3 гігабайту.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

У попередньому розділі були обрані методи для програмної реалізації, які зроблять розробку більш зручною та ефективною.

Програмна реалізація містить в собі:

− Створення API для реєстрації, входу, відновлення паролю тощо;

− Створення API для створення, отримування, модифікацій та видалення постів, написаних користувачем та іншими користувачами сервісу.

− Перегляд інших користувачів сервісу та деталей про них;

− Реалізацію підтримки асинхронних задач;

− Реалізацію кешування за допомогою Redis;

− Впровадження стандартів коду, таких як mypy, black, isort, тощо.

3.1 Опис архітектури сервісу

Ключовими елементами системи будуть:

- FastAPI сервер
- FastAPI черга
- База даних PostgreSQL
- База даних Redis
- Поштовий сервіс, такий як Gmail

Розробка архітектури – найважливіший етап створення програмного продукту. Від даної роботи залежить те, наскільки легко система зможе масштабуватися у майбутньому та наскільки ефективно вона буде використувати відведені їй ресурси, а також швидкодія системи, що напряму впливає на досвід користувачів. На рисунку 3.1 наведено загальну схему архітектури проєкту “Chop-chop”.

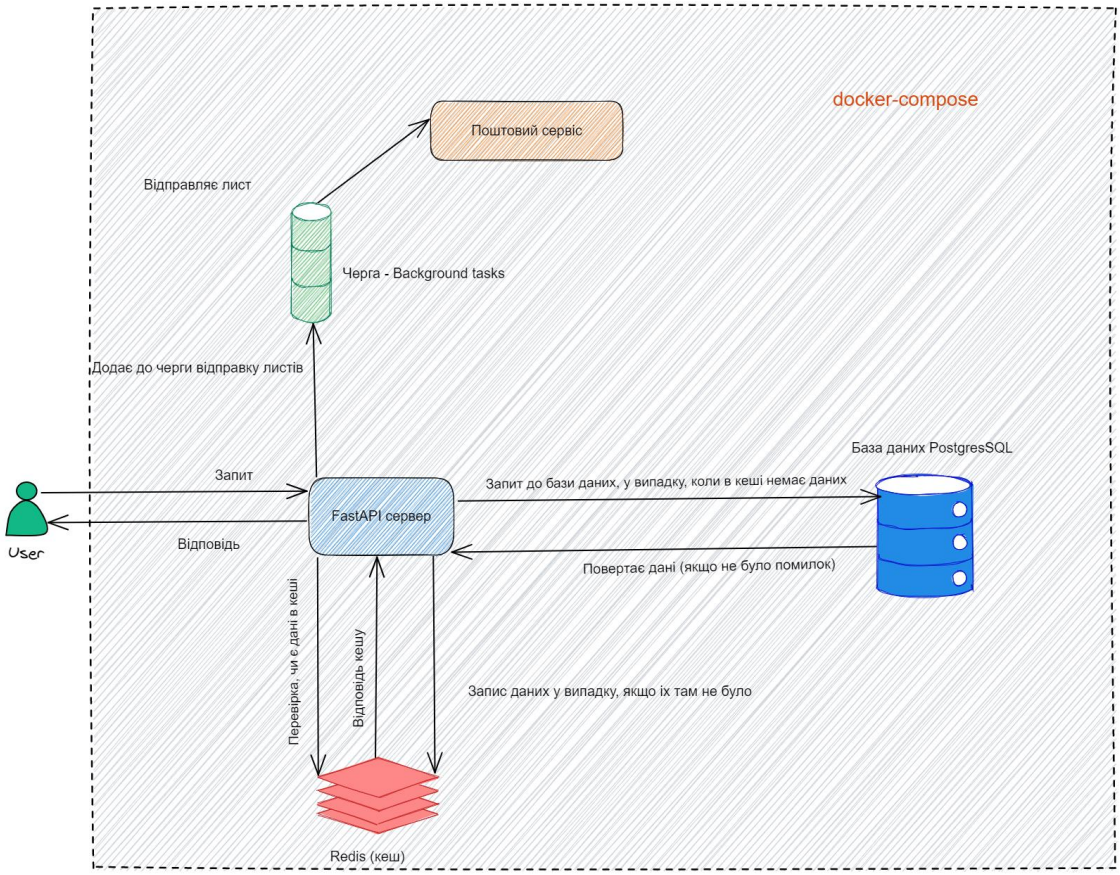

Рисунок 3.1 – Загальна архітектура системи

Розробка зрозумілого API також є важливим кроком для створення стабільної та гнучкої системи, яку можна легко доповнити за необхідності. Розроблений API відповідає вимогам REST та дотримується конвенцій щодо назв шляхів та параметрів. Інструменти, такі як Swagger або Redoc дозволяють бачити документацію, згенеровану на основі написаного коду, а також дані, що потрібні від користувача для виклику тої чи іншої функції. Ще однією особливістю таких систем є візуальне відображення моделей даних, що поверне сервер, що робить інтеграцію з ним дуже простою. На рисунках 3.2 та 3.2 показана візуальна складова, а саме Swagger та Redoc.

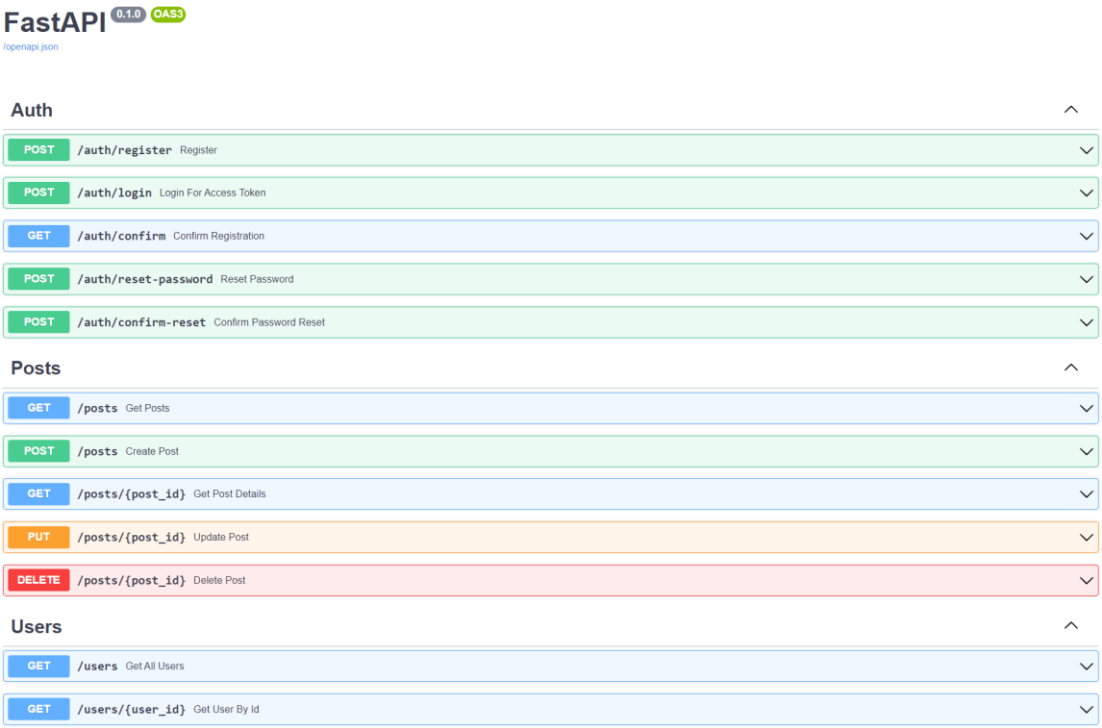

Рисунок 3.2 – Swagger сервісу chop-chop

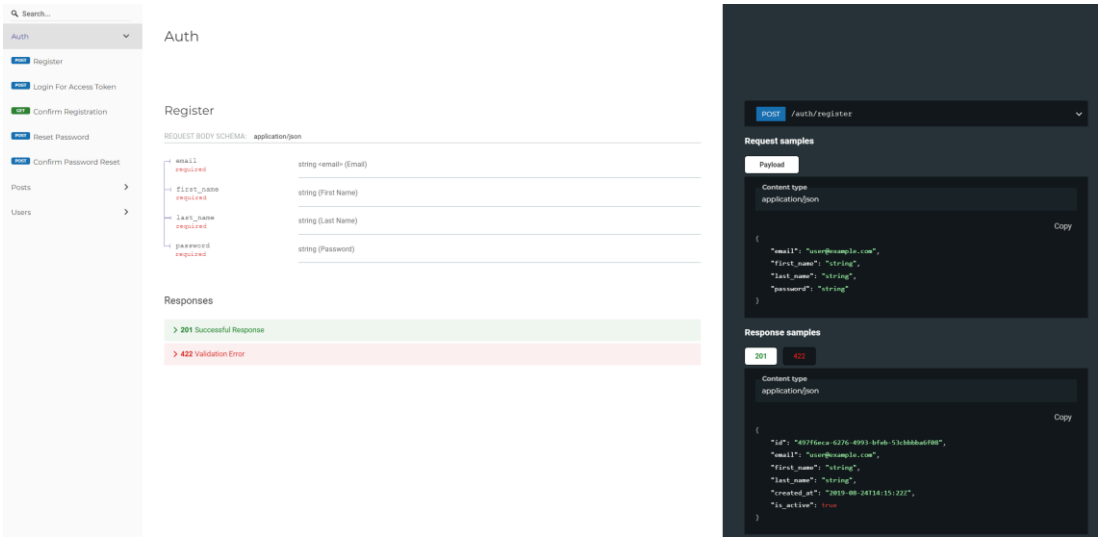

Рисунок 3.3 – Redoc сервісу chop-chop

3.2 Програмна реалізація

Проєкт був структурований за логічними модулями, таким чином усі пакети у ньому мають приблизно однакову структуру, що підвищує читабельність коду (рис 3.4).

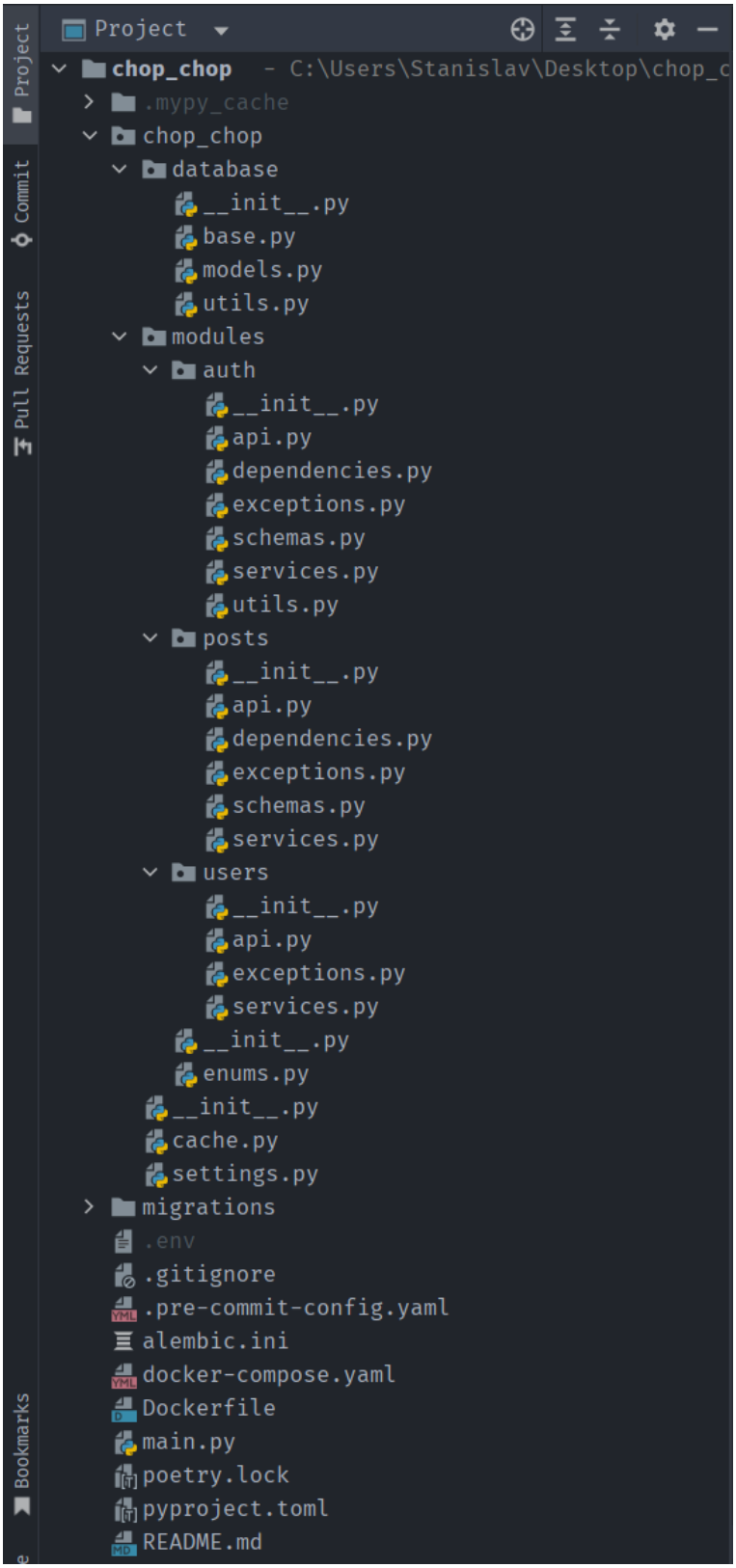

Рисунок 3.4 – Організація коду

Використання кешу виглядає так. Тестування було проведене на 1130 (рис 3.5 та рис 3.6) та 2480 (рис 3.7 та рис 3.8) записах у базі даних, які видавалися за раз.

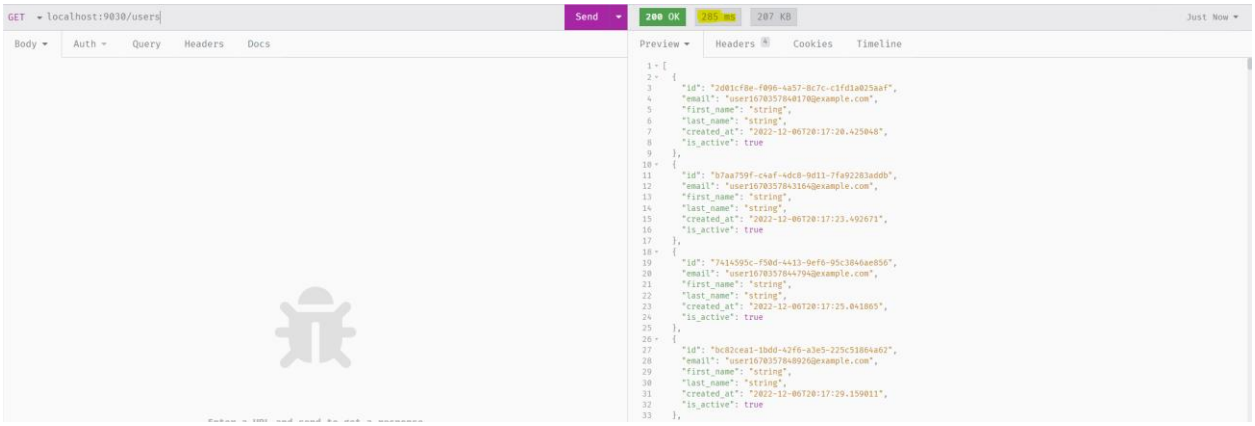

Рисунок 3.5 – Час відповіді без кешування 285 мілісекунд, 1130 записів

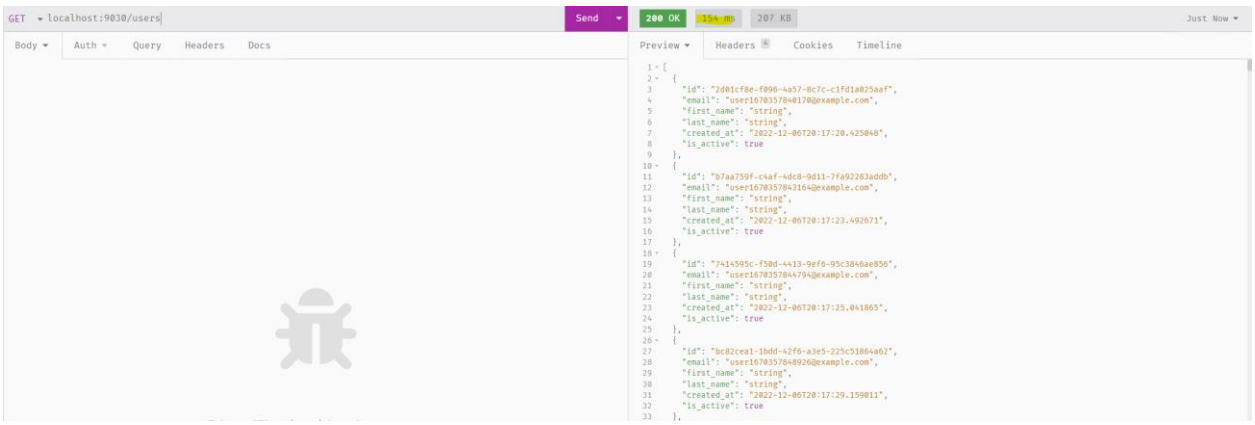

Рисунок 3.6 – Час відповіді з кешу 154 мілісекунд, 1130 записів

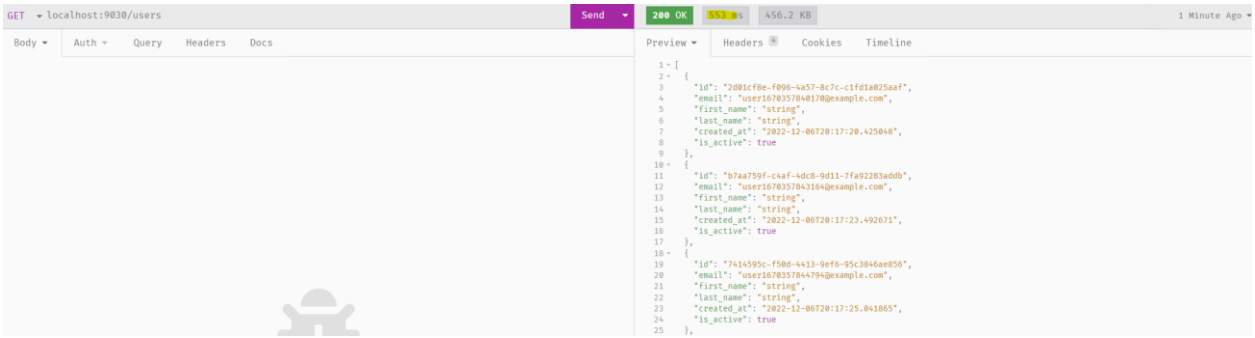

Рисунок 3.7 – Час відповіді без кешування 553 мілісекунд, 2480 записів

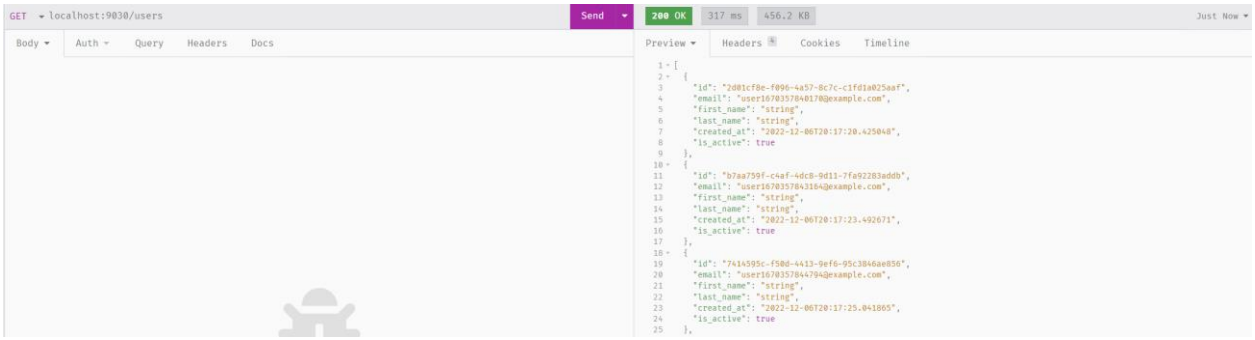

Рисунок 3.8 – Час відповіді з кешу 317 мілісекунд , 2480 записів

З цих показників бачимо, що на малій кількості даних, сервер відпрацьовує швидше, не дивлячись на те, що ніяких змін до коду не робилося. Типовим значенням, для веб запитів є 100 записів на сторінку, таким чином бачимо, що продуктивність може вирости приблизно в 5 разів (рис 3.9 та 3.10).

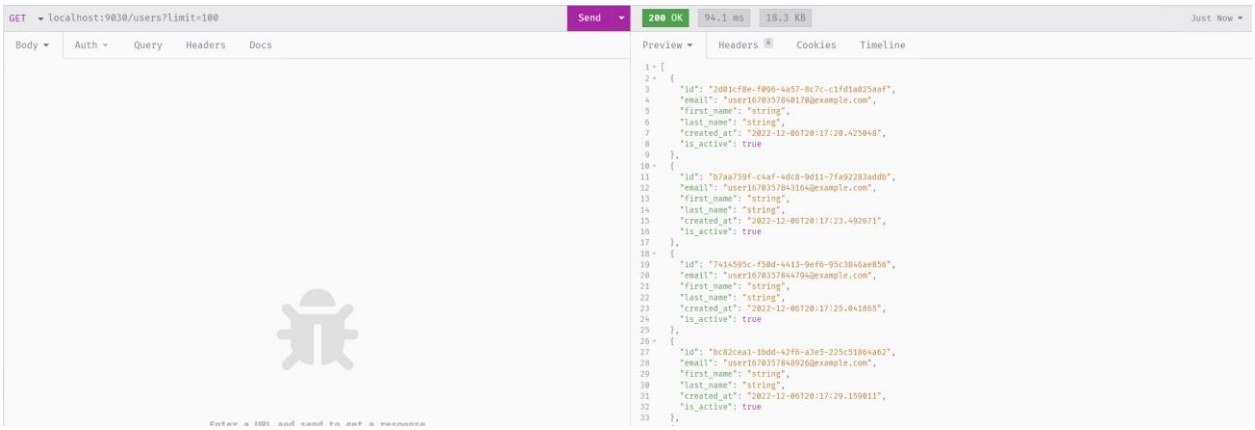

Рисунок 3.9 – Час відповіді без використання кешу – 94 мс

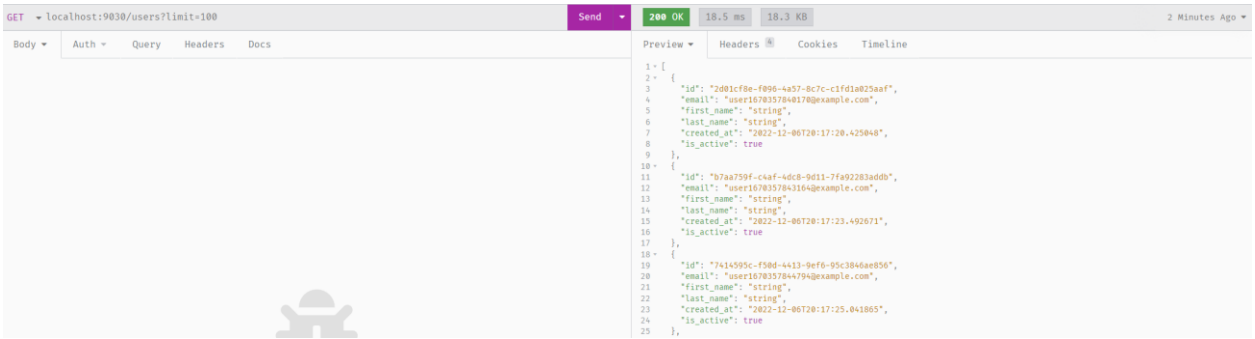

Рисунок 3.10 – Час відповіді без використання кешу – 18 мс

На рисунку 3.11 показане типове використання декоратору, а на рисунках 3.12 та 3.13 можна побачити вигляд даних у базі даних Redis. Риснок 3.14 демонструє результати тестування додатку.

```python
@redis_cache()
async def get_users(pagination: Pagination) -> list[UserDetailsOut]:
    async with get_session() as db_session:
        query = select(User).where(User.is_active).limit(pagination.limit).offset(pagination.offset)
        users = await db_session.execute(query)
        return [UserDetailsOut.from_orm(user) for user in users.scalars().all()]
```

Рисунок 3.11 – Використання декоратору

```
Terminal:  Local  Local (2)  +
127.0.0.1:6379> keys *
1) "chop_chop.modules.users.services.get_users.(Pagination(limit=100, offset=0),).[]"
127.0.0.1:6379>
```

Рисунок 3.12 – Ключ в базі даних Redis, під яким знаходится результат функції

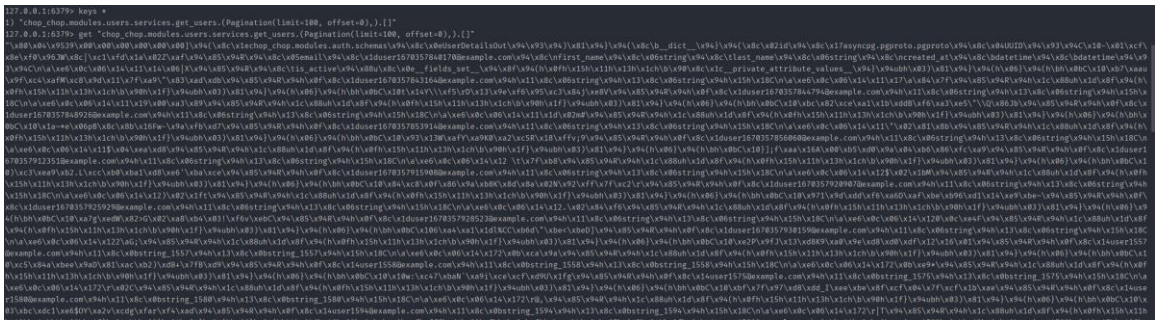

Рисунок 3.13 – Значення в базі даних, закодоване в байти

rows	no cache	cache
30	16	8
50	22	10
100	30	18
300	72	44
500	120	76
700	201	92
1000	256	131
2000	440	256

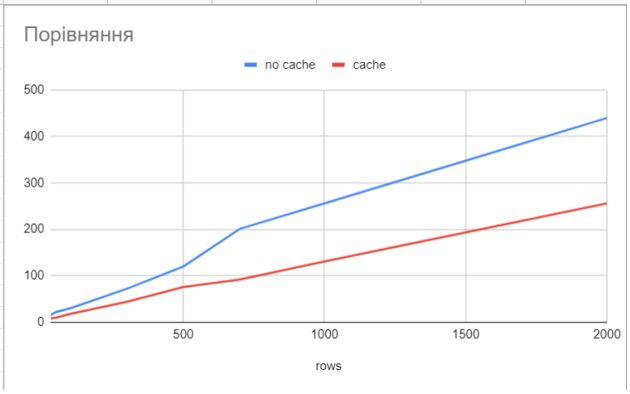

Рисунок 3.14 – Результати тестування додатку

ВИСНОВКИ

В ході роботи була створена інформаційна технологія оптимізації доступності та швидкодії серверних додатків за допомогою мови Python та фреймворку FastAPI. Це дозволило побачити, що використання асинхронного програмування допомагає збільшити кількість користувачів, які одночасно звертаються до серверу, але час відповіді може все ще бути довгим, через брак ресурсів сервера.

Використання Docker та технологій як Kubernetes допомагають масштабувати код легко, описуючи інфраструктуру у форматі yaml файлів. Чим більше ресурсів (контейнерів) є, тим швидше сервер буде відповідати.

Використання кешування, такого як Redis або Memcached ще більше скорочують час відповіді, тому то час не витрачається на дорогі операції, такі як встановлення зв’язку з БД.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ulrike Hack. What’s the real story behind the explosive growth of data? URL: https://opentextbc.ca/dbdesign01/chapter/chapter-7-the-relational-data-model/ (дата звернення: 05.11.2022).
2. Watt A. Chapter 7 The Relational Data Model. URL: https://opentextbc.ca/dbdesign01/chapter/chapter-7-the-relational-data-model/ (дата звернення: 05.11.2022).
3. Seguin K. The little Redis Book. URL: https://www.openmymind.net/redis.pdf (дата звернення: 05.11.2022).
4. Salonen O. What is the Big-O for SQL select? URL: https://stackoverflow.com/questions/1347552/what-is-the-big-o-for-sql-select (дата звернення: 05.11.2022).
5. Internet World Stats. INTERNET GROWTH STATISTICS. URL: https://www.internetworldstats.com/emarketing.htm (дата звернення: 05.11.2022).
6. Claire J. A Minute on the Internet in 2021. URL: https://www.statista.com/chart/25443/estimated-amount-of-data-created-on-the-internet-in-one-minute/ (дата звернення: 05.11.2022).
7. Statista. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025. URL: https://www.statista.com/statistics/871513/worldwide-data-created/ (дата звернення: 05.11.2022).
8. Touchstone. Horizontal vs Vertical Scaling. URL: https://touchstonesecurity.com/horizontal-vs-vertical-scaling-what-you-need-to-know/ (дата звернення: 05.11.2022).
9. Cloudfit public docs. Python Asyncio Part 1 – Basic Concepts and Patterns. URL: https://bbc.github.io/cloudfit-public-docs/asyncio/asyncio-part-1.html (дата звернення: 05.11.2022).

10. Grinberg M. Asynchronous Python for the Complete Beginner PyCon 2017. URL: https://www.youtube.com/watch?v=iG6fr81xHKA&t=269s&ab_channel=PyCon2017 (дата звернення: 05.11.2022).
11. Solomon B. Async IO in Python: A Complete Walkthrough. URL: https://realpython.com/async-io-python/ (дата звернення: 05.11.2022).
12. IBM. Containerization URL: https://www.ibm.com/cloud/learn/containerization (дата звернення: 05.11.2022).
13. Waveworks. Docker vs Virtual Machines (VMs): A Practical Guide to Docker Containers and VMs. URL: https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms (дата звернення: 05.11.2022).
14. Raju K. PostgreSQL – Index Types. URL: https://www.geeksforgeeks.org/postgresql-index-types/ (дата звернення: 05.11.2022).
15. AWS. Caching Overview URL: https://aws.amazon.com/caching/ (дата звернення: 05.11.2022).
16. Redis. Performance baseline. URL: https://redis.io/docs/stack/json/performance/#baseline (дата звернення: 05.11.2022).
17. TechStacks. Redis. URL: https://techstacks.io/tech/redis (дата звернення: 05.11.2022).
18. Aditya Y. Bhargava Grokking Algorithms (дата звернення: 05.11.2022).
19. phoenixNAP. Podman vs Docker. URL https://phoenixnap.com/kb/podman-vs-docker (дата звернення: 18.11.2022)
20. Dev.to. The Best Python Web Frameworks 2022. URL: https://dev.to/theme_selection/the-best-python-web-frameworks-d2d (дата звернення: 18.11.2022)

21. Coursera. What Is Python Used For? A Beginner’s Guide. URL: https://www.coursera.org/articles/what-is-python-used-for-a-beginners-guide-to-using-python (дата звернення: 18.11.2022)
22. FastAPI. FastAPI. URL: https://fastapi.tiangolo.com/ (дата звернення: 18.11.2022)
23. Pydantic. Pydantic overview. URL: https://pydantic-docs.helpmanual.io/#rationale (дата звернення: 18.11.2022)
24. Redis. Introduction to Redis. URL: https://redis.io/docs/about/ (дата звернення: 18.11.2022)
25. Altexsoft. The Good and the Bad of Java Programming. URL: https://www.altexsoft.com/blog/engineering/pros-and-cons-of-java-programming/ (дата звернення: 28.11.2022)
26. Shehroz Azam. What are the advantages and disadvantages of JavaScript. URL: https://linuxhint.com/advantages-disadvantages-javascript/ (дата звернення: 28.11.2022)
27. Varun Bhagat. Pros and Cons of Python Programming Language. URL: https://www.pixelcrayons.com/blog/python-pros-and-cons/ (дата звернення: 28.11.2022)
28. Digital Guide. PostgreSQL: a closer look at the object-relational database management system. URL: https://www.ionos.com/digitalguide/server/know-how/postgresql/ (дата звернення: 28.11.2022)

ДОДАТОК А
ЛІСТИНГ МОДУЛІВ ЕКСПЕРТНОЇ СИСТЕМИ

Файл main.py

```
from fastapi import FastAPI

from chop_chop.modules.auth.api import auth
from chop_chop.modules.posts.api import posts
from chop_chop.modules.users.api import users


def create_app() -> FastAPI:
    application = FastAPI()
    application.include_router(auth)
    application.include_router(posts)
    application.include_router(users)
    return application


app = create_app()
```

Файл settings.py

```
from typing import cast

from fastapi.security import OAuth2PasswordBearer
from fastapi_mail import ConnectionConfig, FastMail
from passlib.context import CryptContext
from pydantic import BaseSettings, PostgresDsn, validator


class Settings(BaseSettings):

    # App settings
    PORT: int = 9030
```

```
    HOST: str = "localhost"

    APP_POSTGRES_HOST: str = "localhost"
    POSTGRES_PORT: int = 5432
    POSTGRES_USER: str = "postgres"
    POSTGRES_PASSWORD: str = "password"
    POSTGRES_DB: str = "chop_chop"

    SQLALCHEMY_URL: PostgresDsn | None = None

    @validator("SQLALCHEMY_URL")
    def create_sqlalchecmy_url(cls, v: PostgresDsn, values:
dict[str, str | int]) -> PostgresDsn:
        host = values.get("APP_POSTGRES_HOST")
        port = values.get("POSTGRES_PORT")
        user = values.get("POSTGRES_USER")
        password = values.get("POSTGRES_PASSWORD")
        db_name = values.get("POSTGRES_DB")
        return cast(PostgresDsn,
f"postgresql+asyncpg://{user}:{password}@{host}:{port}/{db_name}
")

    REDIS_HOST: str = "redis"
    REDIS_PORT: int = 6379

    SECRET_KEY = "DEV"
    ALGORITHM = "HS256"
    ACCESS_TOKEN_EXPIRE_MINUTES = 60
    pwd_context = CryptContext(schemes=["bcrypt"],
deprecated="auto")
    oauth2_scheme = OAuth2PasswordBearer(tokenUrl="auth/login")
    dev_mailcatcher_conf = ConnectionConfig(
        MAIL_USERNAME="",
        MAIL_PASSWORD="",
        MAIL_FROM="chop-chop@dev.com",  # noqa
```

```
        MAIL_PORT=1025,
        MAIL_SERVER="mailcatcher",
        MAIL_FROM_NAME="Chop-Chop",
        MAIL_STARTTLS=False,
        MAIL_SSL_TLS=False,
        USE_CREDENTIALS=False,
        VALIDATE_CERTS=False,
    )
    fm = FastMail(dev_mailcatcher_conf)


settings = Settings()
```

Файл cache.py

```
from __future__ import annotations

import pickle
from functools import wraps
from typing import Any, Callable, TypeVar, cast

from redis.asyncio import Redis

from chop_chop.settings import settings

F = TypeVar("F", bound=Callable[..., Any])  # Function with any
arguments and any return type


class RedisCache:
    _instance: RedisCache | None = None

    def __new__(cls, host: str, port: int, *args: Any, **kwargs:
Any) -> RedisCache:
        if not cls._instance:
```

```
            cls._instance = super().__new__(cls, *args,
**kwargs)
        return cls._instance

    def __init__(self, host: str, port: int) -> None:
        self.host = host
        self.port = port

    def __call__(self, ttl: int = 300) -> Callable[[F], F]:
        def wrapper(function: F) -> F:
            @wraps(function)
            async def _wrapper(*args: Any, **kwargs: Any) ->
Any:
                connection = Redis(host=self.host,
port=self.port)  # type: ignore
                key = self._key_from_args(function, args,
kwargs)
                if result := await connection.get(key):
                    return pickle.loads(result)

                fn_result = await function(*args, **kwargs)
                await connection.setex(key, ttl,
pickle.dumps(fn_result))
                return fn_result

            return cast(F, _wrapper)

        return wrapper

    def _key_from_args(self, function: F, args: tuple[Any, ...],
kwargs: dict[Any, Any]) -> str:
        ordered_kwargs = sorted(kwargs.items())
        return f"{function.__module__ or
''}.{function.__name__}.{str(args)}.{str(ordered_kwargs)}"
```

```
redis_cache = RedisCache(settings.REDIS_HOST,
settings.REDIS_PORT)
```

Файл base.py

```
from sqlalchemy.ext.asyncio import AsyncEngine, AsyncSession,
create_async_engine
from sqlalchemy.orm import declarative_base, sessionmaker

from chop_chop.settings import settings

Base = declarative_base()
engine: AsyncEngine =
create_async_engine(settings.SQLALCHEMY_URL, pool_size=10,
max_overflow=30)
async_session_factory = sessionmaker(engine,
expire_on_commit=False, class_=AsyncSession)
```

Файл models.py

```
from uuid import uuid4

from sqlalchemy import Boolean, Column, DateTime, ForeignKey,
Integer, String, func
from sqlalchemy.dialects.postgresql import UUID
from sqlalchemy.orm import relationship

from chop_chop.database.utils import Base


class User(Base):
    __tablename__ = "users"

    id = Column(UUID(as_uuid=True), primary_key=True,
default=uuid4)
```

```
    email = Column(String(50), unique=True, nullable=False,
index=True)
    first_name = Column(String(50), nullable=False)
    last_name = Column(String(50), nullable=False)
    password = Column(String, nullable=False)
    is_active = Column(Boolean, default=False)
    created_at = Column(DateTime, default=func.now())

    posts = relationship("Post", back_populates="created_by")
    comments = relationship("Comment",
back_populates="created_by")


class Post(Base):
    __tablename__ = "posts"

    id = Column(UUID(as_uuid=True), primary_key=True,
default=uuid4)
    title = Column(String, nullable=False)
    owner_id = Column(UUID(as_uuid=True),
ForeignKey("users.id"), nullable=False)
    body = Column(String, nullable=False)
    complexity = Column(String, nullable=False)
    likes = Column(Integer, nullable=True, default=0)
    created_at = Column(DateTime, default=func.now())

    created_by = relationship("User", back_populates="posts")


class Comment(Base):
    __tablename__ = "comments"

    id = Column(UUID(as_uuid=True), primary_key=True,
default=uuid4)
    body = Column(String, nullable=False)
```

```
    owner_id = Column(UUID(as_uuid=True),
ForeignKey("users.id"))
    created_at = Column(DateTime, default=func.now())

    created_by = relationship("User", back_populates="comments")


class Ingredient(Base):
    __tablename__ = "ingredients"
    id = Column(Integer, primary_key=True)
    name = Column(String(100), nullable=False, unique=True,
index=True)


class IngredientToPost(Base):
    __tablename__ = "ingredients_to_posts"

    post_id = Column(UUID(as_uuid=True), ForeignKey("posts.id"),
primary_key=True)
    ingredient_id = Column(Integer,
ForeignKey("ingredients.id"), primary_key=True)
```

Файл utils.py

```
from contextlib import asynccontextmanager
from typing import AsyncIterator

from sqlalchemy.exc import DBAPIError
from sqlalchemy.ext.asyncio import AsyncSession

from chop_chop.database.base import Base, async_session_factory,
engine


@asynccontextmanager
async def get_session() -> AsyncIterator[AsyncSession]:
```

```
    async with async_session_factory() as session:
        try:
            yield session
        except DBAPIError:
            await session.rollback()
        finally:
            await session.close()


async def init_models() -> None:
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.drop_all)
        await conn.run_sync(Base.metadata.create_all)
```

Файл enums.py

```
from enum import Enum


class RecipeComplexity(str, Enum):
    easy = "easy"
    medium = "medium"
    hard = "hard"
```

Файл auth/api.py

```
from fastapi import APIRouter, BackgroundTasks, Depends, status
from pydantic import EmailStr

from chop_chop.modules.auth import services
from chop_chop.modules.auth.dependencies import get_login_data
from chop_chop.modules.auth.schemas import (
    Token,
    UserDetailsOut,
    UserLoginSchema,
    UserRegisterSchema,
```

```
    UserResetPassword,
)

auth = APIRouter(prefix="/auth", tags=["Auth"])


@auth.post("/register", status_code=status.HTTP_201_CREATED,
response_model=UserDetailsOut)
async def register(
    background_tasks: BackgroundTasks,
    user_data: UserRegisterSchema,
) -> UserDetailsOut:
    return await services.register_user(user_data,
background_tasks)


@auth.post("/login", status_code=status.HTTP_200_OK,
response_model=Token)
async def login_for_access_token(user_login_data:
UserLoginSchema = Depends(get_login_data)) -> Token:
    return await services.login(user_login_data)


@auth.get("/confirm", status_code=status.HTTP_200_OK,
response_model=UserDetailsOut)
async def confirm_registration(token: str) -> UserDetailsOut:
    return await services.confirm_user_registration(token)


@auth.post("/reset-password", status_code=status.HTTP_200_OK)
async def reset_password(email: EmailStr, background_tasks:
BackgroundTasks) -> dict[str, str]:
    return await services.send_reset_link(background_tasks,
email)
```

```
@auth.post("/confirm-reset", status_code=status.HTTP_200_OK)
async def confirm_password_reset(password: UserResetPassword,
token: str) -> UserDetailsOut:
    return await services.confirm_password_reset(token,
password)
```

Файл auth/dependencies.py

```
from fastapi import Depends
from fastapi.security import OAuth2PasswordRequestForm

from chop_chop.modules.auth.schemas import UserLoginSchema


def get_login_data(form_data: OAuth2PasswordRequestForm =
Depends()) -> UserLoginSchema:
    return UserLoginSchema(email=form_data.username,
password=form_data.password)
```

Файл auth/exceptions.py

```
from fastapi import HTTPException, status


class UserAlreadyExists(HTTPException):
    def __init__(self):
        super().__init__(
            status_code=status.HTTP_400_BAD_REQUEST,
detail="User with this email already exists."
        )


class IncorrectCredentials(HTTPException):
    def __init__(self):
```

```
super().__init__(status_code=status.HTTP_401_UNAUTHORIZED,
detail="Incorrect username or password.")


class InactiveAccount(HTTPException):
    def __init__(self):

super().__init__(status_code=status.HTTP_401_UNAUTHORIZED,
detail="Email is not confirmed.")


class JWTCredentialsException(HTTPException):
    def __init__(self):
        super().__init__(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Could not validate credentials",
            headers={"WWW-Authenticate": "Bearer"},
        )


class CannotConfirmRegistration(HTTPException):
    def __init__(self):
        super().__init__(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Registration confirmation failed, invalid
token provided.",
        )


class CannotResetPassword(HTTPException):
    def __init__(self):
        super().__init__(
            status_code=status.HTTP_400_BAD_REQUEST,
```

```
            detail="Password reset failed, invalid token
provided.",
        )


class CannotDecodeJWT(Exception):
    def __init__(self):
        super().__init__("Cannot decode given JWT token.")
```

Файл auth/schemas.py

```
from datetime import datetime
from uuid import UUID

from pydantic import BaseModel, EmailStr, validator


class UserRegisterSchema(BaseModel):
    email: EmailStr
    first_name: str
    last_name: str
    password: str

    class Config:
        orm_mode = True


class UserDetailsOut(BaseModel):
    id: UUID
    email: EmailStr
    first_name: str
    last_name: str
    created_at: datetime
    is_active: bool

    class Config:
```

```
        orm_mode = True


class UserLoginSchema(BaseModel):
    email: str
    password: str


class Token(BaseModel):
    token_type: str
    expires_in: int
    access_token: str


class UserResetPassword(BaseModel):
    password: str
    confirm_password: str

    @validator("confirm_password")
    def passwords_match(cls, v: str, values: dict[str, str]) ->
str:
        password = values.get("password")
        if password != v:
            raise ValueError("Passwords do not match.")
        return v
```

Файл auth/services.py

```
from datetime import timedelta

from fastapi import BackgroundTasks
from passlib.hash import bcrypt
from sqlalchemy import select
from sqlalchemy.exc import IntegrityError

from chop_chop.database.models import User
```

```
from chop_chop.database.utils import get_session
from chop_chop.modules.auth import exceptions as exc
from chop_chop.modules.auth.schemas import (
    Token,
    UserDetailsOut,
    UserLoginSchema,
    UserRegisterSchema,
    UserResetPassword,
)
from chop_chop.modules.auth.utils import (
    create_access_token,
    get_email_from_jwt_token,
    send_confirmation_email,
    send_reset_password_email,
)
from chop_chop.settings import settings


async def register_user(user: UserRegisterSchema,
background_tasks: BackgroundTasks) -> UserDetailsOut:
    async with get_session() as db_session:
        hashed_password = bcrypt.hash(user.password)
        user_model = User(
            email=user.email,
            first_name=user.first_name,
            last_name=user.last_name,
            password=hashed_password,
        )
        db_session.add(user_model)
        try:
            await db_session.commit()
            await db_session.refresh(user_model)
            send_confirmation_email(background_tasks, user)
            return UserDetailsOut.from_orm(user_model)
        except IntegrityError:
```

```
            raise exc.UserAlreadyExists


async def login(user: UserLoginSchema) -> Token:
    async with get_session() as db_session:
        query = select(User).where(User.email == user.email)
        user_from_db = (await
db_session.execute(query)).scalars().first()
        if not user_from_db or not bcrypt.verify(user.password,
user_from_db.password):
            raise exc.IncorrectCredentials
        if not user_from_db.is_active:
            raise exc.InactiveAccount
    access_token_expires =
timedelta(minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(payload={"user":
user.email}, expires_delta=access_token_expires)
    return Token(access_token=access_token, token_type="Bearer",
expires_in=access_token_expires.seconds)


async def confirm_user_registration(token: str) ->
UserDetailsOut:
    try:
        email = get_email_from_jwt_token(token)
    except exc.CannotDecodeJWT:
        raise exc.CannotConfirmRegistration
    async with get_session() as db_session:
        query = select(User).where(User.email == email)
        db_user = (await
db_session.execute(query)).scalars().first()
        if not db_user:
            raise exc.CannotConfirmRegistration

        db_user.is_active = True
```

```
        await db_session.commit()
        await db_session.refresh(db_user)
        return UserDetailsOut.from_orm(db_user)


async def send_reset_link(background_tasks: BackgroundTasks,
email: str) -> dict[str, str]:
    async with get_session() as db_session:
        query = select(User).filter_by(email=email)
        query_result = await db_session.execute(query)
        if user_from_db := query_result.scalars().first():
            send_reset_password_email(background_tasks,
UserDetailsOut.from_orm(user_from_db))
    return {"message": "Recovery link sent, please, check your
email."}


async def confirm_password_reset(token: str, password:
UserResetPassword) -> UserDetailsOut:
    try:
        email_from_jwt = get_email_from_jwt_token(token)
    except exc.CannotDecodeJWT:
        raise exc.CannotResetPassword
    async with get_session() as db_session:
        query = select(User).where(User.email == email_from_jwt)
        query_result = await db_session.execute(query)
        user_from_db = query_result.scalars().first()
        if not user_from_db:
            raise exc.CannotResetPassword
        user_from_db.password = bcrypt.hash(password.password)
        await db_session.commit()
        await db_session.refresh(user_from_db)
    return UserDetailsOut.from_orm(user_from_db)
```

Файл auth/utils.py

```
from datetime import datetime, timedelta

import jwt
from fastapi import BackgroundTasks, Header
from fastapi_mail import MessageSchema, MessageType
from jose import JWTError
from sqlalchemy import select

from chop_chop.cache import redis_cache
from chop_chop.database import models, utils
from chop_chop.modules.auth import exceptions as exc
from chop_chop.modules.auth.schemas import UserDetailsOut,
UserRegisterSchema
from chop_chop.settings import settings


def create_access_token(payload: dict[str, str], expires_delta:
timedelta | None = None) -> str:
    """Encodes given user data into JWT token"""
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)
    payload.update({"expires_in": str(expire)})
    encoded_jwt = jwt.encode(payload, settings.SECRET_KEY,
algorithm=settings.ALGORITHM)
    return encoded_jwt


@redis_cache(settings.ACCESS_TOKEN_EXPIRE_MINUTES)
async def get_current_user(token: str = Header(...,
alias="authorization")) -> UserDetailsOut:
    """Based on a given JWT token get the current user."""
    if token.startswith("Bearer "):
```

```
        token = token.split()[1]
    try:
        payload = jwt.decode(token, settings.SECRET_KEY,
algorithms=[settings.ALGORITHM])
        email, expires_in = payload.get("user"),
str(payload.get("expires_in"))
        if not email or datetime.fromisoformat(expires_in) <
datetime.utcnow():
            raise exc.JWTCredentialsException
    except (JWTError, jwt.exceptions.InvalidSignatureError):
        raise exc.JWTCredentialsException
    query = select(models.User).filter_by(email=email)
    async with utils.get_session() as db_session:
        query_result = await db_session.execute(query)
    user = query_result.scalars().first()
    if not user:
        raise exc.JWTCredentialsException
    return UserDetailsOut.from_orm(user)


def send_confirmation_email(background_tasks: BackgroundTasks,
user: UserRegisterSchema) -> None:
    """Sends an email in background with the token to finish
user's registration."""
    payload = {"email": user.email, "first_name":
user.first_name}
    token = jwt.encode(payload, settings.SECRET_KEY,
algorithm=settings.ALGORITHM)

    message = MessageSchema(
        subject="Finish registration",
        recipients=[user.email],
        body=f"Hello, you've created an account in Chop-Chop,
click the link to confirm! \n"
```

```
f"http://{settings.HOST}:{settings.PORT}/auth/confirm?token={tok
en}",
        subtype=MessageType.html,
    )
    background_tasks.add_task(settings.fm.send_message, message)


def send_reset_password_email(background_tasks: BackgroundTasks,
user: UserDetailsOut) -> None:
    """Sends an email with generated token to reset a password
for a user."""
    payload = {"email": user.email, "first_name":
user.first_name}
    token = jwt.encode(payload, settings.SECRET_KEY,
algorithm=settings.ALGORITHM)

    message = MessageSchema(
        subject="Reset your password",
        recipients=[user.email],
        body=f"Hello, you've asked to reset your password, click
the link to confirm! \n"
        f"http://{settings.HOST}:{settings.PORT}/auth/confirm-
reset?token={token}",
        subtype=MessageType.html,
    )
    background_tasks.add_task(settings.fm.send_message, message)


def get_email_from_jwt_token(token: str) -> str:
    try:
        payload = jwt.decode(token, settings.SECRET_KEY,
algorithms=[settings.ALGORITHM])
        return payload.get("email")  # type: ignore
```

```
    except (jwt.exceptions.InvalidSignatureError,
jwt.exceptions.DecodeError):
        raise exc.CannotDecodeJWT

from pydantic import BaseModel, conint
```

Файл common/schemas.py

```
class Pagination(BaseModel):
    limit: conint(ge=0)
    offset: conint(ge=0)
```

Файл posts/api.py

```
from uuid import UUID

from fastapi import APIRouter, Depends, status

from chop_chop.modules.auth.schemas import UserDetailsOut
from chop_chop.modules.auth.utils import get_current_user
from chop_chop.modules.posts import services
from chop_chop.modules.posts.schemas import (
    CreatePostSchema,
    PostSchema,
    UpdatePostSchema,
)

posts = APIRouter(prefix="/posts", tags=["Posts"])


@posts.get("", status_code=status.HTTP_200_OK,
response_model=list[PostSchema])
async def get_posts() -> list[PostSchema]:
    return await services.get_posts()
```

```
@posts.get("/{post_id}", status_code=status.HTTP_200_OK,
response_model=PostSchema)
async def get_post_details(post_id: UUID) -> PostSchema:
    return await services.get_single_post(post_id)


@posts.post("", status_code=status.HTTP_201_CREATED,
response_model=PostSchema)
async def create_post(
    post: CreatePostSchema,
    current_user: UserDetailsOut = Depends(get_current_user),
) -> PostSchema:
    return await services.create_post(post, current_user.id)


@posts.put("/{post_id}", status_code=status.HTTP_200_OK,
response_model=PostSchema)
async def update_post(
    post_id: UUID,
    post: UpdatePostSchema,
    current_user: UserDetailsOut = Depends(get_current_user),
) -> PostSchema:
    return await services.update_post(post, post_id,
current_user)


@posts.delete("/{post_id}", status_code=status.HTTP_200_OK,
response_model=PostSchema)
async def delete_post(
    post_id: UUID,
    current_user: UserDetailsOut = Depends(get_current_user),
) -> PostSchema:
    return await services.delete_post(post_id, current_user)
```

Файл posts/exceptions.py

```
from fastapi import HTTPException, status


class PostDoesNotExist(HTTPException):
    def __init__(self):
        super().__init__(status_code=status.HTTP_404_NOT_FOUND)


class NotAuthorizedToEdit(HTTPException):
    def __init__(self):

super().__init__(status_code=status.HTTP_401_UNAUTHORIZED,
detail="Not authorized to edit this post")
```

Файл posts/schemas.py

```
from datetime import datetime
from uuid import UUID

from pydantic import BaseModel

from chop_chop.modules.enums import RecipeComplexity


class CreatePostSchema(BaseModel):
    title: str
    body: str
    complexity: RecipeComplexity


class UpdatePostSchema(CreatePostSchema):
    ...
```

```
class PostSchema(CreatePostSchema):
    class Config:
        orm_mode = True

    id: UUID
    owner_id: UUID
    likes: int | None
    created_at: datetime
```

Файл posts/services.py

```
from uuid import UUID

from sqlalchemy import select

from chop_chop.cache import redis_cache
from chop_chop.database.models import Post
from chop_chop.database.utils import get_session
from chop_chop.modules.auth.schemas import UserDetailsOut
from chop_chop.modules.posts import exceptions as exc
from chop_chop.modules.posts.schemas import (
    CreatePostSchema,
    PostSchema,
    UpdatePostSchema,
)


@redis_cache()
async def get_posts() -> list[PostSchema]:
    async with get_session() as db_session:
        posts = await db_session.execute(select(Post))  # TODO:
add filters -> .where({filters})
    return [PostSchema.from_orm(post) for post in
posts.scalars().all()]
```

```
@redis_cache()
async def get_single_post(post_id: UUID) -> PostSchema:
    async with get_session() as db_session:
        post = await db_session.get(Post, post_id)
        if not post:
            raise exc.PostDoesNotExist
    return PostSchema.from_orm(post)


async def create_post(post: CreatePostSchema, created_by: UUID)
-> PostSchema:
    db_post = Post(title=post.title, body=post.body,
complexity=post.complexity, owner_id=created_by)
    async with get_session() as db_session:
        db_session.add(db_post)
        await db_session.commit()
        await db_session.refresh(db_post)
    return PostSchema.from_orm(db_post)


async def update_post(
    updated_post: UpdatePostSchema, post_id: UUID, current_user:
UserDetailsOut
) -> PostSchema:
    query = select(Post).where(Post.id == post_id)
    async with get_session() as db_session:
        db_post: Post = (await
db_session.execute(query)).scalars().first()
        if not db_post:
            raise exc.PostDoesNotExist
        if db_post.owner_id != current_user.id:
            raise exc.NotAuthorizedToEdit
        db_post.title = updated_post.title
        db_post.body = updated_post.body
        db_post.complexity = updated_post.complexity
```

```
        await db_session.commit()
        await db_session.refresh(db_post)
    return PostSchema.from_orm(db_post)


async def delete_post(post_id: UUID, current_user:
UserDetailsOut) -> PostSchema:
    query = select(Post).where((Post.id == post_id) &
(Post.owner_id == current_user.id))
    async with get_session() as db_session:
        db_post: Post = (await
db_session.execute(query)).scalars().first()
        if not db_post:
            raise exc.PostDoesNotExist
        await db_session.delete(db_post)
        await db_session.commit()
    return PostSchema.from_orm(db_post)
```

Файл users/api.py

```
from uuid import UUID

from fastapi import APIRouter, Depends, status

from chop_chop.modules.auth.schemas import UserDetailsOut
from chop_chop.modules.common.schemas import Pagination
from chop_chop.modules.users import services
from chop_chop.modules.users.dependencies import get_pagination

users = APIRouter(prefix="/users", tags=["Users"])


@users.get("", status_code=status.HTTP_200_OK,
response_model=list[UserDetailsOut])
async def get_all_users(pagination: Pagination =
Depends(get_pagination)) -> list[UserDetailsOut]:
```

```
    return await services.get_users(pagination)


@users.get("/{user_id}", status_code=status.HTTP_200_OK,
response_model=UserDetailsOut)
async def get_user_by_id(user_id: UUID) -> UserDetailsOut:
    return await services.get_user_by_id(user_id)
```

Файл users/dependencies.py

```
from fastapi import HTTPException, status
from pydantic import ValidationError

from chop_chop.modules.common.schemas import Pagination


def get_pagination(limit: int = 1000, offset: int = 0) ->
Pagination:
    try:
        return Pagination(limit=limit, offset=offset)
    except ValidationError as exc:
        raise
HTTPException(status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
detail=exc.errors())
```

Файл users/exceptions.py

```
from uuid import UUID

from fastapi import HTTPException, status


class UserNotFound(HTTPException):
    def __init__(self, user_id: UUID):
        super().__init__(status_code=status.HTTP_404_NOT_FOUND,
detail=f"No user with id {user_id}")
```

Файл users/services.py

```
from uuid import UUID

from sqlalchemy import select

from chop_chop.cache import redis_cache
from chop_chop.database.models import User
from chop_chop.database.utils import get_session
from chop_chop.modules.auth.schemas import UserDetailsOut
from chop_chop.modules.common.schemas import Pagination
from chop_chop.modules.users import exceptions as exc


@redis_cache()
async def get_users(pagination: Pagination) ->
list[UserDetailsOut]:
    async with get_session() as db_session:
        query =
select(User).where(User.is_active).limit(pagination.limit).offse
t(pagination.offset)
        users = await db_session.execute(query)
        return [UserDetailsOut.from_orm(user) for user in
users.scalars().all()]


@redis_cache()
async def get_user_by_id(user_id: UUID) -> UserDetailsOut:
    async with get_session() as db_session:
        post = await db_session.get(User, user_id)
        if not post:
            raise exc.UserNotFound(user_id)
    return UserDetailsOut.from_orm(post)
```

Файл migrations/versions/693936b50ce0_initial_migration.py

```
"""Initial migration

Revision ID: 693936b50ce0
Revises:
Create Date: 2022-11-12 15:14:03.555669

"""
import sqlalchemy as sa
from alembic import op
from sqlalchemy.dialects import postgresql

# revision identifiers, used by Alembic.
revision = "693936b50ce0"
down_revision = None
branch_labels = None
depends_on = None


def upgrade() -> None:
    op.create_table(
        "ingredients",
        sa.Column("id", sa.Integer(), nullable=False),
        sa.Column("name", sa.String(length=100),
nullable=False),
        sa.PrimaryKeyConstraint("id"),
    )
    op.create_index(op.f("ix_ingredients_name"), "ingredients",
["name"], unique=True)
    op.create_table(
        "users",
        sa.Column("id", postgresql.UUID(), nullable=False),
        sa.Column("email", sa.String(length=50),
nullable=False),
```

```
        sa.Column("first_name", sa.String(length=50),
nullable=False),
        sa.Column("last_name", sa.String(length=50),
nullable=False),
        sa.Column("password", sa.String(), nullable=False),
        sa.Column("is_active", sa.Boolean(), nullable=True),
        sa.Column("created_at", sa.DateTime(), nullable=True),
        sa.PrimaryKeyConstraint("id"),
    )
    op.create_index(op.f("ix_users_email"), "users", ["email"],
unique=True)
    op.create_table(
        "comments",
        sa.Column("id", postgresql.UUID(), nullable=False),
        sa.Column("body", sa.String(), nullable=False),
        sa.Column("owner_id", postgresql.UUID(), nullable=True),
        sa.Column("created_at", sa.DateTime(), nullable=True),
        sa.ForeignKeyConstraint(
            ("owner_id",),
            ["users.id"],
        ),
        sa.PrimaryKeyConstraint("id"),
    )
    op.create_table(
        "posts",
        sa.Column("id", postgresql.UUID(), nullable=False),
        sa.Column("title", sa.String(), nullable=False),
        sa.Column("owner_id", postgresql.UUID(),
nullable=False),
        sa.Column("body", sa.String(), nullable=False),
        sa.Column("complexity", sa.String(), nullable=False),
        sa.Column("likes", sa.Integer(), nullable=True),
        sa.Column("created_at", sa.DateTime(), nullable=True),
        sa.ForeignKeyConstraint(
            ("owner_id",),
```

```
            ["users.id"],
        ),
        sa.PrimaryKeyConstraint("id"),
    )
    op.create_table(
        "ingredients_to_posts",
        sa.Column("post_id", postgresql.UUID(), nullable=False),
        sa.Column("ingredient_id", sa.Integer(),
nullable=False),
        sa.ForeignKeyConstraint(
            ("ingredient_id",),
            ["ingredients.id"],
        ),
        sa.ForeignKeyConstraint(
            ("post_id",),
            ["posts.id"],
        ),
        sa.PrimaryKeyConstraint("post_id", "ingredient_id"),
    )


def downgrade() -> None:
    op.drop_table("ingredients_to_posts")
    op.drop_table("posts")
    op.drop_table("comments")
    op.drop_index(op.f("ix_users_email"), table_name="users")
    op.drop_table("users")
    op.drop_index(op.f("ix_ingredients_name"),
table_name="ingredients")
    op.drop_table("ingredients")
```

Файл Dockerfile

```
FROM python:3.10-alpine

ENV PYTHONUNBUFFERED=1
ENV APP_NAME=chop-chop
ENV APP_HOME /app
ENV PORT=9030
ENV POETRY_HOME=/opt/poetry
ENV PATH=/opt/poetry/bin:$PATH

WORKDIR $APP_HOME

# Utils setup
RUN apk update
RUN apk --no-cache add curl

# Poetry setup
COPY pyproject.toml poetry.lock ./
RUN (curl -sSL https://install.python-poetry.org | python3 -)
RUN poetry config virtualenvs.create false --local


COPY . ./
RUN poetry install

EXPOSE $PORT
CMD uvicorn main:app --host 0.0.0.0 --port 9030
```

Файл docker-compose.yaml

```
services:
  chop-chop:
    container_name: chop-chop
    build:
      context: .
```

```
    restart: always
    ports:
      - 9030:9030
    networks:
      - chop-chop-net
    depends_on:
      - postgres
      - redis
      - mailcatcher
    volumes:
      - .:/app
    env_file:
      - .env
    entrypoint: >
      sh -c "alembic upgrade head && uvicorn main:app --host
0.0.0.0 --port 9030 --reload"

  postgres:
    container_name: postgres
    restart: always
    image: postgres:alpine
    ports:
      - 5432:5432
    expose:
      - 5432
    networks:
      - chop-chop-net
    env_file:
      - .env
    volumes:
      - pgdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready", "-d", "chop_chop"]
      interval: 30s
      timeout: 3s
```

```
      retries: 3
      start_period: 5s

  redis:
    container_name: redis
    restart: always
    image: redis:alpine
    ports:
      - 6379:6379
    networks:
      - chop-chop-net

  mailcatcher:
    container_name: mailcatcher
    restart: always
    image: jeanberu/mailcatcher
    ports:
      - 1080:1080
      - 1025:1025
    expose:
      - 1080
      - 1025
    networks:
      - chop-chop-net


networks:
  chop-chop-net:
    name: chop-chop-network

volumes:
  pgdata:
```