Apr 25, 18 21:45 **UserStatisticsViewer.java** Page 1/1

```
import com.google.gson.Gson;
import com.rabbitmq.client.*;

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

public class UserStatisticsViewer extends RabbitMQProcess {

    public UserStatisticsViewer(String host) throws IOException, TimeoutExceptio
n {
        super(host);

        // declare USERS_STATS exchange
        getChannel().exchangeDeclare(Configuration.UsersStatisticsExchange,
                BuiltinExchangeType.FANOUT);
    }

    @Override
    public void run() throws IOException {
        consumeStatistics();
    }

    private String consumeStatistics() throws IOException {
        String statisticsQueue = getChannel().queueDeclare().getQueue();
        getChannel().queueBind(statisticsQueue,
                Configuration.UsersStatisticsExchange, "");

        Consumer consumerStatistics = new DefaultConsumer(getChannel()) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                String json = new String(body, "UTF-8");
                UsersSecondsListenedStatistics statistics = new Gson().fromJson
                        (json, UsersSecondsListenedStatistics.class);

                System.out.println(" [x] Showing users who listened most: ");
                for (UserSecondsListened userStats :
                        statistics.getUsersMostListenedSeconds()) {
                    System.out.println(userStats.getUsername() + ":" +
                            userStats.getSecondsListened());
                }
            }
        };

        // consume de una cola temporal a traves de un exchange
        // por lo que no tiene sentido ack manual
        return getChannel().basicConsume(statisticsQueue, true,
                consumerStatistics);
    }

    public static void main(String[] argv) throws Exception {
        UserStatisticsViewer statisticsViewer =
                new UserStatisticsViewer(Configuration.RabbitMQHost);
        statisticsViewer.run();
    }
}
```

Apr 25, 18 21:04 **UsersSecondsListenedStatistics.java** Page 1/1

```
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;

public class UsersSecondsListenedStatistics {

    private List<UserSecondsListened> usersMostListenedSeconds;

    public UsersSecondsListenedStatistics(Collection<UserSecondsListened> stats)
 {
        this.setUsersMostListenedSeconds(new LinkedList<>(stats));
    }

    public List<UserSecondsListened> getUsersMostListenedSeconds() {
        return usersMostListenedSeconds;
    }

    public void setUsersMostListenedSeconds(List<UserSecondsListened> usersMostL
istenedSeconds) {
        this.usersMostListenedSeconds = usersMostListenedSeconds;
    }
}
```

```java
import java.util.Comparator;


public class UsersSecondsListenedComparator
        implements Comparator<UserSecondsListened> {
    @Override
    public int compare(UserSecondsListened o1, UserSecondsListened o2) {

        return (int)Math.signum(o2.getSecondsListened() -
                o1.getSecondsListened());
    }
}
```

```java
public class UserSecondsListened {

    private String username;
    private long secondsListened;

    public UserSecondsListened(String username, long secondsListened) {
        this.setUsername(username);
        this.setSecondsListened(secondsListened);
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public long getSecondsListened() {
        return secondsListened;
    }

    public void setSecondsListened(long secondsListened) {
        this.secondsListened = secondsListened;
    }
}
```

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.*;
import java.util.concurrent.TimeoutException;

public class UsersDBHandler extends DBHandlerWithStatistics<User> {

    private String queueName;

    public UsersDBHandler(String host, Database<User> database,
                          List<String> masks)
            throws IOException, TimeoutException {
        super(host, database);

        // declare USERS_DB exchange
        getChannel().exchangeDeclare(Configuration.UsersDBExchange,
                BuiltinExchangeType.TOPIC);

        // declare USERS_STATS exchange
        getChannel().exchangeDeclare(Configuration.UsersStatisticsExchange,
                BuiltinExchangeType.FANOUT);

        this.queueName = Configuration.UsersDBExchange + "_" +
                Configuration.maskListToStr(masks);
        getChannel().queueDeclare(queueName, true, false, false, null);
        for (String mask : masks) {
            getChannel().queueBind(queueName,
                    Configuration.UsersDBExchange, mask);
        }
    }

    @Override
    public void run() throws IOException {

        Consumer usersConsumer = new DefaultConsumer(getChannel()) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                // parse request
                String jsonRequest = new String(body, "UTF-8");

                DatabaseRequest request = new Gson().fromJson(jsonRequest,
                        DatabaseRequest.class);

                if (request.getType() == Configuration.UsersTypeConnect) {
                    consumeConnection(request.getSerializedRequest());
                } else if (request.getType() ==
                        Configuration.UsersTypeDisconnect) {
                    consumeDisconnection(request.getSerializedRequest());
                } else if (request.getType() ==
                        Configuration.UsersTypeKeepAlive) {
                    consumeKeepAlive(request.getSerializedRequest());
                } else {
                    Logger.output("Invalid request type received: " +
                    request.getType() + ", request: " +
                            request.getSerializedRequest());
                }

                getChannel().basicAck(envelope.getDeliveryTag(), false);
            }
        };

        getChannel().basicConsume(queueName, false, usersConsumer);
    }

    @Override
    protected List<StatisticTask> getStatistics() {
        List<StatisticTask> operations = new LinkedList<>();

```

```java
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                // get statistics
                LimitedSortedSet<UserSecondsListened> usersMostListened =
                        new LimitedSortedSet<>(Configuration.UserStatisticsN,
                                new UsersSecondsListenedComparator());
                for (User row : database.getRows()) {
                    UserSecondsListened userStats =
                            new UserSecondsListened(row.username,
                                    row.secondsListening);
                    usersMostListened.add(userStats);
                }

                UsersSecondsListenedStatistics stats =
                        new UsersSecondsListenedStatistics(usersMostListened);
                String jsonStats = new Gson().toJson(stats);

                try {
                    getChannel().basicPublish(
                            Configuration.UsersStatisticsExchange, "", null,
                            jsonStats.getBytes());
                } catch (IOException e) {
                    Logger.output("IOEXception during statistics publish");
                }
            }
        };

        operations.add(new StatisticTask(runnable,
                Configuration.UsersStatisticsPeriodSeconds));

        return operations;
    }

    public void consumeConnection(String jsonRequest) throws IOException {

        UserConnectRequest request = new Gson().fromJson(jsonRequest,
                UserConnectRequest.class);
        UserConnectResponse response = new UserConnectResponse(request);

        // get user record from DB
        User user = database.getRow(request.getUsername());
        if (user == null) {
            System.out.println("User: " + request.getUsername() + " not " +
                    "found, creating new user");
            user = new User(request.getUsername());
        }
        // first check if any connection is not active and remove it
        int connectionId = 0;
        Date now = new Date();
        ListIterator<UserRadioConnection> connIter =
                user.connections.listIterator();
        while(connIter.hasNext()){
            UserRadioConnection connection = connIter.next();

            Date then = connection.keepAlive;
            if (now.getTime() - then.getTime() >
                    Configuration.SecondsUntilDropConnection * 1000) {
                UserDisconnectRequest disconnectRequest = new
                        UserDisconnectRequest(request.getUsername(),
                        connection.radio,
                        connection.connectionID);
                response.getClosedConnections().add(disconnectRequest);
                connIter.remove();

                System.out.println(" [x] Closing old connection to: " +
                        connection.radio);
            } else {
                if (connection.connectionID > connectionId) {
                    connectionId = connection.connectionID;
                }
            }
        }
        connectionId = (connectionId + 1) %
                Configuration.MaxConnectionsPerUnlimitedUser;
        // then check if user can connect to radio
        if (user.connections.size() < user.connectionsLimit) {
            UserRadioConnection connection =
                    new UserRadioConnection(response.getRadio(),
                            new Date(), connectionId);
            user.connections.add(connection);
            response.setCouldConnect(true);
            response.setConnectionId(connectionId);

            System.out.println(" [x] User: " + user.username +
                    " connected to: " + request.getRadio());
        } else {
            response.setCouldConnect(false);

            System.out.println(" [x] User: " + user.username +
                    " not connected to: " + request.getRadio());
        }
        // update user
        database.updateRow(user);

        String jsonResponse = new Gson().toJson(response);
        getChannel().basicPublish("",
                Configuration.ConnMgrUsersDBResponseQueue, null,
                jsonResponse.getBytes());
    }

    public void consumeDisconnection(String jsonRequest) throws IOException {

        UserDisconnectRequest request = new Gson().fromJson(jsonRequest,
                UserDisconnectRequest.class);

        // get user record from DB
        User user = database.getRow(request.getUsername());
        if (user ≠ null) {
            ListIterator<UserRadioConnection> connIter =
                    user.connections.listIterator();
            while(connIter.hasNext()){
                UserRadioConnection connection = connIter.next();
                if (connection.connectionID ≡ request.getConnectionId() ∧
                        connection.radio.equals(request.getRadio())) {
                    connIter.remove();
                    System.out.println(" [x] Removing connection " +
                            "from: " + user.username + " to radio: "
                            + connection.radio);
                    break;
                }
            }
        }
        // update user
        database.updateRow(user);
    }

    public void consumeKeepAlive(String jsonRequest) throws IOException {

        KeepAliveRequest request = new Gson().fromJson(jsonRequest,
                KeepAliveRequest.class);

        // get user record from DB
        User user = database.getRow(request.getUsername());
        if (user ≡ null) {
            System.out.println(" [x] Error: user who sent keep alive " +
                    "does not exist");
            return;
        }

        // refresh keep alive
        ListIterator<UserRadioConnection> connIter =
                user.connections.listIterator();
        while(connIter.hasNext()){
            UserRadioConnection connection = connIter.next();
            if (connection.connectionID ≡ request.getConnectionId() ∧
```

```
                    connection.radio.equals(request.getRadio())) {
                connection.keepAlive = new Date();
                connIter.set(connection);
                System.out.println(" [x] Refreshing keepalive " +
                        "from: " + user.username + " to radio: "
                        + connection.radio + " id: " +
                        connection.connectionID);
                break;
            }
        }

        // add to user total listened minutes
        user.secondsListening += Configuration.KeepAlivePeriodSeconds;

        // update user
        database.updateRow(user);
    }

    public static void main(String[] argv) throws Exception {
        if (argv.length < 1) {
            System.out.println("Usage: UsersDBHAndler mask1 mask2 mask3");
            return;
        }
        List<String> masks = new LinkedList<>(Arrays.asList(argv));

        // define database
        Database<User> database = new DatabaseJson<>(
                Configuration.UsersDBExchange + "_" +
                        Configuration.maskListToStr(masks), User.class);

        // start database handler
        UsersDBHandler handler = new UsersDBHandler(Configuration.RabbitMQHost,
                database, masks);
        handler.run();
    }
}
```

```java
import java.util.Date;

public class UserRadioConnection {

    public String radio;
    // para diferencias un usuario con varias conexiones a la misma radio
    public int connectionID;
    public Date keepAlive;

    public UserRadioConnection(String radio, Date keepAlive,
                               int connectionID){
        this.radio = radio;
        this.keepAlive = keepAlive;
        this.connectionID = connectionID;
    }
}
```

```java
import java.util.LinkedList;
import java.util.List;

public class User extends DatabaseRow {

    public String username;
    public List<UserRadioConnection> connections = new LinkedList<>();
    public long secondsListening;
    public int connectionsLimit;

    public User(String username) {
        super(username);

        this.username = username;
        this.secondsListening = 0;
        this.connectionsLimit = Configuration.MaxConnectionsPerFreeUser;
    }
}
```

```java
public class UserDisconnectRequest {

    // request
    private String username;
    private String radio;
    private int connectionId;

    public UserDisconnectRequest(String username, String radio, int
            connectionId) {
        this.setUsername(username);
        this.setRadio(radio);
        this.setConnectionId(connectionId);
    }

    public String toLogLine() {
        return Configuration.LogsDisconnectionTag + " " + getUsername() + " " +
                getRadio() + " " + getConnectionId();
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getRadio() {
        return radio;
    }

    public void setRadio(String radio) {
        this.radio = radio;
    }

    public int getConnectionId() {
        return connectionId;
    }

    public void setConnectionId(int connectionId) {
        this.connectionId = connectionId;
    }
}
```

```java
import java.util.LinkedList;
import java.util.List;

public class UserConnectResponse {

    // request
    private String username;
    private String radio;

    // id
    private String returnQueueName;

    // response
    private boolean couldConnect = false;
    private int connectionId;
    private List<UserDisconnectRequest> closedConnections = new LinkedList<>();

    public UserConnectResponse(UserConnectRequest request) {
        this.setUsername(request.getUsername());
        this.setRadio(request.getRadio());
        this.setReturnQueueName(request.getReturnQueueName());
    }

    public String toLogLine() {
        return Configuration.LogsConnectionTag + " " + getUsername() + " " +
                getRadio() + " " + getConnectionId();
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getRadio() {
        return radio;
    }

    public void setRadio(String radio) {
        this.radio = radio;
    }

    public String getReturnQueueName() {
        return returnQueueName;
    }

    public void setReturnQueueName(String returnQueueName) {
        this.returnQueueName = returnQueueName;
    }

    public boolean isCouldConnect() {
        return couldConnect;
    }

    public void setCouldConnect(boolean couldConnect) {
        this.couldConnect = couldConnect;
    }

    public int getConnectionId() {
        return connectionId;
    }

    public void setConnectionId(int connectionId) {
        this.connectionId = connectionId;
    }

    public List<UserDisconnectRequest> getClosedConnections() {
        return closedConnections;
    }

    public void setClosedConnections(List<UserDisconnectRequest> closedConnectio
ns) {
            this.closedConnections = closedConnections;
    }
}
```

```java

public class UserConnectRequest {

    // request
    private String username;
    private String radio;

    // id
    private String returnQueueName;

    public UserConnectRequest(String username, String radio,
                              String returnQueueName) {
        this.setUsername(username);
        this.setRadio(radio);
        this.setReturnQueueName(returnQueueName);
    }

    public UserConnectRequest(UserConnectResponse resp) {
        this.username = resp.getUsername();
        this.radio = resp.getRadio();
        this.returnQueueName = resp.getReturnQueueName();
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getRadio() {
        return radio;
    }

    public void setRadio(String radio) {
        this.radio = radio;
    }

    public String getReturnQueueName() {
        return returnQueueName;
    }

    public void setReturnQueueName(String returnQueueName) {
        this.returnQueueName = returnQueueName;
    }
}
```

```java
public class StatisticTask {

    private Runnable runnable;
    private int period;

    public StatisticTask(Runnable runnable, int period) {
        this.runnable = runnable;
        this.period = period;
    }

    public Runnable getRunnable() {
        return runnable;
    }

    public void setRunnable(Runnable runnable) {
        this.runnable = runnable;
    }

    public int getPeriod() {
        return period;
    }

    public void setPeriod(int period) {
        this.period = period;
    }
}
```

```java
public class RadiosUpdateRequest {

    private String radio;
    private String username;

    public RadiosUpdateRequest(String radio, String username) {
        this.setRadio(radio);
        this.setUsername(username);
    }

    public RadiosUpdateRequest(UserConnectRequest req) {
        this.radio = req.getRadio();
        this.username = req.getUsername();
    }

    public RadiosUpdateRequest(UserDisconnectRequest req) {
        this.radio = req.getRadio();
        this.username = req.getRadio();
    }

    public String getRadio() {
        return radio;
    }

    public void setRadio(String radio) {
        this.radio = radio;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

Apr 25, 18 21:45 **RadioStatisticsViewer.java** Page 1/1

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

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

public class RadioStatisticsViewer extends RabbitMQProcess {

    public RadioStatisticsViewer(String host) throws IOException,
            TimeoutException {
        super(host);

        // declare RADIOS_STATS exchange
        getChannel().exchangeDeclare(Configuration.RadiosStatisticsExchange,
                BuiltinExchangeType.FANOUT);
    }

    @Override
    public void run() throws IOException {
        consumeStatistics();
    }

    private String consumeStatistics() throws IOException {
        String statisticsQueue = getChannel().queueDeclare().getQueue();
        getChannel().queueBind(statisticsQueue,
                Configuration.RadiosStatisticsExchange, "");

        Consumer consumerStatistics = new DefaultConsumer(getChannel()) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                String json = new String(body, "UTF-8");
                RadiosConnectionsStatistics statistics = new Gson().fromJson
                        (json, RadiosConnectionsStatistics.class);

                System.out.println(" [x] Showing connections per radio: ");
                for (String radio : statistics.getRadioConnections().keySet()) {
                    System.out.println(radio + ": " +
                            statistics.getRadioConnections().get(radio));
                }
            }
        };
        // consume de una cola temporal a traves de un exchange
        // por lo que no tiene sentido ack manual
        return getChannel().basicConsume(statisticsQueue,
                true, consumerStatistics);
    }

    public static void main(String[] argv) throws Exception {
        RadioStatisticsViewer statisticsViewer =
                new RadioStatisticsViewer(Configuration.RabbitMQHost);
        statisticsViewer.run();
    }
}
```

Apr 26, 18 3:20 **RadioStation.java** Page 1/1

```java
import com.rabbitmq.client.*;

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

public class RadioStation extends RabbitMQProcess {

    private String exchangeName;
    private ScheduledExecutorService transmissionScheduler;
    private ScheduledFuture<?> transmissionHandle;

    private RadioSource source;

    public RadioStation(String host, String radioName, RadioSource source) throw
s
            IOException, TimeoutException {
        super(host);

        this.source = source;
        source.init();

        // declare BROADCAST exchange
        exchangeName = Configuration.RadioExchangePrefix + radioName;
        getChannel().exchangeDeclare(exchangeName, BuiltinExchangeType.FANOUT);
    }

    @Override
    public void run() {
        scheduleTransmission();
    }

    private void scheduleTransmission() {
        transmissionScheduler = Executors
                .newScheduledThreadPool(1);

        transmissionHandle =
                transmissionScheduler.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                byte[] nextBlock = source.getNextByteBlock();

                try {
                    getChannel().basicPublish(exchangeName, "",
                            null, nextBlock);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }, Configuration.RadioSendPeriodMilliseconds,
                        Configuration.RadioSendPeriodMilliseconds,
                        TimeUnit.MILLISECONDS);
    }

    @Override
    protected void close() throws IOException, TimeoutException {
        super.close();

        transmissionHandle.cancel(true);
        transmissionScheduler.shutdown();
        source.close();
    }

    public static void main(String[] argv) throws Exception {
        RadioSource source = argv.length ≡ 2 ?
                new RadioSourceFile(argv[1]) :
                new RadioSourceRandomNumbers();
        RadioStation radio = new RadioStation(Configuration.RabbitMQHost, argv[0
], source);
        radio.run();
    }

}
```

```java
import java.util.Base64;
import java.util.concurrent.ThreadLocalRandom;

public class RadioSourceRandomNumbers implements RadioSource {


    @Override
    public void init() {

    }

    @Override
    public byte[] getNextByteBlock() {
        int randomNum = ThreadLocalRandom.current().nextInt(0, 100 + 1);
        Logger.output(" [x] Sent: " + randomNum);
        return Base64.getEncoder().encode(Integer.toString(randomNum)
                .getBytes());
    }

    @Override
    public void close() {

    }
}
```

```java
import java.io.FileNotFoundException;

public interface RadioSource {

    void init() throws FileNotFoundException;

    byte[] getNextByteBlock();

    void close();
}
```

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Base64;

public class RadioSourceFile implements RadioSource {

    String filename;
    FileInputStream audio;
    int bytesPerRead;
    byte[] buffer;

    public RadioSourceFile(String filename) {
        this.filename = filename;
        this.bytesPerRead = 1000 * Configuration.RadioSendPeriodMilliseconds;
        buffer = new byte[this.bytesPerRead];
    }

    @Override
    public void init() throws FileNotFoundException {
        audio = new FileInputStream(filename);
    }

    @Override
    public byte[] getNextByteBlock() {
        try {
            int bytesSent = audio.read(buffer);
            if (bytesSent ≡ -1) {
                audio.close();
                audio = new FileInputStream(filename);
            }
            Logger.output(" [x] Sent: " + bytesSent + " bytes");
            return Base64.getEncoder().encode(buffer);
        } catch (IOException e) {
            Logger.output("IOException while reading blocks from file");
        }
        return "STATIC".getBytes();
    }

    @Override
    public void close() {
        try {
            audio.close();
        } catch (IOException e) {
            Logger.output("IOException while closing");
        }
    }
}
```

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.*;

public class RadiosDBHandler extends DBHandlerWithStatistics<Radio> {

    private String queueName;

    public RadiosDBHandler(String host, Database<Radio> database,
                           List<String> masks)
            throws IOException, TimeoutException {
        super(host, database);
        this.database = database;

        // declare RADIOS_DB exchange
        getChannel().exchangeDeclare(Configuration.RadiosDBExchange,
                BuiltinExchangeType.TOPIC);

        // declare RADIOS_STATS exchange
        getChannel().exchangeDeclare(Configuration.RadiosStatisticsExchange,
                BuiltinExchangeType.FANOUT);

        this.queueName = Configuration.RadiosDBExchange + "_" +
                Configuration.maskListToStr(masks);
        getChannel().queueDeclare(queueName, true, false, false, null);
        for (String mask : masks) {
            getChannel().queueBind(queueName,
                    Configuration.RadiosDBExchange, mask);
        }
    }

    @Override
    public void run() throws IOException {

        Consumer radiosConsumer = new DefaultConsumer(getChannel()) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                // parse request
                String jsonRequest = new String(body, "UTF-8");

                DatabaseRequest request = new Gson().fromJson(jsonRequest,
                        DatabaseRequest.class);

                if (request.getType() ≡ Configuration.UsersTypeConnect) {
                    consumeConnection(request.getSerializedRequest());
                } else if (request.getType() ≡
                        Configuration.UsersTypeDisconnect) {
                    consumeDisconnection(request.getSerializedRequest());
                } else {
                    Logger.output("Invalid request type received: " +
                            request.getType() + ", request: " +
                            request.getSerializedRequest());
                }

                getChannel().basicAck(envelope.getDeliveryTag(), false);
            }
        };

        getChannel().basicConsume(queueName, false, radiosConsumer);
    }

    @Override
    protected List<StatisticTask> getStatistics() {
        List<StatisticTask> operations = new LinkedList<>();

        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                // get statistics
                RadiosConnectionsStatistics stats = new RadiosConnectionsStatist
ics();
                for (Radio row : database.getRows()) {
                    stats.getRadioConnections().put(row.getName(), row.getConnec
tedUsers());
                }
                String jsonStats = new Gson().toJson(stats);

                try {
                    getChannel().basicPublish(
                            Configuration.RadiosStatisticsExchange, "",
                            null, jsonStats.getBytes());
                } catch (IOException e) {
                    Logger.output("IOEXception during statistics publish");
                }
            }
        };

        operations.add(new StatisticTask(runnable,
                Configuration.RadioStatisticsPeriodSeconds));

        return operations;
    }

    private void consumeConnection(String jsonRequest) throws IOException {

        UserConnectRequest request = new Gson().fromJson(jsonRequest,
                UserConnectRequest.class);

        // get radio record from DB
        Radio radio = database.getRow(request.getRadio());
        if (radio ≡ null) {
            radio = new Radio(request.getRadio());
        }

        // add one connection to counter
        radio.setConnectedUsers(radio.getConnectedUsers() + 1);
        System.out.println(" [x] Adding one connection to radio: " +
                radio.getName());

        // save changes to db
        database.updateRow(radio);
    }

    private void consumeDisconnection(String jsonRequest) throws IOException {

        UserDisconnectRequest request = new Gson().fromJson(jsonRequest,
                UserDisconnectRequest.class);

        // get radio record from DB
        Radio radio = database.getRow(request.getRadio());
        if (radio ≡ null) {
            radio = new Radio(request.getRadio());
        }

        // add one connection to counter
        radio.setConnectedUsers(radio.getConnectedUsers() - 1);
        System.out.println(" [x] Removing one connection from " +
                "radio: " + radio.getName());

        // save changes to db
        database.updateRow(radio);
    }

    public static void main(String[] argv) throws Exception {
        if (argv.length < 1) {
            System.out.println("Usage: UsersDBHAndler mask1 mask2 mask3");
            return;
        }
```

```java
        List<String> masks = new LinkedList<>(Arrays.asList(argv));

        // define database
        Database<Radio> database = new DatabaseJson<>(
                Configuration.RadiosDBExchange + "_" +
                        Configuration.maskListToStr(masks), Radio.class);

        // start database handler
        RadiosDBHandler handler = new RadiosDBHandler(Configuration.RabbitMQHost
,
                database, masks);
        handler.run();
    }
}
```

```java
import java.util.HashMap;
import java.util.Map;

public class RadiosConnectionsStatistics {

    private Map<String, Integer> radioConnections = new HashMap<>();

    public Map<String, Integer> getRadioConnections() {
        return radioConnections;
    }

    public void setRadioConnections(Map<String, Integer> radioConnections) {
        this.radioConnections = radioConnections;
    }
}
```

```java
public class Radio extends DatabaseRow {
    private String name;
    private int connectedUsers;

    public Radio(String name) {
        super(name);

        this.setName(name);
        this.setConnectedUsers(0);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getConnectedUsers() {
        return connectedUsers;
    }

    public void setConnectedUsers(int connectedUsers) {
        this.connectedUsers = connectedUsers;
    }
}
```

Apr 26, 18 2:55 **RabbitMQProcess.java** Page 1/1

```java
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

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

public abstract class RabbitMQProcess {

    private Connection connection;
    private Channel channel;

    public RabbitMQProcess(String host) throws IOException, TimeoutException {

        // load configuration
        Configuration.loadConfiguration("config");

        // init RabbitMQ connection and channel
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost(host);
        connection = factory.newConnection();
        channel = connection.createChannel();

        Logger.init();

        addShutdownHook();
    }

    protected Connection getConnection() {
        return connection;
    }

    protected Channel getChannel() {
        return channel;
    }

    public void addShutdownHook() {

        RabbitMQProcess instance = this;
        Thread mainThread = Thread.currentThread();
        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {

                System.out.println("Calling shutdown hook");

                try {
                    instance.close();
                } catch (IOException e) {
                    Logger.output("IOEXception during shutdown hook close");
                } catch (TimeoutException e) {
                    Logger.output("TimeoutException during shutdown hook " +
                            "close");
                }
                try {
                    mainThread.join();
                } catch (InterruptedException e) {
                    Logger.output("InterruptedException during shutdown hook" +
                            " close");
                }
            }
        });
    }

    protected void close() throws IOException, TimeoutException {
        channel.close();
        connection.close();
    }

    public abstract void run() throws IOException;
}
```

Apr 25, 18 21:36 **Logger.java** Page 1/2

```java
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintStream;
import java.io.PrintWriter;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Logger {

    public static logLevel currentLogLevel = logLevel.INFO;
    private static PrintWriter logWriter = null;
    private static PrintStream out = null;

    public enum logLevel {
        ERROR,
        WARNING,
        INFO
    }

    public static logLevel intToLogLevel(int i) {
        switch (i) {
            case 0:
                return logLevel.ERROR;
            case 1:
                return logLevel.WARNING;
            case 2:
                return logLevel.INFO;
            default:
                return logLevel.INFO;
        }
    }

    private static String logLevelToString(logLevel level) {
        switch(level) {
            case ERROR:
                return "[ERROR]";
            case WARNING:
                return "[WARNING]";
            case INFO:
                return "[INFO]";
            default:
                return "[INVALID LOGLEVEL]";
        }
    }

    public static void init() {
        out = System.out;
    }

    public static void init(String filename) {
        init();
        try {
            logWriter = new PrintWriter(new FileWriter(filename));

            String timeStamp = new SimpleDateFormat(
                    "yyyy/MM/dd/ HH:mm:ss").format(new Date());
            logWriter.println("***************" +
                    timeStamp + "***************");
        } catch (IOException e) {
            log("Logger", "Couldn't open logfile for writing",
                    logLevel.ERROR);
        }
    }

    public static void close() {
        if (logWriter ≠ null) {
            logWriter.close();
        }
    }

    public static void log(String name, String message, logLevel level) {

        String logLine = Thread.currentThread().getName() + "\t" +
                logLevelToString(level) + "\t" + name + ":" + message;

        // output to screen
        if (currentLogLevel.ordinal() ≥ level.ordinal()) {
            if (out ≠ null) {
                out.println(logLine);
            } else {
                System.out.println(logLine);
            }
        }
        // output to logfile
        if (logWriter ≠ null) {
            logWriter.println(logLine);
        }
    }

    public static void output(String outString) {
        if (out ≠ null) {
            out.println(outString);
        } else {
            System.out.println(outString);
        }
        if (logWriter ≠ null) {
            logWriter.println(outString);
        }
    }

}
```

```java
import java.util.Collection;
import java.util.Comparator;
import java.util.TreeSet;

// un sorted set que automaticamente borra elementos
// de si mismo si se pasa del maximo
class LimitedSortedSet<E> extends TreeSet<E> {

    private int maxSize;

    LimitedSortedSet( int maxSize ) {
        this.maxSize = maxSize;
    }

    LimitedSortedSet( int maxSize, Comparator<? super E> comparator ) {
        super(comparator);
        this.maxSize = maxSize;
    }

    @Override
    public boolean addAll( Collection<? extends E> c ) {
        boolean added = super.addAll( c );
        if( size() > maxSize ) {
            E firstToRemove = (E)toArray( )[maxSize];
            removeAll( tailSet( firstToRemove ) );
        }
        return added;
    }

    @Override
    public boolean add( E o ) {
        boolean added =  super.add( o );
        while (size() > maxSize) {
            remove(last());
        }
        return added;
    }


}
```

Apr 25, 18 21:01 **KeepAliveRequest.java** Page 1/1

```java
public class KeepAliveRequest {

    private String username;
    private int connectionId;
    private String radio;

    public KeepAliveRequest(String username, int connectionId, String radio) {
        this.setUsername(username);
        this.setConnectionId(connectionId);
        this.setRadio(radio);
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public int getConnectionId() {
        return connectionId;
    }

    public void setConnectionId(int connectionId) {
        this.connectionId = connectionId;
    }

    public String getRadio() {
        return radio;
    }

    public void setRadio(String radio) {
        this.radio = radio;
    }
}
```

Apr 26, 18 0:58 **KeepAliveManager.java** Page 1/1

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

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

public class KeepAliveManager extends RabbitMQProcess {

    public KeepAliveManager(String host) throws IOException, TimeoutException {
        super(host);

        // declare USERS_DB exchange
        getChannel().exchangeDeclare(Configuration.UsersDBExchange,
                BuiltinExchangeType.TOPIC);
    }

    @Override
    public void run() throws IOException {
        consumeKeepAlives();
    }

    private String consumeKeepAlives() throws IOException {
        // KEEP ALIVE consumer
        getChannel().queueDeclare(Configuration.KeepAliveQueue, true,
                false, false, null);
        Consumer consumer_keepalive = new DefaultConsumer(getChannel()) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                String json = new String(body, "UTF-8");
                KeepAliveRequest clientRequest = new Gson().fromJson(json,
                        KeepAliveRequest.class);
                System.out.println(" [x] Received keep alive request from: "
                        + clientRequest.getUsername() + " to: " +
                        clientRequest.getRadio() + " id: " +
                        clientRequest.getConnectionId());

                // ask usersDB to register connection
                DatabaseRequest usersdbRequest = new DatabaseRequest
                        (Configuration.UsersTypeKeepAlive, json,
                                clientRequest.getUsername());
                getChannel().basicPublish(Configuration.UsersDBExchange,
                        usersdbRequest.getRoutingKey(), null,
                        new Gson().toJson(usersdbRequest).getBytes());

                getChannel().basicAck(envelope.getDeliveryTag(), false);
            }
        };
        return getChannel().basicConsume(Configuration.KeepAliveQueue, false,
                consumer_keepalive);
    }

    public static void main(String[] argv) throws Exception {
        KeepAliveManager manager =
                new KeepAliveManager(Configuration.RabbitMQHost);
        manager.run();
    }
}
```

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import java.util.List;
import java.util.ListIterator;
import java.util.concurrent.TimeoutException;

public abstract class FileLogger extends RabbitMQProcess {

    PrintWriter logWriter;
    String logsQueue;

    public FileLogger(String host, String logFilename) throws
            IOException, TimeoutException {
        super(host);

        // declare LOGS exchange
        getChannel().exchangeDeclare(Configuration.LogsExchange,
                BuiltinExchangeType.DIRECT);

        logWriter = new PrintWriter(new FileWriter(logFilename, true));

        logsQueue = getChannel().queueDeclare().getQueue();
        for (String tag : getBindings()) {
            getChannel().queueBind(logsQueue, Configuration.LogsExchange, tag);
        }
    }

    @Override
    public void run() throws IOException {
        consumeLogs();
    }

    protected abstract List<String> getBindings();

    public String consumeLogs() throws IOException {
        // consume connection logs
        Consumer connectConsumer = new DefaultConsumer(getChannel()) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                // write log to file
                String logLine = new String(body, "UTF-8");
                logWriter.println(logLine);

                System.out.println(" [x] Received: " + logLine);
            }
        };

        // consume de una cola temporal a traves de un exchange
        // por lo que no tiene sentido ack manual
        return getChannel().basicConsume(logsQueue, true, connectConsumer);
    }

    @Override
    protected void close() throws IOException, TimeoutException {
        super.close();
        logWriter.close();
    }
}
```

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

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

public class DisconnectionManager extends RabbitMQProcess {

    public DisconnectionManager(String host) throws IOException,
            TimeoutException {
        super(host);

        // declare USERS_DB exchange
        getChannel().exchangeDeclare(Configuration.UsersDBExchange,
                BuiltinExchangeType.TOPIC);

        // declare RADIOS_DB exchange
        getChannel().exchangeDeclare(Configuration.RadiosDBExchange,
                BuiltinExchangeType.TOPIC);

        // declare LOGS exchange
        getChannel().exchangeDeclare(Configuration.LogsExchange,
                BuiltinExchangeType.DIRECT);
    }

    @Override
    public void run() throws IOException {
        consumeDisconnections();
    }

    private String consumeDisconnections() throws IOException {
        // DISCONNECTIONS consumer
        getChannel().queueDeclare(Configuration.DisconnectionsQueue, true,
                false, false, null);
        Consumer consumer_disconnect = new DefaultConsumer(getChannel()) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                String json = new String(body, "UTF-8");
                UserDisconnectRequest clientRequest = new Gson().fromJson(json,
                        UserDisconnectRequest.class);
                System.out.println(" [x] Received request to disconnect user:" +
                        " " + clientRequest.getUsername() + " from: " +
                        clientRequest.getRadio());

                // assemble database request
                DatabaseRequest dbRequest = new DatabaseRequest
                        (Configuration.UsersTypeDisconnect, json,
                                clientRequest.getUsername());

                // ask usersDB to register disconnection
                getChannel().basicPublish(Configuration.UsersDBExchange,
                        dbRequest.getRoutingKey(), null,
                        new Gson().toJson(dbRequest).getBytes());

                // ask radiosDB to register disconnection
                getChannel().basicPublish(Configuration.RadiosDBExchange,
                        dbRequest.getRoutingKey(), null,
                        new Gson().toJson(dbRequest).getBytes());

                // send disconnects to file logger
                getChannel().basicPublish(Configuration.LogsExchange,
                        Configuration.LogsDisconnectionTag, null,
                        clientRequest.toLogLine().getBytes());

                getChannel().basicAck(envelope.getDeliveryTag(), false);
            }
        };
        return getChannel().basicConsume(Configuration.DisconnectionsQueue,
                false, consumer_disconnect);
    }

    public static void main(String[] argv) throws Exception {
        DisconnectionManager manager =
                new DisconnectionManager(Configuration.RabbitMQHost);
        manager.run();
    }
}
```

```java
import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.*;

public abstract class DBHandlerWithStatistics<T extends DatabaseRow>
        extends RabbitMQProcess {

    Database<T> database;
    private ScheduledExecutorService statisticsScheduler = null;
    private List<ScheduledFuture<?>> statisticsHandles = null;

    public DBHandlerWithStatistics(String host, Database database) throws
            IOException,
            TimeoutException {
        super(host);
        this.database = database;

        List<StatisticTask> statisticTasks = getStatistics();
        //List<Integer> statisticTasksPeriods = getStatisticsPeriodsSeconds();
        if (statisticTasks.size() > 0) {
            statisticsScheduler = Executors
                    .newScheduledThreadPool(
                            Configuration.PoolSizeForDBstatistics);
            statisticsHandles = new LinkedList<>();

            for (StatisticTask task : statisticTasks) {
                Runnable r = task.getRunnable();
                int period = task.getPeriod();
                ScheduledFuture<?> statisticsHandle =
                        statisticsScheduler.scheduleAtFixedRate(r, period,
                                period, TimeUnit.SECONDS);
                statisticsHandles.add(statisticsHandle);
            }
        }
    }

    @Override
    protected void close() throws IOException, TimeoutException {
        super.close();

        if (statisticsScheduler ≠ null) {
            for (ScheduledFuture<?> f : statisticsHandles) {
                f.cancel(true);
            }
            statisticsScheduler.shutdown();
        }
    }

    protected abstract List<StatisticTask> getStatistics();
}
```

```java
public abstract class DatabaseRow {

    private String primary_key;

    public DatabaseRow(String primary_key) {
        this.setPrimary_key(primary_key);
    }

    public String getPrimary_key() {
        return primary_key;
    }

    public void setPrimary_key(String primary_key) {
        this.primary_key = primary_key;
    }
}
```

```java
public class DatabaseRequest {

    private int type;
    private String serializedRequest;
    private String routingKey;

    public DatabaseRequest(int type, String serializedRequest, String
            username) {
        this.type = type;
        this.serializedRequest = serializedRequest;
        this.setRoutingKey(username.substring(0, 1));
    }

    public int getType() {
        return type;
    }

    public void setType(int type) {
        this.type = type;
    }

    public String getSerializedRequest() {
        return serializedRequest;
    }

    public void setSerializedRequest(String serializedRequest) {
        this.serializedRequest = serializedRequest;
    }

    public String getRoutingKey() {
        return routingKey;
    }

    public void setRoutingKey(String routingKey) {
        this.routingKey = routingKey;
    }
}
```

Apr 25, 18 22:54 **DatabaseRAM.java** Page 1/1

```java
import com.google.gson.internal.LinkedTreeMap;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

public class DatabaseRAM<T extends DatabaseRow> implements Database<T> {

    private Map<String, T> database = new HashMap<>();

    public T getRow(String key) {
        T row = database.getOrDefault(key, null);
        return row;
    }

    @Override
    public List<T> getRows() {
        return new LinkedList<>(database.values());
    }

    @Override
    public boolean createRow(T row) {
        if (database.put(row.getPrimary_key(), row) ≡ null) {
            return true;
        }
        return false;
    }

    @Override
    public boolean updateRow(T row) {
        if (¬database.containsKey(row.getPrimary_key())) {
            return createRow(row);
        }
        database.put(row.getPrimary_key(), row);
        return true;
    }

    @Override
    public boolean removeRow(String primary_key) {
        if (database.remove(primary_key) ≠ null) {
            return true;
        }
        return false;
    }

    protected Map<String, T> getDatabase() {
        return database;
    }

    protected void setDatabase(Map<String, T> db) {
        this.database = db;
    }
}
```

Apr 25, 18 23:17 **DatabaseJson.java** Page 1/2

```java
import com.google.gson.Gson;

import java.io.*;
import java.util.HashMap;
import java.util.Map;

public class DatabaseJson<T extends DatabaseRow> extends DatabaseRAM<T> {

    private String filename;
    private Class<T> classOfT;

    public DatabaseJson(String filename, Class<T> classOfT) {
        this.filename = filename;
        this.classOfT = classOfT;

        loadFromFile();
    }

    @Override
    public boolean createRow(T row) {
        boolean result = super.createRow(row);
        if (result) {
            saveToFile();
        }
        return result;
    }

    @Override
    public boolean updateRow(T row) {
        boolean result = super.updateRow(row);
        if (result) {
            saveToFile();
        }
        return result;
    }

    @Override
    public boolean removeRow(String primary_key) {
        boolean result = super.removeRow(primary_key);
        if (result) {
            saveToFile();
        }
        return result;
    }

    private void loadFromFile() {
        try {

            BufferedReader br = new BufferedReader(new FileReader(filename));
            Map<String, T> db = new HashMap<>();

            Gson gson = new Gson();
            String line;
            while ((line = br.readLine()) ≠ null) {
                Logger.output(line);
                T row = gson.fromJson(line, classOfT);
                db.put(row.getPrimary_key(), row);
            }
            br.close();
            setDatabase(db);

        } catch (IOException e) {
            Logger.output("Unable to load database from file: " + filename);
        }
    }

    private void saveToFile() {
        try {
            PrintWriter writer = new PrintWriter(new FileWriter(filename));
            Map<String, T> db = getDatabase();
            Gson gson = new Gson();

            for (String key : db.keySet()) {
                String jsonRow = gson.toJson(db.get(key), classOfT);
                writer.println(jsonRow);
            }
            writer.close();
        } catch (IOException e) {
            Logger.output("Unable to save database to file: " + filename);
        }
    }
}
```

```java
import java.util.List;

public interface Database<T extends DatabaseRow> {

    T getRow(String key);

    List<T> getRows();

    boolean createRow(T row);

    boolean updateRow(T row);

    boolean removeRow(String key);
}
```

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;
import sun.security.krb5.Config;

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

public class ConnectionManager extends RabbitMQProcess {

    public ConnectionManager(String host) throws IOException, TimeoutException {
        super(host);

        // declare USERS_DB exchange
        getChannel().exchangeDeclare(Configuration.UsersDBExchange,
                BuiltinExchangeType.TOPIC);

        // declare usersDB responses queue
        getChannel().queueDeclare(Configuration.ConnMgrUsersDBResponseQueue,
                true, false, false, null);

        // declare RADIOS_DB exchange
        getChannel().exchangeDeclare(Configuration.RadiosDBExchange,
                BuiltinExchangeType.TOPIC);

        // declare LOGS exchange
        getChannel().exchangeDeclare(Configuration.LogsExchange,
                BuiltinExchangeType.DIRECT);
    }

    @Override
    public void run() throws IOException {
        //consumeConnections();
        consumeUsersDB();
    }

    private String consumeUsersDB() throws IOException {
        // usersDB consume
        Consumer consumer_usersdb = new DefaultConsumer(getChannel()) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                String json = new String(body, "UTF-8");
                UserConnectResponse response = new Gson().fromJson(json,
                        UserConnectResponse.class);

                for (UserDisconnectRequest disconn :
                        response.getClosedConnections()) {

                    // register closed connections in radios DB
                    DatabaseRequest dbRequest = new DatabaseRequest
                            (Configuration.UsersTypeDisconnect, new Gson()
                                    .toJson(disconn), disconn.getUsername());
                    getChannel().basicPublish(Configuration.RadiosDBExchange,
                            dbRequest.getRoutingKey(), null,
                            new Gson().toJson(dbRequest).getBytes());

                    // send disconnects to file logger
                    getChannel().basicPublish(Configuration.LogsExchange,
                            Configuration.LogsDisconnectionTag, null,
                            disconn.toLogLine().getBytes());
                }

                if (response.isCouldConnect()) {
                    // send connect to file logger
                    getChannel().basicPublish(Configuration.LogsExchange,
                            Configuration.LogsConnectionTag, null,
                            response.toLogLine().getBytes());

                    // register connection in radios DB
                    DatabaseRequest dbRequest = new DatabaseRequest
                            (Configuration.UsersTypeConnect, new Gson()
```

```java
                                    .toJson(new UserConnectRequest(response)),
                                    response.getUsername());
                    getChannel().basicPublish(Configuration.RadiosDBExchange,
                            dbRequest.getRoutingKey(), null,
                            new Gson().toJson(dbRequest).getBytes());

                    System.out.println(" [X] User: " + response.getUsername() +
                            " connected to radio: " + response.getRadio());
                } else {
                    System.out.println(" [X] User: " + response.getUsername() +
                            " denied connection to radio: " + response.getRadio());
                }

                String jsonResponse = new Gson().toJson(response);
                getChannel().basicPublish("", response.getReturnQueueName(), nul
l,
                        jsonResponse.getBytes());

                getChannel().basicAck(envelope.getDeliveryTag(), false);
            }
        };
        return getChannel().basicConsume(Configuration.ConnMgrUsersDBResponseQue
ue,
                false, consumer_usersdb);
    }

    public static void main(String[] argv) throws Exception {

        ConnectionManager manager =
                new ConnectionManager(Configuration.RabbitMQHost);
        manager.run();
    }

}
```

```java
import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.TimeoutException;

public class ConnDisconnFileLogger extends FileLogger {

    public ConnDisconnFileLogger(String host, String logFilename) throws
            IOException, TimeoutException {
        super(host, logFilename);
    }

    @Override
    protected List<String> getBindings() {
        List<String> bindings = new LinkedList<>();
        bindings.add(Configuration.LogsConnectionTag);
        bindings.add(Configuration.LogsDisconnectionTag);
        return bindings;
    }

    public static void main(String[] argv) throws Exception {

        ConnDisconnFileLogger fileLogger =
                new ConnDisconnFileLogger(Configuration.RabbitMQHost,
                        argv[0]);
        fileLogger.run();
    }
}
```

```java
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.List;

public class Configuration {

    public static int RadioSendPeriodMilliseconds = 1000;
    public static int RadioAudiofileBytesPerSecond = 102400;
    public static int KeepAlivePeriodSeconds = 5;
    public static int SecondsUntilDropConnection = 10;
    public static int MaxConnectionsPerFreeUser = 3;
    public static int MaxConnectionsPerUnlimitedUser = 999;

    public static String RabbitMQHost = "localhost";

    public static int PoolSizeForDBstatistics = 1;

    public static String UsersDBExchange = "USERS_DB";
    public static int UsersTypeConnect = 1;
    public static int UsersTypeDisconnect = 2;
    public static int UsersTypeKeepAlive = 3;
    public static String UsersStatisticsExchange = "USERS_STATS";
    public static int UsersStatisticsPeriodSeconds = 10;
    public static int UserStatisticsN = 100;

    public static String RadiosDBExchange = "RADIOS_DB";
    public static String RadiosStatisticsExchange = "RADIOS_STATS";
    public static int RadioStatisticsPeriodSeconds = 10;

    public static String ConnMgrUsersDBResponseQueue =
            "usersDBResponseQueueName";

    public static String ConnectionsQueue = "CONNECTIONS";
    public static String DisconnectionsQueue = "DISCONNECTIONS";
    public static String KeepAliveQueue = "KEEP_ALIVE";

    public static String RadioExchangePrefix = "BROADCAST-";

    public static String LogsExchange = "LOGS";
    public static String LogsConnectionTag = "connect";
    public static String LogsDisconnectionTag = "disconnect";

    public static String maskListToStr(List<String> masks) {
        return String.join("", masks)
                .replace(".", "")
                .replace("#", "");
    }

    public static boolean loadConfiguration(String configFilename) {

        try {
            // read json config
            BufferedReader br = new BufferedReader(
                    new FileReader( configFilename));
            String jsonString = "";
            String s;
            while ((s = br.readLine()) ≠ null) {
                jsonString += s;
            }

            // esto es para que gson serialize variables estaticas
            GsonBuilder gsonBuilder  = new GsonBuilder();
            gsonBuilder.excludeFieldsWithModifiers(
                    java.lang.reflect.Modifier.TRANSIENT);

            Gson gson = gsonBuilder.create();
            // load to object
            Configuration config = gson.fromJson(jsonString,
                    Configuration.class);
        }
        catch (FileNotFoundException e) {
            Logger.output("FileNotFoundException loading " +
                    "configuration file, using defaults");
        } catch (IOException e) {
            Logger.output("IOException loading configuration file, using " +
                    "defaults");
        } finally {
            return true;
        }
    }
}
```

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;
import java.util.Base64;

import java.io.FileOutputStream;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.NoSuchElementException;
import java.util.Scanner;
import java.util.concurrent.*;

public class Client extends  RabbitMQProcess {

    private String radioExchange = "";

    private String username = "";
    private String radio;
    private int connectionId;
    private String radioConsumeTag = "";
    FileOutputStream transmissionWriter = null;

    private ScheduledExecutorService keepAliveScheduler =
            Executors.newScheduledThreadPool(1);
    private ScheduledFuture<?> keepAliveHandle;

    public Client(String host) throws IOException, TimeoutException {
        super(host);
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setRadio(String radio) {
        this.radio = radio;
    }

    public boolean requestConnectionToRadio() throws IOException,
            InterruptedException {

        if (username.equals("")) {
            Logger.output("ERROR: Did you specify a username?");
            return false;
        }

        // define callback queue
        String callbackQueueName = getChannel().queueDeclare().getQueue();

        // create request
        UserConnectRequest request = new UserConnectRequest(username, radio,
                callbackQueueName);
        String requestJson = new Gson().toJson(request);

        // publish to usersDB exchange to start register connection operation
        DatabaseRequest usersdbRequest = new DatabaseRequest
                (Configuration.UsersTypeConnect, requestJson, username);
        getChannel().basicPublish(Configuration.UsersDBExchange,
                usersdbRequest.getRoutingKey(), null,
                new Gson().toJson(usersdbRequest).getBytes());

        final BlockingQueue<String> responseQueue =
                new ArrayBlockingQueue<String>(1);

        // es una cola temporaria, no sirve de nada el ack
        String callbackTag = getChannel().basicConsume(callbackQueueName,true,
                new DefaultConsumer(getChannel()) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {
                responseQueue.offer(new String(body, "UTF-8"));
            }
        });

        String jsonResponse = responseQueue.take();
        getChannel().basicCancel(callbackTag);
        UserConnectResponse response = new Gson().fromJson(jsonResponse,
                UserConnectResponse.class);
        if (¬response.isCouldConnect()) {
            Logger.output("ERROR: Connection refused, are " +
                    "you already connected on 3 devices?");
            return false;
        }

        connectionId = response.getConnectionId();
        radioExchange = Configuration.RadioExchangePrefix + response.getRadio();
        return true;
    }

    public boolean listenToRadio() throws IOException {

        if (radioExchange.equals("")) {
            return false;
        }

        // declare radio broadcast exchange
        getChannel().exchangeDeclare(radioExchange, BuiltinExchangeType.FANOUT);

        // declare temporary queue and bind
        String queueName = getChannel().queueDeclare().getQueue();
        getChannel().queueBind(queueName, radioExchange, "");
        Logger.output("Creating queue: " + queueName);

        // open new file for transmission
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy−MM−dd−HH−mm−ss");
        String transmissionName = "client" + "−" + username + "−" + radio +
                "−" + connectionId + "−" + sdf.format(new Date()) + ".wav";
        transmissionWriter = new FileOutputStream(transmissionName);

        Consumer consumer = new DefaultConsumer(getChannel()) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {
                String message = new String(body, "UTF−8");
                Logger.output(" [x] Received '" + message + "'");
                byte[] decodedBody = Base64.getDecoder().decode(body);
                transmissionWriter.write(decodedBody);
            }
        };
        // es una cola temporaria, no sirve de nada el ack
        radioConsumeTag = getChannel().basicConsume(queueName, true,
                consumer);
        return true;
    }

    public void scheduleKeepAlive() {
        final Runnable sendKeepAlive = new Runnable() {
            @Override
            public void run() {
                KeepAliveRequest request = new KeepAliveRequest(username,
                        connectionId, radio);
                String requestJson = new Gson().toJson(request);
                try {
                    getChannel().basicPublish("",
                            Configuration.KeepAliveQueue, null,
                            requestJson.getBytes());
                } catch (IOException e) {
                    Logger.output("IOException while listening to" +
                            "rado: " + radio + ", user: " + username +
                            ", connection id: " + connectionId);
                }
            }
        };
        keepAliveHandle = keepAliveScheduler.scheduleAtFixedRate
```

```
                (sendKeepAlive, Configuration.KeepAlivePeriodSeconds,
                        Configuration.KeepAlivePeriodSeconds, TimeUnit.SECONDS);
    }

    public void stopKeepAlive() {
        keepAliveHandle.cancel(true);
    }

    public void stopListeningToRadio() throws IOException {
        if (radioConsumeTag.equals("")) {
            Logger.output("ERROR: not listening to radio");
        } else {
            // create request
            UserDisconnectRequest request = new UserDisconnectRequest(username,
                    radio, connectionId);
            String requestJson = new Gson().toJson(request);

            // publish to DISCONNECTIONS queue
            getChannel().basicPublish("", Configuration.DisconnectionsQueue,
                    null, requestJson.getBytes());

            // stop receiving transmission
            getChannel().basicCancel(radioConsumeTag);
            radioConsumeTag = "";

            // close transmission file
            transmissionWriter.close();
            transmissionWriter = null;
        }
    }

    public void printOptions() {
        Logger.output("\n");
        Logger.output("Choose an action: ");
        Logger.output("\t" + "1. Set user");
        Logger.output("\t" + "2. Connect to radio");
        Logger.output("\t" + "3. Disconnect from radio");
        Logger.output("\t" + "4. Exit");
    }

    public boolean mainMenu(Scanner in) throws IOException,
            InterruptedException {
        String choiceStr = in.nextLine();
        int choice = Integer.parseInt(choiceStr);
        switch (choice) {
            case 1:
                System.out.print("Please specify a username: ");
                String username = in.nextLine();
                setUsername(username);
                break;
            case 2:
                Logger.output("Please specify a radio: ");
                String radio = in.nextLine();
                setRadio(radio);
                if (¬requestConnectionToRadio()) {
                    break;
                }
                listenToRadio();
                scheduleKeepAlive();
                break;
            case 3:
                stopListeningToRadio();
                stopKeepAlive();
                break;
            case 4:
                Logger.output("Press CTRL+C to exit");
                return true;
            default:
                Logger.output("ERROR: Invalid option");
                break;
        }

        printOptions();
```

```java
            return false;
        }

        @Override
        protected void close() throws IOException, TimeoutException {
            super.close();
            if (transmissionWriter ≠ null) {
                transmissionWriter.close();
            }
        }

        @Override
        public void run() throws IOException {
            Scanner in = new Scanner(System.in);
            printOptions();

            boolean end = false;
            while (¬end) {
                try {
                    end = mainMenu(in);
                // esta excepcion aparece al apretar ctrl+c
                } catch (NoSuchElementException e) {
                    end = true;
                } catch (InterruptedException e) {
                    end = true;
                }
            }
        }


        public static void main(String[] argv) throws Exception {
            Client client = new Client(Configuration.RabbitMQHost);
            client.run();
        }

    }
```

Table of Contents