ГУАП

ФАКУЛЬТЕТ ДОПОЛНИТЕЛЬНОГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ассистент |  |  |  | С. Ю. Гуков |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ПРАКТИЧЕСКОЙ РАБОТЕ |
| ЗАДАНИЕ ПО ТЕМЕ: КЛИЕНТ-СЕРВЕРНОЕ ПРИЛОЖЕНИЕ |
| по курсу: ВВЕДЕНИЕ В РАЗРАБОТКУ КОРПОРАТИВНЫХ ПРИЛОЖЕНИЙ НА JAVA |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4941 |  |  |  | Н. С. Горбунов |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2022

**Задание:**

Разработать клиент-серверное приложение, работающее по протоколу TCP. Клиентская и серверная часть должны быть реализованы на

языке программирования высокого уровня Java.

Конвертер величин «Мера давления»

**Решение:**

Проект разделен на три «модуля»:

1. TCPServer – серверная часть приложения. Принимает входящие TCP подключения и обрабатывает их в отдельном потоке.
2. TCPClient – клиенсткая часть приложения. Содержит в себе прототип пользовательского интерфейса на JavaFX.
3. TCPShared – содержит в себе общие классы, подразумевающие модели передаваемых данных

Клиент и сервер обмениваются данными по протоколу TCP. Формат данных: JSON. В котором строго определены два верхних узла:

* headers – заголовки сообщения для того, чтобы обозначить конечного получателя сообщения
* body – тело сообщения, в котором хранится ещё один JSON узел.

Листинг 1. Класс сообщения

package com.example.shared;  
  
import com.fasterxml.jackson.annotation.JsonIgnore;  
import com.fasterxml.jackson.core.JsonProcessingException;  
import com.fasterxml.jackson.databind.JsonMappingException;  
import com.fasterxml.jackson.databind.JsonNode;  
import com.fasterxml.jackson.databind.ObjectMapper;  
import lombok.Data;  
import lombok.Setter;  
  
@Data  
public class TCPMessage {  
 private String[] headers;  
 private JsonNode body;  
 private static ObjectMapper *objectMapper* = new ObjectMapper();  
  
 @Override  
 public String toString() {  
 try {  
 return *objectMapper*.writerWithDefaultPrettyPrinter().writeValueAsString(this);  
 } catch (JsonProcessingException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 public static void validate(String data) throws JsonProcessingException {  
 *objectMapper*.readTree(data);  
 }  
}

Такой формат сообщения не случаен: когда сообщение приходит на сервер, то в зависимости от содержания заголовков решается, кто будет обрабатывать этот запрос.

Листинг 2. Класс Main. Принимает входящее TCP подключение и создает новую нить.

package com.example.tcp.server;  
  
import java.io.\*;  
import java.net.ServerSocket;  
import java.net.Socket;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
public class Main  
{  
 static ExecutorService *executeIt* = Executors.*newFixedThreadPool*(2);  
  
 */\*\*  
 \** ***@param*** *args  
 \*/* public static void main(String[] args) {  
  
 // стартуем сервер на порту 3345 и инициализируем переменную для обработки консольных команд с самого сервера  
 try (ServerSocket server = new ServerSocket(3345);  
 BufferedReader br = new BufferedReader(new InputStreamReader(System.*in*))) {  
 System.*out*.println("Server socket created, command console reader for listen to server commands");  
  
 // стартуем цикл при условии, что серверный сокет не закрыт  
 while (!server.isClosed()) {  
  
 // проверяем поступившие команды из консоли сервера если такие  
 // были  
 if (br.ready()) {  
 System.*out*.println("Main Server found any messages in channel, let's look at them.");  
  
 // если команда - quit то инициализируем закрытие сервера и  
 // выход из цикла раздачи нитей монопоточных серверов  
 String serverCommand = br.readLine();  
 if (serverCommand.equalsIgnoreCase("quit")) {  
 System.*out*.println("Main Server initiate exiting...");  
 server.close();  
 break;  
 }  
 }  
  
 Socket client = server.accept();  
  
 // после получения запроса на подключение сервер создаёт сокет  
 // для общения с клиентом и отправляет его в отдельную нить  
 // в Runnable(при необходимости можно создать Callable)  
 // монопоточную нить = сервер и тот  
 // продолжает общение от лица сервера  
 *executeIt*.execute(new MessageThread(client));  
 System.*out*.print("Connection accepted.");  
 }  
  
 // закрытие пула нитей после завершения работы всех нитей  
 *executeIt*.shutdown();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

Листинг 3. Класс нити-обработчика запроса

package com.example.tcp.server;  
  
import com.example.shared.TCPMessage;  
import com.fasterxml.jackson.databind.ObjectMapper;  
import lombok.extern.slf4j.Slf4j;  
  
import java.io.DataInputStream;  
import java.io.DataOutputStream;  
import java.io.IOException;  
import java.io.InputStream;  
import java.net.Socket;  
import java.nio.charset.StandardCharsets;  
import java.util.Arrays;  
  
@Slf4j  
public class MessageThread implements Runnable {  
 private final Socket socket;  
 private final SolverFabric solverFabric;  
  
 MessageThread(Socket socket) {  
 this.socket = socket;  
 this.solverFabric = new SolverFabric();  
 }  
  
 public void run() {  
 try {  
 DataOutputStream out = new DataOutputStream(socket.getOutputStream());  
 *log*.debug("DataOutputStream created");  
  
 DataInputStream in = new DataInputStream(socket.getInputStream());  
 *log*.debug("DataInputStream created");  
  
 *log*.info("Server reading from channel");  
  
 String entry = in.readUTF();  
 //*TODO обработка ошибок* ObjectMapper objectMapper = new ObjectMapper();  
 TCPMessage tcpMessage = objectMapper.readValue(entry, TCPMessage.class);  
  
  
 *log*.info("READ from clientDialog message - {}", tcpMessage);  
  
 Solver solver = solverFabric.getSolver(Arrays.*stream*(tcpMessage.getHeaders()).findAny().get());  
 String response = solver.solve(tcpMessage);  
 out.writeUTF(response);  
  
 *log*.info("Server try writing to channel");  
 out.writeUTF("Server reply - " + entry + " - OK");  
 *log*.info("Server Wrote message to clientDialog.");  
  
 out.flush();  
  
  
 *log*.info("Client disconnected");  
 *log*.info("Closing connections & channels.");  
  
 in.close();  
 out.close();  
  
 socket.close();  
  
 *log*.info("Closing connections & channels - DONE.");  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
}

Как видно из листинга 3, как именно будет обрабатываться сообщение решает SolverFabric, представленный в листинге 4.

Листинг 4. SolverFabric

package com.example.tcp.server;  
  
import com.example.tcp.server.solvers.DefaultSolver;  
import com.example.tcp.server.solvers.PressureSolver;  
  
public class SolverFabric {  
  
 public Solver getSolver(String header){  
 switch (header){  
 case "PressureConverter":  
 return new PressureSolver();  
 default:  
 return new DefaultSolver();  
 }  
 }  
}

Согласно паттерну «Фабрика», он возвращает экземпляр интерфейса Solver по листингу 5.

Листинг 5. Интерфейс для фабрики

package com.example.tcp.server;  
  
import com.example.shared.TCPMessage;  
  
public interface Solver {  
 String solve(TCPMessage data);  
}

Для решения задачи «Конвертер величин «мера давления»» был добавлен класс PressureSolver из листинга 6

Листинг 6. PressureSolver

ackage com.example.tcp.server.solvers;  
  
import com.example.shared.TCPMessage;  
import com.example.shared.entity.PressureOutMessage;  
import com.example.tcp.server.Solver;  
import com.fasterxml.jackson.databind.JsonNode;  
import com.fasterxml.jackson.databind.ObjectMapper;  
  
public class PressureSolver implements Solver {  
 @Override  
 public String solve(TCPMessage data) {  
 double pressure = data.getBody().get("pressure").asDouble();  
 String pressureType = data.getBody().get("type").asText();  
 double PascalPressure = 0;  
 double BarPressure = 0;  
 double MmPressure = 0;  
 switch (pressureType) {  
 case "Па":  
 PascalPressure = pressure;  
 BarPressure = PascalToBar(PascalPressure);  
 MmPressure = PascalToMM(PascalPressure);  
 break;  
 case "Бар":  
 BarPressure = pressure;  
 PascalPressure = BarToPascal(BarPressure);  
 MmPressure = PascalToMM(PascalPressure);  
 break;  
 case "мм рт. ст.":  
 MmPressure = pressure;  
 PascalPressure = MmToPascal(MmPressure);  
 BarPressure = PascalToBar(PascalPressure);  
 break;  
 }  
 ObjectMapper mapper = new ObjectMapper();  
 data.setBody(mapper.valueToTree(new PressureOutMessage(PascalPressure, BarPressure, MmPressure)));  
 return data.toString();  
 }  
  
 private double PascalToBar(double Pascal)  
 {  
 return Pascal \* Math.*pow*(10, -5);  
 }  
 private double PascalToMM(double Pascal)  
 {  
 return Pascal \* Math.*pow*(10, -3) \* 7.5006;  
 }  
 private double BarToPascal(double Bar)  
 {  
 return Bar \* Math.*pow*(10, 5);  
 }  
 private double MmToPascal(double Mm)  
 {  
 return Mm \* 133.322;  
 }  
}

После обработки сообщения из Solver’а мы получаем уже готовую JSON строку, которую можем отправить на клиент. После отправки сообщения TCP подключение закрывается, чтобы не нагружать ни сервер, ни канал связи.

На этом разбор серверной части можно считать исчерпывающим.

В клиентской части был использован JavaFX для прототипа пользовательского интерфейса. В листинге 7 представлена точка входа, классическая для такого типа приложений

Листинг 7. Main метод JavaFX

package org.openjfx;  
  
import javafx.application.Application;  
import javafx.fxml.FXMLLoader;  
import javafx.scene.Parent;  
import javafx.scene.Scene;  
import javafx.stage.Stage;  
  
import java.io.IOException;  
  
*/\*\*  
 \* JavaFX App  
 \*/*public class App extends Application {  
  
 private static Scene *scene*;  
  
 @Override  
 public void start(Stage stage) throws IOException {  
 *scene* = new Scene(*loadFXML*("primary"));  
 *scene*.getStylesheets().add(App.class.getResource("styles.css").toExternalForm());  
 stage.setScene(*scene*);  
 stage.show();  
 }  
  
 static void setRoot(String fxml) throws IOException {  
 *scene*.setRoot(*loadFXML*(fxml));  
 }  
  
 private static Parent loadFXML(String fxml) throws IOException {  
 FXMLLoader fxmlLoader = new FXMLLoader(App.class.getResource(fxml + ".fxml"));  
 return fxmlLoader.load();  
 }  
  
 public static void main(String[] args) {  
 *launch*();  
 }  
  
}

Два логически связанных файла: PrimaryConrtroller.java и primary.fxml - cодержат в себе описание формы. В primary.fxml – FXML разметка графических элементов, а в PrimaryConrtroller.java логика работы

Листинг 8. PrimaryConrtroller.java

package org.openjfx;  
  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.InputStreamReader;  
import java.nio.charset.StandardCharsets;  
import java.util.stream.Collectors;  
  
import javafx.collections.FXCollections;  
import javafx.collections.ObservableList;  
import javafx.event.ActionEvent;  
import javafx.fxml.FXML;  
import javafx.scene.control.ComboBox;  
import javafx.scene.control.TextArea;  
import javafx.scene.input.KeyEvent;  
  
public class PrimaryController {  
  
 public TextArea inputArea;  
 public TextArea outputArea;  
 public ComboBox<String> comboBox;  
 @FXML  
 public void initialize() {  
 comboBox.setItems(*combo*);  
 comboBox.setValue("Default");  
  
 try (InputStream inputStream = PrimaryController.class.getClassLoader().getResourceAsStream("Default.json")) {  
 String text = new BufferedReader(  
 new InputStreamReader(inputStream, StandardCharsets.*UTF\_8*))  
 .lines()  
 .collect(Collectors.*joining*("\n"));  
 inputArea.setText(text);  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 private static final ObservableList<String> *combo* = FXCollections.*observableArrayList*("Default", "Pressure");  
 @FXML  
 private void switchToSecondary() {  
 Client client = new Client(inputArea.getText());  
 client.run();  
 String res = client.getResult();  
 outputArea.setText(res);  
 }  
  
 public void initComboBox(KeyEvent keyEvent) {  
  
 }  
  
 public void selComboBox(ActionEvent actionEvent) {  
 try (InputStream inputStream = PrimaryController.class.getClassLoader().getResourceAsStream(comboBox.getValue() +".json")) {  
 String text = new BufferedReader(  
 new InputStreamReader(inputStream, StandardCharsets.*UTF\_8*))  
 .lines()  
 .collect(Collectors.*joining*("\n"));  
 inputArea.setText(text);  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

Листинг 9. primary.fxml

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import javafx.geometry.\*?>  
<?import javafx.scene.control.\*?>  
<?import javafx.scene.layout.\*?>  
  
<VBox alignment="CENTER" spacing="20.0" xmlns="http://javafx.com/javafx/17.0.2-ea" xmlns:fx="http://javafx.com/fxml/1" fx:controller="org.openjfx.PrimaryController">  
 <children>  
 <ComboBox fx:id="comboBox" onKeyPressed="#initComboBox" onAction="#selComboBox" >  
 </ComboBox>  
 <Label text="Запрос" />  
 <TextArea fx:id="inputArea" />  
 <Label text="Ответ" />  
 <TextArea fx:id="outputArea" disable="true" />  
 <Button fx:id="primaryButton" onAction="#switchToSecondary" text="Отправить" />  
 </children>  
  
 <padding>  
 <Insets bottom="20.0" left="20.0" right="20.0" top="20.0" />  
 </padding>  
</VBox>

Из листингов 8 и 9 видим, что на форме расположен элемент с ниспадающим списком, два текстовых поля и кнопка.

В ниспадающем списке мы можем выбрать шаблон JSON запроса. После выбора он будет отображаться в текстовом поле inputArea.

По нажатию кнопки запрос, записанный в inputArea отправляется на сервер. Ответ на запрос записывается в outputArea.

Так как задание подразумевает изучение TCP протокола и особенностей клиент-серверных приложений, то нашел нецелесообразным придумывать сложный графический интерфейс.

Связь с сервером выполняет класс Client из листинга 10.

Листинг 10. Client

package org.openjfx;  
  
  
import com.example.shared.TCPMessage;  
import com.fasterxml.jackson.core.JsonProcessingException;  
import lombok.extern.slf4j.Slf4j;  
  
import java.io.DataInputStream;  
import java.io.DataOutputStream;  
import java.io.IOException;  
import java.net.Socket;  
import java.nio.charset.StandardCharsets;  
  
@Slf4j  
public class Client implements Runnable {  
  
 private Socket socket;  
 private final String data;  
 private String result;  
  
 public Client(String data) {  
 this.data = data;  
 try {  
 // создаём сокет общения на стороне клиента в конструкторе объекта  
 socket = new Socket("localhost", 3345);  
 System.*out*.println("Client connected to socket");  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 public void run() {  
 try (DataOutputStream oos = new DataOutputStream(socket.getOutputStream());  
 DataInputStream ois = new DataInputStream(socket.getInputStream())) {  
 *log*.info("Client oos & ois initialized");  
  
  
 // пишем сообщение автогенерируемое циклом клиента в канал  
 // сокета для сервера  
 *log*.info("send - " + data);  
 oos.writeUTF(data);  
  
 // проталкиваем сообщение из буфера сетевых сообщений в канал  
 oos.flush();  
 // ждём чтобы сервер успел прочесть сообщение из сокета и  
 // ответить  
 *log*.info("Client wrote & start waiting for data from server...");  
  
 // забираем ответ из канала сервера в сокете  
 // клиента и сохраняем её в ois переменную, печатаем на  
 // консоль  
 *log*.info("reading...");  
 result = ois.readUTF();  
 *log*.info("Receive {}", result);  
  
  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public String getResult() {  
 return result;  
 }  
}

Вывод: изучил на практике TCP протокол и методы создания клиент-серверных приложений. По итогам написанного приложения моуг выделить некоторые особенности.

Плюсы:

* Отличная расширяемость. Благодаря тому, что TCP подключение в кодовой базе отделено от обработки сообщений, можно добавлять новый функционал без проблем и не боясь сломать то, что уже работало.
* Низкая нагрузка. Так как TCP соединение не держится открытым постоянно, то сервер освобождается от дополнительно нагрузки.

Минусы:

* Не выполнена обработка ошибка в формате принятого сообщения на сервер
* Не выполнена обработка множества заголовков, хотя она и предусмотрена в сообщении.