Теория

“Исследование методов межсервисного взаимодействия для передачи данных большого объема.”

# АННОТАЦИЯ

**- Выпускная квалификационная работа содержит \_ страниц, \_ рисунков, \_ таблиц и \_ источников.**

**- Ключевые слова:** межсервисная передача данных, протоколы передачи данных, сервисная архитектура, архитектуры передачи данных, скорость передачи, данные большого объема.

**- Объектом исследования являются** методы межсервисного взаимодействия.

**- Предметом исследования является** алгоритм сравнения передачи данных большого объема в сервисно-ориентированной архитектуре.

**- Целью работы является** исследовать и сравнить методы передачи данных большого объема в сервисно-ориентированной архитектуре. (практически выяснить положительные и отрицательные стороны различных подходов передачи данных большого объема)

**- Результатом работы является** практически доказанное сравнение передачи данных большого объема в сервисно-ориентированной архитектуре, обоснование возможности или невозможности, а также удобства использования каждого метода. (получить наилучший способ передачи, понять негативные стороны остальных методов)

# АННОТАЦИЯ 2

**- Выпускная квалификационная работа содержит \_ страниц, \_ рисунков, \_ таблиц и \_ источников.**

**Ключевые слова:** межсервисное взаимодействие, передача данных, Python, сервисно-ориентированная архитектура, большие файлы, изображения, производительность.

**Объектом исследования является:** Система передачи данных большого объема в сервисно-ориентированной архитектуре.

**Предметом исследования является:** Анализ и сравнение методов передачи больших объемов данных в сервисно-ориентированной архитектуре, включая вариации передачи одного большого файла, а также множества маленьких.

**Целью работы является:** Определение наиболее эффективного и быстрого метода передачи данных большого объема в сервисно-ориентированной архитектуре.

**Результатом работы является:** Выявление оптимального метода передачи данных в сервисно-ориентированной архитектуре, позволяющего обеспечить наиболее быструю и эффективную передачу как одного большого файла, так и множества маленьких файлов

# ОГЛАВЛЕНИЕ

**- Список определений и сокращений**

**- Введение**

**- Постановка задачи**

- Глава 1. Обзор сервисно-ориентированных архитектур и протоколов HTTP

- 1.1. Что такое сервисы, виды архитектур?

- 1.2. Про протоколы http 1, http 2

- 1.3. Сравнение архитектурных стилей API: SOAP, REST, GraphQL, RPC и WebSocket

- 1.4. Вывод по главе 1

- Глава 2. Про методы передачи данных большого объема клиент-серверной системы на Python.

- 2.1. Про FastApi

- 2.1.1. Синхронная передача

- 2.1.2. Асинхронная передача

- 2.1.3. Почему не использую Uploadfile

- 2.1.4. Про потоки (concurrent.futures)

- 2.1.5. Про uvicorn и hypercorn

- 2.2. gRPC :

- 2.2.1. Унарная передача

- 2.2.2. Поток от клиента

- 2.2.3. Двунаправленная потоковая передача

- 2.3. Веб сокеты

- 2.4. Защищенный http 1

- 2.5. Вывод по главе 2

- Глава 3. Про реализацию вышеупомянутых методов.

- 3.1. Про реализацию всех этих методов на http 1 (для многих и одного)

- 3.2. Отличие реализации на http 2 (для многих и одного)

- 3.3. Разница передачи

- 3.4. Вывод по главе 3

- Глава 4. Сравнение производительности различных методов передачи данных и выводы.

- 4.1. Методология сравнения

- 4.2. Результаты сравнения передачи одного большого файла

- 4.3. Результаты сравнения передачи множества маленьких файлов

- 4.4. Вывод по главе 4

- Глава 5. Комплекс мероприятий по охране труда для обеспечения безопасных условий на рабочем месте.

**- Заключение**

**- Список используемой литературы**

# СПИСОК ОПРЕДЕЛЕНИЙ И СОКРАЩЕНИЙ

В таблице 1 приведены определения и сокращения, используемые в данной работе.

| ASGI (Asynchronous Server Gateway Interface) | клиент-серверный протокол который дает доступ к функциям параллельного выполнения кода |
| --- | --- |
| WSGI (Web Server Gateway Interface) | протокол с последовательной обработкой запросов |
|  |  |
|  |  |
|  |  |
|  |  |

# ВВЕДЕНИЕ

**- Актуальность**

**- Про подобные исследования. Сравнение с другими имеющимися сейчас в свободном доступе работами.**

**- В данной работе объектом исследования является**

**- Предметом исследования является**

**- Целью работы является**

**- Для достижения поставленной цели в данной работе решался ряд задач:**

1. **Задача 1**
2. **Задача 2**
3. **Задача 3**

**- При проведении исследований использовались следующие методы:**

1. **метод 1**
2. **метод 2**
3. **метод 3**

**- Научной новизной обладают следующие работы:**

1. **новизна 1**
2. **новизна 2**

**- Практическая значимость работы**

**- Основная часть магистерской диссертации состоит из \_ разделов:**

1. **раздел 1**
2. **раздел 2**
3. **раздел 3**

# ВВЕДЕНИЕ 2

**- Актуальность**

В эпоху цифровизации и роста объемов данных, эффективность и скорость передачи данных становятся критически важными. С каждым днем количество информации, которую необходимо обрабатывать и передавать, увеличивается, что создает потребность в более эффективных методах передачи данных. Это особенно важно в контексте сервисно-ориентированных архитектур, где межсервисное взаимодействие является ключевым элементом. Для быстрого обмена данными между сервисами требуется изучить современные методы передачи данных. Использование современных алгоритмов открывает большое количество возможностей, в том числе способность работать в реальном времени. Данная работа направлена на изучение и оптимизацию этого аспекта, что делает ее актуальной и важной в современном мире.

**- Подобные исследования. Сравнение с другими работами**

Существует множество исследований, посвященных передаче данных большого объема. Однако большинство из них сосредоточены на определенных типах данных или не учитывают специфику сервисно-ориентированных архитектур. Кроме того, многие из них не рассматривают различные сценарии передачи данных, такие как передача одного большого файла или множества маленьких файлов. Ваша работа направлена на заполнение этих пробелов. Вы не только рассматриваете различные сценарии передачи данных, но и стремитесь найти наиболее эффективный метод для каждого из них. Это делает вашу работу уникальной и важной для текущего исследовательского контекста. Ваша работа может стать отправной точкой для дальнейших исследований в этой области и способствовать развитию новых и более эффективных методов передачи данных.

**- В данной работе объектом исследования является** система передачи данных большого объема в сервисно-ориентированной архитектуре.

**- Предметом исследования является** анализ и сравнение методов передачи больших объемов данных в сервисно-ориентированной архитектуре, включая вариации передачи одного большого файла, а также множества маленьких.

**- Целью работы является** определение наиболее эффективного и быстрого метода передачи данных большого объема в сервисно-ориентированной архитектуре.

**Для достижения поставленной цели в данной работе решались следующие задачи:**

**Задача 1:** Изучение сервисно-ориентированных архитектур и протоколов HTTP 1 и HTTP 2.

**Задача 2:** Разработка и реализация клиент-серверной системы на Python для передачи данных большого объема.

**Задача 3:** Сравнение производительности различных методов передачи данных.

**При проведении исследований использовались следующие методы:**

**Метод 1:** Анализ источников, находящихся в свободном доступе, описывающих работу приложений с сервисно - ориентированной архитектурой

**Метод 2:** Анализ и сравнение различных методов передачи данных.

**Метод 3:** Разработка и тестирование клиент-серверной системы, проверка работы различных подходов и протоколов.

**Метод 4:** Использование синхронной и асинхронной передачи данных в FastApi, потоков, gRPC и веб-сокетов для передачи данных.

**Метод 5:** Процентное соотношение и визуальное сравнение результатов

**Научной новизной обладают следующие работы:**

**Новизна 1:** Исследование и сравнение производительности различных методов передачи данных в сервисно-ориентированных архитектурах.

**Новизна 2:** Рассмотрение различных протоколов передачи, рассмотрение передачи данных в общем виде.

**Практическая значимость работы:** Результаты этой работы могут быть использованы для оптимизации передачи данных в сервисно-ориентированных архитектурах. Они могут помочь разработчикам выбрать наиболее эффективный метод передачи данных, что в свою очередь может улучшить производительность и надежность их систем.

**Основная часть магистерской диссертации состоит из следующих разделов:**

**Раздел 1:** Обзор сервисно-ориентированных архитектур и протоколов HTTP 1 и HTTP 2.

**Раздел 2:** Содержит информациюпро методы передачи данных большого объема клиент - серверной системы на Python.

**Раздел 3:** Содержит информацию про реализацию вышеупомянутых методов.

**Раздел 4:** Сравнение производительности различных методов передачи данных и выводы.

**Раздел 5:** Комплекс мероприятий по охране труда для обеспечения безопасных условий на рабочем месте.

# ПОСТАНОВКА ЗАДАЧИ

**- В данной работе необходимо** изучить сервисно - ориентированную архитектуру, протоколы передачи данных. После этого исследовать методы передачи данных в рассмотренной архитектуре. Также ее нужно реализовать и протестировать на различные методы передачи больших данных. Необходимо проделать вышеперечисленную работу для двух видов передаваемых данных - большого количества файлов маленького размера и одного файла большого размера. Для достижения поставленной цели необходимо произвести сравнение результатов, сделать выводы.

**- В рамках выполнения данной работы были поставлены следующие задачи:**

1. Изучение сервисно - ориентированной архитектуры и протоколов передачи данных
2. Исследование существующих подходов взаимодействия для передачи данных большого объема в сервисной архитектуре
3. Разработка алгоритма взаимодействия
4. Проведение исследований для оценки скорости передачи каждого метода.

# ЗАКЛЮЧЕНИЕ

**- Целью данной работы являлось**

**- Для достижения поставленной цели в работе был проведен**

**- Результаты исследований показывают, что**

**- Представленные результаты показывают, что все цели поставленные в данной работе были достигнуты.**

# СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

—---------------------------------—---------------------------------—---------------------------------—------------

#### Про сервисы

Различные приложения на различных узлах сети функционируют на разных аппаратно-программных платформах, и используют различные технологии и языки. Чтобы связать все это и предоставить возможность одним приложениям обмениваться данными с другими, и были придуманы веб-сервисы.

По сути, веб-сервисы — это реализация абсолютно четких интерфейсов обмена данными между различными приложениями, которые написаны не только на разных языках, но и распределены на разных узлах сети. Именно с появлением веб-сервисов развилась идея SOA — сервис-ориентированной архитектуры веб-приложений (Service Oriented Architecture).

Связывание сервисов позволяет разделять сложные системы на отдельные модули, каждый из которых отвечает за конкретную функциональность. Это упрощает понимание и поддержку кода, позволяет повторно использовать уже написанный функционал и легко вносить изменения, не затрагивая другие модули.

Связывание сервисов позволяет легко добавлять новые сервисы и компоненты в систему.

Связывание сервисов позволяет конструировать системы с помощью различных комбинаций сервисов.Можно использовать только те сервисы, которые нужны, и не тратить ресурсы на ненужный функционал.

Каждый сервис может быть масштабирован независимо от других.

## Глава 1 Обзор сервисно-ориентированных архитектур и протоколов HTTP

### **1.1. Что такое сервисы, виды архитектур.**

Сервисы в контексте программного обеспечения - это отдельные функциональные единицы, которые могут быть использованы в различных приложениях и бизнес-процессах. Сервисы обычно предоставляют определенный набор функций и взаимодействуют друг с другом через определенные протоколы и интерфейсы.

Сервисно-ориентированная архитектура (SOA) - это подход к разработке программного обеспечения, который позволяет создавать приложения в виде набора слабо связанных сервисов. В SOA, сервисы обычно взаимодействуют друг с другом посредством протоколов обмена сообщениями.

Существуют различные виды архитектур, включая монолитную, микросервисную, и серверную архитектуру.

* Монолитная архитектура представляет собой традиционный подход к разработке программного обеспечения, где все компоненты приложения объединены в один модуль или единицу. Это может упростить разработку и тестирование, но может привести к проблемам с масштабируемостью и гибкостью приложения.
* Микросервисная архитектура - это подход, при котором приложение строится как набор небольших сервисов, каждый из которых выполняет конкретную бизнес-функцию и может разрабатываться, развертываться и масштабироваться независимо. Это может улучшить гибкость и масштабируемость приложения, но также может увеличить сложность управления и координации между сервисами.
* Серверная архитектура - это подход, при котором логика приложения разделена между сервером и клиентом. Сервер обычно отвечает за обработку данных и бизнес-логику, в то время как клиент отвечает за представление данных пользователю. Это может улучшить производительность и эффективность приложения, но также может увеличить сложность управления и синхронизации между сервером и клиентом.

Каждый из этих подходов имеет свои преимущества и недостатки, и выбор между ними обычно зависит от конкретных требований и контекста приложения. В контексте данной работы, вы заострено внимание на сервисно-ориентированной архитектуре и изучаются методы передачи данных в таких системах.

Существует большое количество различных способов передачи информации между сервисами. Все они основаны на различных подходах к хранению и обмену данными. Для лучшего понимания стоит более детально рассмотреть протоколы, среди которых особое внимание заслуживают НТТР и его последняя версия - HTTP/2.

### 1.1.1. Про API

API (Application Programming Interface) — технология, позволяющая соединить функциональность разных компьютерных программ. API можно сравнить с официантом, который получает от клиента заказ из ограниченных пунктов меню, передает его на кухню («системе»), а после приготовления, возвращает готовые «блюда» заказчику.

Почти все, с чем мы сталкиваемся в интернете, имеет отношение к API, а точнее к версиям этого программного интерфейса, использующим для работы HTTP-запросы. Когда мы хотим узнать прогноз погоды, интерфейс браузера или мобильного приложения вызывает API Яндекс.Погоды или API Gismeteo. Когда прокладываем кратчайший маршрут из одного места в другое, Яндекс.Карты вызывают соответствующее API.

Пользовательские API-интерфейсы могут быть реализованы на Python с использованием нескольких фреймворков. В этой статье остановимся на особенностях работы с одним из самых популярных вариантов — платформой FastAPI, библиотеки которой активно используют такие технологические гиганты, как Microsoft, Netflix, Uber. Речь пойдет о некоторых расширенных функциях FastAPI, которые могут использовать в своих проектах те разработчики, у кого уже есть базовые знания о фреймворке.

### **1.2. Про протоколы http 1, http 2**

HTTP/1.1 поддерживает множество запросов через одно соединение, дает возможность отправлять несколько запросов без ожидания ответов, также позволяет виртуальным хостам существовать на одном сервере, но он все еще подвержен проблемам блокировок из-за одного соединения, особенно при множестве мелких файлов.

HTTP/2 был представлен в 2015 году для улучшения производительности. Множество запросов может выполняться параллельно в рамках одного соединения, разделенного на потоки. Его плюсами также является уменьшение объема передаваемых данных, возможность устанавливать приоритеты запросов, возможность сервера предварительно отправлять ресурсы, которые могут быть запрошены клиентом.

В целом HTTP/2 предоставляет значительное улучшение производительности за счет множественных потоков, сжатия заголовков и других оптимизаций. Он особенно эффективен при загрузке больших веб-страниц и ресурсов, снижая задержки и ускоряя время загрузки.

Программы обычно сталкиваются с данными в различных форматах: данные, хранящиеся в памяти, и данные, предназначенные для записи в файлы и передачи по сети. Обычные структуры данных в памяти оптимизированы для эффективного взаимодействия с центральным процессором. Однако, для передачи по сети или записи в файл, эти данные необходимо преобразовать в последовательность байтов. Этот процесс называется кодированием или сериализацией. Разнообразные форматы, такие как JSON, XML и CSV, используются для представления данных в виде последовательности байтов.

Существует несколько способов организации взаимодействия процессов по сети, включая SOAP, REST, GraphQL, RPC и WebSocket.

### **1.3. Сравнение архитектурных стилей API: SOAP, REST, GraphQL, RPC и WebSocket**

В сервисно-ориентированной архитектуре (SOA) используются различные стили API, каждый из которых имеет свои особенности, преимущества и недостатки.

**SOAP** (Simple Object Access Protocol) - это протокол обмена структурированными сообщениями в распределенной вычислительной среде. SOAP был разработан для обеспечения взаимодействия между объектами по сети, в основном для реализации удаленного вызова процедур (RPC). SOAP использует формат XML для запросов и ответов, что обеспечивает строгую типизацию, но может увеличить объем передаваемых данных. SOAP также поддерживает различные протоколы транспорта, включая HTTP, SMTP и даже JMS, что делает его универсальным решением для различных сценариев использования.

**REST** (Representational State Transfer) - это архитектурный стиль, который использует принципы HTTP для создания API. В отличие от SOAP, который использует формат XML для запросов и ответов, REST не имеет фиксированного формата. В REST можно обмениваться сообщениями на основе XML, JSON или любого другого удобного формата. REST использует стандартные HTTP-методы, такие как GET, POST, PUT и DELETE, для взаимодействия с ресурсами, что делает его простым и интуитивно понятным для разработчиков.

**GraphQL** - это относительно новый стиль API, который позволяет клиентам определять структуру запросов и ответов. Это может упростить интеграцию и улучшить производительность за счет уменьшения количества данных, передаваемых по сети. GraphQL позволяет клиентам запрашивать только те данные, которые им действительно нужны, что может существенно уменьшить объем передаваемых данных и улучшить производительность.

**RPC** (Remote Procedure Call) - это спецификация, которая позволяет удаленно выполнять функцию в другом контексте. RPC расширяет понятие локального вызова процедуры, но помещает его в контекст HTTP API. RPC может быть реализован с использованием различных протоколов, включая SOAP и JSON-RPC, и может поддерживать различные стили взаимодействия, включая запрос-ответ, уведомления и даже потоковую передачу данных.

**WebSocket** - это протокол, который обеспечивает двустороннее взаимодействие между клиентом и сервером в реальном времени. WebSocket может быть полезен для приложений, которым требуется непрерывное взаимодействие между клиентом и сервером, таких как чаты, игры в реальном времени и другие интерактивные приложения.

### **1.4. Вывод главы 1**

Сервисы представляют собой отдельные функциональные единицы, которые могут быть использованы в различных приложениях и бизнес-процессах. В сервисно-ориентированной архитектуре (SOA) сервисы обычно взаимодействуют друг с другом посредством протоколов обмена сообщениями.

Рассмотренные протоколы HTTP 1 и HTTP 2 являются основой для обмена данными в веб-приложениях и имеют ряд отличий, которые могут влиять на производительность и надежность приложений.

Проанализировав различные стилей API, включая REST, RPC и WebSocket, можно переходить к реализации их определенных сценариев использования.

## Глава 2. Про методы передачи данных большого объема клиент-серверной системы на Python.

В этой главе мы погрузимся в детали различных методов передачи данных большого объема в контексте клиент-серверной системы, реализованной на языке программирования Python. Мы начнем с обзора FastAPI - современного, быстрого (высокопроизводительного), веб-фреймворка для создания API с Python 3.6+ на основе стандартных Python-типов подсказок.

Мы рассмотрим различные способы реализации передачи данных с помощью FastAPI, включая синхронную и асинхронную передачу данных. Мы обсудим, почему в нашем случае не используется Uploadfile и какие преимущества мы получаем от передачи данных в виде байтов.

Далее мы перейдем к обсуждению потоков (concurrent.futures) и их использованию для улучшения производительности нашей системы. Мы также рассмотрим uvicorn и hypercorn - два сервера ASGI, которые мы используем для обработки HTTP-запросов.

Затем мы перейдем к gRPC - современному, открытому протоколу удаленного вызова процедур (RPC), который может работать поверх HTTP/2. Мы рассмотрим различные методы, которые предоставляет gRPC, и как они могут быть использованы для передачи данных.

Наконец, мы обсудим веб-сокеты и защищенный HTTP 1, два дополнительных метода передачи данных, которые могут быть полезны в определенных сценариях.

### 2.1. Про FastApi

—-----------------------------

Почему FastAPI

FastAPI — современная, высокопроизводительная веб-инфраструктура для создания готовых к эксплуатации HTTP API-серверов с помощью Python 3+. Производительность FastAPI можно сравнить с NodeJS и Go, поэтому он считается одним из самых быстрых фреймворков Python. Структура разработки очень похожа на Flask, который де-факто является базовым вариантом для всех, кто начинает веб-разработку с использованием Python.

FastAPI прост в использовании, хорошо задокументирован и обладает встроенными валидацией, сериализацией и асинхронностью «из коробки». Фреймворк предлагает все стандартные функции для создания API, но не ограничивается этим. Инструмент обладает большой гибкостью и множеством полезных функций (например, поддержкой обратного монтирования WSGI), которые остаются вне поля внимания большинства пользователей.

—---------------------------------

FastAPI - является легковесным асинхронным фреймворком для Python, который используют преимущественно для разработки API-сервисов.

Плюсы разработки на FastAPI:

* Скорость работы. FastAPI обходит все Python-фреймворки по производительности
* Гибкость. Нет какой-либо утвержденной архитектуры, что дает волю реализации различных подходов разработки, по сравнению с Django (использует MVC), в котором проект часто превращается в странную субстанцию из папок в папках.
* Автоматическая OpenAPI документация. В Django и Flask, нужно подключать отдельные пакеты и создавать специальные классы для подключения документации к ним. FastAPI автоматически генерирует схему OpenAPI для API, которую можно использовать с интерактивными системами документации.
* Асинхронность. FastAPI использует ASGI-сервера по умолчанию, а например в Django нужно заниматься конфигурацией приложения с WSGI на ASGI, что занимает достаточно времени. Flask совсем не поддерживает асинхронность и работает лишь под WSGI. FastAPI же поддерживает асинхронную обработку запросов с использованием ключевых слов async и await. Это позволяет ему эффективно обрабатывать большое количество запросов параллельно.

FastAPI использует asyncio для обработки запросов асинхронно, что обеспечивает высокую производительность и масштабируемость. Он также использует Starlette для обработки HTTP-запросов, что обеспечивает высокую производительность и надежность.

* Встроенная валидация данных. Pydantic V2 в FastAPI, ускоряет валидацию данных в 20 раз. Pydantic предоставляет готовые типы данных для валидации email, uuid, url и т.д. Вместо того, чтобы вручную писать проверки для содержимого json-файлов и обрабатывать ошибки валидации, он автоматизирует эти процессы.
* Проработанная документация. Документация разделена на две части: для новичков и более продвинутых разработчиков. В ней большое количество примеров и лучших практик, что также является плюсом.

Минусы:

* Мало информации. Это относится к JWT-токенам, S3 облачным хранилищам, сборке образа в Docker Compose и другим технологиям. Зачастую вам приходится изучать реализации на других фреймворках, так как информации не все всегда бывает достаточно в документации FastAPI.
* Плохая реализация асинхронных библиотек. У асинхронных библиотек на Python зачастую нечитабельная документация, из-за чего их подключении к FastAPI иногда отнимает кучу времени.

Остановив выбор на использовании фреймворка FastAPI для реализации клиент - серверного приложения, ниже будут рассмотрены различные способы передачи данных.

#### 2.1.1. Синхронная передача

Синхронная передача данных - это метод передачи, при котором отправитель и получатель данных синхронизируются друг с другом. Это означает, что получатель должен быть готов принять данные в тот момент, когда отправитель готов их отправить.

В контексте клиент-серверной архитектуры, синхронная передача обычно означает, что клиент (или отправитель данных) ожидает ответа от сервера (или получателя данных) перед тем, как продолжить свою работу. Клиент блокируется или ожидает, пока сервер не обработает запрос и не вернет ответ.

В контексте FastAPI, синхронная передача данных может быть реализована с использованием обычных функций Python. Когда клиент делает запрос к серверу, сервер обрабатывает запрос, выполняет необходимые действия (например, чтение или запись данных) и затем возвращает ответ. В течение всего этого процесса клиент ожидает и не может выполнять другие действия.

Синхронная передача данных может быть проста в реализации, но она может быть неэффективной, если обработка запроса занимает много времени, поскольку клиент должен ожидать окончания обработки, прежде чем он сможет продолжить свою работу. Это может привести к задержкам и снижению производительности, особенно при обработке больших объемов данных или при выполнении сложных операций. По этой причине в некоторых случаях может быть предпочтительнее использовать асинхронную передачу данных, которую мы обсудим в следующем разделе.

#### 2.1.2. Асинхронная передача

Асинхронное программирование стало неотъемлемой частью разработки веб-приложений на Python. Это позволяет обрабатывать множество одновременных запросов без блокировки ресурсов, что в свою очередь повышает производительность и отзывчивость приложения.

Asyncio — это библиотека для асинхронного программирования на Python с использованием сопрограмм (coroutines) и цикла событий (event loop). Она позволяет создавать асинхронные приложения, которые могут выполнять множество задач одновременно без блокировки ресурсов.

вызываем эту функцию с использованием ключевого слова **await**, чтобы дождаться её выполнения.

Основные принципы асинхронного программирования, включая понятие событийного цикла, корутин и их использование для реализации асинхронных приложений.

Асинхронная передача данных - это метод передачи, при котором отправитель и получатель данных не обязаны синхронизироваться друг с другом. Это означает, что отправитель может продолжать выполнять другие задачи после отправки данных, а получатель может обрабатывать данные, когда они становятся доступными.

В контексте клиент-серверной архитектуры, асинхронная передача обычно означает, что клиент (или отправитель данных) не ожидает ответа от сервера (или получателя данных) перед тем, как продолжить свою работу. Клиент может продолжать выполнять другие действия, пока сервер обрабатывает запрос и возвращает ответ.

В контексте FastAPI, асинхронная передача данных может быть реализована с использованием асинхронных функций Python, которые определяются с помощью ключевых слов async и await. Когда клиент делает запрос к серверу, сервер начинает обрабатывать запрос, но вместо того чтобы блокировать выполнение и ожидать окончания обработки, он может “отложить” выполнение и переключиться на другие задачи. Когда обработка завершена, сервер возвращает ответ, и клиент может продолжить работу с этого места.

Асинхронная передача данных может быть более сложной в реализации по сравнению с синхронной передачей, но она может значительно улучшить производительность и эффективность системы, особенно при обработке больших объемов данных или при выполнении операций, которые могут занять значительное время (например, операции ввода-вывода). Это делает асинхронную передачу данных особенно полезной в современных веб-приложениях и сервисно-ориентированных архитектурах.

Благодаря ключевому слову async функция обернутая в type - type(fun1) теперь возвращает не <class 'NoneType'>, а <class 'coroutine'>.

Корутина — это разновидность генератора.

Как в python устроен генератор? Генератор это это то, что функция начинает возвращать, если в нее добавить yield вместо return.

Корутина дает интерпретатору возможность возобновить базовую функцию, которая была приостановлена в месте размещения ключевого слова await.

Корутина — это то, что возвращает функция с await. А не сама функция, которая называется - “асинхронная (корутинная) функция”.

С генераторами так же. “Генераторная функция” - функция содержащая yield, а генератор - это именно тот объект, который генераторная функция возвращает.

#### 2.1.3. Почему не использую Uploadfile

UploadFile - это класс, предоставляемый FastAPI для работы с загружаемыми файлами. Он предоставляет удобный интерфейс для чтения и сохранения файлов, полученных от клиента.

Преимущества UploadFile:

Простота использования: UploadFile обеспечивает удобный интерфейс для работы с файлами. Он автоматически обрабатывает многие детали работы с файлами, такие как чтение и запись данных, управление памятью и т.д.

Интеграция с FastAPI: UploadFile тесно интегрирован с FastAPI и может быть легко использован вместе с функциями валидация запросов, обработка исключений и т.д.

Недостатки UploadFile:

Передача метаданных: Когда вы используете UploadFile, вместе с файлом передаются и его метаданные. Это может включать в себя такие вещи, как имя файла, тип содержимого и т.д. В некоторых случаях эта дополнительная информация может замедлить передачу данных.

Неэффективность при работе с большими файлами: UploadFile может быть неэффективным при работе с очень большими файлами. Это связано с тем, что FastAPI должен обрабатывать весь файл, прежде чем он сможет начать отправку ответа. Это может привести к задержкам и увеличению использования памяти.

Решение использовать байты вместо UploadFile для передачи данных основано на том, что передача данных в виде байтов может быть гораздо быстрее. Это особенно важно при работе с большими объемами данных, где любые задержки или неэффективности могут иметь значительное влияние на производительность вашего приложения. Кроме того, передача данных в виде байтов позволяет избежать передачи лишних метаданных, что также может ускорить процесс.

#### 2.1.4. Про потоки (concurrent.futures)

Потоки - это основной инструмент для многозадачности и параллельного выполнения в Python. Они позволяют выполнять несколько операций одновременно, что может значительно улучшить производительность приложения, особенно при работе с операциями ввода-вывода или другими операциями, которые могут блокировать выполнение программы.

Модуль concurrent.futures в Python предоставляет высокоуровневый интерфейс для асинхронного выполнения вызовов функций с использованием потоков или процессов. Он предоставляет классы ThreadPoolExecutor и ProcessPoolExecutor, которые позволяют создавать пул потоков или процессов и использовать их для выполнения функций в фоновом режиме.

В контексте FastAPI и асинхронной передачи данных, потоки могут быть полезны для выполнения длительных операций ввода-вывода без блокировки основного потока выполнения. Это может быть особенно полезно при работе с большими объемами данных, так как позволяет серверу продолжать обрабатывать другие запросы, пока выполняется операция ввода-вывода.

Однако стоит отметить, что использование потоков также может ввести дополнительную сложность в код и может привести к проблемам с производительностью или надежностью, если они не используются правильно.

#### 2.1.5. Про uvicorn и hypercorn

Uvicorn и Hypercorn - это два сервера ASGI, которые используются для обработки HTTP-запросов в асинхронных веб-приложениях на Python, таких как FastAPI.

Uvicorn - это очень быстрый сервер ASGI, который поддерживает веб-сокеты, HTTP/2, автоматическую перезагрузку и многое другое. Uvicorn построен на uvloop и httptools и предоставляет высокопроизводительную основу для ваших асинхронных приложений.

Hypercorn - это также сервер ASGI, который поддерживает HTTP/1, HTTP/2, веб-сокеты и ASGI 3.0. Одним из ключевых преимуществ Hypercorn является его поддержка HTTP/2, что может улучшить производительность приложения за счет использования одного соединения для нескольких потоков данных.

Оба сервера могут быть использованы с FastAPI и другими асинхронными фреймворками Python, и выбор между ними обычно зависит от конкретных требований вашего приложения. В вашем случае, вы используете Uvicorn для обработки запросов по протоколу HTTP/1 и Hypercorn для обработки запросов по протоколу HTTP/2.

### 2.2. gRPC

Текст

#### 2.2.1. Унарная передача

Текст

#### 2.2.2. Поток от клиента

Текст

#### 2.2.3. Двунаправленная потоковая передача

Текст

[Конкурентность и параллельность](https://www.youtube.com/watch?v=GVv7MNo_i2Y)

Конкурентность (concurrency) - запуск на выполнение сразу нескольких задач (не обязательно в 1 момент времени выполняется несколько). Зависит от ПО. Первые ОС с процессором без ядер -использовали только ее. Пример с шахматистом - когда 1 шахматист играет 10 партий, переходя от одной доске к другой

Параллельность (parallel) - конкурентность, когда 2+ задачи выполняются одновременно. Зависит от железа. Вы не можете одновременно (!) выполнять больше задач, чем есть ядер в системе.

Пример с шахматистом - 10 шахматистов параллельно играют партии (10 или 100 партий не важно)

GIL (Global Interpreter Lock) - глобальная блокировка интерпретатора, механизм гарантирующий, что в любой момент времени выполняется только 1 инструкция в питоне.

Задачи для Python могут быть: CPU-bound - зависит от мощности процессора. (например расчеты какие-то)

IO-bound - зависит от системы ввода/вывода (например обращение к сайтам, к базам, файлам)

GIL не трогает IO-bound задачи

threading - IO-bound задачи asyncio - IO-bound задачи multiprocessing - любые задачи

Если мы выполняем что- то внутри питона, например считаем что-то в коде, то GIL не позволит делить задачу и не ускорит работу. даже если задать 10 потоков, исполнитель (грубо говоря 1 ядро) будет бегать между 10 потоками и время выполнения будет такое же как и без потоков.

Но если например задача будет 10 раз обратиться к яндексу по ссылке через библиотеку request, то выполнение через 10 потоков будет гораздо быстрее чем без потоков, так как за выходом из питона полномочия GIL заканчиваются.

Untitled

синие стрелочки - IO запрос

# **Multithreading**

multithreading - многопоточность, подходит для IO-bound задач, использует ОС, страдает от GIL (важно помнить)

Полезно для ускорения выполнения задач или для того, чтобы текущий поток занялся другой задачей.

Любая программа это минимум один процесс и один поток Полезно использовать daemon=True, очереди, pool exeсutor

Плюсы:

* просто(сравнительно с asyncio например)
* быстро
* не умирает из-за одного(!) Если один поток завис, остальные продолжат выполняться , потом останется подтянуть этот один. Минусы:
* потребление ресурсов (ОС) Это стоит памяти оперативной, времени на переключение между потоками, ряд ограничений по количеству потоков связанных с OC.
* неуправляемость (старт, приостановка, переключение) - когда какой поток работает. (эта проблема решена в asyncio)
* проблемы потоков (гонка, блокировки)

На любую программу даже hello world минимум 1 процесс и 1 поток

Процесс - грубо говоря программа (некий набор ресурсов который дает OS по команде. выделяет память, ресурсы, дает id для той программы которую хотим запустить)

Потоки - некие сущности, которые и выполняют работу

daemon = True - все потоки прекратятся сразу как закончится главный поток. при него использовании не забывать писать

for t in threads:

t.join()

то есть дожидаться выполнения остальных потоков.

Не нужно создавать слишком много потоков, так как это использует ресурсы OS

Race condition. Гонка потоков. Когда много потоков дерутся за один ресурс. Это если много потоков заходят в одну и ту же функцию, и начинают решать одну и ту же задачу не обращая внимание на решение других соседних потоков.

Например есть функция в которой к первоначальному числу = 0 прибавляется 1. То есть если функция выполнится последовательно 100 раз , то будет число 100. Если создать 100 потоков и запустить, число все равно будет 100, потому что время выполнения этой задачи очень маленькое (ограничение GIL до 5 милисекунд) но если будет более долгая задача, то эти потоки одновременно будут выполнять эту задачу с точки входа и каждый к 0 будет прибавлять 1, и в итоге на выходе будет 1.

counter = [0]

def inc():

c = сounter[0]

time.sleep(1)

counter[0] = c+1

чтобы решить эту проблему в потоках используется блокировка - Lock()

counter = [0]

lock = threading.Lock()

def inc():

lock.acquire() # в этом месте все потоки ждут очередь пока работу не завершит предыдущий

c = сounter[0]

time.sleep(1)

counter[0] = c+1

lock.release() # освобождение блокировки, продолжение работы, переход к следующему потоку

но это является бутылочным горлышком и замедляет работу, так как задачи выполняются последовательно

Если забыть освободить блокировку, то будет происходить зависание (Дедлок)

Очереди Queue(). Используются для обхода блокировок, также когда нужно использовать в потоках общие ресурсы. Например когда просто нужно создать 1000 файлов нет смысла использовать очередь, потому что это IO - задача. Создают очередь из потоков в задаче, и каждый следующий поток знает результат предыдущего потока и использует его. По времени так же долго как и Lock(), как и чисто последовательное выполнение.

counter = [0]

queue = Queue()

queue.put(0) # изначально в очередь надо что-то положить

def inc\_queue():

c = queue.get() # берет результат предыдущего выполнения

time.sleep(1)

queue.put(c+1) # записывает новое значение

Очень хорошо паралелятся независимые функции не влияющие друг на друга и не зависимые друг от друга.

Код примера

import os

import threading

import time

from queue import Queue

from threading import Thread

def waiting(timeout):

while timeout > 0:

timeout -= 1

time.sleep(1)

print("OK")

#################################

def activity():

requests.get("<https://yandex.ru>")

def run(threaded=False):

start = time.time()

if not threaded:

for e in range(10):

activity()

else:

threads = [threading.Thread(target=activity, deamon=True) for \_ in range(10)]

for e in threads:

e.start()

for e in threads:

e.join()

end = time.time()

print(f'Time: {end - start} seconds')

if \_\_name\_\_=='\_\_main\_\_':

run(threaded=False)

#################################

def info():

pid = os.getpod() # номер процесса

name = threading.current\_thread().name # номер потока

print(f"Process {pid}, name {name}")

counter = [0]

def inc():

c = сounter[0]

time.sleep(1)

counter[0] = c+1

def inc\_queue():

c = queue.get() # берет результат предыдущего выполнения

time.sleep(1)

queue.put(c+1) # записывает новое значение

if \_\_name\_\_=='\_\_main\_\_':

threads = [Thread(target=inc\_queue, daemon=True) for \_ in range(100)]

for t in threads:

t.start()

for t in threads:

t.join()

print(queue.size())

print(queue.get\_nowait())

# info()

### ThreadPoolExecutor()

### Asyncio

Асинхронное выполнение подходит для IO-bound задач, работает ровно 1 поток

Всегда в единичный момент времени выполняется 1 корутина.

Event Loop хорошо между ними переключается.

Он переключается только в месте где стоит await. То есть если в какой-то корутиной функции не стоит await, то другие события не начнут выполняться пока не закончится это.

Плюсы:

* скорость и экономия времени, вместо x + y + z = max(x, y, z)
* управляемость
* меньше потребление ресурсов (в сравнении с потоками)

Минусы:

* "тормозит" из-за одного блокирующего вызова (!)
* не безразмерный, нужно понимать, что корутины не бесплатные

важные принципы:

1. корутина работает как генератор
2. async - явный флаг, что данная функция является асинхронной (корутиной)
3. await - явный флаг, что в этом месте функция встает на паузу и дает работать другим, пока ждет свои данные.
4. event loop - цикл событий, механизм, который отвечает за планирование и запуск корутин. Можно представить как список/очередь, из которого в вечном цикле достаются и запускаются корутины

Частые ошибки:

* не использование await внутри корутины
* создание корутины, но использование ее, как функции
* использование внутри корутин синхронного(блокирующего) кода, в том числе IO

import asyncio

import aiohttp

import requests

import time

# использование блокирующих IO задач

async def blocking():

resp = requests.get("<https://yandex.ru>") # пока get невернет данные дальше выполнение не идет

print(resp.status\_code)

# использование НЕ блокирующих IO задач

async def async\_http():

async with aiohttp.ClientSession() as session:

async with session.get("<https://yandex.ru>") as resp:

print(resp.status)

async def one():

print("Start one")

await asyncio.sleep(3)

print("Stop one")

async def two():

print("Start two")

await asyncio.sleep(3)

print("Stop two")

# корутинная функция

async def three():

print("Start three")

await asyncio.sleep(3)

print("Stop three")

async def main():

# await asyncio.gather(one(), two(), three())

# await asyncio.gather(\*(blocking() for \_ in range(5))) # \* - распаковывает результаты

# gather сам собирает задачи в список eventloop (цикл событий) и выполняет их, самостоятельно следя за переключением и выполнением между функциями

asyncio.create\_task(one()) # создаем задачи, добавляем их в список eventloop под капотом

asyncio.create\_task(two())

await asyncio.create\_task(three())

# await значит что программа не продолжит работать пока не выполнится функция после await