```
//client.cpp
#include <sys/socket.h>
#include <arpa/inet.h>
#include <iostream>
#include <netinet/in.h>
#include <cstring>
#include <unistd.h>
#include <fcntl.h>

#include "../json_reader/include/json_reader.h"

#define PATH_TO_JSON "../server_core/resources/config.json"
#define PATH_TO_DOWNLOADS "../client_downloads"
#define BUFFER_SIZE 1024

int get_code_status(const char* buffer) {
    char buff[3];
    for(int i = 7; i < 10; i++) {
        buff[i - 7] = buffer[i];
    }
    return std::stoi(buff);
}

void send_to_command_socket(const int& command_socket, const
char* command, bool& is_login, bool& is_password)
{
    if (send(command_socket, command, strlen(command), 0) == -1)
{
        if (errno == EPIPE) {
            std::cout << "Connection with server_core is
interrupted" << std::endl;
            is_login = false;
            is_password = false;
        } else {
            std::cerr << "Error of sending data: " <<
strerror(errno) << std::endl;
            is_login = false;
            is_password = false;
        }
    }
}
void send_to_command_socket(const int& command_socket, const
char* command, bool& stop_flag)
{
    if (send(command_socket, command, strlen(command), 0) == -1)
{
        if (errno == EPIPE) {
            std::cout << "Connection with server_core is
interrupted" << std::endl;
            stop_flag = true;
        } else {
            std::cerr << "Error of sending data: " <<
strerror(errno) << std::endl;
            stop_flag = true;
        }
    }
}
void send_to_data_socket(const int& data_socket, const char*
buff, bool& stop_flag)
{
    if (send(data_socket, buff, strlen(buff), 0) == -1) {
        if (errno == EPIPE) {
```

```
            std::cout << "Connection with server_core is
interrupted" << std::endl;
            stop_flag = true;
        } else {
            std::cerr << "Error of sending data: " <<
strerror(errno) << std::endl;
            stop_flag = true;
        }
    }
}
void send_to_data_socket(const int& data_socket, const char*
buff, bool& is_login, bool& is_password)
{
    if (send(data_socket, buff, strlen(buff), 0) == -1) {
        if (errno == EPIPE) {
            std::cout << "Connection with server_core is
interrupted" << std::endl;
            is_login = false;
            is_password = false;
        } else {
            std::cerr << "Error of sending data: " <<
strerror(errno) << std::endl;
            is_login = false;
            is_password = false;
        }
    }
}

void check_bytes_received(ssize_t bytes_received, bool&
stop_flag) {
    if (bytes_received == 0) {
        std::cout << "Connection with server_core is
interrupted" << std::endl;
        stop_flag = true;
    } else if (bytes_received == -1) {
        std::cerr << "Error of getting data" << strerror(errno)
<< std::endl;
        stop_flag = true;
    }
}
void check_bytes_received(ssize_t bytes_received, bool&
is_login, bool& is_password) {
    if (bytes_received == 0) {
        std::cout << "Connection with server_core is
interrupted" << std::endl;
        is_login = false;
        is_password = false;
    } else if (bytes_received == -1) {
        std::cerr << "Error of getting data" << strerror(errno)
<< std::endl;
        is_login = false;
        is_password = false;
    }
}



void echo_command(const char* command, char* buff,  const int&
command_socket, const int& data_socket, bool& stop_flag) {

    send_to_command_socket(command_socket, command, stop_flag);
    send_to_data_socket(data_socket, buff, stop_flag);
```

```
    ssize_t bytes_received = recv(command_socket, buff,
BUFFER_SIZE, 0);
    buff[bytes_received] = '\0';

    if (bytes_received == 0) {
        std::cout << "Connection with server_core is
interrupted" << std::endl;
        stop_flag = true;
    } else if (bytes_received == -1) {
        std::cerr << "Error of getting data" << strerror(errno)
<< std::endl;
        stop_flag = true;
    }

    std::cout << buff << std::endl;

    if(get_code_status(buff) == 501 || get_code_status(buff) ==
503)
        return;

    if(get_code_status(buff) == 221){
        stop_flag = true;
        return;
    }

    bytes_received = recv(data_socket, buff, BUFFER_SIZE, 0);
    buff[bytes_received] = '\0';

    if (bytes_received == 0) {
        std::cout << "Connection with server_core is
interrupted" << std::endl;
        stop_flag = true;
    } else if (bytes_received == -1) {
        std::cerr << "Error of getting data" << strerror(errno)
<< std::endl;
        stop_flag = true;
    }

    std::cout << buff << std::endl;

}

void list_command(const char* command, char* buff, const int&
command_socket, const int& data_socket, bool& stop_flag) {

    send_to_command_socket(command_socket, command, stop_flag);

    ssize_t bytes_received = recv(command_socket, buff,
BUFFER_SIZE, 0);
    buff[bytes_received] = '\0';

    check_bytes_received(bytes_received, stop_flag);

    std::cout << buff << std::endl;

    int status_code = get_code_status(buff);

    if(status_code == 501 || status_code == 503 || status_code
== 500)
        return;
```

```
    if(status_code == 221){
        stop_flag = true;
        return;
    }

    int flags = fcntl(data_socket, F_GETFL, 0);
    if (flags == -1) {
        std::cerr << "Error of getting sockets flags" <<
std::endl;
        return;
    }
    if (fcntl(data_socket, F_SETFL, flags | O_NONBLOCK) == -1) {
        std::cerr << "Error of setting sockets flags" <<
std::endl;
        return;
    }

    std::string result;
    char buffer[1024];
    bytes_received = 0;

    while (true) {
        ssize_t received = recv(data_socket, buffer,
sizeof(buffer), 0);
        if (received == -1 || received == 0)
            break;
        result.append(buffer, received);
        bytes_received += received;
    }


    flags &= ~O_NONBLOCK;
    if (fcntl(data_socket, F_SETFL, flags) == -1) {
        std::cerr << "Error of setting sockets flags" <<
std::endl;
        return;
    }

    std::cout << result << std::endl;

}

void cwd_command(const char* command, char* buff, const int&
command_socket, const int& data_socket, bool& stop_flag) {
    send_to_command_socket(command_socket, command, stop_flag);
    send_to_data_socket(data_socket, buff, stop_flag);

    ssize_t bytes_received = recv(command_socket, buff,
BUFFER_SIZE, 0);
    buff[bytes_received] = '\0';

    check_bytes_received(bytes_received, stop_flag);

    std::cout << buff << std::endl;

    int status_code = get_code_status(buff);

    if(status_code == 501 || status_code == 503 || status_code
== 500 || status_code == 404)
        return;

    if(status_code == 221){
```

```
        stop_flag = true;
        return;
    }

    bytes_received = recv(data_socket, buff, BUFFER_SIZE, 0);
    buff[bytes_received] = '\0';

    check_bytes_received(bytes_received, stop_flag);

    std::cout << buff << std::endl;

}

void authorize(int fcs, int fds) {
    bool is_login = false;
    bool is_password = false;


    while (!is_login) {
        char command[BUFFER_SIZE];
        char data[BUFFER_SIZE];

        std::cin >> command;
        if (std::cin.peek() == ' ') {
            std::cin.ignore();
            std::cin.getline(data, sizeof(data));
        } else {
            data[0] = '\0';
        }

        send_to_command_socket(fcs, command, is_login,
is_password);
        send_to_data_socket(fds, data, is_login, is_password);

        ssize_t bytes_received = recv(fcs, command, BUFFER_SIZE,
0);
        command[bytes_received] = '\0';

        check_bytes_received(bytes_received, is_login,
is_password);

        std::cout << command << std::endl;

        int status_code = get_code_status(command);
        if (status_code == 331)
            is_login = true;

        memset(command, 0, BUFFER_SIZE);
        memset(command, 0, BUFFER_SIZE);

    }

    while (!is_password) {
        char command[BUFFER_SIZE];
        char data[BUFFER_SIZE];

        std::cin >> command;
        if (std::cin.peek() == ' ') {
            std::cin.ignore();
            std::cin.getline(data, sizeof(data));
        } else {
            data[0] = '\0';
```

```
        }

        send_to_command_socket(fcs, command, is_login,
is_password);
        send_to_data_socket(fds, data, is_login, is_password);

        ssize_t bytes_received = recv(fcs, command, BUFFER_SIZE,
0);
        command[bytes_received] = '\0';

        check_bytes_received(bytes_received, is_login,
is_password);

        std::cout << command << std::endl;

        int status_code = get_code_status(command);
        if (status_code == 230)
            is_password = true;

        memset(command, 0, BUFFER_SIZE);
        memset(command, 0, BUFFER_SIZE);

    }
}

void quit_command(const char* command, char* buff, const int&
command_socket, const int& data_socket, bool& stop_flag) {
    send_to_command_socket(command_socket, command, stop_flag);
    ssize_t bytes_received = recv(command_socket, buff,
BUFFER_SIZE, 0);
    buff[bytes_received] = '\0';

    check_bytes_received(bytes_received, stop_flag);

    std::cout << buff << std::endl;
    stop_flag = true;
}

void retr_command(const char* command, char* buff, const int&
command_socket, const int& data_socket, bool& stop_flag) {
    int status_code;
    int size_of_file_from_server;
    std::string path_to_file = std::string(PATH_TO_DOWNLOADS) +
"/" + buff;

    timeval tv_recv{};
    tv_recv.tv_sec = 3;
    tv_recv.tv_usec = 0;
    setsockopt(data_socket, SOL_SOCKET, SO_RCVTIMEO, &tv_recv,
sizeof(tv_recv));

    send_to_command_socket(command_socket, command, stop_flag);
    send_to_data_socket(data_socket, buff, stop_flag);

    std::ofstream file (path_to_file, std::ios::binary |
std::ios::trunc);
    if (!file.is_open()) {
        std::cerr << "Failed to open file for writing." <<
std::endl;
    }
```

```
    //проверка, есть ли такой файл в текущей директории и
получилось ли его открыть.(сервер возвращает 200 в случае успеха
)
    ssize_t bytes_received = recv(command_socket, buff,
BUFFER_SIZE, 0);
    buff[bytes_received] = '\0';
    check_bytes_received(bytes_received, stop_flag);

    status_code = get_code_status(buff);
    if(status_code == 550 || status_code == 503 || status_code
== 501) {
        std::cout << buff << std::endl;
        file.close();
        return;
    }


    std::string result;
    char buffer[1024];
    bytes_received = 0;
    recv(data_socket, &size_of_file_from_server, sizeof(int),
0);
    while (true) {
        ssize_t received = recv(data_socket, buffer,
sizeof(buffer), 0);
        if (received == -1 || received == 0)
            break;
        file.write(buffer, received);
        bytes_received += received;
    }

    if(bytes_received != size_of_file_from_server) {
        std::cout << "Doesnt similar sizes of files on server
and on client" << std::endl;
    }

    bytes_received = recv(command_socket, buff, BUFFER_SIZE, 0);
    buff[bytes_received] = '\0';
    check_bytes_received(bytes_received, stop_flag);
    status_code = get_code_status(buff);
    //проверка на успех передачи со стороны сервера ( возращает
226 в случае успеха )

    if(status_code != 226) {

        std::cout << buff << std::endl;
        file.close();
        std::remove(path_to_file.c_str());
        return;
    }

    std::cout << buff << std::endl;

    tv_recv.tv_sec = 0;
    tv_recv.tv_usec = 0;
    setsockopt(data_socket, SOL_SOCKET, SO_RCVTIMEO, &tv_recv,
sizeof(tv_recv));
    file.close();

}

int main() {
```

```
    std::string json = Json_Reader::get_json(PATH_TO_JSON);
    int server_port = stoi(Json_Reader::find_value(json,
"serverPort"));
    std::string local_ip_address = Json_Reader::find_value(json,
"localIpAddress");

    int command_socket = socket(AF_INET, SOCK_STREAM, 0);
    int data_socket = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in server_addr{};
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(server_port);
    server_addr.sin_addr.s_addr =
inet_addr(local_ip_address.data());
    if(connect(command_socket, (struct sockaddr*)&server_addr,
sizeof(server_addr)) < 0 ) {
        std::cout << "cannot to connect to host" << std::endl;
    }

    server_addr.sin_port = htons(server_port);
    if(connect(data_socket, (struct sockaddr*)&server_addr,
sizeof(server_addr)) < 0 ) {
        std::cout << "cannot to connect to host" << std::endl;
    }

    char buff[BUFFER_SIZE];
    recv(command_socket, buff, sizeof(buff), 0);
    std::cout << buff << std::endl;

    authorize(command_socket, data_socket);

    bool stop_flag = false;
    while (!stop_flag) {
        char command[BUFFER_SIZE];
        char data[BUFFER_SIZE];

        std::cin >> command;
        if (std::cin.peek() == ' ') {
            std::cin.ignore();
            std::cin.getline(data, sizeof(data));
        } else {
            data[0] = '\0';
        }

        if(strcmp(command, "ECHO") == 0) {
            echo_command(command, data, command_socket,
data_socket, stop_flag);
        }else if (strcmp(command, "LIST") == 0){
            list_command(command, data, command_socket,
data_socket, stop_flag);
        }else if (strcmp(command, "CWD") == 0){
            cwd_command(command, data, command_socket,
data_socket, stop_flag);
        }else if(strcmp(command, "RETR") == 0){
            retr_command(command, data, command_socket,
data_socket, stop_flag);
        }else if (strcmp(command, "QUIT") == 0){
            quit_command(command, data, command_socket,
data_socket, stop_flag);
            break;
        }
```

```
        memset(command, 0, BUFFER_SIZE);
        memset(command, 0, BUFFER_SIZE);

    }

    close(data_socket);
    close(command_socket);

    return 0;


}
//FTPSpecification.cpp
#include "FTPSpecification.h"

std::mutex FTPSpecification::retr_mutex;

void FTPSpecification::handler(char* command , int fcs, int fds)
{

    if(strcmp(command, ECHO_COMMAND) == 0) {
        echo_handler(fcs, fds);
    }else if(strcmp(command, LIST_COMMAND) == 0) {
        list_handler(fcs, fds);
    }else if(strcmp(command, CWD_COMMAND) == 0) {
        cwd_handler(fcs, fds);
    }else if(strcmp(command, DOWNLOAD_COMMAND) == 0) {
        {
            std::lock_guard<std::mutex> lock(retr_mutex);
            retr_handler(fcs, fds);
        }
    }else
    {
        send(fcs, BAD_SEQUENCE_OF_COMMANDS,
strlen(BAD_SEQUENCE_OF_COMMANDS), 0);
        clear_socket_data(fds);
    }

}

void FTPSpecification::echo_handler(int fcs, int fds) {

    char buff[1024];
    ssize_t valread;

    timeval tv_recv{};
    tv_recv.tv_sec = 1;
    tv_recv.tv_usec = 0;
    setsockopt(fds, SOL_SOCKET, SO_RCVTIMEO, &tv_recv,
sizeof(tv_recv));

    valread = recv(fds, buff, sizeof(buff), 0);
    if(valread == -1 || valread == 0) {
        send(fcs, SYNTAX_ERROR, strlen(SYNTAX_ERROR), 0);
        return;
    }

    tv_recv.tv_sec = 0;
    tv_recv.tv_usec = 0;
```

```
    setsockopt(fds, SOL_SOCKET, SO_RCVTIMEO, &tv_recv,
sizeof(tv_recv));

    buff[valread] = '\0';
    std::cout << "\033[1;34mECHO command:\033[0m " << buff <<
get_client_info(fcs) << std::endl;
    send(fds, buff, strlen(buff), 0);
    send(fcs, DONE_SUCCESSFULLY, strlen(DONE_SUCCESSFULLY), 0);

}

void FTPSpecification::clear_socket_data(int socket_fd) {
    int bytes_available;
    ioctl(socket_fd, FIONREAD, &bytes_available);
    char buffer[bytes_available];
    if(bytes_available > 0) {
        recv(socket_fd, buffer, bytes_available, 0);
    }
}

void FTPSpecification::list_handler(int fcs, int fds) {

    std::string result = parse_current_dir();
    if(strcmp(result.data(), INTERNAL_SERVER_ERROR) == 0) {
        send(fcs, INTERNAL_SERVER_ERROR,
strlen(INTERNAL_SERVER_ERROR), 0);
        return;
    }
    ssize_t bytes_sent = 0;
    const char* data = result.c_str();
    while(bytes_sent < result.length())
    {
        int bytes_to_send = std::min(1024, (int)result.length()
- (int)bytes_sent);
        ssize_t sent = send(fds, data + bytes_sent,
bytes_to_send, 0);
        if (sent == -1) {
            send(fcs, INTERNAL_SERVER_ERROR,
strlen(INTERNAL_SERVER_ERROR), 0);
            return;
        }
        bytes_sent += sent;
    }
    send(fcs, LIST_TRANSFER_DONE, strlen(LIST_TRANSFER_DONE),
0);
    std::cout << "\033[1;34mLIST command:\033[0m " <<
get_client_info(fcs) << std::endl;


}

std::string FTPSpecification::parse_current_dir() {
    DIR* dir = opendir(current_dir.c_str());
    std::string result;
    if (dir == nullptr) {
        std::cerr << "Error of opening dir: " << current_dir <<
std::endl;
        return INTERNAL_SERVER_ERROR;
    }
    struct dirent* entry;
    while ((entry = readdir(dir)) != nullptr) {
```

```
        if (entry->d_type == DT_REG || entry->d_type == DT_DIR)
{
            if (entry->d_name[0] != '.') {
                if (entry->d_type == DT_DIR) {
                    result.append("\033[1;34m"); // Blue colour
                } else {
                    result.append("\033[1;35m"); // Pink colour
                }
                result.append(entry->d_name);
                result.append("\033[0m"); // Reset colour
                result.append("\n");
            }
        }
    }

    closedir(dir);
    return result;
}

void FTPSpecification::cwd_handler(int fcs, int fds) {
    char buff[1024];
    ssize_t valread;
    std::string old_current_dir = current_dir;

    int bytes_available;
    ioctl(fds, FIONREAD, &bytes_available);
    if(bytes_available == 0) {
        send(fcs, SYNTAX_ERROR, strlen(SYNTAX_ERROR), 0);
        return;
    }
    valread = recv(fds, buff, sizeof(buff), 0);
    buff[valread] = '\0';
    std::cout << "\033[1;34mCWD command:\033[0m " << buff <<
get_client_info(fcs) << std::endl;


    if (chdir(old_current_dir.c_str()) == -1) {
        send(fcs, INVALID_PATH, strlen(INVALID_PATH), 0);
        return;
    }

    struct stat statbuf{};
    if (stat(buff, &statbuf) == -1) {
        send(fcs, INVALID_PATH, strlen(INVALID_PATH), 0);
        return;
    }

    if (chdir(buff) == -1) {
        send(fcs, INVALID_PATH, strlen(INVALID_PATH), 0);
        return;
    }
    current_dir = std::filesystem::current_path();

    if (chdir(baser_dir.c_str()) == -1) {
        send(fcs, INVALID_PATH, strlen(INVALID_PATH), 0);
        return;
    }


    send(fds, current_dir.c_str(), strlen(current_dir.c_str()),
0);
    send(fcs, SUCCESSFUL_CHANGE, strlen(SUCCESSFUL_CHANGE), 0);
```

```
}

void FTPSpecification::retr_handler(int fcs, int fds) {
    char buff[BUFFER_SIZE];
    ssize_t valread;
    int size_of_file;
    std::string path_to_file;

    valread = recv(fds, buff, sizeof(buff), 0);
    buff[valread] = '\0';
    path_to_file = current_dir + "/" + buff;
    std::cout << "\033[1;34mRETR command:\033[0m " << buff <<
get_client_info(fcs) << std::endl;

    if (!std::filesystem::exists(path_to_file)) {
        send(fcs, FILE_UNAVAILABLE, strlen(FILE_UNAVAILABLE),
0);
        return;
    }
    if (!std::filesystem::is_regular_file(path_to_file)) {
        send(fcs, FILE_UNAVAILABLE, strlen(FILE_UNAVAILABLE),
0);
        return;
    }

    std::ifstream file(path_to_file, std::ios::binary);

    if(!file.is_open()) {
        send(fcs, FILE_UNAVAILABLE, strlen(FILE_UNAVAILABLE),
0);
        return;
    }

    size_of_file = std::filesystem::file_size(path_to_file);

    send(fcs,DONE_SUCCESSFULLY, strlen(DONE_SUCCESSFULLY), 0);

    send(fds, &size_of_file, sizeof(int), 0);

    char buffer_to_send[1024];
    ssize_t total_bytes_sent = 0;
    while (true) {

        file.read(buffer_to_send, sizeof(buffer_to_send));
        ssize_t bytes_read = file.gcount();
        if (bytes_read == 0) {
            break;
        } else if (bytes_read == -1) {
            send(fcs, ERROR_SENDING_FILE,
strlen(ERROR_SENDING_FILE), 0);
            file.close();
            return;
        }
        ssize_t bytes_sent = send(fds, buffer_to_send,
bytes_read, 0);
        total_bytes_sent += bytes_sent;
        if (bytes_sent == -1) {
            send(fcs, ERROR_SENDING_FILE,
strlen(ERROR_SENDING_FILE), 0);
            file.close();
            return;
        }
```

```
    }

    if(total_bytes_sent != size_of_file) {
        send(fcs, ERROR_SENDING_FILE,
strlen(ERROR_SENDING_FILE), 0);
    } else {
        send(fcs, SUCCESSFUL_DOWNLOAD,
strlen(SUCCESSFUL_DOWNLOAD), 0);
    }

}

std::string FTPSpecification::get_client_info(int fcs) {
    sockaddr_in address {};
    int addrlen = sizeof(address);
    getpeername(fcs, (struct sockaddr*)&address,
(socklen_t*)&addrlen);
    std::string result;
    result.append("\033[1;34m IP:\033[0m "); // Синий цвет для
IP
    result.append("\033[1;32m"); // Зеленый цвет для адреса
    result.append(inet_ntoa(address.sin_addr));
    result.append("\033[0m, ");
    result.append("\033[1;34mPORT:\033[0m "); // Синий цвет для
PORT
    result.append("\033[1;32m"); // Зеленый цвет для порта
    result.append(std::to_string(ntohs(address.sin_port)));
    result.append("\033[0m");
    return result;
}

std::vector<std::string> FTPSpecification::split_path(const
std::string &path_string) {
    std::vector<std::string> commands;
    std::istringstream iss(path_string);
    std::string token;

    while (std::getline(iss, token, '/')) {
        if (!token.empty()) {
            commands.push_back(token);
        }
    }

    return commands;
}

//json_reader.cpp
#include "../include/json_reader.h"

using namespace std;


string Json_Reader::get_json(const string& path){
    ifstream file;
    file.open(path, ios::in);
    string line;
    string context = "";
    while (getline(file, line))
    {
        for(int i = 0; i < line.size(); i++)
        {
```

```
            if (line[i] != ' ' && line[i] != '\t' && line[i] !=
'\n'){
                context += line[i];
            }
        }
    }
    return context;
}

string Json_Reader::find_value(string json, const string& key)
{
    string result;
    int first_index, second_index;
    int i = 0,j;
    while(i < json.size() ){
        while(i < json.size() && json[i] != QOUTATION) i++;
        first_index = ++i;
        while(i < json.size() && json[i] != QOUTATION) i++;
        second_index = i;
        i += 2;
        if (json.substr(first_index, second_index - first_index)
== key){
            if (json[i] == LBRACK){
                first_index = i++;
                while(i < json.size() && json[i] != RBRACK) i++;
                second_index = i++;
                result = json.substr(first_index, second_index -
first_index + 1);
            }
            else if(json[i] == QOUTATION){
                first_index = i++;
                while(i < json.size() && json[i] != QOUTATION)
i++;
                second_index = i++;
                result = json.substr(first_index+1, second_index
- first_index - 1);
            }
            else{
                first_index = i;
                while(i < json.size() && json[i] != COMMA &&
json[i] != RBRACE ) i++;
                second_index = i;
                result = json.substr(first_index, second_index -
first_index);
            }
            return result;
        }
        if (json[i] == LBRACK){
            while(i < json.size() && json[i] != RBRACK) i++;
            while(i < json.size() && json[i] != COMMA) i++;
        }
        else{
            while(i < json.size() && json[i] != COMMA) i++;
        }
    }
    return "";
}

vector<string> Json_Reader::split_array(string array)
{
    int i = 0;
    int first_index, second_index;
```

```
    vector<string> result;
    if (array[i] != LBRACK) return result;
    i++;
    while(i < array.size())
    {
        while (i < array.size() && array[i] != LBRACE &&
array[i] != QOUTATION) i++;
        if(i < array.size() && array[i] == LBRACE)
        {
            first_index = i;
            while (i < array.size() && array[i] != RBRACE) i++;
            second_index = ++i;
            result.push_back(array.substr(first_index,
second_index - first_index));
            while (i < array.size() && array[i] != COMMA &&
array[i] != RBRACK) i++;
            i++;
        }
        else if (i < array.size() && array[i] == QOUTATION)
        {
            first_index = ++i;
            while(i < array.size() && array[i] != QOUTATION)
i++;
            second_index = i;
            result.push_back(array.substr(first_index,
second_index - first_index));
            while (i < array.size() && array[i] != COMMA &&
array[i] != RBRACK) i++;
            i++;
        }
    }
    return result;
}
//ServerCore.cpp
#include "../include/ServerCore.h"

///NOTE IF YOU WANT TO CHANGE DIRECTORY AND AFTER THAT CONNECT
FROM ANOTHER CLIENT
///YOU HAVE TO GO BACK TO HOME DIRECTORY BECAUSE JSON CHDIR
CHANGES DIRECTORY
///AND MAKE CWD COMMAND PROPERLY ( use path in class )

/**

    @brief Starts the server application.
    The start function first calls the
create_bind_listen_sockets method to set up the sockets in the
required state.
    Then, it invokes the thread_pool, where the handlingAccept
function is asynchronously executed in the background thread.
    The handlingAccept function contains an infinite loop for
accepting new clients.
    Once a client is accepted, another thread is spawned from
the same thread pool to handle the server-side client
operations,
    including the authentication process through the
handle_command function.
    @note This function should be called to initiate the server
application.
    */
void ServerCore::start() {
```

```
    create_bind_listen_sockets();

}

void ServerCore::create_bind_listen_sockets() {

    std::string json = Json_Reader::get_json(PATH_TO_JSON);
    server_port = std::stoi(Json_Reader::find_value(json,
"serverPort"));
    local_ip_address = Json_Reader::find_value(json,
"localIpAddress");
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        std::cerr << "Error of creating a server_socket" <<
std::endl;
        return;
    }

    sockaddr_in server_hint{};
    server_hint.sin_family = AF_INET;
    server_hint.sin_port = htons(server_port);
    server_hint.sin_addr.s_addr =
inet_addr(local_ip_address.data());



    if (bind(server_socket, reinterpret_cast<struct
sockaddr*>(&server_hint), sizeof(server_hint)) < 0) {
        std::cout << "Failed to bind server_socket." <<
std::endl;
        return;
    }


    if(listen(server_socket, SOMAXCONN) == -1) {
        std::cout << "Failed to listen data_socket" <<
std::endl;
        return;
    }

    thread_pool.addJob([this]{handlingAccept();});


}


/**

    @brief Function for accepting new clients.
    The handlingAccept function is executed in a background
thread to continuously accept new client connections.
    Upon accepting a client, a new thread is created to handle
server-side client operations,
    such as the authentication process through the
handle_command function.
    This function runs indefinitely until the server is stopped.
    */
void ServerCore::handlingAccept() {

    while (true) {

        auto* new_client = new ServerClient;
```

```
        sockaddr_in client_addr{};

        socklen_t addrlen = sizeof(client_addr);

        new_client->command_socket = accept(server_socket,
(struct sockaddr *) &client_addr, &addrlen);

        new_client->data_socket = accept(server_socket, (struct
sockaddr *) &client_addr, &addrlen);

        new_client->connected();

        /**

        @brief Handles the command received from the client.
        The handle_command function is responsible for
processing commands received from the client.
        It performs the necessary operations, including
authentication, based on the received command.
        @param client The client connection object.
        */
        std::thread tr([new_client](){
            char buffer[1024];
            ssize_t valread;

            new_client->authorize();
            while(true) {

                valread = new_client-
>get_command_from_client(buffer);
                if (strcmp(buffer, "QUIT") == 0 || valread == -1
|| valread == 0)
                {
                    if(new_client->is_authorized) {
                        send(new_client->command_socket,
SUCCESSFUL_QUIT, strlen(SUCCESSFUL_QUIT), 0);
                        new_client->disconnect();
                    }

                    break;
                }
                else
                    new_client->handle_command(buffer);

            }

        });
        tr.detach();

    }

}

void ServerCore::joinLoop() {
    thread_pool.join();
}

void ServerClient::disconnect() {

    sockaddr_in address {};
    int addrlen = sizeof(address);
```

```
    getpeername(command_socket, (struct sockaddr*)&address,
(socklen_t*)&addrlen);
    std::cout << "\033[1;31mGuest disconnected, ip\033[0m " <<
inet_ntoa(address.sin_addr)
              << " , \033[1;31mport\033[0m " <<
ntohs(address.sin_port) << std::endl;
    close(data_socket);
    close(command_socket);
    command_socket = 0;
    data_socket = 0;

}

void ServerClient::connected() const {
    sockaddr_in address {};
    int addrlen = sizeof(address);
    getpeername(command_socket, (struct sockaddr*)&address,
(socklen_t*)&addrlen);
    std::cout << "\033[1;32mGuest connected, ip\033[0m " <<
inet_ntoa(address.sin_addr)
              << " , \033[1;32mport\033[0m " <<
ntohs(address.sin_port) << std::endl;
    send(command_socket, SUCCESSFULLY_CONNECTED,
strlen(SUCCESSFULLY_CONNECTED), 0);
}

ssize_t ServerClient::get_command_from_client(char buffer[])
const {
    ssize_t valread;
    valread = recv(command_socket, buffer, PACKET_SIZE, 0);
    buffer[valread] = '\0';
    return valread;
}

void ServerClient::handle_command(char command[]) const {
    ftp_specification->handler(command, command_socket,
data_socket);
}

void ServerClient::authorize() {

    char buffer[1024];
    size_t valread;
    bool is_login = false;
    bool is_password = false;
    std::string json = Json_Reader::get_json(PATH_TO_JSON);
    std::string login_name;



    while(!is_login) {
        memset(buffer, 0, sizeof(buffer));
        valread = get_command_from_client(buffer);
        if (strcmp(buffer, "QUIT") == 0 || valread == -1 ||
valread == 0)
        {
            send(command_socket, SUCCESSFUL_QUIT,
strlen(SUCCESSFUL_QUIT), 0);
            disconnect();
            free(this);
            is_password = true;
```

```
            break;
        }
        else if (strcmp(buffer, "USER") == 0)
        {

            valread = get_data_from_client(buffer);
            if(valread == -1 || valread == 0) {
                send(command_socket,
INVALID_USERNAME_OR_PASSWORD,
strlen(INVALID_USERNAME_OR_PASSWORD), 0);
                clear_socket_data(data_socket);
                continue;
            }
            std::vector<std::string> json_vector =
Json_Reader::split_array(Json_Reader::find_value(json,
"users"));
            std::string name;
            for (const auto &user_info: json_vector) {
                name = Json_Reader::find_value(user_info,
"user");
                if (strcmp(name.c_str(), buffer) == 0) {
                    login_name = name.c_str();
                    is_login = true;
                }
            }
            if (is_login) {
                send(command_socket, USERNAME_ACCEPTED,
strlen(USERNAME_ACCEPTED), 0);
                clear_socket_data(data_socket);
            } else {
                send(command_socket,
INVALID_USERNAME_OR_PASSWORD,
strlen(INVALID_USERNAME_OR_PASSWORD), 0);
                clear_socket_data(data_socket);

            }

        } else {
            send(command_socket, NEED_FOR_ACCOUNT,
strlen(NEED_FOR_ACCOUNT), 0);
            clear_socket_data(data_socket);
        }
    }

    while(!is_password) {
        memset(buffer, 0, sizeof(buffer));
        valread = get_command_from_client(buffer);
        if (strcmp(buffer, "QUIT") == 0 || valread == -1 ||
valread == 0)
        {
            send(command_socket, SUCCESSFUL_QUIT,
strlen(SUCCESSFUL_QUIT), 0);
            disconnect();
            free(this);
            break;
        }
        else if (strcmp(buffer, "PASS") == 0)
        {
            valread = get_data_from_client(buffer);
            if(valread == -1 || valread == 0) {
```

```
                send(command_socket,
INVALID_USERNAME_OR_PASSWORD,
strlen(INVALID_USERNAME_OR_PASSWORD), 0);
                clear_socket_data(data_socket);
                continue;
            }
            std::vector<std::string> json_vector =
Json_Reader::split_array(Json_Reader::find_value(json,
"users"));
            std::string password;
            std::string name;
            for (const auto &user_info: json_vector) {
                password = Json_Reader::find_value(user_info,
"password");
                name = Json_Reader::find_value(user_info,
"user");
                if (strcmp(password.c_str(), buffer) == 0 &&
strcmp(login_name.c_str(), name.c_str()) == 0) {
                    is_password = true;
                }
            }
            if (is_password) {
                send(command_socket, PASSWORD_ACCEPTED,
strlen(PASSWORD_ACCEPTED), 0);
                clear_socket_data(data_socket);
            } else {
                send(command_socket,
INVALID_USERNAME_OR_PASSWORD,
strlen(INVALID_USERNAME_OR_PASSWORD), 0);
                clear_socket_data(data_socket);
            }


        } else {
            send(command_socket, NEED_FOR_ACCOUNT,
strlen(NEED_FOR_ACCOUNT), 0);
            clear_socket_data(data_socket);
        }
    }

    if(is_password && is_login) {
        std::cout << "\033[1;32mAuthorized successfully\033[0m"
<< std::endl;
        is_authorized = true;
    }

}

size_t ServerClient::get_data_from_client(char *buffer) {

    timeval tv_recv{};
    tv_recv.tv_sec = 1;
    tv_recv.tv_usec = 0;
    setsockopt(data_socket, SOL_SOCKET, SO_RCVTIMEO, &tv_recv,
sizeof(tv_recv));

    size_t valread;
    valread = recv(data_socket, buffer, PACKET_SIZE, 0);
    buffer[valread] = '\0';

    tv_recv.tv_sec = 0;
    tv_recv.tv_usec = 0;
```

```
    setsockopt(data_socket, SOL_SOCKET, SO_RCVTIMEO, &tv_recv,
sizeof(tv_recv));

    return valread;
}

void ServerClient::clear_socket_data(int socket_fd) {
    int bytes_available;
    ioctl(socket_fd, FIONREAD, &bytes_available);
    char buffer[bytes_available];
    if(bytes_available > 0) {
        recv(socket_fd, buffer, bytes_available, 0);
    }
}
//server.cpp
#include <iostream>
#include "../server_core/include/ServerCore.h"



int main() {
    std::cout << "\033[1;32mStarted a work of
server_core\033[0m" << std::endl;

    ServerCore server;
    server.start();
    server.joinLoop();

    return 0;
}
//FTPSpecification.h
#pragma once

#include <cstring>
#include <sys/ioctl.h>
#include <iostream>
#include <sys/socket.h>
#include <dirent.h>
#include <sys/stat.h>
#include <csignal>

#include <filesystem>
#include <fstream>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <vector>
#include <mutex>


#define LIST_COMMAND "LIST"
#define CWD_COMMAND "CWD"
#define DOWNLOAD_COMMAND "RETR"
#define ECHO_COMMAND "ECHO"

#define SUCCESSFULLY_CONNECTED "\033[1;32m220: Welcome to FTP
Server\033[0m" // green
#define BAD_SEQUENCE_OF_COMMANDS "\033[1;31m503: Bad sequence of
commands.\033[0m" // red
#define INVALID_USERNAME_OR_PASSWORD "\033[1;31m430: Invalid
username or password\033[0m" // red
```

```
#define USERNAME_ACCEPTED "\033[1;32m331: User name okay, need
password.\033[0m" // green
#define PASSWORD_ACCEPTED "\033[1;32m230: User logged in,
proceed. Logged out if appropriate.\033[0m" // green
#define SUCCESSFUL_QUIT "\033[1;32m221: Successful Quit.\033[0m"
// green
#define SYNTAX_ERROR "\033[1;31m501: Syntax error in parameters
or arguments.\033[0m" // red
#define INVALID_PATH "\033[1;31m404: No such directory\033[0m"
// red
#define NEED_FOR_ACCOUNT "\033[1;31m332: Need account for
login.\033[0m" // red
#define INTERNAL_SERVER_ERROR "\033[1;31m500: Error\033[0m" //
red
#define LIST_TRANSFER_DONE "\033[1;32m226: List transfer
done.\033[0m" // green
#define SUCCESSFUL_CHANGE "\033[1;32m250: Successful
change.\033[0m" // green
#define SUCCESSFUL_DOWNLOAD "\033[1;32m226: Successful
download.\033[0m" // green
#define FILE_UNAVAILABLE "\033[1;31m550: File
unavailable.\033[0m" // red
#define ERROR_SENDING_FILE "\033[1;31m451: Error sending
file.\033[0m" // red
#define DONE_SUCCESSFULLY "\033[1;32m200: Ok\033[0m" // green

#define BUFFER_SIZE 1024


/**

    @class FTPSpecification
    @brief Handles FTP commands and communication between the
server and client.
    The FTPSpecification class processes FTP commands received
from the client. All command-specific handler
    functions are utilized within the handler function, which
selects the appropriate handler based on the command
    received from the client.
    The server creates two sockets within the client:
command_socket is used for sending commands to the server, and
    the server returns response codes and corresponding messages
via the same channel. The response code, typically
    represented by the initial digits of the response, should be
processed by the client.
    On the other hand, data_socket is used for transmitting data
from the client to the server. For example, in the
    case of the RETR command, the data following the command
(e.g., "file.txt") is sent via the data socket. From the
server's perspective, all data is sent to the client via the
data socket.
    When working with FTPSpecification.cpp, pay attention to the
order of commands sent from the server to the client,
    as the order of commands impacts how the client should
handle them. Typically, data is first read from the command
    socket and data socket on the server side, processed, and
then response codes are sent back to the client. This
    process may occur multiple times, so ensure that you handle
the sequencing correctly when implementing the client.
    @note This class encapsulates the FTP command processing and
handles the communication between the server and client
    according to the FTP protocol specifications.
```

```
    */
class FTPSpecification {

private:
     std::string current_dir = ".";
     std::string baser_dir = std::filesystem::current_path();
     static std::mutex retr_mutex;

public:
    void handler(char command[], int fcs, int fds);

private:
    void echo_handler(int fcs, int fds);
    void list_handler(int fcs, int fds);
    void cwd_handler(int fcs, int fds);
    void retr_handler(int fcs, int fds);

    void clear_socket_data(int socket_fd);
    std::string parse_current_dir();
    std::string get_client_info(int fcs);
    std::vector<std::string> split_path(const std::string&
path_string);

};

//json_reader.h
#pragma once

#include <iostream>
#include <string>
#include <vector>
#include <fstream>

#define COMMA ','
#define QOUTATION '\"'
#define COLON ':'
#define LBRACK '['
#define RBRACK ']'
#define LBRACE '{'
#define RBRACE '}'

class Json_Reader{
public:
    static std::string get_json(const std::string& path);

    static std::string find_value(std::string json, const
std::string& key);

    static std::vector<std::string> split_array(std::string
array);
};
//ServerCore.h
#pragma once

#include "general.h"
#include "../../ftp_specification/FTPSpecification.h"
#include "../../json_reader/include/json_reader.h"


#include <unistd.h>
#include <string>
#include <vector>
```

```
#include <unordered_map>
#include <semaphore>

#include <sys/socket.h>
#include <arpa/inet.h>
#include <iostream>
#include <netinet/in.h>

/**

    @file configuration.cpp
    @brief Configuration file for the server and client
applications.
    This file contains the necessary configuration settings for
the server and client applications.
    The configuration file must be located in the following
directory for the server and client applications to correctly
retrieve data from it:
        For the server: server_core/include/ServerCore.h
        For the client: client_run/client.cpp
    If you want to change the location of the configuration file
directory, you should navigate to
server_core/include/ServerCore.h
    and modify the value of the PATH_TO_JSON constant.
    Required configuration settings for the client:
        serverPort: The port number of the server.
        localIpAddress: The local IP address of the client.
    Required configuration settings for the server:
        serverPort: The port number of the server.
        localIpAddress: The local IP address of the server.
    Additional configuration settings:
        users: A list of users for the server.
        */

#define PATH_TO_JSON "../server_core/resources/config.json"


#define PACKET_SIZE 1024


/**

    @class ServerClient
    @brief Represents a server-side client entity.
    The ServerClient class represents a client from the server's
perspective. Each instance of this class is created within a
separate thread and dynamically allocated on the heap.
    It is not a member of the ServerCore class, but rather
exists independently within each thread.
    The ServerClient class contains an instance of the
FTPSpecification class, which is provided to the handler
function. The handler function is responsible for processing all
commands sent by the client.
    For more information on the functionality of
FTPSpecification, refer to the corresponding header file.
    @note This class encapsulates the server-side client
behavior and facilitates command processing and communication
with the client.
    */
class ServerClient {
public:
    int command_socket;
```

```
    int data_socket;
    void (*handler)(char command[], int fcs, int fds);
    bool is_authorized = false;
    FTPSpecification* ftp_specification = new
FTPSpecification();

    bool operator==(const ServerClient &other) const {
        return command_socket == other.command_socket;
    }

public:
    void disconnect();
    void connected() const;
    ssize_t get_command_from_client(char buffer[]) const;
    size_t get_data_from_client(char buffer[]);
    void handle_command(char command[]) const;
    void authorize();
    void clear_socket_data(int socket_fd);

};



/**

    @class ServerCore
    @brief The core component of our server application.
    The ServerCore class represents the heart of our server. It
is responsible for managing client threads and creating
instances of the ServerClient class within those threads.
    Each ServerClient object is dynamically allocated on the
heap and exists until the associated thread is terminated, which
occurs upon client disconnection.
    @note This class encapsulates the essential functionality of
the server and serves as a central component for handling client
interactions.
    */
class ServerCore {
private:
    int server_socket;
    int server_port;
    std::string local_ip_address;
    ThreadPool thread_pool;
private:
    void create_bind_listen_sockets();
    void handlingAccept();
public:
    void start();
    void joinLoop();

};
```