

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

«Дальневосточный федеральный университет»

(ДВФУ)

**Институт математики и компьютерных технологий**

**Департамент информационных и компьютерных систем**

РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ

Макунев Евгений Александрович

КУРСОВАЯ РАБОТА НА ТЕМУ

|  |
| --- |
| **«Разработка распределенной системы управления нагрузкой**  **вычислительной системы с помощью ограничения запросов»** |

|  |
| --- |
| Студент гр. Б8120-09.03.03-пиуп  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е. А. Макунев |
| Проверил ст. преп.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е. А. Елсукова |
| Регистрационный № \_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  подпись И. О. Фамилия  «\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20 г. | Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_ Е. А. Елсукова  подпись И. О. Фамилия  «\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20 г. |
|  |  |

г. Владивосток

2023

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ ПО ДИСЦИПЛИНЕ

«РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ»

Студент: Макуневу Евгению Александровичу

Группа: Б8120 - 09.03.03 - пиуп

Тема: Разработка распределенной системы управления нагрузкой с помощью ограничения запросов

Техническое задание:

1. Выполнить анализ предметной области;
2. Определить основных участников РС и их функции;
3. Изучить типы архитектур распределенных систем, обосновать выбранный тип и используемые средства разработки;
4. Выполнить проектирование системы и хранилища данных. В проект системы должно входить несколько принципиально различающихся по архитектуре, бизнес-логики, работе с различными видами взаимодействия независимых подсистем.
5. Описать взаимодействие разрабатываемых подсистем;
6. Разработать прототип системы.

Дата выдачи задания «27» марта 2023 г.

Дата сдачи проекта «19» июня 2023 г.

Дата защиты «26» июня 2023 г.

Руководитель: ст. преп.: \_\_\_\_\_\_\_\_\_\_\_\_\_ Е. А. Елсукова

(подпись)

Задание получил: \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (И. О. Фамилия)

Содержание

[Ключевые слова 4](#_Toc138630360)

[1. Постановка задачи 5](#_Toc138630361)

[1.1. Описание проблемы 5](#_Toc138630362)

[1.2. Концепция системы 6](#_Toc138630363)

[1.3. Основные участники распределенной системы 7](#_Toc138630364)

[2. Проектирование системы 9](#_Toc138630365)

[2.1. Описание системы 9](#_Toc138630366)

[2.2. Фронтенд 10](#_Toc138630367)

[2.3. Бэкенд 11](#_Toc138630368)

[2.3.1. API пути сервиса 11](#_Toc138630369)

[2.3.2. Алгоритм валидации 12](#_Toc138630370)

[2.4. База данных 13](#_Toc138630371)

[2.4.1. Правило (rule) 14](#_Toc138630372)

[2.4.2. Контейнер (bucket) 15](#_Toc138630373)

[2.4.3. Контейнер для аналитики 16](#_Toc138630374)

[2.5. Docker 17](#_Toc138630375)

[3. Реализация 18](#_Toc138630376)

[3.1. Docker 19](#_Toc138630377)

[3.2. Фронтенд 20](#_Toc138630378)

[3.3. Бекенд 23](#_Toc138630379)

[Заключение 29](#_Toc138630380)

[Используемые ресурсы 30](#_Toc138630381)

# Ключевые слова

Распределенная система

Ограничение запросов

Алгоритм token bucket

API

Клиент сервисная архитектура

Python

FastAPI

OpenAPI

SwaggerUI

Redis

Docker

# 1. Постановка задачи

## 1.1. Описание проблемы

С ростом популярности web-приложения возникает необходимость обрабатывать и обслуживать большое количество запросов от пользователей одновременно. Увеличивается нагрузка и это приводит к снижению производительности, задержкам при обработке запросов и даже отказу в обслуживании, что негативно влияет на пользовательский опыт.

Для решения проблемы высокой нагрузки веб-приложений можно применить следующие методы:

1. Масштабирование инфраструктуры: Увеличение ресурсов серверов или добавление дополнительных серверов с балансировкой их нагрузки для равномерного распределения запросов и предотвращения перегрузки отдельных серверов.
2. Оптимизация кода: Анализ и оптимизация кода приложения с целью улучшения его производительности и эффективности. Это может включать сокращение запросов к базе данных, улучшение алгоритмов обработки данных, использование кэширования результатов и минимизацию объема передаваемых данных.
3. Оптимизация баз данных: Оптимизация структуры баз данных, индексирование таблиц, настройка параметров запросов и использование кэширования для улучшения производительности базы данных и снижения нагрузки.
4. Кэширование: Использование кэширования для хранения часто запрашиваемых данных или предварительного вычисления результатов операций. Это может существенно снизить нагрузку на сервер и ускорить отклик приложения.

Все эти методы хорошо подходят для решения задачи, но как можно заметить они являются довольно трудозатратными для разработчиков и из-за этого дорогими для бизнеса. Более того они не предохраняют от излишне затратного использования ресурсов пользователями или от атак злоумышленниками.

Эта курсовая работа посвящена другому способу борьбы с этой проблемой: ограничением запросов.

## 1.2. Концепция системы

Ограничение запросов заключается в установке ограничений на количество запросов, которые может отправлять один пользователь или IP-адрес в определенный период времени. Таким образом мы предотвращаем возможность превышения его возможностей и обеспечиваем более стабильную работу приложения.

Помимо искомого снижения нагрузки на сервер мы также увеличиваем безопасность приложения за счет:

1. Защиты от распределенных атак отказа в обслуживании (DDoS), где злоумышленники намеренно перегружают сервер большим количеством запросов;
2. Предотвращения злоупотребления или автоматического сканирования ресурсов приложения, таких как попытки перебора паролей или нежелательные боты.

Как и у любого другого решения, у представленного есть свои обратные стороны – неудобство со стороны пользователей, в том числе возможность ограничения легитимных пользователей, а также возможная сложность настройки оптимальных параметров для ограничений.

## 1.3. Основные участники распределенной системы

Разрабатываемое приложение будет предоставлять заказчику ограничение запросов. И на большой картине выглядит следующим образом:



Рисунок 1.1 – Использование приложения

Ограничитель запросов (API Limiter) встроен на серверной части заказчика для простоты его добавления в систему.

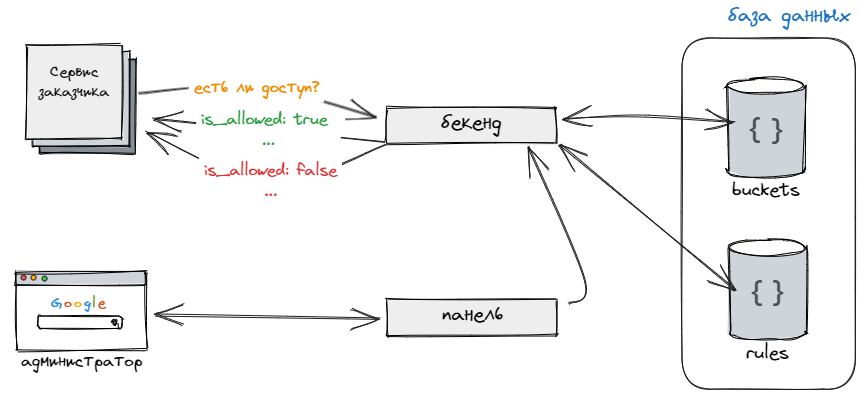


Рисунок 1.2 – Архитектура приложения

Приложение состоит из трех систем:

1. Бэкенд. Приложение которое осуществляет заявленный функционал – определяет может ли клиент получить доступ к ресурсу;
2. Панель управления (фронтенд). Веб приложение которое позволяет конфигурировать Бэкенд, выставлять настройки доступа и собирать аналитику;
3. База данных. Хранилище для конфигурации и действующих ограничений.

# 2. Проектирование системы

## 2.1. Описание системы

Приложение предназначено для ограничения количества запросов к какому-либо ресурсу для снижения нагрузки на этот ресурс. Очень важными его качествами являются скорость и отказоустойчивость.

Оно будет интегрировано на серверной части для простоты установки в существующее приложение. По этой же причине приложение будет использовать IP адрес для наложения ограничений.

Для идентификации ресурса можно использовать домен или весь путь целиком (URL). Выбор упал на URL так как с развитием микросервисной архитектуры разные части приложения могут предоставлять различные сервисы которые отличаются в своей мощности, и ограничения на них могут отличаться. Также в идентификации будет участвовать HTTP метод которым происходит запрос.

Приложение должно состоять из трех составных частей (систем):

* Клиентское приложение (фронтенд) – веб приложение для создания новых правил, редактирования и удаления старых, просмотра действующих ограничений;
* Серверное приложение (Бэкенд) – серверное приложение обеспечивающее бизнес-логику для поддержки ограничений и управления правилами;
* База данных – хранение существующих правил, действующих ограничений и аналитики.

## 2.2. Фронтенд

Фронтенд приложение представляет собой админ панель для работы с приложением – создание и удаление правил, просмотр действующих ограничений, сбор аналитических данных.

Функционал:

* Возможность управления существующими правилами
* Возможность просмотра действующих ограничений
* Скачивание данные запросов в виде csv или json файла

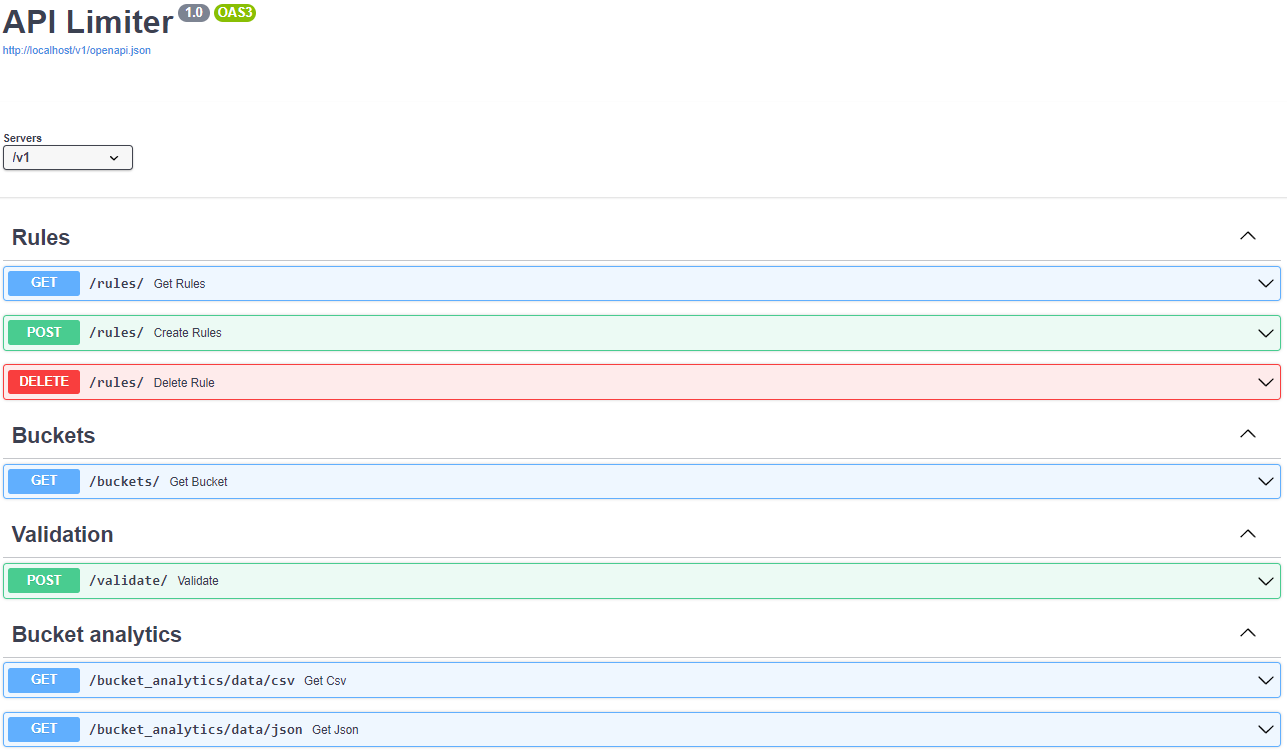


Рисунок 2.1 – Веб интерфейс приложения

## 2.3. Бэкенд

Бэкенд приложение предоставляющее ограничение запросов.

Для разработки будет использован язык Python. С фреймворком FastAPI.

Python – это высокоуровневый, интерпретируемый язык программирования, который обладает простым и понятным синтаксисом. Он широко используется для разработки различных типов приложений, включая веб-приложения.

FastAPI – это быстрый и простой в использовании фреймворк для разработки веб-приложений. Он построен на основе типизации и асинхронности, используя мощные возможности языка Python 3.7+. Благодаря интеграции с Pydantic, FastAPI обеспечивает удобную работу с валидацией данных.

### 2.3.1. API пути сервиса

**/rules** – путь для управления правилами.

GET / Получение списка правил

POST / Создание правил

DELETE / Удаление правила

**/buckets** – путь просмотра ограничений

GET / Получение ограничения

**/validation** – путь валидации запросов

POST / Валидация доступа

/**bucket\_analytics** – путь получения аналитики

GET /bucket\_analytics/data/csv Получение данных в формате csv

GET /bucket\_analytics/data/json Получение данных в формате json

### 2.3.2. Алгоритм валидации

Используемый алгоритм был вдохновлен алгоритмом token bucket.

При использовании этого алгоритма на каждый ресурс выделяется контейнер ‘bucket’ с токенами, и изначально он полон. При прохождении запроса расходуется один токен. Если в контейнере не осталось токенов, то запрос не проходит. Во время получения запроса происходит проверка на то, нужно ли сейчас восстанавливать токены.

Этот алгоритм прост в реализации, производителен и эффективен по памяти, так как каждый контейнер не содержит в себе много информации.

## 2.4. База данных

Так как база данных должна быть максимально быстрой и специфика ее работы предполагает что сохранять данные после завершения работы не обязательно – хорошим выбором является Redis.

Redis – это система хранения данных с открытым исходным кодом (лицензия BSD), используемая как база данных, кэш, посредник сообщений и потоковый движок. Redis предоставляет структуры данных, такие как строки, хэши, списки, множества, упорядоченные множества с запросами диапазона, битовые карты, гиперлоглоги, географические индексы и потоки. Redis имеет встроенную репликацию, скрипты Lua, вытеснение на основе LRU, транзакции и различные уровни сохранения на диске, а также обеспечивает высокую доступность с помощью Redis Sentinel и автоматического разделения с Redis Cluster.

Redis сохраняет данные в формате ключ-значение, и явной организации данных нет. Для этого нужно использовать префиксы в имени, например “users:{user\_id}”. Принято в качестве разделителей использовать двоеточие “:”, но так как для идентификации правил и ограничений используется url который содержит в себе двоеточие, для разделителя будет использоваться хеш “#”.

Redis не предоставляет ограничений целостности или изоляции операций и нужно быть очень аккуратным в том какие данные в него записываются.

Далее описаны сущности которые хранятся в базе данных.

### 2.4.1. Правило (rule)

Объявление ограничения на ресурс. Эта модель идентифицируется по URL ресурса и HTTP методу обращения к нему, и хранит максимальное количество запросов к нему и частоту обновления в секундах

ключ: "rules#{url}#{method}"

тело:

refresh\_rate: int

requests\_max: int

Описание полей:

* В ключе:
  + url. Абсолютный URL ресурса
  + method. HTTP метод обращения к этому ресурсу
* В теле:
  + refresh\_rate. Частота обновления действующего ограничения в секундах
  + requests\_max. Максимальное количество обращений к ресурсу за квоту

### 2.4.2. Контейнер (bucket)

Контейнер с текущим ограничением. Эта модель идентифицируется по URL ресурса, HTTP методу обращения к нему и IP адресу, и хранит количество оставшихся запросов и время первого запроса. Подробнее об алгоритме читайте в описании бекенд приложения.

ключ: "buckets#{url}#{method}#{ip}"

тело:

requests\_left: int

first\_request\_timestamp: float

Описание полей:

* В ключе:
  + url. Абсолютный URL ресурса
  + method. HTTP метод обращения к этому ресурсу
  + ip. IP v4 или IP v6 адрес устройства которое пытается получить доступ
* В теле:
  + requests\_left. Количество доступных запросов
  + first\_request\_timestamp. Время первого запроса к ресурсу. Используется для восстановления количество доступных запросов

### 2.4.3. Контейнер для аналитики (bucketsanalytics)

Контейнер с данными о запросе. Эта модель идентифицируется по URL ресурса, HTTP методу обращения к нему, IP адресу и временной метке, и хранит информацию о том был ли запрос успешен.

ключ: "bucketsanalytics#{url}#{method}#{ip}#{timestamp}"

тело:

was\_allowed: bool

Описание полей:

* В ключе:
  + url. Абсолютный URL ресурса
  + method. HTTP метод обращения к этому ресурсу
  + ip. IP v4 или IP v6 адрес устройства которое пытается получить доступ
  + timestamp. POSIX временная метка с временем запроса
* В теле:
  + was\_allowed. Был ли запрос успешен

## 2.5. Docker

Разрабатываемая система является распределенной и для простоты разработки и сопровождения будет использоваться Docker.

Docker – это платформа для виртуализации на уровне операционной системы, которая позволяет упаковывать приложения и все их зависимости в единую единицу, называемую контейнером. Контейнеры Docker обеспечивают легковесную и изолированную среду выполнения приложений, которая позволяет разработчикам и системным администраторам упрощать развертывание и управление приложениями.

Каждая часть приложения – фронтенд, бекенд и база данных, будут собраны в контейнеры и в дальнейшем могу быть развернуты с помощью Kubernetes каждый на своем сервере. Но сейчас так как приложение только находится в разработке будет использоваться docker-compose для упрощения разработки и определения отношений между системами

# 3. Реализация

Структура проекта выглядит следующим образом:

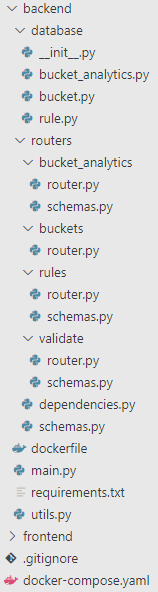


Рисунок 3.1 – Структура проекта

## 3.1. Docker

Ниже приведен docker-compose.yaml файл для сборки проекта:

version: "3.9"

services:

  redis:

    image: redis:7.0.11-alpine3.18

    volumes:

      - redis\_db:/var/lib/postgresql/data

    networks:

      - main

  backend:

    build: ./backend

    depends\_on:

      - redis

    ports:

      - "80:80"

    networks:

      - main

  frontend:

    build: ./frontend

    depends\_on:

      - backend

    ports:

      - "8080:80"

    networks:

      - main

volumes:

  redis\_db:

networks:

  main:

Здесь описаны 3 сервиса:

* redis. База данных с прикрепленным к ней томом (volume) для сохранения данных (правил). База данных доступна только для остальных сервисов в сети для безопасности;
* backend. Бекенд приложение доступное на порту 80;
* frontend. Фронтенд приложение доступное на порту 8080.

Все 3 сервиса находятся в одной сети main

## 3.2. Фронтенд

Благодаря тому что FastAPI предоставляет автоматически сгенерированную документацию и openapi.json документ с ее описанием, фронтенд приложение можно сильно упростить использовав готовую библиотеку для рендера этой документации. В этой работе я остановился на Swagger UI.

Swagger UI – это интерактивное веб-приложение, которое позволяет визуализировать и тестировать API на основе спецификации OpenAPI (ранее называлась Swagger). Swagger UI предоставляет пользовательский интерфейс для взаимодействия с API, позволяя разработчикам легко понять и использовать доступные конечные точки и операции.

Ниже приведен index.html файл:

<!DOCTYPE html>

<html>

<head>

    <link type="text/css" rel="stylesheet" href="https://cdn.jsdelivr.net/npm/swagger-ui-dist@4/swagger-ui.css">

    <link rel="shortcut icon" href="https://fastapi.tiangolo.com/img/favicon.png">

    <title>API Limiter</title>

</head>

<body>

    <div id="swagger-ui">

    </div>

    <script src="https://cdn.jsdelivr.net/npm/swagger-ui-dist@4/swagger-ui-bundle.js"></script>

    <!-- `SwaggerUIBundle` is now available on the page -->

    <script>

        const ui = SwaggerUIBundle({

            url: "http://localhost/v1/openapi.json",

            dom\_id: "#swagger-ui",

            layout: "BaseLayout",

            deepLinking: true,

            showExtensions: true,

            showCommonExtensions: true,

            presets: [

                SwaggerUIBundle.presets.apis,

                SwaggerUIBundle.SwaggerUIStandalonePreset

            ],

        })

    </script>

</body>

</html>

Этот код интегрирует Swagger UI для отображения интерактивной документации API. Swagger UI использует OpenAPI-спецификацию, которая загружается с URL-адреса http://localhost/v1/openapi.json. Swagger UI будет размещен внутри элемента с идентификатором #swagger-ui. Также включены настройки для расширений, глубокой привязки и макета, а также пресеты для загрузки документации API.

Ниже приведен dockerfile для сборки изображения:

FROM nginx:1.25.1-alpine3.17-slim

COPY . /usr/share/nginx/html/

CMD ["nginx", "-g", "daemon off;"]

Для предоставления доступа к файлу используется сервер nginx в который загружается html файл.

Nginx – это свободный, открытый веб-сервер и прокси-сервер, который также может выполнять роль обратного прокси, балансировщика нагрузки, HTTP кеша и раздачи статического контента. Он был разработан с целью обеспечить высокую производительность, надежность, масштабируемость и эффективность обработки одновременных подключений. Nginx использует асинхронную и событийно-ориентированную модель обработки, которая позволяет эффективно обслуживать большое количество одновременных запросов при небольшом потреблении ресурсов.

## 3.3. Бекенд

Высокоуровневая структура бекенд приложения включает в себя:

* requirements.txt. Файл с зависимостями приложения
* /main,py. Файл с FastAPI приложением, здесь происходит его создание и настройка
* /database/. Папка с объектами для работы с базой данных
  + /database/\_\_init\_\_.py. Подключение к базе данных и создание Singleton экземпляров для классов репозиториев
  + /database/bucket.py. Класс данных Bucket и репозиторий BucketStorage для работы с ним
  + /database/rule.py. Класс данных Rule и репозиторий RuleStorage для работы с ним
* /routers/. Папка с ендпоинтами и сервисной логикой. Содержит в себе подпапки для каждого ендпоинта, каждая из которых может содержать:
  + router.py. Файл с рутером
  + schemas.py. Файл с моделями данных которые принимаются от клиента или отдаются ему

A picture containing text, diagram, screenshot, line

Description automatically generated

Рисунок 3.2 – Жизнь запроса

Ниже приведен **dockerfile** для сборки изображения:

FROM python:3.11.4-alpine3.18

ADD ./requirements.txt /requirements.txt

RUN pip install --upgrade -r /requirements.txt

COPY . /app

CMD [ "uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80" ]

Для запуска приложения используется сервер uvicorn.

Uvicorn (короткое от "UVloop" и "COroutines") – это ASGI (Asynchronous Server Gateway Interface) сервер, который обеспечивает высокопроизводительное веб-приложение на основе асинхронного программирования. Uvicorn создан для запуска веб-приложений, написанных на языке Python, с использованием фреймворков, таких как FastAPI, Starlette и других, которые поддерживают ASGI.

**Для подключения к базе данных** redis используется клиентская библиотека redis версии 4.5.5. Подключение выглядит следующим образом:

connection = Redis(

    host="redis",

    port=6379,

    decode\_responses=True,

)

Где host – имя докер контейнера на котором хостится redis, decode\_responses – флаг при выставлении которого библиотека декодирует ответы redis из байтов в строки.

Для абстракции логики хранения данных были созданы классы репозитории, например RuleStorage ниже.

class RuleStorage:

    \_\_connection: Redis

    def \_\_init\_\_(self, connection: Redis) -> None:

        self.\_\_connection = connection

    def save(self, rule: Rule) -> None:

        self.\_\_connection.hset(

            name=f"rules#{rule.url}#{rule.method.value}",

            mapping={

                "refresh\_rate": rule.refresh\_rate,

                "requests\_max": rule.requests,

            },

        )

    def delete(self, url: str, method: HTTPMethod) -> bool:

        return bool(self.\_\_connection.delete(f"rules#{url}#{method.value}"))

    def get(self, url: str = None, method: HTTPMethod = None) -> list[Rule]:

        if url is not None and method is not None:

            redis\_rule = self.\_\_connection.hgetall(f"rules#{url}#{method.value}")

            if not redis\_rule:

                return []

            return [Rule(

                url=url,

                method=method,

                refresh\_rate=int(redis\_rule["refresh\_rate"]),

                requests=int(redis\_rule["requests\_max"]),

            )]

        pattern = f"rules#{url if url is not None else '\*'}#{method.value if method is not None else '\*'}"

        keys: list[str] = self.\_\_connection.keys(pattern)

        rules = []

        for key in keys:

            redis\_rule = self.\_\_connection.hgetall(key)

            key = key.split("#")

            url, method = key[1], HTTPMethod(key[2])

            rules.append(Rule(

                url=url,

                method=method,

                refresh\_rate=int(redis\_rule["refresh\_rate"]),

                requests=int(redis\_rule["requests\_max"]),

            ))

        return rules

Такая практика осложняет использование сессий, но благодаря тому что в redis их нет, об этом можно не беспокоится.

**Валидацию данных** нам предоставляет библиотека pydantic, возьмем пример с одним из ендпоинтов /rules рутера:

@router.post("/", response\_model=list[Rule])

def create\_rules(rules\_body: CreateRules\_Body):

    rules = []

    for url in rules\_body.urls:

        for method in rules\_body.methods:

            rules.append(Rule(

                url=url,

                method=method,

                refresh\_rate=rules\_body.refresh\_rate,

                requests=rules\_body.requests,

            ))

    for rule in rules:

        rule\_storage.save(rule)

    return rules

Здесь CreateRules\_Body является моделью данных которую пользователь отправляет в приложение. Ее определение:

class CreateRules\_Body(BaseModel):

    urls: list[HttpUrl]

    methods: list[HTTPMethod]

    requests: int

    refresh\_rate: int

    @validator("urls", "methods")

    def v\_lists\_cannot\_be\_empty(cls, v: list[str]) -> list[str]:

        if len(v) < 1:

            raise ValueError("cannot be empty")

        return v

    @validator("urls", "methods")

    def v\_lists\_should\_be\_unique(cls, v: list[str]) -> list[str]:

        if len(set(v)) != len(v):

            raise ValueError("all values should be unique")

        return v

    @validator("requests", "refresh\_rate")

    def v\_integers\_should\_be\_more\_than\_zero(cls, v: int) -> int:

        if v < 1:

            raise ValueError("should be more than 0")

        return v

С помощью стандартных типов например HttpUrl и кастомных валидаторов мы можем полностью валидировать входящие данные вне бизнес логики.

Однако в некоторых местах библиотека нас ограничивает, в примере выше модель данных передается в теле запроса. Если же мы захотим использовать ее в query параметрах запроса, то использование кастомных валидаторов будет нежелательным так как теряется стандартное отображение 422 Validation Error ошибок.

Тем не менее это ограничение можно обойти:

@router.get("/", response\_model=list[Rule])

def get\_rules(body: GetRules\_Body = Depends(make\_dependable(GetRules\_Body))):

    return rule\_storage.get(body.url, body.method)

class GetRules\_Body(BaseModel):

    url: HttpUrl | None

    method: HTTPMethod | None

def make\_dependable(cls):

    def init\_cls\_and\_handle\_errors(\*args, \*\*kwargs):

        try:

            signature(init\_cls\_and\_handle\_errors).bind(\*args, \*\*kwargs)

            return cls(\*args, \*\*kwargs)

        except ValidationError as e:

            for error in e.errors():

                error['loc'] = ['query'] + list(error['loc'])

            raise HTTPException(422, detail=e.errors()) from e

    init\_cls\_and\_handle\_errors.\_\_signature\_\_ = signature(cls)

    return init\_cls\_and\_handle\_errors

Ссылку на детальный разбор проблемы и решения я оставлю в используемых ресурсах.

**Ограничение запросов** происходит в рутере /validation и выглядит следующим образом:

A picture containing text, screenshot, diagram, line

Description automatically generated

Рисунок 3.3 – Алгоритм ограничения запросов

# Заключение

В заключение данной курсовой работы можно с уверенностью сказать, что разработка распределенной системы управления нагрузкой вычислительной системы с использованием ограничения запросов является актуальной и значимой задачей в современном информационном обществе.

В ходе работы было разработано веб-приложение, которое предоставляет механизм ограничения запросов к вычислительной системе. Приложение было реализовано с использованием языка программирования Python и фреймворка FastAPI. Этот выбор технологий обеспечил высокую производительность, гибкость и удобство в разработке.

Приложение предоставляет возможность устанавливать ограничения на количество запросов, которые могут быть обработаны за определенный период времени. Это позволяет эффективно контролировать и распределять нагрузку на вычислительную систему, предотвращая ее перегрузку и обеспечивая стабильную работу приложений.

Основные преимущества разработанного приложения включают простоту в использовании, гибкость настройки правил ограничения, а также возможность масштабирования для работы в распределенной среде. Приложение также обладает надежной системой обработки ошибок и механизмами безопасности для защиты от злоумышленных действий.

# Используемые ресурсы

1. Alex Hu – System design interview 
   * url: <https://www.amazon.com/System-Design-Interview-insiders-Second/dp/B08CMF2CQF>
2. Python documentation
   * url: https://docs.python.org/3/
3. FastAPI documentation
   * url: https://fastapi.tiangolo.com/em/
4. FastAPI Best Practices
   * url: https://github.com/zhanymkanov/fastapi-best-practices
5. Use pydantic's BaseModel as dependency with validators
   * url: https://github.com/tiangolo/fastapi/issues/1474
6. Redis documentation
   * url: https://redis.io/docs/
7. Docker documentation
   * url: https://docs.docker.com/
8. 4 Rate Limit Algorithms Every Developer Should Know
   * url: <https://betterprogramming.pub/4-rate-limit-algorithms-every-developer-should-know-7472cb482f48>
9. Diagrams.net
   * url: https://draw.io/
10. Excalidraw | Hand-drawn look & feel
    * url: https://excalidraw.com/