```java
import Message.*;
import org.apache.log4j.Logger;
import sun.misc.Signal;

import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

public class UserGestor {
    private static final Logger LOGGER = Logger.getLogger(UserGestor.class);

    private static final Logger CONNECTION_LOGGER = Logger.getLogger("connection");

    private static final Settings SETTINGS = Settings.from("user-gestor.properties");

    private static final String RABBITMQ_HOST   = SETTINGS.get("RABBITMQ_HOST", "localhost");
    private static final int RABBITMQ_PORT      = SETTINGS.get("RABBITMQ_PORT", 5672);
    private static final String CLIENT_QUEUE = SETTINGS.get("CLIENT_QUEUE","CLIENT");

    private String consumerClientTag;

    private final CommunicationWrapper communication;

    private DB db;

    UserGestor() throws Exception {
        communication = CommunicationWrapper.getConnection(RABBITMQ_HOST,RABBITMQ_PORT);
        if (communication ≡ null) {
            LOGGER.fatal("Cannot open communication");
            throw new Exception("Cannot open communication");
        }

        if (¬communication.queueDeclare(CLIENT_QUEUE)) {
            LOGGER.fatal("Cannot declare queue " + CLIENT_QUEUE);
            communication.close();
            throw new Exception("Cannot declare queue " + CLIENT_QUEUE);
        }

        db = new BlockDatabase();
    }

    private void registerSIGINT() throws InterruptedException {
        Semaphore semaphore = new Semaphore(0);
        Runtime.getRuntime().addShutdownHook(new Thread(() → {
                    LOGGER.info("SIGINT detected. Closing connection");
                    communication.detach(consumerClientTag);
                    communication.close();
                    LOGGER.info("Connection closed");
                    semaphore.release();
                })
        );
        semaphore.acquire();
    }

    private Message handlerRequestConnection(Message request) {
        LOGGER.info("Request connection from " + request.getUser() + " in radio " + request.getRadio());
        MessageBuilder messageBuilder = new MessageBuilder();
        if (db.existStation(request.getRadio())) {
            if (db.userCanHearRadio(request.getUser(), request.getRadio())) {
                db.addUserInRadio(request.getUser(), request.getUserQueue(), request.getRadio());
```



```java
                LOGGER.info("Accepted connection from " + request.getUser() + " to radio " + request.getRadio());
                CONNECTION_LOGGER.info(String.format("[CONNECTION] user '%s' to radio '%s'",request.getUser(),request.getRadio()));
                messageBuilder.setType(MessageType.CONNECTION_ACCEPTED);
            } else {
                messageBuilder
                        .setType(MessageType.CONNECTION_DENIED)
                        .setError("You can not listen to more radios with current user").build();
                LOGGER.info("Revoke request connection from " + request.getUser() + " in radio " + request.getRadio());
            }
        } else {
            messageBuilder
                    .setType(MessageType.CONNECTION_DENIED)
                    .setError("Radio no exist or not in transmission").build();
            LOGGER.info("Revoke request connection from " + request.getUser() + ". Radio " + request.getRadio() + " no exist");
        }
        return messageBuilder.build();
    }



    private void start() {
        LOGGER.info("Waiting client request");

        consumerClientTag = communication.append(CLIENT_QUEUE, message → {
            if (message.getType() ≡ MessageType.REQUEST_CONNECTION) {
                Message response = handlerRequestConnection(message);
                communication.put(message.getUserQueue(), response);
            } else if (message.getType() ≡ MessageType.REQUEST_RADIOS) {
                StringBuilder stationsFlat = new StringBuilder();
                db.getStations().forEach(s → stationsFlat.append("- ").append(s).append("\n"));
                Message response = new MessageBuilder()
                        .setType(MessageType.RESPONSE_RADIOS)
                        .setInfo(stationsFlat.toString())
                        .build();
                communication.put(message.getUserQueue(),response);
            } else if (message.getType() ≡ MessageType.KEEP_ALIVE) {
                db.updateUserActivity(message.getUser());
                LOGGER.info("Updated activity for user " + message.getUser());
            } else if (message.getType() ≡ MessageType.END_CONNECTION) {
                db.deleteUserFromRadio(message.getUser(), message.getUserQueue(), message.getRadio());
                LOGGER.info(String.format("Deleted user %s from radio %s", message.getUser(), message.getRadio()));
                CONNECTION_LOGGER.info(String.format("[DISCONNECTION] user '%s' to radio '%s'",message.getUser(),message.getRadio()));
            } else {
                LOGGER.warn("Unhandled Message with type " +  message.getType());
            }
        });

        try {
            registerSIGINT();
        } catch (InterruptedException e) {
            LOGGER.warn("Interrupt signal");
            LOGGER.debug(e);
        }
        LOGGER.info("Exiting");
    }

    public static void main(String[] argv) {
        UserGestor userGestor = null;
```

```
118         try {
119             userGestor = new UserGestor();
120         } catch (Exception e) {
121             LOGGER.info("Cannot start user gestor");
122             LOGGER.debug(e);
123             System.exit(1);
124         }
125         userGestor.start();
126     }
127 }
```



```
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.Semaphore;
9  import java.util.concurrent.TimeUnit;
10 import java.util.concurrent.atomic.AtomicBoolean;
11
12 public class Station {
13     private static final Logger LOGGER = Logger.getLogger(Station.class);
14     private static final Settings SETTINGS = Settings.from("station.properties");
15
16     private static final String RADIO_QUEUE = SETTINGS.get("RADIO_QUEUE","RADI
   O");
17     private static final String RABBITMQ_HOST = SETTINGS.get("RABBITMQ_HOST","l
   ocalhost");
18     private static final int RABBITMQ_PORT = SETTINGS.get("RABBITMQ_PORT",5672)
   ;
19     private static final int MESSAGE_EXPIRATION_TIME_SECONDS = SETTINGS.get("ME
   SSAGE_EXPIRATION_TIME_SECONDS",60);
20     private static final int PACKAGE_BYTE_SIZE = SETTINGS.get("PACKAGE_BYTE_SIZ
   E",192000);
21     private static final int TIME_PER_PACKAGE_SECONDS = SETTINGS.get("TIME_PER_P
   ACKAGE_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, RABBIT
   MQ_PORT);
29         if (communication ≡ null) {
30             LOGGER.warn("Cannot establish communication");
31             throw new Exception("Cannot establish communication");
32
33         }
34
35         if (¬communication.queueDeclare(RADIO_QUEUE)) {
36             LOGGER.warn("Cannot declare Queue " + RADIO_QUEUE);
37             throw new Exception("Cannot declare Queue " + RADIO_QUEUE);
38         }
39
40         this.name = name;
41         this.transmissionStarted = new AtomicBoolean(false);
42
43     }
44
45     private void startTransmission(String filePath) throws InterruptedException
   {
46         File file = new File(filePath);
47         try (FileInputStream fis = new FileInputStream(file)) {
48             String fileExtension = filePath.substring(filePath.lastIndexOf('.')
   + 1);
49             int totalBytes = fis.available();
50             LOGGER.info("Attempt to send " + file.getName() + ". Content type " + fileExte
   nsion);
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) {
```

abr 21, 18 15:01 **Station.java** Page 2/2

```
57                  byteCountRead+= byteCount;
58                  Message message = new MessageBuilder()
59                          .setType(MessageType.RADIO_PACKAGE)
60                          .setRadio(name)
61                          .setContentType(fileExtension)
62                          .setPayload(Arrays.copyOfRange(bytes,0,byteCount))
63                          .build();
64                  communication.put(RADIO_QUEUE, message,MESSAGE_EXPIRATION_TIME_S
    ECONDS);
65                  LOGGER.debug("ByteCount " + byteCount + ". Left: " + (totalBytes - by
    teCountRead));
66                  LOGGER.info("Sent " + byteCount + " bytes. " + (byteCountRead*100)/t
    otalBytes + "%");
67
68                  TimeUnit.SECONDS.sleep(TIME_PER_PACKAGE_SECONDS);
69              }
70              LOGGER.info("End stream");
71          } catch (IOException e) {
72              LOGGER.debug(e);
73              LOGGER.warn("Cannot read file stream " + filePath);
74          }
75      }
76
77      private void stopTransmission() {
78          if (transmissionStarted.compareAndSet(true, false)) {
79              LOGGER.info("Closing connection (send end transmission message");
80              Message endMessage = new MessageBuilder()
81                      .setType(MessageType.END_TRANSMISSION)
82                      .setRadio(name)
83                      .build();
84              communication.put(RADIO_QUEUE, endMessage,MESSAGE_EXPIRATION_TIME_SE
    CONDS);
85              communication.close();
86          }
87      }
88
89      public static void main(String[] args) {
90          if (args.length < 2) {
91              System.out.println("Use: ./station <<name>> <<file>>");
92              LOGGER.fatal("Invalid parameters");
93              return;
94          }
95
96          try {
97              Station station = new Station(args[0]);
98              Runtime.getRuntime().addShutdownHook(new Thread(() → {
99                          LOGGER.info("SIGINT detected. Closing Station");
100                         station.stopTransmission();
101                         if (LOGGER.isDebugEnabled()) {
102                             try {
103                                 TimeUnit.SECONDS.sleep(1);
104                             } catch (InterruptedException ignored) {}
105                         }
106                     })
107             );
108             station.startTransmission(args[1]);
109             station.stopTransmission();
110         } catch (Exception e) {
111             LOGGER.debug(e);
112             LOGGER.warn("Cannot create station");
113         }
114     }
115 }
```

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",propertie
    sFile));
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 propert
    y: " + 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(defau
    ltValue)));
46      }
47
48      public String get(String name, String defaultValue) {
49          return properties.getProperty(name, defaultValue);
50      }
51  }
```

```java
import Message.*;

import org.apache.log4j.*;
import sun.misc.Signal;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.Consumer;

public class RadioListener {
    private static final Logger LOGGER = Logger.getLogger(RadioListener.class);
    private static final Settings SETTINGS = Settings.from("radio-listener.properties");

    private static final String RABBITMQ_HOST   = SETTINGS.get("RABBITMQ_HOST", "localhost");
    private static final int RABBITMQ_PORT      = SETTINGS.get("RABBITMQ_PORT", 5672);
    private static final String CLIENT_QUEUE    = SETTINGS.get("CLIENT_QUEUE","CLIENT");
    private static final int TIMEOUT_SECONDS    = SETTINGS.get("TIMEOUT_SECONDS",10);
    private static final int KEEP_ALIVE_POLL_SECONDS  = SETTINGS.get("KEEP_ALIVE_POLL_SECONDS",60);
    private static final int POOL_SIZE = SETTINGS.get("POOL_SIZE",10);


    private final String user;
    private final String radio;
    private File streamFile;

    private CommunicationWrapper comm;
    private String consumerTag;
    private String listenQueue;
    private AtomicBoolean isConnected;

    private ScheduledExecutorService keepAliveScheduler;
    private ScheduledExecutorService connectedScheduler;


    private void initCommunication() throws IOException {
        comm = CommunicationWrapper.getConnection(RABBITMQ_HOST, RABBITMQ_PORT);
        if (comm == null) {
            LOGGER.fatal("Cannot open connection. Server is down");
            throw new IOException("Cannot open connection. Server is up?");
        }

        listenQueue = comm.queueDeclare();
        if (listenQueue == null) {
            throw new IOException("Cannot declare queue to receive response");
        }
        isConnected = new AtomicBoolean(false);
    }

    private RadioListener() throws IOException {
        initCommunication();
        this.user = null;
        this.radio = null;
        this.keepAliveScheduler = null;
        this.connectedScheduler = null;
        this.streamFile = null;
```



```java
    }

    private RadioListener(String user, String radio) throws IOException {
        initCommunication();
        this.user = user;
        this.radio = radio;
        this.keepAliveScheduler = Executors.newScheduledThreadPool(POOL_SIZE);

    }

    private File createFile(String extension) throws IOException {
        String fileName = this.user + "-" + this.radio + "." + extension;
        File fileStream = new File(fileName);
        int i = 1;
        while (fileStream.exists()) {
            fileName = this.user + "-" + this.radio + "-" + i + "." + extension;
            fileStream = new File(fileName);
            i++;
        }
        if (¬fileStream.createNewFile()) {
            if (¬fileStream.exists()) {
                throw new IOException("Failed on create file to write");
            }
        }
        LOGGER.info("Created file " + fileName + " to store radio packages");
        this.streamFile = fileStream;
        return fileStream;
    }

    private void handleRadioPackage(Message message) {
        FileOutputStream out;
        try {
            File fileStream = this.streamFile ≡ null ? createFile(message.getContentType()) : this.streamFile;
            out = new FileOutputStream(fileStream, true);
            byte[] bytes = message.getPayload();
            LOGGER.debug("Write " + bytes.length + " in " + fileStream.getName());
            out.write(bytes);
            out.close();
        } catch (IOException e) {
            LOGGER.warn("Cannot write radio package. Ignoring it");
            LOGGER.debug(e);
        }
    }

    private void handleResponse(Message res) {
        if (res.getType() ≡ MessageType.CONNECTION_ACCEPTED) {
            LOGGER.info("Receive Connection accepted");
            isConnected.set(true);
            startSchedulerToSendKeepAlive();
            System.out.println("Connected to radio '" + radio + "'");
        }
        if (res.getType() ≡ MessageType.CONNECTION_DENIED) {
            LOGGER.info("Connection denied");
            System.out.println("Cannot connect with radio. Error: \"" + res.getError() + "\"");
        }
        if (res.getType() ≡ MessageType.RADIO_PACKAGE) {
            LOGGER.info("Receive radio package");
            handleRadioPackage(res);
        }
        if (res.getType() ≡ MessageType.END_CONNECTION) {
            LOGGER.info("Receive end connection");
            stop();
        }
    }
```

```
126
127     private void startSchedulerToSendKeepAlive() {
128         keepAliveScheduler.scheduleAtFixedRate(() → {
129             LOGGER.info("Send keep alive to server");
130             Message message = new MessageBuilder()
131                     .setType(MessageType.KEEP_ALIVE)
132                     .setUser(user)
133                     .build();
134             comm.put(CLIENT_QUEUE, message);
135         }, KEEP_ALIVE_POLL_SECONDS, KEEP_ALIVE_POLL_SECONDS, TimeUnit.SECONDS);
136     }
137
138     private void disconnect() {
139         comm.put(CLIENT_QUEUE, new MessageBuilder()
140                 .setUser(user)
141                 .setRadio(radio)
142                 .setClientQueue(listenQueue)
143                 .setType(MessageType.END_CONNECTION).build()
144         );
145     }
146
147     private synchronized void stop() {
148         if (isConnected.compareAndSet(true, false)) {
149             disconnect();
150         }
151         if (keepAliveScheduler ≠ null) {
152             keepAliveScheduler.shutdownNow();
153             keepAliveScheduler = null;
154         }
155         if (connectedScheduler ≠ null) {
156             connectedScheduler.shutdown();
157             connectedScheduler = null;
158         }
159         if (¬comm.detach(consumerTag)) {
160             LOGGER.warn("Cannot detach");
161         }
162         if (listenQueue ≠ null) {
163             comm.deleteQueue(listenQueue);
164         }
165         if (comm ≠ null) {
166             comm.close();
167             comm = null;
168         }
169         LOGGER.info("Exit");
170     }
171
172     private void waitResponseWithTimeout(Consumer<Message> handler) {
173         connectedScheduler = Executors.newScheduledThreadPool(POOL_SIZE);
174         connectedScheduler.schedule(() → {}, Long.MAX_VALUE, TimeUnit.DAYS);
175
176         ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(PO
    OL_SIZE);
177         LOGGER.info("Start timeout to wait response. TIMEOUT=" + TIMEOUT_SECONDS);
178         scheduler.schedule(() → {
179                     LOGGER.info("TIMEOUT: Wake−up. Close connection. The servers are not working
    ");
180                     connectedScheduler.shutdownNow();
181                 }, TIMEOUT_SECONDS, TimeUnit.SECONDS
182         );
183
184         // Wait responses
185         LOGGER.info("Waiting response");
186         consumerTag = comm.append(listenQueue, res → {
187             handler.accept(res);
188             scheduler.shutdownNow();
189         });
```



```
190
191         scheduler.shutdown();
192         connectedScheduler.shutdown();
193         try {
194             scheduler.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);
195             connectedScheduler.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);
196         } catch (InterruptedException ignored) {
197             LOGGER.info("Interrupted Exception");
198         }
199     }
200
201     private void startListener() {
202
203         Message request = new MessageBuilder().setType(MessageType.REQUEST_CONNE
    CTION)
204                 .setClientQueue(listenQueue)
205                 .setRadio(radio)
206                 .setUser(user)
207                 .build();
208         comm.put(CLIENT_QUEUE, request, TIMEOUT_SECONDS);
209
210         waitResponseWithTimeout(this::handleResponse);
211
212         if (¬isConnected.get()) {
213             stop();
214         }
215     }
216
217     private void listRadios() {
218
219         // Send request
220         Message request = new MessageBuilder().
221                 setType(MessageType.REQUEST_RADIOS)
222                 .setClientQueue(listenQueue)
223                 .build();
224         comm.put(CLIENT_QUEUE, request, TIMEOUT_SECONDS);
225
226
227         waitResponseWithTimeout(res → {
228             LOGGER.info("Received response from server");
229             if (res.getType() ≡ MessageType.RESPONSE_RADIOS) {
230                 System.out.println("Radios:\n" + res.getInfo());
231             }
232             if (this.connectedScheduler ≠ null) {
233                 this.connectedScheduler.shutdownNow();
234             }
235         });
236
237         stop();
238
239     }
240
241     private void start() {
242         if (this.user ≡ null) {
243             listRadios();
244         } else {
245             startListener();
246         }
247     }
248
249     public static void main(String[] args) {
250         if (args.length ≡ 0) {
251             System.out.println(" − Listen Radio: ./radio−listener <<user>> <<radio>>");
252             System.out.println(" − List Radios: ./radio−listener list");
253             return;
254         }
```

abr 21, 18 16:12 **RadioListener.java** Page 5/5

```
255
256         boolean listRadios = (args.length ≡ 1 ∧ args[0].toLowerCase().equals("list
    "));
257         try {
258             RadioListener radioListener = (listRadios) ? new RadioListener() : n
    ew RadioListener(args[0], args[1]);
259             Runtime.getRuntime().addShutdownHook(new Thread(() → {
260                 LOGGER.info("SIGINT detected. Closing connection");
261                 radioListener.stop();
262                 LOGGER.info("Connection closed");
263             }));
264             radioListener.start();
265             LOGGER.info("Goodbye");
266         } catch (IOException e) {
267             LOGGER.fatal("Cannot start radio-listener");
268             LOGGER.debug(e);
269         }
270
271     }
272
273
274 }
```

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         comm.queueDeclare(SETTINGS.get("ADMIN_RESPONSE_QUEUE","ADMIN_RESPONSE
    "));
20
21         comm.queueDeclare(SETTINGS.get("RADIO_QUEUE","RADIO"));
22
23         comm.queueDeclare(SETTINGS.get("CLIENT_QUEUE","CLIENT"));
24
25         comm.close();
26         LOGGER.info("OK. Initializer");
27
28     }
29 }
```

```java

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

public class Main {
    private static String printThread() {
        return Thread.currentThread().getName();
    }

    private static void testExecutors() throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(5);

        Object mutex = new Object();

        Callable<Integer> task = () → {
            synchronized (mutex) {
                System.out.println(printThread() + " Tomo mutex");
                TimeUnit.SECONDS.sleep(3);
                System.out.println(printThread() + " Libero mutex");
            }
            return 1;
        };

        List<Callable<Integer>> runnables = new ArrayList<>();
        while (runnables.size() < 10) {
            runnables.add(task);
        }

        executor.invokeAll(runnables);
        executor.shutdown();
        executor.awaitTermination(Long.MAX_VALUE, TimeUnit.MILLISECONDS);

        System.out.println(printThread() + " Hilo principal!");
    }


    private static void testFileLock() {
        try {
            FileCellBlock file = new FileCellBlock("dummydb", 100);
            file.insert("1");
            file.insert("2");
            file.insert("3");
            file.iterFile(System.out::println);
            file.delete("1", String::compareTo);
            System.out.println("-----");
            file.iterFile(System.out::println);
            System.out.println("-----");
            file.insert("4");
            file.iterFile(System.out::println);
            System.out.println(file.find(s → true).size());
            file.clean();
        } catch (IOException | IndexOutOfBoundsException e) {
            e.printStackTrace();
        }


    }

    public static void main(String[] args) throws InterruptedException {
        //testExecutors();
        testFileLock();
        System.exit(0);
    }
```



```java
}
```

abr 20, 18 19:51 **Destructor.java** Page 1/1

```java
import org.apache.log4j.Logger;

import java.io.IOException;

public class Destructor {
    private static final Logger LOGGER = Logger.getLogger(Destructor.class);
    private static final Settings SETTINGS = Settings.from("destructor.properties");
    public static void main(String[] args) {
        LOGGER.info("Desruct all Queues and Databases");
        CommunicationWrapper comm = CommunicationWrapper.getConnection(
                SETTINGS.get("RABBITMQ_HOST","localhost"),
                SETTINGS.get("RABBITMQ_PORT",5672)
        );
        if (comm == null) {
            LOGGER.fatal("Cannot connect");
            return;
        }

        comm.deleteQueue(SETTINGS.get("ADMIN_REQUEST_QUEUE","ADMIN_REQUEST"));

        comm.deleteQueue(SETTINGS.get("ADMIN_RESPONSE_QUEUE","ADMIN_RESPONSE"));

        comm.deleteQueue(SETTINGS.get("RADIO_QUEUE","RADIO"));

        comm.deleteQueue(SETTINGS.get("CLIENT_QUEUE","CLIENT"));

        if (SETTINGS.get("CLEAN_DATABASES",false)) {
            LOGGER.info("DBs cleaned");
            try {
                DB database = new BlockDatabase();
                database.cleanDatabases();
            } catch (IOException e) {
                LOGGER.warn("Cannot clean databases");
            }
        }

        comm.close();
        LOGGER.info("OK. End destructor");
    }
}
```

abr 20, 18 20:22 **JSONDatabase.java** Page 1/5

```java
import org.apache.log4j.Logger;
import org.json.simple.JSONArray;
import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;
import org.json.simple.parser.ParseException;

import java.io.*;
import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

@SuppressWarnings("unchecked")
public class JSONDatabase implements DB {
    private static final Logger LOGGER = Logger.getLogger(JSONDatabase.class);
    private static final Settings SETTINGS = Settings.from("../database.properties");

    private static final int MAX_RADIOS_PER_CLIENT = SETTINGS.get("MAX_RADIOS_PER_CLIENT",3);

    private static final String WORKING_DIR     = SETTINGS.get("WORKING_DIR","../database/");
    private static final String USERS_DB        = SETTINGS.get("USER_DB","user");
    private static final String STATIONS_DB     = SETTINGS.get("STATION_DB", "station");
    private static final String CONNECTIONS_DB  = SETTINGS.get("CONNECTION_DB", "connection");

    private static final int OFFSET_TIMESTAMP   = SETTINGS.get("OFFSET_TIMESTAMP", 2);

    JSONDatabase() throws IOException {
        File workingDir = new File(WORKING_DIR);
        if (¬workingDir.exists()) {
            if (¬workingDir.mkdir()) {
                if (¬workingDir.exists()) {
                    LOGGER.fatal("Cannot create working dir");
                    throw new IOException("Cannot create working dir");
                }
            }
            LOGGER.debug("Created working dir for DB: " + WORKING_DIR);
        }
        String[] files = {STATIONS_DB, CONNECTIONS_DB, USERS_DB};
        for (String file : files) {
            String path = WORKING_DIR + file;
            File f = new File(path);
            if (¬f.exists()) {
                if (¬f.createNewFile()) {
                    if (¬f.exists()) {
                        LOGGER.fatal("Cannot create file " + path);
                        throw new IOException("Cannot create file " + path);
                    }
                }
                LOGGER.info(String.format("Created file DB: \"%s\"",path));
            }
        }
    }

    public void cleanDatabases() {
        String[] DBNames = {STATIONS_DB, USERS_DB, CONNECTIONS_DB};
        for (String DBName: DBNames) {
            if (writeJSON(new JSONObject(), DBName)) {
                LOGGER.info(DBName + " cleaned");
            } else {
                LOGGER.warn("Cannot clean " + DBName);
```

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



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

```
187                 }
188                 usersActivity.put(userName, userActivity);
189             } else {
190                 JSONObject userActivity = new JSONObject();
191                 userActivity.put("last", timestamp.getTime());
192                 userActivity.put("total", 1);
193                 usersActivity.put(userName, userActivity);
194             }
195             if (writeJSON(usersActivity, CONNECTIONS_DB)) {
196                 LOGGER.info("Update activity for user " + userName);
197             }
198 
199         }
200 
201         public List<String> getStations() {
202             ArrayList<String> stationsArray = new ArrayList<>();
203             JSONObject stations = readJSON(STATIONS_DB);
204             if (stations ≡ null) {
205                 return stationsArray;
206             }
207             stationsArray.addAll(stations.keySet());
208             return stationsArray;
209         }
210 
211         public List<String> getTopUsers(int count) {
212             JSONObject users = readJSON(CONNECTIONS_DB);
213             if (users ≡ null) {
214                 return new ArrayList<>();
215             }
216             return (List<String>) users.keySet().stream()
217                     .sorted( (u1,u2) → {
218                         JSONObject user1 = (JSONObject) users.get(u1);
219                         JSONObject user2 = (JSONObject) users.get(u2);
220                         Long total1 = (Long) user1.get("total");
221                         Long total2 = (Long) user2.get("total");
222                         return -total1.compareTo(total2);
223                     })
224                     .limit(count)
225                     .map(userName → userName + "|total: " + ((JSONObject)users.get(use
    rName)).get("total") + " sec.")
226                     .collect(Collectors.toList());
227         }
228 
229         public List<String> getUsersInStation(String station) {
230             List<String> userQueue = new ArrayList<>();
231             JSONObject stations = readJSON(STATIONS_DB);
232             if (stations ≡ null ∨ ¬stations.containsKey(station)) {
233                 return userQueue;
234             }
235             JSONArray usersQueue = (JSONArray) stations.get(station);
236             for (Object queue : usersQueue) {
237                 userQueue.add((String)queue);
238             }
239             return userQueue;
240         }
241 
242         public void addStation(String station) {
243             JSONObject stations = readJSON(STATIONS_DB);
244             if (stations ≡ null ∨ stations.containsKey(station)) {
245                 return;
246             }
247             stations.put(station, new JSONArray());
248             if (writeJSON(stations, STATIONS_DB)) {
249                 LOGGER.info("Added station " + station + " in DB");
250             }
251         }
```



```
252 
253         public void deleteStation(String station) {
254             JSONObject stations = readJSON(STATIONS_DB);
255             if (stations ≡ null ∨ ¬stations.containsKey(station)) {
256                 return;
257             }
258             stations.remove(station);
259             if (writeJSON(stations, STATIONS_DB)) {
260                 LOGGER.info("Deleted station " + station + " in DB");
261             }
262         }
263 
264         public List<String> getCountUserPerStation() {
265             return new ArrayList<>();
266         }
267     }
```

```java
import org.apache.log4j.Logger;

import java.io.*;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.FileLock;
import java.nio.file.StandardOpenOption;
import java.util.*;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.function.BiPredicate;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;

public class FileCellBlock {
    private static final Logger LOGGER = Logger.getLogger(FileCellBlock.class);

    private static final char NULL = '\0';

    private final int blockSize;
    private final File file;

    FileCellBlock(String file, int blockSize) throws IOException {
        this.blockSize = blockSize;
        this.file = new File(file);
        createFile();

    }

    private void createFile() throws IOException {
        if (¬file.exists() ∧ ¬file.createNewFile() ∧ ¬file.exists()) {
            if (¬file.createNewFile()) {
                if (¬file.exists()) {
                    LOGGER.fatal("Cannot create file " + file.toString());
                    throw new IOException("Cannot create file " + file.toString());
                }
            }
            LOGGER.info(String.format("Created file DB: \"%s\"",file.toString()));
        }
    }

    private byte[] generateNullBlock() {
        // Default NULL Block.
        byte[] block = new byte[blockSize];
        Arrays.fill(block, (byte) NULL);
        return block;
    }

    private byte[] toBlock(String s) {
        byte[] string = s.replaceAll(String.valueOf(NULL),"").getBytes();
        if (string.length > blockSize) {
            return null;
        }
        byte[] block = generateNullBlock();
        System.arraycopy(string, 0, block, 0, string.length);
        return block;
    }

    private String toString(byte[] block) {
        if (Arrays.equals(block, generateNullBlock())) {
            return "<<FREE BLOCK>>";
        }
        String str = new String(block);
        int indexOfEnd = str.indexOf('\0');
        return indexOfEnd ≠ -1 ? str.substring(0, str.indexOf(NULL)) : str;
    }
```



```java

    private void writeBlockInEnd(final String s) {
        // Write s in the end
        byte[] block = toBlock(s);
        if (block ≡ null) {
            return;
        }
        try (FileOutputStream out = new FileOutputStream(file,true)) {
            FileLock lock = out.getChannel().lock(out.getChannel().position(), blockSize, false);
                out.write(block);
            lock.release();
        } catch (IOException e) {
            LOGGER.error(String.format("Cannot write file '%s'",file.toString()));
        }
    }

    public void clean() {
        try (FileOutputStream ignored = new FileOutputStream(file,false)) {
            LOGGER.info(String.format("Cleaned file '%s'",file.toString()));
        } catch (IOException e) {
            LOGGER.error(String.format("Cannot clean file '%s'",file.toString()));
        }
    }

    private void writeBlock(final String s, int position) {
        // Write Block in position
        // NOTE: The file was block from position*blocSize a blockSize length
        // If position < 0. Write on end
        if (position < 0) {
            writeBlockInEnd(s);
            return;
        }
        byte[] block = toBlock(s);
        if (block ≡ null) {
            return;
        }
        try (FileChannel out = FileChannel.open(file.toPath(), StandardOpenOption.WRITE)) {
            long offset = position * blockSize;
            FileLock lock = out.lock(position, blockSize, false);
                out.write(ByteBuffer.wrap(block), offset);
            lock.release();
        } catch (IOException e) {
            LOGGER.error(String.format("Cannot write file '%s'",file.toString()));
        }
    }

    public void iterFile(final BiPredicate<Integer, String> handleBlock) {
        // Iter file lockin to read per block
        // If handleBlock return true. Stop iter.
        try (FileInputStream in = new FileInputStream(file)) {
            int byteCount = 0;
            long offset = 0L;
            int position = 0;
            boolean stop = false;
            while (byteCount ≠ -1 ∧ ¬stop) {
                FileLock lock = in.getChannel().lock(offset, blockSize, true);
                byte[] bytes = new byte[blockSize];
                byteCount = in.read(bytes);
                if (byteCount ≠ -1) {
                    stop = handleBlock.test(position, toString(bytes));
                }
                lock.release();
                offset += blockSize;
                position += 1;
```

```
131             }
132         } catch (IOException e) {
133             LOGGER.error(String.format("Cannot write file '%s'",file.toString()));
134             LOGGER.debug(e);
135         }
136     }
137
138     public void iterFile(final Consumer<String> handleBlock) {
139         // Iter file. Pass handler to handled al String per block
140         iterFile((pos, string) → {
141             handleBlock.accept(string);
142             return false;
143         });
144     }
145
146     private int getPosition(String s, Comparator<String> comparator) {
147         AtomicInteger position = new AtomicInteger(-1);
148         iterFile((pos, string) → {
149             if ( comparator.compare(s, string) ≡ 0) {
150                 position.set(pos);
151                 return true;
152             }
153             return false;
154         });
155         return position.get();
156     }
157
158     public void update(String defaultValue, Function<String, String> updater) {
159         // Iter all block and collect position who updater change value of block
160         // If collected position is null. Write in the end default value
161         Map<Integer, String> newPositionsStrings = new HashMap<>();
162         Map<Integer, String> oldPositionsStrings = new HashMap<>();
163         iterFile((pos, string) → {
164             String newValue = updater.apply(string);
165             if (¬string.equalsIgnoreCase(newValue)) {
166                 newPositionsStrings.put(pos, newValue);
167                 oldPositionsStrings.put(pos, string);
168             }
169             return false;
170         });
171         if (¬newPositionsStrings.isEmpty()) {
172             newPositionsStrings.forEach((key, value) → {
173                 writeBlock(value, key);
174                 LOGGER.debug(String.format("Updated block %d.\n'%s' -> '%s'", key, oldP
    ositionsStrings.get(key), value));
175             });
176         } else {
177             writeBlockInEnd(defaultValue);
178             LOGGER.debug(String.format("Update not find block to update. Write '%s' on the end",
    defaultValue));
179         }
180
181     }
182
183     public void insert(String s) {
184         // Write block in first free position.
185         // Is no exist free position. write in the end
186         String nullString = toString(generateNullBlock());
187         writeBlock(s, getPosition(nullString, String::compareTo));
188     }
189
190     public void delete(String s, Comparator<String> comparator) {
191         // Delete first block who match with s
192         // Do nothing if not exist
193         String nullString = toString(generateNullBlock());
194         int position = getPosition(s, comparator);
```



```
195         if (position ≠ -1) {
196             writeBlock(nullString, position);
197         }
198     }
199
200     public void delete(Predicate<String> predicate) {
201         // Delete blocks who predicate(stringBlock) returns true
202         List<Integer> positionsToDelete = new ArrayList<>();
203         iterFile((pos, string) → {
204             if (predicate.test(string)) {
205                 positionsToDelete.add(pos);
206             }
207             return false;
208         });
209         positionsToDelete.forEach(pos → writeBlock(toString(generateNullBlock()
    ), pos));
210     }
211
212     private boolean isNotNull(String s) {
213         return ¬s.equalsIgnoreCase(toString(generateNullBlock()));
214     }
215
216     public List<String> find(Predicate<String> comparator) {
217         // Iter al blocks and collect who predicate return true
218         List<String> collect = new ArrayList<>();
219         iterFile((pos, string) → {
220             if (isNotNull(string) ∧ comparator.test(string)) {
221                 collect.add(string);
222             }
223             return false;
224         });
225         return collect;
226     }
227
228     public File getFile() {
229         return this.file;
230     }
231 }
```

abr 20, 18 15:17 **UserListenCount.java** Page 1/1

```java
package Entities;

public class UserListenCount {
    private static final String SEPARATOR = "===";

    private final String name;
    protected int count;

    public UserListenCount(String name, int count) {
        this.name = name;
        this.count = count;
    }

    public static UserListenCount from(String row) {
        String[] cols = row.split(SEPARATOR);
        if (cols.length ≠ 2) {
            return null;
        }
        return new UserListenCount(cols[0], Integer.parseInt(cols[1]));
    }

    public boolean is(String userName) {
        return userName.equalsIgnoreCase(name);
    }

    public void addListen() {
        this.count += 1;
    }

    public void removeListen() {
        this.count -= 1;
        if (this.count < 0) {
            this.count = 0;
        }
    }

    public String toString() {
        return name + SEPARATOR + count;
    }

    public int getCount() {
        return this.count;
    }
}
```

abr 21, 18 15:54 **UserActivity.java** Page 1/2

```java
package Entities;

import java.sql.Timestamp;
import java.util.Comparator;
import java.util.Date;


public class UserActivity {
    private static final String SEPARATOR = "===";
    private static final int INIT_TOTAL_SEC = 1;

    private final String name;
    private long lastTimestampMilliseconds;
    private int total;

    public UserActivity(String name, long lastTimestamp, int total) {
        this.name = name;
        this.lastTimestampMilliseconds = lastTimestamp;
        this.total = total;
    }

    public static UserActivity from(String row) {
        String[] cols = row.split(SEPARATOR);
        if (cols.length ≠ 3) {
            return null;
        }
        return new UserActivity(cols[0], Long.parseLong(cols[1]), Integer.parseI
nt(cols[2]));
    }

    private static long getTimestamp() {
        return new Date().getTime();
    }

    public static UserActivity init(String userName) {
        return new UserActivity(userName, getTimestamp(), INIT_TOTAL_SEC);
    }

    public boolean is(String userName) {
        return name.equalsIgnoreCase(userName);
    }

    public void update(int offsetTimestampSeconds) {
        Long timestamp = getTimestamp();
        Long lastTimestamp = this.lastTimestampMilliseconds;
        Long diff = timestamp - lastTimestamp;
        this.lastTimestampMilliseconds = timestamp;
        Long add = (diff > offsetTimestampSeconds*1000) ? diff/6000 : INIT_TOTAL
_SEC;
        this.total = this.total + add.intValue();
    }

    public String toString() {
        return name + SEPARATOR + lastTimestampMilliseconds + SEPARATOR + total;
    }

    public int getTotal() {
        return this.total;
    }

    public static int compareRow(String r1, String r2) {
        UserActivity userActivity1 = UserActivity.from(r1);
        UserActivity userActivity2 = UserActivity.from(r2);
        if (userActivity1 ≡ null ∧ userActivity2 ≡ null) {
            return 0;
        }
```

abr 21, 18 15:54 **UserActivity.java** Page 2/2

```
65          if (userActivity1 ≠ null ∧ userActivity2 ≡ null) {
66              return -1;
67          }
68          if (userActivity1 ≡ null) {
69              return 1;
70          }
71          return -Integer.compare(userActivity1.total, userActivity2.total);
72      }
73
74      public String getUserName() {
75          return this.name;
76      }
77  }
```

abr 20, 18 20:51 **DB.java** Page 1/1

```
1   import java.util.*;
2
3   public interface DB {
4
5       void cleanDatabases();
6
7       void addUserInRadio(String userName, String userQueue, String radio);
8
9       List<String> getUsersInStation(String radio);
10
11      boolean userCanHearRadio(String userName, String radio);
12
13      void deleteUserFromRadio(String userName, String userQueue, String radio);
14
15      void addStation(String station);
16
17      boolean existStation(String radio);
18
19      void deleteStation(String station);
20
21      void updateUserActivity(String userName);
22
23      List<String> getStations();
24
25      List<String> getTopUsers(int count);
26
27      List<String> getCountUserPerStation();
28  }
```

```java
import Entities.UserActivity;
import Entities.UserListenCount;
import org.apache.log4j.Logger;

import java.io.*;
import java.util.*;
import java.util.stream.Collectors;

@SuppressWarnings("unchecked")
public class BlockDatabase implements DB {
    private static final Logger LOGGER = Logger.getLogger(BlockDatabase.class);
    private static final Settings SETTINGS = Settings.from("../database.properties");

    private static final int MAX_RADIOS_PER_CLIENT = SETTINGS.get("MAX_RADIOS_PER_CLIENT",3);

    private static final String WORKING_DIR     = SETTINGS.get("WORKING_DIR","../database/");
    private static final String STATIONS_DIR     = SETTINGS.get("STATION_DIR", "stations/");
    private static final String USERS_DB        = SETTINGS.get("USER_DB","user");
    private static final String CONNECTIONS_DB  = SETTINGS.get("CONNECTION_DB", "connection");

    private static final int OFFSET_TIMESTAMP   = SETTINGS.get("OFFSET_TIMESTAMP", 2);

    private static final int BLOCK_SIZE         = SETTINGS.get("BLOCK_SIZE", 100);

    private static final String STATION_PREFIX = "station.";

    private final String stationsDir;
    private HashMap<String,FileCellBlock> DB;
    private HashMap<String,FileCellBlock> DBStation;

    BlockDatabase() throws IOException {
        createDir(WORKING_DIR);
        stationsDir = WORKING_DIR + STATIONS_DIR;
        createDir(WORKING_DIR + STATIONS_DIR);
        String[] files = {CONNECTIONS_DB, USERS_DB};
        DB = new HashMap<>();
        for (String file : files) {
            String path = WORKING_DIR + file;
            DB.put(file, new FileCellBlock(path, BLOCK_SIZE));
        }
        DBStation =  new HashMap<>();
        loadStations();
    }

    private void loadStations() throws IOException {
        File file = new File(stationsDir);
        for (File stationFile : Objects.requireNonNull(file.listFiles())) {
            if (stationFile.isFile() ∧ ¬DBStation.containsKey(stationFile.getName())) {
                DBStation.put(stationFile.getName(), new FileCellBlock(stationFile.getPath(), BLOCK_SIZE));
                LOGGER.debug("Load database " + stationFile.getName());
            }
        }
    }

    private void reloadStations() {
        try {
            loadStations();
        } catch (IOException e) {
            LOGGER.warn("Cannot reload stations");
            LOGGER.debug(e);
        }
    }

    private void createDir(String path) throws IOException {
        File workingDir = new File(path);
        if (¬workingDir.exists()) {
            if (¬workingDir.mkdir()) {
                if (¬workingDir.exists()) {
                    LOGGER.fatal("Cannot create working dir");
                    throw new IOException("Cannot create working dir");
                }
            }
            LOGGER.debug("Created working dir for DB: " + WORKING_DIR);
        }
    }

    public void cleanDatabases() {
        DB.forEach((key, value) → value.clean());
        Set<Map.Entry<String, FileCellBlock>> stations = DBStation.entrySet();
        stations.forEach(e → deleteStation(stationName(e.getKey())));
        DBStation.clear();
    }

    private String stationKey(String radio) {
        return STATION_PREFIX + radio;
    }

    private String stationName(String key) {
        return key.replace(STATION_PREFIX, "");
    }

    private FileCellBlock getStationDB(String name) throws IOException {
        String key = stationKey(name);
        String newFilePath = WORKING_DIR + STATIONS_DIR + key;
        return DBStation.getOrDefault(key, new FileCellBlock(newFilePath, BLOCK_SIZE));
    }

    public void addUserInRadio(String userName, String userQueue, String radio) {
        try {
            getStationDB(radio).insert(userQueue);
            UserListenCount defaultValue = new UserListenCount(userName, 1);
            DB.get(USERS_DB).update(defaultValue.toString(), row → {
                UserListenCount userRow = UserListenCount.from(row);
                if (userRow ≠ null ∧ userRow.is(userName)) {
                    userRow.addListen();
                    return userRow.toString();
                }
                return row;
            });
        } catch (IOException e) {
            LOGGER.warn(String.format("Cannot add user '%s' in radio '%s'",userName,radio));
            LOGGER.debug(e);
        }
    }

    public void deleteUserFromRadio(String userName, String userQueue, String radio) {
        try {
            getStationDB(radio).delete(row → row.equalsIgnoreCase(userQueue));
            UserListenCount defaultValue = new UserListenCount(userName, 0);
```

```
120             DB.get(USERS_DB).update(defaultValue.toString(), row → {
121                 UserListenCount userRow = UserListenCount.from(row);
122                 if (userRow ≠ null ∧ userRow.is(userName)) {
123                     userRow.removeListen();
124                     return userRow.toString();
125                 }
126                 return row;
127             });
128         } catch (IOException e) {
129             LOGGER.warn(String.format("Cannot add user '%s' in radio '%s'",userName,radio)
    );
130             LOGGER.debug(e);
131         }
132     }
133
134     public boolean existStation(String radio) {
135         File radioDB = new File(WORKING_DIR + STATIONS_DIR + stationKey(radio));
136         return radioDB.exists();
137     }
138
139     public boolean userCanHearRadio(String userName, String radio) {
140
141         FileCellBlock users = DB.get(USERS_DB);
142
143         List<String> result = users.find(r → {
144             UserListenCount user = UserListenCount.from(r);
145             return  (user ≠ null ∧ user.is(userName));
146         });
147         if (result.isEmpty()) {
148             return true;
149         }
150         UserListenCount user = UserListenCount.from(result.get(0));
151         return user ≠ null ∧ user.getCount() < MAX_RADIOS_PER_CLIENT;
152
153     }
154
155     public void updateUserActivity(String userName) {
156         UserActivity initialActivity = UserActivity.init(userName);
157         DB.get(CONNECTIONS_DB).update(initialActivity.toString(), row → {
158             UserActivity userRow = UserActivity.from(row);
159             if (userRow ≠ null) {
160                 if (userRow.is(userName)) {
161                     userRow.update(OFFSET_TIMESTAMP);
162                     return userRow.toString();
163                 }
164             }
165             return row;
166         });
167     }
168
169     public List<String> getStations() {
170         reloadStations();
171         List<String> stations = new ArrayList<>();
172         DBStation.entrySet().stream()
173                 .sorted(Comparator.comparing(Map.Entry::getKey))
174                 .forEach(e → stations.add(stationName(e.getKey())));
175         return stations;
176     }
177
178     public List<String> getUsersInStation(String station) {
179         reloadStations();
180         String key = stationKey(station);
181         return DBStation.containsKey(key) ? DBStation.get(key).find(s → true) :
    new ArrayList<>();
182     }
183
```



```
184     public List<String> getTopUsers(int count) {
185         List<String> topUser = new ArrayList<>();
186         DB.get(CONNECTIONS_DB).find(s→ true).stream()
187                 .sorted(UserActivity::compareRow)
188                 .limit(count)
189                 .forEach(row → {
190                     UserActivity activity = UserActivity.from(row);
191                     if (activity ≠ null) {
192                         topUser.add(String.format("%s %d", activity.getUserName(
    ),activity.getTotal()));
193                     }
194                 });
195         return topUser;
196     }
197
198     public void addStation(String station) {
199         try {
200             getStationDB(station);
201         } catch (IOException e) {
202             LOGGER.warn("Cannot add station " + station);
203             LOGGER.debug(e);
204         }
205     }
206
207     public void deleteStation(String station) {
208         reloadStations();
209         if (DBStation.containsKey( stationKey(station) )) {
210             try {
211                 File file = getStationDB(station).getFile();
212                 if (file.delete()) {
213                     DB.remove( stationKey(station) );
214                     LOGGER.debug("Deleted station DB " + station);
215                     return;
216                 }
217             } catch (IOException e) {
218                 LOGGER.debug(e);
219             }
220             LOGGER.warn("Cannot delete station " + station);
221         }
222     }
223
224     public List<String> getCountUserPerStation() {
225         reloadStations();
226         HashMap<String, Integer> stationUserCount = new HashMap<>();
227
228         DBStation.forEach((station, file) → stationUserCount.put(station, file.
    find(s → true).size()));
229
230         List<String> userCountPerRadio = new ArrayList<>();
231
232         stationUserCount.entrySet().stream()
233                 .sorted((e1, e2) → -e1.getValue().compareTo(e2.getValue()))
234                 .forEach(e → userCountPerRadio.add(String.format("%s (%d)",e.get
    Key(),e.getValue())));
235         return userCountPerRadio;
236     }
237 }
```

abr 18, 18 12:15 **MessageType.java** Page 1/1

```java
package Message;

public enum MessageType {
    REQUEST_RADIOS(0),
    RESPONSE_RADIOS(1),
    REQUEST_CONNECTION(2),
    CONNECTION_ACCEPTED(3),
    CONNECTION_DENIED(4),
    RADIO_PACKAGE(5),
    KEEP_ALIVE(6),
    END_TRANSMISSION(7),
    END_CONNECTION(8),
    ADMIN_REQUEST_STATS(9),
    ADMIN_RESPONSE_STATS(10),
    INVALID(-1); //Default MessageType


    public static MessageType from(int x) {
        MessageType[] values = MessageType.values();
        if (x ≥ values.length) {
            return INVALID;
        }
        return values[x];
    }

    private final int value;
    MessageType(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

abr 16, 18 16:19 **Message.java** Page 1/2

```java
package Message;

import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;
import org.json.simple.parser.ParseException;

import java.util.Base64;

import static java.nio.charset.StandardCharsets.UTF_8;

public class Message {
    private static final JSONParser PARSER = new JSONParser();

    private String raw;
    private JSONObject json;

    public Message(String rawJSON) throws MessageException {
        try {
            json = (JSONObject) PARSER.parse(rawJSON);
            raw = rawJSON;
        } catch (ParseException e) {
            throw new MessageException("Invalid JSON string", e);
        }
    }

    public Message(byte[] bytes) throws MessageException {
        try {
            raw = new String(bytes, UTF_8);
            json = (JSONObject) PARSER.parse(raw);
        } catch (ParseException e) {
            throw new MessageException("Invalid Byte data", e);
        }
    }

    public Message(JSONObject json) {
        this.json = json;
        this.raw = json.toJSONString();
    }


    public MessageType getType() {
        Long type = (Long) json.get("type");
        return MessageType.from(type.intValue());
    }

    public byte[] getPayload() {
        String encoded = json.get("payload").toString();
        return Base64.getDecoder().decode(encoded);
    }

    public String getStringPayload() {
        return new String(getPayload(), UTF_8);
    }

    public String getContentType() {
        return json.get("content_type").toString();
    }

    public byte[] toBytes() {
        return raw.getBytes();
    }

    public String toString() {
        return raw;
    }

}
```

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
    use); }
9       public MessageException(Throwable cause) { super(cause); }
10  }
```

abr 16, 18 16:19 **MessageBuilder.java** Page 1/1

```java
package Message;

import org.json.simple.JSONObject;

import java.util.Base64;


@SuppressWarnings("unchecked")
public class MessageBuilder {
    private final JSONObject messageData;

    public MessageBuilder() {
        messageData = new JSONObject();
    }

    public MessageBuilder setType(MessageType type) {
        messageData.put("type", type.getValue());
        return this;
    }

    public MessageBuilder setPayload(String payload) {
        messageData.put("payload",payload);
        return this;
    }

    public MessageBuilder setPayload(byte[] bytes) {
        return setPayload(Base64.getEncoder().encodeToString(bytes));
    }


    public MessageBuilder setClientQueue(String clientQueue) {
        messageData.put("user_queue", clientQueue);
        return this;
    }

    public MessageBuilder setRadio(String radio) {
        messageData.put("radio", radio);
        return this;
    }

    public MessageBuilder setUser(String user) {
        messageData.put("user", user);
        return this;
    }

    public MessageBuilder setContentType(String contentType) {
        messageData.put("content_type", contentType);
        return this;
    }

    public MessageBuilder setError(String error) {
        messageData.put("error", error);
        return this;
    }

    public MessageBuilder setInfo(String info) {
        messageData.put("info", info);
        return this;
    }

    public Message build() {
        return new Message(messageData);
    }
}
```

abr 21, 18 0:05 **CommunicationWrapper.java** Page 1/3

```java
import Message.*;
import com.rabbitmq.client.*;
import org.apache.log4j.Logger;

import java.io.IOException;
import java.util.concurrent.TimeoutException;
import java.util.function.Consumer;

public class CommunicationWrapper {
    private static final Logger LOGGER = Logger.getLogger(CommunicationWrapper.class);

    static CommunicationWrapper getConnection(String host, int port) {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost(host);
        factory.setPort(port);
        Connection connection = null;
        Channel channel = null;
        try {
            connection = factory.newConnection();
            channel = connection.createChannel();
            return new CommunicationWrapper(channel);
        } catch (IOException | TimeoutException e) {
            LOGGER.error("Error: " + e.getMessage());
        }
        return null;
    }

    private final Channel channel;

    CommunicationWrapper(Channel channel) {
        this.channel = channel;
    }

    void close() {
        try {
            Connection connection = channel.getConnection();
            channel.close();
            connection.close();
        } catch (IOException | TimeoutException e) {
            LOGGER.warn("Cannot close connection " + e.getMessage());
        }
    }

    boolean queueDeclare(String name) {
        try {
            AMQP.Queue.DeclareOk result = channel.queueDeclare(name, true, false, false, null);
            LOGGER.info("Created queue " + result.getQueue());
            return true;
        } catch (IOException e) {
            LOGGER.warn("Cannot declare queue " + name + "." + e.getMessage());
            return false;
        }
    }

    public String queueDeclare() {
        try {
            AMQP.Queue.DeclareOk result = channel.queueDeclare();
            LOGGER.info("Created queue " + result.getQueue());
            return result.getQueue();
        } catch (IOException e) {
            LOGGER.warn("Cannot declare queue. " + e.getMessage());
            return null;
        }
    }
```

```
65
66      private boolean put(String queue, Message message, AMQP.BasicProperties prop
    s) {
67          try {
68              channel.basicPublish("", queue, null, message.toBytes());
69              LOGGER.debug("Send message [" + message.toString().hashCode() + "] on queu
    e " + 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(channe
    l) {
91                  @Override
92                  public void handleDelivery(String consumerTag, Envelope env, AMQ
    P.BasicProperties props, byte[] body) {
93                      try {
94                          Message message = new Message(body);
95                          LOGGER.debug("Receive message [" + message.toString().hashCo
    de() + "] 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) {
```



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

abr 21, 18 16:01 **Broadcast.java** Page 1/2

```java
import Message.*;
import org.apache.log4j.Logger;
import sun.misc.Signal;

import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

public class Broadcast {
    private static final Logger LOGGER = Logger.getLogger(Broadcast.class);
    private static final Settings SETTINGS = Settings.from("broadcast.properties");

    private static final String RABBITMQ_HOST   = SETTINGS.get("RABBITMQ_HOST", "localhost");
    private static final int RABBITMQ_PORT      = SETTINGS.get("RABBITMQ_PORT", 5672);
    private static final String RADIO_QUEUE     = SETTINGS.get("RADIO_QUEUE","RADIO");
    private static final int MESSAGE_EXPIRATION_SECONDS = SETTINGS.get("MESSAGE_EXPIRATION_SECONDS", 30);

    private String consumerRadioTag;

    private final CommunicationWrapper communication;

    private DB db;

    Broadcast() throws Exception {
        communication = CommunicationWrapper.getConnection(RABBITMQ_HOST,RABBITMQ_PORT);
        if (communication ≡ null) {
            LOGGER.fatal("Cannot open communication");
            throw new Exception("Cannot open communication");
        }

        if (¬communication.queueDeclare(RADIO_QUEUE)) {
            LOGGER.fatal("Cannot declare queue " + RADIO_QUEUE);
            communication.close();
            throw new Exception("Cannot declare queue " + RADIO_QUEUE);
        }

        db = new BlockDatabase();
    }

    private void registerSIGINT() throws InterruptedException {
        Semaphore semaphore = new Semaphore(0);
        Runtime.getRuntime().addShutdownHook(new Thread(() → {
                LOGGER.info("SIGINT detected. Closing Broadcast");
                communication.detach(consumerRadioTag);
                communication.close();
                LOGGER.info("Broadcast closed");
                semaphore.release();
                if (LOGGER.isDebugEnabled()) {
                    try {
                        TimeUnit.SECONDS.sleep(1);
                    } catch (InterruptedException ignored) {}
                }
            })
        );
        semaphore.acquire();
    }

    void start() {
        LOGGER.info("Waiting Radio message");

        consumerRadioTag = communication.append(RADIO_QUEUE, message → {
            if (message.getType() ≡ MessageType.RADIO_PACKAGE) {
```

abr 21, 18 16:01 **Broadcast.java** Page 2/2

```java
                db.addStation(message.getRadio());
                for (String userQueue : db.getUsersInStation(message.getRadio()) ) {
                    communication.put(userQueue, message, MESSAGE_EXPIRATION_SECONDS);
                }
            } else if (message.getType() ≡ MessageType.END_TRANSMISSION) {
                Message messageEnd = new MessageBuilder()
                        .setType(MessageType.END_CONNECTION)
                        .build();
                for (String userQueue : db.getUsersInStation(message.getRadio()) ) {
                    LOGGER.info(String.format("Send end transmission (%s) to user(queue) %s", message.getRadio(), userQueue));
                    communication.put(userQueue, messageEnd, MESSAGE_EXPIRATION_SECONDS);
                }
                db.deleteStation(message.getRadio());
                LOGGER.info("Deleted station " + message.getRadio());
            } else {
                LOGGER.warn("Unhandled message with type: " + message.getType());
            }
        });


        try {
            registerSIGINT();
        } catch (InterruptedException e) {
            LOGGER.warn("Interrupt signal");
        }
        LOGGER.info("Exiting");
    }

    public static void main(String[] argv) {
        Broadcast broadcast = null;
        try {
            broadcast = new Broadcast();
        } catch (Exception e) {
            LOGGER.info("Cannot start broadcast");
            LOGGER.debug(e);
            System.exit(1);
        }
        broadcast.start();
    }
}
```

```java
import Message.*;
import org.apache.log4j.Logger;
import sun.misc.Signal;

import java.io.IOException;
import java.util.Timer;
import java.util.TimerTask;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;

public class Admin {
    private static final Logger LOGGER = Logger.getLogger(Admin.class);
    private static final Settings SETTINGS = Settings.from("admin.properties");

    private static final String RABBITMQ_HOST       = SETTINGS.get("RABBITMQ_HOST","localhost");
    private static final int RABBITMQ_PORT          = SETTINGS.get("RABBITMQ_PORT",5672);
    private static final String ADMIN_REQ_QUEUE     = SETTINGS.get("ADMIN_REQUEST_QUEUE","ADMIN_REQUEST");
    private static final String ADMIN_RES_QUEUE     = SETTINGS.get("ADMIN_RESPONSE_QUEUE","ADMIN_RESPONSE");
    private static final int REQUEST_POLL_SECONDS   = SETTINGS.get("REQUEST_POLL_SECONDS",10);
    private static final int POOL_SIZE              = SETTINGS.get("POOL_SIZE",5);

    private CommunicationWrapper communication;

    private AtomicBoolean isConnected;
    private String consumerTag;

    private Admin() throws IOException {
        communication = CommunicationWrapper.getConnection(RABBITMQ_HOST,RABBITMQ_PORT);
        if (communication == null) {
            LOGGER.error("Cannot get connection");
            throw new IOException("Cannot connect");
        }
        isConnected = new AtomicBoolean(false);
        consumerTag = null;
    }

    private Timer startScheduledRequests() {
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(new TimerTask() {
            @Override
            public void run() {
                Message statsRequest = new MessageBuilder()
                        .setType(MessageType.ADMIN_REQUEST_STATS)
                        .build();
                communication.put(ADMIN_REQ_QUEUE, statsRequest, REQUEST_POLL_SECONDS);
                LOGGER.info("Sent stats request");
            }
        }, 0, (int)TimeUnit.SECONDS.toMillis(REQUEST_POLL_SECONDS));

        return timer;

    }

    private String startResponseListener() {
        return communication.append(ADMIN_RES_QUEUE, res -> {
```



```java
            isConnected.set(true);
            LOGGER.info("Receive message from " + ADMIN_RES_QUEUE);
            if (res.getType() == MessageType.ADMIN_RESPONSE_STATS) {
                System.out.println(res.getInfo());
            } else {
                LOGGER.warn("Unhandled message type");
            }
        });
    }

    private void start() throws InterruptedException {
        LOGGER.info("Init admin-client");

        LOGGER.info("Starting admin-scheduler collector");

        Timer schedule = startScheduledRequests();

        consumerTag = startResponseListener();

        Semaphore semaphore = new Semaphore(0);
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
                    LOGGER.info("SIGINT detected. Closing Admin-Handler");
                    schedule.cancel();
                    communication.detach(consumerTag);
                    communication.close();
                    LOGGER.info("Admin-Handler closed");
                    semaphore.release();
                    if (LOGGER.isDebugEnabled()) {
                        try {
                            TimeUnit.SECONDS.sleep(1);
                        } catch (InterruptedException ignored) {}
                    }
                }
        ));
        semaphore.acquire();
        LOGGER.info("Exiting");
    }

    public static void main(String[] strings) {
        try {
            Admin admin = new Admin();
            admin.start();
        } catch (IOException | InterruptedException e) {
            LOGGER.fatal(e);
            LOGGER.warn("Error. Closed admin");
        }
    }
}
```

```java
import Message.*;
import org.apache.log4j.Logger;
import sun.misc.Signal;

import java.util.List;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

public class AdminHandler {
    private static final Logger LOGGER = Logger.getLogger(AdminHandler.class);
    private static final Settings SETTINGS = Settings.from("admin-handler.properties");


    private static final String RABBITMQ_HOST       = SETTINGS.get("RABBITMQ_HOST","localhost");
    private static final int RABBITMQ_PORT          = SETTINGS.get("RABBITMQ_PORT",5672);
    private static final String ADMIN_REQ_QUEUE     = SETTINGS.get("ADMIN_REQUEST_QUEUE","ADMIN_REQUEST");
    private static final String ADMIN_RES_QUEUE     = SETTINGS.get("ADMIN_RESPONSE_QUEUE","ADMIN_RESPONSE");
    private static final int COUNT_TOP_USERS        = SETTINGS.get("COUNT_TOP_USERS",10);

    private final CommunicationWrapper communication;

    private DB db;

    AdminHandler() throws Exception {
        communication = CommunicationWrapper.getConnection(RABBITMQ_HOST,RABBITMQ_PORT);
        if (communication == null) {
            LOGGER.fatal("Cannot open communication");
            throw new Exception("Cannot open communication");
        }

        db = new BlockDatabase();
    }

    private String generatePrintableStats() {
        List<String> topUsers = db.getTopUsers(COUNT_TOP_USERS);
        List<String> usersPerStations = db.getCountUserPerStation();
        StringBuilder stats = new StringBuilder("Number of users per station");
        for (String userCount : usersPerStations) {
            stats.append("\n\t-").append(userCount);
        }
        stats.append("\n\nTop users (minutes)");
        for (String user : topUsers) {
            stats.append("\n\t-").append(user);
        }
        return stats.toString();

    }

    void start() {
        LOGGER.info("Waiting admin request");

        String consumerTag = communication.append(ADMIN_REQ_QUEUE,req -> {
            Message stats = new MessageBuilder()
                    .setType(MessageType.ADMIN_RESPONSE_STATS)
                    .setInfo(generatePrintableStats())
                    .build();
            communication.put(ADMIN_RES_QUEUE, stats);
        });

```



```java

        Semaphore semaphore = new Semaphore(0);
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
                LOGGER.info("SIGINT detected. Closing Admin-Handler");
                communication.detach(consumerTag);
                communication.close();
                LOGGER.info("Admin-Handler closed");
                semaphore.release();
                if (LOGGER.isDebugEnabled()) {
                    try {
                        TimeUnit.SECONDS.sleep(1);
                    } catch (InterruptedException ignored) {}
                }
            })
        );
        try {
            semaphore.acquire();
        } catch (InterruptedException e) {
            LOGGER.warn("Interrupt signal");
            LOGGER.debug(e);
        }
        LOGGER.info("Exiting");
    }

    public static void main(String[] argv) {
        try {
            AdminHandler adminHandler = new AdminHandler();
            adminHandler.start();
        } catch (Exception e) {
            LOGGER.info("Cannot start Admin Handler");
            LOGGER.debug(e);
        }
    }
}
```

abr 26, 18 16:14 **Table of Content** Page 1/1

Table of Contents