Pivotal RabbitMQ

Lab Instructions

Version 3.6.1b

This Page Intentionally Left Blank

2016

Table of Contents

Chapter 1. Setting Up the IDE

1.1. Overview

The labs for this training are shipped as Maven projects. They don't depend on any proprietary libraries, so you can import them as Maven projects if your IDE supports Maven.

You require an Internet connection to download the dependencies. You might encounter download issues if you're behind a firewall, or a proxy. The ZIP file of the labs contains all the dependencies. Decompress the ZIP file in a directory. Refer to this directory as LAB_HOME.

The projects must be imported into your IDE, instructions are provided below for Eclipse. The Spring Tool Suite IDE, based on Eclipse, is recommended. There are two distinct cases: whether your Eclipse distribution has support for Maven or not.

1.2. Eclipse Distribution with Maven Support (e.g., Spring Tool Suite, or m2eclipse)

1. Create a settings.xml file in any directory with the following content:

```
<settings>
  <localRepository>/home/rabbit/repository</localRepository>
  <offline>true</offline>
</settings>
```

2. In the localRepository tag, specifiy the `LAB_HOME/repo` directory.
3. Now go back to Eclipse and select “Window” in the menu bar if running on Windows or "Spring Tool Suite" if running on Mac OSX, and then “Preferences”.
4. Expand the “Maven” tree and select “User settings”.
5. For the “User settings” field, click “Browse”.
6. Navigate to the `settings.xml` file you created previously and click OK.
7. Check the “Local repository” field has changed to the `LAB_HOME/repository`.
8. Click OK to close the preferences panel.
9. Select “File” in the menu bar, and then “Import”.
10. Choose “Existing projects into workspace” in the “General” menu item.
11. Click on “Browse”, navigate to `LAB_HOME/rabbitmq-x.x.x.RELEASE`, and click OK.
12. All the projects should appear in the panel. Ensure they're all checked and click on “Finish”

That's it, all the projects are in your Eclipse workspace, you're ready to work!

1.3. Eclipse Distribution without Maven Support

1. Select “Window” in the menu bar, and then “Preferences”.
2. In the filter, type “classpath” and choose “Classpath Variables”.
3. Select "New” to create a new variable called `M2_REPO`. If there’s already an `M2_REPO` variable and it’s marked as non-modifiable, your IDE probably has support for Maven, so you should use the corresponding procedure.
4. The `M2_REPO` classpath variable should point to the `LAB_HOME/repository` directory.
5. Click OK to close the preferences panel.
6. Select “File” in the menu bar, and then “Import”.
7. Choose “Existing projects into workspace” in the “General” menu item.
8. Click “Browse”, navigate to `LAB_HOME/rabbitmq-x.x.x.RELEASE`, and click OK.
9. All the projects should appear in the panel. Ensure they’re all checked and click "Finish”.

That’s it, all the projects are in your Eclipse workspace, you’re ready to work!

Chapter 2. Installation and Management

2.1. Objective

In this lab, you will install RabbitMQ server. At the end of this lab, you will have a working RabbitMQ installation, with the management plugin enabled.

This lab is divided into 2 parts:

1. Install RabbitMQ instance.
2. Enable the management console, and use the web user interface to configure and monitor the RabbitMQ artifacts, such as exchanges and queues.

2.2. Installation

Depending on your system, execute the instructions from the slides (lesson) to install RabbitMQ. Windows users can use the Erlang installer and the RabbitMQ installer provided with the training material.

2.2.1. Windows

The Windows installer installs RabbitMQ as a service. After the installation, the RabbitMQ service will automatically start.

Execute the following steps to verify whether the RabbitMQ service is started:

1. Open a command prompt in the `RABBITMQ_INSTALL_DIR/rabbitmq_server-xxx/sbin` directory.
2. Run the rabbitmqctl.bat status command. You should see the status of the RabbitMQ instance as running.

```
>rabbitmqctl.bat status
Status of node 'rabbit@mybox' ...
[{pid,3544},
 {running_applications,[{rabbit,"RabbitMQ","3.6.1"},
                        {mnesia,"MNESIA  CXC 138 12","4.13.3"},
                        {xmerl,"XML parser","1.3.10"},
                        {os_mon,"CPO  CXC 138 46","2.4"},
                        {rabbit_common,[],"3.6.1"},
                        {ranch,"Socket acceptor pool for TCP protocols.",
                               "1.2.1"},
                        {sasl,"SASL  CXC 138 11","2.7"},
                        {stdlib,"ERTS  CXC 138 10","2.8"},
                        {kernel,"ERTS  CXC 138 10","4.2"}]},
 {os,{win32,nt}},
 {erlang_version,"Erlang/OTP 18 [erts-7.3] [64-bit] [smp:8:8] [async-threads:30]\n"},
 {memory,[{total,49057088},
          {connection_readers,0},
          {connection_writers,0},
          {connection_channels,0},
          {connection_other,0},
          {queue_procs,2712},
          {queue_slave_procs,0},
          {plugins,0},
          {other_proc,22075608},
          {mnesia,61480},
          {mgmt_db,0},
          {msg_index,41024},
          {other_ets,927616},
          {binary,23792},
          {code,17432179},
          {atom,662409},
          {other_system,7830268}]},
 {alarms,[]},
 {listeners,[{clustering,25672,"::"},{amqp,5672,"::"},{amqp,5672,"0.0.0.0"}]},
 {vm_memory_high_watermark,0.4},
 {vm_memory_limit,6832155852},
 {disk_free_limit,50000000},
 {disk_free,183844171776},
 {file_descriptors,[{total_limit,8092},
                    {total_used,2},
                    {sockets_limit,7280},
                    {sockets_used,0}]},
 {processes,[{limit,1048576},{used,162}]},
 {run_queue,0},
 {uptime,78},
 {kernel,{net_ticktime,60}}]
```

`Note:` You might encounter a "Failed to create cookie file" error. This is because the HOMEDRIVE environment variable is not correctly set. To solve this issue, set the environment variable to the appropriate disk drive, where you installed Erlang and the RabbitMQ instance. For example, if you installed both the software on 'C' drive, specify the following before executing any RabbitMQ commands:

```
set HOMEDRIVE=C:
```

Now, let's stop and restart the server.

1. Run the `rabbitmqctl stop` command. The server should confirm it's stopped.
2. Run the `rabbitmq-server` command. The server confirms it's started:

```
>rabbitmq-server

              RabbitMQ 3.6.1. Copyright (C) 2007-2016 Pivotal Software, Inc.
  ##  ##      Licensed under the MPL.  See http://www.rabbitmq.com/
  ##  ##
  ##########  Logs: C:/Users/myuser/AppData/Roaming/RabbitMQ/log/RABBIT~1.LOG
  ######  ##        C:/Users/myuser/AppData/Roaming/RabbitMQ/log/RABBIT~2.LOG
  ##########
              Starting broker... completed with 0 plugins.
```

`Note:`The installer creates shortcuts for all these commands in the Start Menu.

Congratulations, you completed the first part of the lab.

2.2.2. Mac OSX

To install RabbitMQ on Mac OSX, you will use Homebrew. Homebrew is a package manager, which installs the UNIX tools in Mac OSX.

Before executing the install commands, make sure the CC (C compiler) or GCC (GNU Compiler Collection) is installed. This can be verified by typing CC or GCC in your terminal. If the execution of the command outputs, "c: command not found", then you'll have to install the Command Line Tools for Xcode. It's a download that includes all of the tools required by Homebrew. The Command Line Tools for Xcode can be found in https://developer.apple.com/downloads/index.action

1. To install Homebrew, execute the following command in the terminal.

```
/usr/bin/ruby -e "$(/usr/bin/curl -fksSL https://raw.github.com/Homebrew/homebrew/go/install)"
```

   The script installs Homebrew to `/usr/local` directory.
2. To ensure that you have installed Homebrew properly, execute the following command

```
brew doctor
```

3. Install RabbitMQ server by executing the following command:

```
brew install rabbitmq
```

   `Note:` The RabbitMQ server scripts are installed into `/usr/local/sbin`. This is not automatically added to your path. To add the directory to the path, add the following command

```
PATH=$PATH:/usr/local/sbin
```

   to your `.bash_profile` or `.profile`.

Let's check if the server is properly running.

1. Run the `rabbitmqctl status` command. You should see RabbitMQ running:

```
$ rabbitmqctl status

Status of node rabbit@C02GW18DDV16 ...
[{pid,13252},
 {running_applications,[{rabbit,"RabbitMQ","2.7.1"},
                        {mnesia,"MNESIA  CXC 138 12","4.6"},
                        {os_mon,"CPO  CXC 138 46","2.2.8"},
                        {sasl,"SASL  CXC 138 11","2.2"},
                        {stdlib,"ERTS  CXC 138 10","1.18"},
                        {kernel,"ERTS  CXC 138 10","2.15"}]},
 {os,{unix,darwin}},
 {erlang_version,"Erlang R15B (erts-5.9) [source] [64-bit] [smp:4:4] [async-threads:30] [hipe] [kernel-poll:true]\n"},
 {memory,[{total,26977056},
          {processes,10248086},
          {processes_used,10248072},
          {system,16728970},
          {atom,504409},
          {atom_used,475746},
          {binary,419968},
          {code,11916223},
          {ets,906864}]},
 {vm_memory_high_watermark,0.3999999999174177},
 {vm_memory_limit,2906192281}]
...done.
C02GW18DDV16:~ admin$ rabbitmqctl status
Status of node rabbit@C02GW18DDV16 ...
[{pid,13252},
 {running_applications,[{rabbit,"RabbitMQ","2.7.1"},
                        {mnesia,"MNESIA  CXC 138 12","4.6"},
                        {os_mon,"CPO  CXC 138 46","2.2.8"},
                        {sasl,"SASL  CXC 138 11","2.2"},
                        {stdlib,"ERTS  CXC 138 10","1.18"},
                        {kernel,"ERTS  CXC 138 10","2.15"}]},
 {os,{unix,darwin}},
 {erlang_version,"Erlang R15B (erts-5.9) [source] [64-bit] [smp:4:4] [async-threads:30] [hipe] [kernel-poll:true]\n"},
 {memory,[{total,26949320},
          {processes,10215822},
          {processes_used,10215808},
          {system,16733498},
          {atom,504409},
          {atom_used,475775},
          {binary,421480},
          {code,11916223},
          {ets,906856}]},
 {vm_memory_high_watermark,0.3999999999174177},
 {vm_memory_limit,2906192281}]
...done.
```

`Note:` Depending on your system, you might require to run rabbitmqctl (for other RabbitMQ commands as well).

Now, let's stop and restart the server.

2. Run the `rabbitmqctl stop` command. The server confirms it's stopped:
3. Run the `rabbitmq-server` command. The server confirms it's started:

```
$ rabbitmq-server

Activating RabbitMQ plugins ...
0 plugins activated:

+---+   +---+
|   |   |   |
|   |   |   |
|   |   |   |
|   +---+   +-------+
|                   |
| RabbitMQ  +---+   |
|           |   |   |
|   v2.7.1  +---+   |
|                   |
+-------------------+
AMQP 0-9-1 / 0-9 / 0-8
Copyright (C) 2007-2011 VMware, Inc.
Licensed under the MPL.  See http://www.rabbitmq.com/

node           : rabbit@C02GW18DDV16
..
broker running
```

2.2.3. Linux and Unix

If you used the package manager to install RabbitMQ, the broker should already be running, because it is installed as a service.

If you used the generic UNIX package, you need to start the broker with the rabbitmq-server command.

Execute the following steps to verify whether the RabbitMQ service is started:

1. Run the `rabbitmqctl status` command. You should see the status as running:

```
$ rabbitmqctl status

Status of node 'rabbit@vm-master1' ...
[{pid,14400},
 {running_applications,[{rabbit,"RabbitMQ","3.6.1"},
                        {mnesia,"MNESIA  CXC 138 12","4.13.3"},
                        {os_mon,"CPO  CXC 138 46","2.4"},
                        {xmerl,"XML parser","1.3.10"},
                        {rabbit_common,[],"3.6.1"},
                        {ranch,"Socket acceptor pool for TCP protocols.",
                               "1.2.1"},
                        {sasl,"SASL  CXC 138 11","2.7"},
                        {stdlib,"ERTS  CXC 138 10","2.8"},
                        {kernel,"ERTS  CXC 138 10","4.2"}]},
 {os,{unix,linux}},
 {erlang_version,"Erlang/OTP 18 [erts-7.3] [source-d2a6d81] [64-bit] [async-threads:64] [hipe] [kernel-poll:true]\n"},
 {memory,[{total,41042560},
          {connection_readers,0},
          {connection_writers,0},
          {connection_channels,0},
          {connection_other,0},
          {queue_procs,2680},
          {queue_slave_procs,0},
          {plugins,0},
          {other_proc,18476368},
          {mnesia,58200},
          {mgmt_db,0},
          {msg_index,45248},
          {other_ets,885096},
          {binary,18904},
          {code,17392636},
         {atom,662409},
         {other_system,3501019}]},
 {alarms,[]},
 {listeners,[{clustering,25672,"::"},{amqp,5672,"::"}]},
 {vm_memory_high_watermark,0.4},
 {vm_memory_limit,416628736},
 {disk_free_limit,50000000},
 {disk_free,40166559744},
 {file_descriptors,[{total_limit,924},
                    {total_used,2},
                    {sockets_limit,829},
                    {sockets_used,0}]},
 {processes,[{limit,1048576},{used,137}]},
 {run_queue,0},
 {uptime,1472},
 {kernel,{net_ticktime,60}}]
```

`Note:` Depending on the system, you need to execute `rabbitmqctl` with `sudo` (for other RabbitMQ commands as well).

Now, let's stop and restart the server.

2. Run the `rabbitmqctl stop` command. The server confirms it's stopped:
3. Run the `rabbitmq-server` command. The server confirms it's started:

```
$ rabbitmq-server

              RabbitMQ 3.6.1. Copyright (C) 2007-2016 Pivotal Software, Inc.
  ##  ##      Licensed under the MPL.  See http://www.rabbitmq.com/
  ##  ##
  ##########  Logs: /var/log/rabbitmq/rabbit@vm-master1.log
  ######  ##        /var/log/rabbitmq/rabbit@vm-master1-sasl.log
  ##########
              Starting broker... completed with 0 plugins.
```

Congratulations, you completed the first part of the lab!

2.3. Using the Management Plugin

In this part, you will use the management plugin to send and receive a message.

1. Enable the management plugin.
2. Restart the server.
3. Access the web UI from a browser and explore the different tabs.

Now, what about sending a first message to an exchange and consume it from a queue? Perform the following steps to accomplish this task:

1. Click on the "Queues" tab

2. Expand the "Add a new queue" menu
3. Add a `stock.us` queue - you can keep default values for properties. With AMQP, messages aren't sent to the queues, but sent to the exchanges. The exchanges are bound to queues. The next step is to bind an exchange to a queue.
4. Click on the "Exchanges" tab
5. Select the amq.fanout exchange
6. Expand the "Binding" menu
7. In the "Add binding" form, enter `stock.us` and click "Bind". Time now to send the message.
8. Expand the "Publish message" menu
9. Enter a random string in the Routing Key and Payload fields, and click Publish Message. You should get a confirmation.
10. Now go to the queues tabs and select the queue we created. Note, you can check the binding in the "Bindings" menu of the details of a queue.
11. Expand the "Get messages" menu and click on "Get Messages". Your message should appear below.

`Note:` Don't worry if you don't know what a fanout is, you'll learn more about the types of exchanges in the next module.

Congratulations, you installed RabbitMQ, its management console, and published the first message!

If you have some time before the instructor moves on to the next topic, you can explore the following topics:

1. Locate and check the content of configuration files and log files (their path is indicated on the slides)
2. Check the (?) icons of the menus you used to publish the message and bind the exchange to the queue.

Chapter 3. Java Message Sender and Receiver

3.1. Objective

In this lab, you will use the RabbitMQ's Java binding to send and receive messages. The messages can either be consumed by subscription (asynchronous reception) or one-by-one consumption.

You'll use a simple quotation use-case for this lab.

At the end of this lab, you'll have a good understanding of how to use the Java binding.

This lab requires basic Java and Eclipse skills. If you're not comfortable with these technologies, don't hesitate to ask the instructor for a quick demo of Java development with Eclipse.

3.2. Setting Up the Exchange and the Queue

Create a `quotations` fanout exchange, and bind it to a `quotations` queue.

3.3. Sending Messages

You'll use a Java program with an infinite loop to send messages. Messages will be sent after every one second.

1. Open the `QuotationSender` class.
2. Write the Java code to connect to the broker, and create a channel.
3. Launch the program to test the connection.
4. Create an infinite loop (`while(true)`).
5. In the loop, wait (use the `letsWait()` method) and send a message to the `quotations` exchange with a routing key, namely `nasq`. The payload of the message is the result of a call to the `next()` method of the `QuotationService` (don't hesitate to take a look at the code of the `QuotationService`).
6. Execute the program.

3.4. Checking Messages with the Management Plugin

Let's check if the messages arrived correctly in the queue.

1. In the web UI of the management plugin, check the messages in the `quotations` queue.

2. Purge the queue

3.5. Asynchronous Consumption

Write a Java program that displays the quotations on the console as soon as they arrive on the queue. Asynchronous consumption is the simplest way to do that.

1. Open the `QuotationConsumer` class.
2. Write the Java code to asynchronously consume the messages and display the quotation on the console (use the `String(byte[])` constructor to convert the payload into a Java `String`).
3. Execute the sender program, and the consumer program (in any order). You should see the quotations on the console.

Congrats! You're done with the part of the lab.

3.6. Retrieving Messages

You can explicitly ask for the messages with the basicGet method. If there's no message on the queue, you'll get null. You will use the Java binding to send a message and receive it immediately afterwards. This time, you'll be writing a JUnit test.

1. Be sure to stop any Java program that could already be running and to purge the `quotations` queue.
2. Open the `SendingAndReceivingTest` class. The structure has already been provided, and you just need to implement the `sendAndReceiving()` method.
3. Connect to the broker and send a quotation to the quotations exchange (use the `QuotationService` to generate the quotation).
4. Run the test. With the management plugin, check the quotation is on the queue (don't forget to purge the queue afterwards).
5. After sending the message, do a `basicGet()` on the queue to retrieve the message. Write a while loop that checks whether the `GetResponse` is `null` or not, and does a maximum of 5 attempts.
6. It's time now for some JUnit assertions. After the reception, assert the `GetResponse` isn't null with the `Assert.assertNotNull` JUnit call.
7. Re-create the quotation from the payload of the received message, and compare it with the quotation you sent.
8. Run the test.

Congratulations, you're done with this lab.

3.7. Bonus

Improve your test by purging the queue at the beginning. This will avoid the failure of the test, if there are existing messages on the queue. You can also declare the queue, the exchange, and the binding in the test. Remember that declarations are idempotent: it works even the resource (exchange, queue) already exists, i.e. you don't get an error saying there's a conflict. The limitation is the resource must be EXACTLY the same (e.g., you can't declare a non-durable `quotations` queue if a durable `quotations` queue already exists).

Chapter 4. Using the Full Routing Capabilities of RabbitMQ

4.1. Objective

In this lab, you will discover the routing capabilities of RabbitMQ. You know by now messages are sent to exchanges and consumed from queues. This lab is about what can happen between the exchanges and queues, as they can be bound to each other.

The lab is made of two parts: first, you'll practice with the different types of exchanges from the management plugin. Second, you'll implement a couple of typical messaging patterns thanks to RabbitMQ's routing features.

By the end of this lab, you'll have strong foundations about the AMQP model and what you can achieve with it in your own systems.

4.2. Exchange Types and Bindings

Remember, there are 4 types of exchanges in AMQP: fanout, direct, headers, and topic. You saw them in the slides, but it's always good to practice by yourself to really understand how things work. In this part, we'll see the 4 types of exchanges one after the other, by sending messages to them and see in which queue(s) they end up. Let's start by defining our queues!

4.2.1. Queue Definitions

We keep on working with our quotation use case: the queues will represent different stock markets.

1. From the management web UI, define 4 queues: `us` (for US stocks), `eu` (for European stocks), `nyse` (for US stocks exchanged at New York), and `world` (for all the stocks).

Let's start working the simplest kind of exchange: fanout.

4.2.2. Fanout

The fanout does simple broadcasting: it sends messages to all the queues bound to it (a fanout exchange doesn't care about the routing key).

1. Create a `stock.fanout` exchange of type `fanout`.

2. Go to the details page of the fanout.
3. Bind the `us` queue to the exchange.
4. Send a message to the exchange and check it arrived on the us queue

Play around with the binding to the queue (specify or not a value for “Routing key”) and the routing key of the messages you’re sending. You’ll see the messages always arrive to the queue; a fanout doesn’t care about the routing key!

Let's bind other queues:

1. Bind the `eu` and `world` queues to the fanout.
2. Send messages to the exchange and check they all arrive on the queues.

You can purge the queues and remove the bindings.

OK, enough with the fanout for now, let’s work with a direct exchange!

4.2.3. Direct

The direct exchange uses the routing key of a message to route to a queue bound with the same key.

1. Create a `stock.direct` exchange of type `direct`.
2. Bind the `eu` queue to the exchange with the routing key `market.eu`.
3. Bind the `us` queue to the exchange with the routing key `market.us`.
4. Send a message with the routing key `market.eu` and check it ends up in the `eu` queue.
5. Send a message with the routing key `market.us` and check it ends up in the `us` queue.
6. Send a message with the routing key `market.world` and check it is dropped.

So far, so good, but wouldn’t it be great if the world queue could also receive messages?

1. Create 2 new bindings to the world queue: one with `market.us` for the routing key and another with `market.eu` for the routing key.
2. Send messages with `market.us` and `market.eu` for the routing keys and check where they end up.

OK, we’re done with the direct exchange. Clean up the queues before moving on to the next type of exchange.

4.2.4. Headers

A headers exchange ignores the routing key, it works against headers in the message that it matches with arguments specified in the binding.

1. Create a `stock.headers` exchange of type `headers`.
2. Bind the `eu` queue, specify no routing key, and add a parameter `market` with value `eu`.

3. Bind the `us` queue, specify no routing key, and add a parameter `market` with value `us`.
4. Send a message with a market header equal to `eu` and check the message arrived on the `eu` queue.
5. Send a message with a market header equal to `us` and check the message arrived on the `us` queue.

Like with the direct exchange, you can bind the world queue twice, with two different values for the market header and check messages for the different market places arrive also on the world queue.

Purge the queues and let's see the topic exchange.

4.2.5. Topic

The topic exchange is powerful, as it can use wildcards in the routing key to route message to queues.

Here are some examples of what we want to do with our quotations:

- A message with a routing key starting with `stock.us` should go to the `us` and `world` queues.
- A message with a routing key equal to `stock.us.nyse` should go the `nyse` queue… but also to the `us` and `world` queues!
- A message with a routing key starting with `stock.eu` should go to the `eu` and `world` queues.
- Any message with a routing key starting with `stock` should go to the `world` queue.

We guided you for the previous bindings, now you're on your own. With four bindings, dots, the * and # wildcard, this should be fairly easy!

When you're done, move on to the next part, where you'll see exchanges, bindings, and queues in action.

4.3. Patterns

In this part, we cover a couple of typical messaging patterns. Exchanges, bindings, and queues are just features; they don't provide much functionality per-se. Nevertheless, they provide foundations to implement patterns commonly needed and used in messaging applications.

The patterns covered here are independent; you can choose to focus on only one if you're a little short of time.

4.3.1. Publish/Subscribe

The publish/suscribe pattern consists in delivering a message to multiple consumers. This pattern provides good decoupling: the sender doesn't know anything about the consumers (as usual with messaging) and adding a new feature to the system consists in adding a new consumer.

This pattern can be easily implemented with RabbitMQ thanks to a fanout. Remember you just need to bind a queue to a fanout, so that the messages sent to the fanout end up in the queue. If you bind several queues, the messages are delivered to all of them.

In our publish/subscribe use case, a producer sends stock quotations every second. One consumer would log these quotations into a file (the “logger”), the other consumer would display the last ten quotations in web page (the “web UI”).

Let’s focus on the sender first.

1. Open the `QuotationSender` class and check its content. It sends quotations to an exchange.
2. In the `QuotationSender` class, write the code to declare the exchange (a fanout), just after the channel creation.

Feel free to run the program and to verify it’s working properly with the management plugin (by creating a dummy queue for example).

Now let’s see the file logger.

1. Open the `QuotationLogger` class. Some code has already been written for you.
2. Follow the instructions in the code to finish the `QuotationLogger`.

You can now run the sender and the logger, and check the log file (it’s at the root of the lab project). Don’t hesitate to ask to the instructor if something goes wrong.

When the programs work fine, stop them. Time to focus on the web UI consumer.

1. Open the `QuotationWebUi` class. Its job is to memorize the quotation and display the last ten ones in a web page. The web part has already been written for you. Don’t hesitate to have a look if you’re curious.
2. Follow the instructions in the code to finish the `QuotationWebUi`.

The `QuotationWebUi` uses asynchronous consumption. Note our program isn’t bullet-proof: if it runs a long time, it will consume all the memory, as the quotation collection is never cleaned up (a background thread should clean it up from times to times). It does the job for now!

Launch the sender and the web UI. Check the web page at the following URL: http://localhost:8085.

Run the logger at the same time, and the messages will be processed by two consumers at the same. Congratulations, you made it through the publish/subscribe pattern!

4.3.2. Request/Reply (aka RPC)

Messaging is very useful for fire-and-forget scenarios: the sender publishes a message and doesn’t wait any response, at least it doesn’t need a response to move ahead. But sometimes, the sender needs a response to its request, and RabbitMQ makes this synchronous request/reply scenario possible. The advantage to use RabbitMQ for such a scenario is the sender doesn’t know anything about who processed its request: it can be any consumer, anywhere. All of this, thanks to the decoupling provided by exchanges and queues.

The trick to handle request/reply with RabbitMQ is the following: the sender must provide a reply queue in the request message and wait for the response on this queue immediately afterwards he has sent the request. The consumer must be a good citizen and send back the response on the reply queue.

Our request/reply use case will consist in sending a stock quotation message and wait for a response to know if we should buy some of the stocks (the payload of the response message will contain “yes” or “no”).

Let’s have a look at the sender first.

1. Open the `QuotationSubmitter` class.
2. Follow the instructions in the code to finish the `QuotationSubmitter`.

The `QuotationSubmitter` uses the simplest way to handle the request/reply pattern: it dynamically creates the reply queue. This queue will be used only for the response it awaits. This techniques works but doesn’t scale with a lot of senders. A more advanced techniques consists in sharing a reply queue between multiple senders and filtering responses thanks to a correlation ID.

Once you’re done with the `QuotationSubmitter`, you can work on the Java program that handles the request and sends back a response.

1. Open the `QuotationOfferDecider` class.
2. Follow the instructions in the code to finish the `QuotationOfferDecider`.

The `QuotationOfferDecider` is a simple asynchronous consumer. Its only purpose is to send a response message immediately afterward the reception.

It’s time to test: launch the QuotationOfferDecider and then submit some quotations by launching several times the QuotationSubmitter. You should see the submitter outputting the response of the decider.

Congratulations, you’re done with the lab!

4.4. Bonus

What if the decider takes a long time to decide and the submitter can’t actually wait? This is a common requirement: you don’t want a user or a system to wait too long after a request, 5 seconds max for example. But with distributed systems, it’s always difficult to estimate the duration of complex processing. So the “fail fast” philosophy consists in waiting a given amount of time, giving up if the processing takes too long, and giving back a negative answer to the requester (or something like “the processing takes too long right now, you’ll have your response in your mail box ASAP”). The “fail fast” is easy to implement with RabbitMQ. Modify the `QuotationSubmitter` to wait for only 5 seconds for the response. Output something accordingly on the console. To simulate a long processing on the decider side, add a call to the `letsWait()` method before publishing the response.

Chapter 5. Building a Reliable Message Flow

5.1. Objective

In this lab, you will make the message flow reliable by implementing message durability, message acknowledgment, and transactions.

5.2. Durability and Persistence, Acknowledgements, and Transactions

In this section, you'll configure message persistence, and acknowledgement of a message after being consumed.

5.2.1. Durability and Persistence

What happens if your broker crashes? If you're not careful enough, you can lose all your settings (e.g. exchanges of some specific types). But, even worse, you can lose all the unconsumed messages. Let's see what can go wrong, before configuring a reliable messaging flow.

1. Delete any `quotations` exchange, and any `quotations` queue from the broker.
2. Open the `QuotationSender` class and check its content. It sends quotations every one second. Note that it creates all the resources it requires (exchange, queue, and binding).
3. Launch `QuotationSender`, and let it run a little, just enough to see messaging arriving in the quotations queue (use the management plugin to check).
4. Stop the Java program. Messages stop arriving, but sent messages are still in the queue.
5. Stop the broker with `rabbitmqctl`.
6. Restart the broker and check if the messages are still in the queue.

   There's nothing left: no exchange, no queue, and no message. In a real application, you would lose data in case someone accidently shut down the broker, or because of a server crash.
7. Change `QuotationSender` to make the exchange durable.
8. Do the same verification as previously (run the program, check messages, stop the program, stop and start the broker, check messages). This time, the exchange is still there, but the queue and the messages are gone. The reason to make an exchange durable is, a client cannot create an exchange with the same name and a different type. What you think will happen, if you configured a `quotations` fanout exchange and discover a client application created a `quotations` topic exchange after a broker restart? That could be a nasty security hole.
9. Change `QuotationSender` to make the queue durable.

10. Do the same verification as previously. Still no luck for the messages. They're gone again. But the exchange and the queue survived the restart. You lost data, but making a queue durable allows keeping settings and bindings between restarts.
11. Change the `QuotationSender` to make the sent messages persistent (use the `MessageProperties` class if you want a shortcut for persistent text messages).
12. Do the same verification as done previously. At last, you didn't lose the messages.

You've now learned what you need to avoid losing data. Let's try a couple of other combinations (don't forget to delete the exchange and the queue between the tests).

1. Exchange isn't durable, queue and messages are persistent (you don't lose the messages).
2. Exchange is durable and messages are persistent, queue isn't durable (you lose the messages).

You know by now how to keep your messages and settings between restarts or crashes, let's see now how to properly use transactions and acknowledgments (note that for simplicity's sake, you won't deal with durability and persistence in the remaining parts of the lab).

5.2.2. Transactional Sending

Sometimes messages make a group, they all need to reach the broker or none of them should. An example would be the line items of a specific order. What you think will happen, if you receive only part of your order? And imagine if you were in charge of the store, each partial order would certainly imply a manual operation.

Fortunately, RabbitMQ provides transaction semantics when sending messages. Let's see what it means.

1. Open the `TransactionalQuotationSender` class and take a look at its content. It must send atomically 5 quotations, but half of the time, a poison pill makes one sending fail. Note the program purges the queue it sends messages to on start.
2. Execute the `TransactionalQuotationSender` several times. Each time, check the `transactional.quotations` queue. There's something wrong: when everything goes fine, the queue ends up with 5 messages, but sometimes, it has only 4 messages! That's very bad for our system if you consider there's a business rule saying the messages must be sent atomically (remember the partial orders!).

You need to correct this, but before, let's see the hidden advantages of this solution. You send messages outside a transaction. As soon as you call `basicPublish()` method, the message leaves to the broker. `basicPublish()` returns immediately, without waiting for any confirmation that the broker received the message. You don't know whether the message made it to the broker or not. This is known as "fire and forget". This is fast, but doesn't provide any reliability. Use this when you have tons of messages to send and you don't care about losing messages (you do care about losing line items of an order, but you can live with losing a couple of quotations, this is business-specific).

1. Modify the `TransactionalQuotationSender` to make the sending of the 5 messages atomic.
2. Run the program several times and verify each time there's 0 or 5 messages on the queue.

Transactions are good when you need atomicity, and a guarantee that the message made it to the broker. But, remember they impede performance.

It’s now time to switch to the reception side.

5.2.3. Reliable Reception with Acknowledgments

When a consumer acknowledges a message, it tells to the broker it’s done with the message. When the broker receives the acknowledgment, it knows it can definitely remove the message from the queue. Let’s see what the benefits of acknowledgment are.

1. Open the `FairQuotationConsumer` class and check its content. It consumes messages from the quotations queue. Note you simulate a long processing (2 seconds).
2. If the `quotations` queue already exists, ensure it's empty (purge it from the management console if necessary).
3. Run the `FairQuotationConsumer` class and then the `QuotationSender`.

   The `FairQuotationConsumer` outputs messages on the console. This consumer is a bottleneck: it takes 2 seconds to process a message, whereas a new message comes in every second.
4. Monitor the quotations queue with the management plugin.

   The queue is always empty: as soon as a new message arrives, the consumer takes it.
5. Stop the sender first, and then stop the consumer.
6. Monitor the queue.

The queue is empty. Let’s do the math: if you let the sender run for 60 seconds, it sent 60 messages. As it takes 2 seconds to the consumer to process a message, it must have processed 30 messages. So where are the 30 remaining messages? They’re gone! Even if they were marked as persistent!

What happened? The short story is: the client “swallowed” the messages. Here is the complete explanation. As soon as a new message arrives in the queue, the broker distributes it to the consumer. The consumer uses auto-ack: when it receives a new message, it acknowledges it, and then processes it. So the message is removed from the queue before it has been processed.

A channel uses one thread for its consumers. So the processing is single-threaded, but it doesn’t prevent the channel from receiving messages. Messages are queued on the client side. This kind of queue isn’t persistent at all, messages are kept in memory. If the client is shut down, the queued messages on the client side disappear!

This is all auto-ack’s fault! Always remember that auto-ack can lead to lost messages.

Let’s make this more reliable.

1. Change `FairQuotationConsumer` to use acknowledgments.

2. Purge the `quotations` queue from the management console.
3. Start the consumer and the sender.
4. Monitor the `quotations` queue. You should see it contains un-acked messages.
5. Stop the sender first, and then stop the consumer.
6. Monitor the queue.

You should see that the un-acked messages go back to the queue when the consumer is shut down. Thanks to acknowledgment, you no longer lose messages if the consumer is too slow to process message and crashes.

But is it good to send messages to the consumer as soon as they arrive on the queue? It doesn't process them right away, as it's already busy processing one message. This means the consumer accumulates un-acked messages, and this can saturate its memory.

Fortunately, there's a way to tell the broker not to send another messages before the consumer has acknowledged the previous ones. This is known as fair dispatch. Thanks to fair dispatch, messages can remain safely on the broker side and don't accumulate on the consumer side.

You can achieve this thanks to the `Channel.basicQos(int)` method, where the integer parameter is the maximum number of messages.

1. Modify `FairQuotationConsumer` to use fair dispatch (1 message at a time).
2. Purge the `quotations` queue from the management console.
3. Start the consumer and the sender.
4. Monitor the queue. You should see messages accumulating and that there's only one un-acked message at a time.

If you don't want to see messages accumulating in the broker queue, just start another `FairQuotationConsumer` process. The broker will distribute the messages to both consumers and messages will be processed as they arrive on the queue.

Congrats, you're done with the first part of this lab!

5.3. Multiple Transactional Resources and Best Effort

In this second part, you'll see what can go wrong when the processing of messages implies a transactional resource, like a database.

The system consists of:

- A message sender, `DatabaseQuotationSender`, that sends quotations it also inserts into its own database. By inserting the quotations into the sender database, you'll be able to know exactly what the sender has sent.
- A message consumer, `DatabaseQuotationLogger`, that listens to messages and records them into its own database.

- A Java server, `StartServers`, that runs a Java-based database server and a Java-based HTTP server. This server provides us with a web page that allows comparing the sender and consumer databases.

Don't worry, you don't have to write these classes, you'll only have to tweak them. The point of this part is to simulate errors on the consumer side, see the consequences, and use the appropriate options to prevent bad consequences.

First, you'll check if everything works fine.

1. Start the `StartServers`, `DatabaseQuotationLogger`, and `DatabaseQuotationSender` in this specific order.
2. Go to http://localhost:8085/summary. You should see a web page, with two parts: on the left the sender information, and on the right side, the consumer information. As you haven't introduced errors yet, information is exactly the same: same number of quotations, same chart.

You can refresh the web page to see the evolution of the quotations. Time now to simulate the errors.

1. Stop the 3 programs.
2. In DatabaseQuotationLogger, change the code in the Consumer to the following:

```
Quotation quotation = Quotation.read(new String(body));
if(Math.random() < 0.20) {
  System.out.println("something went wrong, no recording");
} else {
  JdbcUtils.insertQuotationIntoDb(dbConn, quotation);
}
```

3. Start the `StartServers`, `DatabaseQuotationLogger`, and `DatabaseQuotationSender` in this specific order.
4. Go to http://localhost:8085/summary. 20% of the messages aren't recorded by the consumer. You should see random "holes" in the sender chart.

The consumer is using auto-ack. If its processing fails, messages are definitely lost. Losing messages isn't always problematic: in our case, people are usually interested in the last value of a quotation. Our main problem is our chart doesn't look nice! But imagine messages were money transfers, it would be problematic to lose them.

The solution is to use acknowledgments.

1. Stop the 3 programs.
2. Modify DatabaseQuotationLogger to acknowledge messages once they've been processed, like the following:

```
channel.basicConsume(
    CONSUMER_QUEUE, false,new DefaultConsumer(channel) {
  @Override
  public void handleDelivery(String consumerTag, Envelope envelope,
         BasicProperties properties, byte[] body) throws IOException {
    Quotation quotation = Quotation.read(new String(body));
    if(Math.random() < 0.20) {
      System.out.println("something went wrong, no recording");
    } else {
      JdbcUtils.insertQuotationIntoDb(dbConn, quotation);
      channel.basicAck(envelope.getDeliveryTag(),false);
    }
  }
});
```

3. Start the `StartServers`, `DatabaseQuotationLogger`, and `DatabaseQuotationSender` in this specific order.
4. Go to http://localhost:8085/summary. You should see the consumer still misses messages. The messages are not lost, but they can’t be redelivered by the broker.
5. Stop the `DatabaseQuotationSender`, and the `DatabaseQuotationLogger`.
6. Monitor the queue with the management plugin. Messages are still in the queue.
7. Start the `DatabaseQuotationLogger` again. It will dequeue the messages again, use the web page to check. As the consumer still fails for 20% of the messages, you’ll need perhaps to stop and relaunch the consumer several times, before the consumer has the same number of quotations as the sender. You don’t lose messages any more, but something else can go wrong. What if the network fails after the processing, but before the acknowledgment? Let’s find out.
8. Make sure the 3 programs are stopped.
9. Modify the consumer as follows:

```
channel.basicConsume(CONSUMER_QUEUE, false,
                                 new DefaultConsumer(channel) {
 @Override
 public void handleDelivery(String consumerTag,Envelope envelope,
                        BasicProperties properties, byte[] body)
                                              throws IOException {
      Quotation quotation = Quotation.read(new String(body));
      JdbcUtils.insertQuotationIntoDb(dbConn, quotation);
      if(Math.random() < 0.20) {
          System.out.println("something went wrong, no ack");
      } else {
          channel.basicAck(envelope.getDeliveryTag(),false);
      }
   }
});
```

10. Start the `StartServers`, `DatabaseQuotationLogger`, and `DatabaseQuotationSender` in this specific order.
11. Go to http://localhost:8085/summary. This time, all the quotations appear in the consumer summary. But the console should tell you sometimes an acknowledgement wasn’t sent (it typically simulates a network failure).
12. Stop the `DatabaseQuotationSender`, and the `DatabaseQuotationLogger`.
13. Monitor the queue with the management plugin. Messages are still in the queue.
14. Start the `DatabaseQuotationLogger` again. It will dequeue the messages again, use the web page to check. Note the consumer has more quotations than the sender.

The case is quite interesting. You processed the same messages multiple times, as you have more messages in the consumer database than in the sender database. But the chart is the same (once all the messages have been dequeued). Inserting in the database isn’t an idempotent operation (issuing the same insert multiple times doesn’t lead to the same result as one single insert). The creation process of the chart is idempotent (adding the same (x,y) coordinates several times doesn’t modify the rendering of the chart).

When acknowledging messages just after the processing, there’s still a small window for failure (just after the processing and before the ack). The window is very small but it does exist. So, you must be sure that the processing is idempotent. You could make our database inserts idempotent: you’d need to check if there’s already a quotation for the given stock at the given time and insert the quotation accordingly.

Congrats, you’ve made it to the end of this lab.

Chapter 6. Clustering

6.1. Objective

In this lab, you will set up a 3-node RabbitMQ cluster on your local machine. You will also create exchanges, queues, and bindings on different nodes of the cluster to see how RabbitMQ replicates these declarations on all the nodes in the cluster.

Note to Mac-OS Homebrew Users: Before starting this lab, disable the STOMP and MQTT plugins to avoid port conflicts.

```
rabbitmq-plugins disable rabbitmq_stomp
rabbitmq-plugins disable rabbitmq_mqtt
```

6.2. Preparing the Nodes for Clustering

You need to set up clean nodes for clustering.

6.2.1. Preparing the First Node

1. Start the first node. The command is different depending on your operating system.

   For Windows users, you need to set up environment variables in dedicated commands and then start the node:

```
> set RABBITMQ_NODE_PORT=5672
> set RABBITMQ_NODENAME=server1
> set RABBITMQ_SERVER_START_ARGS=-rabbitmq_management listener [{port,15672}]
> start rabbitmq-server.bat
```

   Make sure to run each command separately (note a command can be long and doesn’t fit on one line in this document).

   For UNIX-like users, you can start the node in one, long command:

```
$ RABBITMQ_NODENAME=server1 RABBITMQ_NODE_PORT=5672
    RABBITMQ_SERVER_START_ARGS="-rabbitmq_management
    listener [{port,15672}]"
    rabbitmq-server -detached
```

   Again, make sure to launch one command, even if the command doesn't fit on one line of this document.

2. Check the status of the node:

```
rabbitmqctl —n server1 status
```

   Now you're sure the node is running, let's clean it up.

3. Stop and reset the node:

```
rabbitmqctl —n server1 stop_app
rabbitmqctl —n server1 force_reset
```

   Why are you using `force_reset` instead of `reset`?

   A simple `reset` could fail if the node was already part of a cluster and some nodes of this cluster are currently down. `force_reset` is the drastic solution, useful when you want to do some cluster experimentation on a clean RabbitMQ instance (you shouldn't use `force_reset` on a production environment unless you know exactly what you're doing).

4. Restart the application and check the cluster status:

```
rabbitmqctl —n server1 start_app
rabbitmqctl —n server1 cluster_status
```

   You should see something like this:

```
$ rabbitmqctl -n server1 cluster_status
Cluster status of node 'server1@mylaptop' ...
[{nodes,[{disc,['server1@mylaptop ']}]},
 {running_nodes,['server1@mylaptop ']},
 {partitions,[]}]
...done.
```

   The node is single in its cluster. It waits for other nodes to join its cluster. You need perform the exact setup with the 2 other nodes.

6.2.2. Preparing the Other 2 Nodes

Here is a reminder of the commands for the second node.

To start the second node, for Windows users:

```
> set RABBITMQ_NODE_PORT=5673
> set RABBITMQ_NODENAME=server2
> set RABBITMQ_SERVER_START_ARGS=-rabbitmq_management
    listener [{port,15673}]
> start rabbitmq-server
```

To start the second node, for UNIX-like users:

```
$ RABBITMQ_NODENAME=server2 RABBITMQ_NODE_PORT=5673
    RABBITMQ_SERVER_START_ARGS="-rabbitmq_management
    listener [{port,15673}]"
    rabbitmq-server -detached
```

Here are the commands to check the second node has started correctly and get it ready for clustering:

```
$ rabbitmqctl -n server2 status
$ rabbitmqctl -n server2 cluster_status
$ rabbitmqctl -n server2 stop_app
```

For server3, make it listen on port *5674*, and make the management plugin listen on port *15674*.

6.3. Creating the Cluster

In a production environment, the nodes will be on separate machines, and they need to share the same Erlang cookie. As you're working on the same machine, you don't need to deal with the Erlang cookie. But, even if you're working on a local machine, Erlang requires the name of the nodes and the hostname (e.g. nodename@hostname).

To know the hostname of your machine, use the `hostname` command (works on Linux and Windows).

1. Make `server2` join `server1`'s cluster:

```
rabbitmqctl -n server2 join_cluster --ram server1@my-laptop
```

   Note: Change "my-laptop" to your hostname
2. Start `server2`'s application.
3. Check the cluster status for both server1 and server2

   You should see something like:

```
$ rabbitmqctl -n server1 cluster_status
Cluster status of node 'server1@my-laptop' ...
[{nodes,[{disc,['server1@my-laptop]},
         {ram,['server2@my-laptop']}]},
 {running_nodes,['server2@my-laptop',
                 'server1@my-laptop']},
 {partitions,[]}]
...done.
$
```

   Note that `server1` is a disc node, and `server2` joined as RAM node.
4. Repeat the above instructions to add `server3` to the cluster.

Now the cluster is up, let's see how it handles the declaration of exchanges and queues.

6.4. Declaring Resources on the Cluster

You will declare resources (exchanges and queues) on the cluster, and check these resources appear on each node.

1. Connect to `server1` management plugin (http://localhost:15672/). Note that the cluster information is available from a single node (nodes, type of the nodes, AMQP port of the nodes, etc.) The management plugin is cluster-aware.
2. Declare a quotations direct exchange from `server1`.
3. Go to `server2` and `server3` management plugin, and check the exchange has also been created on those nodes.

   An exchange declaration from one of the node is automatically replicated to the other nodes. This means clients can connect to any node to send messages to this exchange.
4. From `server1` management plugin, declare a `market.us` queue. Note the UI to declare a queue is also cluster-aware: it has a "node" field. Choose "server1".
5. Check the queue declaration has replicated to `server2` and `server3`.

   Remember the defaults for a queue in a cluster. Its metadata are available on all the nodes of the cluster, but only one node (`server1`) actually hosts the queue.

   What about bindings? Are they also replicated?
6. Bind the `quotations` exchange to the `market.us` queue from any node management plugin.
7. Check if the binding has been replicated to other nodes.

Let's see how messages are sent and consumed in a cluster.

6.5. Sending and Consuming Messages

1. From `server3` management plugin, send a message to the `quotations` exchange.
2. Verify that the messages from all the nodes reached the queue.
3. Consume the message from `server2`.

Conclusion: Any node can be used to send messages to an exchange. The node uses its local bindings to know where the message should be routed, and where the target queue is (thanks to the cluster metadata). Any node can consume messages from a queue, even if the queue isn't actually hosted on the node.

6.6. Bonus

If you have some extra time, don't hesitate to experiment with RabbitMQ clustering:

- Stop and restart nodes
- Change the type (disk/RAM) of your nodes
- Create multiple clusters on your local system. You created a 3-node cluster. Now create a 3-node cluster, and a 2-node cluster. Switch a node from one cluster to the other.

Chapter 7. High Availability

7.1. Objective

In this lab, you will configure RabbitMQ clusters for high availability.

7.2. Cleaning up the Cluster

You require a 3-node cluster for performing this lab. You should use the cluster reset procedure of the previous lab, if you have experimented a lot with your cluster.

7.3. Failover of a Durable, Non-mirrored Queue

The objective is to see the limitations of a normal queue in a cluster.

1. Create a `quotations` fanout exchange, and a `market.us` durable, non-mirrored queue on `server2`. Bind them together.
2. Simulate `server3` going down by executing the following command:

```
rabbitmqctl –n server3 stop_app
```

   You can check the status of the cluster from `server1` or `server2` management plugin. You see `server3` in red.
3. Ensure the `market.us` queue still appears on `server1` and `server2`. It means the sent messages can still be routed to the `market.us` queue, and consumers can pull messages from the `market.us` queue.
4. Bring up `server3` again:

```
rabbitmqctl -n server3 start_app
```

5. Ensure that `server3` is part of the cluster again.

Conclusion: If a non-owner node of a queue goes down, the cluster works the same way regarding that queue. Connected clients connect to another node, or wait until the node comes up again, to access the queue.

Let's see what happens if `server2` is shut down (`server2` is the owner of the `market.us` queue).

1. Shut down server2:

```
rabbitmqctl –n server2 stop_app
```

2. Check the status of the cluster, server2 should appear as down.
3. Check the `market.us` queue on the management console. It appears as down.

The `market.us` queue disappeared with its owner node. Try to reconnect to another node, and declare the queue.

Connect to `server1` management plugin and try to declare a durable, non-mirrored `market.us` queue. You encounter an error.

You cannot re-declare a durable, non-mirrored queue if its owning node is down. This has profound implications on the clients. Consumers need to wait until the owning node comes up again, before they can consume messages. Publishers must be able to resend their messages to the queue.

Imagine a publishing online store application that sends order messages. The messages are consumed by the inventory system. The online store application has a "confirmed" flag on orders, and could resend unconfirmed orders every hour. How is the confirmed flag set to true? The inventory system publishes a message when it processes an order. The online store application consumes the message and set the confirmed flag of the corresponding order to true. The inventory system can also produce a CSV file with all the processed orders, and the online store application could read the CSV file to confirm its orders. This type of "business acknowledgment" is very common in messaging systems.

Bring back up `server2`. It should reappear in the cluster.

7.4. Failover of a Mirrored Queue

A mirrored queue is replicated to all nodes in the cluster.

1. Declare an appropriate policy to make the `market.us` queue mirrored on all nodes.
2. Check the `market.us` queue became a mirrored queue and the `quotations` exchange is still bound to the queue.
3. Send a message to the `quotations` exchange.
4. Ensure the message is on the queue.
5. Stop `server2`. Verify from the other nodes that the queue still appears in the cluster. A mirrored queue still exists in the cluster even if its master fails.
6. Send some messages that will be routed to the queue, and check if they appear in the queue.
7. Bring back up `server2`.
8. Check the queue status. The management plugin display that one of the slaves (`server2`) isn't synchronized. Remember that, by default, a node doesn't synchronize when it joins the cluster (or restarts in the cluster like `server2`).
9. Launch the `QuotationSender` class. It connects to `server3` and sends a message every second.
10. Check the messages on the market.us queue, and `server2` is still not synchronized. The only way to synchronize `server2` is to consume messages.

11. Launch the `QuotationConsumer` class. It consumes messages from the `market.us` queue. All the slaves should naturally get synchronized.

This part showed a mirrored queue is highly available. A node can fail (even the master node) but the queue is still accessible.

Bonus: If you want to focus on mirrored queues and/or have time, you can try out the slave synchronization features (automatic and manual), as shown in the slides.

Let’s see now how to handle failover from the client side.

7.5. Failover for a Publisher

In this part, you’ll see how a client that publishes messages to a server can handle the server's failure.

1. Delete the `market.us` queue and the HA policy. Re-declare a durable, non-mirrored `market.us` queue. Choose `server2` as the owner of the queue.
2. Bind the `quotations` exchange to the `market.us` queue.
3. Launch the `HaQuotationSender` class. It connects to `server2`, and sends messages to the `quotations` exchange.
4. Check whether the messages are routed to the queue.
5. Stop `server2`. What happens to the Java program? The sender crashes because there’s nothing in the code to handle a failure of the node it is connected to.
6. Make sure the `HaQuotationSender` program isn't running.
7. Surround the sending (`basicPublish` method) with a try/catch block to handle the re-connection:

```
try {
  channel.basicPublish("quotations", "",
     null,quotation.getBytes()
  );
} catch (Exception e) {
  channel = connect();
}
```

8. Restart `server2`.
9. Start `HaQuotationSender` class, and verify that the messages are routed to the queue.
10. Stop `server2`. `HaQuotationSender` doesn’t crash. It tries to reconnect to `server2`.
11. Restart `server2`. HaQuotationSender reconnects and starts sending messages again.

The client survives the crash of the node it is connected to. You can make the sender even more robust by making it choose randomly another node of the cluster (you just have to switch the ports for your local cluster). This couples the client to the nodes, but it could be easily externalized in a real-world application. You can also set up a load balancer in front of the nodes.

Let’s see now failover handling for a consumer.

7.6. Failover for a Consumer

In this part you'll explore two failure scenarios an asynchronous consumer can face. For each scenario, you'll need a sender, such as the `HaQuotationSender` class, that publishes to `server1` (port 5672). Make sure to modify your sender accordingly, or to use the `HaQuotationSender` class in the `ha-solution` project.

7.6.1. Setting up the Resources and the Sender

1. Make sure your cluster has a `quotations` fanout exchange.
2. Make sure your cluster has a non-mirrored, durable `market.us` queue, with `server2` as the owner node.
3. Make sure the queue is bound to the exchange.
4. Launch your quotation sender, and check the messages on the queue.

7.6.2. Failure of a Non-owner Node

The first failure scenario:

A consumer consumes messages from a queue. The node that the consumer is connected to does not own the queue. What happens if this node crashes?

1. Launch `HaQuotationConsumer`. It consumes messages from `server3`. Each time it gets a message, it displays it on the console.
2. Stop `server3`. Do you encounter any error message?
3. Start `server3`. Does the consumer start consuming messages again?

   The answer is no. Imagine if it is a real-world application, and you need to restart the whole application to consume messages again. You need to make the consumer "shutdown-aware".
4. Modify the `createConnection()` method in `HaQuotationConsumer` class to add a ShutdownListener on the newly-created connection. You need to check if the shutdown was explicit, or because of any error. Call the `connectAndListen()` method inside the `shutdownCompleted()` method of the listener in case of a hard error.
5. Make sure the old version of the `HaQuotationConsumer` class isn't running anymore, and start the changed version.
6. Stop `server3`. You see the consumer connecting to another node and consuming messages.

The consumer survives the death of its node. The trick was to add the shutdown listener to reconnect automatically. Note that the reconnection code tries to reconnect randomly. This wouldn't be necessary if you were using a load balancer.

Let's see now another failure scenario: the death of the owner of the queue.

7.6.3. Failure of the Queue Owner Node

The failure scenario:

A consumer consumes messages from a queue. The node that the consumer is connected to does not own the queue. What happens if the node that owns the queue crashes?

1. Make sure all the nodes of the cluster are running, messages are sent to the `market.us` queue, and `HaQuotationConsumer` is consuming from `server3` (it’s the case by default, at least on the first connection).
2. Stop `server2`. Remember that stopping the owner of a non-mirrored queue makes the queue disappear from the cluster. This means messages routed to the queue are discarded, but what about consumers?

   You should see `HaQuotationConsumer` doesn’t receive messages any more. It’s actually trying to consume from a queue that doesn’t exist. Do you encounter any error messages? What happens if you restart `server2`?
3. Restart `server2`.
4. Check with the management plugin that messages are enqueued on `market.us`.
5. Check on the console if `HaQuotationConsumer` receives messages.

You might think `HaQuotationConsumer` would start receiving messages again as soon as `server2` (the owner of the queue) restarts. After all, nothing wrong has happened to the node `HaQuotationConsumer` is connected to (`server3`). This isn’t the case, it’s like `HaQuotationConsumer` is consuming from a phantom queue. Fortunately, RabbitMQ provides an extension when consumers want to be notified that the owner of the queue they’re consuming has failed.

1. Modify the consumer in the `connectAndListen()` method to use the consumer cancellation extension (`handleCancel()` method). You just have to call the `connectAndListen()` method in the callback.
2. Stop the `HaQuotationConsumer` if it’s still running, and restart it.
3. Check whether the messages are consumed.
4. Stop `server2`. You see `HaQuotationConsumer` issuing error messages. It manages to reconnect to another node, but gets an error when trying to consume from `market.us` queue. Remember the queue doesn’t exist anymore because its owner node is down.
5. Restart `server2`. `HaQuotationConsumer` restarts consuming messages.

If you’re using the `HaQuotationSender` from the `ha-solution` project, you can even try to stop `server1`. You can shut down any node and restart it. Both `HaQuotationSender` and `HaQuotationConsumer` will be able to reconnect and consume messages.

7.7. Automatic recovery with the Java client (Optional)

This section is optional, it covers the automatic recovery feature of the RabbitMQ Java client. Don't hesitate to work on this section if you plan to use the Java client.

You've just seen how to make Rabbit clients applications more resilient to failures. Unfortunately, this can be a little tricky. The good news is the Java client has an automatic recovery feature: it can reconnect and re-register listeners after a failure. This is exactly what we're going to test in this section. Remember this is a special feature of the official Java client: not all Rabbit clients - especially the community-driven ones - implement automatic recovery.

Here are the pre-requisites for this section:

1. A 3-node RabbitMQ cluster
2. A `market.us` *mirrored* queue, bound to a `quotations` exchange. The owner of the queue is `server2`.

Here is the scenario we're about to test: a sender program sends a message every second, a consumer program consumes those messages, and the node the client is connected to fails. As the automatic recovery is enabled on the consumer, we'll see it automatically handles the recovery (reconnection and re-registration of listeners.) Let's start!

1. Stop any program running.
2. In `HaQuotationConsumer`, enable the automatic recovery flag (at the `ConnectionFactory` level) and use an array of `Addresses` to create the connection (note this array is already defined in an `ADDRESSES` static property.) The `createConnection` method should look like the following:

```
// defined at the top of the class, don't declare it again
private static final Address [] ADDRESSES = new Address[] {
        new Address("localhost",5672),
        new Address("localhost",5673),
        new Address("localhost",5674)
};

private static Connection createConnection(int port) throws Exception {
  ConnectionFactory factory = new ConnectionFactory();
  factory.setUsername("guest");
  factory.setPassword("guest");
  factory.setVirtualHost("/");

  factory.setAutomaticRecoveryEnabled(true);
  Connection conn = factory.newConnection(ADDRESSES);
  return conn;
}
```

3. If you did the other parts of the lab, be careful to remove (or comment out) the listener you added to the connection: it was there to handle a reconnection and is now useless, as the reconnection will be handled by the Java client itself. Don't forget to also remove the reconnection code in the consumer cancellation method. Remember, this code is in the `basicConsume` method. The listener could now be like this:

```
channel.basicConsume("market.us", true,new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag, Envelope envelope,
             BasicProperties properties, byte[] body) throws IOException {
      System.out.println("receiving quotation: "+new String(body));
    }

    @Override
    public void handleCancel(String consumerTag) throws IOException {
      // NO OP
    }

});
```

4. Launch the `HaQuotationSender` of the `ha-solution` project. It connects to the cluster, and sends messages to the `quotations` exchange. It handles reconnection.
5. Check the messages are routed to the queue.
6. Starts the `HaQuotationConsumer` you modified and ensure the message appears in the console.
7. Thanks to the "Connections" tab in the web UI, find the node the consumer is connected is (it's likely to be `server1`.) Stop this node. What happens to the listener? It should stop for a few seconds and restart listening again, all by itself!

What happened exactly? This is bad scenario: the node the client is connected to has died. In the other sections of the lab, we had to handle the reconnection and re-registration manually, thanks to the shutdown notification RabbitMQ sends. This works, but is a little cumbersome.

With automatic recovery enabled, the Java client handles the reconnection and the re-registration by itself. Note we had to specify the location of the nodes (with the array of `Addresses`.) Remember automatic recovery is specific to the Java client. If you don't use this client, check the documentation of your client to see if it implements a similar feature.

After this lab, you should have learned that failures of nodes aren’t transparent to clients, and may need to be handled explicitly. You used low-level APIs here, but remember some higher-level abstraction like Spring AMQP can make this failover handling more transparent.

Chapter 8. Performance

8.1. Comparing and Benchmarking Classic Configurations

8.1.1. Objective

In this lab, you will see how different factors and configurations impact performance of the broker.

8.1.2. Experiencing Message Size

This first part will show you how the message size impacts performances.

1. Open the `BigQuotationSender` class.
2. Complete the TODO tags and uncomment the line.
3. Run the Java program, and observe the time increasing.

8.1.3. Experiencing the Delivery Mode and Persistence

1. Open the `DeliveryModeSender` class.
2. Complete the TODO tags.
3. Run the Java program, and observe the delta between the 2 different times.

8.1.4. Experiencing the Acknowledgements Mechanism

1. Open the `AcknowledgementBencher` class.
2. Complete the TODO tags.
3. Run the Java program, and observe the delta between the 2 different times.

Congrats! You’re done with this lab. You've learned how client parameters can affect performance.

8.2. Preventing RabbitMQ from Accepting too much Traffic

8.2.1. Objective

In this lab, you will learn how you can prevent the RabbitMQ broker from accepting too much traffic from clients.

By the end of the lab, you'll know how to configure the broker efficiently.

8.2.2. Configuring the High Memory Watermark

By default, RabbitMQ is configured to raise an alarm and turn on the flow controller when the used memory is above 40% of the total installed RAM. In order to speed-up the saturation process, you will decrease this value to 20%.

1. Stop the RabbitMQ broker.
2. Replace the content of the configuration file by the following (remember, the configuration file is `/etc/rabbitmq/rabbitmq.config` on UNIX-like systems, and `%APPDATA%\RabbitMQ\rabbitmq.config` on Windows systems)

```
[
{rabbit, [{vm_memory_high_watermark, 0.2}]}
].
```

3. Restart the broker.
4. Run the `rabbitmqctl status` command to be sure your new parameter has been loaded correctly.

8.2.3. Experiencing the Flow Controller

1. Open the `BrokerFlooder` class.
2. Write the Java code to send messages in an infinite loop.
3. Then modify the Java code to print on the console the time elapsed between 10,000 messages sent (it prints the time every message presently).
4. Open a terminal, and go to your `$RABBITMQ_HOME/log`.
5. Open your main log file (`rabbit@<node_name>.log`), and check it regularly.
6. Run the Java program, and observe the time that is required to send 10.000 messages to the broker. Look at the same time in the log file. The broker issues a warning in the log when it reaches the memory high watermark. The publisher is then blocked.

Note getting the warning can be difficult, as queues can swap out their content to disc when under pressure. If you don't get the warning, stop the broker and try to lower the memory watermark (e.g. 0.1 or even 0.05).

Congrats! You're done with this lab, and you have learned how to prevent your RabbitMQ from flooders.

Chapter 9. Securing RabbitMQ

9.1. Objective

In this lab, you'll learn how to secure a default RabbitMQ instance. You'll also learn how to isolate exchanges and queues for a given application in a virtual host. You'll configure the permissions on these exchanges and queues. You'll also learn how to secure the communication between the broker and clients by using TLS.

9.2. Changing the Default Access

RabbitMQ creates a default guest user with a password set to "guest" when it initializes its internal Mnesia database. This is handy for development and testing, but can be a security hole for a production broker. Let's follow a best practice and change this default access strategy.

1. Shutdown the broker and delete the Mnesia data files (the data files are in `/var/lib/rabbitmq/mnesia/NODENAME` on UNIX-like systems, and in `%APPDATA%\RabbitMQ\db\NODENAME` on Windows).
2. Set the default generated username to "admin" and their password to "changeit" (this must be done in the configuration file. Remember, the configuration file is `/etc/rabbitmq/rabbitmq.config` on UNIX-like systems, and `%APPDATA%\RabbitMQ\rabbitmq.config` on Windows systems).
3. Restart the broker.
4. Connect to the management plugin with the new default credentials.

Note: Remember that RabbitMQ creates this default user the first time it starts or when it detects its internal database has been deleted.

9.3. Creating a Virtual Host and Access Control for a Client

It's common to have several messaging applications working with a single broker. How to avoid collisions in exchanges and queues names in such situations? Moreover, how to avoid applications to access other applications' exchanges and queues? Virtual hosts are a great way to avoid all these problems.

You need to isolate the quotations applications in a dedicated virtual host. Here are the requirements:

- A `quotations` virtual host is dedicated to the application.
- Clients will use a `quotations_app` user to connect to the virtual host (use `test` as a password).
- The `quotations_app` user can only access the `quotations` virtual host. It cannot configure any resource, but can write and read all resources in the virtual host.

Now you have the requirements, you can configure the access control strategy by implementing the following guidelines:

- You can use the management plugin or the `rabbitmqctl` commands to configure the access control. Remember the command line can be scripted (handy for automated deployment), so this is probably what you'll end up using in real deployments.
- You can use a `quotations` fanout exchange bound to a `market` queue to test the behavior with the `SecureClient` Java program.
- You are encouraged to experiment with the `SecureClient`: connection to the default virtual host without any permission on it, declaration of an exchange, etc. You can observe what kind of exceptions the Java binding throws, which can be very useful for troubleshooting real-world applications.

9.4. Securing RabbitMQ with TLS

The communication between the broker and its clients can be secured by using TLS. The communication can be encrypted and both the server and the client can authenticate with certificates. In this lab, we'll be encrypting the communication and using server authentication (not client authentication). All the necessary files (certificates, private key, etc) have already been generated with OpenSSL for you.

1. Shutdown the broker and delete the Mnesia data files (the data files are in `/var/lib/rabbitmq/mnesia/NODENAME` on UNIX-like systems, and in `%APPDATA%\RabbitMQ\db\NODENAME` on Windows).
2. Take a look at the `tls-configuration` directory at the root of the project. It contains all the necessary files to set up TLS in RabbitMQ. Remember from the slides: the `cacertfile` contains the certificate of the certificate authority (the root certificate) and client certificates the broker will trust. In our case, the file is `tls-configuration/testca/cacert.pem` and contains only the root certificate. Then, the `certfile` contains the broker certificate, which is signed by the root certificate. This file is `tls-configuration/server/cert.pem`. At last, the `keyfile` contains the broker private key. This file is `tls-configuration/server/key.pem`. Following the instructions in the slides, modify the broker configuration file to enable TLS. Remember, the configuration file is `/etc/rabbitmq/rabbitmq.config` on UNIX-like systems, and `%APPDATA%\RabbitMQ\rabbitmq.config` on Windows systems).
3. Restart the broker.
4. Connect to the management plugin and look at the "ports and contexts" section of the home page. You should see the `AMPQ/SSL` protocol is enabled on the port you chose (e.g. 5671). If it's not, ask to the instructor for help.
5. It's now time to connect to the broker through TLS. Open the `SslClient` class and look at the code: it sends and consumes a message on an exclusive queue. Modify the client to connect on the appropriate TLS port and to use a secured connection with the `factory.useSslProtocol()` method. When you're ready, launch the application. You should see in the console that the client has received the message it has sent. Congratulations, you have configured TLS correctly!

The communication between the client and the server is now encrypted, they can exchange sensitive data without fearing someone eavesdropping on their conversation. But we can be even more secure quite easily: the client can check the identity of the broker. We just need to add the server certificate to the trust store of the client. By using the no-argument version of `factory.useSslProtocol()` method, the client doesn't use any trust store and trusts everyone. Let's fix that.

As the client is in Java, we need to put the server certificate into a Java key store, but this is has been done for you, the keystore is the `tls-configuration/rabbitstore` file and is protected by a password: `rabbitstore`.

1. Following the instructions in the slides, configure the trust store at the beginning of the `main` method of the `SslClient` class. This code is a little bit tricky, so most of it is provided (commented out) at the bottom of the class. You just have to copy/paste it, remove the comments, and fill in the gaps (the password and the location of the trust store file).
2. You end up with a `SSLContext c` variable. Pass this variable in to the `factory.useSslProtocol()` method. The client will now blindly trust the certificate(s) in the trust store.
3. Launch the `SslClient` again, it should send and receive its message, just like previously. The difference is now it checks the broker identity. If you want to check everything is working properly, you can make the client fail this way: change the `SSLContext` initialization by providing a `null` reference as the second parameter `c.init(null, null, null)`. Launch the class again, it should now fail, which the expected behavior. What happened? We ask the client to check the identity of the broker, but its trust store is empty, so it doesn't trust anyone!

Congratulations! You managed to configure TLS in RabbitMQ!

Chapter 10. Spring AMQP (optional)

10.1. Objective

In this lab, you will use Spring AMQP to send and receive messages, just as in the first exercise. Both synchronous and asynchronous message listeners using the AMQP template will be demonstrated.

You'll use a simple quotation use-case for this lab.

At the end of this lab, you'll have a good understanding of how to configure the Spring AMQP Template for sending and receiving messages.

10.2. Setting Up the Exchange and the Queue

If they don't exist, re-create the `quotations` fanout exchange, and bind it to a `quotations` queue.

10.3. Sending Messages

You'll use the Spring AMQP Template with an infinite loop to send messages. Messages will be sent after every one second.

1. Open the `SenderConfiguration` class.
2. Create a `CachingConnectionFactory` Spring bean.
3. Create a `RabbitTemplate` Spring bean and link it to the `ConnectionFactory`.
4. Open the `QuotationSender` class.
5. Write the Java code to fetch the `RabbitTemplate` bean into a local variable. (Hint: Obtain it using the `getBean()` method on the context.)
6. In the loop, send a message to the `quotations` exchange with a routing key of `nasq`, using the `RabbitTemplate convertAndSend()` method. By using the `convertAndSend()` method, Spring AMQP will automatically convert the object into a byte array and insert it as the message payload.
7. Execute the program.

10.4. Checking Messages with the Management Plugin

Let's check if the messages arrived correctly in the queue.

1. In the web UI of the management plugin, check the messages in the `quotations` queue.

2. Stop the `QuotationSender` and purge the queue

10.5. Asynchronous Consumption

Write a Spring AMQP message listener that displays the quotations on the console as soon as they arrive on the queue. Asynchronous consumption is the simplest way to do that.

1. Open the `ConsumerConfiguration` class.
2. Just as in the previous section, create a `CachingConnectionFactory` Spring bean.
3. Create a `SimpleMessageListenerContainer` Spring bean and configure it to call the `quotationMsgHandler` bean when messages end up in the `quotations` queue.
4. Open the `QuotationConsumer` class and take a look at the code. It creates a Spring application context from the `ConsumerConfiguration` class. The message listener will kick in as soon as the application context starts.
5. Open the `QuotationMsgHandler` class. The `onMessage()` method of this class will be invoked whenever a new message is placed onto the queue. The content of the message is printed in the console.
6. Execute the sender program, and the consumer program (in any order). You should see the quotations on the console.

Congrats! You're done with the part of the lab.

10.6. Retrieving Messages

You can explicitly ask for the messages using the `receiveAndConvert()` method of the template. If there's no message on the queue, you'll get `null`.

You will use the `RabbitTemplate` to send a message and receive it immediately afterwards. This time, you'll be writing a JUnit test.

1. Open the `SendingAndReceivingTest` class. The structure has already been provided, and you just need to implement the `sendAndReceiving()` method.
2. Use the `convertAndSend()` method on the `RabbitTemplate` to publish the quotation to the `quotations` exchange.
3. After sending the message, call `receiveAndConvert()` on the queue to retrieve the message.
4. Purge the queue if necessary.
5. Run the test.

Congratulations, you're done with this lab.

10.7. Bonus

Start up the `QuotationConsumer`, then stop the RabbitMQ server. Notice how the consumer automatically tries to reconnect to the server. Now restart the RabbitMQ server, and observe how the consumer then reconnects successfully. It was not necessary to write any explicit code to reconnect -- the Spring AMQP Template handles it all for you out-of-the-box.