|  |  |  |
| --- | --- | --- |
|  | ЗНАК_МИРЭА_ч_б.tif |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |
| Институт информационных технологий | | |
| Кафедра вычислительной техники | | |

|  |
| --- |
|  |

**КУРСОВАЯ РАБОТА**

по дисциплине Разработка клиент-серверных приложений\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(наименование дисциплины)

**Тема курсовой работы** Разработка чата с использованием библиотеки netty и последующая загрузка сервера в Docker. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Студент группы** ИКБО-05-18 Зонин Георгий Дмитриевич \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(учебная группа, фамилия, имя отчество, студента) (подпись студента)

**Руководитель курсовой работы**\_ст. преподаватель Воронков С.О. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(должность, звание, ученая степень) (подпись руководителя)

**Рецензент** (при наличии)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(должность, звание, ученая степень) (подпись рецензента)

Работа представлена к защите «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_ \_\_г.

Допущен к защите «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Москва 2020 г.

|  |  |  |
| --- | --- | --- |
|  | Рисунок 2 |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |
| Институт информационных технологий | | |
| Кафедра вычислительной техники | | |

|  |  |
| --- | --- |
| **Утверждаю** | |
| Заведующий кафедрой |  |
|  | *Подпись* |
| Платонова О.В. | |
| *ФИО* | |
| «\_\_\_» \_\_\_\_\_\_\_ 2020 г. | |

**ЗАДАНИЕ**

**на выполнение курсовой работы** по дисциплине

«Разработка клиент-серверных приложений»

Студент: Зонин Георгий Дмитриевич Группа: ИКБО-05-18

|  |  |
| --- | --- |
| **Тема:** | Разработка чата с использованием библиотеки netty и последующая загрузка сервера в Docker |

**Исходные данные:**

1. Индивидуальное задание на курсовую работу
2. Язык программирования Java
3. Среда разработки Intellij IDEA
4. Платформа Docker

**Перечень вопросов, подлежащих разработке, и обязательного графического материала:**

1. Подробное описание задания с примером интерфейса программы клиента
2. Исходный код программы на языке Java
3. Подробное описание работы с платформой Docker

**Срок представления к защите курсовой работы:** до «29» декабря 2020 г.

**Задание на курсовую работу выдал** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (Воронков С.О.)

*Подпись руководителя Ф.И.О. руководителя*

**Задание на курсовую работу получил** «14» октября 2020 г

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (Зонин Г.Д.)

*Подпись обучающегося* *Ф.И.О. исполнителя*

Москва 2020 г.

Оглавление

Оглавление 4

Постановка задачи 5

Написание чата 5

Загрузка сервера в Docker 5

Ход работы 5

Создание проекта 5

Создание серверной части приложения 6

Создание клиентской части приложения 7

Загрузка приложения в Docker 10

Список используемой литературы 12

Приложение 1. Исходный код чата. 13

Общая часть 13

Серверная часть 13

Клиентская часть 15

Постановка задачи

Курсовая работа состоит из двух заданий – написание клиентской и серверной части чата, и загрузка сервера в Docker, после чего необходимо совершить подключение клиентов к серверу и убедиться в исправной работе чата. Для выполнения первого задания должен использоваться язык Java с библиотекой Netty. Для выполнения второго задания необходимо установить Docker Hub и зарегистрироваться на сайте hub.docker.com для получения id.

Написание чата

Для клиентской части чата необходимо разработать интерфейс. Описать работу с сетью, создать конвейеры обработки данных и наладить отправку и получение строковых данных по указанному ip-адресу и порту.

Для серверной части необходимо настроить получение и обработку строковых данных, полученных от клиентов, а также отправку ответных данных всем клиентам.

Загрузка сервера в Docker

Необходимо создать исполняемый jar-файл программы сервера и собрать образ, который будет создавать конвейеры с сервером. Далее необходимо загрузить сервер в Docker Hub.

Ход работы

Создание проекта

Для более удобной работы со сторонними библиотеками вместо классического приложения Java создадим проект с добавлением модуля Maven. После создания в папке проекта будет файл pom.xml. В нем описываются все добавляемые библиотеки, используемая версия Java, версии различных компонентов, модули приложения и многое другое. Для добавления библиотеки netty требуется перейти на страницу mvnrepository.com, с помощью поиска найти страницу библиотеки “Netty / All in One” и выбрать одну из версий. Далее необходимо скопировать код со страницы и вставить внутри тега <dependencies> в файле pom.xml. После этого Maven скачает и установит все необходимые данные.

Для того, чтобы не создавать разные проекты для клиентской и серверной части создадим отдельные модули Chat-client и Chat-server. Вся логика, описанная внутри одного модуля, не распространяется за его пределы. Таким образом, для клиентской и серверной части программы, у нас есть два логически изолированных блока.

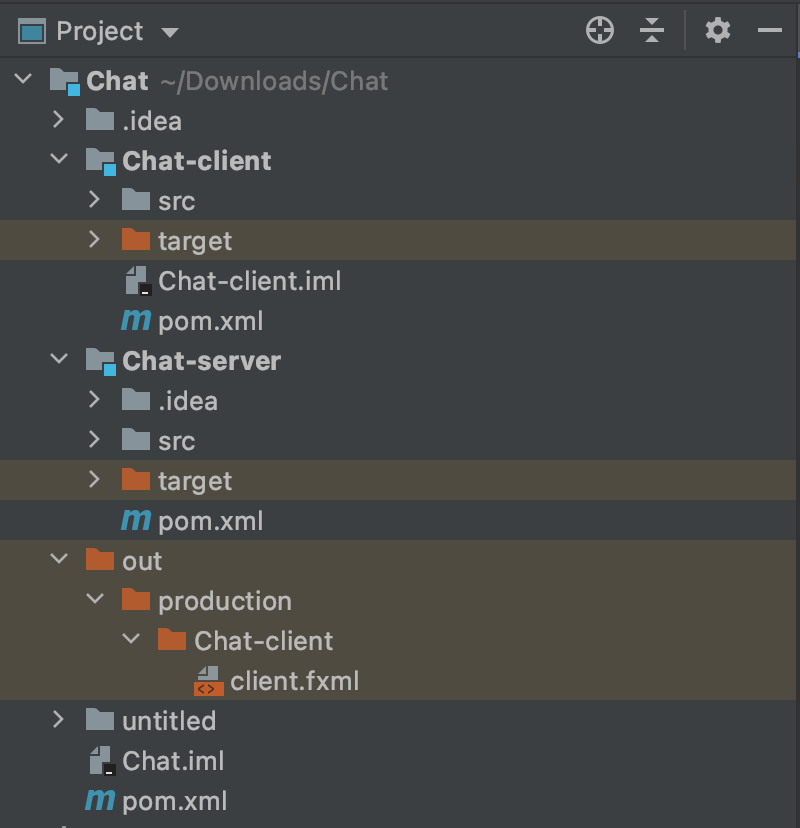


Рисунок 1. Итоговая иерархия программы.

Создание серверной части приложения

Для начала создадим класс ServerApp. В этом классе будет находиться метод main, в котором сервер и будет запускаться. В нем создадим два пула потоков. Один –для подключения пользователей, а второй – для обработки получаемой от них информации. Чтобы настроить сервер создадим объект класса ServerBootstrap. В нем укажем используемые пулы потоков, используемый канал (NioServerSocketChannel.class) и способ общения с сервером – объект класса ChannelInitializer с перегруженным методом initChannel. Здесь мы будем задавать обработчики для входных и выходных данных. Далее задаем порт и запускаем наш сервер. Порт необходим для того, чтобы ОС могла определить для какого приложения пришли данные из сети. После этого сервер может запускаться, но пока он не обрабатывает никакой информации.

Чтобы передать данные от клиента к серверу необходимо преобразовать их в множество байт (байт буффер), поэтому, для работы с клиентом, добавим несколько обработчиков: StringEncoder (преобразует строку в поток байт), StringDecoder (наоборот, преобразует поток байт в строку) и наш собственный обработчик MainHandler.

Класс MainHandler, унаследованный от класса SimpleChannelInboundHandler<String> будет являться обработчиком входящих сообщений. Он будет принимать строку на вход и выполнять какие-либо операции с ней. Строку он будет получать от обработчика StringDecoder. Класс MainHandler перегружает несколько методов: channelActive, channelRead0, exceptionCaught. Первый срабатывает, когда клиент подключается. Здесь будем сохранять канал клиента в листе, присваивать клиенту имя и рассылать остальным клиентам сообщение о том, что новый клиент подключился. Второй метод срабатывает, когда приходит сообщение от одного из клиентов. Т.к. мы реализуем чат, то это сообщение будет переслано всем клиентам для отображения. Также в этом методе мы можем обрабатывать какие-либо специальные строки, например, строки, начинающиеся с символа “/” будут расцениваться как команды и не будут отправляться другим пользователям. Последний метод вызывается, когда выбрасывается исключение от одного из клиентов. В этом методе будем отключать клиента от сервера и сообщать об этом остальным клиентам.

Создание клиентской части приложения

Для создания клиентского приложения создадим новый проект JavaFX. И скопируем из него файлы Main, Controller и sample.fxml (переименовав в client.fxml). Это позволит не писать стартовый код вручную, что сэкономит время. В fxml файле описывается интерфейс приложения, класс Controller отвечает за взаимодействие элементов интерфейса с пользователем, а класс Main с одноименным методом является точкой входа в программу.

Перед созданием интерфейса приложения укажем в fxml файле путь к контроллеру. Все элементы интерфейса будут находиться внутри элемента VBox. В этом случае они будут располагаться по вертикали друг за другом. Первый элемент внутри VBox – TextArea. Запретив редактирование этого элемента и настроив так, чтобы он занимал как можно больше места по вертикали, получим окно для вывода текстовых сообщений чата. Следующие элементы будут находиться внутри элемента HBox. Этот элемент похож на элемент VBox, но располагает элементы по горизонтали. В него добавим элементы TextField и Button. Это будут поле ввода текста и кнопка отправки сообщений соответственно.

Также добавим небольшое меню вверху программы с помощью элементов MenuBar, Menu и MenuItem. В меню будет один пункт – выход из программы.

В классе Main пропишем путь к файлу client.fxml, из которого будет загружаться интерфейс, а также укажем отображаемое название приложение и его размер. После этого можно запустить приложение.

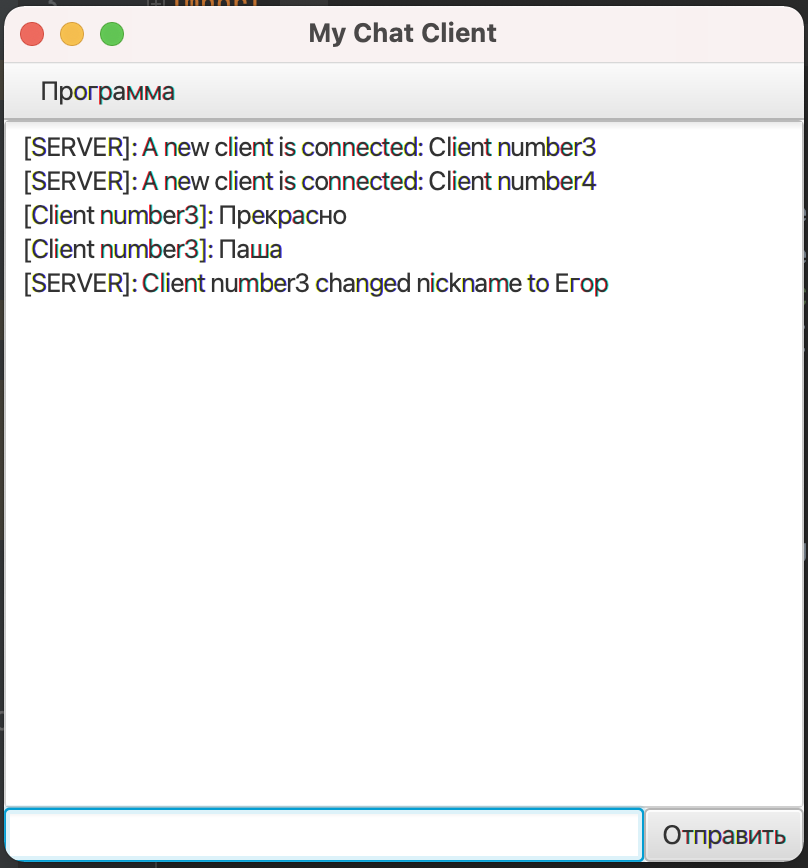


Рисунок 2. Внешний вид клиентского приложения.

Для работы с сетью опишем отдельный класс Network. В конструкторе этого класса будет запускаться параллельный поток, в котором и будет обрабатываться вся логика взаимодействия с сервером. Здесь также, как и на сервере, создадим пул потоков, но только один – на обработку данных. Создадим объект класса Bootstrap, в котором проведем настройку клиента: укажем используемый пул потоков, канал и настроим конвейер обработки данных. После этого подключимся к серверу по ip-адресу “localhost” (т.к. клиентское приложение запускается на том же устройстве, на котором и сервер) и порту, который мы указывали при создании сервера.

Добавим в класс Controller реализацию интерфейса Initializable. Теперь, когда приложение запустится вызовется метод Initialize. В этом методе создадим объект класса Network, после чего, в конструкторе этого объекта, клиент подключится к серверу. В этом же классе опишем логику, что при нажатии на кнопку (или клавиши Enter) сообщение из поля TextField отправится по клиентскому каналу на сервер.

Как и у сервера у клиента будет два обработчика StringDecoder и StringEncoder для преобразования строк в байт буфферы или наоборот, а также собственный обработчик ClientHandler. Чтобы передать для отображения строку, полученную обработчиком ClientHandler, в класс Controller, необходимо создать интерфейс Callback, описывающий один метод, принимающий множество объектов типа Object. Добавим объект, реализующий этот интерфейс, в аргументы конструкторов классов Network и ClientHandler. Теперь при создании объекта Network в методе initialize класса Controller мы сможем указать метод, который будет вызываться в обработчике ClientHandler при получении сообщения.

На этом создание клиентского приложения и сервера закончено.



Рисунок 3. Пример работы приложения.

Загрузка приложения в Docker

С помощью IntelliJ IDEA можно с легкостью собрать серверную часть приложения в исполняемый .jar файл. После этого его можно запустить с помощью консольной команды “java –jar Chat-server.jar” или “java –cp Chat-server.jar com.Krizin.Chat.server.ServerApp ”.

Для запуска сервера в контейнере докера, необходимо сначала создать образ с нашим приложением. Для этого необходимо создать файл Dockerfile. В нем описываются команды для создания образа. Dockerfile и исполняемый файл приложения должны находиться в одной директории.

Всего будет прописано 4 команды.

FROM openjdk:11-slim

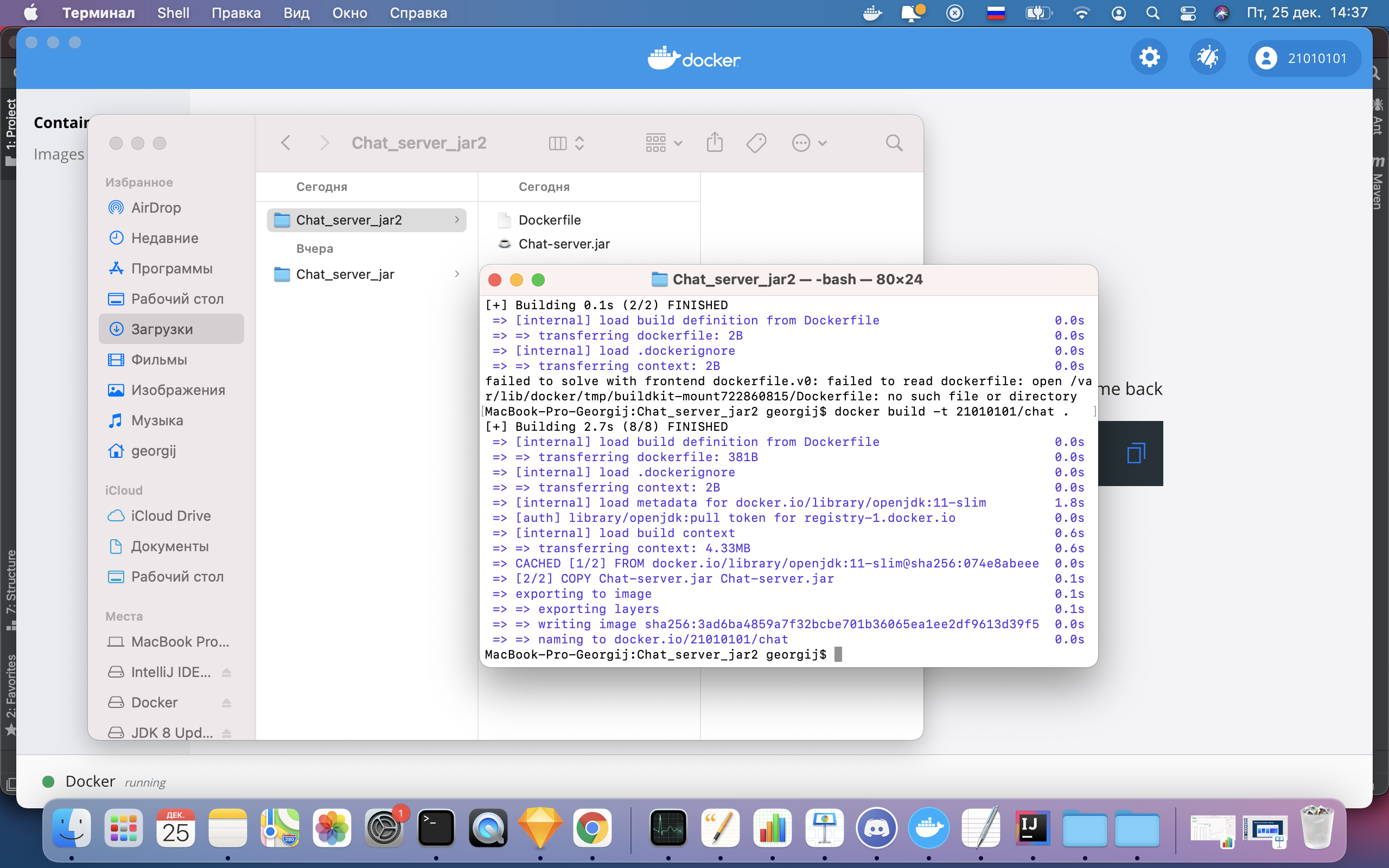
EXPOSE 8189

COPY Chat-server.jar Chat-server.jar

ENTRYPOINT ["java", "-cp", "Chat-server.jar", "com.Krizin.Chat.server.ServerApp"]

Первая команда говорит о том, что наш образ будет основываться на другом базовом образе openjdk:11-slim. Вторая строка указывает порт, который следует открыть. Третья копирует наш исполняемый файл Chat-server.jar в одноименный файл в контейнере. И последняя строка предоставляет команду с аргументами для вызова во время выполнения контейнера.

Вызовем консоль из папки, в которой лежат исполняемый .jar файл и Dockerfile и вызовем команду “docker build -t 21010101/chat .”. После этого соберется наш образ.

Рисунок 4. Сборка образа.

Для запуска контейнера необходимо ввести команду “docker run –p 8189:8189 21010101/chat”. Аргумент –p связывает порт контейнера с портом хоста.

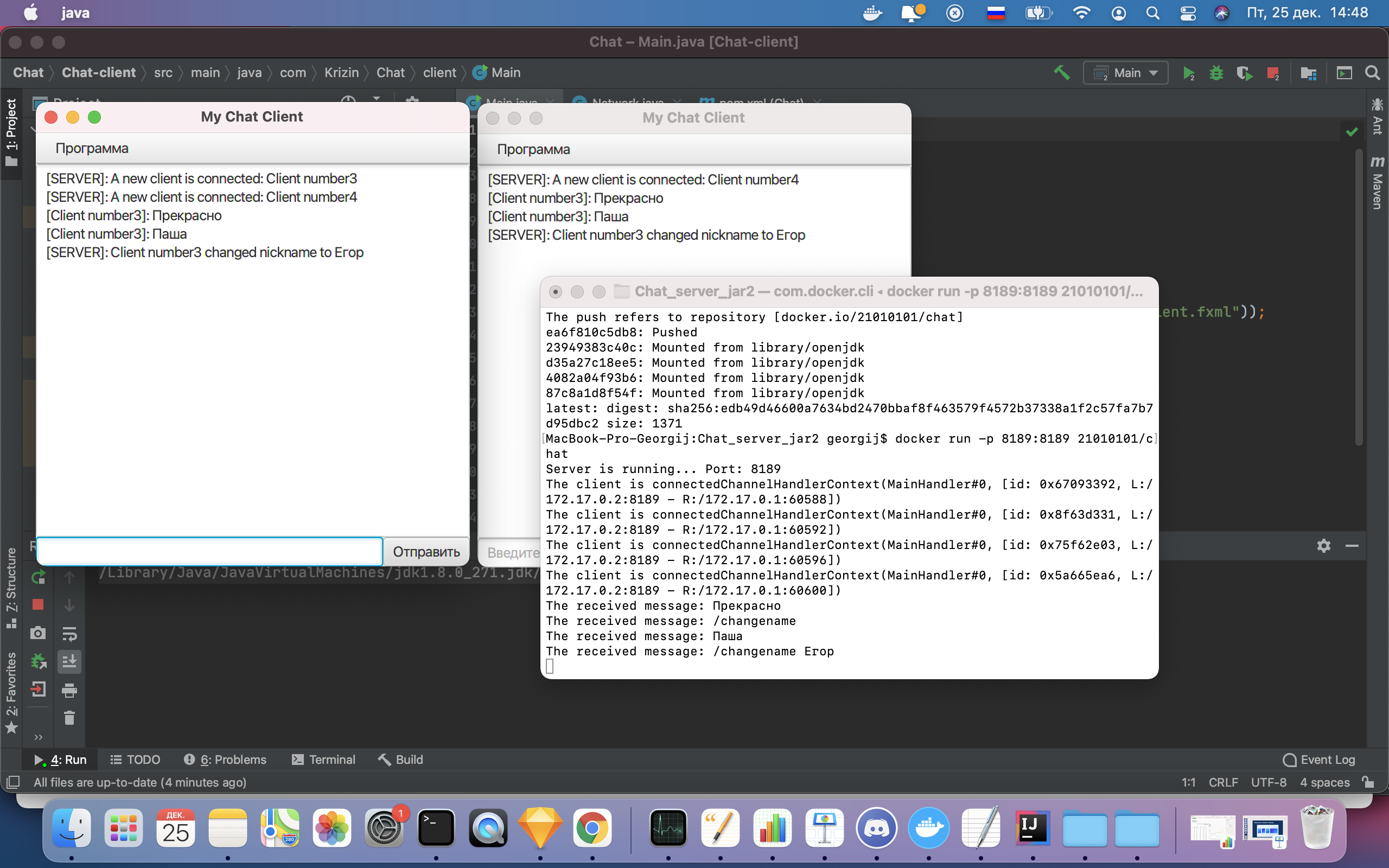
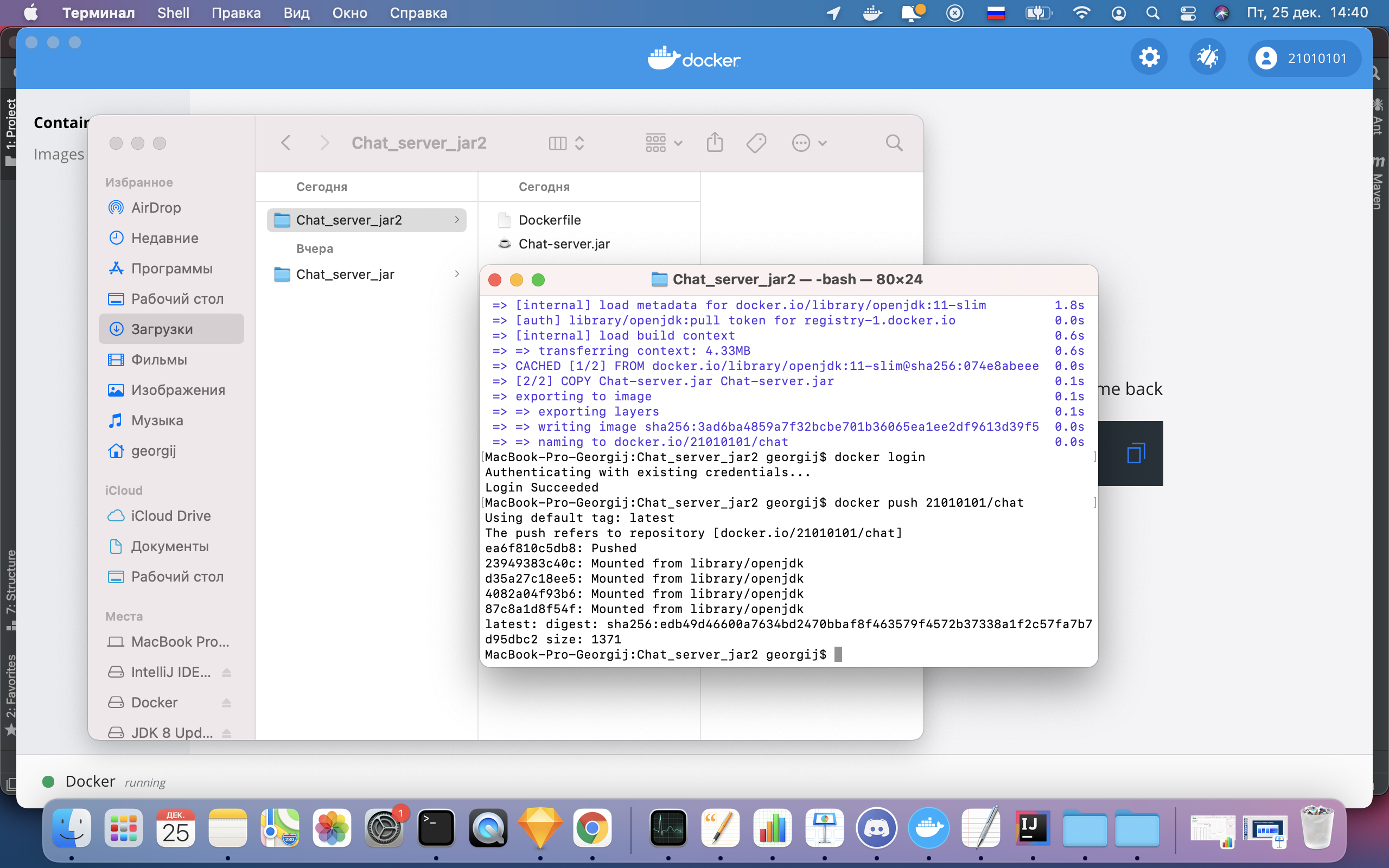
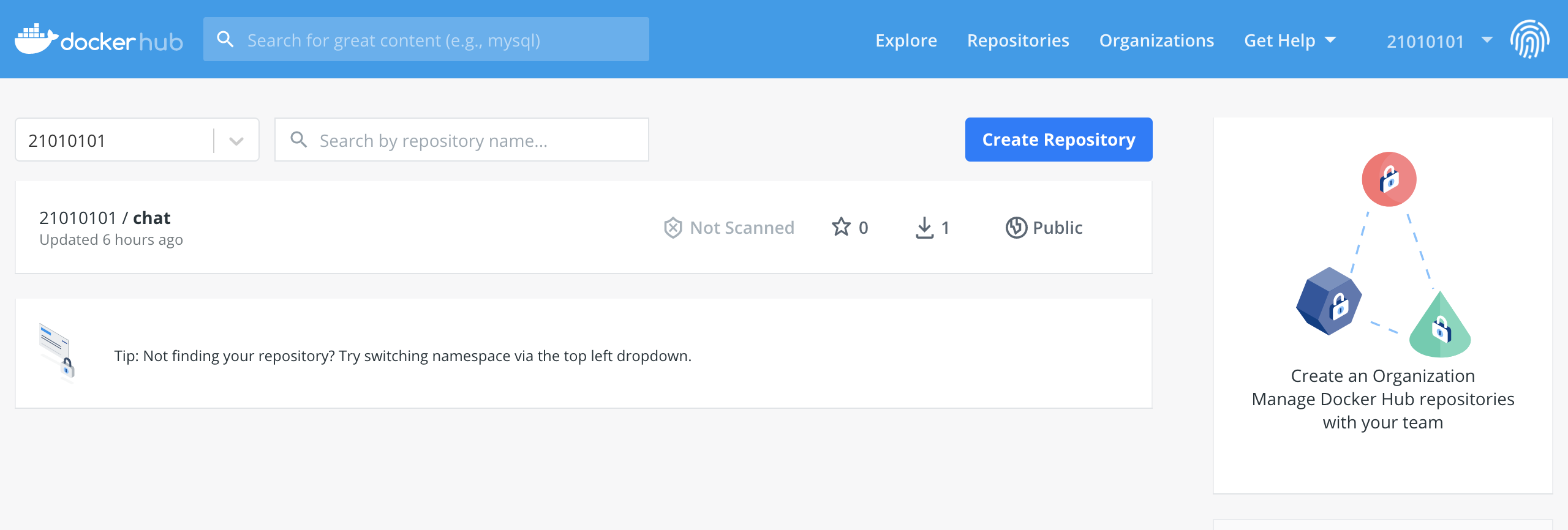


Рисунок 5. Результат работы чата.

Для того, чтобы выложить созданный образ в docker hub, необходимо прописать команды docker login и docker push <\*имя пользователя\*/\*имя образа\*>. Docker login авторизуется на сервере, а docker push 21010101/chat отправит на него наш образ.

Рисунок 6. Отправка образа в Docker Hub.

После этого можно зайти на сайт https://hub.docker.com/ и найти на нем репозиторий с загруженным образом.



Список используемой литературы

1. Шилдт, Герберт. Java. Полное руководство, 10-е изд. : Пер. с англ. –СПб. : ООО “Альфа-книга”, 2018. – 1488 с. : ил. – Парал. тит. англ.
2. https://docs.docker.com/
3. <https://netty.io/wiki/index.html/>

4) <https://www.youtube.com>

Приложение 1. Исходный код чата.

Общая часть

***pom.xml***

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**groupId**>com.Krizin.Chat</**groupId**>  
 <**artifactId**>Chat</**artifactId**>  
 <**packaging**>pom</**packaging**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 <**modules**>  
 <**module**>Chat-server</**module**>  
 <**module**>Chat-client</**module**>  
 </**modules**>  
  
 <**properties**>  
 <**maven.compiler.source**>8</**maven.compiler.source**>  
 <**maven.compiler.target**>8</**maven.compiler.target**>  
 </**properties**>  
  
 <**dependencies**>  
 *<!-- https://mvnrepository.com/artifact/io.netty/netty-all -->* <**dependency**>  
 <**groupId**>io.netty</**groupId**>  
 <**artifactId**>netty-all</**artifactId**>  
 <**version**>4.1.56.Final</**version**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

Серверная часть

***ServerApp.java***

**package** com.Krizin.Chat.server;  
  
**import** io.netty.bootstrap.ServerBootstrap;  
**import** io.netty.channel.ChannelFuture;  
**import** io.netty.channel.ChannelInitializer;  
**import** io.netty.channel.EventLoopGroup;  
**import** io.netty.channel.nio.NioEventLoopGroup;  
**import** io.netty.channel.socket.SocketChannel;  
**import** io.netty.channel.socket.nio.NioServerSocketChannel;  
**import** io.netty.handler.codec.string.StringDecoder;  
**import** io.netty.handler.codec.string.StringEncoder;  
  
**public class** ServerApp {  
 **private static final int *PORT*** = 8189;  
  
 **public static void** main(String[] args) {  
 EventLoopGroup bossGroup = **new** NioEventLoopGroup(1); *// пул потоков для подключения пользователей* EventLoopGroup workerGroup = **new** NioEventLoopGroup(); *// пул потоков для работы с пользователями* **try** {  
 ServerBootstrap b = **new** ServerBootstrap();  
  
 *// Настройка сервера* b.group(bossGroup, workerGroup)  
 .channel(NioServerSocketChannel.**class**)  
 .childHandler(**new** ChannelInitializer<SocketChannel>() { *// SocketChannel - вся инфа о подключении* @Override  
 **protected void** initChannel(SocketChannel socketChannel) **throws** Exception {  
 socketChannel.pipeline().addLast(**new** StringDecoder(), **new** StringEncoder(), **new** MainHandler());  
 }  
 });  
 *// Запуск сервера* ChannelFuture future = b.bind(***PORT***).sync(); *// Установка порта и запуск*

System.***out***.println(**"Server is running... PORT: "** + ***PORT***);future.channel().closeFuture().sync();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 bossGroup.shutdownGracefully(); *// Закрытие пулов потоков* workerGroup.shutdownGracefully();  
 }  
  
 }  
}

***MainHandler.java***

**package** com.Krizin.Chat.server;  
  
**import** io.netty.buffer.ByteBuf;  
**import** io.netty.channel.Channel;  
**import** io.netty.channel.ChannelHandlerContext;  
**import** io.netty.channel.ChannelInboundHandlerAdapter;  
**import** io.netty.channel.SimpleChannelInboundHandler;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** MainHandler **extends** SimpleChannelInboundHandler<String> {  
 **private static final** List<Channel> ***channels*** = **new** ArrayList<>();  
 **private** String **clientName**;  
 **private static int** *newClientIndex* = 1;  
  
 @Override  
 **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  
 System.***out***.println(**"Клиент подключился"** + ctx);  
 ***channels***.add(ctx.channel());  
 **clientName** = **"Client "** + *newClientIndex*;  
 *newClientIndex*++;  
 broadcastMessage(**"SERVER"**, **"New Client Connected: "** + **clientName**);  
 }  
  
 @Override  
 **protected void** channelRead0(ChannelHandlerContext channelHandlerContext, String s) **throws** Exception {  
 System.***out***.println(**"Получено сообщение: "** + s);  
 **if** (s.startsWith(**"/"**)) {  
 **if** (s.startsWith(**"/changename "**)) {  
 String newNickname = s.split(**"\\s"**, 2)[1];  
 broadcastMessage(**"SERVER"**, **clientName** + **" changed nickname to "** + newNickname);  
 **clientName** = newNickname;  
 }  
 **return**;  
 }  
 broadcastMessage(**clientName**, s);  
 }  
  
 **public void** broadcastMessage(String clientName, String message) {  
 String out = String.*format*(**"[%s]: %s\n"**, clientName, message);  
 **for** (Channel c : ***channels***) {  
 c.writeAndFlush(out);  
 }  
 }  
  
 @Override  
 **public void** exceptionCaught(ChannelHandlerContext ctx, Throwable cause) **throws** Exception {  
 System.***out***.println(**"Клиент "** + **clientName** + **" отвалился"**);  
 ***channels***.remove(ctx.channel());  
 broadcastMessage(**"SERVER"**, **"Client "** + **clientName** + **" disconnected"**);  
 ctx.close();  
 }  
}

Клиентская часть

***client.fxml***

*<?***import javafx.geometry.Insets***?>  
<?***import javafx.scene.layout.GridPane***?>  
  
<?***import javafx.scene.control.Button***?>  
<?***import javafx.scene.control.Label***?>  
<?***import javafx.scene.layout.VBox***?>  
<?***import javafx.scene.control.TextArea***?>  
<?***import javafx.scene.layout.HBox***?>  
<?***import javafx.scene.control.TextField***?>  
<?***import javafx.scene.control.MenuBar***?>  
<?***import javafx.scene.control.Menu***?>  
<?***import javafx.scene.control.MenuItem***?>*<**VBox fx:controller="com.Krizin.Chat.client.Controller" xmlns:fx="http://javafx.com/fxml"**>  
 <**MenuBar**>  
 <**Menu text="Программа"**>  
 <**MenuItem text="Выйти из программы" onAction="#exitAction"**></**MenuItem**>  
 </**Menu**>  
 </**MenuBar**>  
 <**TextArea fx:id="mainArea" editable="false" VBox.vgrow="ALWAYS"**/>  
 <**HBox**>  
 <**TextField fx:id="msgField" HBox.hgrow="ALWAYS" promptText="Введите сообщение..." onAction="#sendMsgAction"**/>  
 <**Button text="Отправить" onAction="#sendMsgAction"**/>  
 </**HBox**>  
</**VBox**>

***Main.java***

**package** com.Krizin.Chat.client;  
  
**import** javafx.application.Application;  
**import** javafx.fxml.FXMLLoader;  
**import** javafx.scene.Parent;  
**import** javafx.scene.Scene;  
**import** javafx.stage.Stage;  
  
**public class** Main **extends** Application {  
  
 @Override  
 **public void** start(Stage primaryStage) **throws** Exception{  
 Parent root = FXMLLoader.*load*(getClass().getResource(**"/client.fxml"**));  
 primaryStage.setTitle(**"My Chat Client"**);  
 primaryStage.setScene(**new** Scene(root, 400, 400));  
 primaryStage.show();  
 }  
  
  
 **public static void** main(String[] args) {  
 *launch*(args);  
 }  
}

***Controller.java***

**package** com.Krizin.Chat.client;  
  
**import** javafx.application.Platform;  
**import** javafx.event.ActionEvent;  
**import** javafx.fxml.FXML;  
**import** javafx.fxml.Initializable;  
**import** javafx.scene.control.TextArea;  
**import** javafx.scene.control.TextField;  
  
**import** java.net.URL;  
**import** java.util.ResourceBundle;  
  
**public class** Controller **implements** Initializable {  
 **private** Network **network**;  
  
 @FXML  
 TextField **msgField**;  
  
 @FXML  
 TextArea **mainArea**;  
  
 @Override  
 **public void** initialize(URL location, ResourceBundle resources) {  
 **network** = **new** Network((args) -> {  
 **mainArea**.appendText((String) args[0]);  
 });  
 }  
  
  
 **public void** sendMsgAction(ActionEvent actionEvent) {  
 **network**.sendMessage(**msgField**.getText());  
 **msgField**.clear();  
 **msgField**.requestFocus();  
 }  
  
 **public void** exitAction(ActionEvent actionEvent) {  
 **network**.close();  
 Platform.*exit*();  
 }  
}

***Network.java***

**package** com.Krizin.Chat.client;  
  
**import** io.netty.bootstrap.Bootstrap;  
**import** io.netty.channel.\*;  
**import** io.netty.channel.nio.NioEventLoopGroup;  
**import** io.netty.channel.socket.SocketChannel;  
**import** io.netty.channel.socket.nio.NioSocketChannel;  
**import** io.netty.handler.codec.string.StringDecoder;  
**import** io.netty.handler.codec.string.StringEncoder;  
  
**public class** Network {  
 **private** SocketChannel **channel**;  
  
 **private static final** String ***HOST*** = **"localhost"**;  
 **private static final int *PORT*** = 8189;  
  
 **public** Network(Callback onMessageReceivedCallback) {  
 Thread t = **new** Thread(() -> {  
 EventLoopGroup workerGroup = **new** NioEventLoopGroup();  
 **try** {  
 Bootstrap b = **new** Bootstrap();  
 b.group(workerGroup)  
 .channel(NioSocketChannel.**class**)  
 .handler(**new** ChannelInitializer<SocketChannel>() {  
 @Override  
 **protected void** initChannel(SocketChannel socketChannel) **throws** Exception {  
 **channel** = socketChannel;  
 socketChannel.pipeline().addLast(**new** StringDecoder(), **new** StringEncoder(), **new** ClientHandler(onMessageReceivedCallback));  
 }  
 });  
 ChannelFuture future = b.connect(***HOST***, ***PORT***).sync();  
 future.channel().closeFuture().sync();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }**finally** {  
 workerGroup.shutdownGracefully();  
 }  
  
 });  
 t.setDaemon(**true**);  
 t.start();  
 }  
  
 **public void** close() {  
 **channel**.close();  
 }  
  
 **public void** sendMessage(String str) {  
 **channel**.writeAndFlush(str);  
 }  
}

***ClientHandler.java***

**package** com.Krizin.Chat.client;  
  
**import** io.netty.channel.ChannelHandlerContext;  
**import** io.netty.channel.SimpleChannelInboundHandler;  
  
**public class** ClientHandler **extends** SimpleChannelInboundHandler<String> {  
 **private** Callback **onMessageReceivedCallback**;  
  
 **public** ClientHandler(Callback onMessageReceivedCallback) {  
 **this**.**onMessageReceivedCallback** = onMessageReceivedCallback;  
 }  
  
 @Override  
 **protected void** channelRead0(ChannelHandlerContext channelHandlerContext, String s) **throws** Exception {  
 **if** (**onMessageReceivedCallback** != **null**) {  
 **onMessageReceivedCallback**.callback(s);  
 }  
 }  
}

***Callback.java***

**package** com.Krizin.Chat.client;  
  
**public interface** Callback {  
 **void** callback(Object... args) ;  
}