



# RabbitMQ

# Agenda



- Crash course in JMS
- AMQP + RabbitMQ + JMS
- RabbitMQ server configuration
- Java client API
- Patterns + code examples



# Crash course in JMS

# What is JMS?

# What is JMS?

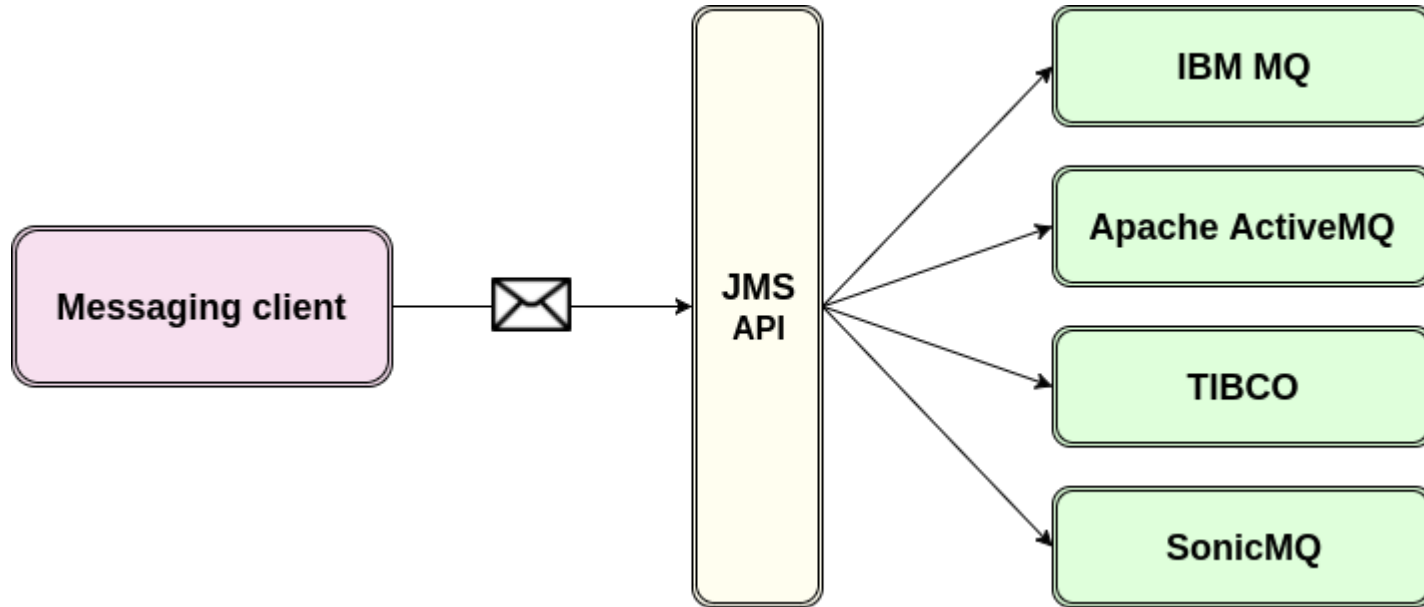
## **JMS is:**

- A Java API that allows applications to create, send, receive, and read messages
- Used for enterprise messaging
- Part of Java EE
- Loosely coupled

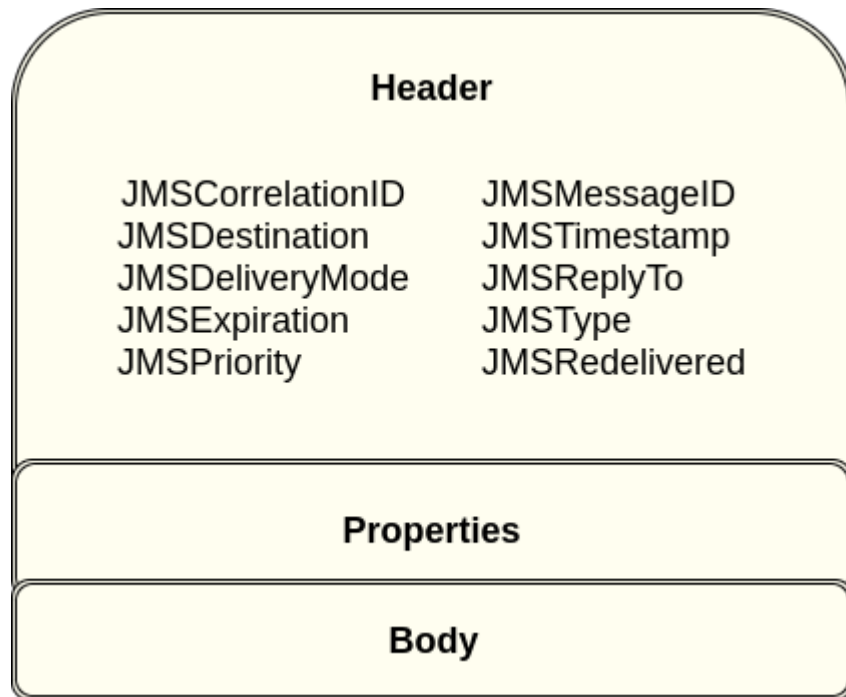
## **JMS is not:**

- A message broker implementation

# JMS abstracts message brokers



# The JMS message



# Message types

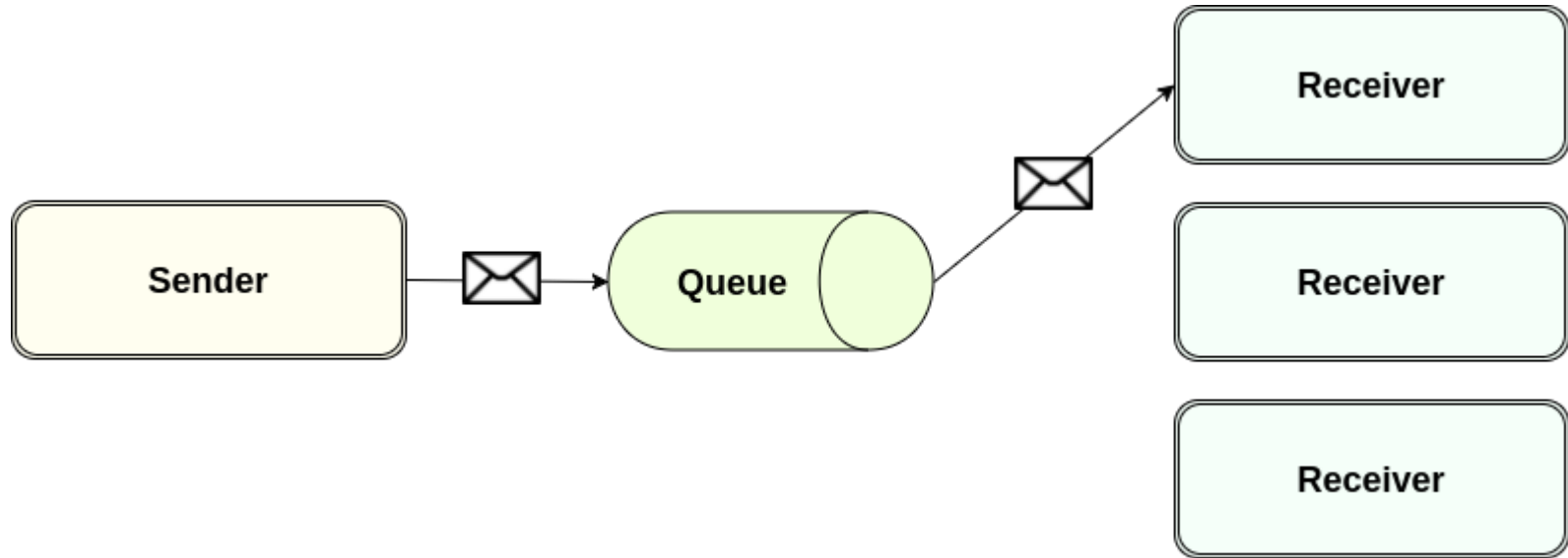
- StreamMessage
- MapMessage
- TextMessage
- ObjectMessage
- ByteMessage

All these types are subclasses of `javax.jms.Message`

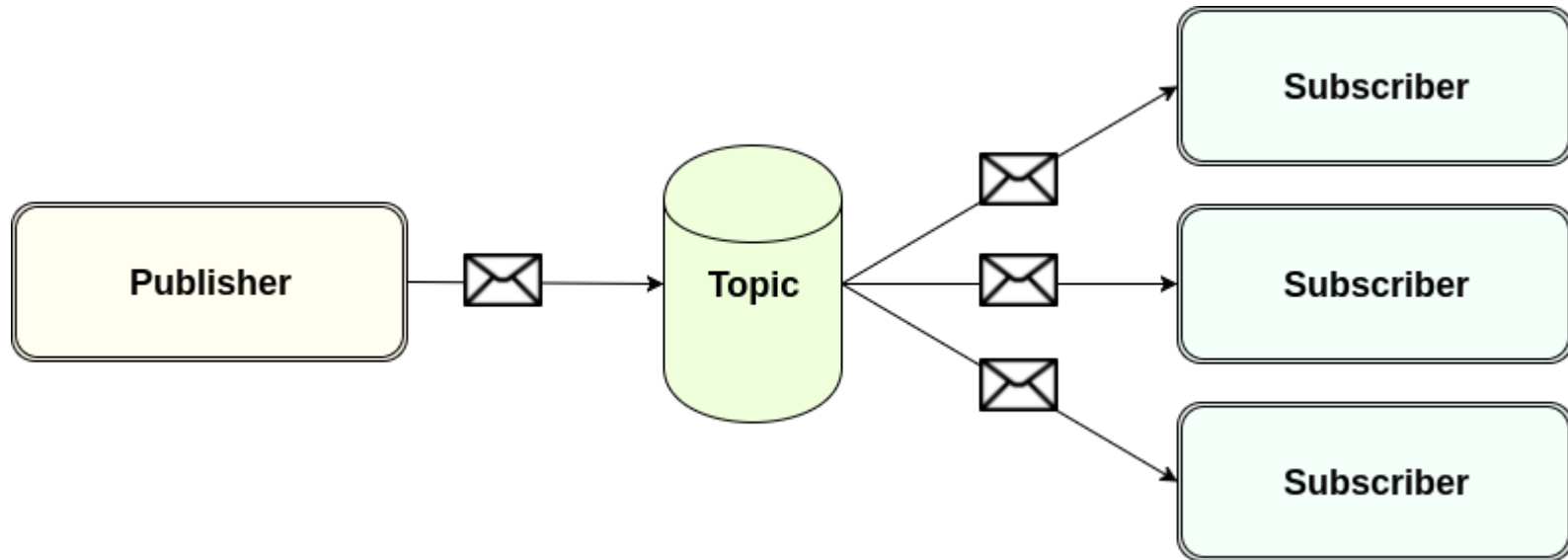


# Messaging domains

# Point-to-Point messaging

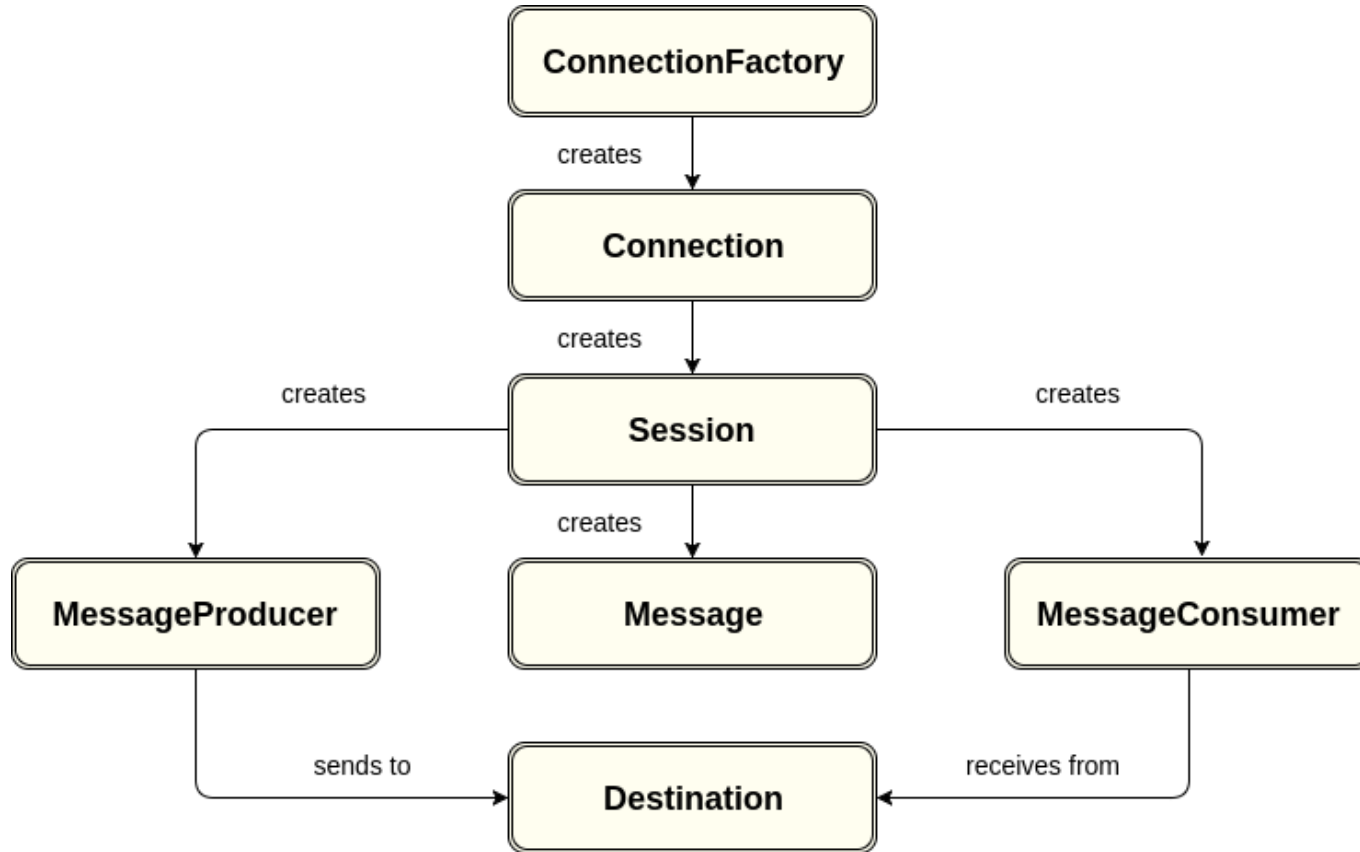


# Publish-Subscribe messaging

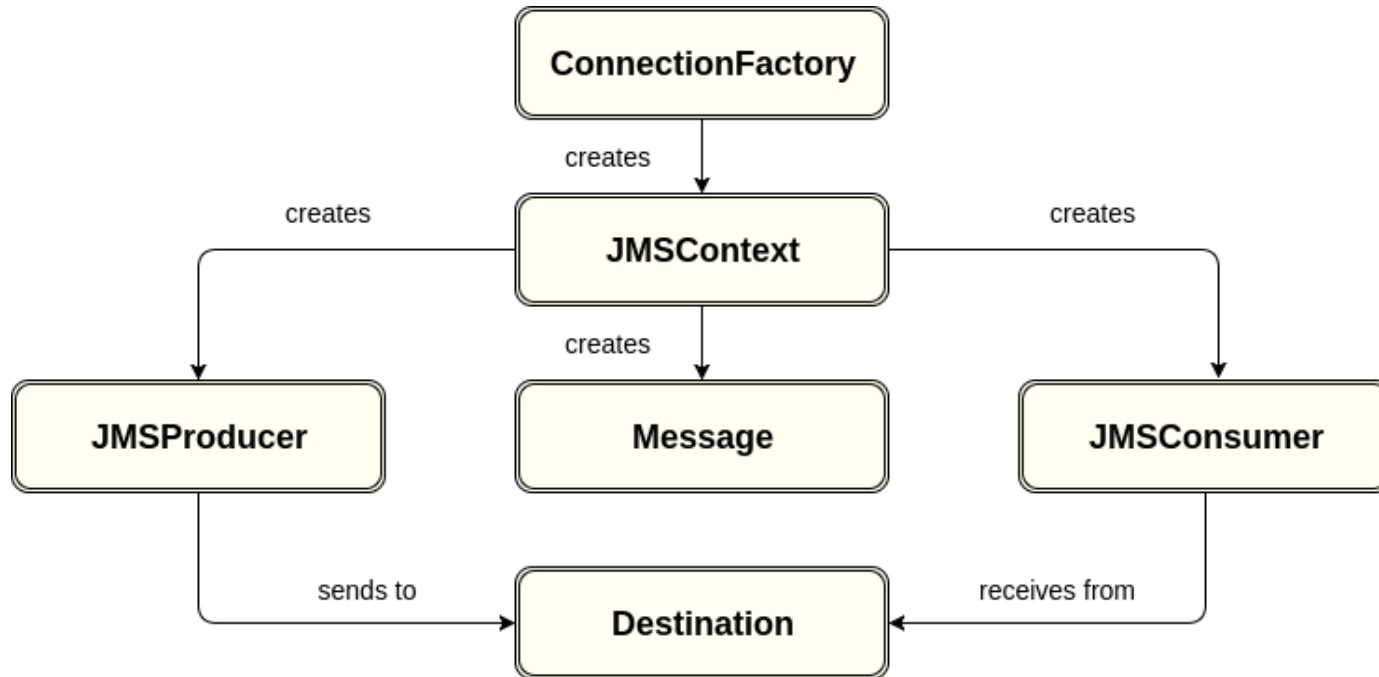


# JMS 1.1 vs. JMS 2.0

# Classic API - JMS 1.1 - 2002



# Simplified API - JMS 2.0 - 2013



# New messaging features in JMS 2.0

- Multiple Consumers Allowed on the Same Topic Subscription
- Delivery Delay
- Sending Messages Asynchronously
- JMSXDeliveryCount
- MDB Configuration Properties



# What is RabbitMQ?



# RabbitMQ

- Open source message broker
  - More than 35,000 production deployments worldwide
- Lightweight and easy to deploy
- Provides a wide range of developer tools for most popular languages
  - Java and Spring, .NET, Ruby, Python, PHP, Objective-C and Swift, JavaScript, C / C++, etc.
- Written in Erlang

# Supported protocols

- Directly
  - AMQP 0-9-1 - the "core" protocol
- Via plugins
  - STOMP
  - MQTT
  - AMQP 1.0
  - HTTP

# AMQP

# What is AMQP

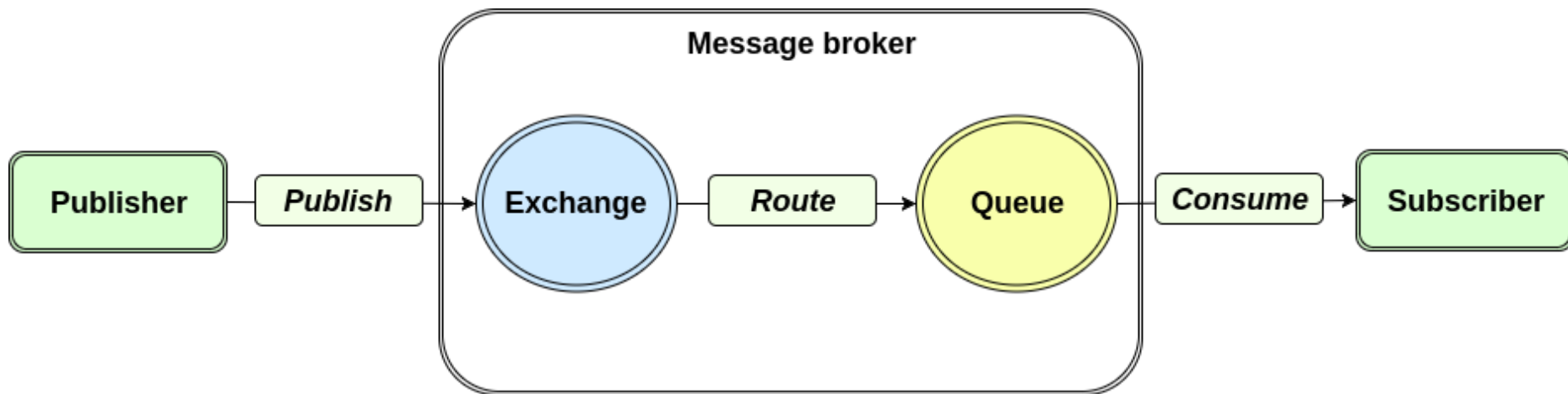
- Not JMS!
- Advanced Message Queuing Protocol
- Standard (1.0)
- Wire-level binary protocol



# Scope of AMQP

- AMQP model
  - Set of components that route and store messages
  - Set of rules for wiring these components
- A network wire-level protocol that lets:
  - Client applications talk to the server
  - Interact with the AMQ model it implements

# AMQP model



# AMQP 0-9-1

- "Core" protocol supported by RabbitMQ
- Binary protocol
- Large number of implementations for many languages and environments
- Programmable protocol

# RabbitMQ JMS client



# JMS and AMQP 0-9-1

- RabbitMQ is not a JMS provider
  - But it includes a plugin needed to support JMS
- **RabbitMQ JMS client** implements the JMS 1.1 specification on top of the **RabbitMQ Java client**
- JMS Client for RabbitMQ is compliant with both JMS and AMQP

# Unsupported JMS 1.1 features

- Doesn't support server sessions.
- XA transaction support interfaces are not implemented.
- Topic selectors are supported with the RabbitMQ JMS topic selector plugin.
- Queue selectors are not yet implemented.
- SSL and socket options for RabbitMQ connections are supported, but only using SSL connection protocols that the RabbitMQ client provides.
- The JMS NoLocal subscription feature is not supported.

# Needed components

- JMS client library and its dependent libraries.
- Enabling RabbitMQ JMS topic selector plugin:

```
$ rabbitmq-plugins enable rabbitmq_jms_topic_exchange
```

# Enabling JMS client

- To enable the JMS Client in a Java container:
  - Install the JMS client JAR files and its dependencies in the container,
  - Define JMS resources in the container's naming system.
- For standalone applications:
  - Add the JMS client JAR files and its dependencies to the application classpath.
  - The JMS resources can be defined programmatically or through a dependency injection framework like Spring.

# Defining JMS ConnectionFactory

Define the JMS ConnectionFactory in JNDI:

```
<Resource    name="jms/ConnectionFactory"  
    type="javax.jms.ConnectionFactory"  
    factory="com.rabbitmq.jms.admin.RMQObjectFactory"  
    username="guest"  
    password="guest"  
    virtualHost="/"  
    host="localhost"/>
```

# Defining JMS ConnectionFactory

Equivalent Spring bean example:

*@Bean*

```
public ConnectionFactory jmsConnectionFactory() {  
    RMQConnectionFactory connectionFactory = new RMQConnectionFactory();  
    connectionFactory.setUsername("guest");  
    connectionFactory.setPassword("guest");  
    connectionFactory.setVirtualHost("/");  
    connectionFactory.setHost("localhost");  
    return connectionFactory;  
}
```

# Available attributes/properties

Attribute/property	Description
name	<b>JNDI only.</b> Name in JNDI.
type	<b>JNDI only.</b> Name of the JMS interface the object implements, usually <code>javax.jms.ConnectionFactory</code> .
factory	<b>JNDI only.</b> JMS Client for RabbitMQ ObjectFactory class, always <code>com.rabbitmq.jms.admin.RMQObjectFactory</code> .
username	Name to use to authenticate a connection with the RabbitMQ broker. The default is "guest".
password	Password to use to authenticate a connection with the RabbitMQ broker. The default is "guest".

# Available attributes/properties

Attribute/property	Description
virtualHost	RabbitMQ virtual host. The default is "/".
host	Host on which RabbitMQ is running. The default is "localhost".
port	RabbitMQ port used for connections. The default is "5672" unless this is an SSL connection, in which case the default is "5671".
ssl	Whether to use an SSL connection to RabbitMQ. The default is "false".
uri	The AMQP 0-9-1 URI string used to establish a RabbitMQ connection. The value can encode the host, port, username, password and virtualHost.



# Destination interoperability

- Define JMS 'amqp' destinations that read and/or write to non-JMS RabbitMQ resources.
- JMS application can send Messages to a predefined RabbitMQ 'destination' using the JMS API.
  - Any AMQP 0-9-1 client can read them without having to understand the JMS format.
  - Only BytesMessages and TextMessages can be written in this way.
- JMS application can read messages from a predefined RabbitMQ queue using the JMS API.
  - RabbitMQ JMS Client packs them up into JMS Messages automatically.
  - Messages read in this way are, by default, BytesMessages.
  - Messages can be marked TextMessage (by adding an AMQP message property "JMSType": "TextMessage").
- A single 'amqp' destination can be defined for both reading and writing.

# JMS 'amqp' RMQDestination

- RMQDestination class implements Destination in the JMS interface.
- This is extended with a new constructor:

```
public RMQDestination(String destinationName, String amqpExchangeName,  
    String amqpRoutingKey, String amqpQueueName);
```

# Defining destination

Define in JNDI:

```
<Resource name="jms/Queue"  
  type="javax.jms.Queue"  
  factory="com.rabbitmq.jms.admin.RMQObjectFactory"  
  destinationName="myQueue"  
  amqp="true"  
  amqpQueueName="rabbitQueueName" />
```

# Defining destination

Equivalent Spring bean example:

`@Bean`

```
public Destination jmsDestination() {  
    RMQDestination jmsDestination = new RMQDestination();  
    jmsDestination.setDestinationName("myQueue");  
    jmsDestination.setAmqp(true);  
    jmsDestination.setAmqpQueueName("rabbitQueueName");  
    return jmsDestination;  
}
```

# Available attributes/properties

Attribute/property	Description
name	<b>JNDI only.</b> Name in JNDI.
type	<b>JNDI only.</b> Name of the JMS interface the object implements, usually <code>javax.jms.Queue</code> .
factory	<b>JNDI only.</b> JMS Client for RabbitMQ ObjectFactory class, always <code>com.rabbitmq.jms.admin.RMQObjectFactory</code> .
amqp	"true" means this is an 'amqp' destination. Default "false".

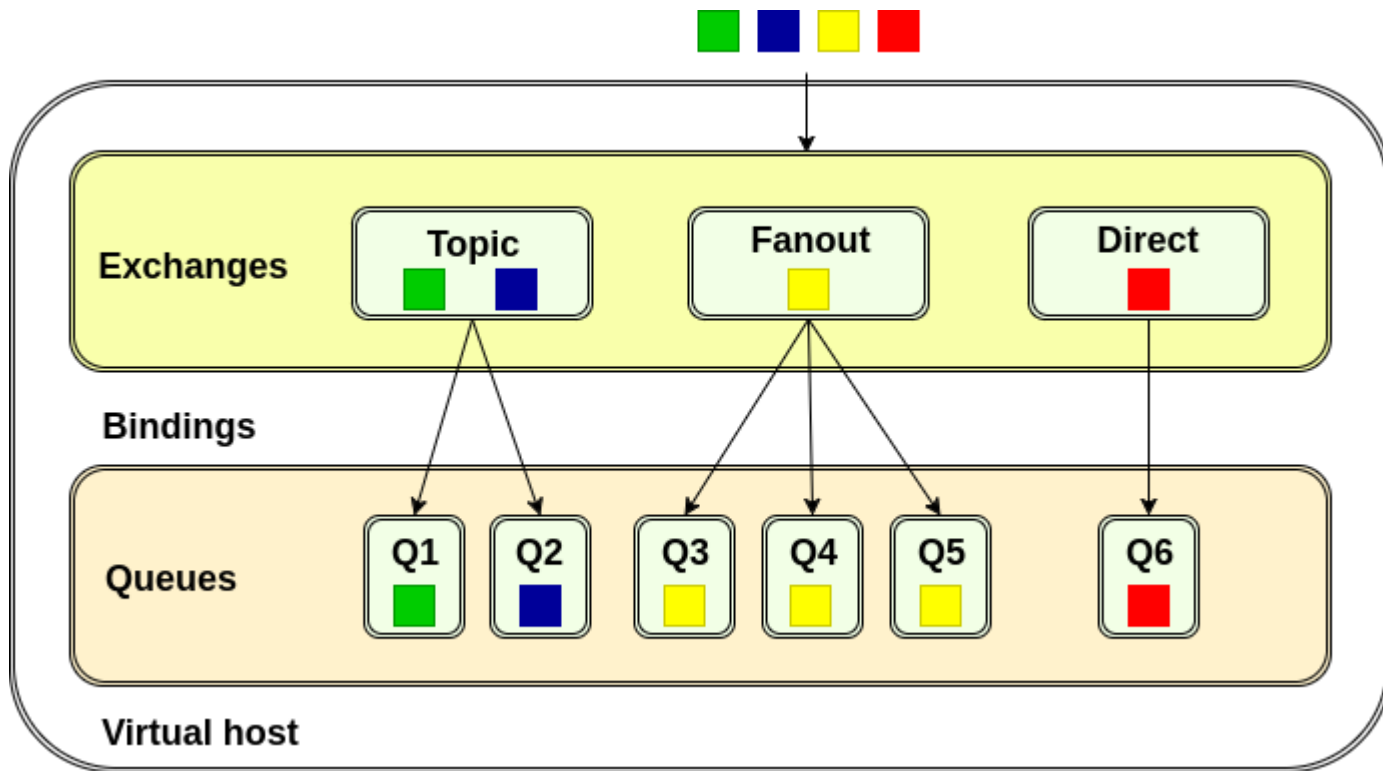
# Available attributes/properties

Attribute/property	Description
amqpExchangeName	Name of the RabbitMQ exchange to publish messages when an 'amqp' destination.
amqpRoutingKey	The routing key to use when publishing messages when an 'amqp' destination.
amqpQueueName	Name of the RabbitMQ queue to receive messages from when an 'amqp' destination.
destinationName	Name of the JMS destination.



# Message flow in RabbitMQ

# Message flow in RabbitMQ





# Exchanges and exchange types

# Exchange

- Messages are not published directly to a queue.
- Producer sends messages to an exchange.
- Exchange routes messages to 0 or more queues.
- The routing algorithm depends on the exchange type and rules called bindings.

# Exchange types

Name	Default pre-declared names
<b>Direct exchange</b>	(Empty string) and amq.direct
<b>Fanout exchange</b>	amq.fanout
<b>Topic exchange</b>	amq.topic
<b>Headers exchange</b>	amq.match (and amq.headers in RabbitMQ)

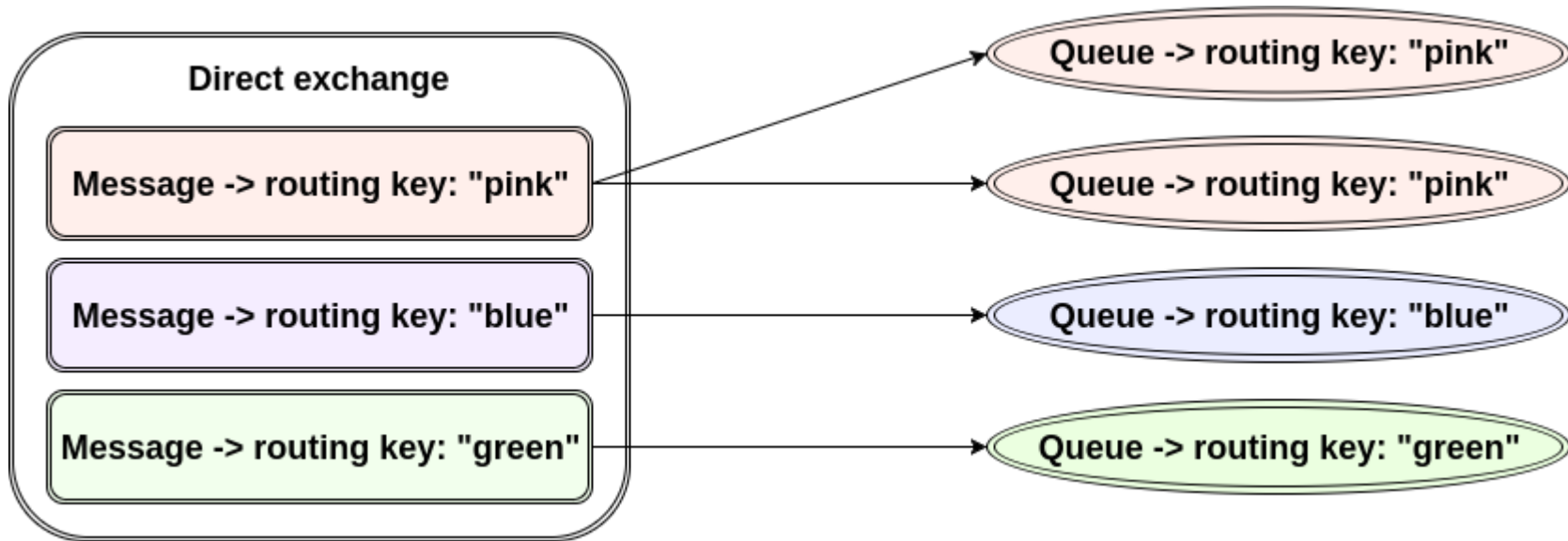
# Exchange attributes

- Name
- Durability
  - Durable or transient
- Auto-delete
- Arguments

# Direct exchange

- Delivers messages to queues **based on the message routing key**.
  - A queue binds to the exchange with a routing key K.
  - A new message with routing key R arrives at the direct exchange.
  - If  $K = R$ , exchange routes it to the queue.
- Often used to distribute tasks between multiple workers.

# Direct exchange routing



# Default exchange

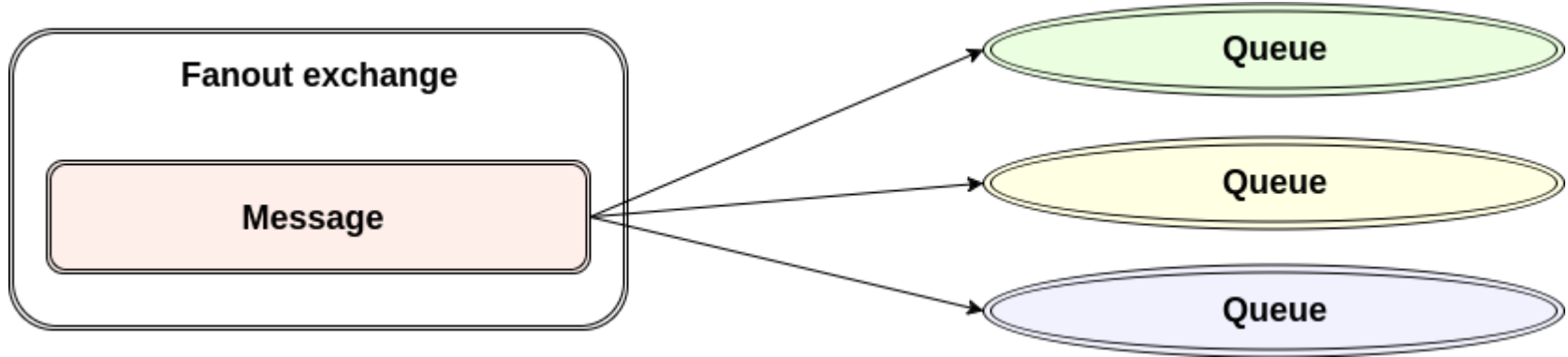
- It is a direct exchange with no name (empty string) pre-declared by the broker.
- Every queue that is created is automatically bound to it with a routing key = queue name.

# Fanout exchange

- Routes messages to **all of the queues bound to it**.
- Routing key is ignored.
- Ideal for the broadcast routing of messages.



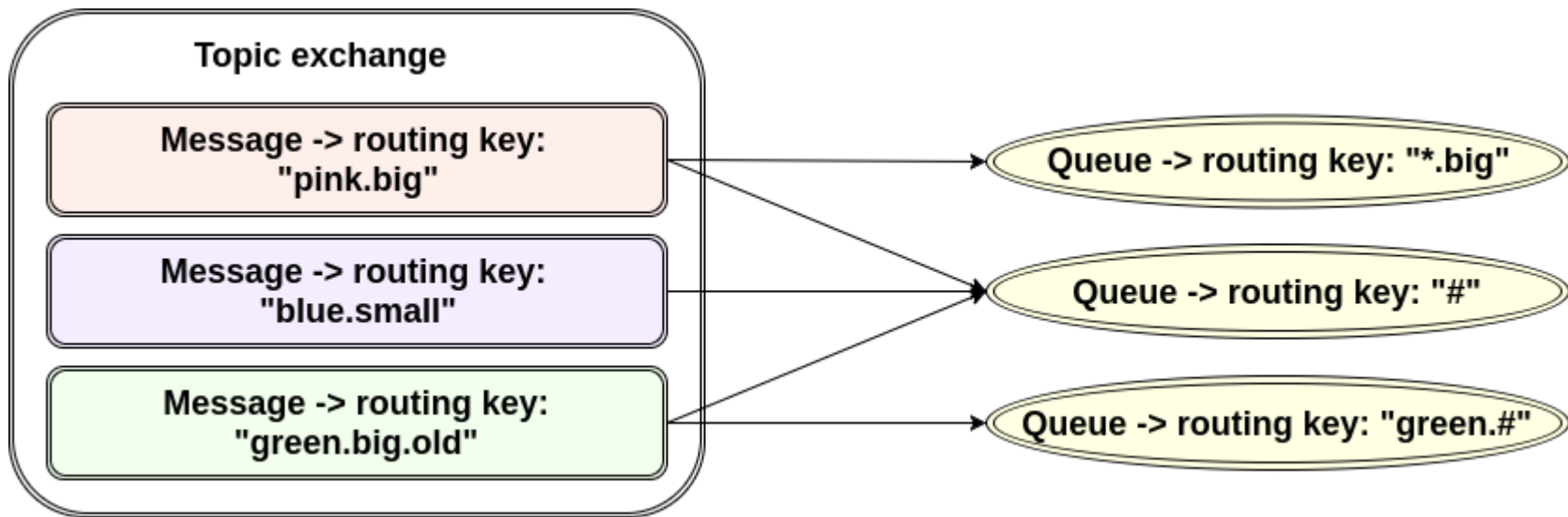
# Fanout exchange routing



# Topic exchange

- Routes messages to one or many queues **based on matching between a message routing key and the pattern** that was used to bind a queue to an exchange.
- Often used to implement various publish/subscribe pattern variations.
- Commonly used for the multicast routing of messages.

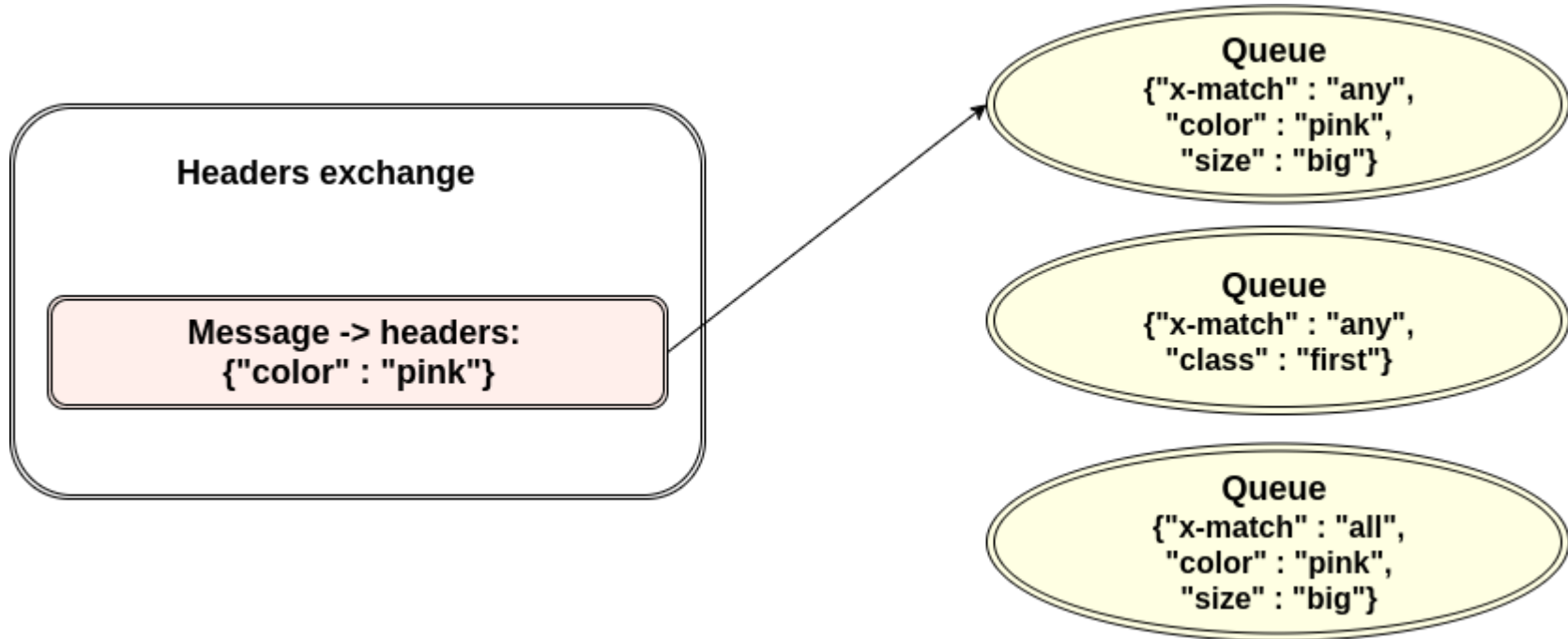
# Topic exchange routing



# Headers exchange

- Routes messages based on headers.
- Ignores routing key attribute.
- A message is considered matching if the value of the header equals the value specified upon binding.
- It is possible to use more than one header for matching.
  - This is what the "x-match" binding argument is for. It can be set to "any" or "all".
- Headers exchanges can be looked upon as "direct exchanges on steroids"
  - They can be used as direct exchanges where the routing key does not have to be a string.

# Headers exchange routing



# Queues

# Queue

- Stores messages that are consumed by applications.
- Properties:
  - Name
  - Durable
  - Exclusive
  - Auto-delete
  - Arguments
- Before a queue can be used it has to be declared.

# Queue names

- Applications may pick queue names or ask the broker to generate a name for them.
- Queue names may be up to 255 bytes of UTF-8 characters.
- Queue names starting with "amq." are reserved for internal use by the broker.



# Queue durability

- Durable queues are persisted to disk and thus survive broker restarts.
- Queues that are not durable are called transient.
- Durability of a queue does not make messages that are routed to that queue durable.

# Bindings

# Bindings and routing keys

- Bindings are rules that exchanges use to route messages to queues.
- To instruct an exchange to route messages to a queue, it has to be bound to it.
- Bindings may have an optional routing key attribute.
- The purpose of the routing key is to select certain messages published to an exchange to be routed to the bound queue.
- If AMQP message cannot be routed to any queue, it is either dropped or returned to the publisher, depending on message attributes.

# Bindings analogy

- Queue is like your destination in New York city
- Exchange is like JFK airport
- Bindings are routes from JFK to your destination.
  - There can be zero or many ways to reach it

# Virtual host

# Virtual host

- Virtual host is a namespace for queues, exchanges, bindings, users, policies.
- Virtual hosts are created using **rabbitmqctl** or **HTTP API** instead.

# Virtual hosts and client connections

- A virtual host has a name.
- When an AMQP 0-9-1 client connects to RabbitMQ, it specifies a vhost name to connect to.
- Connections to a vhost can only operate on exchanges, queues, bindings, and so on in that vhost.

# Default virtual host and user

- Virtual host named “/”.
- User named “guest” with a default password of “guest”.
- User granted full access to the “/” virtual host.





# Message attributes and payload

# Message attributes

- Content type
- Content encoding
- Routing key
- Delivery mode (persistent or not)
- Message priority
- Message publishing timestamp
- Expiration period
- Publisher application id

Some attributes are optional and known as **headers**.

# Message payload

- AMQP messages have a payload (the data that they carry).
- AMQP brokers treat it as a byte array.
- The broker will not inspect or modify the payload.
- It is possible for messages to contain only attributes and no payload.

# Message acknowledgement

- When should the AMQP broker remove messages from queues?
  - After broker sends a message to an application.
  - After the application sends back an acknowledgement.
- AMQP 0-9-1 has a built-in feature called **message acknowledgements** (*acks*).
- Consumers use it to confirm message delivery and/or processing.
- If an application crashes, if an acknowledgement for a message was expected but not received by the AMQP broker, the message is re-queued.



# Server configuration

# RabbitMQ configuration

# Ways to customise server

- Environment variables
- Configuration file
- Runtime parameters and policies

# Customise RabbitMQ environment

- Unix (general)
  - Create/edit **rabbitmq-env.conf**.
- Windows
  - Create or edit environment variables in Windows dialogue.
  - Create/edit **rabbitmq-env-conf.bat**.



# RabbitMQ environment variables

- RabbitMQ environment variable names have the prefix RABBITMQ\_.
- A typical variable called **RABBITMQ\_var\_name** is set as follows:
  - a shell environment variable called **RABBITMQ\_var\_name** is used if this is defined;
  - *otherwise*, a variable called **var\_name** is used if this is set in the rabbitmq-env.conf file;
  - *otherwise*, a system-specified default value is used.

# RabbitMQ environment variables include\*

Name	Default
RABBITMQ_NODE_IP_ADDRESS	the empty string
RABBITMQ_NODE_PORT	5672
RABBITMQ_NODENAME	Unix (general): rabbit@\$HOSTNAME Windows: rabbit@%COMPUTERNAME%
RABBITMQ_CONF_ENV_FILE	UNIX (general): \$RABBITMQ_HOME/etc/rabbitmq/rabbitmq-env.conf Windows: %APPDATA%\RabbitMQ\rabbitmq-env-conf.bat

\*include, but are not limited to

# Define environment variables in file

- Use the standard environment variable names (but drop the RABBITMQ\_ prefix):

*#example rabbitmq-env.conf file entries*

*#Rename the node*

NODENAME=bunny@myhost

*#Config file location and new filename bunnies.config*

CONFIG\_FILE=/etc/rabbitmq/testdir/bunnies

# Configuration file

- The configuration file **rabbitmq.config** allows to configure:
  - RabbitMQ core application,
  - Erlang services,
  - RabbitMQ plugins.
- It is a standard Erlang configuration file.
- To override RabbitMQ config file location, use the RABBITMQ\_CONFIG\_FILE environment variable.

# Configuration file example

```
[  
  {rabbit, [  
    {tcp_listeners, [{"127.0.0.1", 5672}]},  
    {ssl_listeners, [{"127.0.1.1", 5671}]},  
    {ssl_options, [  
      {cacertfile, "/usr/local/etc/rabbitmq/ssl/testca/cacert.pem"},  
      {certfile, "/usr/local/etc/rabbitmq/ssl/server/cert.pem"},  
      {keyfile, "/usr/local/etc/rabbitmq/ssl/server/key.pem"},  
      {verify, verify_none},  
      {fail_if_no_peer_cert, false}  
    ]}  
  ]}  
].
```

# Configurable variables include\*

Key	Documentation
tcp_listeners	Ports or hostname/pair on which to listen for AMQP connections. Default: [5672]
num_tcp_acceptors	Number of Erlang processes that will accept connections for the TCP listeners. Default: 10
handshake_timeout	Maximum time for AMQP handshake. Default: 10000 ms
disk_free_limit	Disk free space limit of the partition on which RabbitMQ is storing data. Default: 50000000 bytes
log_levels	A list of log event category and log level pairs. Default: [{connection, info}]

\*include, but are not limited to

# Configuration entry encryption

- Sensitive configuration entries can be encrypted in the RabbitMQ configuration file.
- The broker decrypts encrypted entries on start.
- No sensitive data appears in plain text in configuration files.
- Encrypted values must be inside an Erlang encrypted tuple: {encrypted, ...}.

# Configuration entry encryption example

```
[
  {rabbit, [
    {default_user, <<"guest">>},
    {default_pass, {encrypted, <<"cPAymwqmMnbPXXRVqVzpxJdrS8mHEKuo2V+3vt1u/
      fymexD9oztQ2G/oJ4PAaSb2c5N/hRJ2aqP/X0VAfx8xOQ==">>}}
  ]},
  {config_entry_decoder, [
    {passphrase, <<"mypassphrase">>}}
  ]}
].
```



# rabbitmq.config, rabbitmq-env.conf location

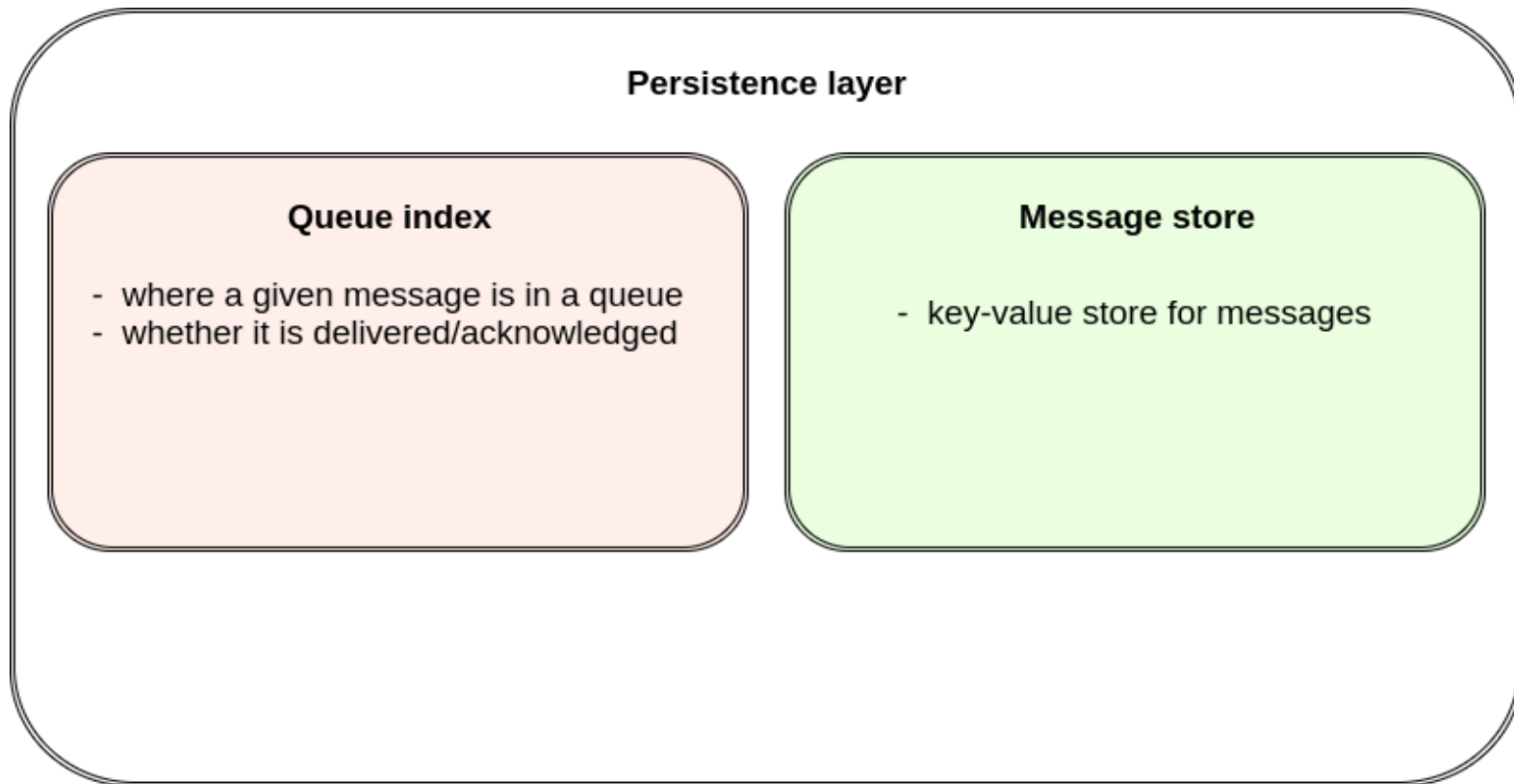
- The location of these files is distribution-specific.
- By default, they are not created, but expect to be located in the following places:
  - Generic UNIX - `$RABBITMQ_HOME/etc/rabbitmq/`
  - Windows - `%APPDATA%\RabbitMQ\`
- If `rabbitmq-env.conf` doesn't exist, it can be created manually in the location, specified by the `RABBITMQ_CONF_ENV_FILE` variable.
  - On Windows systems, it is named `rabbitmq-env.bat`.
- If `rabbitmq.config` doesn't exist, it can be created manually.
  - Set the `RABBITMQ_CONFIG_FILE` environment variable if you change the location.

# Persistence configuration

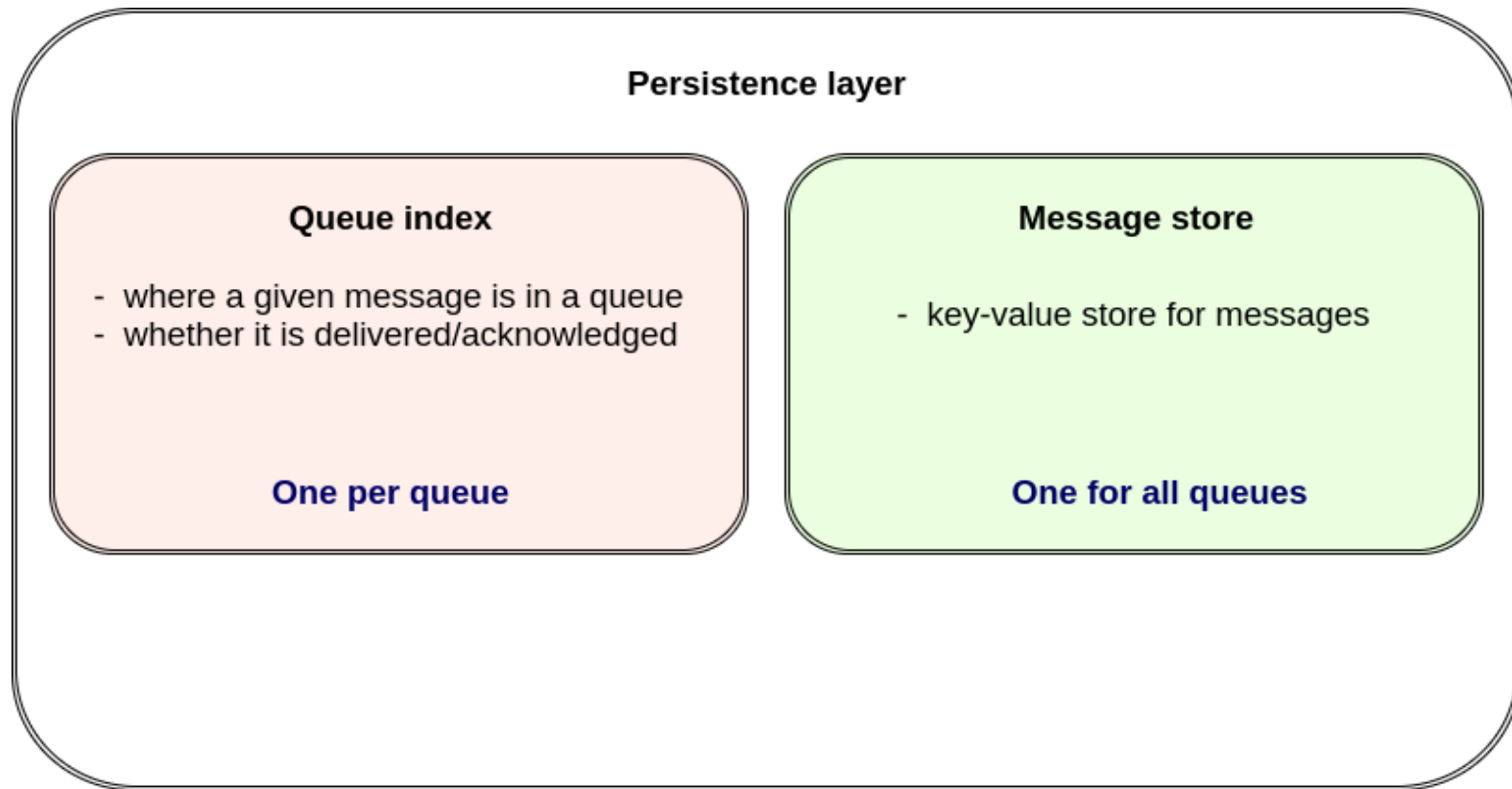
# Persistence background

- Both persistent and transient messages can be written to disk.
- Persistent messages will be written to disk as soon as they reach the queue.
- Transient messages will be written to disk only to be evicted from memory under memory pressure.
- Persistent messages are kept in memory when possible and only evicted from memory under memory pressure.
- The "persistence layer" refers to the mechanism used to store messages of both types to disk.

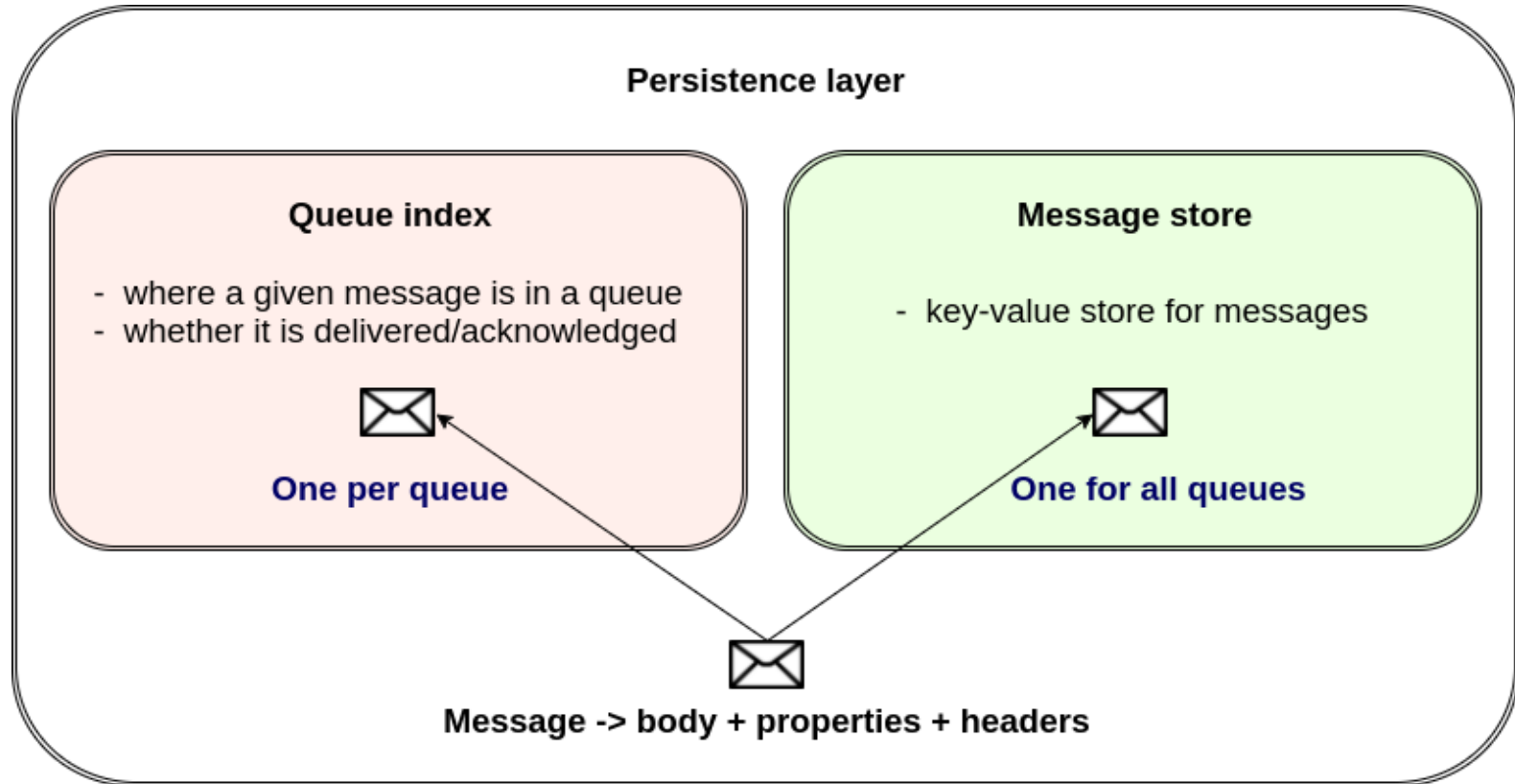
# Persistence layer



# Persistence layer



# Persistence layer



# Memory costs

- Under memory pressure:
  - Write as much out to disk.
  - Remove as much from memory.
- Must remain in memory:
  - Queue metadata for each unacknowledged message.
  - Index for every message in the message store.

# Messages in the queue index

## Advantages:

- Write to disk in one operation rather than two.
- Messages do not require an entry in the message store index.

## Disadvantages:

- The queue index keeps blocks of a fixed number of records in memory.
- If a message is routed to multiple queues, it will be written to multiple queue indices.
- Unacknowledged messages whose destination is the queue index are always kept in memory.



# Messages in the queue index

- The intent is for very small messages to be stored in the queue index as an optimisation.
- This is controlled by the configuration item `queue_index_embed_msgs_below`.
- By default, messages with a serialised size of less than 4096 bytes are stored in the queue index.

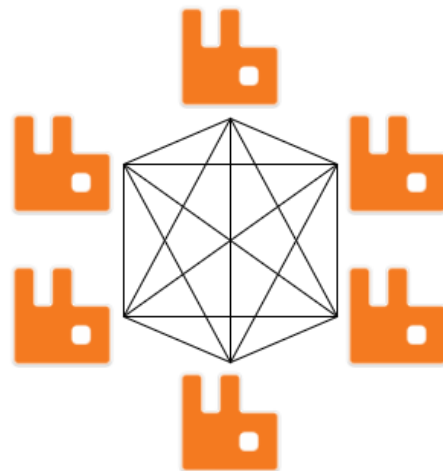
# Clustering

# Clustering background

- RabbitMQ broker is a logical grouping of one or several Erlang nodes.
- Each node is running the RabbitMQ application and sharing:
  - users,
  - virtual hosts,
  - queues,
  - exchanges,
  - bindings,
  - runtime parameters.
- We refer to the collection of nodes as a **cluster**.

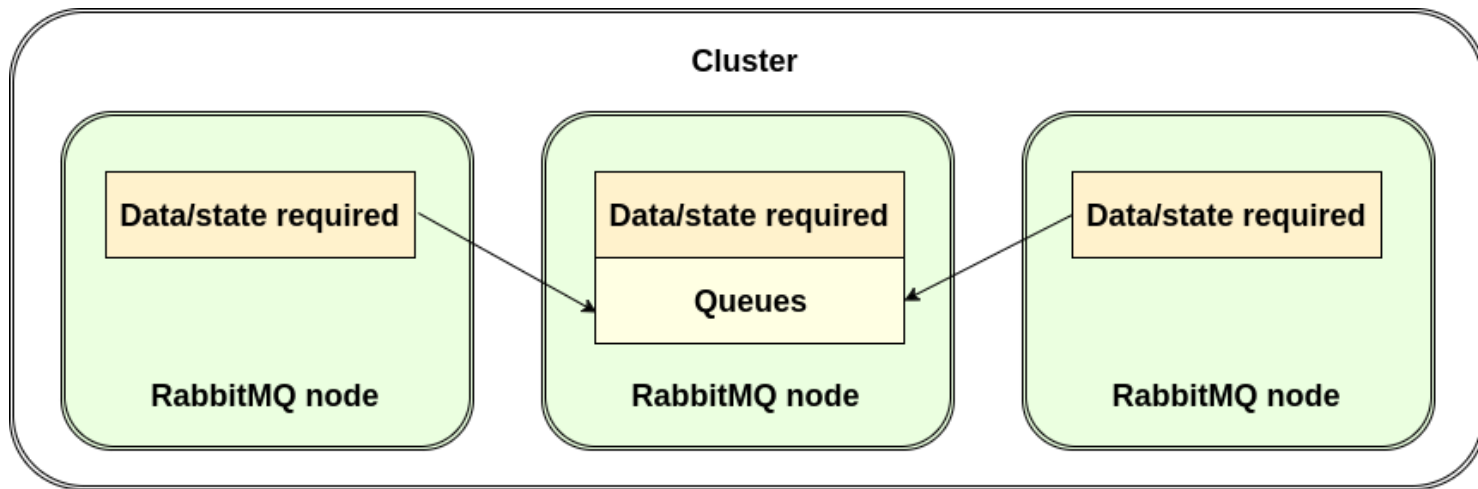
# RabbitMQ clustering

- Allowing consumers and producers to keep running if one Rabbit node dies.
- Linearly scaling messaging throughput by adding more nodes.
- Using across LAN.
- Adding high-available queues.



# Architecture of a cluster

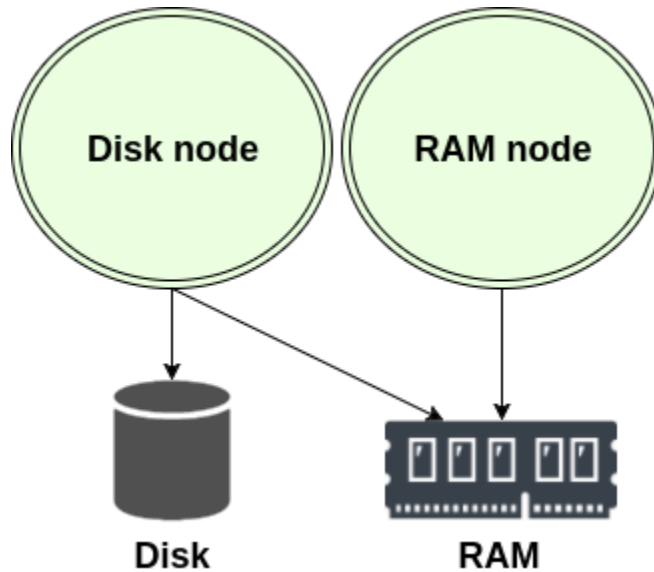
- All data/state required for the operation of a RabbitMQ broker is replicated across all nodes.
- Message queues by default reside on one node, though they are visible from all nodes.
- To replicate queues across nodes in a cluster, use mirrored queues.



# Cluster formation

- Manually with rabbitmqctl.
- Declaratively by listing cluster nodes in config file.
- Declaratively with rabbitmq-autocluster (a plugin).
- Declaratively with rabbitmq-clusterer (a plugin).

# Disk and RAM nodes



# Disk and RAM nodes

- Single-node systems are only allowed to be disk nodes.
- In a cluster, you can configure RAM nodes.
- At least one node in a cluster must be a disk one.
- While joining or leaving cluster, nodes should notify at least one disk node of the change.
- If you only have one disk node and that node happens to be down, your cluster can continue to route messages but you can't:
  - Create queues, exchanges, bindings.
  - Add users.
  - Change permissions.
  - Add or remove cluster nodes.



# Nodes communication

- Cookie determines whether nodes are allowed to communicate with each other.
- The cookie is just a string of alphanumeric characters.
- Every cluster node must have the same cookie.
- Erlang VM automatically creates random cookie file when the RabbitMQ server starts up.

# Setting up cluster

- Starting independent nodes

```
rabbit1$ rabbitmq-server -detached
```

```
rabbit2$ rabbitmq-server -detached
```

- Creating the cluster

- In order to link nodes in a cluster, we tell one node to join the cluster of another one:

```
rabbit2$ rabbitmqctl stop_app
```

```
Stopping node rabbit@rabbit2 ...done.
```

```
rabbit2$ rabbitmqctl join_cluster rabbit@rabbit1
```

```
Clustering node rabbit@rabbit2 with [rabbit@rabbit1] ...done.
```

# Manipulating cluster

- Restarting cluster nodes

```
rabbit1$ rabbitmqctl stop
```

```
Stopping and halting node rabbit@rabbit1 ...done.
```

```
rabbit2$ rabbitmqctl cluster_status
```

```
Cluster status of node rabbit@rabbit2 ...
```

```
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2,rabbit@rabbit3]}]},  
 {running_nodes,[rabbit@rabbit3,rabbit@rabbit2]}}]  
...done.
```

```
rabbit1$ rabbitmq-server -detached
```

```
rabbit1$ rabbitmqctl cluster_status
```

```
Cluster status of node rabbit@rabbit1 ...
```

```
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2,rabbit@rabbit3]}]},  
 {running_nodes,[rabbit@rabbit2,rabbit@rabbit1]}}]  
...done.
```

# Manipulating cluster

- Breaking up a cluster
  - Nodes need to be removed explicitly from a cluster.

```
rabbit2$ rabbitmqctl stop_app
Stopping node rabbit@rabbit2 ...done.
rabbit2$ rabbitmqctl reset
Resetting node rabbit@rabbit2 ...done.
rabbit2$ rabbitmqctl start_app
Starting node rabbit@rabbit2 ...done.

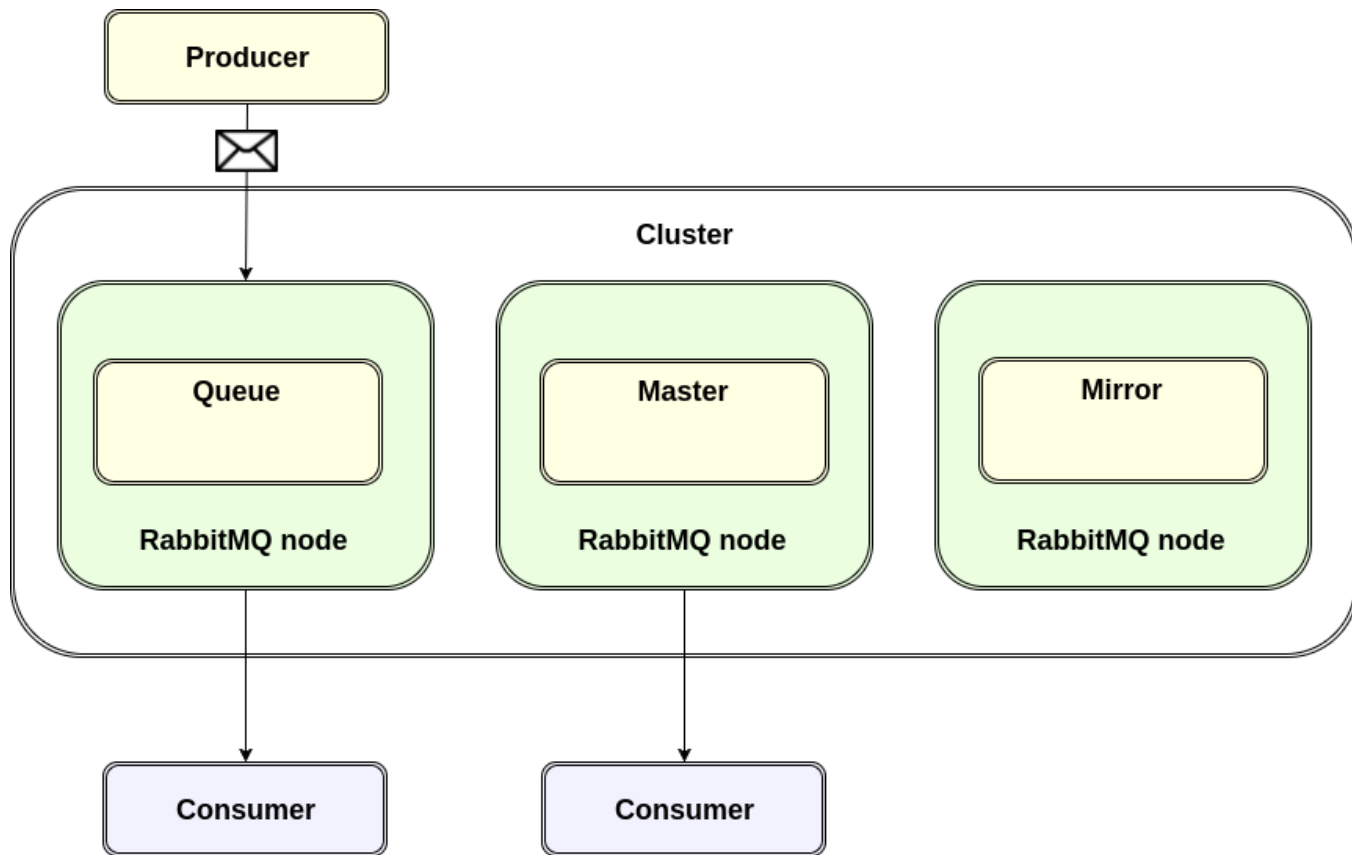
rabbit1$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes,[{disc,[rabbit@rabbit1]}]},
{running_nodes,[rabbit@rabbit1]}]
...done.
```

# Highly available (mirrored) queues

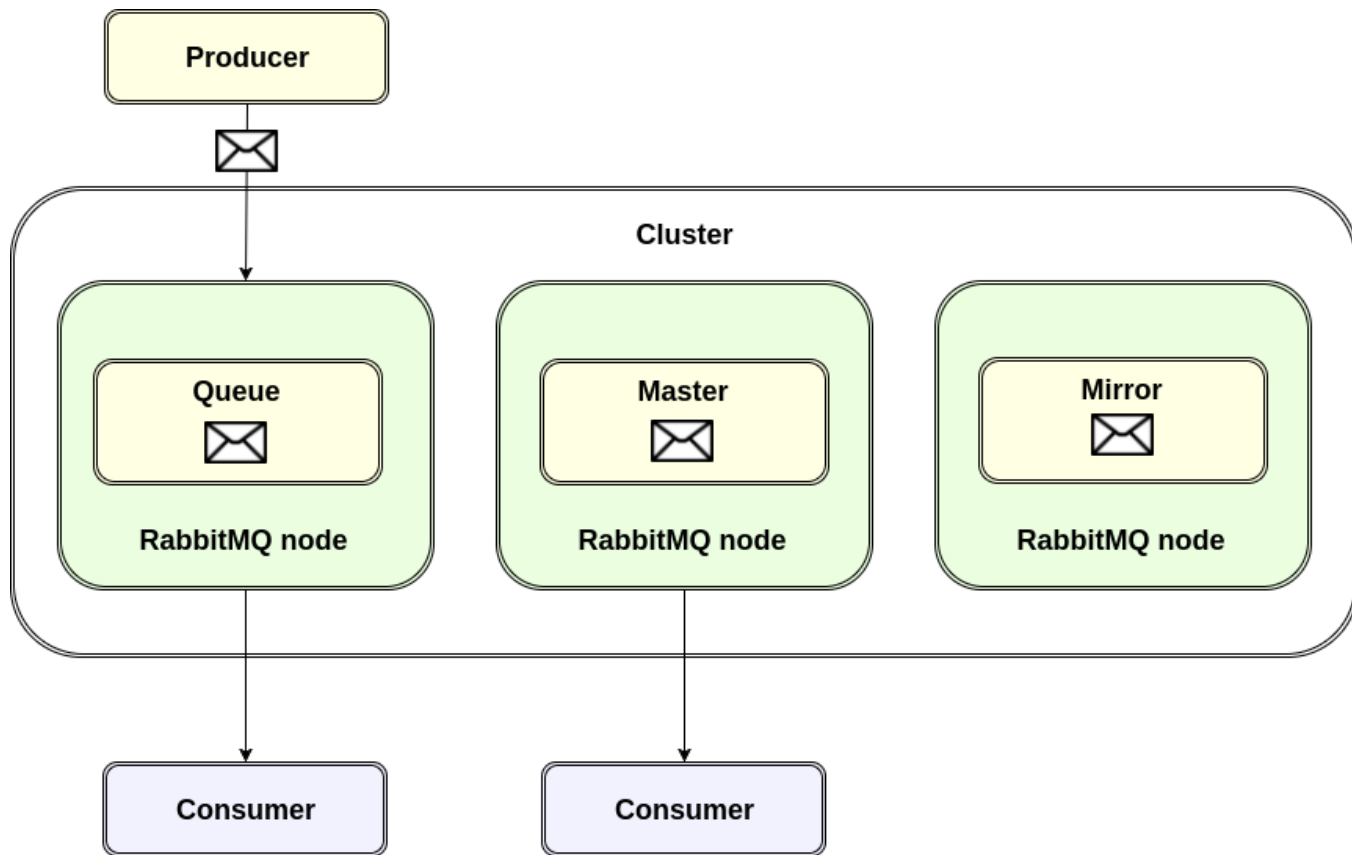
# Queue mirroring

- Requires a RabbitMQ cluster.
- By default, queues within a RabbitMQ cluster are located on a single node.
- Queues can optionally be made *mirrored* across multiple nodes.
- Each mirrored queue consists of one *master* and one or more *mirrors*.
- The oldest mirror becomes new master if the old master disappears for any reason.
- Messages published to the queue are replicated to all mirrors.
- Queue mirroring enhances availability, but does not distribute load across nodes.

# Queue mirroring

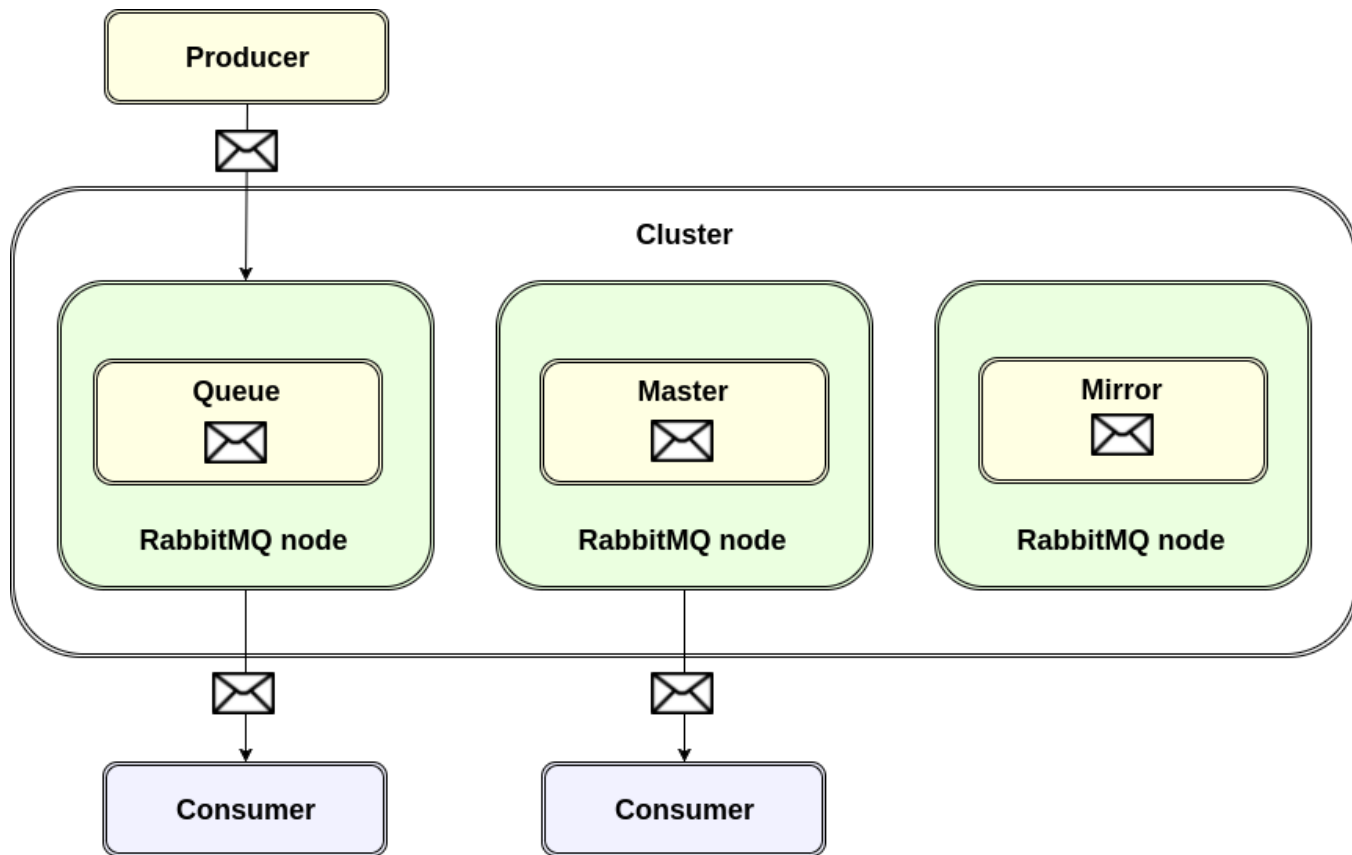


# Queue mirroring





# Queue mirroring



# Mirroring configuring

- Queues in RabbitMQ have optional parameters (arguments), sometimes referred to as **x-arguments**.
- Using x-arguments to configure mirroring parameters:
  - via client-provided arguments,
  - via policies - recommended.
- A policy matches one or more queues by name (using a regular expression pattern) and appends its definition (a map of optional arguments) to the x-arguments of the matching queues.

# Queue arguments that control mirroring

- Queues typically have mirroring enabled via policy.
- To cause queues to become mirrored, you should create a policy which:
  - matches queues,
  - sets policy keys **ha-mode** and (optionally) **ha-params**.

# Options for ha-mode and ha-params

ha-mode	ha-params	Result
all	(absent)	Queue is mirrored across all nodes in the cluster.
exactly	count	Queue is mirrored to count nodes in the cluster.
nodes	node names	Queue is mirrored to the nodes listed in node names.

- For clusters of 3 and more nodes it is recommended to mirror to a majority of nodes.
- For inherently transient or very time sensitive data, it can be perfectly reasonable to use a lower number of mirrors (or even not use any mirroring).

# Queue master

- Every queue in RabbitMQ has a home node - *queue master*.
- All queue operations go through the master first and then are replicated to mirrors.
- Strategies to distribute queue masters between nodes:
  - min-masters
  - client-local
  - random

# Queue master

- Ways to distribute queue masters:
  - using the x-queue-master-locator queue declare argument,
  - setting the queue-master-locator policy key,
  - defining the queue\_master\_locator key in the configuration file.

# Exclusive queues

- Exclusive queues will be deleted when the connection that declared them is closed.
- For this reason, exclusive queues are never mirrored (even if they match a policy).
- They are also never durable (even if declared as such).

A collection of various blue geometric shapes including triangles, squares, and circles, some of which contain icons like a gear and a lightbulb, arranged in a loose cluster on the left side of the slide.

# RabbitMQ Java Client API



# Java Client API overview

# Overview

- RabbitMQ Java client uses `com.rabbitmq.client` as its top-level package.
- The key classes and interfaces are:
  - Channel
  - Connection
  - ConnectionFactory
  - Consumer

# Connecting to broker

```
ConnectionFactory factory = new ConnectionFactory();  
factory.setUsername(userName);  
factory.setPassword(password);  
factory.setVirtualHost(virtualHost);  
factory.setHost(hostName);  
factory.setPort(portNumber);  
Connection conn = factory.newConnection();
```

or alternatively:

```
ConnectionFactory factory = new ConnectionFactory();  
factory.setUri("amqp://userName:password@hostName:portNumber/virtualHost");  
Connection conn = factory.newConnection();
```

# Opening a channel

- Open channel to send and receive messages:

```
Channel channel = conn.createChannel();
```

- To disconnect, simply close the channel and the connection:

```
channel.close();  
conn.close();
```

# Using exchanges and queues

- Declare exchange before it can be used:

- Durable, non-autodelete exchange of "direct" type:

```
channel.exchangeDeclare(exchangeName, "direct", true);
```

- Declare queue before it can be used:

- Non-durable, exclusive, autodelete queue with a generated name:

```
String queueName = channel.queueDeclare().getQueue();
```

- Durable, non-exclusive, non-autodelete queue with a well-known name:

```
channel.queueDeclare(queueName, true, false, false, null);
```

- Bind the queue to the exchange with the given routing key:

```
channel.queueBind(queueName, exchangeName, routingKey);
```

# Publishing messages

# Publishing message

- Publish a message to an exchange:

```
byte[] messageBodyBytes = "Hello, world!".getBytes();  
channel.basicPublish(exchangeName, routingKey, null, messageBodyBytes);
```

- You can specify the mandatory flag, or use pre-set message properties:

```
channel.basicPublish(exchangeName, routingKey, mandatory,  
    MessageProperties.PERSISTENT_TEXT_PLAIN,  
    messageBodyBytes);
```

# Publishing message

- Add custom headers:

```
Map<String, Object> headers = new HashMap<String, Object>();  
headers.put("latitude", 51.5252949);  
headers.put("longitude", -0.0905493);  
  
channel.basicPublish(exchangeName, routingKey,  
    new AMQP.BasicProperties.Builder()  
        .contentType("text/plain")  
        .deliveryMode(2)  
        .priority(1)  
        .headers(headers)  
        .build()),  
    messageBodyBytes);
```



# Channels and thread safety

- Channel instances must not be shared between threads.
- Applications should prefer using a Channel per thread.

# Receiving messages by subscription

# Setting up a subscription

- The messages will be delivered automatically as they arrive, rather than having to be explicitly requested.
- Individual subscriptions are always referred to by their consumer tags.
- Tags can be either client- or server-generated.
- Distinct Consumers on the same Channel must have distinct consumer tags.

# Setting up a subscription

```
boolean autoAck = false;

channel.basicConsume(queueName, autoAck, "myConsumerTag",
    new DefaultConsumer(channel) {
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope,
            AMQP.BasicProperties properties, byte[] body) throws IOException {
            String routingKey = envelope.getRoutingKey();
            String contentType = properties.getContentType();
            long deliveryTag = envelope.getDeliveryTag();
            // process the message components here ...
            channel.basicAck(deliveryTag, false);
        }
    });
```

# Callbacks to consumer

- You can explicitly cancel a particular Consumer:

```
channel.basicCancel(consumerTag);
```

- Callbacks to Consumers are dispatched on a thread separate from the thread managed by Connection.
- Consumers can safely call blocking methods on the Connection or Channel, such as `queueDeclare`, `txCommit`, `basicCancel` or `basicPublish`.
- Each Channel has its own dispatch thread.

# Retrieving individual messages

# Explicitly retrieve messages

```
boolean autoAck = false;

GetResponse response = channel.basicGet(queueName, autoAck);

if (response == null) {
    // No message retrieved.
} else {
    AMQP.BasicProperties props = response.getProperties();
    byte[] body = response.getBody();
    long deliveryTag = response.getEnvelope().getDeliveryTag();
    ...
    channel.basicAck(method.deliveryTag, false);
}
```

# Handling unroutable messages

- If a message is published with the "mandatory" flag, but cannot be routed, the broker will return it to the sending client.
- To be notified of such returns, configure a return listener, or returned messages will be dropped.

```
channel.setReturnListener(new ReturnListener() {  
    public void handleBasicReturn(int replyCode,  
        String replyText,  
        String exchange,  
        String routingKey,  
        AMQP.BasicProperties properties,  
        byte[] body) throws IOException { ... }  
});
```



# Shutdown protocol

# AMQP client shutdown

- Connection and channel lifecycle states:
  - open
  - closing
  - closed
- Connection and channel shutdown-related methods:
  - `addShutdownListener(ShutdownListener listener)` and `removeShutdownListener(ShutdownListener listener)`
  - `getCloseReason()`
  - `isOpen()`
  - `close(int closeCode, String closeMessage)`

# Shutdown circumstances

The ShutdownSignalException class provides methods to analyze the reason of the shutdown:

```
public void shutdownCompleted(ShutdownSignalException cause) {  
    if (cause.isHardError()) {  
        Connection conn = (Connection) cause.getReference();  
        if (!cause.isInitiatedByApplication()) {  
            Method reason = cause.getReason();  
            ...  
        }  
        ...  
    } else {  
        Channel ch = (Channel) cause.getReference();  
        ...  
    }  
}
```

# Use of isOpen() method

Use of the isOpen() method of channel and connection objects is not recommended for production code:

```
public void brokenMethod(Channel channel) {  
    if (channel.isOpen()) {  
        // The following code depends on the channel being in open state.  
        // However there is a possibility of the change in the channel state  
        // between isOpen() and basicQos(1) call  
        ...  
        channel.basicQos(1);  
    }  
}
```

# Advanced connection options

# Consumer thread pool

- Consumer threads are automatically allocated in a new `ExecutorService` thread pool by default.
- If greater control is required supply an `ExecutorService` on the `newConnection()` method:

```
ExecutorService executorService = Executors.newFixedThreadPool(20);  
Connection conn = factory.newConnection(executorService);
```

- When the connection is closed a default `ExecutorService` will be `shutdown()`, but a user-supplied `ExecutorService` will *not* be `shutdown()`.

# Using list of hosts

- It is possible to pass an Address array to `newConnection()`.
- An Address is simply a convenience class with *host* and *port* components.

```
Address[] addresses = new Address[]{ new Address(hostname1, portnumber1),  
    new Address(hostname2, portnumber2)};  
  
Connection conn = factory.newConnection(addresses);  
  
// will attempt to connect to hostname1:portnumber1,  
// and if that fails to hostname2:portnumber2
```

# AddressResolver

- It is possible to let an implementation of AddressResolver choose where to connect when creating a connection:

```
Connection conn = factory.newConnection(addressResolver);
```

- The AddressResolver interface is like the following:

```
public interface AddressResolver {  
    List<Address> getAddresses() throws IOException;  
}
```

- The first Address returned will be tried first, then the second if the client fails to connect to the first, and so on.



# Heartbeat timeout

- The heartbeat timeout defines after what period of time the peer TCP connection should be considered unreachable
- The timeout is in seconds, and default value is 60.
- Heartbeat frames are sent about every timeout / 2 seconds.
- After two missed heartbeats, the peer is considered to be unreachable.
- Configuring the heartbeat timeout:

```
ConnectionFactory connectionFactory = new ConnectionFactory();  
// set the heartbeat timeout to 90 seconds  
connectionFactory.setRequestedHeartbeat(90);
```

# NIO support

- With the default blocking IO mode, each connection uses a thread to read from the network socket.
- With the NIO mode, you can control the number of threads that read and write from/to the network socket.
- NIO must be enabled explicitly:

```
ConnectionFactory connectionFactory = new ConnectionFactory();  
connectionFactory.useNio();  
connectionFactory.setNioParams(new NioParams().setNbIoThreads(4));
```

# Automatic recovery from network failures

# Connection recovery

- Network connection between clients and RabbitMQ nodes can fail.
- RabbitMQ Java client supports automatic recovery of connections.
- The automatic recovery process for many applications follows the following steps:
  - Reconnect
  - Restore connection listeners
  - Re-open channels
  - Restore channel listeners
  - Restore channel basic.qos setting, publisher confirms and transaction settings
- Automatic recovery is enabled by default.

# Connection recovery

- To disable automatic connection recovery:

```
ConnectionFactory factory = new ConnectionFactory();  
factory.setUsername(userName);  
factory.setPassword(password);  
factory.setVirtualHost(virtualHost);  
factory.setHost(hostName);  
factory.setPort(portNumber);  
factory.setAutomaticRecoveryEnabled(false);  
Connection conn = factory.newConnection();
```

- If recovery fails due to an exception, it will be retried after a fixed time interval (default is 5 sec):

```
ConnectionFactory factory = new ConnectionFactory();  
factory.setNetworkRecoveryInterval(10000);
```

# Topology recovery

- RabbitMQ Java client supports automatic recovery of topology (queues, exchanges, bindings, and consumers).
- Topology recovery includes the following actions, performed for every channel:
  - Re-declare exchanges
  - Re-declare queues
  - Recover all bindings
  - Recover all consumers
- Automatic recovery is enabled by default.

# Topology recovery

- Topology recovery can be disabled explicitly if needed:

```
ConnectionFactory factory = new ConnectionFactory();  
Connection conn = factory.newConnection();  
// disable topology recovery  
factory.setTopologyRecoveryEnabled(false);
```

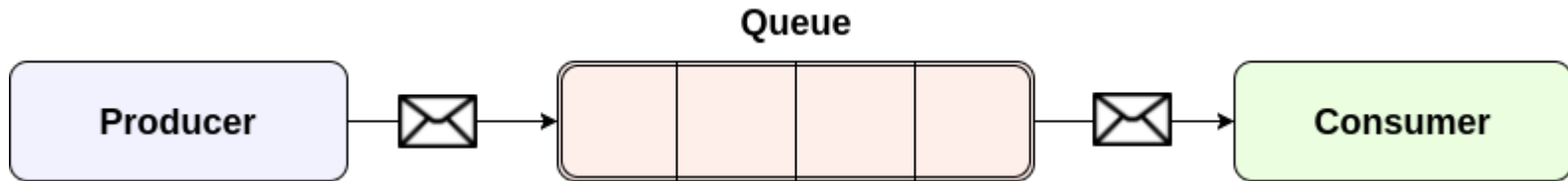


# “Hello World” example



# “Hello World” of messaging

- Producer sends a single message.
- Consumer receives messages and prints them.



# Preparing

- Download and install Erlang/OTP.
- Download and install RabbitMQ.
- RabbitMQ server will start automatically on localhost on standard port (5672).
- RabbitMQ Java client and its dependencies will be downloaded and managed by Maven.

# Sending

ex1

- Create a connection to the server and a channel:


```
ConnectionFactory factory = new ConnectionFactory();  
// connect to a broker on the local machine  
factory.setHost("localhost");  
Connection connection = factory.newConnection();  
Channel channel = connection.createChannel();
```

# Sending

ex1

- Declare a queue and publish a message:

```
// queue will only be created if it doesn't exist already  
channel.queueDeclare(QueueName, false, false, false, null);  
String message = "Hello World!";  
channel.basicPublish("", QueueName, null, message.getBytes());
```



The empty string denotes the default exchange:  
messages are routed to the queue with the name  
specified by QueueName

# Sending

ex1

- Close the channel and the connection:

```
channel.close();  
connection.close();
```

# Receiving

ex1

- Open a connection and a channel, and declare the queue:

```
ConnectionFactory factory = new ConnectionFactory();  
factory.setHost("localhost");  
Connection connection = factory.newConnection();  
Channel channel = connection.createChannel();  
// this queue matches up with the queue that sender publishes to  
channel.queueDeclare(QueueName, false, false, false, null);
```

# Receiving

ex1

- Tell the server to deliver us the messages from the queue:

```
Consumer consumer = new DefaultConsumer(channel) {  
    @Override  
    public void handleDelivery(String consumerTag, Envelope envelope,  
        AMQP.BasicProperties properties, byte[] body) throws IOException {  
        String message = new String(body, "UTF-8");  
    }  
};  
channel.basicConsume(QUEUE_NAME, true, consumer);
```

# Result

- Consumer is running waiting for messages.
- Publisher sends message.
- Consumer gets the message from the publisher.
- Consumer keeps running waiting for messages until the channel and connection are closed.

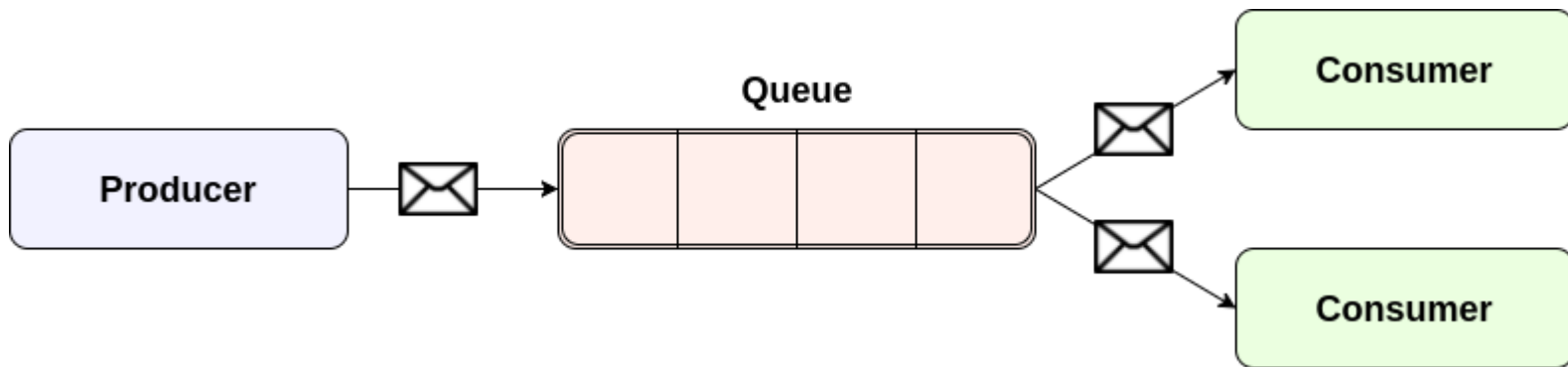




# Competing consumer pattern

# Work queues

- Distribute time-consuming tasks among multiple workers.



# Work queues

- Encapsulate a task as a message and send it to a queue.
- Worker will pop the tasks and eventually execute the job.
- When there are many workers, tasks will be shared between them.

# Sending

ex2

- Create a connection to the server and a channel.
- Declare a queue and publish a message:

```
channel.queueDeclare(QUEUE_NAME, false, false, false, null);  
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
```

- Close the channel and the connection.

# Receiving

ex2

- Create a connection to the server and a channel.
- Handle delivered messages and perform the task:

```
channel.queueDeclare(QUEUE_NAME, false, false, false, null);  
  
Consumer consumer = new DefaultConsumer(channel) {  
    @Override  
    public void handleDelivery(String consumerTag, Envelope envelope,  
        AMQP.BasicProperties properties, byte[] body) throws IOException {  
        ...  
    }  
};  
  
channel.basicConsume(QUEUE_NAME, true, consumer);
```

- Close the channel and the connection.

# Result

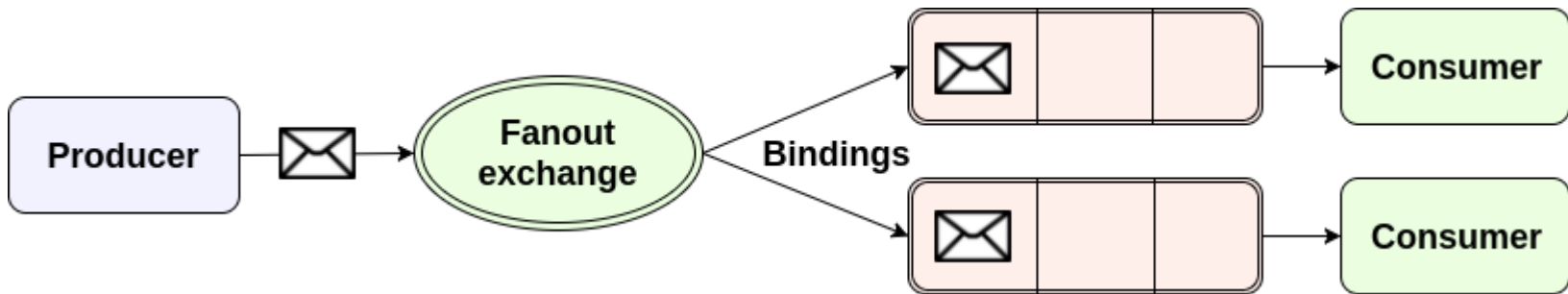
- Consumers are running waiting for messages.
- Publisher sends messages.
- Consumers share sent messages and perform corresponding tasks.
- Consumers keep running waiting for messages until the channel and connection are closed.

A collection of various blue geometric shapes including triangles, squares, and circles, some containing icons like a gear and a lightbulb, scattered on the left side of the slide.

# Publish/Subscribe pattern

# Publisher/Subscriber

- Deliver a message to multiple consumers.





# Publisher/Subscriber

- Producer never sends any messages directly to a queue.
- Producer can only send messages to an exchange.
- Exchange pushes messages to queues.
- The rules for that are defined by the exchange type.
- The **fanout exchange** broadcasts all the messages it receives to all the queues it knows.

# Sending

ex3

- Create a connection to the server and a channel.
- Declare an exchange and publish a message:

```
channel.exchangeDeclare(EXCHANGE_NAME, "fanout");  
channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes());
```

- Close the channel and the connection.



Routing Key is ignored for fanout exchanges

# Temporary queues

- Giving a queue a name is important when sharing it between producers and consumers.
- Deliver a message to multiple consumers requires:
  - Fresh, empty queue whenever we connect to Rabbit.
  - Once the consumer is disconnected, the queue should be automatically deleted.
- `queueDeclare()` method creates a non-durable, exclusive, autodelete queue with a generated name:

```
String queueName = channel.queueDeclare().getQueue();
```

# Receiving

ex3

- Create a connection to the server and a channel.
- Declare an exchange and handle messages:

```
channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
String queueName = channel.queueDeclare().getQueue();
channel.queueBind(queueName, EXCHANGE_NAME, "");
Consumer consumer = new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag, Envelope envelope,
        AMQP.BasicProperties properties, byte[] body) throws IOException {
        ...
    }
};
channel.basicConsume(queueName, true, consumer);
```

- Close the channel and the connection.

# Result

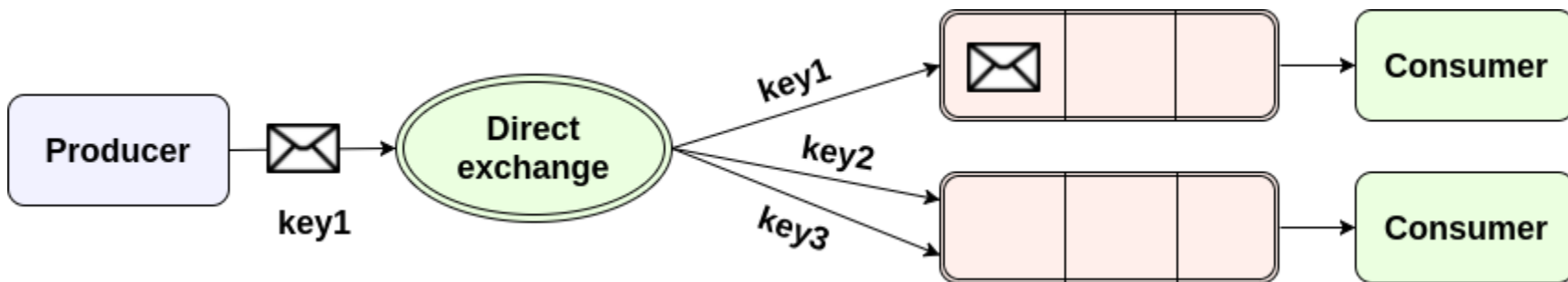
- Subscribers are running waiting for messages.
- Publisher sends messages.
- Each subscriber receives each message.
- Subscribers keep running waiting for messages until the channel and connection are closed.



# Receiving messages selectively

# Routing pattern

- Subscribe only to a subset of the messages.



# Routing pattern

- Bindings can take an extra routingKey parameter:

```
channel.queueBind(queueName, exchangeName, routingKey);
```

- The routing algorithm behind a direct exchange:
  - Message goes to the queues whose binding key exactly matches the message routing key.
- It is legal to bind multiple queues with the same binding key.



# Sending

ex4

- Create a connection to the server and a channel.

- Declare an exchange and publish a message:

```
channel.exchangeDeclare(EXCHANGE_NAME, "direct");  
channel.basicPublish(EXCHANGE_NAME, routingKey, null, message.getBytes());
```

- Close the channel and the connection.

# Receiving

ex4

- Create a connection to the server and a channel.
- Declare an exchange, create bindings, and handle messages:

```
channel.exchangeDeclare(EXCHANGE_NAME, "direct");  
String queueName = channel.queueDeclare().getQueue();  
for (String routingKey : routingKeys) {  
    channel.queueBind(queueName, EXCHANGE_NAME, routingKey);  
}  
Consumer consumer = new DefaultConsumer(channel) {  
    // handle messages  
};  
channel.basicConsume(queueName, true, consumer);
```

- Close the channel and the connection.

# Result

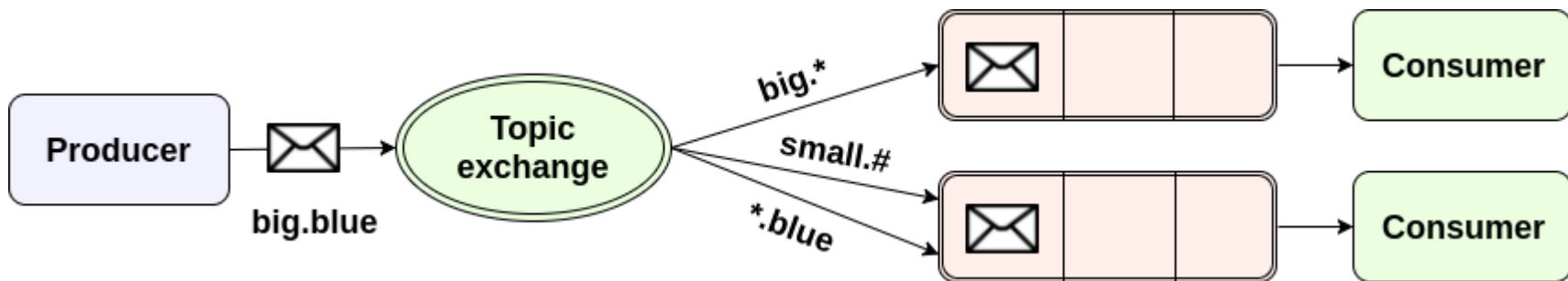
- Subscribers are running waiting for messages.
- Publisher sends messages.
- Subscriber receives message only if its routing key equals message's routing key.
- Subscribers keep running waiting for messages until the channel and connection are closed.



# Receiving messages based on pattern

# Topics

- Subscribe to messages based on pattern.



# Topic exchange

- Message routing key - a list of words, delimited by dots.
  - "quick.orange.rabbit"
- The binding key must be in the same form.
- Message is delivered to all queues with matching binding key.

# Binding keys

- \* (star) can substitute for exactly one word.
- # (hash) can substitute for zero or more words.

# Sending

ex5

- Create a connection to the server and a channel.
- Declare an exchange and publish a message:

```
channel.exchangeDeclare(EXCHANGE_NAME, "topic");  
channel.basicPublish(EXCHANGE_NAME, routingKey, null, message.getBytes());
```

- Close the channel and the connection.



# Receiving

ex5

- Create a connection to the server and a channel.
- Declare an exchange, create bindings, and handle messages:

```
channel.exchangeDeclare(EXCHANGE_NAME, "topic");  
String queueName = channel.queueDeclare().getQueue();  
for (String routingKey : routingKeys) {  
    channel.queueBind(queueName, EXCHANGE_NAME, routingKey);  
}  
Consumer consumer = new DefaultConsumer(channel) {  
    // handle messages  
};  
channel.basicConsume(queueName, true, consumer);
```

- Close the channel and the connection.

# Result

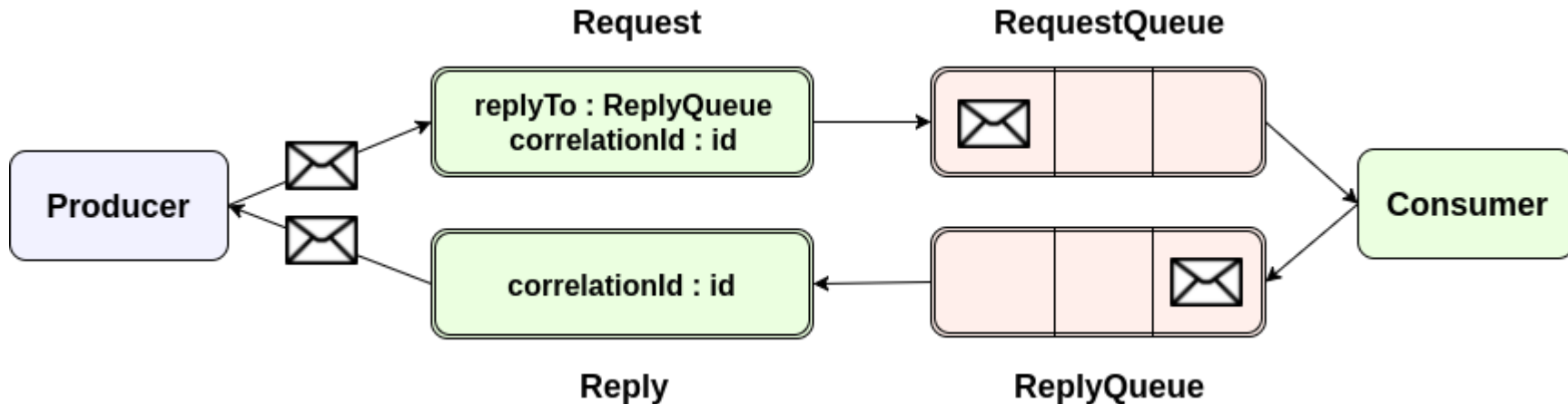
- Subscribers are running waiting for messages.
- Publisher sends messages.
- Subscriber receives message only if its routing key matches message's routing key.
- Subscribers keep running waiting for messages until the channel and connection are closed.



# Request/Reply pattern

# Remote Call Procedure (RCP)

- Run a function on a remote computer and wait for the result.



# Request/Reply pattern

- RPC client sends an RPC request and blocks until the answer is received.
- RPC server executes task and sends reply to client.
- Client sends a 'callback' queue address and correlationId with the request.
  - One callback queue per client.
  - One correlationId per request.

# Message properties

- deliveryMode
- contentType
- replyTo
- correlationId
- etc.

# Sending request

ex6

- Create a connection to the server and a channel.
- Publish a message:

```
AMQP.BasicProperties props = new AMQP.BasicProperties
    .Builder()
    .correlationId(correlationId)
    .replyTo(replyQueueName)
    .build();
channel.basicPublish("", REQUEST_QUEUE_NAME, props, message.getBytes("UTF-8"));
```

# Handling response

ex6

- Handle the response:

```
final BlockingQueue<String> response = new ArrayBlockingQueue<>(1);
channel.basicConsume(replyQueueName, true, new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag, Envelope envelope,
        AMQP.BasicProperties properties, byte[] body) throws IOException {
        if (properties.getCorrelationId().equals(correlationId)) {
            response.offer(new String(body, "UTF-8"));
        }
    }
});
return response.take();
```

- Close the channel and the connection



# Sending response

ex6

- Create a connection to the server and a channel.
- Handle messages and send response:

```
Consumer consumer = new DefaultConsumer(channel) {  
    @Override  
    public void handleDelivery(String consumerTag, Envelope envelope,  
        AMQP.BasicProperties properties, byte[] body) throws IOException {  
        AMQP.BasicProperties replyProps = new AMQP.BasicProperties  
            .Builder().correlationId(properties.getCorrelationId()).build();  
        String message = new String(body, "UTF-8");  
        String response = ... ;  
        channel.basicPublish("", properties.getReplyTo(), replyProps,  
            response.getBytes("UTF-8"));  
    }  
};
```

- Close the channel and the connection

# Result

- Client starts up and creates anonymous exclusive callback queue.
- Client sends a message with two properties: replyTo and correlationId.
- Worker is waiting for requests.
- When request appears, it does the job and sends a message with result to callback queue.
- Client waits for data on the callback queue.
- When message appears, it checks the correlationId.
- If it matches the value from the request it returns the response to the application.

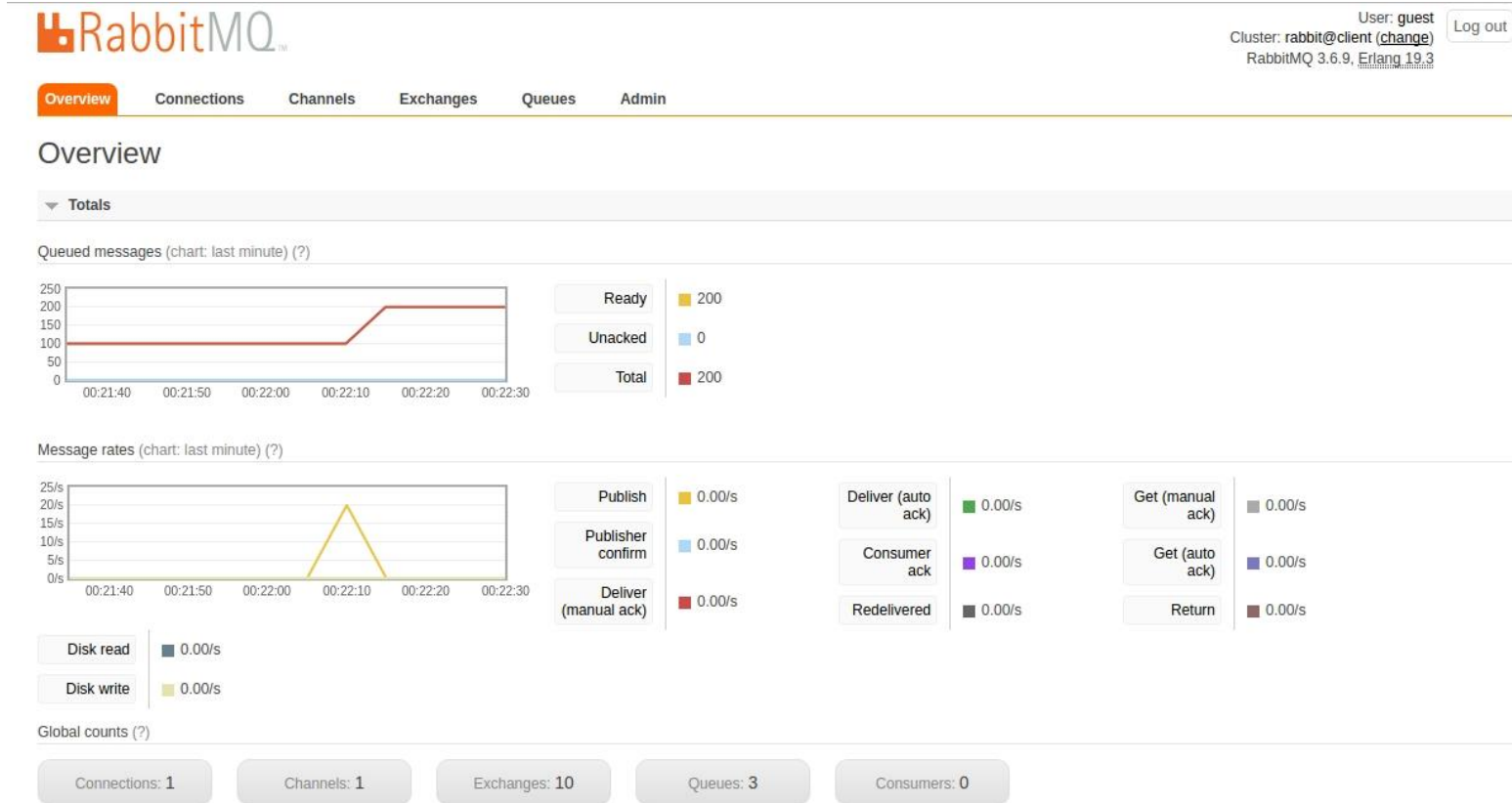


# Management interface

# RabbitMQ management interface

- Plugin for RabbitMQ.
- Single static HTML page which makes background queries to the HTTP API.
- Allows RabbitMQ server monitoring and handling from a web browser.
- Useful for debug or overview of the whole system.
- The Web UI is located at: <http://server-name:15672/>.

# RabbitMQ management interface



# RabbitMQ management interface

- Overview
  - Queued messages, message rate, nodes, etc.
- Connections
- Channels
- Exchanges
- Queues
- Users



Thank you!