

## СОДЕРЖАНИЕ

Введение.....	4
1 Обзор средств и подходов создания многопользовательского чата. Обзор аналогов.....	5
1.1 Основные понятия чата. История их возникновения .....	5
1.2 Виды чатов.....	6
1.3 Программная реализация чатов.....	7
1.4 Основы сетевого взаимодействия .....	8
1.5 Существующие аналоги .....	12
2 Архитектура многопользовательского чата.....	14
2.1 Схема работы сервера многопользовательского чата .....	14
2.2 Проектирование архитектуры приложения .....	15
2.3 Структурная схема приложения.....	15
2.4 Сетевая архитектура работы программного средства.....	17
2.5 Клиентская часть <i>web</i> -приложения.....	18
2.6 Клиентская часть десктопного приложения .....	21
3 Верификация разработанного приложения многопользовательского чата .	23
3.1 Модульное тестирование .....	23
3.2 Нагрузочное тестирование.....	24
3.3 Тестирование запущенного приложения .....	25
3.4 Результаты тестирования приложения.....	29
Заключение.....	30
Список использованных источников.....	31
Приложение А. Листинг программного комплекса.....	32
Приложение Б. Руководство системного программиста .....	63
Приложение В. Руководство программиста .....	65
Приложение Г. Руководство пользователя.....	67
Приложение Д. Схема архитектуры приложения.....	71

## ВВЕДЕНИЕ

В последние годы набирают популярность многопользовательские чаты как способ поддержания связи на расстоянии. Чат позволяет производить обмен не только текстом, но и картинками, файлами и т.д. в режиме реального времени. Приложения по обмену сообщениями стали настолько популярными, что даже некоторые предприятия внедряют корпоративные чаты для сотрудников с целью эффективного обмена информацией.

Для эффективного внедрения чата с минимальными усилиями в любую среду необходимо разработать сетевое приложение, реализующее многопользовательский чат на основе протокола *HTTP*. Данная архитектура позволит внедрить функции чата в любое приложение и сайт, соответствующие минимальным требованиям, без изменения архитектуры самого приложения или сайта.

Данное приложение должно реализовывать систему аутентификации и авторизации пользователей. Авторизованные пользователи имеют доступ только к доступному данному пользователю функционалу, что позволяет защитить важные данные от неавторизованного доступа.

Каждый пользователь будет иметь доступ к истории собственных сообщений, при этом администратор будет иметь доступ к истории любого пользователя. Пользователь может просмотреть любое из сообщений, написанное им когда-либо, что позволит получить информацию, переданную пользователем в прошлом.

# 1 ОБЗОР СРЕДСТВ И ПОДХОДОВ СОЗДАНИЯ МНОГОПОЛЬЗОВАТЕЛЬСКОГО ЧАТА. ОБЗОР АНАЛОГОВ

## 1.1 Основные понятия чата. История их возникновения

Чат представляет собой программу или сервис для моментального обмена сообщениями. В некоторых случаях, чат размещается на странице какого-либо ресурса в виде отдельного окна. Его главное отличие от других подобных сервисов для общения, например, от форума, является то, что общение происходит без задержки. Пользователь отправляет сообщение другому человеку и практически сразу же получает на него ответ.

Кроме интернет-чатов, популярностью пользуются чаты в виде отдельного программного обеспечения. Яркими примерами подобных программ является *ICQ*. Обмен сообщениями в *Skype*, обладает теми же функциями.

Во второй половине XX века начали бурно развиваться компьютеры. Однако долгое время они были большими и слишком дорогими, что препятствовало тому, чтобы расходовать драгоценное машинное время на забавы с обменом сообщениями вместо расчётов атомных бомб. К тому же до конца 60-х годов они не были связаны друг с другом. Предок Интернета, сеть *ARPANET*, в 1969 году насчитывала только четыре связанных друг с другом научных компьютера. Чуть позже, в 1971 году, была придумана электронная почта, которая стала необычайно популярна ввиду своего удобства. Постепенно появились новые службы сообщений, такие, как списки почтовой рассылки, новостные группы и доски объявлений. Однако в то время сеть *ARPANET* ещё не могла легко взаимодействовать с другими сетями, построенными на других технических стандартах, что затрудняло её распространение. Но тем не менее эта проблема вскоре была решена после перехода сетей на протокол обмена данными *TCP/IP*, который успешно применяется до сих пор. Именно в 1983 году термин «Интернет» закрепился за сетью *ARPANET*. Программы для обмена текстовыми строками, несмотря на простоту самой идеи, появились не сразу. Примерно в 1974 году для мэйнфрейма *PLATO* была разработана программа *Talkomatic*, потенциально позволявшая общаться между тысячей терминалов системы. В 1980-х появилась система *Freelancing' Round table*. Однако по-настоящему популярным стал разработанный в 1988 году протокол, названный *Internet Relay Chat (IRC)*, что примерно можно перевести как ретранслируемый интернет-разговор. Примерно в это же время появилось и распространилось само понятие «чат». Общение в *IRC* быстро стало популярным из-за простоты

процесса и дружелюбности среды. В 1991 году во время операции «Буря в пустыне» была организована *IRC*-трансляция новостей – сообщения со всего мира собирались в одном месте и в режиме реального времени передавались в *IRC*. Есть сведения, что подобным образом *IRC* использовался и во время путча в СССР, когда пользователи из Москвы моментально сообщали всему миру о происходящем на улицах. Для клиентов *IRC* написано множество ботов, например, *Eggdrop*, автоматизирующие многие рутинные операции. Самым известным из клиентов *IRC* стал *mIRC*; благодаря простой и эффективной системе команд для него было написано множество скриптов, которые также позволяют выполнять широкий спектр действий. Боты используются для различных игр в каналах – «Мафия», «Викторина» и других.

В настоящее время интерес к чатам падает. Их место заняли социальные сети, с их намного большими функциональностями. Популярность набирают только видеочаты [2].

## 1.2 Виды чатов

Веб-чат. Веб-чаты распространились в 90-х годах XX века. В некоторых случаях под чатом подразумевают веб-чат. Веб-чаты основаны на технологиях интернета (*HTTP* и *HTML*). Первоначально веб-чаты представляли собой страницу с разговором и форму ввода, посредством которой введённый текст отсылался на сервер. В подобных чатах сервер добавлял новые сообщения в текстовую область, удалял старые сообщения, обновлял файл. Чат осуществлялся с задержкой: веб-средства не позволяли серверу сообщить клиенту об изменениях – клиент мог только запрашивать данные сам с некоторой периодичностью. Это задержки были устранены с помощью технологий *AJAX* и *Flash*. Также существовали некоторые другие системы, не имевшие подобных недостатков.

Веб-чаты использовались для атак на пользователей. Этому способствовали уязвимости в программном обеспечении (скриптах). Поэтому многие веб-сервера, где находятся чаты, были вынуждены защищаться от атак.

Видеочат. Видеочаты – это обмен текстовыми сообщениями и транслирование изображений с веб-камер. Поначалу это были не видео-, а скорее, фоточаты: из-за низкой пропускной способности каналов отправлялся не видеопоток, а картинка с некоторыми интервалами, что, однако, давало возможность достаточно оперативно наблюдать смену эмоций у собеседника и было значительным прорывом. Позднее, конечно, стал транслироваться видеопоток, хотя и с низким разрешением. Веб-камеры являются простыми и дешёвыми, хотя обратная сторона этого – низкое разрешение видео и его

плохое качество. Изображение получается с плохой цветопередачей, зашумлённое. Однако для целей общения такого качества более чем достаточно.

Голосовые чаты тоже явились развитием идей обмена сообщениями. В настоящее время в компьютерных играх широко применяется система *Discord*, позволяющая общаться голосом между членами команды, не отвлекаясь от управления игрой. А общение по *Skype* больше напоминает разговор по телефону, чем чат, хотя возможность отправки обычных текстовых сообщений в нём тоже присутствует.

Системы мгновенных сообщений или мессенджер. Это службы мгновенных сообщений (*Instant Messaging Service, IMS*), программы онлайн-консультанты (*OnlineSaler*) и программы-клиенты (*Instant Messenger, IM*) для обмена сообщениями в реальном времени через Интернет. Могут передаваться текстовые сообщения, звуковые сигналы, изображения, видео, а также производиться такие действия, как совместное рисование или игры. Многие из таких программ-клиентов могут применяться для организации групповых текстовых чатов или видеоконференций. Большинство *IM*-клиентов позволяет видеть, подключены ли в данный момент абоненты, занесённые в список контактов. В ранних версиях программ всё, что печатал пользователь, тут же передавалось. Если он делал ошибку и исправлял её, это тоже было видно. В таком режиме общение напоминало телефонный разговор. В современных программах сообщения появляются на мониторе собеседника уже после окончания редактирования и отправки сообщения.

Как правило, мессенджеры не работают самостоятельно, а подключаются к центральному компьютеру сети обмена сообщениями, называемому сервером. Поэтому мессенджеры и называют клиентами (клиентскими программами). Термин является понятием из клиент-серверных технологий.

Теле чаты. Используются на телеканалах, таких, как *MTV*, *RU.TV*. Сообщение передаётся путём отправки *SMS* с мобильного. Чаще всего это объявления о знакомствах или поздравления с праздниками. Также на некоторых каналах ведётся общение с диджеем или ведущим. Однако большинство сообщений платные.

### **1.3 Программная реализация чатов**

Существует несколько разновидностей программной реализации чатов:

– *HTTP* или веб-чаты. Такой чат выглядит как обычная веб-страница, где можно прочесть последние несколько десятков фраз, написанные участниками

чата и модераторами. Страница чата автоматически обновляется с заданной периодичностью;

- чаты, использующие технологию *Adobe Flash*. Вместо периодической перезагрузки страницы между клиентом и сервером открывается сокет, что позволяет моментально отправлять или получать сообщения, расходуя меньше трафика;

- *IRC*, специализированный протокол для чатов. *IRC* (англ. *Internet Relay Chat*) – протокол прикладного уровня для обмена сообщениями в режиме реального времени. Разработан в основном для группового общения, также позволяет общаться через личные сообщения и обмениваться данными, в том числе файлами. *IRC* использует транспортный протокол *TCP* и криптографический *TLS* (опционально);

- программы-чаты для общения в локальных сетях (например, *Vypress Chat, Intranet Chat, Pichat*). Часто есть возможность передачи файлов;

- чаты, реализованные поверх сторонних протоколов (например, чат, использующий *ICQ*);

- чаты, работающие по схеме клиент-сервер, это позволяет использовать их в сетях со сложной конфигурацией, а также управлять клиентскими приложениями (например, *Mychat, Jabber*);

- чаты, работающие в одноранговых сетях. У них нет потребности в отдельном сервере, они часто используют возможности технологий *DHT* и *TCP Relay* (пример: *Tox*);

- чаты, использующие технологию *Push*. Вместо периодической отправки запросов серверу о новых сообщениях, используются входящие сообщения от сервера, что позволяет отправлять и получать сообщения, расходуя меньше трафика (например, *WinGeoChat*);

- полностью анонимные чаты. В них собеседник не предполагает с кем общается и при каждом новом соединении общается с новым человеком [3].

## 1.4 Основы сетевого взаимодействия

В начале 80-х годов международная организация по стандартизации (International Standardization Organization – ISO) разработала модель OSI, которая сыграла значительную роль в развитии сетей. Эталонная модель OSI, иногда называемая стеком OSI представляет собой 7-уровневую сетевую иерархию. горизонтальную модель на базе протоколов, обеспечивающую механизм взаимодействия программ и процессов на различных машинах. Вертикальная модель базируется на основе услуг, обеспечиваемых соседними уровнями друг другу на одной машине. В горизонтальной модели двум

программам требуется общий протокол для обмена данными. В вертикальной – соседние уровни обмениваются данными с использованием интерфейсов API.

В ситуации, когда приложение обращается с запросом к прикладному уровню, например, к файловой службе. На основании этого запроса программное обеспечение прикладного уровня формирует сообщение стандартного формата. Обычное сообщение состоит из заголовка и поля данных. Заголовок содержит служебную информацию, которую необходимо передать через сеть прикладному уровню машины-адресата, чтобы сообщить ему, какую работу надо выполнить. Поле данных сообщения может быть пустым или содержать какие-либо данные, которые необходимо записать в удаленный файл. Но для того чтобы доставить эту информацию по назначению, предстоит решить еще много задач, ответственность за которые несут нижележащие уровни.

После формирования сообщения прикладной уровень направляет его вниз по стеку представительному уровню. Протокол представительного уровня на основании информации, полученной из заголовка прикладного уровня, выполняет требуемые действия и добавляет к сообщению собственную служебную информацию – заголовок представительного уровня, в котором содержатся указания для протокола представительного уровня машины-адресата.

Полученное в результате сообщение передается вниз сеансовому уровню, который в свою очередь добавляет свой заголовок, и т. д.

Наконец, сообщение достигает нижнего, физического уровня, который собственно и передает его по линиям связи машине-адресату. К этому моменту сообщение «обрастает» заголовками всех уровней (рис. 1.1).

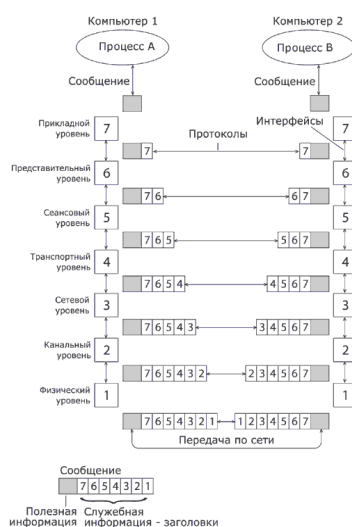


Рисунок 1.1 – Эталонная модель взаимодействия открытых систем (ISO/OSI).

Когда сообщение по сети поступает на машину-адресат, оно принимается ее физическим уровнем и последовательно перемещается вверх с уровня на уровень. Каждый уровень анализирует и обрабатывает заголовок своего уровня, выполняя соответствующие данному уровню функции, а затем удаляет этот заголовок и передает сообщение вышележащему уровню.

В модели *OSI* различаются два основных типа протоколов. В протоколах с установлением соединения перед обменом данными отправитель и получатель должны сначала установить соединение и, возможно, выбрать некоторые параметры протокола, которые они будут использовать при обмене данными. После завершения диалога они должны разорвать это соединение. Вторая группа протоколов – протоколы без предварительного установления соединения. Такие протоколы называются также дейтаграммными протоколами. Отправитель просто передает сообщение, когда оно готово. При взаимодействии компьютеров используются протоколы обоих типов.

Каждый уровень модели *OSI* имеет четкие функции.

**1.4.1** На физическом уровне модели *OSI* определяются следующие характеристики сетевых компонентов:

- типы соединений сред передачи данных, физические топологии сети, способы передачи данных (с цифровым или аналоговым кодированием сигналов), виды синхронизации передаваемых данных;
- разделение каналов связи с использованием частотного и временного мультиплексирования.

Физический уровень не включает описание среды передачи. Однако реализации протоколов физического уровня специфичны для конкретной среды передачи. С физическим уровнем обычно ассоциируется подключение сетевого оборудования.

**1.4.2** Канальный уровень определяет логическую топологию сети, правила получения доступа к среде передачи данных, решает вопросы, связанные с адресацией физических устройств в рамках логической сети и управлением передачей информации (синхронизация передачи и сервис соединений) между сетевыми устройствами. Протоколы канального уровня реализуются для достижения следующих основных целей:

- организации битов физического уровня (двоичные единицы и нули) в логические группы информации, называемые фреймами или кадрами. Фрейм является единицей данных канального уровня, состоящей из непрерывной последовательности сгруппированных битов, имеющей заголовок и окончание;
- обнаружения (а иногда и исправления) ошибок при передаче;



– управления потоками данных (для устройств, работающих на этом уровне, например, мостов);

– идентификации компьютеров в сети по их физическим адресам.

**1.4.3** Сетевой уровень служит для образования единой транспортной системы, объединяющей несколько сетей, причем эти сети могут использовать совершенно различные принципы передачи сообщений между конечными узлами и обладать произвольной структурой связей. Функции сетевого уровня достаточно разнообразны.

Сообщения сетевого уровня принято называть пакетами. При организации доставки пакетов на сетевом уровне используется понятие «номер сети». В этом случае адрес получателя состоит из старшей части - номера сети и младшей – номера узла в этой сети. Все узлы одной сети должны иметь одну и ту же старшую часть адреса

**1.4.4** На пути от отправителя к получателю пакеты могут быть искажены или утеряны. Хотя некоторые приложения имеют собственные средства обработки ошибок, существуют и такие, которые предпочитают сразу иметь дело с надежным соединением. Транспортный уровень обеспечивает приложениям или верхним уровням стека - прикладному и сеансовому - передачу данных с той степенью надежности, которая им требуется.

Модель *OSI* определяет пять классов сервиса, предоставляемых транспортным уровнем. Эти виды сервиса отличаются качеством предоставляемых услуг: срочностью, возможностью восстановления прерванной связи, наличием средств мультиплексирования нескольких соединений между различными прикладными протоколами через общий транспортный протокол, а главное – способностью к обнаружению и исправлению ошибок передачи, таких как искажение, потеря и дублирование пакетов [5].

**1.4.5** Сеансовый уровень обеспечивает управление диалогом: фиксирует, какая из сторон является активной в настоящий момент, предоставляет средства синхронизации. Последние позволяют вставлять контрольные точки в длинные передачи, чтобы в случае отказа можно было вернуться назад к последней контрольной точке, а не начинать все с начала. Сеансовый уровень реализует управление диалогом с использованием одного из трёх способов общения: симплекс, полудуплекс и полный дуплекс.

**1.4.6** Представительный уровень имеет дело с формой представления передаваемой по сети информации, не меняя при этом ее содержания. За счет уровня представления информация, передаваемая прикладным уровнем одной системы, всегда понятна прикладному уровню другой системы. На этом уровне может выполняться шифрование и дешифрование данных, благодаря

которому секретность обмена данными обеспечивается сразу для всех прикладных служб. Примером такого протокола является протокол *Secure Socket Layer (SSL)*, который обеспечивает секретный обмен сообщениями для протоколов прикладного уровня стека *TCP/IP*.

**1.4.7** Прикладной уровень – это в действительности просто набор разнообразных протоколов, с помощью которых пользователи сети получают доступ к разделяемым ресурсам, таким как файлы, принтеры или гипертекстовые *Web*-страницы, а также организуют свою совместную работу, например, с помощью протокола электронной почты. Единица данных, которой оперирует прикладной уровень, обычно называется сообщением. Существует очень большое разнообразие служб прикладного уровня [6 с. 126]. К числу наиболее распространенных протоколов верхних уровней относятся:

- *FTP* – протокол переноса файлов;
- *TFTP* – упрощенный протокол переноса файлов;
- *SMTP* – простой протокол почтового обмена;
- *CMIP* – общий протокол управления информацией;
- *SNMP* – простой протокол управления сетью;
- *NFS* – сетевая файловая система.

## **1.5 Существующие аналоги**

На сегодняшний день существует большое количество многопользовательских чатов, что обусловлено современными тенденциями и образом жизни. Все они чем-то отличаются, но в целом, имеют схожий функционал. Для более подробного анализа следует рассмотреть несколько чатов, например, мессенджер *Viber*.

Данное приложение позволяет отправлять сообщения, сообщать видео- и голосовые звонки через Интернет. Голосовые вызовы между пользователями с установленным *Viber* бесплатны (оплачивается только интернет-трафик по тарифу оператора связи). *Viber* имеет возможность отправлять текстовые, голосовые и видеосообщения, документы, изображения, видеозаписи и файлы, а также в автономном режиме. Преимуществом является то, что в мессенджере предусмотрена функция шифрования, которая обеспечивает тайну переписки. Главным недостатком для авторизации пользователей и поиска контактов приложение использует номер телефона и передает содержимое телефонной адресной книги (имена и телефоны всех контактов) на серверы корпорации *Viber Media S.à r.l.*, Люксембург. Они же собирают информацию о совершенных звонках и переданных сообщениях, длительности звонков, участниках звонков и чатов.

*Facebook Messenger* — приложение для обмена мгновенными сообщениями и видео, созданное *Meta*. Оно интегрировано с системой обмена сообщениями в социальной сети *Facebook*. Данный чат имеет веб-версию, мобильное приложение, а также приложение для персонального компьютера. Поддерживается функция обмена файлами, функция создания опросов, функция добавления друзей через сканирование картинок профиля. Однако, данный чат, кроме преимуществ, имеет недостатки, например, наличие рекламы в приложении, для того, чтобы пользоваться чатом необходимо быть пользователем социальной сети *Facebook*. Главным недостатком является то, что приложение занимает много места на устройстве.

В результате аналитического обзора был выявлен ряд недостатков, главными из которых является отсутствие функции сокрытия личной информации и объем, занимаемый устройством. В связи с этим можно сформулировать задачу, которую необходимо решить.

Требуется разработать сетевое приложение, реализующее многопользовательский чат на основе протокола *HTTP*.

Для решения задачи необходимо разработать приложение на высокоуровневом языке программирования. Также необходимо провести верификации работы разработанного приложения.

Программа должна иметь графический пользовательский интерфейс, содержащий список доступных чатов, кнопку создания чата, поле ввода сообщения и прикрепления файла, дополнительные кнопки для взаимодействия с чатами и пользователями. Результатом работы приложения является функционирующий чат, выполняющий все предусмотренные функции.

## 2 АРХИТЕКТУРА МНОГОПОЛЬЗОВАТЕЛЬСКОГО ЧАТА

### 2.1 Схема работы сервера многопользовательского чата

Протокол *HTTP* является удобным инструментом для создания многопользовательских чатов.

*HTTP* – это протокол, позволяющий получать различные ресурсы, например *HTML*-документы. Протокол *HTTP* лежит в основе обмена данными в Интернете. *HTTP* является протоколом клиент-серверного взаимодействия, что означает инициирование запросов к серверу самим получателем, обычно веб-браузером (*web-browser*). Полученный итоговый документ будет (может) состоять из различных поддокументов, являющихся частью итогового документа: например, из отдельно полученного текста, описания структуры документа, изображений, видео-файлов, скриптов и многого другого.

Клиенты и серверы взаимодействуют, обмениваясь одиночными сообщениями (а не потоком данных). Сообщения, отправленные клиентом, обычно веб-браузером, называются запросами, а сообщения, отправленные сервером, называются ответами. Пример работы протокола *HTTP* представлен на рисунке 2.1.

В интернете существуют строгие правила для передачи данных между клиентом и сервером (стек протоколов *TCP/IP*), но нет жестких правил о том, как устанавливать соединение и структурировать передаваемое сообщение. А это влияет на скорость.

Для того, чтобы сформировать *HTTP*-запрос, необходимо составить стартовую строку, а также задать по крайней мере один заголовок — это заголовок *Host*, который является обязательным, и должен присутствовать в каждом запросе.

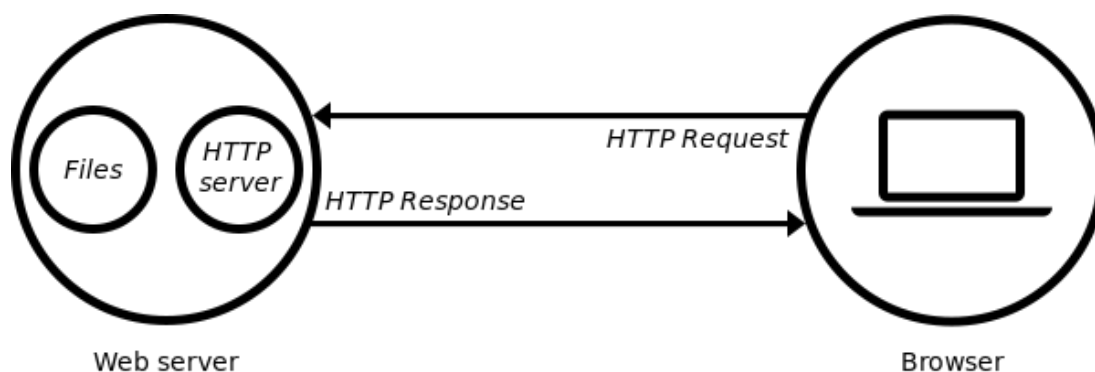


Рисунок 2.1 – Схема обмена сообщениями через протокол *HTTP*

## 2.2 Проектирование архитектуры приложения

Первым делом необходимо разработать архитектуру приложения. Приложение должно состоять из источника данных, слоя доступа к источнику данных, слоя моделей, представляющих данные из источника, слоя бизнес-логики для обработки полученных от источника и пользователя данных, слоя представлений для отображения данных и слоя контроллеров для организации взаимодействия между слоем представлений, доступа к данным и бизнес-логики.

Далее необходимо рассмотреть слой бизнес-логики. Так как данные, находящиеся в использовании в приложении, не нуждаются в дополнительной обработке, единственными классами, представляющими бизнес-логику, является сервис для работы с токенами аутентификации, а также компонент *middleware* для получения токенов. Сервис будет представлен статическим классом, осуществляющим обработку данных пользователя и токенов согласно настройкам в программе.

Для осуществления взаимодействия пользователя с программой необходимо реализовать слой представления. Так как приложение использует архитектуру *HTTP*, слой представления не привязан к приложению. Это значит, что к приложению можно применять любое представление. Тем не менее, для демонстрации необходимо создать страницы на языке *HTML*. Слой будет представлен страницами для регистрации, входа и обмена сообщениями с использованием языков *HTML* и *JavaScript*.

Для организации взаимодействия слоя представлений, слоя доступа к данным и слоя бизнес-логики необходимо создать слой контроллеров. Данный слой будет представлен двумя классами, являющимися контроллерами для работы с авторизацией и обменом сообщениями, а также модели представления для обмена данными.

Схема архитектуры приложения изображена в приложении Д.

## 2.3 Структурная схема приложения

Этапу написания любой программы, предшествует этап проектирования приложения. На этой стадии происходит анализ и выявление классов и их функциональности, что позволяет лучше понять их взаимодействие друг с другом. Поэтому, сначала следует спроектировать модель, основываясь на которой будет писаться сама программа.

В предыдущих главах была рассмотрена клиент-серверная архитектура, на основе которой и будет строиться всё приложение. Из этого следует, что в

программе должны быть компоненты, отвечающие за реализацию клиента и подключение к серверу, а также отвечающие за реализацию отправки и получения сообщений.

Для каждого подключения, серверу необходимо создать некий обработчик, который будет «общаться» непосредственно с клиентом и передавать именно те данные, которые требуется обработать.

Компонентами приложения являются:

- клиент;
- сервер.

Приложение будет состоять из двух основных подсистем: подсистема аутентификации пользователей и ролей и системы обмена сообщениями.

Подсистема аутентификации осуществляет следующий функционал:

- вход пользователя: подсистема получает данные от пользователя, осуществляет проверку данных и генерацию токена, после чего возвращает ответ пользователю в виде токена или сообщения об ошибке;
- выход пользователя: подсистема получает команду от пользователя и удаляет токен из системы;
- регистрация нового пользователя: подсистема получает данные от пользователя, производит их валидацию и в случае валидности полученных данных создает из полученных данных запись в базе данных;
- проверка данных для аутентификации: подсистема получает токен от пользователя, проверяет его на достоверность и возвращает ответ, подтверждающий или опровергающий правильность токена;
- получение роли: подсистема получает токен от пользователя и возвращает роль пользователя, содержащуюся в токене.

Подсистема обмена сообщениями имеет следующий функционал:

- получение списка пользователей: подсистема получает список всех пользователей и возвращает его всем пользователям для выбора получателя сообщения или администратору для выбора пользователя, чью историю сообщений он хочет прочесть;
- получение списка актуальных сообщений: пользователь делает запрос на получение последних сообщений, подсистема проверяет, может ли пользователь получить сообщение, и возвращает ответ со списком сообщений или сообщением об ошибке;
- отправка сообщений: пользователь отправляет данные подсистеме в виде формы, подсистема проверяет правильность заполнения формы и в случае положительного результата отправляет сообщение получателю, а также заносит сообщение в базу данных. Подсистема возвращает ответ

пользователю с сообщением об ошибке в случае неправильно заполненных данных.

Структурная схема приложения изображена на рисунке 2.2.



Рисунок 2.2 – Структурная схема приложения

## 2.4 Сетевая архитектура работы программного средства

Для генерации и валидации токенов необходимо определить сервис, связанный с токенами, необходимо реализовать статический класс *JwtAuthenticationService*. Данный метод содержит два статических метода:

- *GenerateAccessToken(Claim[] claims)* – метод получает список утверждений, связанных с пользователем, и на их основе генерирует токен, который валиден в течении 2 часов, после чего возвращает токен;
- *ValidateToken(string token)* – метод получает токен и производит его проверку по совпадению с присутствующими в программе параметрами. Результат проверки возвращается.

Для хранения параметров, связанных с генерацией и валидацией токенов, необходимо создать файл констант *JwtAuthConstants*. Данный статический класс содержит константы, связанные с ключом подписи, сервером, клиентом и названием схемы аутентификации.

## 2.5 Клиентская часть web-приложения

Уровень контроллеров состоит из двух классов, наследующих базовый класс *ControllerBase* и содержащих атрибут *ApiController*: *UserController* и *ChatController*.

*UserController* представляет собой контроллер аутентификации пользователя. Он хранит экземпляр интерфейса *IMapper*, представляющего из себя способ создания объектов одного класса из объектов другого класса, экземпляр интерфейса *IRepository<User>*, представляющего из себя репозиторий, работающий с таблицей пользователей, и экземпляр *IHubContext<ChatHub>*, представляющего из себя абстракцию контекста хаба *ChatHub*, а также реализующий следующие методы:

- *Task<IActionResult> Register(UserModel model)* – метод «POST», получающий данные от пользователя и производящий регистрацию с возвратом токена или возврат ошибки в случае некорректности данных, а также оповещения остальных пользователей о новом пользователе;

- *IActionResult Login(UserModel model)* – метод «POST», получающий данные от пользователя и производящий вход с возвратом токена или возврат ошибки в случае некорректности данных;

- *IActionResult CheckIfAuthenticate()* – метод «GET», проверяющий токен и возвращающий статус «Ок» в случае корректности токена или «Плохой запрос» в случае некорректности токена;

- *IActionResult GetRole()* – метод «GET», возвращающий роль пользователя;

- *IActionResult Logout()* – метод «DELETE», удаляющий токен.

*ChatController* представляет собой контроллер обмена сообщениями. Он хранит экземпляр *IHubContext<ChatHub>*, представляющего из себя абстракцию контекста хаба *ChatHub*, а также экземпляры интерфейсов *IRepository<Message>* и *IRepository<User>*, представляющие из себя репозитории. Данный контроллер также реализует следующие методы:

- *Task<ActionResult> Get(string nickname = null)* – метод «GET», возвращающий историю текущего пользователя. Если параметр *nickname* не является пустым и пользователь является администратором, метод возвращает историю выбранного пользователя;

- *Task<IActionResult> Post()* – метод «POST», получающий из запроса заполненную пользователем форму сообщения, производящего валидацию и, в случае корректности введенных данных, сохраняет сообщение в базе данных и отправляет выбранному в форме пользователю экземпляр сообщения. В



случае некорректности данных метод возвращает ответ с сообщением об ошибке.

Для определения получателя сообщения контекст хаба по умолчанию использует сгенерированный во время подключения идентификационный указатель, называемый *connectionId*. Для организации логического взаимодействия пользователей в среде хаба, необходимо переопределить метод получения идентификационного указателя пользователя. Для этого необходимо определить класс *CustomUserIdProvider*, наследующий предоставленный библиотекой *SignalR* интерфейс *IUserIdProvider*. В данном классе необходимо переопределить метод *GetUserId(HubConnectionContext connection)*, возвращающий *id* пользователя из экземпляра соединения, чтобы он возвращал никнейм пользователя, после чего внедрить провайдер в приложение.

Для создания из объектов одного класса объектов другого класса необходимо определить профиль маппера, используемого в приложении. Для этого необходимо определить класс *MapperProfile*, наследующий базовый класс *Profile*, предоставленный библиотекой *AutoMapper*, и создающий карту преобразования объекта *UserModel* в объект *User*.

Доступ к большинству методов контроллеров требует авторизации. Для авторизации необходимо получить из запроса токен аутентификации и провести валидацию, на основании которой пользователю будет отказано в доступе к методу, или же пользователь получит доступ. Так как архитектура только ограничивает способ получения от пользователя токена авторизации, можно реализовать общий метод аутентификации в *Web API*. Для этого необходимо реализовать обработчик аутентификации на основе *JWT*-токенов *JwtAuthenticationHandler*. Данный класс наследует базовый обработчик *AuthenticationHandler<T>* и переопределяет метод базового класса *Task<AuthenticateResult> HandleAuthenticateAsync()*. Данный метод получает токен из заголовков запроса и валидирует его. При благоприятном исходе внедряет в систему экземпляр класса *AuthenticationTicket*, представляющего собой объект хранения данных о пользователе; в противном случае возвращает ошибку аутентификации. Благодаря этому для ограничения доступа к методам контроллера достаточно указать атрибут «*Authorize*» с заданной схемой, а для получения информации о пользователе – использовать объект *User*.

Токен невозможно получить из *cookie* через код *JavaScript*, так как они защищены настройками приложения. Для доступа к ним необходимо определить компонент *middleware*, который при наличии токена в *cookie* перемещал бы его в запрос. Класс *AuthenticationMiddleware* содержит

экземпляр объекта *RequestDelegate* и реализует метод *Invoke(HttpContext context)*, помещающий токен из *cookie* в заголовок запроса.

Дополнительные классы необходимо внедрить в приложение. Для этого в классе *Startup* нужно вызвать методы:

- *IServiceCollection.AddSignalR()* – добавляет сервис библиотеки *SignalR*;
- *IServiceCollection.AddAuthentication(string defaultScheme).AddScheme<TOptions, THandler>(string authenticationScheme, null)* – добавляет аутентификацию с заданной схемой и обработчиком в приложение;
- *IServiceCollection.AddAutoMapper(Type type)* – добавляет маппер с конфигурацией, привязанной к данному типу;
- *IApplicationBuilder.UseMiddleware<T>()* – внедряет экземпляр объекта *middleware* в приложение.

Для отображения результата работы и создания шаблона приложения в папке *wwwroot* необходимо создать три страницы: *Chat.html*, *Register.html*, *Login.html*.

Страница *Login.html* содержит форму входа с полями для ввода никнейма и пароля, ссылку на страницу регистрации, а также кнопку вызова функции. Кроме того, страница содержит ссылки на библиотеки *bootstrap*, файлы *css* и библиотеку *jQuery*. Для реализации вызова методов контроллера *UserController* страница содержит привязку кнопки отправки формы к *ajax*-функции вызова метода регистрации. Данная функция отправляет данные формы и получает ответ от сервера, после чего отображает его.

Страница *Register.html* в большинстве своем идентична предыдущей странице. Отличие заключается в замене ссылки на форму регистрации ссылкой на форму входа, а также в замене вызываемого метода на сервере на метод регистрации.

Страница *Chat.html* в добавок к названным библиотекам имеет ссылку на библиотеку *SignalR*, необходимую для организации взаимодействия с сервером в режиме реального времени. Данная страница содержит скрытую форму отображения истории пользователя со списком пользователей, форму для ввода данных сообщения, кнопку отправки формы серверу, ссылки на отображение истории сообщений, кнопку «Выход» и поле для отображения сообщений, связанных с пользователем.

При посещении пользователем страницы происходит вызов *ajax*-функции получения списка пользователей и сообщений.

При нажатии на ссылку «Показать историю» чат с формой сообщения заменяется полем с историей сообщения, а использованная ссылка заменяется на ссылку «Вернуться к чату». Если пользователь является администратором, также появляется поле выбора пользователя.

При нажатии на кнопку «Выход» происходит выход пользователя из учетной записи и переход на страницу авторизации.

При нажатии на кнопку «Отправить сообщение» происходит вызов *ajax*-функции отправки формы сообщения серверу для регистрации нового сообщения, а также добавление нового сообщения в конец уже существующих.

При выборе пользователя в форме для историй сообщений и нажатии на кнопку «Изменить пользователя» происходит вызов *ajax*-функции, возвращающей с сервера историю сообщений для выбранного пользователя.

Так как приложение использует библиотеку *SignalR* на основе протокола *WebSocket*, необходимо организовать постоянное соединение с сервером и реакцию на события, вызванные на сервере. Для этого необходимо создать соединение с хабом с помощью строки `const hubConnection = new signalR.HubConnection().withUrl(hub url).build()`.

После этого необходимо определить реакцию на действия хаба. Для этого нужно вызвать функцию `hubConnection.on(actionName, function())`, где *actionName* – название метода, вызванного на стороне сервера, а *function()* – функция вызываемая при получении оповещения от сервера. Необходимо определить, что при оповещении клиента о новом пользователе нужно добавить имя пользователя в список доступных для общения клиентов, а при оповещении о новом сообщении – с помощью полученных из оповещения данных добавить новое сообщение в начало списка сообщений.

Чтобы начать соединение, необходимо во время входа на страницу вызвать метод `hubConnection.start()`, а для отключения пользователя от общего хаба при выходе из учетной записи необходимо привязать команду `hubConnection.stop()` к нажатию на кнопку «Выход».

## 2.6 Клиентская часть десктопного приложения

Клиент, реализованный с помощью класса *Client.cs*, отправляет запрос на сервер, который реализован с помощью класса *Server.cs*, сервер проверяет наличие пользователя в базе данных, при отсутствии информации о клиенте в базе данных, сервер создает пользователя с информацией о клиенте, далее сервер отправляет ответ клиенту. Доступ к серверу реализован в классе *Client.cs* Подробное описание классов приведено ниже.

Аутентификацию и регистрацию реализует файл *Client.cs*.

Класс *Client.cs* содержит функцию *Chats()*, которая возвращает окно многопользовательского чата. Функция *Chats()* использует импортированные методы из других классов, осуществляет взаимодействие пользователя с

графическим интерфейсом. Асинхронная функция *getFile()* позволяет получать отправленные пользователем картинки. Также с помощью запроса функция получает заголовки, принадлежащие пользователю, если их нет, создает нового пользователя и передает его данные на сервер.

Приложение работает синхронно, то есть при выполнении операции программа ждет и не переходит к выполнению следующего действия.

Синхронизация приложения выполнена с помощью асинхронных функций. Асинхронная функция *handleLogout()* отвечает за выход из аккаунта пользователя. При выходе из аккаунта информация о пользователе обновляется у всех. Также синхронизация происходит посредством функции *setSendingMessage()*. После отправки сообщения пользователем, все остальные пользователи получают отправленное сообщение.

Сетевые коммуникации происходят посредством *HTTP*. *HTTP* отправляет запрос на сервер, для получения ответа, если клиент не получил ответ, *HTTP* снова отправляет запрос, и так до того момента, пока клиент не получит ответ от сервера.

Код всего приложения находится в приложении А.

## 3 ВЕРИФИКАЦИЯ РАЗРАБОТАННОГО ПРИЛОЖЕНИЯ МНО- ГОПОЛЬЗОВАТЕЛЬСКОГО ЧАТА

### 3.1 Модульное тестирование

В первую очередь необходимо провести модульное тестирование тех из классов, которые допускают внедрение *mock*-объектов. Моки являются подмножеством тестовых заглушек (*test doubles*). Термины «тестовая заглушка (*test double*)» и «мок (*mock*)» часто используются как синонимы, но на самом деле это не так:

- тестовая заглушка (*test double*) – общий термин, описывающий любые разновидности фиктивных зависимостей, используемых в тестах;
- мок (*mock*) – всего лишь одна из разновидностей зависимостей [6].

Классами, подходящими под использование объектов-заглушек являются: *JwtAuthenticationService* и некоторые методы класса *ChatController*.

В классе *JwtAuthenticationService* тестируются следующие методы:

- *GenerateAccessToken* – метод генерации токена на основе данных пользователя;
- *ValidateToken* – метод валидации токена на основе параметров сервера.

На основании существующих методов класса присутствуют следующие методы тестирования:

- *void OneTimeSetUp()* – метод создания параметров для генерации и валидации токена;
- *void GenerateAccessToken\_WhenAllDataAreValid\_ShouldReturnToken()* – метод проверки создания токена на основе данных администратора;
- *void ValidateToken\_WhenAllDataAreValid\_ShouldReturnTrue()* – метод проверки валидации, когда сервис получает валидный токен;
- *void ValidateToken\_WhenDataAreNotValid\_ShouldReturnFalse()* – метод проверки валидации, когда токен не валиден.

В классе *ChatController* тестируются следующие методы:

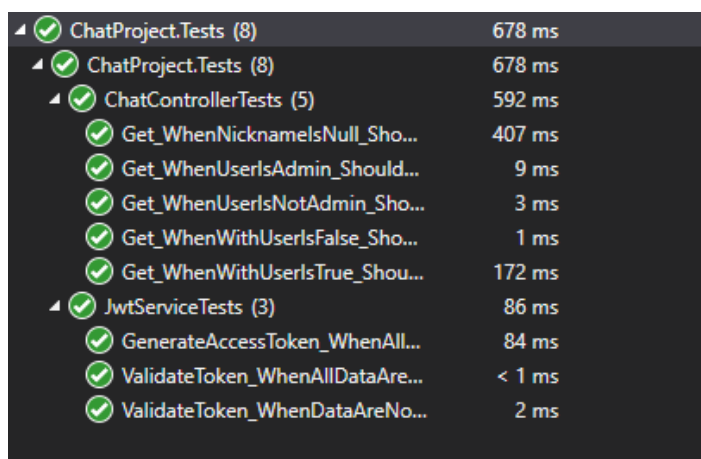
- *Get* (путь «*chat/ListOfUsers*») – метод получения списка пользователей;
- *Get* (путь «*chat/messages*») – метод получения сообщений, связанных с пользователем.

На основании существующих методов класса присутствуют следующие методы тестирования:

- *void Get\_WhenWithUserIsFalse\_ShouldReturnListOfUsersWithoutUser* – метод проверки возврата списка пользователей, когда никнейм нынешнего пользователя не требуется;

- `void Get_WhenWithUserIsTrue_ShouldReturnListOfUsersWithUser()` – метод проверки возврата списка пользователей, когда никнейм нынешнего пользователя требуется;
- `Get_WhenNicknameIsNull_ShouldReturnListOfCurrentUserMessages()` – метод проверки возврата списка сообщений, когда нужно вернуть список текущего пользователя;
- `Get_WhenUserIsNotAdmin_ShouldReturnListOfCurrentUserMessages()` – метод проверки возврата списка сообщений любого пользователя, когда пользователь не является администратором;
- `Get_WhenUserIsAdmin_ShouldReturnListOfChoosenUserMessages()` – метод проверки возврата списка сообщений любого пользователя, когда пользователь является администратором.

Список тестов с результатами их запуска изображен на рисунке 3.1.



ChatProject.Tests (8)	678 ms
ChatProject.Tests (8)	678 ms
ChatControllerTests (5)	592 ms
Get_WhenNicknamesIsNull_Sho...	407 ms
Get_WhenUserIsAdmin_Should...	9 ms
Get_WhenUserIsNotAdmin_Sho...	3 ms
Get_WhenWithUserIsFalse_Sho...	1 ms
Get_WhenWithUserIsTrue_Shou...	172 ms
JwtServiceTests (3)	86 ms
GenerateAccessToken_WhenAll...	84 ms
ValidateToken_WhenAllDataAre...	< 1 ms
ValidateToken_WhenDataAreNo...	2 ms

Рисунок 3.1 – Список тестов

### 3.2 Нагрузочное тестирование

Для нагрузочного тестирования необходимо написать классы, тестирующие работу чата с большим количеством пользователей, или же использовать специальные ресурсы, позволяющие провести нагрузочное тестирование сайта. Предпочтительным является последний вариант, так как данные ресурсы могут рассчитать все основные параметры и эмулировать работу большого количества пользователей.

В качестве ресурса для тестирования чата был выбран сайт *loaddy.com*, который способен произвести стресс-тест с пятьюдесятью пользователями, после чего вывести основные показатели работы сайта. Результатом тестов являются графики и показатели.

На первом графике изображено количество успешных ответов. На втором графике изображены скорость загрузки страницы и время ответа в каждый момент времени тестирования. Над графиками изображены четыре основных показателя: доступность сайта, скорость загрузки сайта, время ответа на запросы и полученное при загрузке сайта количество данных.

Результаты теста изображены на рисунке 3.2.

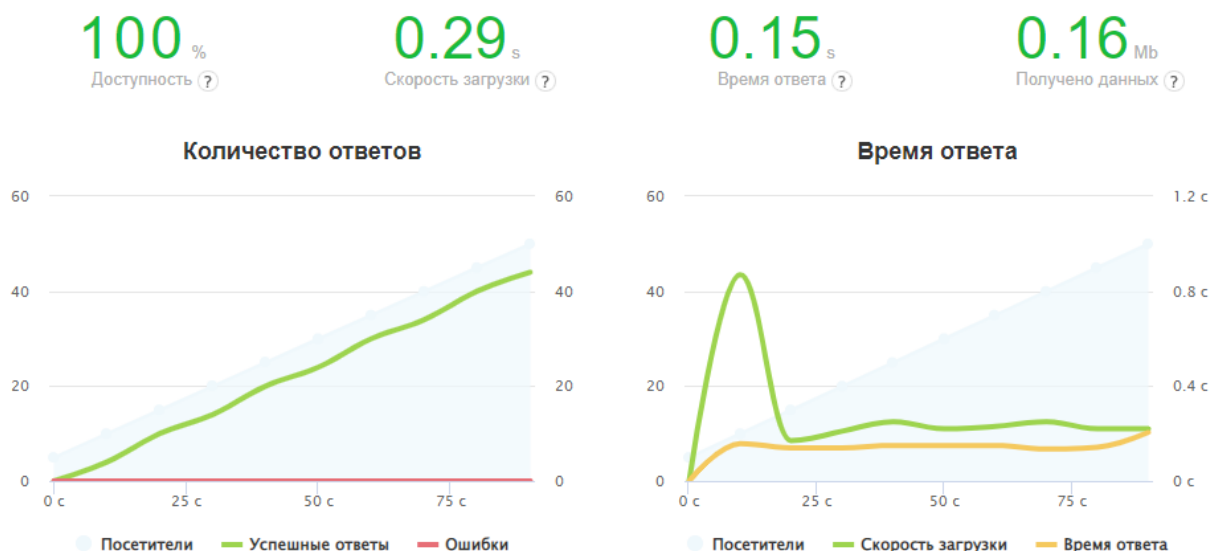


Рисунок 3.2 – Результаты нагрузочного теста

Для показания реальных результатов приложение было запущено на десяти устройствах, после чего в течении около десяти минут пользователи безостановочно отправляли друг другу сообщения, изображения, открывали историю сообщения, а некоторые из них время от времени осуществляли выход из учетной записи и вход.

При данной нагрузке приложение показало хорошие результаты, обеспечивая отправку сообщений с максимальной задержкой примерно в полсекунды. Благодаря библиотеке *SignalR*, оптимизированной для больших нагрузок, и сжатии изображений доставка приложений происходит максимально быстро.

### 3.3 Тестирование запущенного приложения

Для работоспособности чата, в начале необходимо запустить *server*, ввести *IP*-адрес, порт и пароль и нажать кнопку «*Start*». После нажатия кнопки, в окне вывода сообщений появится сообщение о том, что сервер был запущен.

Внешний вид приложения *server* представлен на рисунке 3.3.

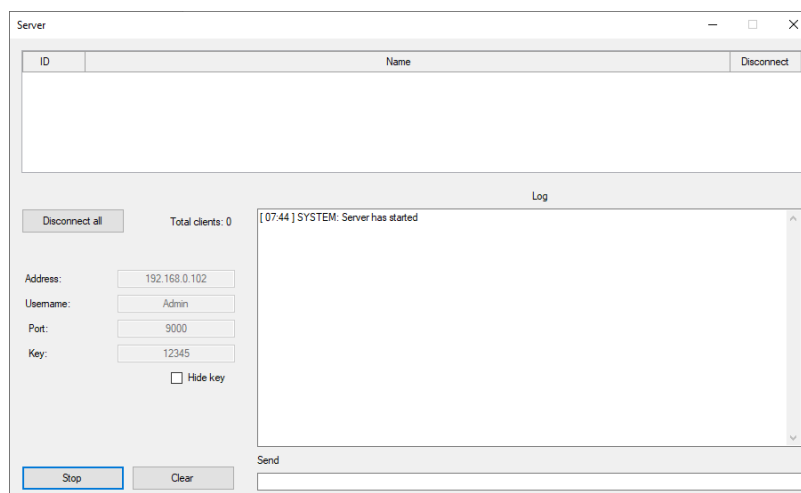


Рисунок 3.3 – Внешний вид приложения *server*

Для входа и полноценного взаимодействия с приложением пользователю необходимо в своем приложении ввести *IP*-адрес и порт сервера, ввести имя пользователя и пароль сервера и нажать кнопку «*Connect*». Если все данные были введены корректно, то по нажатию этой кнопки для пользователя в окне вывода сообщений будет выведено сообщение о том, что он подключился к чату, в противном случае будет выведено сообщение с ошибкой. Вывод сообщения о подключении к чату изображен на рисунке 3.4.



Рисунок 3.4 – Вывод сообщения о подключении пользователя к чату

Также предусмотрена кнопка выхода из аккаунта пользователя. Внешний вид кнопки выхода представлен на рисунке 3.5.



Рисунок 3.5 – Кнопка выхода из аккаунта пользователя



После входа пользователю доступен весь функционал приложения. В правой части экрана приложения *server* находится элемент, отображающий информацию об количестве подключенных пользователей. В правой части этого элемента есть кнопка «*Kick*» позволяющая выгнать пользователя из чата. Внешний вышеописанного элемента, а также содержимое выпадающих списков представлено на рисунке 3.6.

ID	Name	Disconnect
1	Client1	Kick
2	Client2	Kick

Рисунок 3.6 – Информация о чате

Для проверки функции удаления пользователя из чата и необходимо нажать на соответствующую кнопку. По нажатию кнопок пользователь будет удален из списка пользователей и больше не будет иметь доступ к чату, владельца чат нельзя удалить из чата. Результат удаления пользователя из чата и удаления чата представлен на рисунке 3.7.

The screenshot shows a window titled "Server" with a standard Windows interface (minimize, maximize, close buttons). Inside the window, there is a table with columns "ID", "Name", and "Disconnect". The table contains one row: ID "1", Name "Client2", and a "Kick" button. Below the table is a "Log" section with a scrollable text area containing the following messages: [ 08:14 ] SYSTEM: Server has started, [ 08:14 ] SYSTEM: Client1 has connected, [ 08:14 ] SYSTEM: Client2 has connected, and [ 08:14 ] SYSTEM: Client1 has disconnected. To the left of the log, there is a "Disconnect all" button and a "Total clients: 1" label. Below these are input fields for "Address" (192.168.0.102), "Username" (Server), "Port" (9000), and "Key" (12345), along with a "Hide key" checkbox. At the bottom left are "Stop" and "Clear" buttons. At the bottom right is a "Send" label and an empty input field.

Рисунок 3.7 – Результат удаления чата и пользователя из чата

В центральной части экрана расположен элемент, отображающий все сообщения чата, как входящие, так и исходящие. Слева от каждого сообщения написано имя отправителя. Внизу находится поле для ввода сообщений. Для того, чтобы отправить сообщение, пользователь должен ввести его в соответствующее поле и нажать на кнопку «*Enter*» на своей клавиатуре. После проделанных действий сообщение появится в окне отображения сообщений.

Для проверки функционала отправки сообщения необходимо перейти в чат, в поле, предназначенном для ввода сообщений, ввести необходимое для отправки сообщение и нажать кнопку для отправки сообщения. По нажатию кнопки сообщение появится в окне отображения сообщений с соответствующей датой и временем отправки. Для других пользователей будет отображено имя отправителя. Результат отправки и получения сообщения представлен на рисунке 3.8.

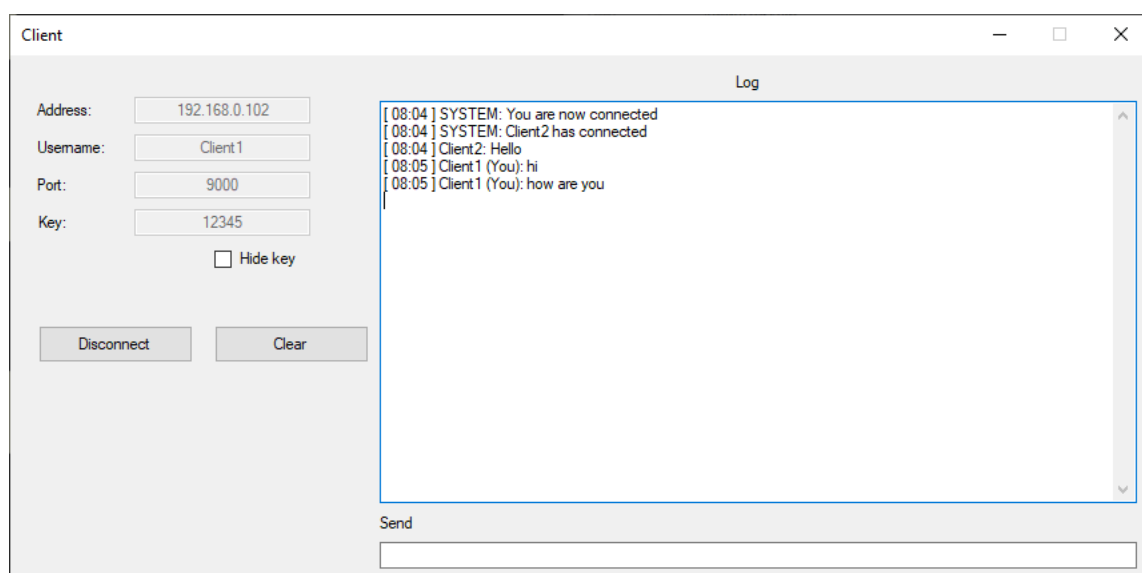


Рисунок 3.8 – Элемент отображения и отправки сообщений

Пользовательский интерфейс приложения очень прост и многофункционален. Благодаря этому у пользователя не возникнет проблем при взаимодействии с приложением. Пользовательский интерфейс информирует пользователя о всех происходящих процессах, что не вводит пользователя в заблуждение при ожидании загрузки или переподключении к серверу, также пользователь видит сообщения при возникновении ошибок.

### **3.4 Результаты тестирования приложения**

Модульные тесты, изображенные на рисунке 3.1, показали, что сервисы для работы с токенами и репозиторий работают корректно, получают данные без потерь и производят требуемые от них манипуляции.

Тестирование интерфейса, проведенное в подпункте 3.3, показало, что все данные отображаются корректно, взаимодействие между пользователем и программой осуществляются по ожидаемым сценариям, а сам интерфейс достаточно дружелюбен для пользователя.

Нагрузочные тесты, результаты которых показаны в подпункте 3.2, подтверждают, что программа обеспечивает стабильное соединение и быстрый обмен сообщениями при интенсивной нагрузке сервера.

Результаты проверки приложения дают гарантию корректности работы приложения и стабильности при больших нагрузках сервера.

## ЗАКЛЮЧЕНИЕ

При выполнении курсового проекта создано сетевое приложение реализации многопользовательского чата. Данное приложение обеспечивает связь с базой данных, отправку и получение текстовых сообщений, отправку и получение изображений, аутентификацию и авторизацию, а также историю сообщений для базового пользователя и историю сообщений любого пользователя для администратора.

Благодаря реализации архитектуры *HTTP* и легко разделяемой иерархии классов приложение доступно для модификаций и улучшений. Также данное приложение легко внедряется в другие приложения и сайты, позволяя реализовать многопользовательский чат в уже готовой архитектуре приложений.

После проведения модульных и нагрузочных тестов была доказана работоспособность данной программы и функционирование всех требуемых ролей и методов. Полученное приложение готово для работы со среднестатистическим пользователем.

Данное приложение предназначено для предприятий, которые стремятся облегчить взаимодействие между работниками, а также небольших сайтов и приложений, которым требуется способ легко и быстро внедрить аналог многопользовательского чата без значимых изменений уже готовых программ. Разработанный продукт станет одним из преимуществ в конкурентной борьбе предприятий и сайтов.

Работа была проверена в системе «Антиплагиат», процент уникальности составил 82,53%.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Дубаков А.А. Сетевое программирование: учебное пособие / А.А. Дубаков – СПб: НИУ ИТМО, 2017. – 248 с.
2. Что такое чат: разновидности и история возникновения. – Электрон. данные. – Режим доступа: <https://ktonanovenkogo.ru/voprosy-i-otvety/chat-cto-eh-to-takoe.html>. – Дата доступа: 07.12.2021.
3. Введение в сетевое программирование. – Электрон. данные – Режим доступа: – <https://russianblogs.com/article/6258464747/> – Дата доступа: 07.12.2021.
4. Столлингс, В. Компьютерные сети, протоколы и технологии Интернета / В. Столлингс. – СПб.: BHV, 2005. - 832 с.
5. Олифер В.Г. Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов / В.Г. Олифер, Н.А. Олифер. – 4-е. изд. – СПб: Питер, 2007. – 960 с.
6. Руденков Н.А., Основы сетевых технологий: Учебник для вузов / Н.А. Руденков, Л.И. Долинер. – Екатеринбург : Изд-во Уральского Федерального ун-та им. Б.Н. Ельцина, 2011. – 342 с.

## ПРИЛОЖЕНИЕ А

(обязательное)

### Листинг программного комплекса

#### Server.cs

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Web.Script.Serialization;
using System.Windows.Forms;

namespace Server
{
    public partial class Server : Form
    {
        private bool active = false;
        private Thread listener = null;
        private long id = 0;
        private struct MyClient
        {
            public long id;
            public StringBuilder username;
            public HttpClient client;
            public NetworkStream stream;
            public byte[] buffer;
            public StringBuilder data;
            public EventWaitHandle handle;
        };
        private ConcurrentDictionary<long, MyClient> clients = new ConcurrentDictionary<long, MyClient>();
        private Task send = null;
        private Thread disconnect = null;
        private bool exit = false;

        public Server()
        {
            InitializeComponent();
        }

        private void Log(string msg = "")
        {

```

```

        if (!exit)
        {
            logTextBox.Invoke((MethodInvoker)delegate
            {
                if (msg.Length > 0)
                {
                    logTextBox.AppendText(string.Format("[{0} {1} {2}]",
DateTime.Now.ToString("HH:mm"), msg, Environment.NewLine));
                }
                else
                {
                    logTextBox.Clear();
                }
            });
        }
    }

    private string ErrorMsg(string msg)
    {
        return string.Format("ERROR: {0}", msg);
    }

    private string SystemMsg(string msg)
    {
        return string.Format("SYSTEM: {0}", msg);
    }

    private void Active(bool status)
    {
        if (!exit)
        {
            startButton.Invoke((MethodInvoker)delegate
            {
                active = status;
                if (status)
                {
                    addrTextBox.Enabled = false;
                    portTextBox.Enabled = false;
                    usernameTextBox.Enabled = false;
                    keyTextBox.Enabled = false;
                    startButton.Text = "Stop";
                    Log(SystemMsg("Server has started"));
                }
                else
                {
                    addrTextBox.Enabled = true;
                    portTextBox.Enabled = true;
                    usernameTextBox.Enabled = true;
                    keyTextBox.Enabled = true;
                }
            });
        }
    }

```

```

        startButton.Text = "Start";
        Log(SystemMsg("Server has stopped"));
    }
});
}
}

private void AddToGrid(long id, string name)
{
    if (!exit)
    {
        clientsDataGridView.Invoke((MethodInvoker)delegate
        {
            string[] row = new string[] { id.ToString(), name };
            clientsDataGridView.Rows.Add(row);
            totalLabel.Text = string.Format("Total clients: {0}", clientsDataGridView.Rows.Count);
        });
    }
}

private void RemoveFromGrid(long id)
{
    if (!exit)
    {
        clientsDataGridView.Invoke((MethodInvoker)delegate
        {
            foreach (DataGridViewRow row in clientsDataGridView.Rows)
            {
                if (row.Cells["identifier"].Value.ToString() == id.ToString())
                {
                    clientsDataGridView.Rows.RemoveAt(row.Index);
                    break;
                }
            }
            totalLabel.Text = string.Format("Total clients: {0}", clientsDataGridView.Rows.Count);
        });
    }
}

private void Read(IAsyncResult result)
{
    MyClient obj = (MyClient)result.AsyncState;
    int bytes = 0;
    if (obj.client.Connected)
    {
        try
        {
            bytes = obj.stream.EndRead(result);
        }
    }
}

```



```

        catch (Exception ex)
        {
            Log(ErrorMsg(ex.Message));
        }
    }
    if (bytes > 0)
    {
        obj.data.AppendFormat("{0}", Encoding.UTF8.GetString(obj.buffer, 0, bytes));
        try
        {
            if (obj.stream.DataAvailable)
            {
                obj.stream.BeginRead(obj.buffer, 0, obj.buffer.Length, new AsyncCallback(Read), obj);
            }
            else
            {
                string msg = string.Format("{0}: {1}", obj.username, obj.data);
                Log(msg);
                Send(msg, obj.id);
                obj.data.Clear();
                obj.handle.Set();
            }
        }
        catch (Exception ex)
        {
            obj.data.Clear();
            Log(ErrorMsg(ex.Message));
            obj.handle.Set();
        }
    }
    else
    {
        obj.client.Close();
        obj.handle.Set();
    }
}

private void ReadAuth(IAsyncResult result)
{
    MyClient obj = (MyClient)result.AsyncState;
    int bytes = 0;
    if (obj.client.Connected)
    {
        try
        {
            bytes = obj.stream.EndRead(result);
        }
        catch (Exception ex)
        {

```

```

        Log(ErrorMsg(ex.Message));
    }
}
if (bytes > 0)
{
    obj.data.AppendFormat("{0}", Encoding.UTF8.GetString(obj.buffer, 0, bytes));
    try
    {
        if (obj.stream.DataAvailable)
        {
            obj.stream.BeginRead(obj.buffer, 0, obj.buffer.Length, new AsyncCallback(ReadAuth),
obj);
        }
        else
        {
            JavaScriptSerializer json = new JavaScriptSerializer(); // feel free to use JSON serializer
            Dictionary<string, string> data = json.Deserialize<Dictionary<string,
string>>(obj.data.ToString());
            if (!data.ContainsKey("username") || data["username"].Length < 1 || !data.Contains-
Key("key") || !data["key"].Equals(keyTextBox.Text))
            {
                obj.client.Close();
            }
            else
            {
                obj.username.Append(data["username"].Length > 200 ? data["username"].Substring(0,
200) : data["username"]);
                Send("{\"status\": \"authorized\"}", obj);
            }
            obj.data.Clear();
            obj.handle.Set();
        }
    }
    catch (Exception ex)
    {
        obj.data.Clear();
        Log(ErrorMsg(ex.Message));
        obj.handle.Set();
    }
}
else
{
    obj.client.Close();
    obj.handle.Set();
}
}

private bool Authorize(MyClient obj)
{

```

```

bool success = false;
while (obj.client.Connected)
{
    try
    {
        obj.stream.BeginRead(obj.buffer, 0, obj.buffer.Length, new AsyncCallback(ReadAuth), obj);
        obj.handle.WaitOne();
        if (obj.username.Length > 0)
        {
            success = true;
            break;
        }
    }
    catch (Exception ex)
    {
        Log(ErrorMsg(ex.Message));
    }
}
return success;
}

private void Connection(MyClient obj)
{
    if (Authorize(obj))
    {
        clients.TryAdd(obj.id, obj);
        AddToGrid(obj.id, obj.username.ToString());
        string msg = string.Format("{0} has connected", obj.username);
        Log(SystemMsg(msg));
        Send(SystemMsg(msg), obj.id);
        while (obj.client.Connected)
        {
            try
            {
                obj.stream.BeginRead(obj.buffer, 0, obj.buffer.Length, new AsyncCallback(Read), obj);
                obj.handle.WaitOne();
            }
            catch (Exception ex)
            {
                Log(ErrorMsg(ex.Message));
            }
        }
        obj.client.Close();
        clients.TryRemove(obj.id, out MyClient tmp);
        RemoveFromGrid(tmp.id);
        msg = string.Format("{0} has disconnected", tmp.username);
        Log(SystemMsg(msg));
        Send(msg, tmp.id);
    }
}

```

```

}

private void Listener(IPAddress ip, int port)
{
    HttpListener listener = null;
    try
    {
        listener = new HttpListener(ip, port);
        listener.Start();
        Active(true);
        while (active)
        {
            if (listener.Pending())
            {
                try
                {
                    MyClient obj = new MyClient();
                    obj.id = id;
                    obj.username = new StringBuilder();
                    obj.client = listener.AcceptHttpClient();
                    obj.stream = obj.client.GetStream();
                    obj.buffer = new byte[obj.client.ReceiveBufferSize];
                    obj.data = new StringBuilder();
                    obj.handle = new EventWaitHandle(false, EventResetMode.AutoReset);
                    Thread th = new Thread(() => Connection(obj))
                    {
                        IsBackground = true
                    };
                    th.Start();
                    id++;
                }
                catch (Exception ex)
                {
                    Log(ErrorMsg(ex.Message));
                }
            }
            else
            {
                Thread.Sleep(500);
            }
        }
        Active(false);
    }
    catch (Exception ex)
    {
        Log(ErrorMsg(ex.Message));
    }
    finally
    {

```

```

        if (listener != null)
        {
            listener.Server.Close();
        }
    }
}

private void StartButton_Click(object sender, EventArgs e)
{
    if (active)
    {
        active = false;
    }
    else if (listener == null || !listener.IsAlive)
    {
        string address = addrTextBox.Text.Trim();
        string number = portTextBox.Text.Trim();
        string username = usernameTextBox.Text.Trim();
        bool error = false;
        IPAddress ip = null;
        if (address.Length < 1)
        {
            error = true;
            Log(SystemMsg("Address is required"));
        }
        else
        {
            try
            {
                ip = Dns.Resolve(address).AddressList[0];
            }
            catch
            {
                error = true;
                Log(SystemMsg("Address is not valid"));
            }
        }
        int port = -1;
        if (number.Length < 1)
        {
            error = true;
            Log(SystemMsg("Port number is required"));
        }
        else if (!int.TryParse(number, out port))
        {
            error = true;
            Log(SystemMsg("Port number is not valid"));
        }
        else if (port < 0 || port > 65535)
    }
}

```

```

        {
            error = true;
            Log(SystemMsg("Port number is out of range"));
        }
        if (username.Length < 1)
        {
            error = true;
            Log(SystemMsg("Username is required"));
        }
        if (!error)
        {
            listener = new Thread(() => Listener(ip, port))
            {
                IsBackground = true
            };
            listener.Start();
        }
    }
}

private void Write(IAsyncResult result)
{
    MyClient obj = (MyClient)result.AsyncState;
    if (obj.client.Connected)
    {
        try
        {
            obj.stream.EndWrite(result);
        }
        catch (Exception ex)
        {
            Log(ErrorMsg(ex.Message));
        }
    }
}

private void BeginWrite(string msg, MyClient obj)
{
    byte[] buffer = Encoding.UTF8.GetBytes(msg);
    if (obj.client.Connected)
    {
        try
        {
            obj.stream.BeginWrite(buffer, 0, buffer.Length, new AsyncCallback(Write), obj);
        }
        catch (Exception ex)
        {
            Log(ErrorMsg(ex.Message));
        }
    }
}

```

```

    }
}

private void BeginWrite(string msg, long id = -1)
{
    byte[] buffer = Encoding.UTF8.GetBytes(msg);
    foreach (KeyValuePair<long, MyClient> obj in clients)
    {
        if (id != obj.Value.id && obj.Value.client.Connected)
        {
            try
            {
                obj.Value.stream.BeginWrite(buffer, 0, buffer.Length, new AsyncCallback(Write),
obj.Value);
            }
            catch (Exception ex)
            {
                Log(ErrorMsg(ex.Message));
            }
        }
    }
}

private void Send(string msg, MyClient obj)
{
    if (send == null || send.IsCompleted)
    {
        send = Task.Factory.StartNew(() => BeginWrite(msg, obj));
    }
    else
    {
        send.ContinueWith(antecedent => BeginWrite(msg, obj));
    }
}

private void Send(string msg, long id = -1)
{
    if (send == null || send.IsCompleted)
    {
        send = Task.Factory.StartNew(() => BeginWrite(msg, id));
    }
    else
    {
        send.ContinueWith(antecedent => BeginWrite(msg, id));
    }
}

private void SendTextBox_KeyDown(object sender, KeyEventArgs e)
{

```

```

if (e.KeyCode == Keys.Enter)
{
    e.Handled = true;
    e.SuppressKeyPress = true;
    if (sendTextBox.Text.Length > 0)
    {
        string msg = sendTextBox.Text;
        sendTextBox.Clear();
        Log(string.Format("{0} (You): {1}", usernameTextBox.Text.Trim(), msg));
        Send(string.Format("{0}: {1}", usernameTextBox.Text.Trim(), msg));
    }
}

private void Disconnect(long id = -1)
{
    if (disconnect == null || !disconnect.IsAlive)
    {
        disconnect = new Thread(() =>
        {
            if (id >= 0)
            {
                clients.TryGetValue(id, out MyClient obj);
                obj.client.Close();
                RemoveFromGrid(obj.id);
            }
            else
            {
                foreach (KeyValuePair<long, MyClient> obj in clients)
                {
                    obj.Value.client.Close();
                    RemoveFromGrid(obj.Value.id);
                }
            }
        })
    {
        IsBackground = true
    };
    disconnect.Start();
}

private void DisconnectButton_Click(object sender, EventArgs e)
{
    Disconnect();
}

private void Server_FormClosing(object sender, FormClosingEventArgs e)
{

```



```

        exit = true;
        active = false;
        Disconnect();
    }

    private void ClientsDataGridView_CellClick(object sender, DataGridViewCellEventArgs e)
    {
        if (e.RowIndex >= 0 && e.ColumnIndex == clientsDataGridView.Columns["dc"].Index)
        {
            long.TryParse(clientsDataGridView.Rows[e.RowIndex].Cells["identifier"].Value.ToString(),
out long id);
            Disconnect(id);
        }
    }

    private void ClearButton_Click(object sender, EventArgs e)
    {
        Log();
    }

    private void CheckBox_CheckedChanged(object sender, EventArgs e)
    {
        if (keyTextBox.PasswordChar == '*')
        {
            keyTextBox.PasswordChar = '\0';
        }
        else
        {
            keyTextBox.PasswordChar = '*';
        }
    }

    private void Server_Load(object sender, EventArgs e)
    {

    }

    private void totalLabel_Click(object sender, EventArgs e)
    {

    }
}

```

### **Client.cs**

```

using System;
using System.Collections.Generic;
using System.Net;

```

```

using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Web.Script.Serialization;
using System.Windows.Forms;

namespace Client
{
    public partial class Client : Form
    {
        private bool connected = false;
        private Thread client = null;
        private struct MyClient
        {
            public string username;
            public string key;
            public HttpClient client;
            public NetworkStream stream;
            public byte[] buffer;
            public StringBuilder data;
            public EventWaitHandle handle;
        };
        private MyClient obj;
        private Task send = null;
        private bool exit = false;

        public Client()
        {
            InitializeComponent();
        }

        private void Log(string msg = "") // clear the log if message is not supplied or is empty
        {
            if (!exit)
            {
                logTextBox.Invoke((MethodInvoker)delegate
                {
                    if (msg.Length > 0)
                    {
                        logTextBox.AppendText(string.Format("[{0} {1} {2}]",
DateTime.Now.ToString("HH:mm"), msg, Environment.NewLine));
                    }
                    else
                    {
                        logTextBox.Clear();
                    }
                });
            }
        }

        private string ErrorMsg(string msg)

```

```

{
    return string.Format("ERROR: {0}", msg);
}

private string SystemMsg(string msg)
{
    return string.Format("SYSTEM: {0}", msg);
}

private void Connected(bool status)
{
    if (!exit)
    {
        connectButton.Invoke((MethodInvoker)delegate
        {
            connected = status;
            if (status)
            {
                addrTextBox.Enabled = false;
                portTextBox.Enabled = false;
                usernameTextBox.Enabled = false;
                keyTextBox.Enabled = false;
                connectButton.Text = "Disconnect";
                Log(SystemMsg("You are now connected"));
            }
            else
            {
                addrTextBox.Enabled = true;
                portTextBox.Enabled = true;
                usernameTextBox.Enabled = true;
                keyTextBox.Enabled = true;
                connectButton.Text = "Connect";
                Log(SystemMsg("You are now disconnected"));
            }
        });
    }
}

private void Read(IAsyncResult result)
{
    int bytes = 0;
    if (obj.client.Connected)
    {
        try
        {
            bytes = obj.stream.EndRead(result);
        }
        catch (Exception ex)
        {
            Log(ErrorMsg(ex.Message));
        }
    }
}

```

```

if (bytes > 0)
{
    obj.data.AppendFormat("{0}", Encoding.UTF8.GetString(obj.buffer, 0, bytes));
    try
    {
        if (obj.stream.DataAvailable)
        {
            obj.stream.BeginRead(obj.buffer, 0, obj.buffer.Length, new AsyncCallback(Read), null);
        }
        else
        {
            Log(obj.data.ToString());
            obj.data.Clear();
            obj.handle.Set();
        }
    }
    catch (Exception ex)
    {
        obj.data.Clear();
        Log(ErrorMsg(ex.Message));
        obj.handle.Set();
    }
}
else
{
    obj.client.Close();
    obj.handle.Set();
}
}

private void ReadAuth(IAsyncResult result)
{
    int bytes = 0;
    if (obj.client.Connected)
    {
        try
        {
            bytes = obj.stream.EndRead(result);
        }
        catch (Exception ex)
        {
            Log(ErrorMsg(ex.Message));
        }
    }
    if (bytes > 0)
    {
        obj.data.AppendFormat("{0}", Encoding.UTF8.GetString(obj.buffer, 0, bytes));
        try
        {
            if (obj.stream.DataAvailable)
            {

```

```

        obj.stream.BeginRead(obj.buffer, 0, obj.buffer.Length, new AsyncCallback(ReadAuth),
null);
    }
    else
    {
        JavaScriptSerializer json = new JavaScriptSerializer();
        Dictionary<string, string> data = json.Deserialize<Dictionary<string,
string>>(obj.data.ToString());
        if (data.ContainsKey("status") && data["status"].Equals("authorized"))
        {
            Connected(true);
        }
        obj.data.Clear();
        obj.handle.Set();
    }
}
catch (Exception ex)
{
    obj.data.Clear();
    Log(ErrorMsg(ex.Message));
    obj.handle.Set();
}
}
else
{
    obj.client.Close();
    obj.handle.Set();
}
}

```

```

private bool Authorize()
{
    bool success = false;
    Dictionary<string, string> data = new Dictionary<string, string>();
    data.Add("username", obj.username);
    data.Add("key", obj.key);
    JavaScriptSerializer json = new JavaScriptSerializer();
    Send(json.Serialize(data));
    while (obj.client.Connected)
    {
        try
        {
            obj.stream.BeginRead(obj.buffer, 0, obj.buffer.Length, new AsyncCallback(ReadAuth), null);
            obj.handle.WaitOne();
            if (connected)
            {
                success = true;
                break;
            }
        }
        catch (Exception ex)
        {

```

```

        Log(ErrorMsg(ex.Message));
    }
}
if (!connected)
{
    Log(SystemMsg("Unauthorized"));
}
return success;
}

private void Connection(IPAddress ip, int port, string username, string key)
{
    try
    {
        obj = new MyClient();
        obj.username = username;
        obj.key = key;
        obj.client = new HttpClient();
        obj.client.Connect(ip, port);
        obj.stream = obj.client.GetStream();
        obj.buffer = new byte[obj.client.ReceiveBufferSize];
        obj.data = new StringBuilder();
        obj.handle = new EventWaitHandle(false, EventResetMode.AutoReset);
        if (Authorize())
        {
            while (obj.client.Connected)
            {
                try
                {
                    obj.stream.BeginRead(obj.buffer, 0, obj.buffer.Length, new AsyncCallback(Read), null);
                    obj.handle.WaitOne();
                }
                catch (Exception ex)
                {
                    Log(ErrorMsg(ex.Message));
                }
            }
            obj.client.Close();
            Connected(false);
        }
    }
    catch (Exception ex)
    {
        Log(ErrorMsg(ex.Message));
    }
}

private void ConnectButton_Click(object sender, EventArgs e)
{
    if (connected)
    {
        obj.client.Close();
    }
}

```

```

}
else if (client == null || !client.IsAlive)
{
    string address = addrTextBox.Text.Trim();
    string number = portTextBox.Text.Trim();
    string username = usernameTextBox.Text.Trim();
    bool error = false;
    IPAddress ip = null;
    if (address.Length < 1)
    {
        error = true;
        Log(SystemMsg("Address is required"));
    }
    else
    {
        try
        {
            ip = Dns.Resolve(address).AddressList[0];
        }
        catch
        {
            error = true;
            Log(SystemMsg("Address is not valid"));
        }
    }
    int port = -1;
    if (number.Length < 1)
    {
        error = true;
        Log(SystemMsg("Port number is required"));
    }
    else if (!int.TryParse(number, out port))
    {
        error = true;
        Log(SystemMsg("Port number is not valid"));
    }
    else if (port < 0 || port > 65535)
    {
        error = true;
        Log(SystemMsg("Port number is out of range"));
    }
    if (username.Length < 1)
    {
        error = true;
        Log(SystemMsg("Username is required"));
    }
    if (!error)
    {
        // encryption key is optional
        client = new Thread(() => Connection(ip, port, username, keyTextBox.Text))
        {
            IsBackground = true

```

```

        };
        client.Start();
    }
}

private void Write(IAsyncResult result)
{
    if (obj.client.Connected)
    {
        try
        {
            obj.stream.EndWrite(result);
        }
        catch (Exception ex)
        {
            Log(ErrorMsg(ex.Message));
        }
    }
}

private void BeginWrite(string msg)
{
    byte[] buffer = Encoding.UTF8.GetBytes(msg);
    if (obj.client.Connected)
    {
        try
        {
            obj.stream.BeginWrite(buffer, 0, buffer.Length, new AsyncCallback(Write), null);
        }
        catch (Exception ex)
        {
            Log(ErrorMsg(ex.Message));
        }
    }
}

private void Send(string msg)
{
    if (send == null || send.IsCompleted)
    {
        send = Task.Factory.StartNew(() => BeginWrite(msg));
    }
    else
    {
        send.ContinueWith(antecedent => BeginWrite(msg));
    }
}

private void SendTextBox_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter)

```



```

    {
        e.Handled = true;
        e.SuppressKeyPress = true;
        if (sendTextBox.Text.Length > 0)
        {
            string msg = sendTextBox.Text;
            sendTextBox.Clear();
            Log(string.Format("{0} (You): {1}", obj.username, msg));
            if (connected)
            {
                Send(msg);
            }
        }
    }
}

private void Client_FormClosing(object sender, FormClosingEventArgs e)
{
    exit = true;
    if (connected)
    {
        obj.client.Close();
    }
}

private void ClearButton_Click(object sender, EventArgs e)
{
    Log();
}

private void CheckBox_CheckedChanged(object sender, EventArgs e)
{
    if (keyTextBox.PasswordChar == '*')
    {
        keyTextBox.PasswordChar = '\0';
    }
    else
    {
        keyTextBox.PasswordChar = '*';
    }
}

private void logTextBox_TextChanged(object sender, EventArgs e)
{
}
}
}

```

JwtServiceTests.cs

```

using ChatProject.WebApplication.Services;
using ChatProject.WebApplication.Settings;

```

```

using FluentAssertions;
using Microsoft.IdentityModel.Tokens;
using NUnit.Framework;
using System;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

namespace ChatProject.Tests
{
    [TestFixture]
    public class JwtServiceTests
    {
        private static TokenValidationParameters _parameters;

        [OneTimeSetUp]
        public void OneTimeSetUp()
        {
            _parameters = new TokenValidationParameters
            {
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(JwtAuthConstants.SigningSecurityKey)),

                ValidateIssuer = true,
                ValidIssuer = JwtAuthConstants.Issuer,

                ValidateAudience = true,
                ValidAudience = JwtAuthConstants.Audience,

                ValidateLifetime = true,

                ClockSkew = TimeSpan.FromSeconds(5),
            };
        }

        [Test]
        public void GenerateAccessToken_WhenAllDataAreValid_ShouldReturnToken()
        {
            // Arrange
            var claims = new Claim[]
            {
                new Claim(ClaimTypes.Role, "Admin"),
                new Claim(ClaimTypes.Name, "Admin123")
            };

            var rightClaimsPrincipal = new ClaimsPrincipal();
            rightClaimsPrincipal.AddIdentity(new ClaimsIdentity(claims, "AuthenticationTypes.Federation"));

            // Act
            var token = JwtAuthenticationService.GenerateAccessToken(claims);

            // Assert
            token.Should().NotBeNull();
        }
    }
}

```

```

ClaimsPrincipal claimsPrincipal = new JwtSecurityTokenHandler().ValidateToken(token, _pa-
rameters, out var _);

claimsPrincipal.Identity.Should().BeEquivalentTo(rightClaimsPrincipal.Identity);
}

[Test]
public void ValidateToken_WhenAllDataAreValid_ShouldReturnTrue()
{
    // Arrange
    var claims = new Claim[]
    {
        new Claim(ClaimTypes.Role, "Admin"),
        new Claim(ClaimTypes.Name, "Admin123")
    };

    var token = JwtAuthenticationService.GenerateAccessToken(claims);

    // Act

    var result = JwtAuthenticationService.ValidateToken(token);

    // Assert
    result.Should().BeTrue();
}

[Test]
public void ValidateToken_WhenDataAreNotValid_ShouldReturnFalse()
{
    // Arrange
    var claims = new Claim[]
    {
        new Claim(ClaimTypes.Role, "Admin"),
        new Claim(ClaimTypes.Name, "Admin123")
    };

    var token = new JwtSecurityToken(
        issuer: "NonChat",
        audience: "NonAudience",
        claims: claims,
        expires: DateTime.Now.AddSeconds(6),
        signingCredentials: new SigningCredentials(
            new SymmetricSecurityKey(Encoding.UTF8.GetBytes(JwtAuthConstants.SigningSecuri-
tyKey)),
            SecurityAlgorithms.HmacSha256));

    string jwtToken = new JwtSecurityTokenHandler().WriteToken(token);

    // Act

    var result = JwtAuthenticationService.ValidateToken(jwtToken);

    // Assert
    result.Should().BeFalse();
}
}

```

```
}
```

# ChatControllerTests.cs

```
using ChatProject.WebApplication.Controllers;
using ChatProject.WebApplication.Models;
using ChatProject.WebApplication.Models.Authentication;
using ChatProject.WebApplication.Services;
using FluentAssertions;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Moq;
using NUnit.Framework;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ChatProject.Tests
{
    [TestFixture]
    public class ChatControllerTests
    {
        [Test]
        public void Get_WhenWithUserIsFalse_ShouldReturnListOfUsersWithoutUser()
        {
            // Arrange
            var userRepository = new Mock<IRepository<User>>();

            userRepository.Setup(item => item.GetAll()).Returns(new List<User>
            {
                new User
                {
                    Id = 1,
                    NickName = "Admin",
                    Password = "Admin",
                    Role = "Admin",
                },
                new User
                {
                    Id = 2,
                    NickName = "User",
                    Password = "User",
                    Role = "User",
                },
            }.AsQueryable());

            var rightResult = new List<string>
            {
                "User"
            };

            var controller = new ChatController(null, null, userRepository.Object);
            var context = new Mock<HttpContext>();
            context.SetupGet(elem => elem.User.Identity.Name).Returns("Admin");
        }
    }
}
```

```

context.SetupGet(elem => elem.User.Identity.IsAuthenticated).Returns(true);
var mockContext = new ControllerContext()
{
    HttpContext = context.Object,
};
controller.ControllerContext = mockContext;

// Act
var actionResult = controller.Get(false);

// Assert
var result = actionResult as OkObjectResult;
result.Should().NotNull();
result.Value.Should().BeEquivalentTo(rightResult);
}

[Test]
public void Get_WhenWithUserIsTrue_ShouldReturnListOfUsersWithUser()
{
    // Arrange
    var userRepository = new Mock<IRepository<User>>();

    userRepository.Setup(item => item.GetAll()).Returns(new List<User>
    {
        new User
        {
            Id = 1,
            NickName = "Admin",
            Password = "Admin",
            Role = "Admin",
        },
        new User
        {
            Id = 2,
            NickName = "User",
            Password = "User",
            Role = "User",
        },
    }.AsQueryable());

    var rightResult = new List<string>
    {
        "User",
        "Admin"
    };

    var controller = new ChatController(null, null, userRepository.Object);
    var context = new Mock<HttpContext>();
    context.SetupGet(elem => elem.User.Identity.Name).Returns("Admin");
    context.SetupGet(elem => elem.User.Identity.IsAuthenticated).Returns(true);
    var mockContext = new ControllerContext()
    {
        HttpContext = context.Object,
    };
    controller.ControllerContext = mockContext;

```

```

// Act
var actionResult = controller.Get(true);

// Assert
var result = actionResult as OkObjectResult;
result.Should().NotNull();
result.Value.Should().BeEquivalentTo(rightResult);
}

[Test]
public async Task Get_WhenNicknameIsNull_ShouldReturnListOfCurrentUserMessages()
{
    // Arrange
    var userRepository = new Mock<IRepository<User>>();
    var messageRepository = new Mock<IRepository<Message>>();

    userRepository.Setup(item => item.GetAll()).Returns(new List<User>
    {
        new User
        {
            Id = 1,
            NickName = "Admin",
            Password = "Admin",
            Role = "Admin",
        },
        new User
        {
            Id = 2,
            NickName = "User",
            Password = "User",
            Role = "User",
        },
    }).AsQueryable();
    userRepository.Setup(item => item.GetById(1)).Returns(Task.FromResult(new User{
        Id = 1,
        NickName = "Admin",
        Password = "Admin",
        Role = "Admin",
    }));
    userRepository.Setup(item => item.GetById(2)).Returns(Task.FromResult(new User
    {
        Id = 2,
        NickName = "User",
        Password = "User",
        Role = "User",
    }));

    var firstDate = DateTime.Now.AddDays(-1);
    var secondDate = DateTime.Now.AddHours(-12);

    messageRepository.Setup(item => item.GetAll()).Returns(new List<Message>
    {
        new Message
        {
            Id = 1,
            DateOfSent = firstDate,

```

```

        Image = null,
        ReceiverId = 1,
        SenderId = 2,
        Value = "Text1",
    },
    new Message
    {
        Id = 2,
        DateOfSent = secondDate,
        Image = null,
        ReceiverId = 2,
        SenderId = 1,
        Value = "Text2",
    },
    new Message
    {
        Image = null,
        Id = 3,
        DateOfSent = DateTime.Now.AddDays(-2),
        ReceiverId = 3,
        SenderId = 4,
        Value = "Text3"
    },
    }.AsQueryable());

var rightResult = new List<SendMessageModel>
{
    new SendMessageModel
    {
        DateOfSent = firstDate,
        Image = null,
        Sender = "User",
        Message = "Text1",
    },
    new SendMessageModel
    {
        DateOfSent = secondDate,
        Image = null,
        Sender = "Admin",
        Message = "Text2",
    },
};

var controller = new ChatController(null, messageRepository.Object, userRepository.Object);
var context = new Mock<HttpContext>();
context.SetupGet(elem => elem.User.Identity.Name).Returns("Admin");
context.SetupGet(elem => elem.User.Identity.IsAuthenticated).Returns(true);
var mockContext = new ControllerContext()
{
    HttpContext = context.Object,
};
controller.ControllerContext = mockContext;

// Act
var actionResult = await controller.Get(null);

```

```

// Assert
var result = actionResult as OkObjectResult;
result.Should().NotNull();
result.Value.Should().BeEquivalentTo(rightResult);
}

[Test]
public async Task Get_WhenUserIsNotAdmin_ShouldReturnListOfCurrentUserMessages()
{
    // Arrange
    var userRepository = new Mock<IRepository<User>>();
    var messageRepository = new Mock<IRepository<Message>>();

    userRepository.Setup(item => item.GetAll()).Returns(new List<User>
    {
        new User
        {
            Id = 1,
            NickName = "Admin",
            Password = "Admin",
            Role = "Admin",
        },
        new User
        {
            Id = 2,
            NickName = "User",
            Password = "User",
            Role = "User",
        },
        new User
        {
            Id = 3,
            NickName = "NewUser",
            Password = "NewUser",
            Role = "User"
        },
        new User
        {
            Id = 4,
            NickName = "NewUser2",
            Password = "NewUser2",
            Role = "User"
        }
    }).AsQueryable();
    userRepository.Setup(item => item.GetById(1)).Returns(Task.FromResult(new User
    {
        Id = 1,
        NickName = "Admin",
        Password = "Admin",
        Role = "Admin",
    }));
    userRepository.Setup(item => item.GetById(2)).Returns(Task.FromResult(new User
    {
        Id = 2,
        NickName = "User",
        Password = "User",
    }));
}

```



```

        Role = "User",
    }));

var firstDate = DateTime.Now.AddDays(-1);
var secondDate = DateTime.Now.AddHours(-12);

messageRepository.Setup(item => item.GetAll()).Returns(new List<Message>
{
    new Message
    {
        Id = 1,
        DateOfSent = firstDate,
        Image = null,
        ReceiverId = 1,
        SenderId = 2,
        Value = "Text1",
    },
    new Message
    {
        Id = 2,
        DateOfSent = secondDate,
        Image = null,
        ReceiverId = 2,
        SenderId = 1,
        Value = "Text2",
    },
    new Message
    {
        Image = null,
        Id = 3,
        DateOfSent = DateTime.Now.AddDays(-2),
        ReceiverId = 3,
        SenderId = 4,
        Value = "Text3"
    },
}).AsQueryable());

var rightResult = new List<SendMessageModel>
{
    new SendMessageModel
    {
        DateOfSent = firstDate,
        Image = null,
        Sender = "User",
        Message = "Text1",
    },
    new SendMessageModel
    {
        DateOfSent = secondDate,
        Image = null,
        Sender = "Admin",
        Message = "Text2",
    },
};

var controller = new ChatController(null, messageRepository.Object, userRepository.Object);

```

```

var context = new Mock<HttpContext>();
context.SetupGet(elem => elem.User.Identity.Name).Returns("User");
context.Setup(elem => elem.User.IsInRole(It.IsAny<string>())).Returns(false);
context.SetupGet(elem => elem.User.Identity.IsAuthenticated).Returns(true);
var mockContext = new ControllerContext()
{
    HttpContext = context.Object,
};
controller.ControllerContext = mockContext;

// Act
var actionResult = await controller.Get("NewUser");

// Assert
var result = actionResult as OkObjectResult;
result.Should().NotNull();
result.Value.Should().BeEquivalentTo(rightResult);
}

[Test]
public async Task Get_WhenUserIsAdmin_ShouldReturnListOfChosenUserMessages()
{
    // Arrange
    var userRepository = new Mock<IRepository<User>>();
    var messageRepository = new Mock<IRepository<Message>>();

    userRepository.Setup(item => item.GetAll()).Returns(new List<User>
    {
        new User
        {
            Id = 1,
            NickName = "Admin",
            Password = "Admin",
            Role = "Admin",
        },
        new User
        {
            Id = 2,
            NickName = "User",
            Password = "User",
            Role = "User",
        },
        new User
        {
            Id = 3,
            NickName = "NewUser",
            Password = "NewUser",
            Role = "User"
        },
        new User
        {
            Id = 4,
            NickName = "NewUser2",
            Password = "NewUser2",
            Role = "User"
        }
    }

```

```

}.AsQueryable());
userRepository.Setup(item => item.GetById(4)).Returns(Task.FromResult(new User
{
    Id = 4,
    NickName = "NewUser2",
    Password = "NewUser2",
    Role = "User"
}));

var firstDate = DateTime.Now.AddDays(-1);
var secondDate = DateTime.Now.AddHours(-12);
var thirdDate = DateTime.Now.AddDays(-2);

messageRepository.Setup(item => item.GetAll()).Returns(new List<Message>
{
    new Message
    {
        Id = 1,
        DateOfSent = firstDate,
        Image = null,
        ReceiverId = 1,
        SenderId = 2,
        Value = "Text1",
    },
    new Message
    {
        Id = 2,
        DateOfSent = secondDate,
        Image = null,
        ReceiverId = 2,
        SenderId = 1,
        Value = "Text2",
    },
    new Message
    {
        Image = null,
        Id = 3,
        DateOfSent = thirdDate,
        ReceiverId = 3,
        SenderId = 4,
        Value = "Text3"
    },
}).AsQueryable());

var rightResult = new List<SendMessageModel>
{
    new SendMessageModel
    {
        Image = null,
        DateOfSent = thirdDate,
        Sender = "NewUser2",
        Message = "Text3"
    },
};

var controller = new ChatController(null, messageRepository.Object, userRepository.Object);

```

```

var context = new Mock<HttpContext>();
context.SetupGet(elem => elem.User.Identity.Name).Returns("Admin");
context.Setup(elem => elem.User.IsInRole(It.IsAny<string>())).Returns(true);
context.SetupGet(elem => elem.User.Identity.IsAuthenticated).Returns(true);
var mockContext = new ControllerContext()
{
    HttpContext = context.Object,
};
controller.ControllerContext = mockContext;

// Act
var actionResult = await controller.Get("NewUser");

// Assert
var result = actionResult as OkObjectResult;
result.Should().NotBeNull();
result.Value.Should().BeEquivalentTo(rightResult);
    }
}
}

```

## **ПРИЛОЖЕНИЕ Б**

(обязательное)

### **Руководство системного программиста**

#### **1 Общие сведения о программе**

##### **1.1 Назначение и функции программы**

Разработанное приложение предназначено для осуществления обмена сообщениями между пользователями.

Приложение позволяет пользователю отправлять и получать сообщения, хранить и просматривать истории пользователя, а также осуществлять аутентификацию пользователя.

##### **1.2 Необходимое техническое обеспечение**

Для нормального функционирования программы необходим компьютер со следующими аппаратными и программными средствами:

- процессор архитектуры *AMD64* или *Intel 64 (x64)*, с тактовой частотой от одного гигагерца;
- оперативная память объемом не менее одного гигабайта в системе;
- свободное место на жестком диске или твердотельном накопителе – не менее одного гигабайта;
- сетевая карта для доступа в сеть Интернет;
- монитор с разрешением 800x600 пикселей или более высоким для комфортной работы;
- клавиатура, компьютерная мышь;
- операционная система *Windows XP* и выше.

#### **2 Структура программы**

Программа состоит из исполняемого файла и списка необходимых библиотек, находящихся в одной с файлом папке. Для запуска приложения необходимо хранить исполняемый файл в папке с библиотеками.

#### **3 Настройка программы**

Так как все необходимые файлы и библиотеки находятся в одной папке, приложение готово к использованию сразу после скачивания.

## **4 Проверка программы**

Для проверки программы требуется провести запуск приложения, после чего ввести адрес в браузере. Если вкладка программы не появилась в браузере после ввода адреса, следовательно, нужно убедиться в наличии всех необходимых библиотек программы.

В случае успешного появления в браузере вкладки приложения, остальные функции должны безупречно работать.

## **5 Дополнительные возможности**

Программа не имеет скрытых от пользователя дополнительных возможностей.

## **6 Сообщения системному программисту**

Во время работы имеется вывод в окно браузера во вкладке программы о всех действиях, которые совершает пользователь. Специализированного мониторинга и записи происходящего программе в программе не имеется.

## **ПРИЛОЖЕНИЕ В**

(обязательное)

### **Руководство программиста**

#### **1 Назначение и условия применения программы**

##### **1.1 Назначение и функции программы**

Разработанное приложение предназначено для осуществления обмена сообщениями между пользователями.

Приложение позволяет пользователю отправлять и получать как сообщения, так и изображения, хранить и просматривать истории пользователя, а также осуществлять вход и регистрацию пользователя.

##### **1.2 Необходимое техническое обеспечение**

Для нормального функционирования программы необходим компьютер со следующими аппаратными и программными средствами:

- процессор архитектуры *AMD64* или *Intel 64 (x64)*, с тактовой частотой от одного гигагерца;
- оперативная память объемом не менее одного гигабайта в системе;
- свободное место на жестком диске или твердотельном накопителе – не менее одного гигабайта;
- сетевая карта для доступа в сеть Интернет;
- монитор с разрешением 800x600 пикселей или более высоким для комфортной работы;
- клавиатура, компьютерная мышь;
- операционная система *Windows XP* и выше.

#### **2 Характеристики программы**

Для настройки программы перед первым запуском необходимо сетевое соединение, которое в дальнейшем необходимо поддерживать.

Программа имеет опциональный графический пользовательский интерфейс, представленный страницами в браузере.

#### **3 Обращение к программе**

Для запуска программы необходимо запустить исполняемый файл *Chat-Project.WebApplication.exe*, после чего откроется консоль с установленным

соединением и указанным портом. Далее необходимо открыть браузер и ввести адрес к странице чата (*http://[адрес сервера]:[порт с консоли].Chat.html*).

#### **4 Входные и выходные данные**

Входными данными являются сообщения и изображения, отправленные пользователем, а также данные, используемые при регистрации.

Выходными данными являются полученные сообщения и изображения, а также история пользователя.

#### **5 Сообщения**

Все сообщения, которые может получить пользователь во время работы с программой, выводятся внутри графического интерфейса для более удобной работы и для того, чтобы не было необходимости закрывать всплывающие окна. При этом интерфейс имеет множество ограничений, которые не дают пользователю возможность совершить ошибку, и, следовательно, получить очередное сообщение об ошибке.



# **ПРИЛОЖЕНИЕ Г**

## **(обязательное)**

### **Руководство пользователя**

#### **1 Введение**

Раздел «Руководство пользователя» содержит материал в обязательном порядке необходимый для ознакомления.

Разработанное приложение предназначено для осуществления коммуникации между пользователями, а также для обмена файлами.

Уровень подготовки пользователя должен соответствовать уровню владения персональным компьютером с использованием периферийных устройств (клавиатура, компьютерная мышь), пользователь должен понимать, как управлять файловой системой, браузером.

#### **2 Назначение и условия применения**

##### **2.1 Виды деятельности**

Приложение позволяет пользователю отправлять и получать как сообщения, так и файлы различных форматов, добавлять пользователей в чат и удалять их из чата.

##### **2.2 Условия для работы программы**

Для нормального функционирования программы необходим компьютер со следующими аппаратными и программными средствами:

- процессор архитектуры *AMD64* или *Intel 64 (x64)*, с тактовой частотой от одного гигагерца;
- оперативная память объемом не менее одного гигабайта в системе;
- свободное место на жестком диске или твердотельном накопителе – не менее одного гигабайта;
- сетевая карта для доступа в сеть Интернет;
- монитор с разрешением 800x600 пикселей или более высоким для комфортной работы;
- клавиатура, компьютерная мышь;
- операционная система *Windows XP* и выше.

Также нужны данные, которыми выступают КТ-изображения позвоночника человека.

Для использования программы не требуется дополнительная подготовка специалистов, достаточно базовых знаний по работе с ПК.

### 3 Подготовка к работе

Состав дистрибутивного носителя данных содержит практически все, что нужно для работы программы, а недостающие библиотеки самым простым образом скачиваются из сети Интернет. После чего, программа абсолютно готова к использованию.

Для проверки работоспособности нужно в корневой папке программы запустить разработанный *server*. В случае если приложение не запускается, следует обратиться к администратору.

### 4 Описание операций

После запуска программы пользователь увидит графическое пользовательское окно, которое показано на рисунке Г.1.

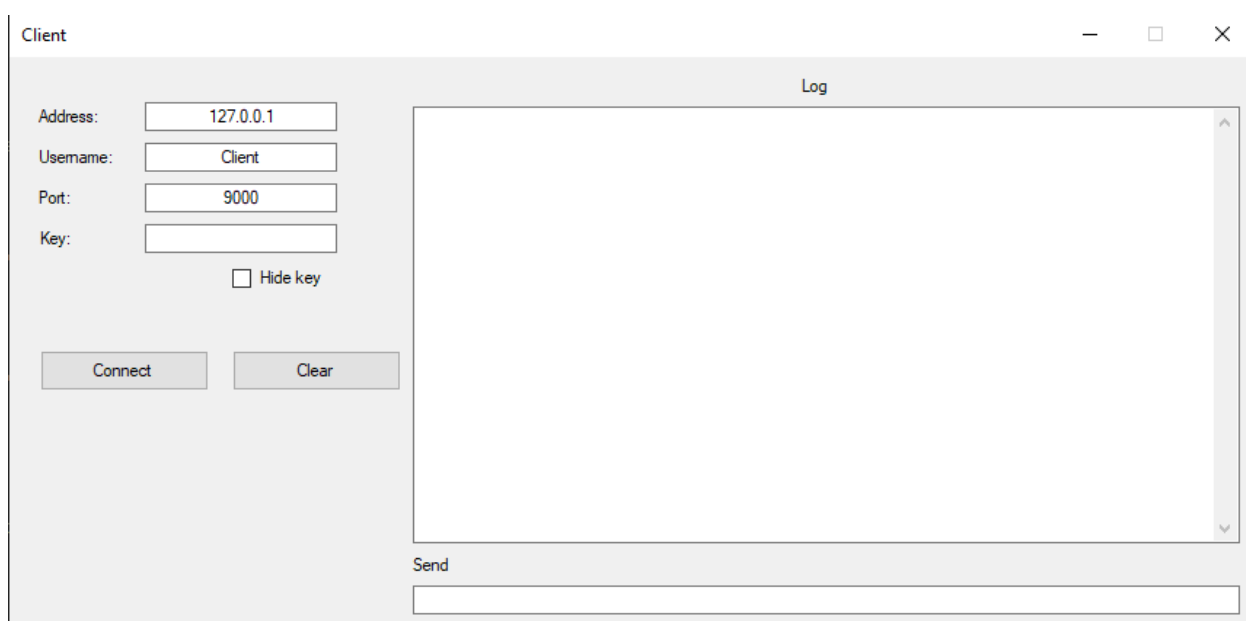


Рисунок Г.1 – Первоначальное окно программы

На первом экране содержится кнопка «*Connect*».

Для начала работы необходимо ввести *IP*-адрес, порт, имя пользователя и нажать кнопку «*Connect*», по нажатию кнопки пользователь будет подключен к чату и получит соответствующее сообщение в поле вывода сообщений

Результат представлен на рисунке Г.2.

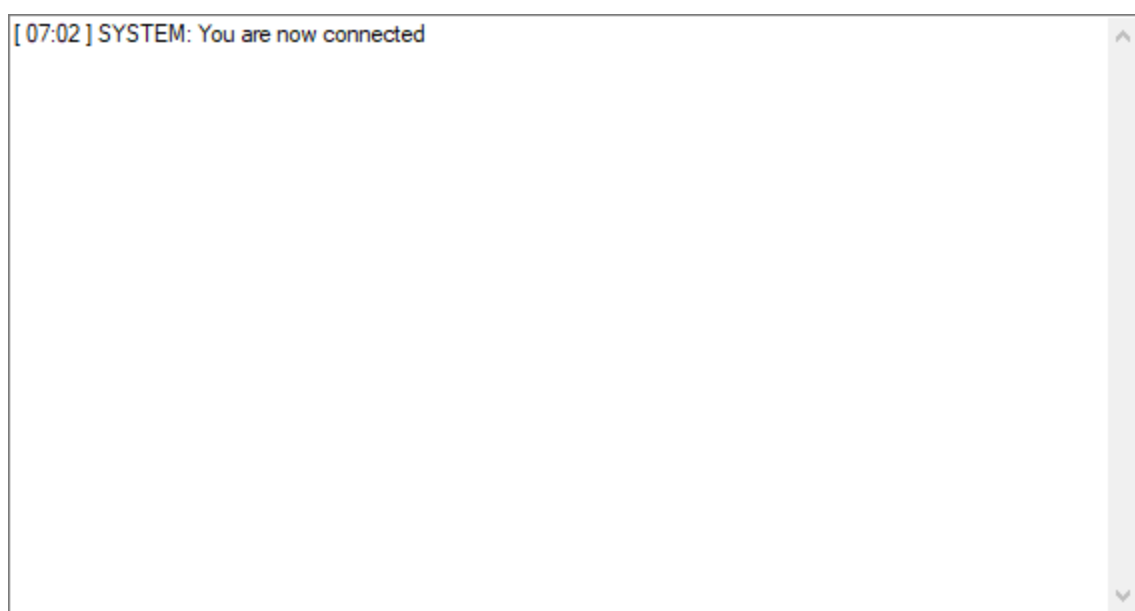


Рисунок Г.2 – Подключение к чату

Для выхода из аккаунта необходимо нажать кнопку «*Disconnect*», расположенной в левой стороне окна приложения. Внешний вид кнопки для выхода из аккаунта представлен на рисунке Г.3.

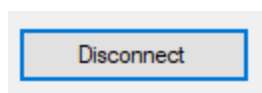


Рисунок Г.3 – Кнопка выхода из аккаунта пользователя

Пользователь, который держит сервер сможет в верхней части окна программы увидеть список участников чата. Так же данный пользователь сможет увидеть в правом верхнем углу кнопку «*Kick*», по нажатию на которую можно удалить участника чата. Внешний вид вышеописанного элемента представлен на рисунке Г.4.

ID	Name	Disconnect
0	Client	Kick

Рисунок Г.4 – Информация о чате

В центральной части экрана расположен элемент, отображающий все сообщения чата, как входящие, так и исходящие. Внизу находится поле для ввода сообщений. Для того, чтобы отправить сообщение, необходимо ввести его в соответствующее поле и нажать на кнопку «*Enter*». После проделанных действий сообщение появится в окне отображения сообщений. Внешний вид центрального элемента приложения представлен на рисунке Г.5.

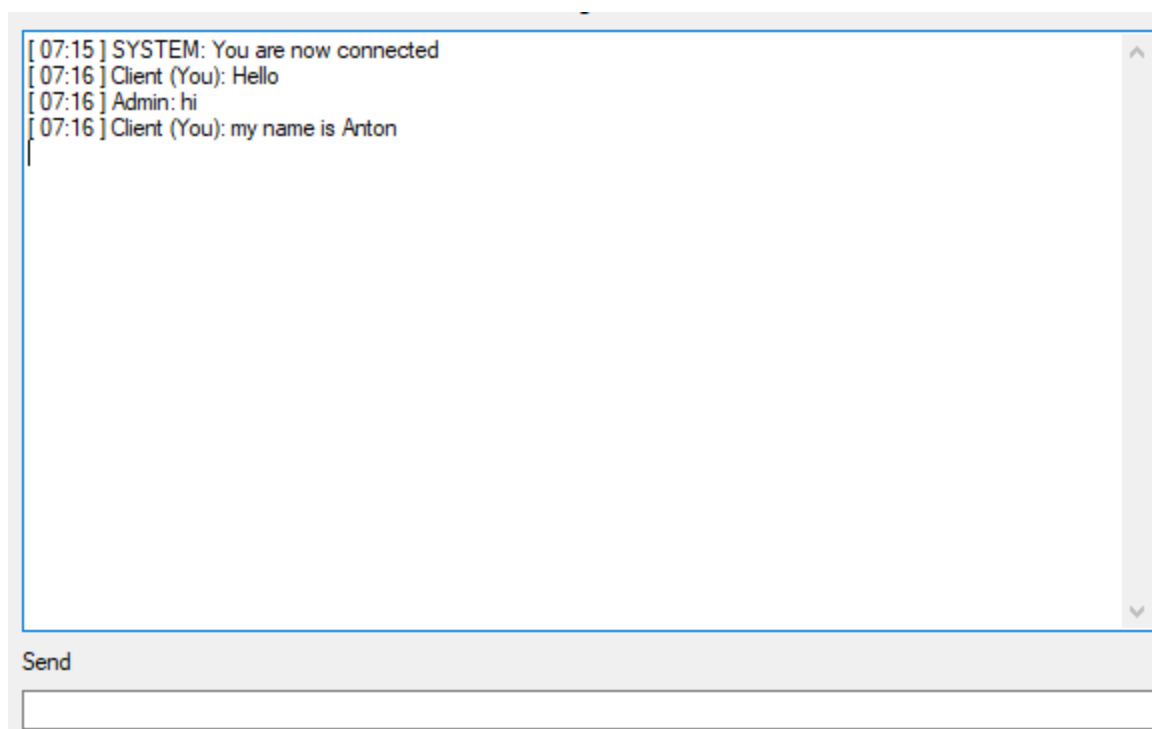


Рисунок Г.5 – Элемент отображения и отправки сообщений

Слева от каждого сообщения написано имя отправителя.

## 5 Аварийные ситуации

Чтобы избежать ошибок при использовании данного приложения, необходимо соблюдать порядок действий и условия использования. При возникновении ошибок следует убедиться, что все рекомендуемые требования выполнены. Если ошибки не решаются, то нужно обратиться к администратору.

## 6 Рекомендации по освоению

Для успешного освоения программы следует соблюдать описанные выше требования и желательно ознакомиться с настоящим руководством пользователя.

**ПРИЛОЖЕНИЕ Д**  
(обязательное)

**Схема архитектуры приложения**