Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное

образовательное учреждение высшего образования

«Уфимский университет науки и технологий»

Институт информатики, математики и робототехники

Кафедра вычислительной математики и кибернетики

Отчет по лабораторной работе №5

по дисциплине «Архитектура вычислительных систем»

на тему «Работа с протоколами прикладного уровня. Реализация HTTP веб-сервера»

Выполнил: ст. гр.

ПРО-331Б

Семенов Л.А.

Проверил:

Юдинцев Б.С.

Уфа 2024

1. **Цель работы**

Получить основные теоретические сведения по работе с протоколами прикладного уровня, подходом AJAX.

# 2. Задание на лабораторной работе

1. Ознакомиться с теоретической информацией по реализации подхода AJAX.
2. Изучить код html-страницы, использующей метод AJAX (см. папку с кодом веб-сервера: www/webclient.html). Проверить работу всех HTTP-методов.
3. Расширить функциональные возможности веб-сервера из лабораторной работы №4 (см. файлы в архиве lab5\_Python\_ПРИМЕР.zip). При обращении по адресу <адрес\_веб\_сервера>/posted реализовать обработку всех HTTP-методов, которые использует страница webclient.html:
   1. GET: сформировать ответ с содержимым файла, указанным в поле «Имя файла», если файла не существует сформировать json-ответ с HTTP-кодом ошибки.
   2. POST: запись данных из поля «Запрос клиента» в файл с уникальным именем (должно быть реализовано на ЛР№4) и сформировать json-ответ с полями, содержащими код успешного выполнения операции и имя созданного файла.
   3. PUT: заменить файл по имени, указанному в поле «Имя файла». При успешном выполнении сформировать json-ответ с HTTP-кодом 201, если файла не существует сформировать json-ответ с HTTPкодом ошибки.
   4. DELETE: удалить файл по имени, указанному в поле «Имя файла». При успешном выполнении сформировать json-ответ с HTTPкодом 200, если файла не существует сформировать json-ответ с HTTP-кодом ошибки.
4. Удалить из javascript-кода страницы инструкцию preventDefault(), проверить выполнение HTTP-операций со страницы. Сделать вывод о том, что поменялось в работе страницы и почему.
5. Запустить веб-сервер на ВМ. Приложить к отчету скриншоты страницы webclient.html с результатами выполнения каждой операции.
6. Приложить листинг модуля contrthread

# 3. Ход выполнения задания

1. Были изучена теоретическая информация по реализации AJAX.

2. Был изучен код html-страницы, использующий метод AJAX. Проверена работа всех методов. Работают только push и get

3. Были расширены функциональные возможности веб-сервера из лабораторной работы №4

4. При удалении из javascript-кода страницы инструкцию preventDefault(), запросы выполняются, но ответ в окне вывода ответа не появляется, так как страница обновляется.

5. Скриншоты работы программы.

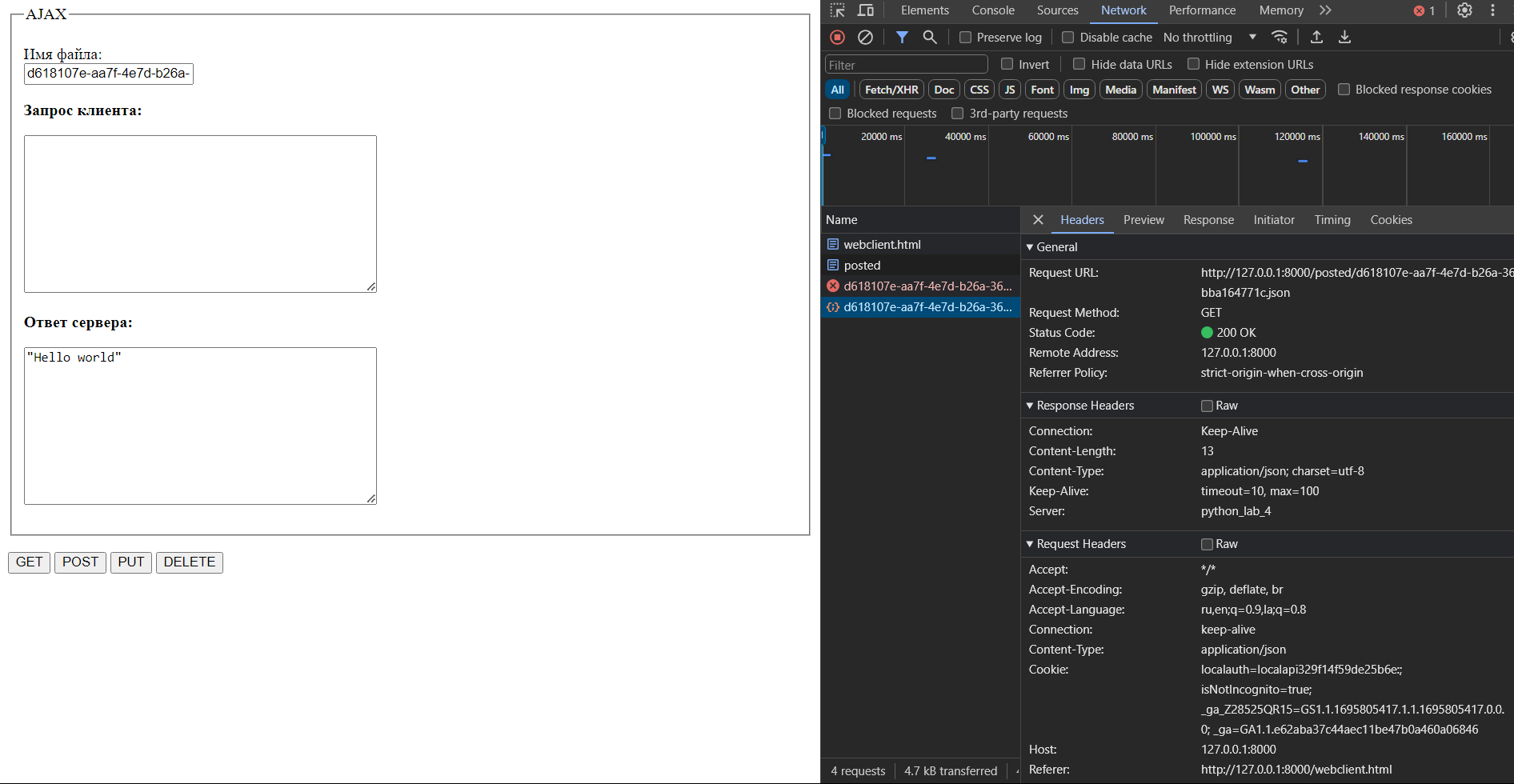


Рисунок 1 - Метод GET

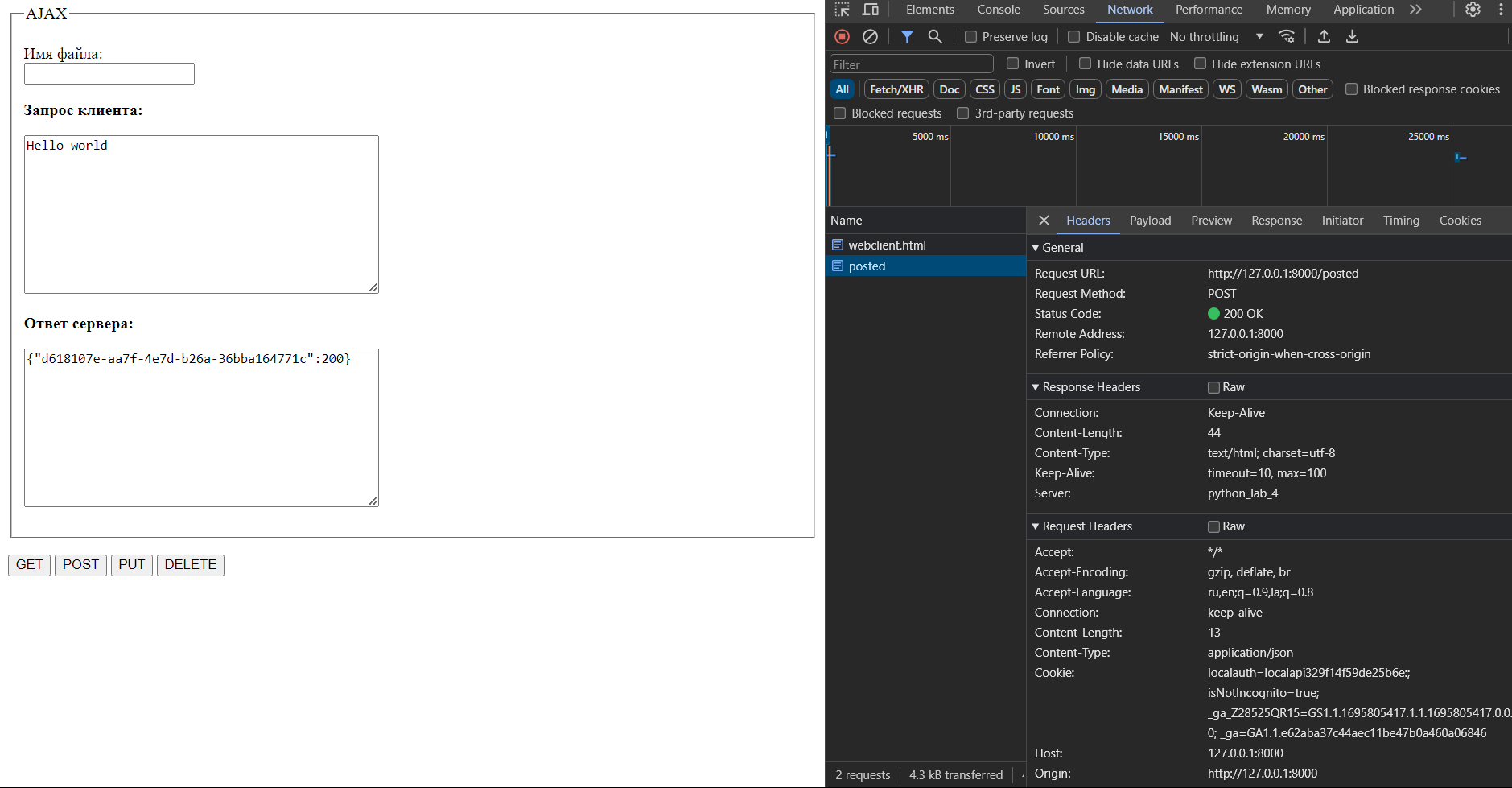


Рисунок 2 - Метод POST

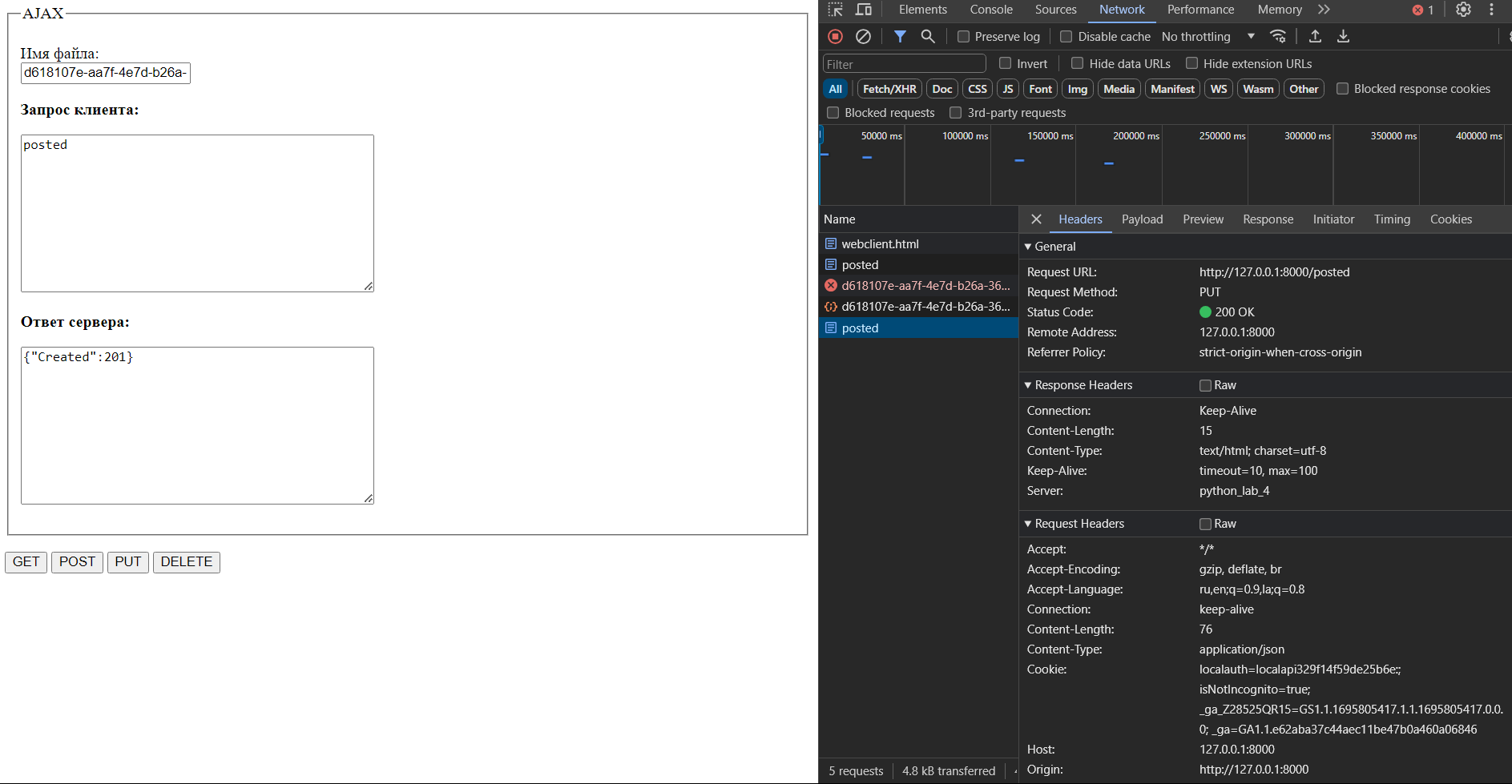


Рисунок 3 - Метод PUT

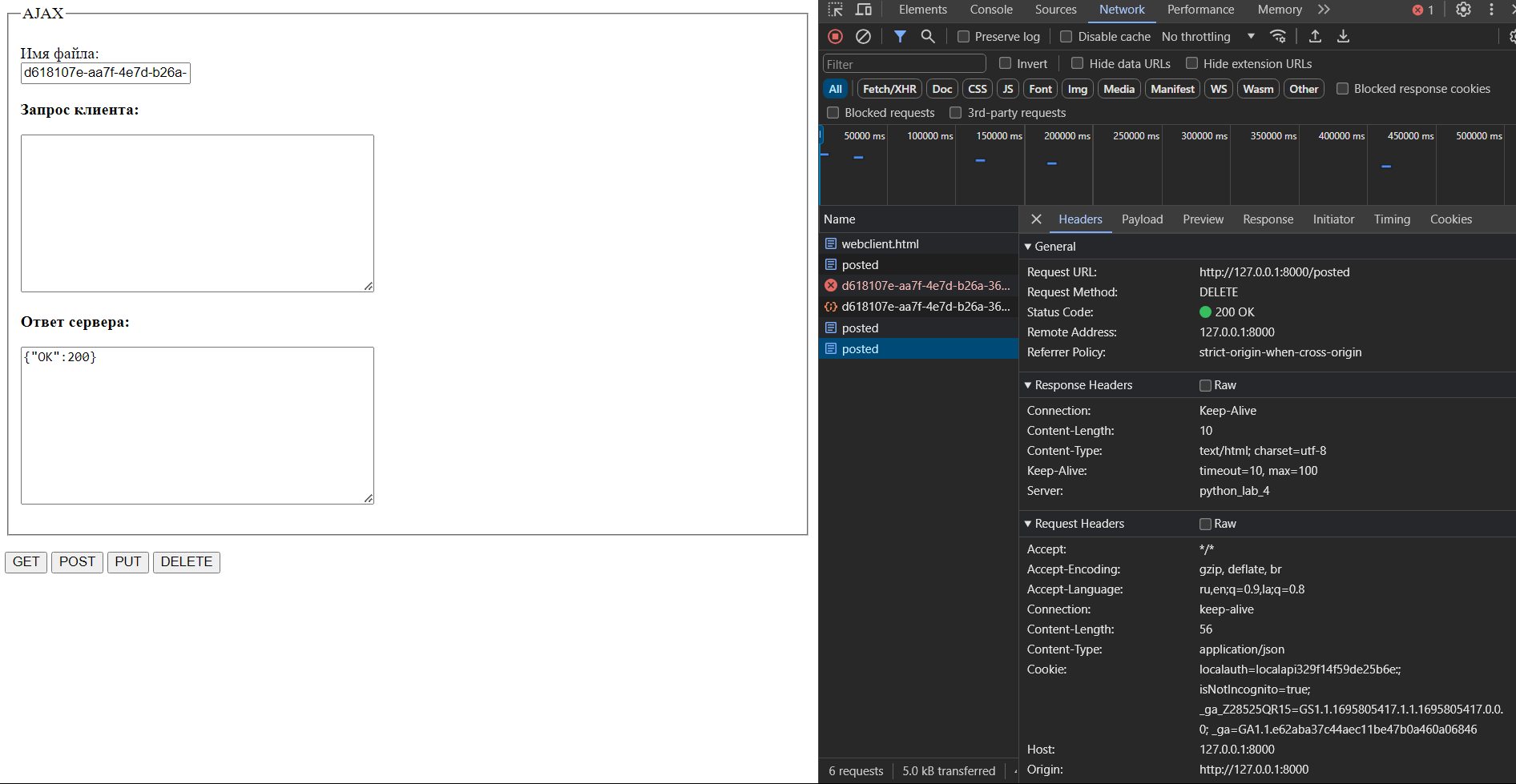


Рисунок 4 - Метод DELETE

Выводы

Получены основные теоретические сведения по работе с протоколами прикладного уровня, подходом AJAX.

Приложение 1

6. Листинг модуля connthread.py

import threading

from threading import Thread

import uuid

import json

import sys

import os

import time

ROOT\_DIR = "www" # Корневой каталог веб-сервера для чтения веб-страниц

POST\_DIR = "posted" # Папка для записи переданных с помощью метода POST данных

# Максимальный размер данных, передаваемый веб-серверу в запросе (в байтах)

POST\_MAX\_SIZE = 1024 \* 1024 \* 2 # 2Mb

KEEP\_ALIVE = 10 # Таймаут HTTP соединения (TCP сессии между веб-сервером и клиентом) в сек.

# Список доступных кодов HTTP

OK = "200 OK"

NOT\_FOUND = "404 Not Found"

BAD\_REQUEST = "400 Bad Request"

SERVER\_ERROR = "500 Internal Server Error"

# Поддерживаемые веб-сервером типы файлов и соответствующие расширения

MIME\_TYPES = [

"text/html",

"application/json",

"application/xml",

]

# Наследуем класс от базового класса Thread

class HTTPConnectionThread(Thread):

# В конструкторе класса определяем внутренние атрибуты

def \_\_init\_\_(self, conn, addr):

# вызываем конструктор родительского класса

Thread.\_\_init\_\_(self)

self.connection = conn

self.address = addr

# Определяем полный путь до папки с приложением

self.\_root\_path = os.path.join(os.path.abspath(os.getcwd()), ROOT\_DIR)

# Переопределяем метод run родительского класса Thread

def run(self):

# запускаем ожидание команд от клиента в бесконечном цикле

request = ""

while request is not None:

try:

request = self.connection.recv(POST\_MAX\_SIZE)

data = request.decode("utf-8", errors='ignore') # декодируем полученные байты в строку

# Если браузер прислал пустую строку, значит истекло время TCP-сессии, заданное в заголовке "Keep-alive"

if data == '':

print("Keep-alive timeout.")

break

response = self.read\_client\_request(data)

self.connection.send(response.encode("utf-8"))

except Exception as error:

print("HTTP Connection error:\n", error)

break

self.connection.close() # закрываем соединение

"""

Создание HTTP-заголовка ответа:

HTTP/1.1 200 OK - версия HTTP и статус-код

Server: имя сервера

Content-Type: тип возвращаемого контента (text/html, text/xml, application/json и т.д.)

Content-Length: размер возвращаемых данных в байтах

Keep-Alive: параметры TCP-сессии (timeout - таймаут в сек., max - максимально допустимое количество запросов в рамках сессии)

Connection: Keep-Alive - заголовок означает, что сервер не будет закрывать сессию сразу после получения запроса и отправки ответа

"""

def make\_http\_header(self, code, length, content\_type='text/html'):

header = f'HTTP/1.1 {code}\r\n' \

f'Server: python\_lab\_4\r\n' \

f'Content-Type: {content\_type}; charset=utf-8\r\n' \

f'Content-Length: {length}\r\n' \

f'Keep-Alive: timeout={KEEP\_ALIVE}, max=100\r\n' \

f'Connection: Keep-Alive\r\n\r\n'

return header

# Чтение запрашиваемой контентной страницы

def get\_content\_page(self, path):

full\_path = self.\_root\_path + path

# если сервер работает на Windows ОС, меняем формат пути

if sys.platform.startswith('win'):

full\_path = full\_path.replace("/", "\\")

# если сраница по заданному пути существует

if os.path.isfile(full\_path):

# читаем страницу и возвращаем код 200

file\_type = full\_path.split(".")[1]

for x in MIME\_TYPES:

if x.find(file\_type) != -1:

file\_type = x

break

with open(full\_path, "r", encoding="utf-8") as page:

content = page.read()

# Возвращаем text/html в браузер

return self.make\_http\_header(OK, len(content.encode("utf-8")), file\_type) + content

else:

return self.get\_json\_code(404)

# Парсинг запроса

def read\_client\_request(self, req):

# Разделяем блок заголовков и контент

req\_data = req.split('\r\n\r\n')

# Разбиваем заголовки по отдельным строкам

head\_data = req\_data[0].split('\r\n')

# Если массив не пуст, парсим первую строку

if len(head\_data) > 0:

main\_header\_data = head\_data[0].split(' ')

req\_type = main\_header\_data[0] # HTTP метод

req\_path = main\_header\_data[1] # Путь к запрашиваемому ресурсу

# если GET-запрос

if req\_type == 'GET':

# если запрос к корневой директории сервера

if req\_path == "/":

req\_path = "/index.html" # назначаем контент по умолчанию

return self.get\_content\_page(req\_path) # читаем контентную страницу

elif req\_type == 'POST':

return self.use\_post(req\_data)

elif req\_type == 'PUT':

return self.use\_put(req\_data)

elif req\_type == 'DELETE':

return self.use\_delete(req\_data)

else:

return self.get\_json\_code(400)

def use\_put(self, req\_data):

head\_data = req\_data[1].split(',')

main\_header\_data = head\_data[0].split('\"')

path = "/posted/" + main\_header\_data[3] # Путь к запрашиваемому ресурсу

full\_path = self.\_root\_path + path

# если сервер работает на Windows ОС, меняем формат пути

if sys.platform.startswith('win'):

full\_path = full\_path.replace("/", "\\")

# если сраница по заданному пути существует

if os.path.isfile(full\_path):

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

page.write(req\_data[1])

return self.get\_json\_code(201)

# return self.make\_http\_header(OK, len(content.encode("utf-8")), file\_type) + content

else:

return self.get\_json\_code(404)

def use\_post(self, req\_data):

for elem in MIME\_TYPES:

if req\_data[0].find(elem) != -1:

uid = str(uuid.uuid4())

path = self.\_root\_path + '/' + POST\_DIR + '/' + uid +'.'+elem.split('/')[1]

if sys.platform.startswith('win'):

path=path.replace("/", "\\")

folder\_path=os.path.dirname(path)

if not os.path.exists(folder\_path):

return self.get\_json\_code(500)

with open(path, "w", encoding="utf-8") as page:

page.write(req\_data[1])

break

return self.get\_json\_code(uid)

def use\_delete(self, req\_data):

head\_data = req\_data[1].split('"')

path = "/posted/" + head\_data[3] # Путь к запрашиваемому ресурсу

full\_path = self.\_root\_path + path

# если сервер работает на Windows ОС, меняем формат пути

if sys.platform.startswith('win'):

full\_path = full\_path.replace("/", "\\")

# если сраница по заданному пути существует

if os.path.isfile(full\_path):

os.remove(full\_path)

return self.get\_json\_code(200)

else:

return self.get\_json\_code(404)

def get\_json\_code(self, selector):

if selector == 400:

content = "{\"Bad Request\":400}"

error = BAD\_REQUEST

elif selector == 404:

content = "{\"Not Found\":404}"

error = NOT\_FOUND

elif selector == 500:

content = "{\"Server error\":500}"

error = SERVER\_ERROR

elif selector == 200:

content = "{\"OK\":200}"

error = OK

elif selector == 201:

content = "{\"Created\":201}"

error = OK

else:

content = "{\"" + selector + "\":200}"

error = OK

return self.make\_http\_header(error, len(content.encode("utf-8"))) + content