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

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

    private void registerSIGINT() {
        Signal.handle(new Signal("INT"), sig → {
                    LOGGER.info("SIGINT detected. Closing connection");
                    communication.detach(consumerClientTag);
                    communication.close();
                    LOGGER.info("Connection closed");
                }
        );
    }

    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())) {
                LOGGER.info("Accepted connection from " + request.getUser() + " in radio " + request.getRadio());
                db.addUserInRadio(request.getUser(), request.getUserQueue(), request.getRadio());
                messageBuilder.setType(MessageType.CONNECTION_ACCEPTED);
                //communication.put(request.getUserQueue(), response);
            } 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());
```

```
                //communication.put(request.getUserQueue(), response);
            }
        } 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");
            //communication.put(request.getUserQueue(), response);
        }
        return messageBuilder.build();
    }



    private void start() {
        LOGGER.info("Client message 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());
            } else if (message.getType() ≡ MessageType.END_CONNECTION) {
                db.deleteUserFromRadio(message.getUser(), message.getUserQueue(), message.getRadio());
            } else {
                LOGGER.warn("Unhandled Message with type " +  message.getType());
            }
        });

        registerSIGINT();
    }

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

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

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Arrays;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;

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

    private static final String RADIO_QUEUE = SETTINGS.get("RADIO_QUEUE","RADIO");
    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 int MESSAGE_EXPIRATION_TIME_SECONDS = SETTINGS.get("MESSAGE_EXPIRATION_TIME_SECONDS",60);
    private static final int PACKAGE_BYTE_SIZE = SETTINGS.get("PACKAGE_BYTE_SIZE",192000);
    private static final int TIME_PER_PACKAGE_SECONDS = SETTINGS.get("TIME_PER_PACKAGE_SECONDS",2);

    private final CommunicationWrapper communication;
    private final String name;
    private AtomicBoolean transmissionStarted;

    private Station(String name) throws Exception {
        communication = CommunicationWrapper.getConnection(RABBITMQ_HOST, RABBITMQ_PORT);
        if (communication ≡ null) {
            LOGGER.warn("Cannot establish communication");
            throw new Exception("Cannot establish communication");

        }

        if (¬communication.queueDeclare(RADIO_QUEUE)) {
            LOGGER.warn("Cannot declare Queue " + RADIO_QUEUE);
            throw new Exception("Cannot declare Queue " + RADIO_QUEUE);
        }

        this.name = name;
        this.transmissionStarted.set(false);

    }

    private void startTransmission(String filePath) throws InterruptedException {
        File file = new File(filePath);
        try (FileInputStream fis = new FileInputStream(file)) {
            String fileExtension = filePath.substring(filePath.lastIndexOf(’.’) + 1);
            int totalBytes = fis.available();
            LOGGER.info("Attempt to send " + file.getName() + ". Content type " + fileExtension);
            LOGGER.info("Total bytes to send " + totalBytes);
            int byteCountRead = 0;
            int byteCount;
            byte[] bytes = new byte[PACKAGE_BYTE_SIZE];
            transmissionStarted.set(true);
            while ((byteCount = fis.read(bytes)) ≠ −1) {
                byteCountRead+= byteCount;
```

```java
                Message message = new MessageBuilder()
                        .setType(MessageType.RADIO_PACKAGE)
                        .setRadio(name)
                        .setContentType(fileExtension)
                        .setPayload(Arrays.copyOfRange(bytes,0,byteCount))
                        .build();
                communication.put(RADIO_QUEUE, message,MESSAGE_EXPIRATION_TIME_SECONDS);
                LOGGER.info("Sent " + byteCount + " bytes. " + (byteCountRead*100)/totalBytes + "%");

                TimeUnit.SECONDS.sleep(TIME_PER_PACKAGE_SECONDS);
            }
            LOGGER.info("End stream");
        } catch (IOException e) {
            LOGGER.debug(e);
            LOGGER.warn("Cannot read file stream " + filePath);
        }
    }

    private void stopTransmission() {
        LOGGER.info("Closing connection");
        if (transmissionStarted.compareAndSet(true, false)) {
            Message endMessage = new MessageBuilder().setType(MessageType.END_TRANSMISSION).build();
            communication.put(RADIO_QUEUE, endMessage,MESSAGE_EXPIRATION_TIME_SECONDS);
            communication.close();
        }
    }

    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Use: ./station <<name>> <<file>>");
            LOGGER.fatal("Invalid parameters");
            return;
        }

        try {
            Station station = new Station(args[0]);
            station.startTransmission(args[1]);
            station.stopTransmission();
        } catch (Exception e) {
            LOGGER.debug(e);
            LOGGER.warn("Cannot create station");
        }
    }
}
```

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

import java.io.*;
import java.util.Properties;

public class Settings {
    private static final Logger LOGGER = Logger.getLogger(Settings.class);
    private final Properties properties;
    private Settings() {
        properties = new Properties();
    }

    public static Settings from(String propertiesFile) {
        InputStream input = null;
        Settings settings = new Settings();
        try {
            input = new FileInputStream(propertiesFile);
            settings.properties.load(input);
            LOGGER.info(String.format("\"%s\" was loaded correctly", propertiesFile));
        } catch (IOException ex) {
            LOGGER.error(String.format("Cannot load \"%s\" using all default values",propertiesFile));
        } finally {
            if (input ≠ null) {
                try {
                    input.close();
                } catch (IOException e) {
                    LOGGER.warn("IOException when attempt to close " + propertiesFile);
                    LOGGER.debug(e);
                }
            }
        }
        return settings;
    }

    public int get(String name, int defaultValue) {
        try {
            return Integer.parseInt(properties.getProperty(name, String.valueOf(defaultValue)));
        } catch (NumberFormatException e) {
            LOGGER.warn("Invalid Int value " + properties.getProperty(name) + " of property: " + name + ". Return default");
            return defaultValue;
        }
    }

    public Boolean get(String name, boolean defaultValue) {
        return Boolean.valueOf(properties.getProperty(name, String.valueOf(defaultValue)));
    }

    public String get(String name, String defaultValue) {
        return properties.getProperty(name, defaultValue);
    }
}
```

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

    private ScheduledExecutorService keepAliveScheduler;


    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;
    }

    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");
            }
        }
        return fileStream;
    }

    private void handleRadioPackage(Message message) {
        FileOutputStream out;
        try {
            File fileStream = createFile(message.getContentType());
            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();
        }
    }

    private void startSchedulerToSendKeepAlive() {
        keepAliveScheduler.scheduleAtFixedRate(() → {
            LOGGER.info("Send keep alive to server");
            Message message = new MessageBuilder()
                    .setType(MessageType.KEEP_ALIVE)
                    .setUser(user)
                    .build();
```

```
            comm.put(CLIENT_QUEUE, message);
        }, KEEP_ALIVE_POLL_SECONDS, KEEP_ALIVE_POLL_SECONDS, TimeUnit.SECONDS);
    }

    private void disconnect() {
        comm.put(CLIENT_QUEUE, new MessageBuilder()
                .setUser(user)
                .setRadio(radio)
                .setType(MessageType.END_CONNECTION).build()
        );
    }

    private synchronized void stop() {
        LOGGER.info("Stop Radio Listener");
        if (isConnected.compareAndSet(true, false)) {
            disconnect();
        }
        if (keepAliveScheduler ≠ null) {
            keepAliveScheduler.shutdownNow();
        }
        if (¬comm.detach(consumerTag)) {
            LOGGER.warn("Cannot detach");
        }
        comm.deleteQueue(listenQueue);
        comm.close();
        LOGGER.info("Exit");
    }

    private void waitResponseWithTimeout(Consumer<Message> handler) {
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1)
;
        LOGGER.info("Start timeout to wait response. TIMEOUT=" + TIMEOUT_SECONDS);
        scheduler.schedule(() →
                LOGGER.info("TIMEOUT: Wake−up. Close connection. The servers are not working"),
                TIMEOUT_SECONDS,
                TimeUnit.SECONDS
        );

        // Wait responses
        LOGGER.info("Waiting response");
        consumerTag = comm.append(listenQueue, res → {
            handler.accept(res);
            scheduler.shutdownNow();
        });


        scheduler.shutdown();
        try {
            scheduler.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);
        } catch (InterruptedException ignored) {
            LOGGER.info("Interrupted Exception");
        }
    }

    private void startListener() {

        Message request = new MessageBuilder().setType(MessageType.REQUEST_CONNE
CTION)
                .setClientQueue(listenQueue)
                .setRadio(radio)
                .setUser(user)
                .build();
        comm.put(CLIENT_QUEUE, request, TIMEOUT_SECONDS);

        waitResponseWithTimeout(this::handleResponse);

        if (¬isConnected.get()) {
            stop();
        }
    }

    private void listRadios() {

        // Send request
        Message request = new MessageBuilder().
                setType(MessageType.REQUEST_RADIOS)
                .setClientQueue(listenQueue)
                .build();
        comm.put(CLIENT_QUEUE, request, TIMEOUT_SECONDS);


        waitResponseWithTimeout(res → {
            LOGGER.info("Received response from server");
            if (res.getType() ≡ MessageType.RESPONSE_RADIOS) {
                System.out.println("Radios:\n" + res.getInfo());
            }
        });

        stop();

    }

    private void start() {
        if (this.user ≡ null) {
            listRadios();
        } else {
            startListener();
        }
    }

    public static void main(String[] args) {
        if (args.length ≡ 0) {
            System.out.println(" − Listen Radio: ./radio−listener <<user>> <<radio>>");
            System.out.println(" − List Radios: ./radio−listener list");
            return;
        }

        boolean listRadios = (args.length ≡ 1 ∧ args[0].toLowerCase().equals("list
"));
        try {
            RadioListener radioListener = (listRadios) ? new RadioListener() : n
ew RadioListener(args[0], args[1]);
            radioListener.start();
            Signal.handle(new Signal("INT"), sig → {
                LOGGER.info("SIGINT detected. Closing connection");
                radioListener.stop();
                LOGGER.info("Connection closed");
            });
        } catch (IOException e) {
            LOGGER.fatal("Cannot start radio−listener");
            LOGGER.debug(e);
        }

    }


}
```

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

public class Initializer {
    private static final Logger LOGGER = Logger.getLogger(Initializer.class);
    private static final Settings SETTINGS = Settings.from("initializer.properties");
    public static void main(String[] args) {
        LOGGER.info("Initialize all queues to use");
        CommunicationWrapper comm = CommunicationWrapper.getConnection(
                SETTINGS.get("RABBITMQ_HOST","localhost"),
                SETTINGS.get("RABBITMQ_PORT",5672)
        );
        if (comm ≡ null) {
            LOGGER.fatal("Cannot connect. Abort initializer");
            return;
        }

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

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

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

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

        comm.close();
        LOGGER.info("OK. Initializer");

    }
}
```

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

```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);
        } catch (IOException | IndexOutOfBoundsException e) {
            e.printStackTrace();
        }


    }

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

abr 18, 18 21:41 **Destructor.java** Page 1/1

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

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");
            DB.cleanDatabases();
        }

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

```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;
import java.util.stream.Collector;
import java.util.stream.Collectors;

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');
```

```java
        return indexOfEnd ≠ -1 ? str.substring(0, str.indexOf(NULL)) : str;
    }

    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(), b
lockSize, false);
                out.write(block);
            lock.release();
        } catch (IOException e) {
            LOGGER.error(String.format("Cannot write 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(), StandardOpenOptio
n.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;
            }
        } catch (IOException e) {
            LOGGER.error(String.format("Cannot write file '%s'",file.toString()));
            LOGGER.debug(e);
        }
    }

    public void iterFile(final Consumer<String> handleBlock) {
        // Iter file. Pass handler to handled al String per block
        iterFile((pos, string) → {
            handleBlock.accept(string);
            return false;
        });
    }

    private int getPosition(String s, Comparator<String> comparator) {
        AtomicInteger position = new AtomicInteger(−1);
        iterFile((pos, string) → {
            if ( comparator.compare(s, string) ≡ 0) {
                position.set(pos);
                return true;
            }
            return false;
        });
        return position.get();
    }

    public void update(Function<String, String> updater) {
        // Search S in all blocks.
        // If exist (comparator return true), then update it.
        // If no exist, write on end.
        Map<Integer, String> newPositionsStrings = new HashMap<>();
        iterFile((pos, string) → {
            String newValue = updater.apply(string);
            if (string.equalsIgnoreCase(newValue)) {
                newPositionsStrings.put(pos, newValue);
            }
            return false;
        });
        newPositionsStrings.entrySet().forEach(e→ writeBlock(e.getValue(), e.getKey()));

    }

    public void insert(String s) {
        // Write block in first free position.
        // Is no exist free position. write in the end
        String nullString = toString(generateNullBlock());
        writeBlock(s, getPosition(nullString, String::compareTo));
    }

    public void delete(String s, Comparator<String> comparator) {
        // Delete block who match with s
        // Ignoring if not exist
        String nullString = toString(generateNullBlock());
        int position = getPosition(s, comparator);
        if (position ≠ −1) {
            writeBlock(nullString, position);
        }
    }

    public List<String> find(Predicate<String> comparator) {
        List<String> collect = new ArrayList<>();
        iterFile((pos, string) → {
            if (comparator.test(string)) {
                collect.add(string);
            }
            return false;
        });
        return collect;
    }
}
```

```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.concurrent.atomic.AtomicBoolean;
import java.util.stream.Collectors;

@SuppressWarnings("unchecked")
public class DB {
    private static final Logger LOGGER = Logger.getLogger(DB.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);

    DB() 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 static void cleanDatabases() {
        try {
            DB db = new DB();
            String[] DBNames = {STATIONS_DB, USERS_DB, CONNECTIONS_DB};
            for (String DBName: DBNames) {
                if (db.writeJSON(new JSONObject(), DBName)) {
                    LOGGER.info(DBName + " clean");
                } else {
                    LOGGER.warn("Cannot clean " + DBName);
                }
            }
        } catch (IOException e) {
            LOGGER.warn("Cannot clean databases");
        }

    }

    private synchronized boolean writeJSON(JSONObject json, String fileName) {
        try (FileWriter file = new FileWriter(WORKING_DIR + fileName)) {
            file.write(json.toJSONString());
            file.close();
            return true;
        } catch (IOException e) {
            LOGGER.warn("Cannot write " + WORKING_DIR + fileName);
            LOGGER.debug(e);
        }
        return false;
    }

    private synchronized JSONObject readJSON(String fileName) {
        JSONParser parser = new JSONParser();

        try {
            Object obj = parser.parse(new FileReader(WORKING_DIR + fileName));
            return (JSONObject) obj;

        } catch (IOException | ParseException e) {
            LOGGER.warn("Cannot read " + WORKING_DIR + fileName);
            LOGGER.debug(e);
        }
        return null;
    }

    public void addUserInRadio(String userName, String userQueue, String radio) {
        JSONObject stations = readJSON(STATIONS_DB);

        if (stations ≡ null) {
            return;
        }
        JSONArray userQueues = stations.containsKey(radio) ? (JSONArray) stations.get(radio) : new JSONArray();
        userQueues.add(userQueue);

        stations.put(radio, userQueues);

        if (writeJSON(stations, STATIONS_DB)) {
            JSONObject users = readJSON(USERS_DB);
            if (users ≡ null) {
                return;
            }
            Long count = users.containsKey(userName) ? (Long) users.get(userName) : 0;
            users.put(userName, count + 1);
            if (writeJSON(users, USERS_DB)) {
                LOGGER.info("Added user '" + userName + "' to radio " + radio);
            }
        }
    }

    public void deleteUserFromRadio(String userName, String userQueue, String radio) {
```

```
        JSONObject stations = readJSON(STATIONS_DB);

        if (stations ≡ null ∨ ¬stations.containsKey(radio)) {
            return;
        }

        JSONArray userQueues = (JSONArray) stations.get(radio);
        JSONArray userQueuesNew = new JSONArray();
        for(Object queue: userQueues){
            if (queue instanceof String) {
                if (¬((String) queue).equalsIgnoreCase(userQueue)) {
                    userQueuesNew.add(queue);
                }
            }
        }
        stations.put(radio, userQueuesNew);

        if (writeJSON(stations, STATIONS_DB)) {
            JSONObject users = readJSON(USERS_DB);
            if (users ≡ null) {
                return;
            }
            Long count = users.containsKey(userName) ? (Long) users.get(userName) : 1;
            users.put(userName, count - 1);
            if (writeJSON(users, USERS_DB)) {
                LOGGER.info("Delete user '" + userName + "' in radio " + radio);
            }
        }
    }

    public boolean existStation(String radio) {
        JSONObject stations = readJSON(STATIONS_DB);
        if (stations ≡ null) {
            return false;
        }
        if (¬stations.containsKey(radio)) {
            return false;
        }
        return true;
    }

    public boolean userCanHearRadio(String userName, String radio) {


        JSONObject users = readJSON(USERS_DB);
        if (users ≡ null) {
            return false;
        }
        if (users.containsKey(userName)) {
            Long radioCount = (Long) users.get(userName);
            return radioCount < MAX_RADIOS_PER_CLIENT;
        }
        return true;

    }

    public void updateUserActivity(String userName) {
        JSONObject usersActivity = readJSON(CONNECTIONS_DB);
        if (usersActivity ≡ null) {
            return;
        }
        Timestamp timestamp = new Timestamp(System.currentTimeMillis());
        if (usersActivity.containsKey(userName)) {
            JSONObject userActivity = (JSONObject) usersActivity.get(userName);
            Long lastTimeStamp = (Long) userActivity.get("last");
```

```java
            userActivity.put("last", timestamp);
            if (timestamp.getTime() − lastTimeStamp > OFFSET_TIMESTAMP) {
                userActivity.put("total", (Long)userActivity.get("total") + 1);
            } else {
                userActivity.put("total", (Long)userActivity.get("total") + (timestamp.getTime() − lastTimeStamp));
            }
            usersActivity.put(userName, userActivity);
        } else {
            JSONObject userActivity = new JSONObject();
            userActivity.put("last", timestamp.getTime());
            userActivity.put("total", 1);
            usersActivity.put(userName, userActivity);
        }
        if (writeJSON(usersActivity, CONNECTIONS_DB)) {
            LOGGER.info("Update activity for user " + userName);
        }

    }

    public List<String> getStations() {
        ArrayList<String> stationsArray = new ArrayList<>();
        JSONObject stations = readJSON(STATIONS_DB);
        if (stations ≡ null) {
            return stationsArray;
        }
        stationsArray.addAll(stations.keySet());
        return stationsArray;
    }

    public List<String> getTopUsers(int count) {
        JSONObject users = readJSON(CONNECTIONS_DB);
        if (users ≡ null) {
            return new ArrayList<>();
        }
        return (List<String>) users.keySet().stream()
                .sorted( (u1,u2) → {
                    JSONObject user1 = (JSONObject) users.get(u1);
                    JSONObject user2 = (JSONObject) users.get(u2);
                    Long total1 = (Long) user1.get("total");
                    Long total2 = (Long) user2.get("total");
                    return −total1.compareTo(total2);
                })
                .limit(count)
                .map(userName → userName + "|total: " + ((JSONObject)users.get(userName)).get("total") + " sec.")
                .collect(Collectors.toList());
    }

    public List<String> getUsersInStation(String station) {
        List<String> userQueue = new ArrayList<>();
        JSONObject stations = readJSON(STATIONS_DB);
        if (stations ≡ null ∨ ¬stations.containsKey(station)) {
            return userQueue;
        }
        JSONArray usersQueue = (JSONArray) stations.get(station);
        for (Object queue : usersQueue) {
            userQueue.add((String)queue);
        }
        return userQueue;
    }

    public void addStation(String station) {
        JSONObject stations = readJSON(STATIONS_DB);
        if (stations ≡ null ∨ stations.containsKey(station)) {
            return;
```

```java
        }
        stations.put(station, new JSONArray());
        if (writeJSON(stations, STATIONS_DB)) {
            LOGGER.info("Added station " + station + " in DB");
        }
    }

    public void deleteStation(String station) {
        JSONObject stations = readJSON(STATIONS_DB);
        if (stations ≡ null ∨ ¬stations.containsKey(station)) {
            return;
        }
        stations.remove(station);
        if (writeJSON(stations, STATIONS_DB)) {
            LOGGER.info("Deleted station " + station + " in DB");
        }
    }

    public List<String> getCountUserPerStation() {
        return new ArrayList<>();
    }
}
```

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

@SuppressWarnings("unchecked")
public class DataBase {
    private static final Logger LOGGER = Logger.getLogger(DataBase.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 HashMap<String,FileCellBlock> DB;

    DataBase() throws IOException {
        createDir(WORKING_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));
        }
    }

    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 static void cleanDatabases() {
        File workingDir = new File(WORKING_DIR);
        for(File file: Objects.requireNonNull(workingDir.listFiles())) {
            if (¬file.isDirectory())
                if (¬file.delete()) {
                    LOGGER.warn("Cannot delete " + file.toString());
```

```java
                }
        }

    }
    /*
    private synchronized boolean writeJSON(JSONObject json, String fileName) {
        try (FileWriter file = new FileWriter(WORKING_DIR + fileName)) {
            file.write(json.toJSONString());
            file.close();
            return true;
        } catch (IOException e) {
            LOGGER.warn("Cannot write " + WORKING_DIR + fileName);
            LOGGER.debug(e);
        }
        return false;
    }

    private synchronized JSONObject readJSON(String fileName) {
        JSONParser parser = new JSONParser();

        try {
            Object obj = parser.parse(new FileReader(WORKING_DIR + fileName));
            return (JSONObject) obj;

        } catch (IOException | ParseException e) {
            LOGGER.warn("Cannot read " + WORKING_DIR + fileName);
            LOGGER.debug(e);
        }
        return null;
    }
    */

    private String stationKey(String radio) {
        return "station." + radio;
    }

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

    public void addUserInRadio(String userName, String userQueue, String radio) {
        // TODO: Add in getStationDB(radio), and update (+1) counter in USERS_DB
    }

    public void deleteUserFromRadio(String userName, String userQueue, String radio) {
        // TODO: Delete from getStationDB(radio), and update counter (-1) in USERS_DB
    }

    public boolean existStation(String radio) {
        return DB.containsKey( stationKey(radio) );
    }

    public boolean userCanHearRadio(String userName, String radio) {

        FileCellBlock users = DB.get(USERS_DB);

        List<String> result = users.find(s → s.contains(userName));
        if (result.isEmpty()) {
            return true;
        }
        // TODO: Number of radios is bigger than result.get(0);
```

```java
            return true;

        }

        public void updateUserActivity(String userName) {
            FileCellBlock connection = DB.get(CONNECTIONS_DB);

            // TODO: Pass function who take value and modify it;
            // Take row. if match modify. and return true. else return false
            // user.update();
        }

        public List<String> getStations() {
            List<String> stations = new ArrayList<>();
            DB.entrySet().stream()
                    .filter(e → e.getKey().matches("station"))
                    .sorted(Comparator.comparing(Map.Entry::getKey))
                    .forEach(e → stations.add(e.getKey()));
            return stations;
        }

        public List<String> getTopUsers(int count) {
            return new ArrayList<>();
        }

        public void addStation(String station) {
            try {
                getStationDB(station);
            } catch (IOException e) {
                LOGGER.warn("Cannot add station " + station);
                LOGGER.debug(e);
            }
        }

        public void deleteStation(String station) {

        }

        public List<String> getCountUserPerStation() {
            return new ArrayList<>();
        }
    }
```

```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;
    }
}
```

```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;
    }

```

```java

    public String getRadio() {
        return json.get("radio").toString();
    }

    public String getError() {
        return json.get("error").toString();
    }

    public String getUserQueue() {
        return json.get("user_queue").toString();
    }

    public String getUser() {
        return json.get("user").toString();
    }

    public String getInfo() {
        return json.get("info").toString();
    }
}

```

```java
package Message;

import java.io.IOException;

public class MessageException extends IOException {
    public MessageException() { super(); }
    public MessageException(String message) { super(message); }
    public MessageException(String message, Throwable cause) { super(message, ca
use); }
    public MessageException(Throwable cause) { super(cause); }
}
```

```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);
    }
}
```

```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;
        }
    }
```

```java

    private boolean put(String queue, Message message, AMQP.BasicProperties props) {
        try {
            channel.basicPublish("", queue, null, message.toBytes());
            LOGGER.debug("Send message [" + message.toString().hashCode() + "] in queue " + queue);
        } catch (IOException e) {
            LOGGER.warn("Cannot put message in " + queue + "." + e.getMessage());
            return false;
        }
        return true;
    }

    boolean put(String queue, Message message, int expiration_seconds) {
        AMQP.BasicProperties props = new AMQP.BasicProperties.Builder()
                .expiration(String.valueOf(expiration_seconds * 1000))
                .build();
        return put(queue, message, props);
    }

    boolean put(String queue, Message message) {
        return put(queue, message, Integer.MAX_VALUE);
    }

    String append(String queue, Consumer<Message> handlerFunction) {
        try {
            return channel.basicConsume(queue, false, new DefaultConsumer(channel) {
                @Override
                public void handleDelivery(String consumerTag, Envelope env, AMQP.BasicProperties props, byte[] body) {
                    try {
                        Message message = new Message(body);
                        LOGGER.debug("Receive message [" + message.toString().hashCode() + "] from queue " + queue);
                        handlerFunction.accept(message);
                        channel.basicAck(env.getDeliveryTag(), false);
                    } catch (IOException  e) {
                        LOGGER.debug(e);
                    }
                }
            });
        } catch (IOException e) {
            LOGGER.error("Error on append in " + queue);
            LOGGER.debug(e);
            return null;
        }
    }

    boolean detach(String consumerTag) {
        try {
            channel.basicCancel(consumerTag);
            return true;
        } catch (IOException e) {
            LOGGER.warn("Cannot detach consumerTag: " + consumerTag);
            LOGGER.debug(e);
            return false;
        }
    }

    void deleteQueue(String queueName) {
        try {
            channel.queueDeleteNoWait(queueName, false, false);
            LOGGER.info("Deleted queue " + queueName);
        } catch (IOException e) {
            LOGGER.warn("Cannot delete queue: " + queueName);
            LOGGER.debug(e);
        }
    }



}
```

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

import java.util.concurrent.TimeUnit;

public class Broadcast {
    private static final Logger LOGGER = Logger.getLogger(Broadcast.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 RADIO_QUEUE     = SETTINGS.get("RADIO_QUEUE","R
ADIO");
    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,RABBITM
Q_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 DB();
    }

    private void registerSIGINT() {
        Signal.handle(new Signal("INT"), sig → {
                LOGGER.info("SIGINT detected. Closing Broadcast");
                communication.detach(consumerRadioTag);
                communication.close();
                LOGGER.info("Broadcast closed");
            }
        );
    }

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

        consumerRadioTag = communication.append(RADIO_QUEUE, message → {
            if (message.getType() ≡ MessageType.RADIO_PACKAGE) {
                db.addStation(message.getRadio());
                for (String userQueue : db.getUsersInStation(message.getRadio())
) {
                    communication.put(userQueue, message, MESSAGE_EXPIRATION_SEC
ONDS);
                }
            } else if (message.getType() ≡ MessageType.END_TRANSMISSION) {
                Message messageEnd = new MessageBuilder()
                        .setType(MessageType.END_CONNECTION)
```

```java
                        .build();
                for (String userQueue : db.getUsersInStation(message.getRadio())) {
                    communication.put(userQueue, messageEnd, MESSAGE_EXPIRATION_SECONDS);
                }
                db.deleteStation(message.getRadio());
            } else {
                LOGGER.warn("Unhandled message with type: " + message.getType());
            }
        });


        registerSIGINT();
    }

    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.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

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

    private ScheduledExecutorService startScheduledRequests() {
        ScheduledExecutorService schedule = Executors.newScheduledThreadPool(POOL_SIZE);
        schedule.scheduleAtFixedRate(() → {
            Message statsRequest = new MessageBuilder()
                    .setType(MessageType.ADMIN_REQUEST_STATS)
                    .build();
            communication.put(ADMIN_REQ_QUEUE, statsRequest, REQUEST_POLL_SECONDS);
        }, REQUEST_POLL_SECONDS, REQUEST_POLL_SECONDS, TimeUnit.SECONDS);
        return schedule;

    }

    private String startResponseListener() {
        return communication.append(ADMIN_RES_QUEUE, res → {
            LOGGER.info("Receive message from " + ADMIN_RES_QUEUE);
            if (res.getType() ≡ MessageType.ADMIN_RESPONSE_STATS) {
                LOGGER.info("Message:\n" + res.getInfo());
                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");

        ScheduledExecutorService schedule = startScheduledRequests();

        String consumerTag = startResponseListener();

        Signal.handle(new Signal("INT"), sig → {
                    LOGGER.info("SIGINT detected. Closing Admin−Handler");
                    schedule.shutdownNow();
                    LOGGER.info("Admin−Handler closed");
                }
        );

        schedule.shutdown();
        schedule.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);

        communication.close();
        communication.detach(consumerTag);

        LOGGER.info("Admin−client closed");
    }

    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.Executors;
import java.util.concurrent.ScheduledExecutorService;
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_H
OST","localhost");
    private static final int RABBITMQ_PORT          = SETTINGS.get("RABBITMQ_PO
RT",5672);
    private static final String ADMIN_REQ_QUEUE     = SETTINGS.get("ADMIN_REQU
EST_QUEUE","ADMIN_REQUEST");
    private static final String ADMIN_RES_QUEUE     = SETTINGS.get("ADMIN_RESPO
NSE_QUEUE","ADMIN_RESPONSE");
    private static final int COUNT_TOP_USERS        = SETTINGS.get("COUNT_TOP_U
SERS",10);

    private final CommunicationWrapper communication;

    private DB db;

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

        db = new DB();
    }

    private String generatePrinteableStats() {
        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("Starting admin−scheduler collector");

        communication.append(ADMIN_REQ_QUEUE,req → {
            Message stats = new MessageBuilder()
                    .setType(MessageType.ADMIN_RESPONSE_STATS)
                    .setInfo(generatePrinteableStats())
                    .build();
            communication.put(ADMIN_RES_QUEUE, stats);
        });

        Signal.handle(new Signal("INT"), sig → {
                LOGGER.info("SIGINT detected. Closing Admin−Handler");
                LOGGER.info("Admin−Handler closed");
            }
        );

        communication.close();
    }

    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);
        }
    }
}
```

Table of Contents