# RabbitMQ LabGuide

**Preparation**

- Download and install Erlang/OTP and RabbitMQ. Use the following instructions https://www.rabbitmq.com/download.html
- RabbitMQ server will start automatically using defaults
- Use RabbitMQ management interface http://localhost:15672 for monitoring
- Make sure you have project with lab tasks
- Enable the topic selector plugin:
  $ rabbitmq-plugins enable rabbitmq_jms_topic_exchange

## Lab1

**Duration:**

60 min

**Objectives:**

Practice in using RabbitMQ to send and receive byte messages. Get familiar with direct exchange and routing keys - learn how to subscribe to a subset of messages.

**Description:**

We have a source folder "src/main/resources/lab1/source" containing images of cats, dogs, pandas, and raccoons. Each image has a specific file name format: names of the cat images should contain “cat”, names of the dog images should contain “dog”, names of the panda images should contain “panda”, and names of the raccoon images should contain “raccoon”.

Our task is to distribute all the images among the following folders: “src/main/resources/lab1/cat”, “src/main/resources/lab1/dog”, “src/main/resources/lab1/fun”. Content of the “cat” and “dog” folders is obvious. As for pandas and raccoons images, they should go to the “fun” folder.

We’ll manage that by means of RabbitMQ messaging.

First we have to create publisher which will send all the images with corresponding routing keys (“cat”, “dog”, or “fun”) to the server. Then we’ll create 3 subscribers waiting for messages. First subscriber will wait for cats images, second one - for dogs images, and third one - for pandas and raccoons ones. Then publisher will finally send images. After receiving message, subscriber will save it to corresponding folder.

**Task:**

1. Complete the Publisher class:
   a. Implement Publisher constructor: **Publisher()**. It constructs new Publisher: creates new connection to the server and new channel. Use "localhost" as a host for ConnectionFactory.
   b. Implement **void close()** method. It closes the channel and the connection.
   c. Implement **byte[] readBytes(File file)** method. It reads all the bytes from a file.

d. Implement **AMQP.BasicProperties createProperties(File file)** method. It creates AMQP.BasicProperties object containing file name header. Subscriber needs it to create a new image at the new location properly.
   This method:
   i. Creates headers map.
   ii. Puts file name to that map.
   iii. Creates AMQP.BasicProperties object with headers.
e. Implement **void send(File file, String routingKey)** method. It sends file content with corresponding routing key to direct exchange. The message goes to the queues whose binding key exactly matches the routing key of the message.
   This method:
   i. Declares direct exchange.
   ii. Reads bytes from the file.
   iii. Creates message header containing file name.
   iv. Publishes message with properties containing that header to declared exchange.

2. Complete the Subscriber class:
   a. Implement Subscriber constructor: **Subscriber(String id, String[] routingKeys, File destination)**. It constructs new Subscriber: sets subscriber's id, routing keys, and file destination (place where to save received file); creates new connection to the server and new channel. Use "localhost" as a host for ConnectionFactory.
   b. Implement **void subscribe()** method. It subscribes to receive messages if its routing keys exactly matches subscriber's ones.
      This method:
      i. Declares direct exchange.
      ii. Creates a non-durable, exclusive, autodelete queue with a generated name.
      iii. Binds that queue to each routing key we are interested in.
      iv. Creates new DefaultConsumer with overridden handleDelivery method, which saves received file to subscriber's destination. It uses file name message header to create proper image file. (Bind subscriber's id to consumer's tag for clear logging inside handleDelivery method.)
      v. Starts consuming messages from the declared queue.
3. Make sure you have added logging and run the Starter class to check whether everything works fine. Before that, make sure that all the following folders exist: "src/main/resources/lab1/source" with images and empty "src/main/resources/lab1/cat", "src/main/resources/lab1/dog", and "src/main/resources/lab1/fun".

## Lab2

**Duration:**

60 min

**Objectives:**

Practice in using RabbitMQ to send and receive byte messages. Get familiar with topic exchange - learn how to route messages based on multiple criteria.

**Description:**

We have the Person class that describes a person with a native country, a name, and a year of birth.

Our task is to use RabbitMQ to send persons and to receive them based on combination of native country, name, and year of birth values. For example, we should be able to create subscriber that would receive only messages with person whose name is “John” and if he was born in 1990.

To manage that, we’ll use topic exchange and special routing keys of the following format: “<native country>.<name>.<year of birth>”. So, each person will be sent with routing key built according to his native country, name, and year of birth. For example, “USA.John.1990”. Therefore, each subscriber will receive only those persons, whose parameters match its binding keys. For example, “*.John.*” or "USA.#".

**Task:**

1. Take a look at the Person class. It implements Serializable, so its objects can be serialized to byte array and sent by means of RabbitMQ.
2. Complete the Publisher class:
    a. Implement Publisher constructor: **Publisher()**. It constructs new Publisher: creates new connection to the server and new channel. Use "localhost" as a host for ConnectionFactory.
    b. Implement **void close()** method. It closes the channel and the connection.
    c. Implement **void send(Person person, String routingKey)** method. It sends person with corresponding routing key to topic exchange.
    This method:
        i. Declares topic exchange.
        ii. Serializes person to byte array.
        iii. Publishes message with routing key to declared exchange.
3. Complete the Subscriber class:
    a. Implement Subscriber constructor: **Subscriber(String id, String[] routingKeys)**. It constructs new Subscriber: sets subscriber's id, routing keys; creates new connection to the server and new channel. Use "localhost" as a host for ConnectionFactory.
    b. Implement **void subscribe()** method. It subscribes to receive messages if its routing keys matches subscriber's ones.
    This method:
        i. Declares topic exchange.
        ii. Creates a non-durable, exclusive, autodelete queue with a generated name.
        iii. Binds that queue to each routing key we are interested in.
        iv. Creates new DefaultConsumer with overridden handleDelivery method, which deserializes message body to Person object and prints

it. (Bind subscriber's id to consumer's tag for clear logging inside handleDelivery method.)

v. Starts consuming messages from the declared queue.

4. Make sure you have added logging and run the Starter class to check whether everything works fine.

## Lab3

**Duration:**

60 min

**Objectives:**

Get familiar with Remote Procedure Call pattern in terms of RabbitMQ.

**Description:**

Use RabbitMQ to build RPC system: client and RPC server.

Client sends requests to server to evaluate factorial of the given number. Server does all calculations and sends reply with result to the client.

Since our client delivery handling is happening in a separate thread, we need something to suspend main thread before response arrives. We'll use BlockingQueue to store response as we need to block client until the answer from the server is received.

Our RPC will work like this:

- When the client sends request to server, it creates an anonymous exclusive callback queue.
- Client sends a message with replyTo property, which is set to the callback queue.
- The request is sent to a special queue for requests.
- Server is waiting for requests on that queue. When a request appears, it does the job and sends a message with the result back to the client, using the queue from the replyTo field.
- The client waits for data on the callback queue. When a message appears, it returns the response to the application.

**Task:**

1. Complete the Server class:
    a. Implement Server constructor: **Server()**. It constructs new Server: creates new connection to the server and new channel. Use "localhost" as a host for ConnectionFactory.
    b. Implement **void close()** method. It closes the channel and the connection.
    c. Implement **int factorial(int number)** method that returns factorial of the given number.
    d. Implement **void receive()** method. It subscribes to receive messages on the queue for requests (QUEUE_NAME). When a request appears, it evaluates factorial and sends a message with the result back to the client, using the queue from the replyTo field.
    This method:

i. Declares non-durable, non-exclusive, non-autodelete queue for requests (QUEUE_NAME).
ii. Creates new DefaultConsumer with overridden handleDelivery method, which retrieves number from message body and evaluates factorial of it; after that the message with result is published to the queue from the replyTo field.
iii. Starts consuming messages from the declared queue.

2. Complete the Client class:
   a. Implement Client constructor: **Client()**. It constructs new Client: creates new connection to the server and new channel. Use "localhost" as a host for ConnectionFactory.
   b. Implement **void close()** method. It closes the channel and the connection.
   c. Implement **Integer call(int number)** method. Sends request to server to evaluate factorial of the given number. Blocks until the answer from the server is received.
      This method:
      i. Declares non-durable, exclusive, autodelete callback queue with a generated name.
      ii. Creates AMQP.BasicProperties object with replyTo property, which is set to the callback queue.
      iii. Converts number to byte array and publishes it to request queue (REQUEST_QUEUE_NAME).
      iv. Creates BlockingQueue to store response as we need to block client until the answer from the server is received.
      v. Creates new DefaultConsumer with overridden handleDelivery method, which retrieves number from message body, which is the result of factorial calculations, and writes it to BlockingQueue.
      vi. Starts consuming messages from the callback queue. Blocks until the answer is received.
      vii. Returns evaluated factorial value.
3. Run the Starter class to check whether everything works fine.

## Lab4

**Duration:**

30 min

**Objectives:**

Practice in using RabbitMQ JMS Client. Learn how a JMS application can send messages to a predefined RabbitMQ 'destination' (exchange/routing key) using the JMS API in the normal way. Similarly, learn how a JMS application can receive messages from RabbitMQ queue using the JMS API.

**Description:**

Use RabbitMQ JMS Client on top of RabbitMQ Java Client to send text message by RabbitMQ sender and receive it by JMS receiver.

Similarly, make it possible to send text message by JMS sender and receive it by RabbitMQ receiver.

**Task:**

1. Take a look at JNDI .bindings file located at src/main/resources/rabbitmq-bindings. See how Context.INITIAL_CONTEXT_FACTORY and Context.PROVIDER_URL were passed as parameters to application in order to create a naming context out of them:

```
Properties environmentParameters = new Properties();
environmentParameters.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.sun.jndi.fscontext.RefFSContextFactory");
environmentParameters.put(Context.PROVIDER_URL,
      "file:src/main/resources/rabbitmq-bindings");
InitialContext initialContext =
      new InitialContext(environmentParameters);
```

   And how this context was used for object lookup:

```
ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("ConnectionFactory");
Queue queue = (Queue) initialContext.lookup("Lab4Queue");
```

2. See how the JMSReceiver class and the JMS Sender class were implemented.
3. Complete the RabbitMQReceiver class:
   a. Implement **void receive()** method. It subscribes to receive messages sent by JMSSender.
      This method:
      i. Creates new connection to the server and new channel. (Using "localhost" as a host for ConnectionFactory.)
      ii. Declares queue to receive messages from JMSSender.
      iii. Creates new DefaultConsumer with overridden handleDelivery method, which converts message body to text and prints it.
      iv. Starts consuming messages from the declared queue.
4. Complete the RabbitMQSender class:
   a. Implement **void send()** method. It sends text message "Hello World!", so that JMSReceiver could receive it.
      This method:
      i. Creates new connection to the server and new channel. (Using "localhost" as a host for ConnectionFactory.)
      ii. Declares queue to send messages to JMSReceiver.
      iii. Publishes message to declared queue.
      iv. Closes channel and connection.
5. Run the JMSRabbitStarter class and the RabbitJMSStarter class to check whether everything works fine.