Console Messenger

Generated by Doxygen 1.9.8

Глава 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Глава 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Глава 3

Class Documentation

3.1 ClientInfo Struct Reference

Информация о подключенном клиенте.

Public Attributes

- int fd
- std::string id
- std::string connected_to
- bool is_speaking = false
- std::string pending_request_from

3.1.1 Detailed Description

Информация о подключенном клиенте.

Definition at line 47 of file main_server.cpp.

3.1.2 Member Data Documentation

3.1.2.1 connected_to

ClientInfo::connected_to

ID клиента, с которым установлена беседа (пусто, если нет).

Definition at line 50 of file main_server.cpp.

3.1.2.2 fd

ClientInfo::fd

Дескриптор сокета клиента.

Definition at line 48 of file main_server.cpp.

3.1.2.3 id

ClientInfo::id

Идентификатор (Telegram ID) клиента.

Definition at line 49 of file main_server.cpp.

3.1.2.4 is_speaking

ClientInfo::is_speaking = false

Флаг права голоса (кто может отправлять сообщения).

Definition at line 51 of file main_server.cpp.

3.1.2.5 pending_request_from

ClientInfo::pending_request_from

Если не пусто — ID клиента, ожидающего подтверждения соединения.

Definition at line 52 of file main_server.cpp.

The documentation for this struct was generated from the following file:

- server/main_server.cpp

3.2 ServerConf Struct Reference

Параметры подключения к серверу.

Public Attributes

- std::string ip
- int port

3.2.1 Detailed Description

Параметры подключения к серверу.

Definition at line 45 of file main_client.cpp.

3.2.2 Member Data Documentation

3.2.2.1 ip

std::string ServerConf::ip

IPv4-адрес сервера.

Definition at line 46 of file main_client.cpp.

3.2.2.2 port

int ServerConf::port

Порт сервера.

Definition at line 47 of file main_client.cpp.

The documentation for this struct was generated from the following file:

- client/main_client.cpp

Глава 4

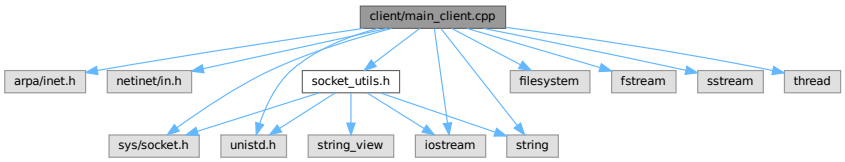
File Documentation

4.1 client/main_client.cpp File Reference

Клиент консольного мессенджера: подключение к серверу и обмен сообщениями.

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>
#include "socket_utils.h"
#include <filesystem>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <thread>
```

Include dependency graph for main_client.cpp:



Classes

- struct ServerConf

  Параметры подключения к серверу.

Functions

- bool valid_ip_port (const std::string &ip, int port)

  Проверить корректность IPv4-адреса и порта.
- ServerConf get_config ()

  Считать или запросить у пользователя настройки сервера.
- void receive_messages (int fd)

  Цикл приёма и вывода сообщений от сервера.
- int main ()

  Точка входа клиентского приложения.

Variables

- constexpr size_t MAX_INPUT = 2000

  Максимально допустимая длина сообщения от пользователя.
- const std::string CFG_DIR = "CLIENT_SETTING"

  Директория для хранения конфигурационного файла.
- const std::string CFG_FILE = "CLIENT_SETTING/ip_port.txt"

  Путь к файлу с настройками (IP и порт сервера).

4.1.1 Detailed Description

Клиент консольного мессенджера: подключение к серверу и обмен сообщениями.

Программа читает конфигурацию сервера (IP и порт), устанавливает TCP-соединение, запускает поток для приёма сообщений и отправляет введённые пользователем строки.

Definition in file main_client.cpp.

4.1.2 Function Documentation

4.1.2.1 get_config()

ServerConf get_config ( )

Считать или запросить у пользователя настройки сервера.

Если файл с конфигурацией существует, пытается прочитать из него строку в формате "IP:порт". Если данные некорректны или файла нет, запрашивает ввод у пользователя до тех пор, пока не будет введена валидная пара. Сохраняет корректные настройки в файл.

Returns

Настройки сервера в виде ServerConf.

Definition at line 77 of file main_client.cpp.

```
00077                     {
00078     std::filesystem::create_directories(CFG_DIR);
00079     std::ifstream fin(CFG_FILE);
00080     std::string ip;
00081     int port;
00082     bool ok = false;
00083     if (fin) {
00084         std::string line;
00085         std::getline(fin, line);
00086         std::istringstream ss(line);
00087         std::getline(ss, ip, ':');
00088         ss >> port;
00089         ok = valid_ip_port(ip, port);
00090     }
00091     while (!ok) {
00092         std::cout << "Enter server IP: ";
00093         std::cin >> ip;
00094         std::cout << "Enter server port: ";
00095         std::cin >> port;
00096         std::cin.ignore();
00097         ok = valid_ip_port(ip, port);
00098         if (!ok)
00099             std::cout << "Invalid IP or port. Try again.\n";
00100     }
00101     std::ofstream(CFG_FILE, std::ios::trunc) << ip << ':' << port << '\n';
00102     return {ip, port};
00103 }
```

4.1.2.2 main()

int main ( )

Точка входа клиентского приложения.

Получает конфигурацию сервера, устанавливает TCP-соединение, запускает поток для приёма сообщений и в цикле отправляет введённые пользователем сообщения.

Returns

Код завершения (0 при успехе, иначе 1).

Definition at line 139 of file main_client.cpp.

```
00139         {
00140     ServerConf conf = get_config();
00141 
00142     int sock = socket(AF_INET, SOCK_STREAM, 0);
00143     if (sock == -1) {
00144         perror("socket");
00145         return 1;
00146     }
00147 
00148     sockaddr_in addr{};
00149     addr.sin_family = AF_INET;
00150     addr.sin_port = htons(conf.port);
00151     inet_pton(AF_INET, conf.ip.c_str(), &addr.sin_addr);
00152 
00153     if (connect(sock, (sockaddr*)&addr, sizeof(addr)) < 0) {
00154         perror("connect");
00155         return 1;
00156     }
00157 
00158     std::thread(receive_messages, sock).detach();
00159 
00160     std::string input;
00161     while (std::getline(std::cin, input)) {
00162         if (input.empty())
00163             continue;
00164         if (input.size() > MAX_INPUT) {
00165             std::cout « "Message longer than 2000 characters. Split it.\n";
00166             continue;
00167         }
00168         if (input == "/exit") {
00169             send_line(sock, "/exit");
00170             std::cout « "\nExiting...\n";
00171             break;
00172         }
00173         send_line(sock, input);
00174     }
00175     close(sock);
00176     return 0;
00177 }
```

4.1.2.3 receive_messages()

void receive_messages (
int fd)

Цикл приёма и вывода сообщений от сервера.

Читает строки из сокета через recv_line() до разрыва соединения. Выводит каждую строку на консоль. При получении специального маркера "*ENDM*" отображает приглашение ввода.

Parameters

fd	Дескриптор подключённого сокета сервера.

Definition at line 114 of file main_client.cpp.

```
00114                                {
00115     std::string line;
00116     while (recv_line(fd, line)) {
00117         if (line.empty())
00118             continue;
00119         if (line == "*ENDM*") {
00120             std::cout « "[You]> " « std::flush;
00121             continue;
00122         }
00123         std::cout « line « '\n';
00124     }
00125     std::cout « "\nDisconnected from server.\n";
00126     close(fd);
00127     exit(0);
00128 }
```

4.1.2.4 valid_ip_port()

```
bool valid_ip_port (
                const std::string & ip,
                int port )
```

Проверить корректность IPv4-адреса и порта.

Использует inet_pton() для валидации формата IPv4 и проверяет, что порт находится в диапазоне 1..65535.

Parameters

ip	Строка с IPv4-адресом.
port	Номер порта.

Returns

true если адрес и порт валидны; false в противном случае.

Definition at line 61 of file main_client.cpp.

```
00061                                        {
00062     sockaddr_in tmp{};
00063     return inet_pton(AF_INET, ip.c_str(), &tmp.sin_addr) == 1 && port > 0 && port < 65536;
00064 }
```

4.1.3 Variable Documentation

4.1.3.1 CFG_DIR

```
const std::string CFG_DIR = "CLIENT_SETTING"
```

Директория для хранения конфигурационного файла.

Definition at line 31 of file main_client.cpp.

4.1.3.2 CFG_FILE

```
const std::string CFG_FILE = "CLIENT_SETTING/ip_port.txt"
```

Путь к файлу с настройками (IP и порт сервера).

Definition at line 36 of file main_client.cpp.

4.1.3.3 MAX_INPUT

constexpr size_t MAX_INPUT = 2000 [constexpr]

Максимально допустимая длина сообщения от пользователя.

Definition at line 26 of file main_client.cpp.

4.2 main_client.cpp

Go to the documentation of this file.

```
00001 /**
00002  * @file main_client.cpp
00003  * @brief Клиент консольного мессенджера: подключение к серверу и обмен сообщениями.
00004  *
00005  * Программа читает конфигурацию сервера (IP и порт),
00006  * устанавливает TCP-соединение, запускает поток
00007  * для приёма сообщений и отправляет введённые пользователем строки.
00008  */
00009
00010 #include <arpa/inet.h>
00011 #include <netinet/in.h>
00012 #include <sys/socket.h>
00013 #include <unistd.h>
00014
00015 #include "socket_utils.h"
00016 #include <filesystem>
00017 #include <fstream>
00018 #include <iostream>
00019 #include <sstream>
00020 #include <string>
00021 #include <thread>
00022
00023 /**
00024  * @brief Максимально допустимая длина сообщения от пользователя.
00025  */
00026 constexpr size_t MAX_INPUT = 2000;
00027
00028 /**
00029  * @brief Директория для хранения конфигурационного файла.
00030  */
00031 const std::string CFG_DIR = "CLIENT_SETTING";
00032
00033 /**
00034  * @brief Путь к файлу с настройками (IP и порт сервера).
00035  */
00036 const std::string CFG_FILE = "CLIENT_SETTING/ip_port.txt";
00037
00038 /**
00039  * @struct ServerConf
00040  * @brief Параметры подключения к серверу.
00041  *
00042  * @var ServerConf::ip   IPv4-адрес сервера.
00043  * @var ServerConf::port Порт сервера.
00044  */
00045 struct ServerConf {
00046     std::string ip; /**< IPv4-адрес сервера. */
00047     int port;       /**< Порт сервера. */
00048 };
00049
00050 /**
00051  * @brief Проверить корректность IPv4-адреса и порта.
00052  *
00053  * Использует inet_pton() для валидации формата IPv4
00054  * и проверяет, что порт находится в диапазоне 1..65535.
00055  *
00056  * @param ip    Строка с IPv4-адресом.
00057  * @param port  Номер порта.
00058  * @return true  если адрес и порт валидны;
00059  *         false в противном случае.
00060  */
00061 bool valid_ip_port(const std::string& ip, int port) {
00062     sockaddr_in tmp{};
00063     return inet_pton(AF_INET, ip.c_str(), &tmp.sin_addr) == 1 && port > 0 && port < 65536;
00064 }
00065
00066 /**
00067  * @brief Считать или запросить у пользователя настройки сервера.
```

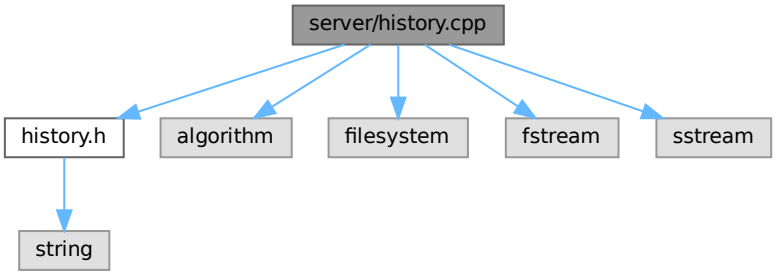
```
00068  *
00069  * Если файл с конфигурацией существует, пытается прочитать из него строку
00070  * в формате "IP:порт". Если данные некорректны или файла нет,
00071  * запрашивает ввод у пользователя до тех пор, пока не будет введена
00072  * валидная пара.
00073  * Сохраняет корректные настройки в файл.
00074  *
00075  * @return Настройки сервера в виде ServerConf.
00076  */
00077 ServerConf get_config() {
00078     std::filesystem::create_directories(CFG_DIR);
00079     std::ifstream fin(CFG_FILE);
00080     std::string ip;
00081     int port;
00082     bool ok = false;
00083     if (fin) {
00084         std::string line;
00085         std::getline(fin, line);
00086         std::istringstream ss(line);
00087         std::getline(ss, ip, ':');
00088         ss » port;
00089         ok = valid_ip_port(ip, port);
00090     }
00091     while (!ok) {
00092         std::cout « "Enter server IP: ";
00093         std::cin » ip;
00094         std::cout « "Enter server port: ";
00095         std::cin » port;
00096         std::cin.ignore();
00097         ok = valid_ip_port(ip, port);
00098         if (!ok)
00099             std::cout « "Invalid IP or port. Try again.\n";
00100     }
00101     std::ofstream(CFG_FILE, std::ios::trunc) « ip « ':' « port « '\n';
00102     return {ip, port};
00103 }
00104
00105 /**
00106  * @brief Цикл приёма и вывода сообщений от сервера.
00107  *
00108  * Читает строки из сокета через recv_line() до разрыва соединения.
00109  * Выводит каждую строку на консоль. При получении специального
00110  * маркера "*ENDM*" отображает приглашение ввода.
00111  *
00112  * @param fd Дескриптор подключённого сокета сервера.
00113  */
00114 void receive_messages(int fd) {
00115     std::string line;
00116     while (recv_line(fd, line)) {
00117         if (line.empty())
00118             continue;
00119         if (line == "*ENDM*") {
00120             std::cout « "[You]> " « std::flush;
00121             continue;
00122         }
00123         std::cout « line « '\n';
00124     }
00125     std::cout « "\nDisconnected from server.\n";
00126     close(fd);
00127     exit(0);
00128 }
00129
00130 /**
00131  * @brief Точка входа клиентского приложения.
00132  *
00133  * Получает конфигурацию сервера, устанавливает TCP-соединение,
00134  * запускает поток для приёма сообщений и в цикле
00135  * отправляет введённые пользователем сообщения.
00136  *
00137  * @return Код завершения (0 при успехе, иначе 1).
00138  */
00139 int main() {
00140     ServerConf conf = get_config();
00141
00142     int sock = socket(AF_INET, SOCK_STREAM, 0);
00143     if (sock == -1) {
00144         perror("socket");
00145         return 1;
00146     }
00147
00148     sockaddr_in addr{};
00149     addr.sin_family = AF_INET;
00150     addr.sin_port = htons(conf.port);
00151     inet_pton(AF_INET, conf.ip.c_str(), &addr.sin_addr);
00152
00153     if (connect(sock, (sockaddr*)&addr, sizeof(addr)) < 0) {
00154         perror("connect");
```

```
00155         return 1;
00156     }
00157
00158     std::thread(receive_messages, sock).detach();
00159
00160     std::string input;
00161     while (std::getline(std::cin, input)) {
00162         if (input.empty())
00163             continue;
00164         if (input.size() > MAX_INPUT) {
00165             std::cout « "Message longer than 2000 characters. Split it.\n";
00166             continue;
00167         }
00168         if (input == "/exit") {
00169             send_line(sock, "/exit");
00170             std::cout « "\nExiting...\n";
00171             break;
00172         }
00173         send_line(sock, input);
00174     }
00175     close(sock);
00176     return 0;
00177 }
```

4.3 server/history.cpp File Reference

Реализация функций для хранения и загрузки истории переписки пользователей.

#include "history.h"
#include <algorithm>
#include <filesystem>
#include <fstream>
#include <sstream>
Include dependency graph for history.cpp:



Functions

- std::string get_history_filename (const std::string &user1, const std::string &user2)

  Построить путь к файлу истории для двух пользователей.
- void ensure_history_folder_exists ()

  Создать папку HISTORY, если она ещё не существует.
- void append_message_to_history (const std::string &user1, const std::string &user2, const std↩ ::string &message)

  Добавить сообщение в конец файла истории для двух пользователей.
- std::string load_history_for_users (const std::string &user1, const std::string &user2)

  Загрузить всю историю переписки между двумя пользователями.

4.3.1 Detailed Description

Реализация функций для хранения и загрузки истории переписки пользователей.

Механизм:

- История для пары пользователей хранится в файле HISTORY/history_<min>_<max>.txt, где <min> и <max> — идентификаторы пользователей в лексикографическом порядке.
- При добавлении сообщения, папка HISTORY создаётся при необходимости, а сообщение дописывается в соответствующий файл.
- При загрузке истории возвращается содержимое файла целиком или пустая строка, если файл не существует.

Definition in file history.cpp.

4.3.2 Function Documentation

4.3.2.1 append_message_to_history()

```
void append_message_to_history (
                const std::string & user1,
                const std::string & user2,
                const std::string & message )
```

Добавить сообщение в конец файла истории для двух пользователей.

Добавить сообщение в историю чата двух пользователей.

Перед записью гарантирует существование папки HISTORY.

Parameters

user1	Идентификатор отправителя или просто один из пользователей чата.
user2	Идентификатор второго пользователя чата.
message	Текст сообщения; может содержать символ новой строки в конце.

Definition at line 60 of file history.cpp.

```
00061                                                   {
00062     ensure_history_folder_exists();
00063     std::ofstream file(get_history_filename(user1, user2), std::ios::app);
00064     if (file) {
00065         file « message;
00066     }
00067 }
```

4.3.2.2 ensure_history_folder_exists()

```
void ensure_history_folder_exists ( )
```

Создать папку HISTORY, если она ещё не существует.

Использует std::filesystem::exists и std::filesystem::create_directory.

Definition at line 45 of file history.cpp.

```
00045                                {
00046     if (!fs::exists("HISTORY")) {
00047         fs::create_directory("HISTORY");
00048     }
00049 }
```

4.3.2.3 get_history_filename()

```
std::string get_history_filename (
                const std::string & user1,
                const std::string & user2 )
```

Построить путь к файлу истории для двух пользователей.

Идентификаторы упорядочиваются лексикографически и соединяются через '_', а затем добавляются префикс и суффикс.

Parameters

user1	Идентификатор первого пользователя.
user2	Идентификатор второго пользователя.

Returns

Строка с путём к файлу истории, например "HISTORY/history_alice_bob.txt".

Definition at line 33 of file history.cpp.

```
00033                                                    {
00034     std::string u1 = user1, u2 = user2;
00035     if (u1 > u2)
00036         std::swap(u1, u2);
00037     return "HISTORY/history_" + u1 + "_" + u2 + ".txt";
00038 }
```

4.3.2.4 load_history_for_users()

```
std::string load_history_for_users (
                const std::string & user1,
                const std::string & user2 )
```

Загрузить всю историю переписки между двумя пользователями.

Если файл с историей существует, читает его содержимое через std::ostringstream.

Parameters

user1	Идентификатор первого пользователя.
user2	Идентификатор второго пользователя.

Returns

Строка с полным содержимым файла истории; пустая строка, если файл недоступен.

Definition at line 78 of file history.cpp.

```
00078                                                    {
00079     ensure_history_folder_exists();
00080     std::ifstream file(get_history_filename(user1, user2));
00081     std::ostringstream ss;
00082     if (file) {
00083         ss << file.rdbuf();
00084     }
00085     return ss.str();
00086 }
```

4.4 history.cpp

Go to the documentation of this file.

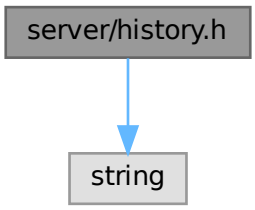
```
00001 /**
00002  * @file history.cpp
00003  * @brief Реализация функций для хранения и загрузки истории переписки пользователей.
00004  *
00005  * Механизм:
00006  * - История для пары пользователей хранится в файле HISTORY/history_<min>_<max>.txt,
00007  *   где <min> и <max> — идентификаторы пользователей в лексикографическом порядке.
00008  * - При добавлении сообщения, папка HISTORY создаётся при необходимости,
00009  *   а сообщение дописывается в соответствующий файл.
00010  * - При загрузке истории возвращается содержимое файла целиком или пустая строка,
00011  *   если файл не существует.
00012  */
00013
00014 #include "history.h"
00015
00016 #include <algorithm>
00017 #include <filesystem>
00018 #include <fstream>
00019 #include <sstream>
00020
00021 namespace fs = std::filesystem;
00022
00023 /**
00024  * @brief Построить путь к файлу истории для двух пользователей.
00025  *
00026  * Идентификаторы упорядочиваются лексикографически и соединяются
00027  * через '_', а затем добавляются префикс и суффикс.
00028  *
00029  * @param user1 Идентификатор первого пользователя.
00030  * @param user2 Идентификатор второго пользователя.
00031  * @return Строка с путём к файлу истории, например "HISTORY/history_alice_bob.txt".
00032  */
00033 std::string get_history_filename(const std::string& user1, const std::string& user2) {
00034     std::string u1 = user1, u2 = user2;
00035     if (u1 > u2)
00036         std::swap(u1, u2);
00037     return "HISTORY/history_" + u1 + "_" + u2 + ".txt";
00038 }
00039
00040 /**
00041  * @brief Создать папку HISTORY, если она ещё не существует.
00042  *
00043  * Использует std::filesystem::exists и std::filesystem::create_directory.
00044  */
00045 void ensure_history_folder_exists() {
00046     if (!fs::exists("HISTORY")) {
00047         fs::create_directory("HISTORY");
00048     }
00049 }
00050
00051 /**
00052  * @brief Добавить сообщение в конец файла истории для двух пользователей.
00053  *
00054  * Перед записью гарантирует существование папки HISTORY.
00055  *
00056  * @param user1  Идентификатор отправителя или просто один из пользователей чата.
00057  * @param user2  Идентификатор второго пользователя чата.
00058  * @param message Текст сообщения; может содержать символ новой строки в конце.
00059  */
00060 void append_message_to_history(const std::string& user1, const std::string& user2,
00061                                const std::string& message) {
```

```
00062     ensure_history_folder_exists();
00063     std::ofstream file(get_history_filename(user1, user2), std::ios::app);
00064     if (file) {
00065         file << message;
00066     }
00067 }
00068
00069 /**
00070  * @brief Загрузить всю историю переписки между двумя пользователями.
00071  *
00072  * Если файл с историей существует, читает его содержимое через std::ostringstream.
00073  *
00074  * @param user1 Идентификатор первого пользователя.
00075  * @param user2 Идентификатор второго пользователя.
00076  * @return Строка с полным содержимым файла истории; пустая строка, если файл недоступен.
00077  */
00078 std::string load_history_for_users(const std::string& user1, const std::string& user2) {
00079     ensure_history_folder_exists();
00080     std::ifstream file(get_history_filename(user1, user2));
00081     std::ostringstream ss;
00082     if (file) {
00083         ss << file.rdbuf();
00084     }
00085     return ss.str();
00086 }
```

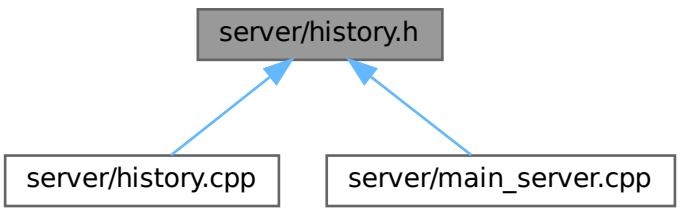
4.5 server/history.h File Reference

Работа с историей переписки между двумя пользователями.

#include <string>
Include dependency graph for history.h:



This graph shows which files directly or indirectly include this file:



Functions

- void append_message_to_history (const std::string &user1, const std::string &user2, const std::string &message)

  Добавить сообщение в историю чата двух пользователей.
- std::string load_history_for_users (const std::string &user1, const std::string &user2)

  Загрузить всю историю переписки между двумя пользователями.

4.5.1 Detailed Description

Работа с историей переписки между двумя пользователями.

Механизм:

- История хранится в каталоге HISTORY.
- Название файла истории для пары пользователей формируется лексикографически: HISTORY/<min>___<max>.txt.

Definition in file history.h.

4.5.2 Function Documentation

4.5.2.1 append_message_to_history()

```
void append_message_to_history (
                const std::string & user1,
                const std::string & user2,
                const std::string & message )
```

Добавить сообщение в историю чата двух пользователей.

Создаёт каталог HISTORY при необходимости и дописывает message в файл для пары пользователей.

Parameters

user1	Идентификатор первого пользователя.
user2	Идентификатор второго пользователя.
message	Текст сообщения, включая символ новой строки.

Добавить сообщение в историю чата двух пользователей.

Перед записью гарантирует существование папки HISTORY.

Parameters

user1	Идентификатор отправителя или просто один из пользователей чата.
user2	Идентификатор второго пользователя чата.
message	Текст сообщения; может содержать символ новой строки в конце.

Definition at line 60 of file history.cpp.

```
00061                                                                   {
00062     ensure_history_folder_exists();
00063     std::ofstream file(get_history_filename(user1, user2), std::ios::app);
00064     if (file) {
00065         file << message;
00066     }
00067 }
```

4.5.2.2 load_history_for_users()

```
std::string load_history_for_users (
                const std::string & user1,
                const std::string & user2 )
```

Загрузить всю историю переписки между двумя пользователями.

Открывает файл HISTORY/<min>___<max>.txt и возвращает его содержимое.

Parameters

user1	Идентификатор первого пользователя.
user2	Идентификатор второго пользователя.

Returns

Строка с полным содержимым истории; пустая строка, если файл не существует или пуст.

Если файл с историей существует, читает его содержимое через std::ostringstream.

Parameters

user1	Идентификатор первого пользователя.
user2	Идентификатор второго пользователя.

Returns

Строка с полным содержимым файла истории; пустая строка, если файл недоступен.

Definition at line 78 of file history.cpp.

```
00078                                                                   {
00079     ensure_history_folder_exists();
00080     std::ifstream file(get_history_filename(user1, user2));
00081     std::ostringstream ss;
00082     if (file) {
00083         ss << file.rdbuf();
00084     }
00085     return ss.str();
00086 }
```

4.6 history.h

Go to the documentation of this file.

```
00001 /**
00002  * @file history.h
```

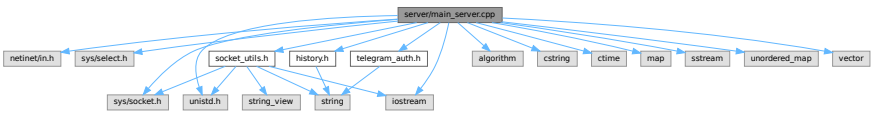
```
00003  * @brief Работа с историей переписки между двумя пользователями.
00004  *
00005  * Механизм:
00006  * - История хранится в каталоге HISTORY.
00007  * - Название файла истории для пары пользователей формируется
00008  *   лексикографически: HISTORY/<min>___<max>.txt.
00009  */
00010
00011 #ifndef HISTORY_H
00012 #define HISTORY_H
00013
00014 #include <string>
00015
00016 /**
00017  * @brief Добавить сообщение в историю чата двух пользователей.
00018  *
00019  * Создаёт каталог HISTORY при необходимости и дописывает @p message
00020  * в файл для пары пользователей.
00021  *
00022  * @param user1 Идентификатор первого пользователя.
00023  * @param user2 Идентификатор второго пользователя.
00024  * @param message Текст сообщения, включая символ новой строки.
00025  */
00026 void append_message_to_history(const std::string& user1, const std::string& user2,
00027                                const std::string& message);
00028
00029 /**
00030  * @brief Загрузить всю историю переписки между двумя пользователями.
00031  *
00032  * Открывает файл HISTORY/<min>___<max>.txt и возвращает его содержимое.
00033  *
00034  * @param user1 Идентификатор первого пользователя.
00035  * @param user2 Идентификатор второго пользователя.
00036  * @return Строка с полным содержимым истории; пустая строка,
00037  *         если файл не существует или пуст.
00038  */
00039 std::string load_history_for_users(const std::string& user1, const std::string& user2);
00040
00041 #endif  // HISTORY_H
```

4.7 server/main_server.cpp File Reference

Реализация сервера консольного мессенджера.

```
#include <netinet/in.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <unistd.h>
#include "telegram_auth.h"
#include "history.h"
#include "socket_utils.h"
#include <algorithm>
#include <cstring>
#include <ctime>
#include <iostream>
#include <map>
#include <sstream>
#include <unordered_map>
#include <vector>
```

Include dependency graph for main_server.cpp:



Classes

- struct ClientInfo

  Информация о подключенном клиенте.

Functions

- std::string get_timestamp ()

  Получить текущую дату и время.
- void disconnect_client (int fd, fd_set &master_fds)

  Отключить клиента и очистить его данные.
- void handle_client_command (int fd, const std::string &msg, fd_set &master_fds)

  Обработать команду клиента в режиме диалога.
- void handle_pending_response (int fd, const std::string &msg)

  Обработать ответ клиента на запрос соединения.
- int main ()

  Точка входа сервера.

Variables

- constexpr int PORT = 9090

  Порт, на котором слушает сервер.

4.7.1 Detailed Description

Реализация сервера консольного мессенджера.

Сервер принимает подключения клиентов по TCP, обеспечивает авторизацию через Telegram-коды, обработку команд клиентов (/connect, /vote, /end, /help, /exit, /shutdown), передачу сообщений между участниками и хранение истории.

Definition in file main_server.cpp.

4.7.2 Function Documentation

4.7.2.1 disconnect_client()

```
void disconnect_client (
            int fd,
            fd_set & master_fds )
```

Отключить клиента и очистить его данные.

Завершает соединение, удаляет из наборов клиентов, уведомляет партнера беседы.

Parameters

fd	Дескриптор сокета клиента для отключения.
master_fds	Ссылка на набор файловых дескрипторов select().

Definition at line 85 of file main_server.cpp.

```
00085                                       {
00086     if (clients.count(fd)) {
00087         std::string id = clients[fd].id;
00088         std::string connected_to = clients[fd].connected_to;
00089         std::cout « "\nDisconnecting client: " « id « " (fd: " « fd « ")\n";
00090 
00091         if (!connected_to.empty() && id_to_fd.count(connected_to)) {
00092             int target_fd = id_to_fd[connected_to];
00093             clients[target_fd].connected_to.clear();
00094             clients[target_fd].is_speaking = false;
00095             const std::string msg = "\nYour conversation partner has left the chat.\n";
00096             send_packet(target_fd, msg.c_str());
00097         }
00098 
00099         clients.erase(fd);
00100         id_to_fd.erase(id);
00101         FD_CLR(fd, &master_fds);
00102         close(fd);
00103     }
00104 }
```

4.7.2.2 get_timestamp()

```
std::string get_timestamp ( )
```

Получить текущую дату и время.

Возвращает строку в формате "YYYY-MM-DD HH:MM".

Returns

Форматированная метка времени.

Definition at line 69 of file main_server.cpp.

```
00069                           {
00070     time_t now = time(nullptr);
00071     char buf[20];
00072     strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M", localtime(&now));
00073     return std::string(buf);
00074 }
```

4.7.2.3 handle_client_command()

```
void handle_client_command (
                int fd,
                const std::string & msg,
                fd_set & master_fds )
```

Обработать команду клиента в режиме диалога.

Поддерживаемые команды:

- /connect <ID>
- /vote
- /end
- /help
- /exit

Parameters

fd	Дескриптор сокета отправителя.
msg	Текст команды (без завершающего).
master_fds	Набор дескрипторов select() для обновления.

Definition at line 120 of file main_server.cpp.

```
00120                                                                                                 {
00121     if (msg.rfind("/connect ", 0) == 0) {
00122         std::string target_id = msg.substr(9);
00123         if (id_to_fd.count(target_id)) {
00124             int target_fd = id_to_fd[target_id];
00125 
00126             if (!clients[target_fd].pending_request_from.empty()) {
00127                 send_packet(fd, "User is busy with another request.\n");
00128                 return;
00129             }
00130 
00131             if (!clients[target_fd].connected_to.empty()) {
00132                 const std::string notice = "\nUser '" + clients[fd].id +
00133                                            "' attempted to connect to you, but you are "
00134                                            "already in a conversation.\n";
00135                 send_packet(target_fd, notice.c_str());
00136                 send_packet(fd, "User is already connected.\n");
00137                 return;
00138             }
00139 
00140             clients[target_fd].pending_request_from = clients[fd].id;
00141             const std::string prompt = "\nUser '" + clients[fd].id + "' wants to connect. Accept? (yes/no)\n";
00142             send_packet(target_fd, prompt.c_str());
00143         } else {
00144             send_packet(fd, "User not found.\n");
00145         }
00146     } else if (msg == "/vote") {
00147         if (clients[fd].is_speaking) {
00148             std::string target_id = clients[fd].connected_to;
00149             if (!target_id.empty() && id_to_fd.count(target_id)) {
00150                 int target_fd = id_to_fd[target_id];
00151                 clients[fd].is_speaking = false;
00152                 clients[target_fd].is_speaking = true;
00153                 send_all(fd, "You passed the microphone.\n");
00154                 send_packet(target_fd, "You are now speaking.\n");
00155             } else {
00156                 send_packet(fd, "No connected client to pass speaking right.\n");
00157             }
00158         } else {
00159             send_packet(fd, "You are not the current speaker.\n");
00160         }
00161     } else if (msg == "/end") {
00162         std::string partner_id = clients[fd].connected_to;
00163         if (!partner_id.empty() && id_to_fd.count(partner_id)) {
00164             int partner_fd = id_to_fd[partner_id];
00165             clients[partner_fd].connected_to.clear();
00166             clients[partner_fd].is_speaking = false;
00167             send_packet(partner_fd, "\nYour conversation partner has ended the chat.\n");
00168         }
00169         clients[fd].connected_to.clear();
00170         clients[fd].is_speaking = false;
00171         send_packet(fd, "You have left the conversation.\n");
00172     } else if (msg == "/help") {
00173         const std::string help =
00174             "Available commands:\n"
00175             "/connect <ID> - request chat with user\n"
00176             "/vote         - pass speaker role\n"
00177             "/end          - end current conversation\n"
00178             "/exit         - exit the chat completely\n"
00179             "/help         - show this message\n";
00180         send_packet(fd, help.c_str());
00181     } else if (msg == "/exit") {
00182         disconnect_client(fd, master_fds);
00183     } else {
00184         send_packet(fd, "Only /connect <ID>, /vote, /end, /exit, /help are allowed.\n");
00185     }
00186 }
```

4.7.2.4 handle_pending_response()

void handle_pending_response (

int fd,
const std::string & msg)

Обработать ответ клиента на запрос соединения.

Если клиент ранее отправил /connect и ожидает ответа, эта функция устанавливает связь и пересылает историю.

Parameters

fd	Дескриптор сокета отвечающего клиента.
msg	Сообщение-ответ ("yes"/"no").

Definition at line 197 of file main_server.cpp.

```
00197                                                  {
00198     ClientInfo& responder = clients[fd];
00199     if (responder.pending_request_from.empty())
00200         return;
00201 
00202     std::string requester_id = responder.pending_request_from;
00203     responder.pending_request_from.clear();
00204 
00205     if (!id_to_fd.count(requester_id)) {
00206         send_packet(fd, "Requester disconnected.\n");
00207         return;
00208     }
00209 
00210     int requester_fd = id_to_fd[requester_id];
00211     if (msg == "yes") {
00212         std::cout « "Clients connected: " « responder.id « " <-> " « requester_id « std::endl;
00213         responder.connected_to = requester_id;
00214         clients[requester_fd].connected_to = responder.id;
00215         clients[requester_fd].is_speaking = true;
00216 
00217         std::string history = load_history_for_users(responder.id, requester_id);
00218         if (!history.empty()) {
00219             send_all(fd, "Chat history:\n");
00220             send_all(fd, history.c_str());
00221             send_all(requester_fd, "Chat history:\n");
00222             send_all(requester_fd, history.c_str());
00223         }
00224         send_packet(requester_fd, "Connection accepted. You are now speaking.\n");
00225         send_all(fd, "Connection established. You are a listener.\n");
00226     } else {
00227         send_packet(requester_fd, "Connection rejected.\n");
00228         send_packet(fd, "Connection declined.\n");
00229     }
00230 }
```

4.7.2.5 main()

int main ( )

Точка входа сервера.

Запускает прослушивание порта, обрабатывает подключения и команды до получения /shutdown.

Returns

0 при корректном завершении, иначе код ошибки.

Definition at line 240 of file main_server.cpp.

```
00240           {
00241     ensure_bot_token();
00242 
00243     int listener = socket(AF_INET, SOCK_STREAM, 0);
00244     if (listener == -1) {
```

```
00245         perror("socket");
00246         return 1;
00247     }
00248
00249     int opt = 1;
00250     setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
00251
00252     sockaddr_in server_addr{};
00253     server_addr.sin_family = AF_INET;
00254     server_addr.sin_port = htons(PORT);
00255     server_addr.sin_addr.s_addr = INADDR_ANY;
00256
00257     if (bind(listener, (sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
00258         perror("bind");
00259         return 1;
00260     }
00261
00262     listen(listener, SOMAXCONN);
00263     std::cout « "Server listening on port " « PORT « std::endl;
00264
00265     fd_set master_fds, read_fds;
00266     FD_ZERO(&master_fds);
00267     FD_SET(listener, &master_fds);
00268     FD_SET(STDIN_FILENO, &master_fds);
00269     int fd_max = listener;
00270
00271     while (true) {
00272         read_fds = master_fds;
00273         if (select(fd_max + 1, &read_fds, nullptr, nullptr, nullptr) == -1) {
00274             perror("select");
00275             break;
00276         }
00277
00278         for (int fd = 0; fd <= fd_max; ++fd) {
00279             if (!FD_ISSET(fd, &read_fds))
00280                 continue;
00281
00282             if (fd == STDIN_FILENO) {
00283                 std::string cmd;
00284                 std::getline(std::cin, cmd);
00285                 if (cmd == "/shutdown") {
00286                     std::cout « "Shutting down server...\n";
00287                     for (auto& [cfd, info] : clients)
00288                         send_all(cfd, "\nServer is shutting down.\n");
00289                     for (auto& [cfd, info] : clients)
00290                         close(cfd);
00291                     close(listener);
00292                     std::cout « "Server stopped.\n";
00293                     return 0;
00294                 }
00295                 continue;
00296             }
00297
00298             if (fd == listener) {
00299                 sockaddr_in client_addr{};
00300                 socklen_t addrlen = sizeof(client_addr);
00301                 int client_fd = accept(listener, (sockaddr*)&client_addr, &addrlen);
00302                 if (client_fd != -1) {
00303                     std::cout « "New client connected, fd: " « client_fd « std::endl;
00304                     FD_SET(client_fd, &master_fds);
00305                     fd_max = std::max(fd_max, client_fd);
00306                     const char* ask_id = "Enter your ID\n";
00307                     send_packet(client_fd, ask_id);
00308                 }
00309             } else {
00310                 std::string msg;
00311                 if (!recv_line(fd, msg)) {
00312                     disconnect_client(fd, master_fds);
00313                     continue;
00314                 }
00315
00316                 if (clients.count(fd) == 0 && !pending_auth.count(fd)) {
00317                     std::string chat_id = msg;
00318                     if (chat_id.empty()) {
00319                         send_packet(fd, "Chat ID cannot be empty. Try again\n");
00320                         continue;
00321                     }
00322
00323                     std::string code = generate_auth_code();
00324                     if (send_telegram_code(chat_id, code)) {
00325                         pending_auth[fd] = chat_id;
00326                         const char* sent = "Telegram code sent. Enter the code to log in\n";
00327                         send_packet(fd, sent);
00328                     } else {
00329                         send_packet(fd,
00330                                     "Failed to send Telegram message.\nUse command /exit to "
00331                                     "exit.\nCheck the telegram ID and write it again");
```

```
00332                 }
00333             }
00334
00335             else if (pending_auth.count(fd)) {
00336                 std::string entered_code = msg;
00337                 std::string chat_id = pending_auth[fd];
00338                 if (verify_auth_code(chat_id, entered_code)) {
00339                     if (id_to_fd.count(chat_id)) {
00340                         int old_fd = id_to_fd[chat_id];
00341                         send_packet(old_fd, "\nYou have been logged out (second login detected).\n");
00342                         disconnect_client(old_fd, master_fds);
00343                     }
00344
00345                     clients[fd] = ClientInfo{fd, chat_id};
00346                     std::cout « "Client authorized: " « chat_id « " (fd: " « fd « ")" « std::endl;
00347                     id_to_fd[chat_id] = fd;
00348                     pending_auth.erase(fd);
00349
00350                     std::string welcome =
00351                         "Welcome, " + chat_id + "! Use /connect <ID>, /vote, /end, /exit, /help\n";
00352                     send_packet(fd, welcome.c_str());
00353                 } else {
00354                     send_packet(fd, "Incorrect code. Try again\n");
00355                 }
00356             }
00357
00358             else if (!clients[fd].pending_request_from.empty()) {
00359                 handle_pending_response(fd, msg);
00360             }
00361
00362             else if (!msg.empty() && msg[0] == '/') {
00363                 handle_client_command(fd, msg, master_fds);
00364             }
00365
00366             else {
00367                 if (clients[fd].connected_to.empty()) {
00368                     send_packet(fd,
00369                                 "You are not in a conversation.\nUse /connect <ID> to "
00370                                 "start chatting.\n");
00371                     continue;
00372                 }
00373                 if (!clients[fd].is_speaking) {
00374                     send_all(fd,
00375                             "You cannot send messages unless you're the current "
00376                             "speaker.\n");
00377                     continue;
00378                 }
00379
00380                 std::string target_id = clients[fd].connected_to;
00381                 if (!target_id.empty() && id_to_fd.count(target_id)) {
00382                     int target_fd = id_to_fd[target_id];
00383                     std::string timestamp = get_timestamp();
00384                     std::string sender = clients[fd].id;
00385                     std::string text = "[" + timestamp + "] " + sender + ": " + msg + "\n";
00386                     send_all(target_fd, text.c_str());
00387                     append_message_to_history(sender, target_id, text);
00388                 } else {
00389                     send_packet(fd, "Not connected. Use /connect <ID>\n");
00390                 }
00391             }
00392         }
00393     }
00394 }
00395
00396 close(listener);
00397 return 0;
00398 }
```

4.7.3 Variable Documentation

4.7.3.1 PORT

constexpr int PORT = 9090 [constexpr]

Порт, на котором слушает сервер.

Definition at line 30 of file main_server.cpp.

4.8 main_server.cpp

Go to the documentation of this file.

```
00001 /**
00002  * @file main_server.cpp
00003  * @brief Реализация сервера консольного мессенджера.
00004  *
00005  * Сервер принимает подключения клиентов по TCP, обеспечивает
00006  * авторизацию через Telegram-коды, обработку команд клиентов
00007  * (/connect, /vote, /end, /help, /exit, /shutdown),
00008  * передачу сообщений между участниками и хранение истории.
00009  */
00010
00011 #include <netinet/in.h>
00012 #include <sys/select.h>
00013 #include <sys/socket.h>
00014 #include <unistd.h>
00015
00016 #include "telegram_auth.h"
00017
00018 #include "history.h"
00019 #include "socket_utils.h"
00020 #include <algorithm>
00021 #include <cstring>
00022 #include <ctime>
00023 #include <iostream>
00024 #include <map>
00025 #include <sstream>
00026 #include <unordered_map>
00027 #include <vector>
00028
00029 /// Порт, на котором слушает сервер.
00030 constexpr int PORT = 9090;
00031
00032 /**
00033  * @struct ClientInfo
00034  * @brief Информация о подключенном клиенте.
00035  *
00036  * @var ClientInfo::fd
00037  * Дескриптор сокета клиента.
00038  * @var ClientInfo::id
00039  * Идентификатор (Telegram ID) клиента.
00040  * @var ClientInfo::connected_to
00041  * ID клиента, с которым установлена беседа (пусто, если нет).
00042  * @var ClientInfo::is_speaking
00043  * Флаг права голоса (кто может отправлять сообщения).
00044  * @var ClientInfo::pending_request_from
00045  * Если не пусто — ID клиента, ожидающего подтверждения соединения.
00046  */
00047 struct ClientInfo {
00048     int fd;
00049     std::string id;
00050     std::string connected_to;
00051     bool is_speaking = false;
00052     std::string pending_request_from;
00053 };
00054
00055 /// Карта: дескриптор сокета -> информация о клиенте.
00056 static std::unordered_map<int, ClientInfo> clients;
00057 /// Карта: Telegram ID клиента -> дескриптор сокета.
00058 static std::unordered_map<std::string, int> id_to_fd;
00059 /// Карта: дескриптор сокета -> Telegram ID (ожидающие код).
00060 static std::unordered_map<int, std::string> pending_auth;
00061
00062 /**
00063  * @brief Получить текущую дату и время.
00064  *
00065  * Возвращает строку в формате "YYYY-MM-DD HH:MM".
00066  *
00067  * @return Форматированная метка времени.
00068  */
00069 std::string get_timestamp() {
00070     time_t now = time(nullptr);
00071     char buf[20];
00072     strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M", localtime(&now));
00073     return std::string(buf);
00074 }
00075
00076 /**
00077  * @brief Отключить клиента и очистить его данные.
00078  *
00079  * Завершает соединение, удаляет из наборов клиентов,
00080  * уведомляет партнера беседы.
00081  *
00082  * @param fd Дескриптор сокета клиента для отключения.
```

```
00083  * @param master_fds Ссылка на набор файловых дескрипторов select().
00084  */
00085 void disconnect_client(int fd, fd_set& master_fds) {
00086     if (clients.count(fd)) {
00087         std::string id = clients[fd].id;
00088         std::string connected_to = clients[fd].connected_to;
00089         std::cout « "\nDisconnecting client: " « id « " (fd: " « fd « ")\n";
00090 
00091         if (!connected_to.empty() && id_to_fd.count(connected_to)) {
00092             int target_fd = id_to_fd[connected_to];
00093             clients[target_fd].connected_to.clear();
00094             clients[target_fd].is_speaking = false;
00095             const std::string msg = "\nYour conversation partner has left the chat.\n";
00096             send_packet(target_fd, msg.c_str());
00097         }
00098 
00099         clients.erase(fd);
00100         id_to_fd.erase(id);
00101         FD_CLR(fd, &master_fds);
00102         close(fd);
00103     }
00104 }
00105 
00106 /**
00107  * @brief Обработать команду клиента в режиме диалога.
00108  *
00109  * Поддерживаемые команды:
00110  *  - /connect <ID>
00111  *  - /vote
00112  *  - /end
00113  *  - /help
00114  *  - /exit
00115  *
00116  * @param fd   Дескриптор сокета отправителя.
00117  * @param msg  Текст команды (без завершающего \n).
00118  * @param master_fds Набор дескрипторов select() для обновления.
00119  */
00120 void handle_client_command(int fd, const std::string& msg, fd_set& master_fds) {
00121     if (msg.rfind("/connect ", 0) == 0) {
00122         std::string target_id = msg.substr(9);
00123         if (id_to_fd.count(target_id)) {
00124             int target_fd = id_to_fd[target_id];
00125 
00126             if (!clients[target_fd].pending_request_from.empty()) {
00127                 send_packet(fd, "User is busy with another request.\n");
00128                 return;
00129             }
00130 
00131             if (!clients[target_fd].connected_to.empty()) {
00132                 const std::string notice = "\nUser '" + clients[fd].id +
00133                                            "' attempted to connect to you, but you are "
00134                                            "already in a conversation.\n";
00135                 send_packet(target_fd, notice.c_str());
00136                 send_packet(fd, "User is already connected.\n");
00137                 return;
00138             }
00139 
00140             clients[target_fd].pending_request_from = clients[fd].id;
00141             const std::string prompt = "\nUser '" + clients[fd].id + "' wants to connect. Accept? (yes/no)\n";
00142             send_packet(target_fd, prompt.c_str());
00143         } else {
00144             send_packet(fd, "User not found.\n");
00145         }
00146     } else if (msg == "/vote") {
00147         if (clients[fd].is_speaking) {
00148             std::string target_id = clients[fd].connected_to;
00149             if (!target_id.empty() && id_to_fd.count(target_id)) {
00150                 int target_fd = id_to_fd[target_id];
00151                 clients[fd].is_speaking = false;
00152                 clients[target_fd].is_speaking = true;
00153                 send_all(fd, "You passed the microphone.\n");
00154                 send_packet(target_fd, "You are now speaking.\n");
00155             } else {
00156                 send_packet(fd, "No connected client to pass speaking right.\n");
00157             }
00158         } else {
00159             send_packet(fd, "You are not the current speaker.\n");
00160         }
00161     } else if (msg == "/end") {
00162         std::string partner_id = clients[fd].connected_to;
00163         if (!partner_id.empty() && id_to_fd.count(partner_id)) {
00164             int partner_fd = id_to_fd[partner_id];
00165             clients[partner_fd].connected_to.clear();
00166             clients[partner_fd].is_speaking = false;
00167             send_packet(partner_fd, "\nYour conversation partner has ended the chat.\n");
00168         }
00169         clients[fd].connected_to.clear();
```

```
00170         clients[fd].is_speaking = false;
00171         send_packet(fd, "You have left the conversation.\n");
00172     } else if (msg == "/help") {
00173         const std::string help =
00174             "Available commands:\n"
00175             "/connect <ID> - request chat with user\n"
00176             "/vote         - pass speaker role\n"
00177             "/end          - end current conversation\n"
00178             "/exit         - exit the chat completely\n"
00179             "/help         - show this message\n";
00180         send_packet(fd, help.c_str());
00181     } else if (msg == "/exit") {
00182         disconnect_client(fd, master_fds);
00183     } else {
00184         send_packet(fd, "Only /connect <ID>, /vote, /end, /exit, /help are allowed.\n");
00185     }
00186 }
00187
00188 /**
00189  * @brief Обработать ответ клиента на запрос соединения.
00190  *
00191  * Если клиент ранее отправил /connect и ожидает ответа,
00192  * эта функция устанавливает связь и пересылает историю.
00193  *
00194  * @param fd  Дескриптор сокета отвечающего клиента.
00195  * @param msg Сообщение-ответ ("yes"/"no").
00196  */
00197 void handle_pending_response(int fd, const std::string& msg) {
00198     ClientInfo& responder = clients[fd];
00199     if (responder.pending_request_from.empty())
00200         return;
00201
00202     std::string requester_id = responder.pending_request_from;
00203     responder.pending_request_from.clear();
00204
00205     if (!id_to_fd.count(requester_id)) {
00206         send_packet(fd, "Requester disconnected.\n");
00207         return;
00208     }
00209
00210     int requester_fd = id_to_fd[requester_id];
00211     if (msg == "yes") {
00212         std::cout « "Clients connected: " « responder.id « " <-> " « requester_id « std::endl;
00213         responder.connected_to = requester_id;
00214         clients[requester_fd].connected_to = responder.id;
00215         clients[requester_fd].is_speaking = true;
00216
00217         std::string history = load_history_for_users(responder.id, requester_id);
00218         if (!history.empty()) {
00219             send_all(fd, "Chat history:\n");
00220             send_all(fd, history.c_str());
00221             send_all(requester_fd, "Chat history:\n");
00222             send_all(requester_fd, history.c_str());
00223         }
00224         send_packet(requester_fd, "Connection accepted. You are now speaking.\n");
00225         send_all(fd, "Connection established. You are a listener.\n");
00226     } else {
00227         send_packet(requester_fd, "Connection rejected.\n");
00228         send_packet(fd, "Connection declined.\n");
00229     }
00230 }
00231
00232 /**
00233  * @brief Точка входа сервера.
00234  *
00235  * Запускает прослушивание порта,
00236  * обрабатывает подключения и команды до получения /shutdown.
00237  *
00238  * @return 0 при корректном завершении, иначе код ошибки.
00239  */
00240 int main() {
00241     ensure_bot_token();
00242
00243     int listener = socket(AF_INET, SOCK_STREAM, 0);
00244     if (listener == -1) {
00245         perror("socket");
00246         return 1;
00247     }
00248
00249     int opt = 1;
00250     setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
00251
00252     sockaddr_in server_addr{};
00253     server_addr.sin_family = AF_INET;
00254     server_addr.sin_port = htons(PORT);
00255     server_addr.sin_addr.s_addr = INADDR_ANY;
00256
```

```
00257     if (bind(listener, (sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
00258         perror("bind");
00259         return 1;
00260     }
00261 
00262     listen(listener, SOMAXCONN);
00263     std::cout « "Server listening on port " « PORT « std::endl;
00264 
00265     fd_set master_fds, read_fds;
00266     FD_ZERO(&master_fds);
00267     FD_SET(listener, &master_fds);
00268     FD_SET(STDIN_FILENO, &master_fds);
00269     int fd_max = listener;
00270 
00271     while (true) {
00272         read_fds = master_fds;
00273         if (select(fd_max + 1, &read_fds, nullptr, nullptr, nullptr) == -1) {
00274             perror("select");
00275             break;
00276         }
00277 
00278         for (int fd = 0; fd <= fd_max; ++fd) {
00279             if (!FD_ISSET(fd, &read_fds))
00280                 continue;
00281 
00282             if (fd == STDIN_FILENO) {
00283                 std::string cmd;
00284                 std::getline(std::cin, cmd);
00285                 if (cmd == "/shutdown") {
00286                     std::cout « "Shutting down server...\n";
00287                     for (auto& [cfd, info] : clients)
00288                         send_all(cfd, "\nServer is shutting down.\n");
00289                     for (auto& [cfd, info] : clients)
00290                         close(cfd);
00291                     close(listener);
00292                     std::cout « "Server stopped.\n";
00293                     return 0;
00294                 }
00295                 continue;
00296             }
00297 
00298             if (fd == listener) {
00299                 sockaddr_in client_addr{};
00300                 socklen_t addrlen = sizeof(client_addr);
00301                 int client_fd = accept(listener, (sockaddr*)&client_addr, &addrlen);
00302                 if (client_fd != -1) {
00303                     std::cout « "New client connected, fd: " « client_fd « std::endl;
00304                     FD_SET(client_fd, &master_fds);
00305                     fd_max = std::max(fd_max, client_fd);
00306                     const char* ask_id = "Enter your ID\n";
00307                     send_packet(client_fd, ask_id);
00308                 }
00309             } else {
00310                 std::string msg;
00311                 if (!recv_line(fd, msg)) {
00312                     disconnect_client(fd, master_fds);
00313                     continue;
00314                 }
00315 
00316                 if (clients.count(fd) == 0 && !pending_auth.count(fd)) {
00317                     std::string chat_id = msg;
00318                     if (chat_id.empty()) {
00319                         send_packet(fd, "Chat ID cannot be empty. Try again\n");
00320                         continue;
00321                     }
00322 
00323                     std::string code = generate_auth_code();
00324                     if (send_telegram_code(chat_id, code)) {
00325                         pending_auth[fd] = chat_id;
00326                         const char* sent = "Telegram code sent. Enter the code to log in\n";
00327                         send_packet(fd, sent);
00328                     } else {
00329                         send_packet(fd,
00330                                     "Failed to send Telegram message.\nUse command /exit to "
00331                                     "exit.\nCheck the telegram ID and write it again");
00332                     }
00333                 }
00334 
00335                 else if (pending_auth.count(fd)) {
00336                     std::string entered_code = msg;
00337                     std::string chat_id = pending_auth[fd];
00338                     if (verify_auth_code(chat_id, entered_code)) {
00339                         if (id_to_fd.count(chat_id)) {
00340                             int old_fd = id_to_fd[chat_id];
00341                             send_packet(old_fd, "\nYou have been logged out (second login detected).\n");
00342                             disconnect_client(old_fd, master_fds);
00343                         }
```

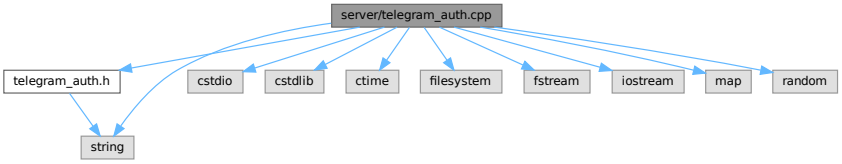
```
00344
00345                     clients[fd] = ClientInfo{fd, chat_id};
00346                     std::cout « "Client authorized: " « chat_id « " (fd: " « fd « ")" « std::endl;
00347                     id_to_fd[chat_id] = fd;
00348                     pending_auth.erase(fd);
00349
00350                     std::string welcome =
00351                         "Welcome, " + chat_id + "! Use /connect <ID>, /vote, /end, /exit, /help\n";
00352                     send_packet(fd, welcome.c_str());
00353                 } else {
00354                     send_packet(fd, "Incorrect code. Try again\n");
00355                 }
00356             }
00357
00358             else if (!clients[fd].pending_request_from.empty()) {
00359                 handle_pending_response(fd, msg);
00360             }
00361
00362             else if (!msg.empty() && msg[0] == '/') {
00363                 handle_client_command(fd, msg, master_fds);
00364             }
00365
00366             else {
00367                 if (clients[fd].connected_to.empty()) {
00368                     send_packet(fd,
00369                                 "You are not in a conversation.\nUse /connect <ID> to "
00370                                 "start chatting.\n");
00371                     continue;
00372                 }
00373                 if (!clients[fd].is_speaking) {
00374                     send_all(fd,
00375                             "You cannot send messages unless you're the current "
00376                             "speaker.\n");
00377                     continue;
00378                 }
00379
00380                 std::string target_id = clients[fd].connected_to;
00381                 if (!target_id.empty() && id_to_fd.count(target_id)) {
00382                     int target_fd = id_to_fd[target_id];
00383                     std::string timestamp = get_timestamp();
00384                     std::string sender = clients[fd].id;
00385                     std::string text = "[" + timestamp + "] " + sender + ": " + msg + "\n";
00386                     send_all(target_fd, text.c_str());
00387                     append_message_to_history(sender, target_id, text);
00388                 } else {
00389                     send_packet(fd, "Not connected. Use /connect <ID>\n");
00390                 }
00391             }
00392         }
00393     }
00394 }
00395
00396 close(listener);
00397 return 0;
00398 }
```

4.9 server/telegram_auth.cpp File Reference

Реализация функций Telegram-аутентификации: генерация, отправка и проверка кодов.

```
#include "telegram_auth.h"
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <map>
#include <random>
#include <string>
```

Include dependency graph for telegram_auth.cpp:



Functions

- void set_bot_token (const std::string &token)

  Установить глобальный токен бота.
- std::string generate_auth_code ()

  Сгенерировать случайный шестизначный код для авторизации.
- void ensure_bot_token ()

  Убедиться, что токен бота загружен из файла.
- bool send_telegram_code (const std::string &chat_id, const std::string &code)

  Отправить код авторизации через Telegram Bot API.
- bool verify_auth_code (const std::string &chat_id, const std::string &code)

  Проверить введённый код авторизации.

Variables

- std::string BOT_TOKEN

  Глобальный токен Telegram-бота, используется при отправке сообщений.
- std::map< std::string, std::string > auth_codes

  Глобальная карта, хранящая соответствие chat_id -> код авторизации.

4.9.1 Detailed Description

Реализация функций Telegram-аутентификации: генерация, отправка и проверка кодов.

Использует CURL (через системный вызов curl) для отправки сообщений Telegram Bot API. Хранит временные коды в глобальной карте auth_codes.

Definition in file telegram_auth.cpp.

4.9.2 Function Documentation

4.9.2.1 ensure_bot_token()

```
void ensure_bot_token ( )
```

Убедиться, что токен бота загружен из файла.

Убедиться, что глобальный токен бота загружен.

Если переменная BOT_TOKEN пуста, пытается считать токен из файла SERVER_SETTINGS/↩ BOT_TOKEN.txt. При отсутствии директории или файла создаёт их. Если после чтения токен остаётся пустым — выводит сообщение об ошибке и завершает программу.

Definition at line 75 of file telegram_auth.cpp.

```
00075                       {
00076     if (!BOT_TOKEN.empty())
00077         return;
00078
00079     namespace fs = std::filesystem;
00080     fs::path dir = "SERVER_SETTINGS";
00081     fs::path file = dir / "BOT_TOKEN.txt";
00082
00083     if (!fs::exists(dir))
00084         fs::create_directories(dir);
00085
00086     if (fs::exists(file)) {
00087         std::ifstream in(file);
00088         std::getline(in, BOT_TOKEN);
00089     } else {
00090         std::ofstream out(file);
00091     }
00092
00093     if (BOT_TOKEN.empty()) {
00094         std::cerr « "[TelegramAuth] Файл " « file
00095                 « " не содержит токен.\n"
00096                   "Добавьте токен в первую строку и перезапустите сервер.\n";
00097         exit(1);
00098     }
00099 }
```

4.9.2.2 generate_auth_code()

```
std::string generate_auth_code ( )
```

Сгенерировать случайный шестизначный код для авторизации.

Использует std::rand(), инициализирует генератор при первом вызове на основе текущего времени.

Returns

Сгенерированный код (строка из 6 цифр).

Definition at line 52 of file telegram_auth.cpp.

```
00052                           {
00053     static bool seeded = false;
00054     if (!seeded) {
00055         std::srand(static_cast<unsigned>(std::time(nullptr)));
00056         seeded = true;
00057     }
00058
00059     std::string digits = "0123456789", code;
00060     for (int i = 0; i < 6; ++i)
00061         code += digits[std::rand() % 10];
00062     return code;
00063 }
```

4.9.2.3 send_telegram_code()

```
bool send_telegram_code (
        const std::string & chat_id,
        const std::string & code )
```

Отправить код авторизации через Telegram Bot API.

Формирует HTTP запрос с помощью системного вызова curl и отправляет код пользователю в чат.

Parameters

chat↩ _id	Идентификатор Telegram-чата.
code	Шестизначный код авторизации.

Returns

true если запрос выполнен успешно (ответ содержит "ok":true), и код сохранён в auth_codes; иначе false.

Definition at line 114 of file telegram_auth.cpp.

```
00114                                                   {
00115     ensure_bot_token();
00116     std::string url = "https://api.telegram.org/bot" + BOT_TOKEN + "/sendMessage";
00117     std::string cmd = "curl -s -X POST \"" + url +
00118                       "\""
00119                       " -d chat_id=" +
00120                       chat_id + " -d text='Your authentication code is: " + code + "'";
00121
00122     FILE* pipe = popen(cmd.c_str(), "r");
00123     if (!pipe)
00124         return false;
00125
00126     std::string response;
00127     char buf[256];
00128     while (fgets(buf, sizeof(buf), pipe))
00129         response += buf;
00130     int status = pclose(pipe);
00131
00132     if (status == 0 && response.find("\"ok\":true") != std::string::npos) {
00133         auth_codes[chat_id] = code;
00134         return true;
00135     }
00136     return false;
00137 }
```

4.9.2.4 set_bot_token()

```
void set_bot_token (
                const std::string & token )
```

Установить глобальный токен бота.

Установить глобальный токен Telegram-бота.

Сохраняет переданный токен в переменную BOT_TOKEN.

Parameters

token	Токен бота, выданный BotFather.

Definition at line 33 of file telegram_auth.cpp.

```
00033                                       {
00034     BOT_TOKEN = token;
00035 }
```

4.9.2.5 verify_auth_code()

```
bool verify_auth_code (
                const std::string & chat_id,
                const std::string & code )
```

Проверить введённый код авторизации.

Проверить введённый пользователем код авторизации.

Сравнивает переданный code с сохранённым в auth_codes для данного chat_id.

Parameters

chat↩ _id	Идентификатор Telegram-чата.
code	Введённый пользователем код.

Returns

true если код корректен и сохранённая запись совпадает; иначе false.

Definition at line 150 of file telegram_auth.cpp.

```
00150                                                 {
00151     return auth_codes.count(chat_id) && auth_codes[chat_id] == code;
00152 }
```

4.9.3 Variable Documentation

4.9.3.1 auth_codes

std::map<std::string, std::string> auth_codes

Глобальная карта, хранящая соответствие chat_id -> код авторизации.

Definition at line 40 of file telegram_auth.cpp.

4.9.3.2 BOT_TOKEN

std::string BOT_TOKEN

Глобальный токен Telegram-бота, используется при отправке сообщений.

Должен быть установлен до вызова send_telegram_code().

Definition at line 22 of file telegram_auth.cpp.

4.10 telegram_auth.cpp

Go to the documentation of this file.

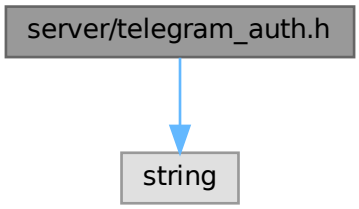
```
00001 /**
00002  * @file telegram_auth.cpp
00003  * @brief Реализация функций Telegram-аутентификации: генерация, отправка и проверка кодов.
00004  *
00005  * Использует CURL (через системный вызов curl) для отправки сообщений
00006  * Telegram Bot API. Хранит временные коды в глобальной карте auth_codes.
00007  */
00008
00009 #include "telegram_auth.h"
00010
00011 #include <cstdio>
00012 #include <cstdlib>
00013 #include <ctime>
00014 #include <filesystem>
00015 #include <fstream>
00016 #include <iostream>
00017 #include <map>
00018 #include <random>
00019 #include <string>
00020
00021 // Глобальная переменная для хранения токена Telegram-бота.
00022 std::string BOT_TOKEN;
00023
00024 //-----------------------------------------------------------------------------
00025
00026 /**
00027  * @brief Установить глобальный токен бота.
00028  *
00029  * Сохраняет переданный токен в переменную BOT_TOKEN.
00030  *
00031  * @param token Токен бота, выданный BotFather.
00032  */
00033 void set_bot_token(const std::string& token) {
00034     BOT_TOKEN = token;
00035 }
00036
00037 //-----------------------------------------------------------------------------
00038
00039 /// Глобальная карта, хранящая соответствие chat_id -> код авторизации.
00040 std::map<std::string, std::string> auth_codes;
00041
00042 //-----------------------------------------------------------------------------
00043
00044 /**
00045  * @brief Сгенерировать случайный шестизначный код для авторизации.
00046  *
00047  * Использует std::rand(), инициализирует генератор при первом вызове
00048  * на основе текущего времени.
00049  *
00050  * @return Сгенерированный код (строка из 6 цифр).
00051  */
00052 std::string generate_auth_code() {
00053     static bool seeded = false;
00054     if (!seeded) {
00055         std::srand(static_cast<unsigned>(std::time(nullptr)));
00056         seeded = true;
00057     }
00058
00059     std::string digits = "0123456789", code;
00060     for (int i = 0; i < 6; ++i)
00061         code += digits[std::rand() % 10];
00062     return code;
00063 }
00064
00065 //-----------------------------------------------------------------------------
00066
00067 /**
00068  * @brief Убедиться, что токен бота загружен из файла.
00069  *
00070  * Если переменная BOT_TOKEN пуста, пытается считать токен из файла
00071  * SERVER_SETTINGS/BOT_TOKEN.txt. При отсутствии директории или файла
00072  * создаёт их. Если после чтения токен остаётся пустым — выводит
00073  * сообщение об ошибке и завершает программу.
00074  */
00075 void ensure_bot_token() {
00076     if (!BOT_TOKEN.empty())
00077         return;
00078
00079     namespace fs = std::filesystem;
00080     fs::path dir = "SERVER_SETTINGS";
00081     fs::path file = dir / "BOT_TOKEN.txt";
00082
```

```
00083     if (!fs::exists(dir))
00084         fs::create_directories(dir);
00085 
00086     if (fs::exists(file)) {
00087         std::ifstream in(file);
00088         std::getline(in, BOT_TOKEN);
00089     } else {
00090         std::ofstream out(file);
00091     }
00092 
00093     if (BOT_TOKEN.empty()) {
00094         std::cerr « "[TelegramAuth] Файл " « file
00095                 « " не содержит токен.\n"
00096                   "Добавьте токен в первую строку и перезапустите сервер.\n";
00097         exit(1);
00098     }
00099 }
00100 
00101 //------------------------------------------------------------------------------
00102 
00103 /**
00104  * @brief Отправить код авторизации через Telegram Bot API.
00105  *
00106  * Формирует HTTP запрос с помощью системного вызова curl и отправляет код
00107  * пользователю в чат.
00108  *
00109  * @param chat_id Идентификатор Telegram-чата.
00110  * @param code    Шестизначный код авторизации.
00111  * @return true если запрос выполнен успешно (ответ содержит "ok":true),
00112  *         и код сохранён в auth_codes; иначе false.
00113  */
00114 bool send_telegram_code(const std::string& chat_id, const std::string& code) {
00115     ensure_bot_token();
00116     std::string url = "https://api.telegram.org/bot" + BOT_TOKEN + "/sendMessage";
00117     std::string cmd = "curl -s -X POST \"" + url +
00118                       "\""
00119                       " -d chat_id=" +
00120                       chat_id + " -d text='Your authentication code is: " + code + "'";
00121 
00122     FILE* pipe = popen(cmd.c_str(), "r");
00123     if (!pipe)
00124         return false;
00125 
00126     std::string response;
00127     char buf[256];
00128     while (fgets(buf, sizeof(buf), pipe))
00129         response += buf;
00130     int status = pclose(pipe);
00131 
00132     if (status == 0 && response.find("\"ok\":true") != std::string::npos) {
00133         auth_codes[chat_id] = code;
00134         return true;
00135     }
00136     return false;
00137 }
00138 
00139 //------------------------------------------------------------------------------
00140 
00141 /**
00142  * @brief Проверить введённый код авторизации.
00143  *
00144  * Сравнивает переданный @p code с сохранённым в auth_codes для данного @p chat_id.
00145  *
00146  * @param chat_id Идентификатор Telegram-чата.
00147  * @param code    Введённый пользователем код.
00148  * @return true если код корректен и сохранённая запись совпадает; иначе false.
00149  */
00150 bool verify_auth_code(const std::string& chat_id, const std::string& code) {
00151     return auth_codes.count(chat_id) && auth_codes[chat_id] == code;
00152 }
```

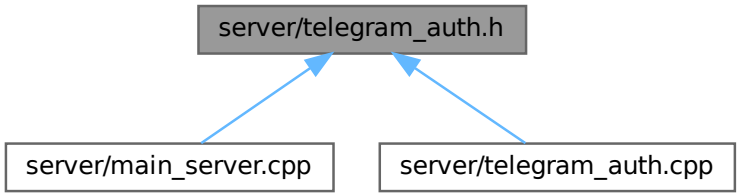
4.11 server/telegram_auth.h File Reference

Интерфейс для Telegram-аутентификации: генерация, отправка и проверка кодов.

#include <string>
Include dependency graph for telegram_auth.h:



This graph shows which files directly or indirectly include this file:



Functions

- void set_bot_token (const std::string &token)
  Установить глобальный токен Telegram-бота.
- std::string generate_auth_code ()
  Сгенерировать случайный шестизначный код для авторизации.
- bool send_telegram_code (const std::string &chat_id, const std::string &code)
  Отправить код авторизации через Telegram Bot API.
- bool verify_auth_code (const std::string &chat_id, const std::string &code)
  Проверить введённый пользователем код авторизации.
- void ensure_bot_token ()
  Убедиться, что глобальный токен бота загружен.

Variables

- std::string BOT_TOKEN
  Глобальный токен Telegram-бота, используется при отправке сообщений.

4.11.1 Detailed Description

Интерфейс для Telegram-аутентификации: генерация, отправка и проверка кодов.

Описание:

- Использует Telegram Bot API для отправки одноразовых кодов авторизации.
- Хранит сгенерированные коды в глобальной карте auth_codes.

Definition in file telegram_auth.h.

4.11.2 Function Documentation

4.11.2.1 ensure_bot_token()

```
void ensure_bot_token ( )
```

Убедиться, что глобальный токен бота загружен.

При первом вызове пытается считать токен из файла SERVER_SETTINGS/BOT_TOKEN.txt. Если файл отсутствует или пуст, создаёт его и выводит сообщение об ошибке. Завершает программу при отсутствии токена.

Убедиться, что глобальный токен бота загружен.

Если переменная BOT_TOKEN пуста, пытается считать токен из файла SERVER_SETTINGS/↩ BOT_TOKEN.txt. При отсутствии директории или файла создаёт их. Если после чтения токен остаётся пустым — выводит сообщение об ошибке и завершает программу.

Definition at line 75 of file telegram_auth.cpp.

```
00075                   {
00076     if (!BOT_TOKEN.empty())
00077         return;
00078
00079     namespace fs = std::filesystem;
00080     fs::path dir = "SERVER_SETTINGS";
00081     fs::path file = dir / "BOT_TOKEN.txt";
00082
00083     if (!fs::exists(dir))
00084         fs::create_directories(dir);
00085
00086     if (fs::exists(file)) {
00087         std::ifstream in(file);
00088         std::getline(in, BOT_TOKEN);
00089     } else {
00090         std::ofstream out(file);
00091     }
00092
00093     if (BOT_TOKEN.empty()) {
00094         std::cerr « "[TelegramAuth] Файл " « file
00095                   « " не содержит токен.\n"
00096                     "Добавьте токен в первую строку и перезапустите сервер.\n";
00097         exit(1);
00098     }
00099 }
```

4.11.2.2 generate_auth_code()

```
std::string generate_auth_code ( )
```

Сгенерировать случайный шестизначный код для авторизации.

Код состоит из цифр [0-9] и всегда имеет длину 6 символов.

Returns

Сгенерированный код (например, "042517").

Использует std::rand(), инициализирует генератор при первом вызове на основе текущего времени.

Returns

Сгенерированный код (строка из 6 цифр).

Definition at line 52 of file telegram_auth.cpp.

```
00052                                {
00053     static bool seeded = false;
00054     if (!seeded) {
00055         std::srand(static_cast<unsigned>(std::time(nullptr)));
00056         seeded = true;
00057     }
00058
00059     std::string digits = "0123456789", code;
00060     for (int i = 0; i < 6; ++i)
00061         code += digits[std::rand() % 10];
00062     return code;
00063 }
```

4.11.2.3 send_telegram_code()

```
bool send_telegram_code (
                const std::string & chat_id,
                const std::string & code )
```

Отправить код авторизации через Telegram Bot API.

Формирует и выполняет HTTP-запрос для отправки сообщения с одноразовым кодом.

Parameters

chat_id	Идентификатор Telegram-чата получателя.
code	Шестизначный код, который будет отправлен.

Returns

true, если сообщение успешно отправлено и код сохранён; false в случае ошибки при вызове curl или неверного ответа API.

Формирует HTTP запрос с помощью системного вызова curl и отправляет код пользователю в чат.

Parameters

chat↩ _id	Идентификатор Telegram-чата.
code	Шестизначный код авторизации.

Returns

true если запрос выполнен успешно (ответ содержит "ok":true), и код сохранён в auth_codes; иначе false.

Definition at line 114 of file telegram_auth.cpp.

```
00114                                                    {
00115     ensure_bot_token();
00116     std::string url = "https://api.telegram.org/bot" + BOT_TOKEN + "/sendMessage";
00117     std::string cmd = "curl -s -X POST \"" + url +
00118                       "\""
00119                       " -d chat_id=" +
00120                       chat_id + " -d text='Your authentication code is: " + code + "'";
00121
00122     FILE* pipe = popen(cmd.c_str(), "r");
00123     if (!pipe)
00124         return false;
00125
00126     std::string response;
00127     char buf[256];
00128     while (fgets(buf, sizeof(buf), pipe))
00129         response += buf;
00130     int status = pclose(pipe);
00131
00132     if (status == 0 && response.find("\"ok\":true") != std::string::npos) {
00133         auth_codes[chat_id] = code;
00134         return true;
00135     }
00136     return false;
00137 }
```

4.11.2.4 set_bot_token()

```
void set_bot_token (
                const std::string & token )
```

Установить глобальный токен Telegram-бота.

Сохраняет переданный token для последующих HTTP-запросов.

Parameters

token	Строка токена, выданная BotFather.

Установить глобальный токен Telegram-бота.

Сохраняет переданный токен в переменную BOT_TOKEN.

Parameters

token	Токен бота, выданный BotFather.

Definition at line 33 of file telegram_auth.cpp.

```
00033                                {
00034     BOT_TOKEN = token;
00035 }
```

4.11.2.5 verify_auth_code()

```
bool verify_auth_code (
                const std::string & chat_id,
                const std::string & code )
```

Проверить введённый пользователем код авторизации.

Сравнивает переданный code с сохранённым в auth_codes для данного chat_id.

Parameters

chat↩ _id	Идентификатор Telegram-чата, для которого код генерировался.
code	Код, введённый пользователем.

Returns

true, если код совпадает с ранее сгенерированным; иначе false.

Проверить введённый пользователем код авторизации.

Сравнивает переданный code с сохранённым в auth_codes для данного chat_id.

Parameters

chat↩ _id	Идентификатор Telegram-чата.
code	Введённый пользователем код.

Returns

true если код корректен и сохранённая запись совпадает; иначе false.

Definition at line 150 of file telegram_auth.cpp.

```
00150                                                {
00151     return auth_codes.count(chat_id) && auth_codes[chat_id] == code;
00152 }
```

4.11.3 Variable Documentation

4.11.3.1 BOT_TOKEN

```
std::string BOT_TOKEN   [extern]
```

Глобальный токен Telegram-бота, используется при отправке сообщений.

Должен быть установлен до вызова send_telegram_code().

Definition at line 22 of file telegram_auth.cpp.

4.12 telegram_auth.h

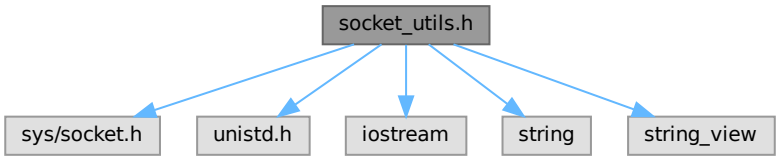
Go to the documentation of this file.

```
00001 /**
00002  * @file telegram_auth.h
00003  * @brief Интерфейс для Telegram-аутентификации: генерация, отправка и проверка кодов.
00004  *
00005  * Описание:
00006  * - Использует Telegram Bot API для отправки одноразовых кодов авторизации.
00007  * - Хранит сгенерированные коды в глобальной карте auth_codes.
00008  */
00009
00010 #ifndef TELEGRAM_AUTH_H
00011 #define TELEGRAM_AUTH_H
00012
00013 #include <string>
00014
00015 /// Глобальный токен Telegram-бота, используется при отправке сообщений.
00016 /**
00017  * Должен быть установлен до вызова send_telegram_code().
00018  */
00019 extern std::string BOT_TOKEN;
00020
00021 /**
00022  * @brief Установить глобальный токен Telegram-бота.
00023  *
00024  * Сохраняет переданный @p token для последующих HTTP-запросов.
00025  *
00026  * @param token Строка токена, выданная BotFather.
00027  */
00028 void set_bot_token(const std::string& token);
00029
00030 /**
00031  * @brief Сгенерировать случайный шестизначный код для авторизации.
00032  *
00033  * Код состоит из цифр [0-9] и всегда имеет длину 6 символов.
00034  *
00035  * @return Сгенерированный код (например, "042517").
00036  */
00037 std::string generate_auth_code();
00038
00039 /**
00040  * @brief Отправить код авторизации через Telegram Bot API.
00041  *
00042  * Формирует и выполняет HTTP-запрос для отправки сообщения с одноразовым кодом.
00043  *
00044  * @param chat_id Идентификатор Telegram-чата получателя.
00045  * @param code    Шестизначный код, который будет отправлен.
00046  * @return true, если сообщение успешно отправлено и код сохранён;
00047  *         false в случае ошибки при вызове curl или неверного ответа API.
00048  */
00049 bool send_telegram_code(const std::string& chat_id, const std::string& code);
00050
00051 /**
00052  * @brief Проверить введённый пользователем код авторизации.
00053  *
00054  * Сравнивает переданный @p code с сохранённым в auth_codes для данного @p chat_id.
00055  *
00056  * @param chat_id Идентификатор Telegram-чата, для которого код генерировался.
00057  * @param code    Код, введённый пользователем.
00058  * @return true, если код совпадает с ранее сгенерированным; иначе false.
00059  */
00060 bool verify_auth_code(const std::string& chat_id, const std::string& code);
00061
00062 /**
00063  * @brief Убедиться, что глобальный токен бота загружен.
00064  *
00065  * При первом вызове пытается считать токен из файла SERVER_SETTINGS/BOT_TOKEN.txt.
00066  * Если файл отсутствует или пуст, создаёт его и выводит сообщение об ошибке.
00067  * Завершает программу при отсутствии токена.
00068  */
00069 void ensure_bot_token();
00070
00071 #endif // TELEGRAM_AUTH_H
```

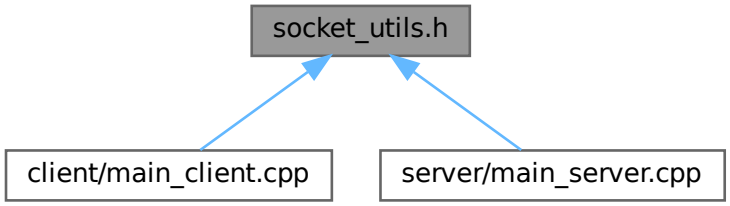
4.13 socket_utils.h File Reference

Обёртки функций отправки и приёма данных по TCP-сокетам.

#include <sys/socket.h>
#include <unistd.h>
#include <iostream>
#include <string>
#include <string_view>
Include dependency graph for socket_utils.h:



This graph shows which files directly or indirectly include this file:



Functions

- bool send_all (int fd, std::string_view data)

  Отправить весь буфер данных через сокет.
- bool send_packet (int fd, std::string_view data)

  Отправить пакет текстовых данных через сокет.
- bool send_line (int fd, std::string_view sv)

  Отправить одну строку через сокет.
- bool recv_line (int fd, std::string &out)

  Прочитать одну строку из сокета до символа новой строки.

4.13.1 Detailed Description

Обёртки функций отправки и приёма данных по TCP-сокетам.

Содержит inline-функции:

- send_all: отправить весь буфер данных;

- send_packet: отправить пакет строки с маркером конца сообщения "*ENDM*";
- send_line: отправить одну строку с терминатором '
';
- recv_line: получить одну строку до символа '
'.

Definition in file socket_utils.h.

4.13.2 Function Documentation

4.13.2.1 recv_line()

```
bool recv_line (
            int fd,
            std::string & out )   [inline]
```

Прочитать одну строку из сокета до символа новой строки.

Читает по одному символу через ::recv() и сохраняет их в out до встречи '
'. Символ '
' не включается.

Parameters

fd	Дескриптор сокета.
out	Переменная для сохранения прочитанной строки.

Returns

true если строка успешно прочитана, false при закрытии соединения или ошибке.

Definition at line 90 of file socket_utils.h.

```
00090                                       {
00091     out.clear();
00092     char ch{};
00093     while (true) {
00094         ssize_t n = ::recv(fd, &ch, 1, 0);
00095         if (n <= 0)
00096             return false;
00097         if (ch == '\n')
00098             break;
00099         out.push_back(ch);
00100     }
00101     return true;
00102 }
```

4.13.2.2 send_all()

```
bool send_all (
            int fd,
            std::string_view data )   [inline]
```

Отправить весь буфер данных через сокет.

Использует ::send() в цикле, пока не отправит все байты из data.

Parameters

fd	Дескриптор сокета.
data	Буфер данных для отправки.

Returns

true если все данные успешно отправлены, false при ошибке.

Definition at line 31 of file socket_utils.h.

```
00031                                          {
00032     std::string message(data);
00033     size_t sent = 0;
00034     while (sent < message.size()) {
00035         ssize_t n = ::send(fd, message.data() + sent, message.size() - sent, 0);
00036         if (n <= 0)
00037             return false;
00038         sent += static_cast<size_t>(n);
00039     }
00040     return true;
00041 }
```

4.13.2.3 send_line()

```
bool send_line (
                int fd,
                std::string_view sv )    [inline]
```

Отправить одну строку через сокет.

Гарантирует наличие символа новой строки '
' в конце и отправляет через send_all().

Parameters

fd	Дескриптор сокета.
sv	Строка для отправки.

Returns

true если строка успешно отправлена, false при ошибке.

Definition at line 71 of file socket_utils.h.

```
00071                                          {
00072     if (sv.empty() || sv.back() != '\n') {
00073         std::string tmp(sv);
00074         tmp.push_back('\n');
00075         return send_all(fd, tmp);
00076     }
00077     return send_all(fd, sv);
00078 }
```

4.13.2.4 send_packet()

```
bool send_packet (
                int fd,
                std::string_view data )    [inline]
```

Отправить пакет текстовых данных через сокет.

Добавляет '
' в конец сообщения, если его нет, затем добавляет маркер "*ENDM*\n" и отправляет через send_all().

Parameters

fd	Дескриптор сокета.
data	Текст данных для отправки.

Returns

true если пакет успешно отправлен, false при ошибке.

Definition at line 53 of file socket_utils.h.

```
00053                                                 {
00054     std::string message(data);
00055     if (message.empty() || message.back() != '\n')
00056         message.push_back('\n');
00057     message += "*ENDM*\n";
00058     return send_all(fd, message);
00059 }
```

4.14 socket_utils.h

Go to the documentation of this file.

```
00001 /**
00002  * @file socket_utils.h
00003  * @brief Обёртки функций отправки и приёма данных по TCP-сокетам.
00004  *
00005  * Содержит inline-функции:
00006  *  - send_all: отправить весь буфер данных;
00007  *  - send_packet: отправить пакет строки с маркером конца сообщения "*ENDM*";
00008  *  - send_line: отправить одну строку с терминатором '\n';
00009  *  - recv_line: получить одну строку до символа '\n'.
00010  */
00011
00012 #ifndef SOCKET_UTILS_H
00013 #define SOCKET_UTILS_H
00014
00015 #include <sys/socket.h>
00016 #include <unistd.h>
00017
00018 #include <iostream>
00019 #include <string>
00020 #include <string_view>
00021
00022 /**
00023  * @brief Отправить весь буфер данных через сокет.
00024  *
00025  * Использует ::send() в цикле, пока не отправит все байты из @p data.
00026  *
00027  * @param fd Дескриптор сокета.
00028  * @param data Буфер данных для отправки.
00029  * @return true если все данные успешно отправлены, false при ошибке.
00030  */
00031 inline bool send_all(int fd, std::string_view data) {
00032     std::string message(data);
00033     size_t sent = 0;
00034     while (sent < message.size()) {
00035         ssize_t n = ::send(fd, message.data() + sent, message.size() - sent, 0);
00036         if (n <= 0)
00037             return false;
00038         sent += static_cast<size_t>(n);
00039     }
00040     return true;
00041 }
00042
00043 /**
```

```
00044  * @brief Отправить пакет текстовых данных через сокет.
00045  *
00046  * Добавляет '\n' в конец сообщения, если его нет,
00047  * затем добавляет маркер "*ENDM*\n" и отправляет через send_all().
00048  *
00049  * @param fd Дескриптор сокета.
00050  * @param data Текст данных для отправки.
00051  * @return true если пакет успешно отправлен, false при ошибке.
00052  */
00053 inline bool send_packet(int fd, std::string_view data) {
00054     std::string message(data);
00055     if (message.empty() || message.back() != '\n')
00056         message.push_back('\n');
00057     message += "*ENDM*\n";
00058     return send_all(fd, message);
00059 }
00060
00061 /**
00062  * @brief Отправить одну строку через сокет.
00063  *
00064  * Гарантирует наличие символа новой строки '\n' в конце
00065  * и отправляет через send_all().
00066  *
00067  * @param fd Дескриптор сокета.
00068  * @param sv Строка для отправки.
00069  * @return true если строка успешно отправлена, false при ошибке.
00070  */
00071 inline bool send_line(int fd, std::string_view sv) {
00072     if (sv.empty() || sv.back() != '\n') {
00073         std::string tmp(sv);
00074         tmp.push_back('\n');
00075         return send_all(fd, tmp);
00076     }
00077     return send_all(fd, sv);
00078 }
00079
00080 /**
00081  * @brief Прочитать одну строку из сокета до символа новой строки.
00082  *
00083  * Читает по одному символу через ::recv() и сохраняет
00084  * их в @p out до встречи '\n'. Символ '\n' не включается.
00085  *
00086  * @param fd Дескриптор сокета.
00087  * @param out Переменная для сохранения прочитанной строки.
00088  * @return true если строка успешно прочитана, false при закрытии соединения или ошибке.
00089  */
00090 inline bool recv_line(int fd, std::string& out) {
00091     out.clear();
00092     char ch{};
00093     while (true) {
00094         ssize_t n = ::recv(fd, &ch, 1, 0);
00095         if (n <= 0)
00096             return false;
00097         if (ch == '\n')
00098             break;
00099         out.push_back(ch);
00100     }
00101     return true;
00102 }
00103
00104 #endif  // SOCKET_UTILS_H
```

Предметный указатель