//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 возращает 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();

};