ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Факультет инфокоммуникационных технологий

Дисциплина:

«Веб-разработка»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

«Работа с сокетами»

Выполнил:

студент группы K33391
Микитчак Иван Михайлович

_____
(подпись)

Проверил:

Говоров Антон Игоревич

_____
(отметка о выполнении)

_____
(подпись)

Санкт-Петербург

2023 г.

Цель: овладеть практическими навыками и умениями реализации web-серверов и использования сокетов.

Ход работы:

Практическое задание 1

Листинги:

Файл “server.py”

```
import socket

host = 'localhost'
port = 5555

# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind the socket to the host and port
sock.bind((host, port))

# Receive the message from the client
message, addr = sock.recvfrom(1024)
print(message.decode())

# Send a response message to the client
response_message = "Hello, client!"
sock.sendto(response_message.encode(), addr)
```

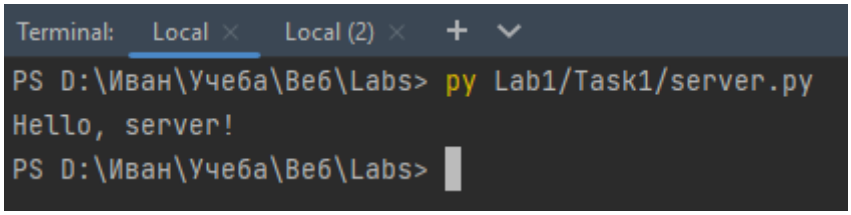
Файл “client.py”

```
import socket

host = 'localhost'
port = 5555

# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Send a message to the server
message = "Hello, server!"
sock.sendto(message.encode(), (host, port))

# Receive a response from the server
```

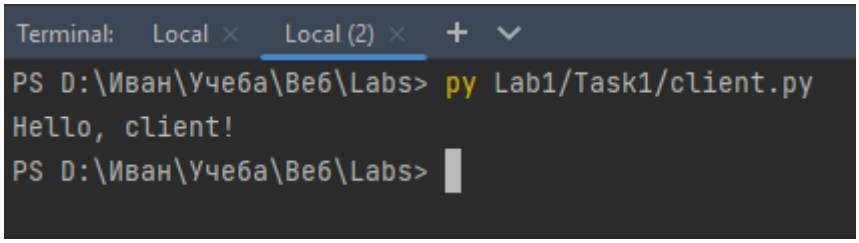
```
response, _ = sock.recvfrom(1024)
print(response.decode())
```

Работа программ:

Процесс “server.py”



Процесс “client.py”



Практическое задание 2 (вариант b)

Листинги:

Файл “server.py”

```
import math
import socket
import json

def tcp_server(host, port):
    # Create a server socket and put it into listening mode
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen(1)

    while True:
        # Accept a client connection
        client_socket, addr = server_socket.accept()

        # Receive a message from the client
        data = client_socket.recv(1024).decode()

        # Parse the message
        # The message is a json object containing a list
        # of three parameters of the quadratic equation
        parameters = json.loads(data)
        a = parameters[0]
        b = parameters[1]
        c = parameters[2]

        # Solve quadratic equation
        solutions = solve_equation(a, b, c)

        # Convert a list of solutions into a json object
        json_string = json.dumps(solutions)

        # Send json object to the client
        client_socket.send(json_string.encode())
```

```
        # Close the connection
        client_socket.close()


def solve_equation(a, b, c):
    D = b ** 2 - 4 * a * c
    if (D > 0):
        x1 = (-b + math.sqrt(D)) / (2 * a)
        x2 = (-b - math.sqrt(D)) / (2 * a)
        return (x1, x2)
    elif (D == 0):
        x = -b / (2 * a)
        return (x,)
    else:
        return ()


if __name__ == "__main__":
    host = 'localhost'
    port = 5555
    tcp_server(host, port)
```

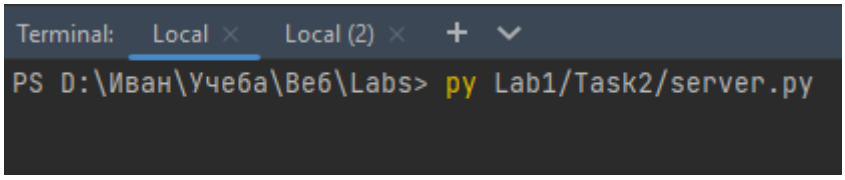
Файл “client.py”

```
import socket
import json

def tcp_client(host, port):
    # Create client socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Connect to the server
    client_socket.connect((host, port))

    # Read quadratic equation parameters from user input
    parameters = read_parameters()

    # Convert parameters into a json object
    json_string = json.dumps(parameters)

   # Send json object to the server
    client_socket.send(json_string.encode())

    # Receive a message from the server
    data = client_socket.recv(1024).decode()
```

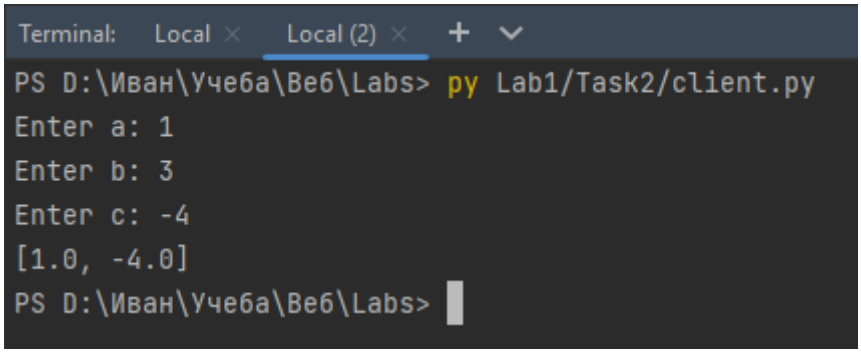
```
   # Parse the message
    # The message is a json object containing
    # a list of solutions
    solutions = json.loads(data)

    # Print solutions
    print(solutions)

    # Close the connection
    client_socket.close()

def read_parameters():
    while True:
        try:
            a = float(input("Enter a: "))
            b = float(input("Enter b: "))
            c = float(input("Enter c: "))
        except Exception:
            print("All values should be numerical. Try again.")
            continue
        break
    return (a, b, c)

if __name__ == '__main__':
    host = 'localhost'
    port = 5555
    tcp_client(host, port)
```

Работа программ:

Процесс “server.py”



Процесс “client.py”



Практическое задание 3

Листинги

Файл “server.py”

```
import socket

def tcp_server(host, port):
    # Create a server socket and put it into listening mode
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen(1)

    while True:
        # Accept a client connection
        client_socket, addr = server_socket.accept()

        # Open the index page file
        f = open("index.html")

        # Send file's content to the client
        client_socket.send(f.read().encode())

        # Close the connection
        client_socket.close()

if __name__ == "__main__":
    host = "localhost"
    port = 5555
    tcp_server(host, port)
```

Практическое задание 4

Листинги:

Файл “server.py”

```
import socket
import threading
import queue
import time

def handle_client(client_socket, clients, message_queue):
    # Recieve a client's login
    name = client_socket.recv(4096).decode("utf-8")

    # Register the new client
    clients.add(client_socket)

    # Serve the client until it breaks connection
    while True:
        try:
            # Recieve a message from the client
            message = client_socket.recv(4096).decode("utf-8")

            # Put the client's login concatenated with a message into the
message queue
            message_queue.put(name + ": " + message)
        except:
            # Once the connection is broken remove the client
            # from the registered clients' list and exit the loop
            clients.remove(client_socket)

            break

def broadcast_messages(clients, message_queue):
    # Broadcast messages forever
    while True:
        # Check if the message queue contains new messages
        # Broadcast all messages to all registered clients
        while not message_queue.empty():
```

```
            # Extract the first message in the queue
            message = message_queue.get()

            # Send the message to all registered clients
            for client in clients:
                client.send(message.encode())

        # Set a small time delay between checks
        time.sleep(0.05)

def run_server(host, port):
    # Create a socket and put in into listening mode
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen()

    # A clients set used to track all registered clients
    # that are in the chat
    clients = set()

    # A message queue used by broadcaster to broadcats
    # new messages to all registered clients
    message_queue = queue.Queue()

    # Start message broadcaster in a separate thread
    threading.Thread(target=broadcast_messages, args=[clients,
message_queue]).start()

    # Serve clients forever
    while True:
        # Accept a client connection
        client_socket, _ = server_socket.accept()

        # Serve the client in a separate thread
        threading.Thread(target=handle_client, args=[client_socket, clients,
message_queue]).start()
```

```
if __name__ == "__main__":
    host = "localhost"
    port = 5555
    run_server(host, port)
```

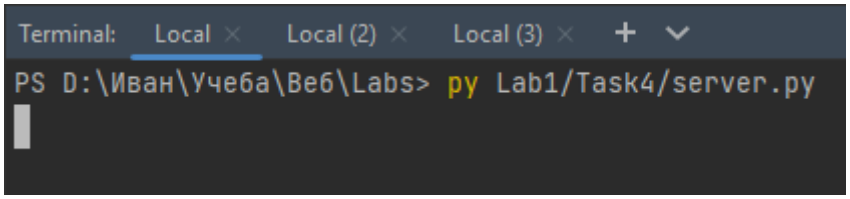
Файл “client.py”

```
import socket
import threading

def handle_network_inputs(client_socket):
    # Listen for incoming messages forever
    while True:
        # Recieve a new message
        message = client_socket.recv(4096).decode()

        # Print the message
        print(message)

def run_client(host, port):
    # Create a client socket and connect to the server
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((host, port))

    # Obtain a login from user input
    name = input("Enter your name: ")

    # Send the login to the server
    client_socket.send(name.encode())

    # Start a message listener in a separate thread
    threading.Thread(target=handle_network_inputs,
args=[client_socket]).start()

    # Send messages forever
    while True:
        # Obtain a message from user input
        message = input()
```

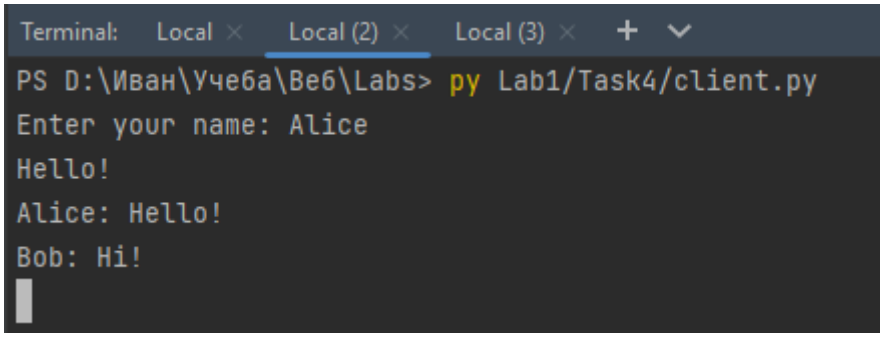
```
        # Send the message to the server
        client_socket.send(message.encode())

if __name__ == "__main__":
    host = "localhost"
    port = 5555
    run_client(host, port)
```

Работа программ:

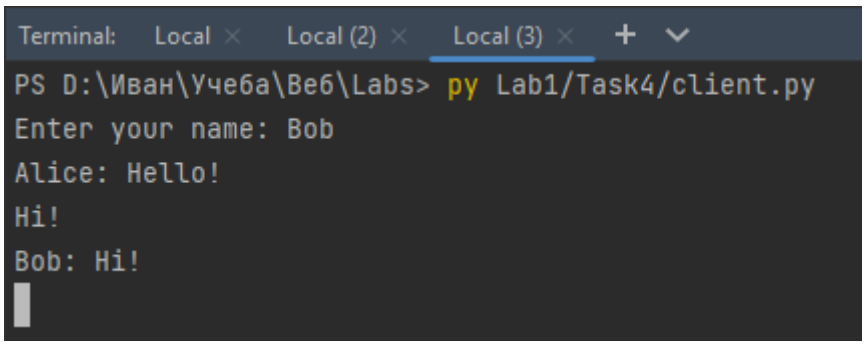
Процесс “server.py”



Процесс “client.py” (“Alice”)



Процесс “client.py” (“Bob”)



Практическое задание 5

Листинги

Файл “server.py”

```
from my_http_server import MyHTTPServer

if __name__ == "__main__":
   host = "localhost"
   port = 5555
   name = "example.local"
   my_server = MyHTTPServer(host, port, name)
   my_server.serve_forever()
```

Файл “my_http_server.py”

```
import socket
from http_request import HTTPRequest
from http_response import HTTPResponse

class MyHTTPServer:
   def __init__(self, host, port, server_name):
      self._host = host
      self._port = port
      self._server_name = server_name

   def serve_forever(self):
      # Create a server socket and put in into listening mode
      serv_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
      serv_sock.bind((self._host, self._port))
      serv_sock.listen()

      # Serve clients forever
      while True:
         # Accept a client connection
         conn, addr = serv_sock.accept()

         # Serve the client
         self.serve_client(conn)
```

```
        serv_sock.close()

    def serve_client(self, conn):
        # Recieve a raw text of client's request
        raw_request = self.recieve_raw_request(conn)

        # Parse the raw text into a request object
        # If the text format was incorrect we get None
        request = self.parse_request(raw_request)

        # Create a response object based on the request object
        response = self.handle_request(request)

        # Convert the respone into raw text
        raw_response = self.convert_response(response)

        # Send the response text to the client
        self.send_raw_response(conn, raw_response)

        # Close the connection
        conn.close()

    def recieve_raw_request(self, conn):
        # Recieve the request text
        return conn.recv(4096).decode()

    def parse_request(self, raw_request):
        # Split raw request text into a list of lines
        lines = raw_request.split("\n")

        # Request is not empty guard check
        if len(lines) == 0:
            return None

        # Split the first line into request line words
        request_params = lines[0].split(" ")
```

```
        # Request line contains three values guard check
        if len(request_params) != 3:
            return None
        method, url, version = list(map(lambda word: word.strip(),
request_params))

        # A headers dictionary
        headers = {}

        # A pointer for the separator line
        separator_line = None

        # Parse the headers section
        for i in range(1, len(lines)):
            # If the line is not empty consider it as a header
            # Otherwise consider it as a separator line
            if len(lines[i].strip()) > 0:
                header_words = list(map(lambda word: word.strip(),
lines[i].split(":")))

                # Incorrect header format guard check
                if len(header_words) < 2:
                    return None

                # The first header word is the header name and the rest
                # of the header words is the header value
                key, value = header_words[0], ":".join(header_words[1:])
                headers[key] = value
            else:
                separator_line = i
                break

        # The rest of the message starting from the line
        # right after the separator is the body
        body = "".join(lines[separator_line + 1:]) if separator_line else None

        return HTTPRequest(method, url, version, headers, body)
```

```
def handle_request(self, request):
   # None corresponds to a bad request
   if request is None:
      return self.handle_bad_request()

   # All version except for HTTP/1.1 are not supported
   if request.version != "HTTP/1.1":
      return self.handle_version_is_not_supported()

   # The only accessible resource on the server is its root
   if request.url != "/":
      return self.handle_not_found()

   # The only allowed methods are GET and POST
   if request.method == "GET":
      return self.handle_get_request(request)
   elif request.method == "POST":
      return self.handle_post_request(request)
   else:
      return self.handle_method_not_allowed()

def handle_bad_request(self):
   return HTTPResponse(400, "Bad Request")

def handle_version_is_not_supported(self):
   return HTTPResponse(505, "HTTP Version Not Supported")

def handle_not_found(self):
   return HTTPResponse(404, "Not Found")

def handle_method_not_allowed(self):
   return HTTPResponse(405, "Method Not Allowed", {"Allow": "GET, POST"})

def handle_get_request(self, request):
   with open("Lab1\Task5\grades.html", "r") as grades_file:
      body = grades_file.read()
```

```
            return HTTPResponse(200, "OK", request.headers, body)

    def handle_post_request(self, request):
        import re

        # The expression contained in the body matches the pattern guard check
        if not re.match("^discipline=.*&grade=.*$", request.body):
            return self.handle_bad_request()

        # Parse the values provided by the user
        discipline, grade = map(lambda statement: statement.split("=")[1],
request.body.split("&"))

        # Open grades.html and read its contents
        with open("Lab1\Task5\grades.html", "r") as read_grades_file:
            html_content = read_grades_file.read()

        # Find the end of the table
        end_table_index = html_content.rfind("</tbody>")

        # Rewrite the file with appended data
        with open("Lab1\Task5\grades.html", "w") as write_grades_file:
            new_row = f"  <tr><td>{discipline}</td><td>{grade}</td></tr>\n\t"
            modified_html = html_content[:end_table_index] + new_row +
html_content[end_table_index:]
            write_grades_file.write(modified_html)

        return HTTPResponse(201, "CREATED", request.headers)

    def convert_response(self, response):
        response_elements = []
        status_line = f"HTTP/1.1 {response.status} {response.reason}"
        response_elements.append(status_line)
        if response.headers:
            headers_lines = "\n".join([f"{key}: {value}" for key, value in
response.headers.items()])
            response_elements.append(headers_lines)
```

```
        response_elements.append("")
        if response.body:
            response_elements.append(response.body)
        return "\n".join(response_elements)

    def send_raw_response(self, conn, raw_response):
        conn.send(raw_response.encode())
```

Файл “my_http_request.py”

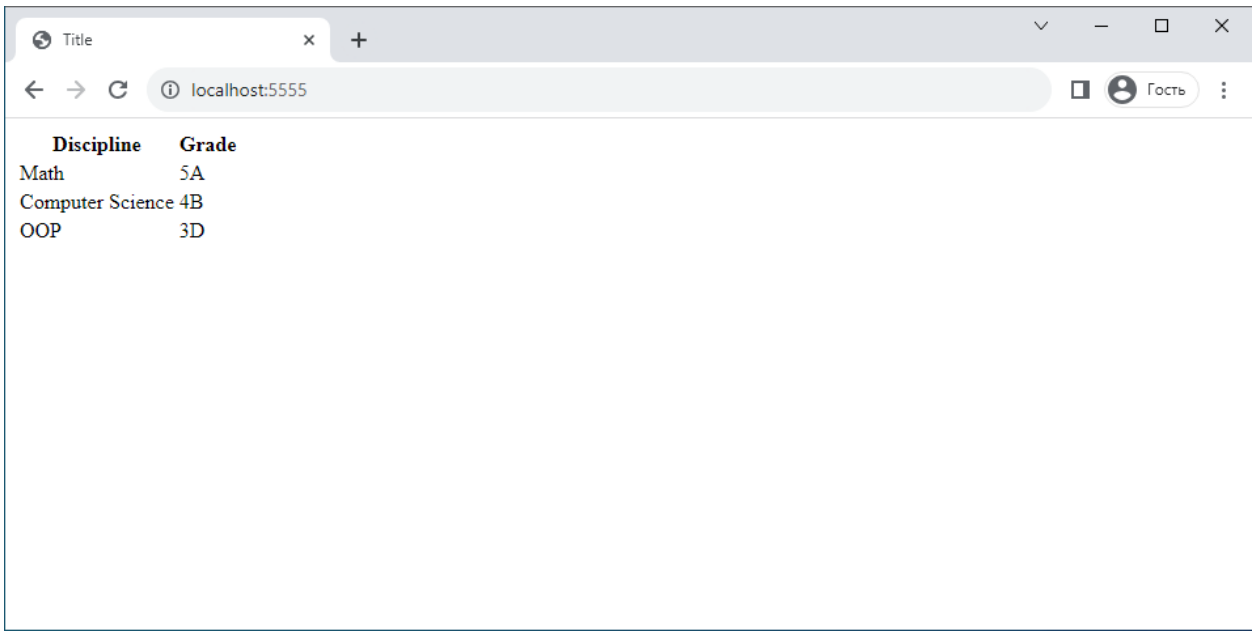
```
class HTTPRequest:
    def __init__(self, method, url, version, headers, body):
        self.method = method
        self.url = url
        self.version = version
        self.headers = headers
        self.body = body
```

Файл “my_http_response.py”

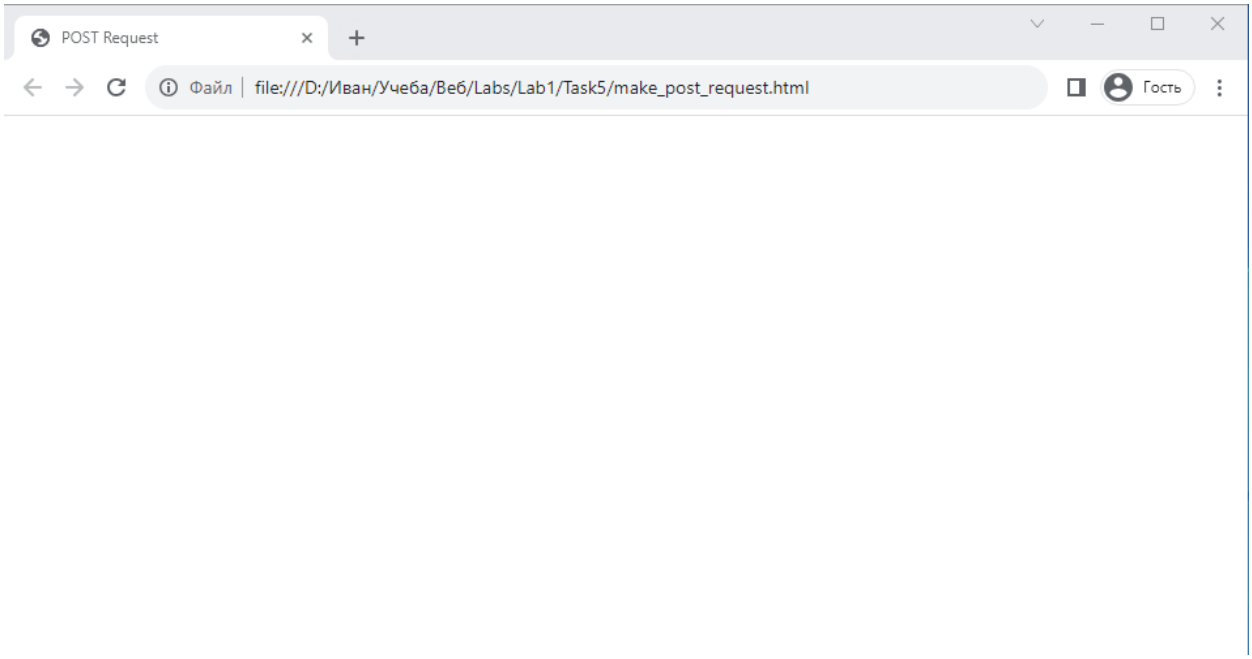
```
class HTTPResponse:
    def __init__(self, status, reason, headers=None, body=None):
        self.status = status
        self.reason = reason
        self.headers = headers
        self.body = body
```

Работа программ:

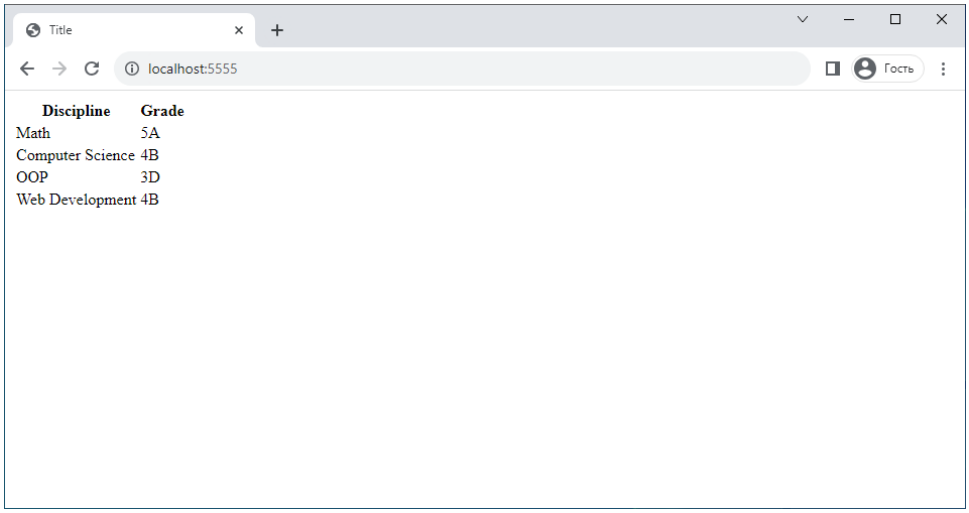
GET-запрос



POST-запрос



Результат POST-запроса



Выводы:

В ходе этой работы я овладел практическими навыками реализации web-серверов и использования с сокетов.