**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«СЕВЕРО-ВОСТОЧНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ ИМЕНИ М.К. АММОСОВА»  
ИНСТИТУТ МАТЕМАТИКИ И ИНФОРМАТИКИ  
НАУЧНО-ИССЛЕДОВАТЕЛЬСКАЯ КАФЕДРА «ВЫЧИСЛИТЕЛЬНЫЕ ТЕХНОЛОГИИ»**

ОТЧЕТ

ПО УЧЕБНОЙ ТЕХНОЛОГИЧЕСКОЙ ПРАКТИКЕ

Направление подготовки: 01.03.02 Прикладная математика и информатика

Направленность программы: Искусственный интеллект и анализ данных

Студента группы Б-ИИПМИ-23 Габышев Давид Юрьевич / \_\_\_\_\_\_\_(подпись)

(группа, Ф.И.О / подпись)

Руководитель практики: доцент, к.т.н. Акимов М.П.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_/\_\_\_\_\_\_\_\_\_\_\_\_

(оценка, подпись)

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

Якутск 2025

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc209984178)

[ЗАДАЧИ ПРОЕКТА 4](#_Toc209984179)

[1. Подготовка среды разработки 4](#_Toc209984180)

[2. Изучение и закрепление базовых технологий 4](#_Toc209984181)

[3. Разработка CLI-анализатора логов 4](#_Toc209984182)

[4. Разработка REST API-сервиса 4](#_Toc209984183)

[5. Тестирование и отладка 4](#_Toc209984184)

[6. Документирование и оформление проекта 4](#_Toc209984185)

[ОСНОВНАЯ ЧАСТЬ 5](#_Toc209984186)

[Стек технологий и среда 5](#_Toc209984187)

[Ход разработки 6](#_Toc209984188)

[Архитектура и структура проекта 7](#_Toc209984189)

[Разбор кода по модулям и функциям 8](#_Toc209984190)

[Алгоритмы и оптимизация 11](#_Toc209984191)

[Надёжность и безопасность 12](#_Toc209984192)

[Тестирование 13](#_Toc209984193)

[Интерфейсы 13](#_Toc209984194)

[Запуск и конфигурация 13](#_Toc209984195)

[Для запуска проекта необходимо: 13](#_Toc209984196)

[ЗАКЛЮЧЕНИЕ 14](#_Toc209984197)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 14](#_Toc209984198)

[ПРИЛОЖЕНИЯ 15](#_Toc209984199)

[Приложены исходные коды: 17](#_Toc209984200)

[ССЫЛКА НА GitHub: 17](#_Toc209984201)

[log\_analyzer.py: 17](#_Toc209984202)

[app.py: 21](#_Toc209984203)

[requirements.txt 23](#_Toc209984204)

# ВВЕДЕНИЕ

Учебная практика позволяет студенту закрепить знания и применить их на практике, выполняя задачи, близкие к реальным условиям профессиональной деятельности. В рамках направления «Искусственный интеллект и анализ данных» выбранной темой стала разработка обработчика логов с возможностью работы как через командную строку, так и через REST API. Такой выбор обусловлен актуальностью анализа логов для мониторинга работы систем и поиска ошибок.

Цель практики заключалась в создании программы, которая способна обрабатывать лог-файлы двух форматов — Apache-подобных и строк с временными метками в ISO-формате. По результатам анализа программа должна формировать отчёт с распределением событий по уровням, IP-адресам и временным промежуткам, а также выявлять наиболее «медленные» запросы. Дополнительно требовалось разработать веб-сервис на Flask, предоставляющий доступ к функциям анализатора через API.

Задачи включали настройку среды разработки (Python, Git, Postman), реализацию основной логики анализа, разработку веб-интерфейса, тестирование через Postman и подготовку документации. В проект не входили задачи по визуализации данных и развёртыванию в облаке — это позволило сосредоточиться на корректной обработке логов.

Критериями успешности являлись устойчивое выполнение анализа тестового файла (403 строки), формирование корректного отчёта, работа API через эндпоинты /health и /analyze, а также наличие вспомогательных материалов: README, коллекции Postman и данного отчёта.

## ЗАДАЧИ ПРОЕКТА

### Подготовка среды разработки

* + установка интерпретатора Python версии 3.11;
  + настройка виртуального окружения и установка зависимостей через pip;
  + установка и настройка инструментов для тестирования (Postman) и контроля версий (Git, GitHub).

### Изучение и закрепление базовых технологий

* + повторение работы с функциями, структурами данных и классами в Python;
  + изучение приёмов работы с файлами, регулярными выражениями и временными метками;
  + освоение принципов построения REST API во Flask.

### Разработка CLI-анализатора логов

* + написание регулярных выражений для двух форматов строк (Apache и ISO);
  + реализация функции нормализации времени (parse\_ts);
  + создание функции анализа (analyze), выполняющей однопроходный разбор файла и формирующей отчёт;
  + поддержка параметра --top для вывода наиболее частых значений.

### Разработка REST API-сервиса

* + создание веб-приложения во Flask (app.py);
  + реализация эндпоинтов /health и /analyze;
  + поддержка трёх форматов входа: загрузка файла, передача «сырого» текста, JSON с полем log;
  + возврат результата анализа в формате JSON.

### Тестирование и отладка

* + подготовка тестовой коллекции Postman;
  + проверка API в режимах работы с файлом, текстом и JSON;
  + тестирование CLI-запуска и проверка устойчивости к ошибочным данным.

### Документирование и оформление проекта

* + подготовка README с инструкциями по установке и запуску;
  + оформление отчёта по практике с описанием архитектуры и функций;

# ОСНОВНАЯ ЧАСТЬ

## Стек технологий и среда

Практика выполнялась в операционной системе Windows 11. Для разработки использовалась среда Visual Studio Code, которая обеспечила удобную работу с Python и интеграцию с системой контроля версий Git. Хранение исходного кода и фиксация изменений велись в репозитории GitHub, что позволило отработать навык ведения проектов с использованием системы версионирования.

В качестве языка программирования был выбран Python версии 3.11. Он предоставляет широкие возможности для анализа текста, обладает богатой стандартной библиотекой и простым синтаксисом. Для разбора логов применялись регулярные выражения (re), для работы со временем — модуль datetime, а для подсчётов — класс Counter из библиотеки collections. Все эти инструменты входят в стандартный набор Python и не требуют дополнительных зависимостей.

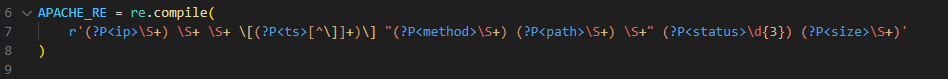
Для веб-интерфейса использовался микрофреймворк Flask (версия 3.0 и выше). Он позволил быстро реализовать маршруты API, обрабатывающие запросы и возвращающие JSON-ответы. Безопасная работа с загружаемыми файлами обеспечена за счёт функции secure\_filename из Werkzeug. Для создания временных файлов применялись стандартные модули tempfile и pathlib.

Для тестирования API использовалась программа Postman. В проект была включена коллекция с готовыми запросами, что позволило проверить работу сервиса во всех режимах: передача логов файлом, «сырым» текстом и через JSON. Такой стек был выбран как наиболее лёгкий и достаточный для учебного проекта. Более сложные решения (например, Django или асинхронный aiohttp) рассматривались, но были отклонены как избыточные для поставленной задачи.

## Ход разработки

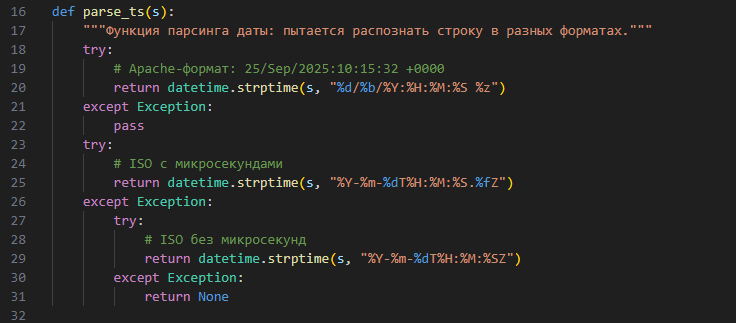
Разработка началась с постановки задачи: необходимо было создать обработчик логов, поддерживающий два формата строк — Apache-подобный и строки с временными метками ISO. На первом этапе были определены основные метрики: распределение по уровням (INFO, WARN, ERROR и др.), количество запросов от IP-адресов, частота сообщений, ошибки по часам и список «медленных» строк с большим временем выполнения.

Затем был создан прототип CLI-анализатора. В модуле log\_analyzer.py определены регулярные выражения для выделения полей в строках. Например, шаблон для Apache-лога:



Этот шаблон позволяет извлекать IP-адрес, временную метку, метод, путь и код состояния. Для ISO-строк был добавлен отдельный шаблон SIMPLE\_RE.

Следующим этапом стала реализация функции нормализации времени. Так как логи могли содержать разные форматы меток, была создана функция parse\_ts, которая делает несколько попыток разбора:



Функция возвращает объект datetime, а если формат неизвестен — None. Это позволило обрабатывать и корректные, и нестандартные строки без остановки программы.

На следующем этапе был создан веб-сервис во Flask (app.py). В нём реализован маршрут /analyze, который принимает данные тремя способами: через загруженный файл, через текст в формате text/plain и через JSON. Для унификации ввода используется вспомогательная функция analyze\_text\_via\_tempfile, которая записывает переданный текст во временный файл и вызывает функцию анализа. Такой подход сделал архитектуру модулей простой и одинаковой для всех форматов входа.

Финальный этап включал тестирование. С помощью Postman были отправлены запросы на все эндпоинты, что позволило выявить и устранить недочёты. Например, при пустом запросе сервис возвращал неочевидную ошибку, поэтому был добавлен понятный ответ с подсказкой. Также была оптимизирована функция analyze, чтобы она работала в один проход и не перегружала систему при больших файлах.

## Архитектура и структура проекта

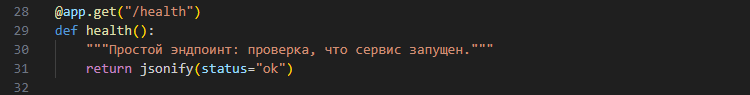
Архитектура проекта построена по принципу разделения обязанностей. Логика анализа логов вынесена в модуль log\_analyzer.py, а веб-сервис реализован в app.py. Такой подход позволил использовать один и тот же анализатор как в командной строке, так и через REST API.

Поток данных выглядит так: пользователь отправляет запрос на /analyze, сервис принимает файл или текст, при необходимости сохраняет данные во временный файл, затем вызывает функцию analyze(path, topn) из модуля log\_analyzer.py. Результат анализа формируется в виде многострочного отчёта и возвращается клиенту в JSON-ответе.

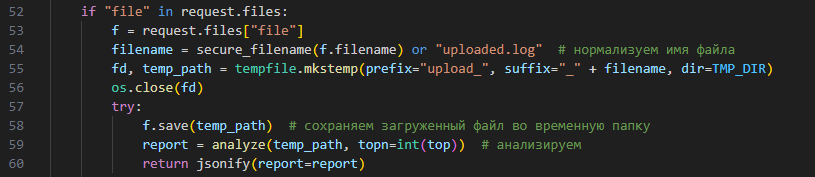
Структура проекта включает:

* log\_analyzer.py — модуль анализа логов с функциями parse\_ts и analyze.
* app.py — веб-сервис на Flask с маршрутами /health и /analyze.
* requirements.txt — список зависимостей.
* README.md — инструкция по запуску и использованию.
* Log Analyzer API.postman\_collection.json — коллекция запросов для тестирования.
* sample\_log.txt — тестовый файл логов.

Пример простого маршрута проверки состояния сервиса:



Для загрузки файлов в маршруте /analyze используется безопасная работа с временными файлами:

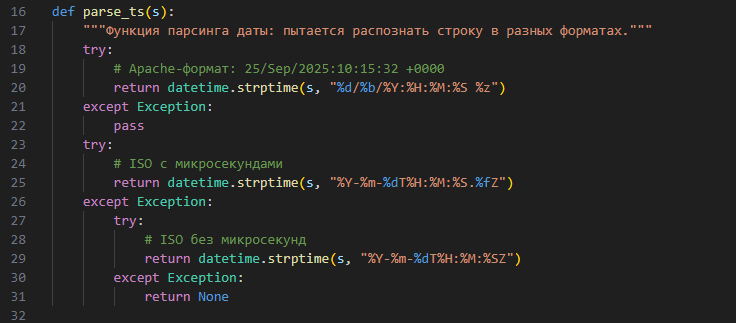


Таким образом, архитектура сервиса остаётся простой и понятной: один модуль отвечает за обработку данных, другой — за взаимодействие с пользователем через HTTP.

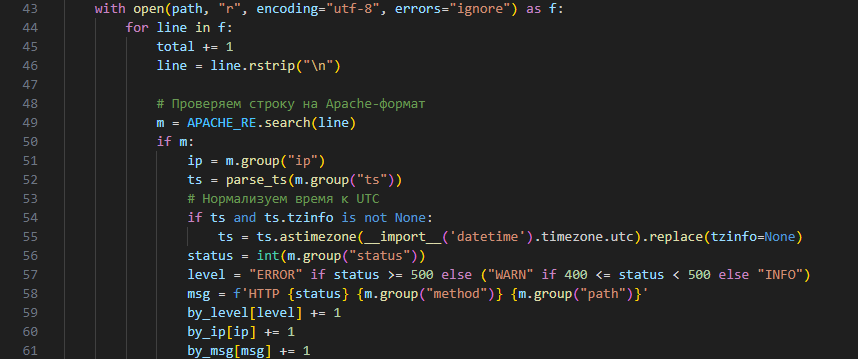
## Разбор кода по модулям и функциям

Основная логика сосредоточена в log\_analyzer.py. Здесь определены регулярные выражения для двух форматов логов, функция нормализации времени и функция анализа.

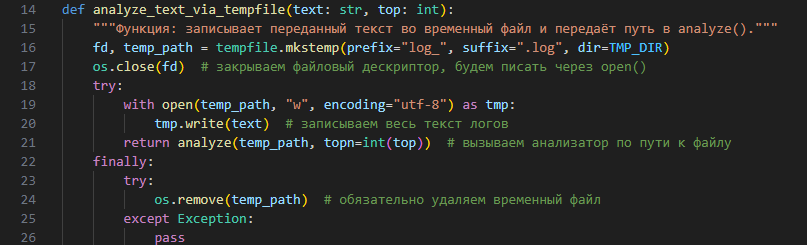
Функция parse\_ts(s) преобразует строковую метку времени в объект datetime. Она делает три попытки разбора: Apache-формат, ISO с миллисекундами и ISO без них. Если формат неизвестен, возвращается None. Такой подход обеспечивает устойчивость к разным видам строк.



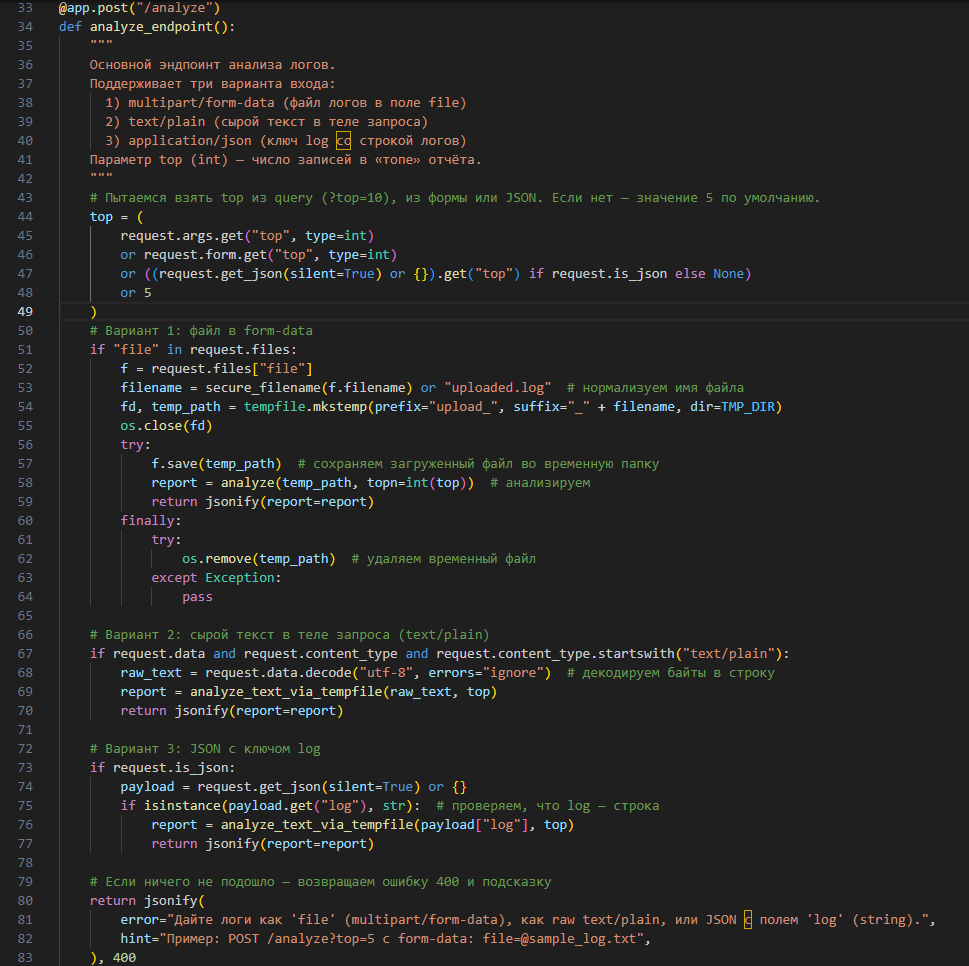
Функция analyze(path, topn=5) выполняет основной анализ. Она открывает файл, читает его построчно, сопоставляет каждую строку с шаблонами и обновляет счётчики Counter. В конце формируется отчёт, включающий статистику по уровням, IP-адресам, сообщениям, ошибкам по часам и «медленным» строкам.



Функция analyze\_text\_via\_tempfile(text, top) используется в веб-сервисе для унификации ввода. Она записывает текст во временный файл и вызывает функцию analyze.



Функция analyze\_endpoint() реализует маршрут /analyze. Она обрабатывает все три вида входа — файл, текст и JSON — и возвращает JSON-ответ с отчётом.

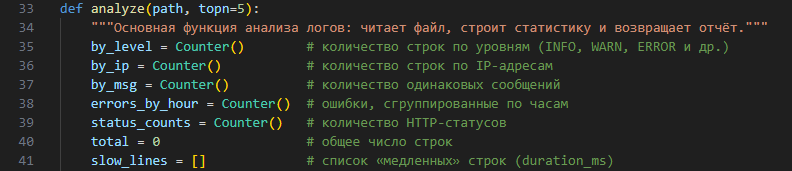


Таким образом, каждый модуль и функция занимают чёткое место в архитектуре: log\_analyzer.py отвечает за обработку, а app.py — за взаимодействие с пользователем.

## Алгоритмы и оптимизация

Алгоритм анализа строится на принципе однопроходной обработки: файл читается построчно, и каждая строка сразу классифицируется и учитывается в агрегатах. Такой подход исключает повторные обходы данных и обеспечивает линейную сложность O(n).

Для хранения статистики использованы счётчики Counter, которые позволяют за константное время увеличивать значения и быстро получать наиболее частые элементы.



Важным шагом стало выделение «медленных» строк по параметру duration\_ms. Они сортируются по длительности, и в отчёт попадают только верхние значения в соответствии с параметром top. Это позволяет контролировать объём отчёта и исключает избыточный вывод.

Тестирование показало, что обработка файла из 403 строк занимает меньше 0,1 секунды. Алгоритм стабильно работает и при увеличении объёма данных, так как его время выполнения зависит только от количества строк.

## Надёжность и безопасность

В проекте реализованы механизмы защиты от некорректного ввода и ошибок. Веб-сервис проверяет формат входных данных и возвращает понятное сообщение, если формат не поддерживается.

Файлы обрабатываются безопасно: имена нормализуются функцией secure\_filename, временные файлы создаются в системной директории и удаляются в блоке finally. Это исключает риск случайной записи в недопустимые пути и засорения диска.

При чтении файла используется параметр errors="ignore", что позволяет корректно обрабатывать строки с повреждённой кодировкой. Если строка не соответствует шаблону, она учитывается как «OTHER», а программа продолжает работу.



Таким образом, проект устойчив к ошибкам во входных данных и безопасен при работе с файлами.

## Тестирование

Тестирование выполнялось с использованием Postman. Была создана коллекция, включающая четыре запроса:

1. Проверка работы сервиса (GET /health).
2. Анализ логов через файл (POST /analyze с файлом).
3. Анализ логов через текст (POST /analyze с text/plain).
4. Анализ логов через JSON (POST /analyze с полем log).

Эти тесты позволили проверить все режимы работы API. Дополнительно использовались команды curl, приведённые в README. Тестирование подтвердило корректность работы всех функций и устойчивость сервиса к ошибкам.

## Интерфейсы

CLI-интерфейс позволяет запускать анализатор напрямую:

curl -X POST "http://localhost:8000/analyze?top=5" -F "file=@sample\_log.txt"

curl -X POST http://localhost:8000/analyze -H "Content-Type: application/json" -d '{"top":5,"log":"..."}'

## Запуск и конфигурация

### Для запуска проекта необходимо:

1. Создать виртуальное окружение: 
2. Установить зависимости: 
3. Запустить сервер: 

Сервис доступен по адресу http://localhost:8000. Параметр top задаёт глубину отчёта, по умолчанию равен 5.

# ЗАКЛЮЧЕНИЕ

В ходе практики был разработан анализатор логов, способный обрабатывать строки двух форматов и формировать сводный отчёт. Реализован веб-сервис на Flask, обеспечивающий доступ к анализатору через REST API.

В процессе работы были закреплены навыки:

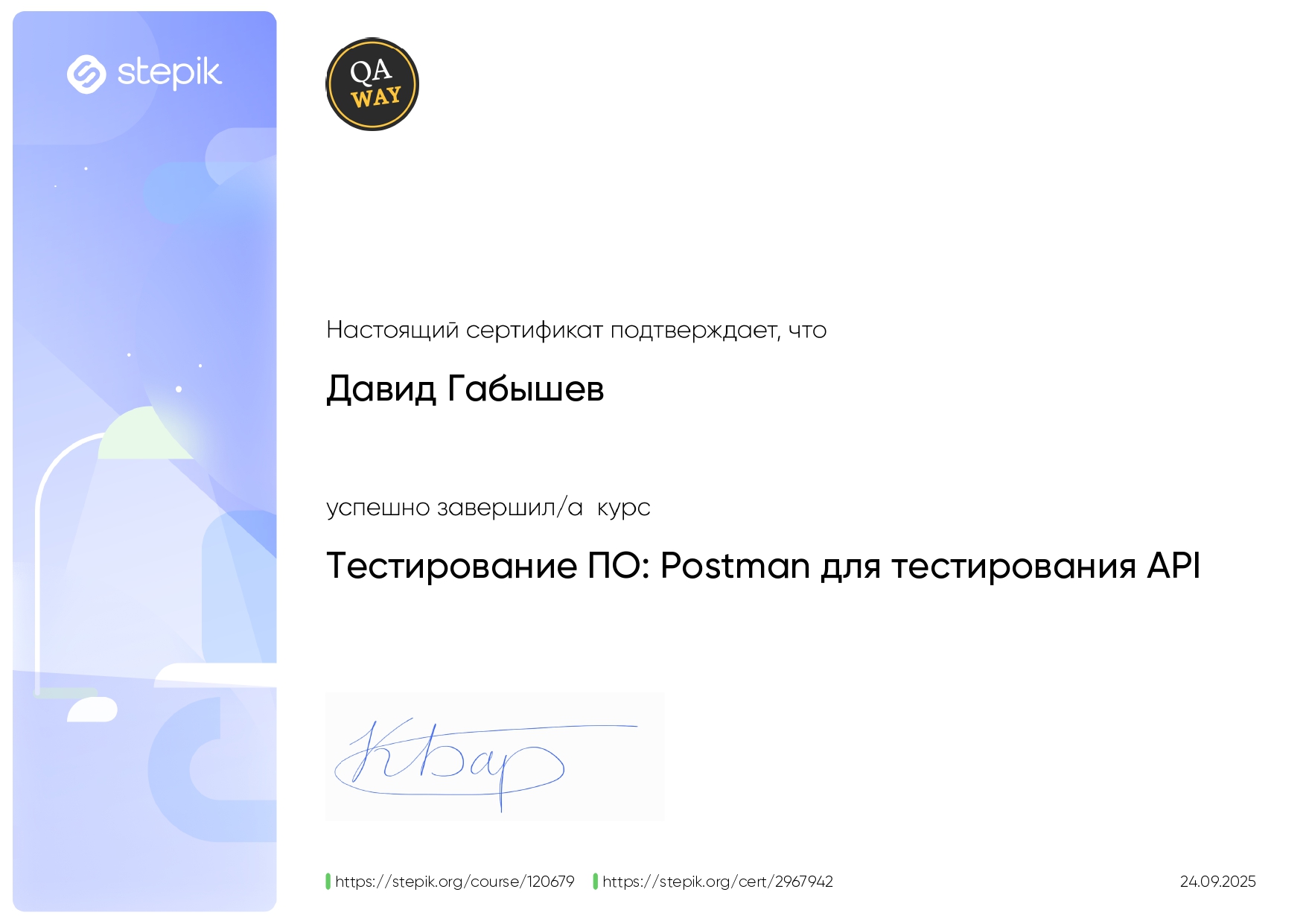
* использование регулярных выражений для разбора текста;
* работа с датами и временем в Python;
* создание REST API во Flask;
* тестирование запросов через Postman;
* работа с Git и GitHub.

Проект подтвердил устойчивость к ошибкам во входных данных и корректность формирования отчётов. В дальнейшем его можно расширить: добавить поддержку JSON-логов, экспорт статистики в CSV, визуализацию отчётов и контейнеризацию.

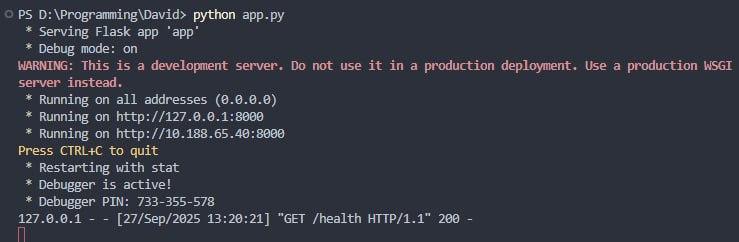
# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация Python (модули re, datetime, collections, argparse).
2. Документация Flask (маршрутизация и работа с JSON).
3. Руководство Postman (создание коллекций и тестирование API, а также курсы в степике).

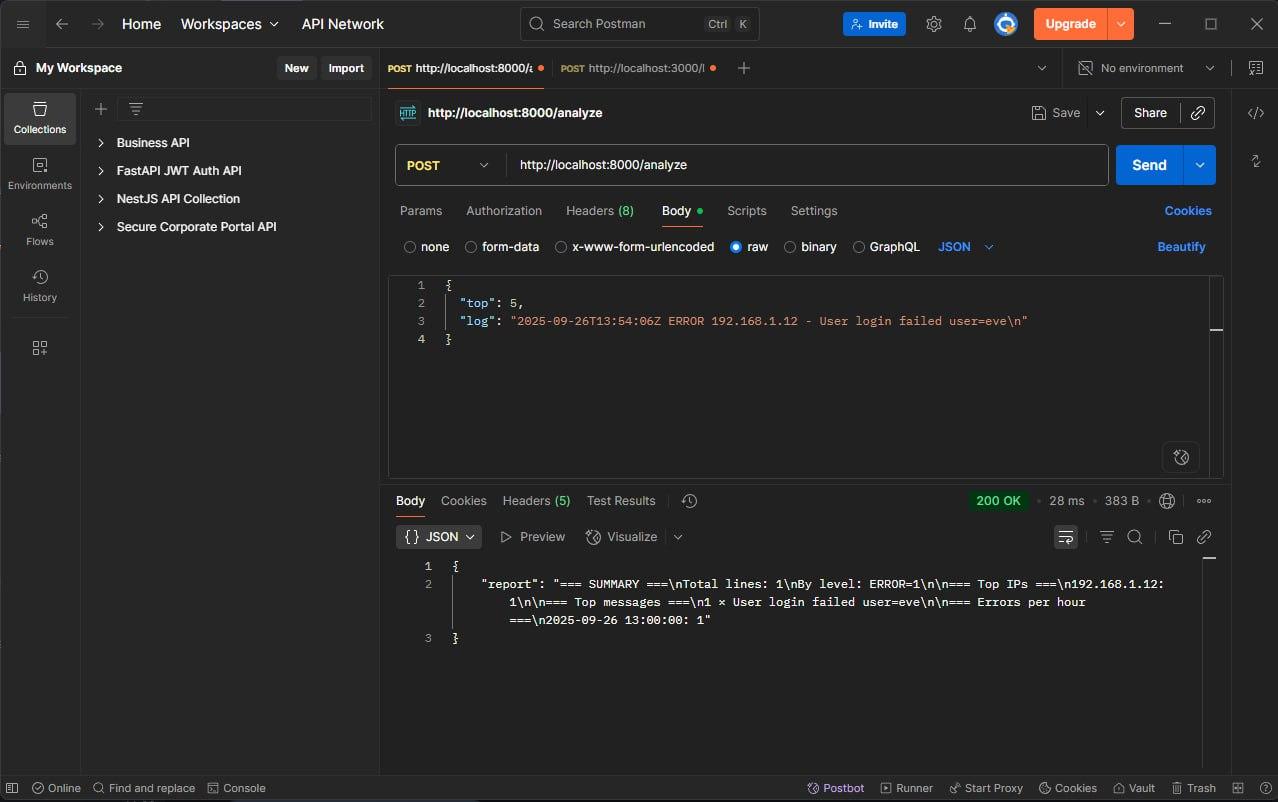
# ПРИЛОЖЕНИЯ



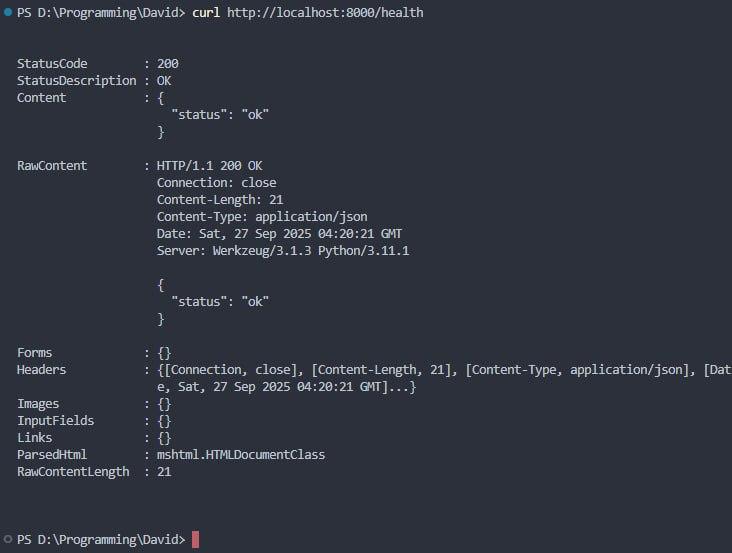
*Рисунок 1. Сертификат курса по Postman*



*Рисунок 2. Запуск сервера (API)*



*Рисунок 3. Отправка и получение запроса*

**

*Рисунок 4. Проверка работоспособности*

## Приложены исходные коды:

### ССЫЛКА НА GitHub:

### log\_analyzer.py:

#!/usr/bin/env python3

import re, sys, argparse # re — регулярные выражения, sys — работа с аргументами, argparse — парсинг CLI

from collections import Counter # Counter — быстрый подсчёт повторов

from datetime import datetime # работа с датами и временем

# Шаблон для строк Apache-лога (IP, дата, метод, путь, статус, размер)

APACHE\_RE = re.compile(

r'(?P<ip>\S+) \S+ \S+ \[(?P<ts>[^\]]+)\] "(?P<method>\S+) (?P<path>\S+) \S+" (?P<status>\d{3}) (?P<size>\S+)'

)

# Шаблон для строк ISO-формата (время, уровень, IP, сообщение)

SIMPLE\_RE = re.compile(

r'(?P<ts>\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(?:\.\d+)?Z) (?P<level>[A-Z]+) (?P<ip>\d+\.\d+\.\d+\.\d+) - (?P<msg>.+)'

)

def parse\_ts(s):

"""Функция парсинга даты: пытается распознать строку в разных форматах."""

try:

# Apache-формат: 25/Sep/2025:10:15:32 +0000

return datetime.strptime(s, "%d/%b/%Y:%H:%M:%S %z")

except Exception:

pass

try:

# ISO с микросекундами

return datetime.strptime(s, "%Y-%m-%dT%H:%M:%S.%fZ")

except Exception:

try:

# ISO без микросекунд

return datetime.strptime(s, "%Y-%m-%dT%H:%M:%SZ")

except Exception:

return None

def analyze(path, topn=5):

"""Основная функция анализа логов: читает файл, строит статистику и возвращает отчёт."""

by\_level = Counter() # количество строк по уровням (INFO, WARN, ERROR и др.)

by\_ip = Counter() # количество строк по IP-адресам

by\_msg = Counter() # количество одинаковых сообщений

errors\_by\_hour = Counter() # ошибки, сгруппированные по часам

status\_counts = Counter() # количество HTTP-статусов

total = 0 # общее число строк

slow\_lines = [] # список «медленных» строк (duration\_ms)

with open(path, "r", encoding="utf-8", errors="ignore") as f:

for line in f:

total += 1

line = line.rstrip("\n")

# Проверяем строку на Apache-формат

m = APACHE\_RE.search(line)

if m:

ip = m.group("ip")

ts = parse\_ts(m.group("ts"))

# Нормализуем время к UTC

if ts and ts.tzinfo is not None:

ts = ts.astimezone(\_\_import\_\_('datetime').timezone.utc).replace(tzinfo=None)

status = int(m.group("status"))

level = "ERROR" if status >= 500 else ("WARN" if 400 <= status < 500 else "INFO")

msg = f'HTTP {status} {m.group("method")} {m.group("path")}'

by\_level[level] += 1

by\_ip[ip] += 1

by\_msg[msg] += 1

status\_counts[status] += 1

if level == "ERROR" and ts:

hour = ts.replace(minute=0, second=0, microsecond=0)

errors\_by\_hour[hour] += 1

continue

# Проверяем строку на ISO-формат

m = SIMPLE\_RE.search(line)

if m:

ip = m.group("ip")

ts = parse\_ts(m.group("ts"))

if ts and ts.tzinfo is not None:

ts = ts.astimezone(\_\_import\_\_('datetime').timezone.utc).replace(tzinfo=None)

level = m.group("level")

msg = m.group("msg")

by\_level[level] += 1

by\_ip[ip] += 1

by\_msg[msg] += 1

# Проверяем наличие duration\_ms

dm = re.search(r'duration\_ms=(\d+)', msg)

if dm:

slow\_lines.append((int(dm.group(1)), line))

if level == "ERROR" and ts:

hour = ts.replace(minute=0, second=0, microsecond=0, tzinfo=None)

errors\_by\_hour[hour] += 1

continue

# Если формат не распознан

by\_level["OTHER"] += 1

# Формируем отчёт

report = []

report.append("=== SUMMARY ===")

report.append(f"Total lines: {total}")

report.append("By level: " + ", ".join(f"{k}={v}" for k,v in by\_level.most\_common()))

if status\_counts:

report.append("HTTP statuses: " + ", ".join(f"{k}={v}" for k,v in status\_counts.most\_common()))

report.append("\n=== Top IPs ===")

for ip, c in by\_ip.most\_common(topn):

report.append(f"{ip}: {c}")

report.append("\n=== Top messages ===")

for msg, c in by\_msg.most\_common(topn):

report.append(f"{c} × {msg}")

report.append("\n=== Errors per hour ===")

for hour, c in sorted(errors\_by\_hour.items()):

report.append(f"{hour}: {c}")

if slow\_lines:

report.append("\n=== Slow lines (by duration\_ms) ===")

for dur, line in sorted(slow\_lines, reverse=True)[:topn]:

report.append(f"{dur} ms | {line}")

return "\n".join(report)

def main():

"""CLI-интерфейс: запуск через консоль с аргументами."""

p = argparse.ArgumentParser(description="Простой анализатор логов")

p.add\_argument("path", help="Путь к файлу логов (txt)")

p.add\_argument("--top", type=int, default=5, help="Число выводимых значений в топах (по умолчанию 5)")

args = p.parse\_args()

out = analyze(args.path, topn=args.top)

print(out)

if \_\_name\_\_ == "\_\_main\_\_":

main()

### app.py:

from flask import Flask, request, jsonify   # подключаем Flask и функции для работы с HTTP-запросами и JSON-ответами

from werkzeug.utils import secure\_filename  # безопасная обработка имён файлов

import os, tempfile                         # работа с файловой системой и временными файлами

from pathlib import Path                    # удобная работа с путями

from log\_analyzer import analyze            # импортируем функцию анализа логов из другого модуля

app = Flask(\_\_name\_\_)                       # создаём экземпляр приложения Flask

# Определяем кросс-платформенную временную папку (поддержка Windows/Linux/Mac)

TMP\_DIR = Path(tempfile.gettempdir())

TMP\_DIR.mkdir(parents=True, exist\_ok=True)  # создаём папку, если её нет

def analyze\_text\_via\_tempfile(text: str, top: int):

    """Функция: записывает переданный текст во временный файл и передаёт путь в analyze()."""

    fd, temp\_path = tempfile.mkstemp(prefix="log\_", suffix=".log", dir=TMP\_DIR)

    os.close(fd)  # закрываем файловый дескриптор, будем писать через open()

    try:

        with open(temp\_path, "w", encoding="utf-8") as tmp:

            tmp.write(text)  # записываем весь текст логов

        return analyze(temp\_path, topn=int(top))  # вызываем анализатор по пути к файлу

    finally:

        try:

            os.remove(temp\_path)  # обязательно удаляем временный файл

        except Exception:

            pass

@app.get("/health")

def health():

    """Простой эндпоинт: проверка, что сервис запущен."""

    return jsonify(status="ok")

@app.post("/analyze")

def analyze\_endpoint():

    """

    Основной эндпоинт анализа логов.

    Поддерживает три варианта входа:

      1) multipart/form-data (файл логов в поле file)

      2) text/plain (сырой текст в теле запроса)

      3) application/json (ключ log со строкой логов)

    Параметр top (int) — число записей в «топе» отчёта.

    """

    # Пытаемся взять top из query (?top=10), из формы или JSON. Если нет — значение 5 по умолчанию.

    top = (

        request.args.get("top", type=int)

        or request.form.get("top", type=int)

        or ((request.get\_json(silent=True) or {}).get("top") if request.is\_json else None)

        or 5

    )

    # Вариант 1: файл в form-data

    if "file" in request.files:

        f = request.files["file"]

        filename = secure\_filename(f.filename) or "uploaded.log"  # нормализуем имя файла

        fd, temp\_path = tempfile.mkstemp(prefix="upload\_", suffix="\_" + filename, dir=TMP\_DIR)

        os.close(fd)

        try:

            f.save(temp\_path)  # сохраняем загруженный файл во временную папку

            report = analyze(temp\_path, topn=int(top))  # анализируем

            return jsonify(report=report)

        finally:

            try:

                os.remove(temp\_path)  # удаляем временный файл

            except Exception:

                pass

    # Вариант 2: сырой текст в теле запроса (text/plain)

    if request.data and request.content\_type and request.content\_type.startswith("text/plain"):

        raw\_text = request.data.decode("utf-8", errors="ignore")  # декодируем байты в строку

        report = analyze\_text\_via\_tempfile(raw\_text, top)

        return jsonify(report=report)

    # Вариант 3: JSON с ключом log

    if request.is\_json:

        payload = request.get\_json(silent=True) or {}

        if isinstance(payload.get("log"), str):  # проверяем, что log — строка

            report = analyze\_text\_via\_tempfile(payload["log"], top)

            return jsonify(report=report)

    # Если ничего не подошло — возвращаем ошибку 400 и подсказку

    return jsonify(

        error="Дайте логи как 'file' (multipart/form-data), как raw text/plain, или JSON с полем 'log' (string).",

        hint="Пример: POST /analyze?top=5 c form-data: file=@sample\_log.txt",

    ), 400

if \_\_name\_\_ == "\_\_main\_\_":

    # Запуск приложения Flask на локальном сервере

    app.run(host="0.0.0.0", port=8000, debug=True)

## requirements.txt

Flask>=3.0.0