

abr 18, 18 22:24

UserGestor.java

Page 1/2

```

1  import Message.*;
2  import org.apache.log4j.Logger;
3  import sun.misc.Signal;
4
5  public class UserGestor {
6      private static final Logger LOGGER = Logger.getLogger(UserGestor.class);
7      private static final Settings SETTINGS = Settings.from("user-gestor.properties");
8
9      private static final String RABBITMQ_HOST = SETTINGS.get("RABBITMQ_HOST",
10 "localhost");
11 private static final int RABBITMQ_PORT = SETTINGS.get("RABBITMQ_PORT",
12 5672);
13 private static final String CLIENT_QUEUE = SETTINGS.get("CLIENT_QUEUE", "CLI
14 ENT");
15
16 private String consumerClientTag;
17
18 private final CommunicationWrapper communication;
19
20 private DB db;
21
22 UserGestor() throws Exception {
23     communication = CommunicationWrapper.getConnection(RABBITMQ_HOST, RABBITM
24 Q_PORT);
25     if (communication == null) {
26         LOGGER.fatal("Cannot open communication");
27         throw new Exception("Cannot open communication");
28     }
29
30     if (!communication.queueDeclare(CLIENT_QUEUE)) {
31         LOGGER.fatal("Cannot declare queue " + CLIENT_QUEUE);
32         communication.close();
33         throw new Exception("Cannot declare queue " + CLIENT_QUEUE);
34     }
35
36     db = new DB();
37
38     private void registerSIGINT() {
39         Signal.handle(new Signal("INT"), sig → {
40             LOGGER.info("SIGINT detected. Closing connection");
41             communication.detach(consumerClientTag);
42             communication.close();
43             LOGGER.info("Connection closed");
44         });
45     }
46
47     private Message handlerRequestConnection(Message request) {
48         LOGGER.info("Request connection from " + request.getUser() + " in radio " + reques
49 t.getRadio());
50         MessageBuilder messageBuilder = new MessageBuilder();
51         if (db.existStation(request.getRadio())) {
52             if (db.userCanHearRadio(request.getUser(), request.getRadio())) {
53                 LOGGER.info("Accepted connection from " + request.getUser() + " in radio "
54 + request.getRadio());
55                 db.addUserInRadio(request.getUser(), request.getUserQueue(), req
56 uest.getRadio());
57                 messageBuilder.setType(MessageType.CONNECTION_ACCEPTED);
58                 //communication.put(request.getUserQueue(), response);
59             } else {
60                 messageBuilder
61                     .setType(MessageType.CONNECTION_DENIED)
62                     .setError("You can not listen to more radios with current user").build();
63                 LOGGER.info("Revoke request connection from " + request.getUser() + " in ra
64 dio " + request.getRadio());

```

abr 18, 18 22:24

UserGestor.java

Page 2/2

```

59         //communication.put(request.getUserQueue(), response);
60     }
61     } else {
62         messageBuilder
63             .setType(MessageType.CONNECTION_DENIED)
64             .setError("Radio no exist or not in transmission").build();
65         LOGGER.info("Revoke request connection from " + request.getUser() + " radio " +
66 request.getRadio() + " no exist");
67         //communication.put(request.getUserQueue(), response);
68     }
69     return messageBuilder.build();
70 }
71
72 private void start() {
73     LOGGER.info("Client message request");
74
75     consumerClientTag = communication.append(CLIENT_QUEUE, message → {
76         if (message.getType() == MessageType.REQUEST_CONNECTION) {
77             Message response = handlerRequestConnection(message);
78             communication.put(message.getUserQueue(), response);
79         } else if (message.getType() == MessageType.REQUEST_RADIOS) {
80             StringBuilder stationsFlat = new StringBuilder();
81             db.getStations().forEach(s → stationsFlat.append("- ").append(s
82 ).append("\n"));
83             Message response = new MessageBuilder()
84                 .setType(MessageType.RESPONSE_RADIOS)
85                 .setInfo(stationsFlat.toString())
86                 .build();
87             communication.put(message.getUserQueue(), response);
88         } else if (message.getType() == MessageType.KEEP_ALIVE) {
89             db.updateUserActivity(message.getUser());
90         } else if (message.getType() == MessageType.END_CONNECTION) {
91             db.deleteUserFromRadio(message.getUser(), message.getUserQueue()
92 , message.getRadio());
93         } else {
94             LOGGER.warn("Unhandled Message with type " + message.getType());
95         }
96     });
97     registerSIGINT();
98 }
99
100 public static void main(String[] argv) {
101     UserGestor userGestor = null;
102     try {
103         userGestor = new UserGestor();
104     } catch (Exception e) {
105         LOGGER.info("Cannot start user gestor");
106         LOGGER.debug(e);
107         System.exit(1);
108     }
109     userGestor.start();
110 }
111 }

```

abr 19, 18 0:31

Station.java

Page 1/2

```

1  import Message.*;
2  import org.apache.log4j.Logger;
3
4  import java.io.File;
5  import java.io.FileInputStream;
6  import java.io.IOException;
7  import java.util.Arrays;
8  import java.util.concurrent.TimeUnit;
9  import java.util.concurrent.atomic.AtomicBoolean;
10
11 public class Station {
12     private static final Logger LOGGER = Logger.getLogger(Station.class);
13     private static final Settings SETTINGS = Settings.from("station.properties");
14
15     private static final String RADIO_QUEUE = SETTINGS.get("RADIO_QUEUE", "RADIO_QUEUE");
16     private static final String RABBITMQ_HOST = SETTINGS.get("RABBITMQ_HOST", "localhost");
17     private static final int RABBITMQ_PORT = SETTINGS.get("RABBITMQ_PORT", 5672);
18
19     private static final int MESSAGE_EXPIRATION_TIME_SECONDS = SETTINGS.get("MESSAGE_EXPIRATION_TIME_SECONDS", 60);
20     private static final int PACKAGE_BYTE_SIZE = SETTINGS.get("PACKAGE_BYTE_SIZE", 192000);
21     private static final int TIME_PER_PACKAGE_SECONDS = SETTINGS.get("TIME_PER_PACKAGE_SECONDS", 2);
22
23     private final CommunicationWrapper communication;
24     private final String name;
25     private AtomicBoolean transmissionStarted;
26
27     private Station(String name) throws Exception {
28         communication = CommunicationWrapper.getConnection(RABBITMQ_HOST, RABBITMQ_PORT);
29         if (communication == null) {
30             LOGGER.warn("Cannot establish communication");
31             throw new Exception("Cannot establish communication");
32         }
33
34         if (!communication.queueDeclare(RADIO_QUEUE)) {
35             LOGGER.warn("Cannot declare Queue " + RADIO_QUEUE);
36             throw new Exception("Cannot declare Queue " + RADIO_QUEUE);
37         }
38
39         this.name = name;
40         this.transmissionStarted.set(false);
41
42     }
43
44     private void startTransmission(String filePath) throws InterruptedException {
45         File file = new File(filePath);
46         try (FileInputStream fis = new FileInputStream(file)) {
47             String fileExtension = filePath.substring(filePath.lastIndexOf('.') + 1);
48             int totalBytes = fis.available();
49             LOGGER.info("Attempt to send " + file.getName() + ". Content type " + fileExtension);
50
51             LOGGER.info("Total bytes to send " + totalBytes);
52             int byteCountRead = 0;
53             int byteCount;
54             byte[] bytes = new byte[PACKAGE_BYTE_SIZE];
55             transmissionStarted.set(true);
56             while ((byteCount = fis.read(bytes)) != -1) {
57                 byteCountRead += byteCount;

```

abr 19, 18 0:31

Station.java

Page 2/2

```

57         Message message = new MessageBuilder()
58             .setType(MessageType.RADIO_PACKAGE)
59             .setRadio(name)
60             .setContentType(fileExtension)
61             .setPayload(Arrays.copyOfRange(bytes, 0, byteCount))
62             .build();
63         communication.put(RADIO_QUEUE, message, MESSAGE_EXPIRATION_TIME_SECONDS);
64         LOGGER.info("Sent " + byteCount + " bytes. " + (byteCountRead*100)/totalBytes + "%");
65
66         TimeUnit.SECONDS.sleep(TIME_PER_PACKAGE_SECONDS);
67
68         LOGGER.info("End stream");
69     } catch (IOException e) {
70         LOGGER.debug(e);
71         LOGGER.warn("Cannot read file stream " + filePath);
72     }
73 }
74
75 private void stopTransmission() {
76     LOGGER.info("Closing connection");
77     if (transmissionStarted.compareAndSet(true, false)) {
78         Message endMessage = new MessageBuilder().setType(MessageType.END_TRANSMISSION).build();
79         communication.put(RADIO_QUEUE, endMessage, MESSAGE_EXPIRATION_TIME_SECONDS);
80         communication.close();
81     }
82 }
83
84 public static void main(String[] args) {
85     if (args.length < 2) {
86         System.out.println("Use: ./station <<name>> <<file>>");
87         LOGGER.fatal("Invalid parameters");
88         return;
89     }
90
91     try {
92         Station station = new Station(args[0]);
93         station.startTransmission(args[1]);
94         station.stopTransmission();
95     } catch (Exception e) {
96         LOGGER.debug(e);
97         LOGGER.warn("Cannot create station");
98     }
99 }
100 }

```

abr 19, 18 0:24

Settings.java

Page 1/1

```

1 import org.apache.log4j.Logger;
2
3 import java.io.*;
4 import java.util.Properties;
5
6 public class Settings {
7     private static final Logger LOGGER = Logger.getLogger(Settings.class);
8     private final Properties properties;
9     private Settings() {
10         properties = new Properties();
11     }
12
13     public static Settings from(String propertiesFile) {
14         InputStream input = null;
15         Settings settings = new Settings();
16         try {
17             input = new FileInputStream(propertiesFile);
18             settings.properties.load(input);
19             LOGGER.info(String.format("%s" was loaded correctly", propertiesFile));
20         } catch (IOException ex) {
21             LOGGER.error(String.format("Cannot load \"%s\" using all default values", propertiesFile));
22         } finally {
23             if (input != null) {
24                 try {
25                     input.close();
26                 } catch (IOException e) {
27                     LOGGER.warn("IOException when attempt to close " + propertiesFile);
28                     LOGGER.debug(e);
29                 }
30             }
31         }
32         return settings;
33     }
34
35     public int get(String name, int defaultValue) {
36         try {
37             return Integer.parseInt(properties.getProperty(name, String.valueOf(defaultValue)));
38         } catch (NumberFormatException e) {
39             LOGGER.warn("Invalid Int value " + properties.getProperty(name) + " of property: " + name + ". Return default");
40             return defaultValue;
41         }
42     }
43
44     public Boolean get(String name, boolean defaultValue) {
45         return Boolean.valueOf(properties.getProperty(name, String.valueOf(defaultValue)));
46     }
47
48     public String get(String name, String defaultValue) {
49         return properties.getProperty(name, defaultValue);
50     }
51 }

```

abr 18, 18 19:23

RadioListener.java

Page 1/4

```

1 import Message.*;
2
3 import org.apache.log4j.*;
4 import sun.misc.Signal;
5
6 import java.io.File;
7 import java.io.FileOutputStream;
8 import java.io.IOException;
9 import java.util.concurrent.Executors;
10 import java.util.concurrent.ScheduledExecutorService;
11 import java.util.concurrent.TimeUnit;
12 import java.util.concurrent.atomic.AtomicBoolean;
13 import java.util.function.Consumer;
14
15 public class RadioListener {
16     private static final Logger LOGGER = Logger.getLogger(RadioListener.class);
17     private static final Settings SETTINGS = Settings.from("radio-listener.properties");
18
19     private static final String RABBITMQ_HOST = SETTINGS.get("RABBITMQ_HOST", "localhost");
20     private static final int RABBITMQ_PORT = SETTINGS.get("RABBITMQ_PORT", 5672);
21     private static final String CLIENT_QUEUE = SETTINGS.get("CLIENT_QUEUE", "CLIENT");
22     private static final int TIMEOUT_SECONDS = SETTINGS.get("TIMEOUT_SECONDS", 10);
23     private static final int KEEP_ALIVE_POLL_SECONDS = SETTINGS.get("KEEP_ALIVE_POLL_SECONDS", 60);
24     private static final int POOL_SIZE = SETTINGS.get("POOL_SIZE", 10);
25
26     private final String user;
27     private final String radio;
28
29     private CommunicationWrapper comm;
30     private String consumerTag;
31     private String listenQueue;
32     private AtomicBoolean isConnected;
33
34     private ScheduledExecutorService keepAliveScheduler;
35
36     private void initCommunication() throws IOException {
37         comm = CommunicationWrapper.getConnection(RABBITMQ_HOST, RABBITMQ_PORT);
38         if (comm == null) {
39             LOGGER.fatal("Cannot open connection. Server is down");
40             throw new IOException("Cannot open connection. Server is up?");
41         }
42
43         listenQueue = comm.queueDeclare();
44         if (listenQueue == null) {
45             throw new IOException("Cannot declare queue to receive response");
46         }
47         isConnected = new AtomicBoolean(false);
48     }
49
50     private RadioListener() throws IOException {
51         initCommunication();
52         this.user = null;
53         this.radio = null;
54         this.keepAliveScheduler = null;
55     }
56
57     private RadioListener(String user, String radio) throws IOException {
58         initCommunication();
59         this.user = user;
60     }
61 }

```

abr 18, 18 19:23

RadioListener.java

Page 2/4

```

62     this.radio = radio;
63     this.keepAliveScheduler = Executors.newScheduledThreadPool(PPOOL_SIZE);
64
65 }
66
67 private File createFile(String extension) throws IOException {
68     String fileName = this.user + "-" + this.radio + "." + extension;
69     File fileStream = new File(fileName);
70     int i = 1;
71     while (!fileStream.exists()) {
72         fileName = this.user + "-" + this.radio + "-" + i + "." + extension;
73         fileStream = new File(fileName);
74         i++;
75     }
76     if (!fileStream.createNewFile()) {
77         if (!fileStream.exists()) {
78             throw new IOException("Failed on create file to write");
79         }
80     }
81     return fileStream;
82 }
83
84 private void handleRadioPackage(Message message) {
85     FileOutputStream out;
86     try {
87         File fileStream = createFile(message.getContentType());
88         out = new FileOutputStream(fileStream, true);
89         byte[] bytes = message.getPayload();
90         LOGGER.debug("Write " + bytes.length + " in " + fileStream.getName());
91         out.write(bytes);
92         out.close();
93     } catch (IOException e) {
94         LOGGER.warn("Cannot write radio package. Ignoring it");
95         LOGGER.debug(e);
96     }
97 }
98
99 private void handleResponse(Message res) {
100     if (res.getType() == MessageType.CONNECTION_ACCEPTED) {
101         LOGGER.info("Receive Connection accepted");
102         isConnected.set(true);
103         startSchedulerToSendKeepAlive();
104         System.out.println("Connected to radio " + radio + "");
105     }
106     if (res.getType() == MessageType.CONNECTION_DENIED) {
107         LOGGER.info("Connection denied");
108         System.out.println("Cannot connect with radio. Error: \"" + res.getError() + "\"");
109     }
110     if (res.getType() == MessageType.RADIO_PACKAGE) {
111         LOGGER.info("Receive radio package");
112         handleRadioPackage(res);
113     }
114     if (res.getType() == MessageType.END_CONNECTION) {
115         LOGGER.info("Receive end connection");
116         stop();
117     }
118 }
119
120 private void startSchedulerToSendKeepAlive() {
121     keepAliveScheduler.scheduleAtFixedRate(() -> {
122         LOGGER.info("Send keep alive to server");
123         Message message = new MessageBuilder()
124             .setType(MessageType.KEEP_ALIVE)
125             .setUser(user)
126             .build();

```

abr 18, 18 19:23

RadioListener.java

Page 3/4

```

127         comm.put(CLIENT_QUEUE, message);
128     }, KEEP_ALIVE_POLL_SECONDS, KEEP_ALIVE_POLL_SECONDS, TimeUnit.SECONDS);
129 }
130
131 private void disconnect() {
132     comm.put(CLIENT_QUEUE, new MessageBuilder()
133         .setUser(user)
134         .setRadio(radio)
135         .setType(MessageType.END_CONNECTION).build()
136     );
137 }
138
139 private synchronized void stop() {
140     LOGGER.info("Stop Radio Listener");
141     if (isConnected.compareAndSet(true, false)) {
142         disconnect();
143     }
144     if (keepAliveScheduler != null) {
145         keepAliveScheduler.shutdownNow();
146     }
147     if (!comm.detach(consumerTag)) {
148         LOGGER.warn("Cannot detach");
149     }
150     comm.deleteQueue(listenQueue);
151     comm.close();
152     LOGGER.info("Exit");
153 }
154
155 private void waitResponseWithTimeout(Consumer<Message> handler) {
156     ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
157
158     ;
159     LOGGER.info("Start timeout to wait response. TIMEOUT=" + TIMEOUT_SECONDS);
160     scheduler.schedule(() -> {
161         LOGGER.info("TIMEOUT: Wake-up. Close connection. The servers are not working"),
162         TIMEOUT_SECONDS,
163         TimeUnit.SECONDS
164     });
165
166     // Wait responses
167     LOGGER.info("Waiting response");
168     consumerTag = comm.append(listenQueue, res -> {
169         handler.accept(res);
170         scheduler.shutdownNow();
171     });
172
173     scheduler.shutdown();
174     try {
175         scheduler.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);
176     } catch (InterruptedException ignored) {
177         LOGGER.info("Interrupted Exception");
178     }
179 }
180
181 private void startListener() {
182     Message request = new MessageBuilder().setType(MessageType.REQUEST_CONNE
183     CTION)
184         .setClientQueue(listenQueue)
185         .setRadio(radio)
186         .setUser(user)
187         .build();
188     comm.put(CLIENT_QUEUE, request, TIMEOUT_SECONDS);
189
190     waitResponseWithTimeout(this::handleResponse);

```

abr 18, 18 19:23

RadioListener.java

Page 4/4

```

191     if (!isConnected.get()) {
192         stop();
193     }
194 }
195
196 private void listRadios() {
197     // Send request
198     Message request = new MessageBuilder().
199         setType(MessageType.REQUEST_RADIOS)
200         .setClientQueue(listenQueue)
201         .build();
202     comm.put(CLIENT_QUEUE, request, TIMEOUT_SECONDS);
203
204     waitResponseWithTimeout(res -> {
205         LOGGER.info("Received response from server");
206         if (res.getType() == MessageType.RESPONSE_RADIOS) {
207             System.out.println("Radios:\n" + res.getInfo());
208         }
209     });
210
211     stop();
212 }
213
214 private void start() {
215     if (this.user == null) {
216         listRadios();
217     } else {
218         startListener();
219     }
220 }
221
222 public static void main(String[] args) {
223     if (args.length == 0) {
224         System.out.println(" - Listen Radio: ./radio-listener <<user>> <<radio>>");
225         System.out.println(" - List Radios: ./radio-listener list");
226         return;
227     }
228
229     boolean listRadios = (args.length == 1 ^ args[0].toLowerCase().equals("list
230 "));
231
232     try {
233         RadioListener radioListener = (listRadios) ? new RadioListener() : n
234 ew RadioListener(args[0], args[1]);
235         radioListener.start();
236         Signal.handle(new Signal("INT"), sig -> {
237             LOGGER.info("SIGINT detected. Closing connection");
238             radioListener.stop();
239             LOGGER.info("Connection closed");
240         });
241     } catch (IOException e) {
242         LOGGER.fatal("Cannot start radio-listener");
243         LOGGER.debug(e);
244     }
245 }
246
247 }
248
249 }

```

abr 18, 18 19:05

Initializer.java

Page 1/1

```

1  import org.apache.log4j.Logger;
2
3  public class Initializer {
4      private static final Logger LOGGER = Logger.getLogger(Initializer.class);
5      private static final Settings SETTINGS = Settings.from("initializer.properties");
6      public static void main(String[] args) {
7          LOGGER.info("Initialize all queues to use");
8          CommunicationWrapper comm = CommunicationWrapper.getConnection(
9              SETTINGS.get("RABBITMQ_HOST", "localhost"),
10             SETTINGS.get("RABBITMQ_PORT", 5672)
11         );
12         if (comm == null) {
13             LOGGER.fatal("Cannot connect. Abort initializer");
14             return;
15         }
16
17         comm.queueDeclare(SETTINGS.get("ADMIN_REQUEST_QUEUE", "ADMIN_REQUEST")
18     );
19
20         comm.queueDeclare(SETTINGS.get("ADMIN_RESPONSE_QUEUE", "ADMIN_RESPONSE
21     "));
22
23         comm.queueDeclare(SETTINGS.get("RADIO_QUEUE", "RADIO"));
24
25         comm.queueDeclare(SETTINGS.get("CLIENT_QUEUE", "CLIENT"));
26
27         comm.close();
28         LOGGER.info("OK. Initializer");
29     }

```

abr 19, 18 16:16

Main.java

Page 1/1

```

1
2
3 import java.io.IOException;
4 import java.util.ArrayList;
5 import java.util.List;
6 import java.util.concurrent.*;
7
8 public class Main {
9     private static String printThread() {
10         return Thread.currentThread().getName();
11     }
12
13     private static void testExecutors() throws InterruptedException {
14         ExecutorService executor = Executors.newFixedThreadPool(5);
15
16         Object mutex = new Object();
17
18         Callable<Integer> task = () -> {
19             synchronized (mutex) {
20                 System.out.println(printThread() + " Tomo mutex");
21                 TimeUnit.SECONDS.sleep(3);
22                 System.out.println(printThread() + " Libero mutex");
23             }
24             return 1;
25         };
26
27         List<Callable<Integer>> runnables = new ArrayList<>();
28         while (runnables.size() < 10) {
29             runnables.add(task);
30         }
31
32         executor.invokeAll(runnables);
33         executor.shutdown();
34         executor.awaitTermination(Long.MAX_VALUE, TimeUnit.MILLISECONDS);
35
36         System.out.println(printThread() + " Hilo principal!");
37     }
38
39
40     private static void testFileLock() {
41         try {
42             FileCellBlock file = new FileCellBlock("dummydb", 100);
43             file.insert("1");
44             file.insert("2");
45             file.insert("3");
46             file.iterFile(System.out::println);
47             file.delete("1", String::compareTo);
48             System.out.println("-----");
49             file.iterFile(System.out::println);
50             System.out.println("-----");
51             file.insert("4");
52             file.iterFile(System.out::println);
53         } catch (IOException | IndexOutOfBoundsException e) {
54             e.printStackTrace();
55         }
56
57     }
58
59
60     public static void main(String[] args) throws InterruptedException {
61         //testExecutors();
62         testFileLock();
63         System.exit(0);
64     }
65 }

```

abr 18, 18 21:41

Destructor.java

Page 1/1

```

1 import org.apache.log4j.Logger;
2
3 public class Destructor {
4     private static final Logger LOGGER = Logger.getLogger(Destructor.class);
5     private static final Settings SETTINGS = Settings.from("destructor.properties");
6     public static void main(String[] args) {
7         LOGGER.info("Desruct all Queues and Databases");
8         CommunicationWrapper comm = CommunicationWrapper.getConnection(
9             SETTINGS.get("RABBITMQ_HOST", "localhost"),
10             SETTINGS.get("RABBITMQ_PORT", 5672)
11         );
12         if (comm == null) {
13             LOGGER.fatal("Cannot connect");
14             return;
15         }
16
17         comm.deleteQueue(SETTINGS.get("ADMIN_REQUEST_QUEUE", "ADMIN_REQUEST"));
18
19         comm.deleteQueue(SETTINGS.get("ADMIN_RESPONSE_QUEUE", "ADMIN_RESPONSE"));
20
21         comm.deleteQueue(SETTINGS.get("RADIO_QUEUE", "RADIO"));
22
23         comm.deleteQueue(SETTINGS.get("CLIENT_QUEUE", "CLIENT"));
24
25         if (SETTINGS.get("CLEAN_DATABASES", false)) {
26             LOGGER.info("DBs cleaned");
27             DB.cleanDatabases();
28         }
29
30         comm.close();
31         LOGGER.info("OK. End destructor");
32     }
33 }

```

abr 19, 18 18:34

FileCellBlock.java

Page 1/3

```

1  import org.apache.log4j.Logger;
2
3  import java.io.*;
4  import java.nio.ByteBuffer;
5  import java.nio.channels.FileChannel;
6  import java.nio.channels.FileLock;
7  import java.nio.file.StandardOpenOption;
8  import java.util.*;
9  import java.util.concurrent.atomic.AtomicInteger;
10 import java.util.function.BiPredicate;
11 import java.util.function.Consumer;
12 import java.util.function.Function;
13 import java.util.function.Predicate;
14 import java.util.stream.Collectors;
15 import java.util.stream.Collectors;
16
17 public class FileCellBlock {
18     private static final Logger LOGGER = Logger.getLogger(FileCellBlock.class);
19
20     private static final char NULL = '\0';
21
22     private final int blockSize;
23     private final File file;
24
25     FileCellBlock(String file, int blockSize) throws IOException {
26         this.blockSize = blockSize;
27         this.file = new File(file);
28         createFile();
29     }
30
31     private void createFile() throws IOException {
32         if (!file.exists() ^ !file.createNewFile() ^ !file.exists()) {
33             if (!file.createNewFile()) {
34                 if (!file.exists()) {
35                     LOGGER.fatal("Cannot create file " + file.toString());
36                     throw new IOException("Cannot create file " + file.toString());
37                 }
38             }
39         }
40         LOGGER.info(String.format("Created file DB: \"%s\"", file.toString()));
41     }
42
43     private byte[] generateNullBlock() {
44         // Default NULL Block.
45         byte[] block = new byte[blockSize];
46         Arrays.fill(block, (byte) NULL);
47         return block;
48     }
49
50     private byte[] toBlock(String s) {
51         byte[] string = s.replaceAll(String.valueOf(NULL), "").getBytes();
52         if (string.length > blockSize) {
53             return null;
54         }
55         byte[] block = generateNullBlock();
56         System.arraycopy(string, 0, block, 0, string.length);
57         return block;
58     }
59
60     private String toString(byte[] block) {
61         if (Arrays.equals(block, generateNullBlock())) {
62             return "<<FREE BLOCK>>";
63         }
64         String str = new String(block);
65         int indexOfEnd = str.indexOf('\0');
66

```

abr 19, 18 18:34

FileCellBlock.java

Page 2/3

```

67     return indexOfEnd != -1 ? str.substring(0, str.indexOf(NULL)) : str;
68 }
69
70 private void writeBlockInEnd(final String s) {
71     // Write s in the end
72     byte[] block = toBlock(s);
73     if (block == null) {
74         return;
75     }
76     try (FileOutputStream out = new FileOutputStream(file, true)) {
77         FileLock lock = out.getChannel().lock(out.getChannel().position(), b
lockSize, false);
78         out.write(block);
79         lock.release();
80     } catch (IOException e) {
81         LOGGER.error(String.format("Cannot write file '%s'", file.toString()));
82     }
83 }
84
85 private void writeBlock(final String s, int position) {
86     // Write Block in position
87     // NOTE: The file was block from position*blockSize a blockSize length
88     // If position < 0. Write on end
89     if (position < 0) {
90         writeBlockInEnd(s);
91         return;
92     }
93     byte[] block = toBlock(s);
94     if (block == null) {
95         return;
96     }
97     try (FileChannel out = FileChannel.open(file.toPath(), StandardOpenOptio
n.WRITE)) {
98         long offset = position * blockSize;
99         FileLock lock = out.lock(position, blockSize, false);
100        out.write(ByteBuffer.wrap(block), offset);
101        lock.release();
102    } catch (IOException e) {
103        LOGGER.error(String.format("Cannot write file '%s'", file.toString()));
104    }
105 }
106
107 public void iterFile(final BiPredicate<Integer, String> handleBlock) {
108     // Iter file lockin to read per block
109     // If handleBlock return true. Stop iter.
110     try (FileInputStream in = new FileInputStream(file)) {
111         int byteCount = 0;
112         long offset = 0L;
113         int position = 0;
114         boolean stop = false;
115         while (byteCount != -1 ^ !stop) {
116             FileLock lock = in.getChannel().lock(offset, blockSize, true);
117             byte[] bytes = new byte[blockSize];
118             byteCount = in.read(bytes);
119             if (byteCount != -1) {
120                 stop = handleBlock.test(position, toString(bytes));
121             }
122             lock.release();
123             offset += blockSize;
124             position += 1;
125         }
126     } catch (IOException e) {
127         LOGGER.error(String.format("Cannot write file '%s'", file.toString()));
128         LOGGER.debug(e);
129     }
130 }

```

abr 19, 18 18:34

FileCellBlock.java

Page 3/3

```

131
132 public void iterFile(final Consumer<String> handleBlock) {
133     // Iter file. Pass handler to handled al String per block
134     iterFile((pos, string) → {
135         handleBlock.accept(string);
136         return false;
137     });
138 }
139
140 private int getPosition(String s, Comparator<String> comparator) {
141     AtomicInteger position = new AtomicInteger(-1);
142     iterFile((pos, string) → {
143         if (comparator.compare(s, string) == 0) {
144             position.set(pos);
145             return true;
146         }
147         return false;
148     });
149     return position.get();
150 }
151
152 public void update(Function<String, String> updater) {
153     // Search s in all blocks.
154     // If exist (comparator return true), then update it.
155     // If no exist, write on end.
156     Map<Integer, String> newPositionsStrings = new HashMap<>();
157     iterFile((pos, string) → {
158         String newValue = updater.apply(string);
159         if (string.equalsIgnoreCase(newValue)) {
160             newPositionsStrings.put(pos, newValue);
161         }
162         return false;
163     });
164     newPositionsStrings.entrySet().forEach(e → writeBlock(e.getValue(), e.getKey()));
165 }
166
167 public void insert(String s) {
168     // Write block in first free position.
169     // Is no exist free position. write in the end
170     String nullString = toString(generateNullBlock());
171     writeBlock(s, getPosition(nullString, String::compareTo));
172 }
173
174 public void delete(String s, Comparator<String> comparator) {
175     // Delete block who match with s
176     // Ignoring if not exist
177     String nullString = toString(generateNullBlock());
178     int position = getPosition(s, comparator);
179     if (position != -1) {
180         writeBlock(nullString, position);
181     }
182 }
183
184 public List<String> find(Predicate<String> comparator) {
185     List<String> collect = new ArrayList<>();
186     iterFile((pos, string) → {
187         if (comparator.test(string)) {
188             collect.add(string);
189         }
190         return false;
191     });
192     return collect;
193 }
194 }
195

```

abr 19, 18 10:18

DB.java

Page 1/5

```

1 import org.apache.log4j.Logger;
2 import org.json.simple.JSONArray;
3 import org.json.simple.JSONObject;
4 import org.json.simple.parser.JSONParser;
5 import org.json.simple.parser.ParseException;
6
7 import java.io.*;
8 import java.sql.Timestamp;
9 import java.util.ArrayList;
10 import java.util.List;
11 import java.util.concurrent.atomic.AtomicBoolean;
12 import java.util.stream.Collectors;
13
14 @SuppressWarnings("unchecked")
15 public class DB {
16     private static final Logger LOGGER = Logger.getLogger(DB.class);
17     private static final Settings SETTINGS = Settings.from("../database.properties");
18
19     private static final int MAX_RADIOS_PER_CLIENT = SETTINGS.get("MAX_RADIOS_PER_CLIENT", 3);
20
21     private static final String WORKING_DIR = SETTINGS.get("WORKING_DIR", "../database/");
22     private static final String USERS_DB = SETTINGS.get("USER_DB", "user");
23     ;
24     private static final String STATIONS_DB = SETTINGS.get("STATION_DB", "station");
25     private static final String CONNECTIONS_DB = SETTINGS.get("CONNECTION_DB", "connection");
26
27     private static final int OFFSET_TIMESTAMP = SETTINGS.get("OFFSET_TIMESTAMP", 2);
28
29     DB() throws IOException {
30         File workingDir = new File(WORKING_DIR);
31         if (!workingDir.exists()) {
32             if (!workingDir.mkdir()) {
33                 if (!workingDir.exists()) {
34                     LOGGER.fatal("Cannot create working dir");
35                     throw new IOException("Cannot create working dir");
36                 }
37             }
38             LOGGER.debug("Created working dir for DB: " + WORKING_DIR);
39         }
40         String[] files = {STATIONS_DB, CONNECTIONS_DB, USERS_DB};
41         for (String file : files) {
42             String path = WORKING_DIR + file;
43             File f = new File(path);
44             if (!f.exists()) {
45                 if (!f.createNewFile()) {
46                     if (!f.exists()) {
47                         LOGGER.fatal("Cannot create file " + path);
48                         throw new IOException("Cannot create file " + path);
49                     }
50                     LOGGER.info(String.format("Created file DB: \"%s\"", path));
51                 }
52             }
53         }
54
55         public static void cleanDatabases() {
56             try {
57                 DB db = new DB();
58                 String[] DBNames = {STATIONS_DB, USERS_DB, CONNECTIONS_DB};
59                 for (String DBName: DBNames) {
60                     if (db.writeJSON(new JSONObject(), DBName)) {

```


abr 19, 18 10:18

DB.java

Page 2/5

```

61     LOGGER.info(DBName + " clean");
62     } else {
63         LOGGER.warn("Cannot clean " + DBName);
64     }
65 }
66 } catch (IOException e) {
67     LOGGER.warn("Cannot clean databases");
68 }
69 }
70 }
71
72 private synchronized boolean writeJSON(JSONObject json, String fileName) {
73     try (FileWriter file = new FileWriter(WORKING_DIR + fileName)) {
74         file.write(json.toJSONString());
75         file.close();
76         return true;
77     } catch (IOException e) {
78         LOGGER.warn("Cannot write " + WORKING_DIR + fileName);
79         LOGGER.debug(e);
80     }
81     return false;
82 }
83
84 private synchronized JSONObject readJSON(String fileName) {
85     JSONParser parser = new JSONParser();
86
87     try {
88         Object obj = parser.parse(new FileReader(WORKING_DIR + fileName));
89         return (JSONObject) obj;
90     } catch (IOException | ParseException e) {
91         LOGGER.warn("Cannot read " + WORKING_DIR + fileName);
92         LOGGER.debug(e);
93     }
94     return null;
95 }
96
97 public void addUserInRadio(String userName, String userQueue, String radio)
98 {
99     JSONObject stations = readJSON(STATIONS_DB);
100
101     if (stations == null) {
102         return;
103     }
104     JSONArray userQueues = stations.containsKey(radio) ? (JSONArray) station
105 s.get(radio) : new JSONArray();
106     userQueues.add(userQueue);
107
108     stations.put(radio, userQueues);
109
110     if (writeJSON(stations, STATIONS_DB)) {
111         JSONObject users = readJSON(USERS_DB);
112         if (users == null) {
113             return;
114         }
115         Long count = users.containsKey(userName) ? (Long) users.get(userName)
116 : 0;
117         users.put(userName, count + 1);
118         if (writeJSON(users, USERS_DB)) {
119             LOGGER.info("Added user " + userName + " to radio " + radio);
120         }
121     }
122
123     public void deleteUserFromRadio(String userName, String userQueue, String ra
124 dio) {

```

abr 19, 18 10:18

DB.java

Page 3/5

```

123     JSONObject stations = readJSON(STATIONS_DB);
124
125     if (stations == null || !stations.containsKey(radio)) {
126         return;
127     }
128
129     JSONArray userQueues = (JSONArray) stations.get(radio);
130     JSONArray userQueuesNew = new JSONArray();
131     for (Object queue : userQueues) {
132         if (queue instanceof String) {
133             if (!((String) queue).equalsIgnoreCase(userQueue)) {
134                 userQueuesNew.add(queue);
135             }
136         }
137     }
138     stations.put(radio, userQueuesNew);
139
140     if (writeJSON(stations, STATIONS_DB)) {
141         JSONObject users = readJSON(USERS_DB);
142         if (users == null) {
143             return;
144         }
145         Long count = users.containsKey(userName) ? (Long) users.get(userName)
146 : 1;
147         users.put(userName, count - 1);
148         if (writeJSON(users, USERS_DB)) {
149             LOGGER.info("Delete user " + userName + " in radio " + radio);
150         }
151     }
152
153     public boolean existStation(String radio) {
154         JSONObject stations = readJSON(STATIONS_DB);
155         if (stations == null) {
156             return false;
157         }
158         if (!stations.containsKey(radio)) {
159             return false;
160         }
161         return true;
162     }
163
164     public boolean userCanHearRadio(String userName, String radio) {
165
166         JSONObject users = readJSON(USERS_DB);
167         if (users == null) {
168             return false;
169         }
170         if (users.containsKey(userName)) {
171             Long radioCount = (Long) users.get(userName);
172             return radioCount < MAX_RADIOS_PER_CLIENT;
173         }
174         return true;
175     }
176
177
178     public void updateUserActivity(String userName) {
179         JSONObject usersActivity = readJSON(CONNECTIONS_DB);
180         if (usersActivity == null) {
181             return;
182         }
183         Timestamp timestamp = new Timestamp(System.currentTimeMillis());
184         if (usersActivity.containsKey(userName)) {
185             JSONObject userActivity = (JSONObject) usersActivity.get(userName);
186             Long lastTimeStamp = (Long) userActivity.get("last");
187

```

abr 19, 18 10:18

DB.java

Page 4/5

```

188         userActivity.put("last", timestamp);
189         if (timestamp.getTime() - lastTimeStamp > OFFSET_TIMESTAMP) {
190             userActivity.put("total", (Long)userActivity.get("total") + 1);
191         } else {
192             userActivity.put("total", (Long)userActivity.get("total") + (timestamp.getTime() - lastTimeStamp));
193         }
194         usersActivity.put(userName, userActivity);
195     } else {
196         JSONObject userActivity = new JSONObject();
197         userActivity.put("last", timestamp.getTime());
198         userActivity.put("total", 1);
199         usersActivity.put(userName, userActivity);
200     }
201     if (writeJSON(usersActivity, CONNECTIONS_DB)) {
202         LOGGER.info("Update activity for user " + userName);
203     }
204 }
205
206 public List<String> getStations() {
207     ArrayList<String> stationsArray = new ArrayList<>();
208     JSONObject stations = readJSON(STATIONS_DB);
209     if (stations == null) {
210         return stationsArray;
211     }
212     stationsArray.addAll(stations.keySet());
213     return stationsArray;
214 }
215
216 public List<String> getTopUsers(int count) {
217     JSONObject users = readJSON(CONNECTIONS_DB);
218     if (users == null) {
219         return new ArrayList<>();
220     }
221     return (List<String>) users.keySet().stream()
222         .sorted((u1,u2) -> {
223             JSONObject user1 = (JSONObject) users.get(u1);
224             JSONObject user2 = (JSONObject) users.get(u2);
225             Long total1 = (Long) user1.get("total");
226             Long total2 = (Long) user2.get("total");
227             return -total1.compareTo(total2);
228         })
229         .limit(count)
230         .map(userName -> userName + "|total: " + ((JSONObject) users.get(userName)).get("total") + " sec.")
231         .collect(Collectors.toList());
232 }
233
234 public List<String> getUsersInStation(String station) {
235     List<String> userQueue = new ArrayList<>();
236     JSONObject stations = readJSON(STATIONS_DB);
237     if (stations == null || !stations.containsKey(station)) {
238         return userQueue;
239     }
240     JSONArray usersQueue = (JSONArray) stations.get(station);
241     for (Object queue : usersQueue) {
242         userQueue.add((String) queue);
243     }
244     return userQueue;
245 }
246
247 public void addStation(String station) {
248     JSONObject stations = readJSON(STATIONS_DB);
249     if (stations == null || stations.containsKey(station)) {
250         return;
251     }

```

abr 19, 18 10:18

DB.java

Page 5/5

```

252     }
253     stations.put(station, new JSONArray());
254     if (writeJSON(stations, STATIONS_DB)) {
255         LOGGER.info("Added station " + station + " in DB");
256     }
257 }
258
259 public void deleteStation(String station) {
260     JSONObject stations = readJSON(STATIONS_DB);
261     if (stations == null || !stations.containsKey(station)) {
262         return;
263     }
264     stations.remove(station);
265     if (writeJSON(stations, STATIONS_DB)) {
266         LOGGER.info("Deleted station " + station + " in DB");
267     }
268 }
269
270 public List<String> getCountUserPerStation() {
271     return new ArrayList<>();
272 }
273 }

```

abr 19, 18 18:34

DataBase.java

Page 1/3

```

1  import org.apache.log4j.Logger;
2  import org.json.simple.JSONArray;
3  import org.json.simple.JSONObject;
4  import org.json.simple.parser.JSONParser;
5  import org.json.simple.parser.ParseException;
6
7  import java.io.*;
8  import java.sql.Timestamp;
9  import java.util.*;
10 import java.util.stream.Collectors;
11
12 @SuppressWarnings("unchecked")
13 public class DataBase {
14     private static final Logger LOGGER = Logger.getLogger(DataBase.class);
15     private static final Settings SETTINGS = Settings.from("../database.properties");
16
17     private static final int MAX_RADIOS_PER_CLIENT = SETTINGS.get("MAX_RADIOS_P
ER_CLIENT", 3);
18
19     private static final String WORKING_DIR = SETTINGS.get("WORKING_DIR", "..
/database/");
20     private static final String STATIONS_DIR = SETTINGS.get("STATION_DIR", "
stations/");
21     private static final String USERS_DB = SETTINGS.get("USER_DB", "user")
;
22     private static final String CONNECTIONS_DB = SETTINGS.get("CONNECTION_DB",
"connection");
23
24     private static final int OFFSET_TIMESTAMP = SETTINGS.get("OFFSET_TIMESTAM
P", 2);
25
26     private static final int BLOCK_SIZE = SETTINGS.get("BLOCK_SIZE", 100
);
27
28     private HashMap<String, FileCellBlock> DB;
29
30     DataBase() throws IOException {
31         createDir(WORKING_DIR);
32         createDir(WORKING_DIR + STATIONS_DIR);
33         String[] files = {CONNECTIONS_DB, USERS_DB};
34         DB = new HashMap<>();
35         for (String file : files) {
36             String path = WORKING_DIR + file;
37             DB.put(file, new FileCellBlock(path, BLOCK_SIZE));
38         }
39     }
40
41     private void createDir(String path) throws IOException {
42         File workingDir = new File(path);
43         if (!workingDir.exists()) {
44             if (!workingDir.mkdir()) {
45                 if (!workingDir.exists()) {
46                     LOGGER.fatal("Cannot create working dir");
47                     throw new IOException("Cannot create working dir");
48                 }
49             }
50             LOGGER.debug("Created working dir for DB: " + WORKING_DIR);
51         }
52     }
53
54     public static void cleanDatabases() {
55         File workingDir = new File(WORKING_DIR);
56         for (File file: Objects.requireNonNull(workingDir.listFiles())) {
57             if (!file.isDirectory())
58                 if (!file.delete()) {
59                     LOGGER.warn("Cannot delete " + file.toString());

```

abr 19, 18 18:34

DataBase.java

Page 2/3

```

60     }
61
62     }
63
64     /*
65     private synchronized boolean writeJSON(JSONObject json, String fileName) {
66         try (FileWriter file = new FileWriter(WORKING_DIR + fileName)) {
67             file.write(json.toJSONString());
68             file.close();
69             return true;
70         } catch (IOException e) {
71             LOGGER.warn("Cannot write " + WORKING_DIR + fileName);
72             LOGGER.debug(e);
73         }
74         return false;
75     }
76
77     private synchronized JSONObject readJSON(String fileName) {
78         JSONParser parser = new JSONParser();
79
80         try {
81             Object obj = parser.parse(new FileReader(WORKING_DIR + fileName));
82             return (JSONObject) obj;
83         } catch (IOException | ParseException e) {
84             LOGGER.warn("Cannot read " + WORKING_DIR + fileName);
85             LOGGER.debug(e);
86         }
87         return null;
88     }
89     */
90
91     private String stationKey(String radio) {
92         return "station." + radio;
93     }
94
95     private FileCellBlock getStationDB(String name) throws IOException {
96         String key = stationKey(name);
97         String newFilePath = WORKING_DIR + STATIONS_DIR + key;
98         return DB.getDefault(key, new FileCellBlock(newFilePath, BLOCK_SIZE));
99     }
100
101     public void addUserInRadio(String userName, String userQueue, String radio)
102     {
103         // TODO: Add in getStationDB(radio), and update (+1) counter in USERS_DB
104     }
105
106     public void deleteUserFromRadio(String userName, String userQueue, String ra
dio) {
107         // TODO: Delete from getStationDB(radio), and update counter (-1) in USE
RS_DB
108     }
109
110     public boolean existStation(String radio) {
111         return DB.containsKey( stationKey(radio) );
112     }
113
114     public boolean userCanHearRadio(String userName, String radio) {
115
116         FileCellBlock users = DB.get(USERS_DB);
117
118         List<String> result = users.find(s -> s.contains(userName));
119         if (result.isEmpty()) {
120             return true;
121         }
122         // TODO: Number of radios is bigger than result.get(0);

```

abr 19, 18 18:34

DataBase.java

Page 3/3

```

123     return true;
124 }
125 }
126
127 public void updateUserActivity(String userName) {
128     FileCellBlock connection = DB.get(CONNECTIONS_DB);
129
130     // TODO: Pass function who take value and modify it;
131     // Take row. if match modify. and return true. else return false
132     // user.update();
133 }
134
135 public List<String> getStations() {
136     List<String> stations = new ArrayList<>();
137     DB.entrySet().stream()
138         .filter(e → e.getKey().matches("station"))
139         .sorted(Comparator.comparing(Map.Entry::getKey))
140         .forEach(e → stations.add(e.getKey()));
141     return stations;
142 }
143
144 public List<String> getTopUsers(int count) {
145     return new ArrayList<>();
146 }
147
148 public void addStation(String station) {
149     try {
150         getStationDB(station);
151     } catch (IOException e) {
152         LOGGER.warn("Cannot add station " + station);
153         LOGGER.debug(e);
154     }
155 }
156
157 public void deleteStation(String station) {
158 }
159
160
161 public List<String> getCountUserPerStation() {
162     return new ArrayList<>();
163 }
164 }

```

abr 18, 18 12:15

MessageType.java

Page 1/1

```

1 package Message;
2
3 public enum MessageType {
4     REQUEST_RADIOS(0),
5     RESPONSE_RADIOS(1),
6     REQUEST_CONNECTION(2),
7     CONNECTION_ACCEPTED(3),
8     CONNECTION_DENIED(4),
9     RADIO_PACKAGE(5),
10    KEEP_ALIVE(6),
11    END_TRANSMISSION(7),
12    END_CONNECTION(8),
13    ADMIN_REQUEST_STATS(9),
14    ADMIN_RESPONSE_STATS(10),
15    INVALID(-1); //Default MessageType
16
17
18    public static MessageType from(int x) {
19        MessageType[] values = MessageType.values();
20        if (x ≥ values.length) {
21            return INVALID;
22        }
23        return values[x];
24    }
25
26    private final int value;
27    MessageType(int value) {
28        this.value = value;
29    }
30
31    public int getValue() {
32        return value;
33    }
34 }

```

abr 16, 18 16:19

Message.java

Page 1/2

```

1 package Message;
2
3 import org.json.simple.JSONObject;
4 import org.json.simple.parser.JSONParser;
5 import org.json.simple.parser.ParseException;
6
7 import java.util.Base64;
8
9 import static java.nio.charset.StandardCharsets.UTF_8;
10
11 public class Message {
12     private static final JSONParser PARSER = new JSONParser();
13
14     private String raw;
15     private JSONObject json;
16
17     public Message(String rawJSON) throws MessageException {
18         try {
19             json = (JSONObject) PARSER.parse(rawJSON);
20             raw = rawJSON;
21         } catch (ParseException e) {
22             throw new MessageException("Invalid JSON string", e);
23         }
24     }
25
26     public Message(byte[] bytes) throws MessageException {
27         try {
28             raw = new String(bytes, UTF_8);
29             json = (JSONObject) PARSER.parse(raw);
30         } catch (ParseException e) {
31             throw new MessageException("Invalid Byte data", e);
32         }
33     }
34
35     public Message(JSONObject json) {
36         this.json = json;
37         this.raw = json.toJSONString();
38     }
39
40
41     public MessageType getType() {
42         Long type = (Long) json.get("type");
43         return MessageType.from(type.intValue());
44     }
45
46     public byte[] getPayload() {
47         String encoded = json.get("payload").toString();
48         return Base64.getDecoder().decode(encoded);
49     }
50
51     public String getStringPayload() {
52         return new String(getPayload(), UTF_8);
53     }
54
55     public String getContentType() {
56         return json.get("content_type").toString();
57     }
58
59     public byte[] toBytes() {
60         return raw.getBytes();
61     }
62
63     public String toString() {
64         return raw;
65     }
66

```

abr 16, 18 16:19

Message.java

Page 2/2

```

67
68     public String getRadio() {
69         return json.get("radio").toString();
70     }
71
72     public String getError() {
73         return json.get("error").toString();
74     }
75
76     public String getUserQueue() {
77         return json.get("user_queue").toString();
78     }
79
80     public String getUser() {
81         return json.get("user").toString();
82     }
83
84     public String getInfo() {
85         return json.get("info").toString();
86     }
87 }
88

```

abr 12, 18 14:15

MessageException.java

Page 1/1

```

1 package Message;
2
3 import java.io.IOException;
4
5 public class MessageException extends IOException {
6     public MessageException() { super(); }
7     public MessageException(String message) { super(message); }
8     public MessageException(String message, Throwable cause) { super(message, ca
9 use); }
10    public MessageException(Throwable cause) { super(cause); }
11 }

```

abr 16, 18 16:19

MessageBuilder.java

Page 1/1

```

1 package Message;
2
3 import org.json.simple.JSONObject;
4
5 import java.util.Base64;
6
7
8 @SuppressWarnings("unchecked")
9 public class MessageBuilder {
10     private final JSONObject messageData;
11
12     public MessageBuilder() {
13         messageData = new JSONObject();
14     }
15
16     public MessageBuilder setType(MessageType type) {
17         messageData.put("type", type.getValue());
18         return this;
19     }
20
21     public MessageBuilder setPayload(String payload) {
22         messageData.put("payload", payload);
23         return this;
24     }
25
26     public MessageBuilder setPayload(byte[] bytes) {
27         return setPayload(Base64.getEncoder().encodeToString(bytes));
28     }
29
30
31     public MessageBuilder setClientQueue(String clientQueue) {
32         messageData.put("user_queue", clientQueue);
33         return this;
34     }
35
36     public MessageBuilder setRadio(String radio) {
37         messageData.put("radio", radio);
38         return this;
39     }
40
41     public MessageBuilder setUser(String user) {
42         messageData.put("user", user);
43         return this;
44     }
45
46     public MessageBuilder setContentType(String contentType) {
47         messageData.put("content_type", contentType);
48         return this;
49     }
50
51     public MessageBuilder setError(String error) {
52         messageData.put("error", error);
53         return this;
54     }
55
56     public MessageBuilder setInfo(String info) {
57         messageData.put("info", info);
58         return this;
59     }
60
61     public Message build() {
62         return new Message(messageData);
63     }
64 }

```

abr 18, 18 19:18

CommunicationWrapper.java

Page 1/3

```

1  import Message.*;
2  import com.rabbitmq.client.*;
3  import org.apache.log4j.Logger;
4
5  import java.io.IOException;
6  import java.util.concurrent.TimeoutException;
7  import java.util.function.Consumer;
8
9  public class CommunicationWrapper {
10     private static final Logger LOGGER = Logger.getLogger(CommunicationWrapper.class);
11
12     static CommunicationWrapper getConnection(String host, int port) {
13         ConnectionFactory factory = new ConnectionFactory();
14         factory.setHost(host);
15         factory.setPort(port);
16         Connection connection = null;
17         Channel channel = null;
18         try {
19             connection = factory.newConnection();
20             channel = connection.createChannel();
21             return new CommunicationWrapper(channel);
22         } catch (IOException | TimeoutException e) {
23             LOGGER.error("Error: " + e.getMessage());
24         }
25         return null;
26     }
27
28     private final Channel channel;
29
30     CommunicationWrapper(Channel channel) {
31         this.channel = channel;
32     }
33
34     void close() {
35         try {
36             Connection connection = channel.getConnection();
37             channel.close();
38             connection.close();
39         } catch (IOException | TimeoutException e) {
40             LOGGER.warn("Cannot close connection " + e.getMessage());
41         }
42     }
43
44     boolean queueDeclare(String name) {
45         try {
46             AMQP.Queue.DeclareOk result = channel.queueDeclare(name, true, false, false, null);
47             LOGGER.info("Created queue " + result.getQueue());
48             return true;
49         } catch (IOException e) {
50             LOGGER.warn("Cannot declare queue " + name + "." + e.getMessage());
51             return false;
52         }
53     }
54
55     public String queueDeclare() {
56         try {
57             AMQP.Queue.DeclareOk result = channel.queueDeclare();
58             LOGGER.info("Created queue " + result.getQueue());
59             return result.getQueue();
60         } catch (IOException e) {
61             LOGGER.warn("Cannot declare queue. " + e.getMessage());
62             return null;
63         }
64     }

```

abr 18, 18 19:18

CommunicationWrapper.java

Page 2/3

```

65
66     private boolean put(String queue, Message message, AMQP.BasicProperties props) {
67         try {
68             channel.basicPublish("", queue, null, message.toBytes());
69             LOGGER.debug("Send message [" + message.toString().hashCode() + "] in queue " + queue);
70         } catch (IOException e) {
71             LOGGER.warn("Cannot put message in " + queue + "." + e.getMessage());
72             return false;
73         }
74         return true;
75     }
76
77     boolean put(String queue, Message message, int expiration_seconds) {
78         AMQP.BasicProperties props = new AMQP.BasicProperties.Builder()
79             .expiration(String.valueOf(expiration_seconds * 1000))
80             .build();
81         return put(queue, message, props);
82     }
83
84     boolean put(String queue, Message message) {
85         return put(queue, message, Integer.MAX_VALUE);
86     }
87
88     String append(String queue, Consumer<Message> handlerFunction) {
89         try {
90             return channel.basicConsume(queue, false, new DefaultConsumer(channel) {
91                 @Override
92                 public void handleDelivery(String consumerTag, Envelope env, AMQP.BasicProperties props, byte[] body) {
93                     try {
94                         Message message = new Message(body);
95                         LOGGER.debug("Receive message [" + message.toString().hashCode() + "] from queue " + queue);
96                         handlerFunction.accept(message);
97                         channel.basicAck(env.getDeliveryTag(), false);
98                     } catch (IOException e) {
99                         LOGGER.debug(e);
100                     }
101                 }
102             });
103         } catch (IOException e) {
104             LOGGER.error("Error on append in " + queue);
105             LOGGER.debug(e);
106             return null;
107         }
108     }
109
110     boolean detach(String consumerTag) {
111         try {
112             channel.basicCancel(consumerTag);
113             return true;
114         } catch (IOException e) {
115             LOGGER.warn("Cannot detach consumerTag: " + consumerTag);
116             LOGGER.debug(e);
117             return false;
118         }
119     }
120
121     void deleteQueue(String queueName) {
122         try {
123             channel.queueDeleteNoWait(queueName, false, false);
124             LOGGER.info("Deleted queue " + queueName);
125         } catch (IOException e) {

```

abr 18, 18 19:18

CommunicationWrapper.java

Page 3/3

```

126     LOGGER.warn("Cannot delete queue: " + queueName);
127     LOGGER.debug(e);
128 }
129 }
130
131
132
133 }
```

abr 18, 18 19:14

Broadcast.java

Page 1/2

```

1  import Message.*;
2  import org.apache.log4j.Logger;
3  import sun.misc.Signal;
4
5  import java.util.concurrent.TimeUnit;
6
7  public class Broadcast {
8      private static final Logger LOGGER = Logger.getLogger(Broadcast.class);
9      private static final Settings SETTINGS = Settings.from("admin.properties");
10
11     private static final String RABBITMQ_HOST = SETTINGS.get("RABBITMQ_HOST",
12 "localhost");
13     private static final int RABBITMQ_PORT = SETTINGS.get("RABBITMQ_PORT",
14 5672);
15     private static final String RADIO_QUEUE = SETTINGS.get("RADIO_QUEUE", "RADIO");
16     private static final int MESSAGE_EXPIRATION_SECONDS = SETTINGS.get("MESSAGE_EXPIRATION_SECONDS", 30);
17
18     private String consumerRadioTag;
19
20     private final CommunicationWrapper communication;
21
22     private DB db;
23
24     Broadcast() throws Exception {
25         communication = CommunicationWrapper.getConnection(RABBITMQ_HOST, RABBITMQ_PORT);
26         if (communication == null) {
27             LOGGER.fatal("Cannot open communication");
28             throw new Exception("Cannot open communication");
29         }
30         if (!communication.queueDeclare(RADIO_QUEUE)) {
31             LOGGER.fatal("Cannot declare queue " + RADIO_QUEUE);
32             communication.close();
33             throw new Exception("Cannot declare queue " + RADIO_QUEUE);
34         }
35         db = new DB();
36     }
37
38     private void registerSIGINT() {
39         Signal.handle(new Signal("INT"), sig → {
40             LOGGER.info("SIGINT detected. Closing Broadcast");
41             communication.detach(consumerRadioTag);
42             communication.close();
43             LOGGER.info("Broadcast closed");
44         });
45     }
46
47     void start() {
48         LOGGER.info("Waiting Radio message");
49
50         consumerRadioTag = communication.append(RADIO_QUEUE, message → {
51             if (message.getType() == MessageType.RADIO_PACKAGE) {
52                 db.addStation(message.getRadio());
53                 for (String userQueue : db.getUsersInStation(message.getRadio())) {
54                     communication.put(userQueue, message, MESSAGE_EXPIRATION_SECONDS);
55                 }
56             } else if (message.getType() == MessageType.END_TRANSMISSION) {
57                 Message messageEnd = new MessageBuilder()
58                     .setType(MessageType.END_CONNECTION)
59 
```


abr 18, 18 19:14

Broadcast.java

Page 2/2

```

60         .build();
61         for (String userQueue : db.getUsersInStation(message.getRadio()))
62         ) {
63             communication.put(userQueue, messageEnd, MESSAGE_EXPIRATION_
64             SECONDS);
65         }
66         db.deleteStation(message.getRadio());
67         } else {
68             LOGGER.warn("Unhandled message with type: " + message.getType());
69         }
70     }
71     registerSIGINT();
72 }
73
74 public static void main(String[] argv) {
75     Broadcast broadcast = null;
76     try {
77         broadcast = new Broadcast();
78     } catch (Exception e) {
79         LOGGER.info("Cannot start broadcast");
80         LOGGER.debug(e);
81         System.exit(1);
82     }
83     broadcast.start();
84 }
85 }

```

abr 18, 18 19:14

Admin.java

Page 1/2

```

1  import Message.*;
2  import org.apache.log4j.Logger;
3  import sun.misc.Signal;
4
5  import java.io.IOException;
6  import java.util.concurrent.Executors;
7  import java.util.concurrent.ScheduledExecutorService;
8  import java.util.concurrent.TimeUnit;
9
10 public class Admin {
11     private static final Logger LOGGER = Logger.getLogger(Admin.class);
12     private static final Settings SETTINGS = Settings.from("admin.properties");
13
14     private static final String RABBITMQ_HOST = SETTINGS.get("RABBITMQ_H
15     OST", "localhost");
16     private static final int RABBITMQ_PORT = SETTINGS.get("RABBITMQ_PO
17     RT", 5672);
18     private static final String ADMIN_REQ_QUEUE = SETTINGS.get("ADMIN_REQU
19     EST_QUEUE", "ADMIN_REQUEST");
20     private static final String ADMIN_RES_QUEUE = SETTINGS.get("ADMIN_RESPO
21     NSE_QUEUE", "ADMIN_RESPONSE");
22     private static final int REQUEST_POLL_SECONDS = SETTINGS.get("REQUEST_POL
23     L_SECONDS", 10);
24     private static final int POOL_SIZE = SETTINGS.get("POOL_SIZE", 5
25     );
26
27     private CommunicationWrapper communication;
28
29     private Admin() throws IOException {
30         communication = CommunicationWrapper.getConnection(RABBITMQ_HOST, RABBITM
31         Q_PORT);
32         if (communication == null) {
33             LOGGER.error("Cannot get connection");
34             throw new IOException("Cannot connect");
35         }
36     }
37
38     private ScheduledExecutorService startScheduledRequests() {
39         ScheduledExecutorService schedule = Executors.newScheduledThreadPool(POOL
40         L_SIZE);
41         schedule.scheduleAtFixedRate(() -> {
42             Message statsRequest = new MessageBuilder()
43                 .setType(MessageType.ADMIN_REQUEST_STATS)
44                 .build();
45             communication.put(ADMIN_REQ_QUEUE, statsRequest, REQUEST_POLL_SECON
46             D);
47         }, REQUEST_POLL_SECONDS, REQUEST_POLL_SECONDS, TimeUnit.SECONDS);
48         return schedule;
49     }
50
51     private String startResponseListener() {
52         return communication.append(ADMIN_RES_QUEUE, res -> {
53             LOGGER.info("Receive message from " + ADMIN_RES_QUEUE);
54             if (res.getType() == MessageType.ADMIN_RESPONSE_STATS) {
55                 LOGGER.info("Message:\n" + res.getInfo());
56                 System.out.println(res.getInfo());
57             } else {
58                 LOGGER.warn("Unhandled message type");
59             }
60         });
61     }
62
63     private void start() throws InterruptedException {
64         LOGGER.info("Init admin-client");
65     }
66 }

```

abr 18, 18 19:14

Admin.java

Page 2/2

```

58     LOGGER.info("Starting admin-scheduler collector");
59
60     ScheduledExecutorService schedule = startScheduledRequests();
61
62     String consumerTag = startResponseListener();
63
64     Signal.handle(new Signal("INT"), sig → {
65         LOGGER.info("SIGINT detected. Closing Admin-Handler");
66         schedule.shutdownNow();
67         LOGGER.info("Admin-Handler closed");
68     });
69
70
71     schedule.shutdown();
72     schedule.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);
73
74     communication.close();
75     communication.detach(consumerTag);
76
77     LOGGER.info("Admin-client closed");
78 }
79
80 public static void main(String[] strings) {
81     try {
82         Admin admin = new Admin();
83         admin.start();
84     } catch (IOException | InterruptedException e) {
85         LOGGER.fatal(e);
86         LOGGER.warn("Error. Closed admin");
87     }
88 }
89 }

```

abr 18, 18 19:14

AdminHandler.java

Page 1/2

```

1  import Message.*;
2  import org.apache.log4j.Logger;
3  import sun.misc.Signal;
4
5  import java.util.List;
6  import java.util.concurrent.Executors;
7  import java.util.concurrent.ScheduledExecutorService;
8  import java.util.concurrent.TimeUnit;
9
10 public class AdminHandler {
11     private static final Logger LOGGER = Logger.getLogger(AdminHandler.class);
12     private static final Settings SETTINGS = Settings.from("admin-handler.properties");
13
14
15     private static final String RABBITMQ_HOST = SETTINGS.get("RABBITMQ_H
16     OST", "localhost");
17     private static final int RABBITMQ_PORT = SETTINGS.get("RABBITMQ_PO
18     RT", 5672);
19     private static final String ADMIN_REQ_QUEUE = SETTINGS.get("ADMIN_REQU
20     EST_QUEUE", "ADMIN_REQUEST");
21     private static final String ADMIN_RES_QUEUE = SETTINGS.get("ADMIN_RESPO
22     NSE_QUEUE", "ADMIN_RESPONSE");
23     private static final int COUNT_TOP_USERS = SETTINGS.get("COUNT_TOP_U
24     SERS", 10);
25
26     private final CommunicationWrapper communication;
27
28     private DB db;
29
30     AdminHandler() throws Exception {
31         communication = CommunicationWrapper.getConnection(RABBITMQ_HOST, RABBITM
32         Q_PORT);
33         if (communication == null) {
34             LOGGER.fatal("Cannot open communication");
35             throw new Exception("Cannot open communication");
36         }
37
38         db = new DB();
39
40     private String generatePrintableStats() {
41         List<String> topUsers = db.getTopUsers(COUNT_TOP_USERS);
42         List<String> usersPerStations = db.getCountUserPerStation();
43         StringBuilder stats = new StringBuilder("Number of users per station");
44         for (String userCount : usersPerStations) {
45             stats.append("\n\t-").append(userCount);
46         }
47         stats.append("\n\nTop users (minutes)");
48         for (String user : topUsers) {
49             stats.append("\n\t-").append(user);
50         }
51         return stats.toString();
52     }
53
54     void start() {
55         LOGGER.info("Starting admin-scheduler collector");
56
57         communication.append(ADMIN_REQ_QUEUE, req → {
58             Message stats = new MessageBuilder()
59                 .setType(MessageType.ADMIN_RESPONSE_STATS)
60                 .setInfo(generatePrintableStats())
61                 .build();
62             communication.put(ADMIN_RES_QUEUE, stats);
63         });
64     }
65 }

```

abr 18, 18 19:14

AdminHandler.java

Page 2/2

```

60
61
62     Signal.handle(new Signal("INT"), sig → {
63         LOGGER.info("SIGINT detected. Closing Admin-Handler");
64         LOGGER.info("Admin-Handler closed");
65     }
66 );
67
68     communication.close();
69 }
70
71 public static void main(String[] argv) {
72     try {
73         AdminHandler adminHandler = new AdminHandler();
74         adminHandler.start();
75     } catch (Exception e) {
76         LOGGER.info("Cannot start Admin Handler");
77         LOGGER.debug(e);
78     }
79 }
80 }

```

abr 19, 18 18:36

Table of Content

Page 1/1

1	Table of Contents					
2	1 <i>UserGestor.java</i>	sheets	1 to	1 (1) pages	1- 2	112 lines
3	2 <i>Station.java</i>	sheets	2 to	2 (1) pages	3- 4	101 lines
4	3 <i>Settings.java</i>	sheets	3 to	3 (1) pages	5- 5	52 lines
5	4 <i>RadioListener.java</i> ..	sheets	3 to	5 (3) pages	6- 9	250 lines
6	5 <i>Initializer.java</i>	sheets	5 to	5 (1) pages	10- 10	30 lines
7	6 <i>Main.java</i>	sheets	6 to	6 (1) pages	11- 11	66 lines
8	7 <i>Destructor.java</i>	sheets	6 to	6 (1) pages	12- 12	34 lines
9	8 <i>FileCellBlock.java</i> ..	sheets	7 to	8 (2) pages	13- 15	196 lines
10	9 <i>DB.java</i>	sheets	8 to	10 (3) pages	16- 20	274 lines
11	10 <i>DataBase.java</i>	sheets	11 to	12 (2) pages	21- 23	165 lines
12	11 <i>MessageType.java</i>	sheets	12 to	12 (1) pages	24- 24	35 lines
13	12 <i>Message.java</i>	sheets	13 to	13 (1) pages	25- 26	89 lines
14	13 <i>MessageException.java</i>	sheets	14 to	14 (1) pages	27- 27	11 lines
15	14 <i>MessageBuilder.java</i> .	sheets	14 to	14 (1) pages	28- 28	65 lines
16	15 <i>CommunicationWrapper.java</i>	sheets	15 to	16 (2) pages	29- 31	134 lines
17	16 <i>Broadcast.java</i>	sheets	16 to	17 (2) pages	32- 33	86 lines
18	17 <i>Admin.java</i>	sheets	17 to	18 (2) pages	34- 35	90 lines
19	18 <i>AdminHandler.java</i> ...	sheets	18 to	19 (2) pages	36- 37	81 lines