Java Client API Guide

This guide covers RabbitMQ Java client API. It is not, however, a tutorial. Those are [available in a different section](https://www.rabbitmq.com/getstarted.html).

5.x release series of this library require JDK 8, both for compilation and at runtime. On Android, this means only [Android 7.0 or later](https://developer.android.com/guide/platform/j8-jack.html) versions are supported. 4.x release series support JDK 6 and Android versions prior to 7.0.

The library is [open-source](https://github.com/rabbitmq/rabbitmq-java-client/), and is triple-licensed under

* [Apache Public License 2.0](http://www.apache.org/licenses/LICENSE-2.0.html)
* [Mozilla Public License](https://www.mozilla.org/MPL/1.1/)
* [GPL 2.0](http://www.gnu.org/licenses/gpl-2.0.html)

This means that the user can consider the library to be licensed under any of the licenses from the list above. For example, the user may choose the Apache Public License 2.0 and include this client into a commercial product. Codebases that are licensed under the GPLv2 may choose GPLv2, and so on.

An [API reference](https://rabbitmq.github.io/rabbitmq-java-client/api/current/) (JavaDoc) is available separately.

There are also [command line tools](https://www.rabbitmq.com/java-tools.html) that used to be shipped with the Java client.

The client API is closely modelled on the AMQP 0-9-1 protocol specification, with additional abstractions for ease of use.

**[Overview](https://www.rabbitmq.com/api-guide.html" \l "classoverview)**

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

* Channel
* Connection
* ConnectionFactory
* Consumer

Protocol operations are available through the Channel interface. Connection is used to open channels, register connection lifecycle event handlers, and close connections that are no longer needed. Connections are instantiated through ConnectionFactory, which is how you configure various connection settings, such as the vhost or username.

**[Connections and Channels](https://www.rabbitmq.com/api-guide.html" \l "connecting)**

The core API classes are Connection and Channel, representing an AMQP 0-9-1 connection and channel, respectively. They are typically imported before used:

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;

[**Connecting to a broker**](https://www.rabbitmq.com/api-guide.html#connecting)

The following code connects to an AMQP broker using the given parameters (host name, port number, etc):

ConnectionFactory factory = new ConnectionFactory();

factory.setUsername(userName);

factory.setPassword(password);

factory.setVirtualHost(virtualHost);

factory.setHost(hostName);

factory.setPort(portNumber);

Connection conn = factory.newConnection();

All of these parameters have sensible defaults for a RabbitMQ server running locally.

Alternatively, [URIs](https://www.rabbitmq.com/uri-spec.html) may be used:

ConnectionFactory factory = new ConnectionFactory();

factory.setUri("amqp://userName:password@hostName:portNumber/virtualHost");

Connection conn = factory.newConnection();

All of these parameters have sensible defaults for a stock RabbitMQ server running locally.

The Connection interface can then be used to open a channel:

Channel channel = conn.createChannel();

The channel can now be used to send and receive messages, as described in subsequent sections.

To disconnect, simply close the channel and the connection:

channel.close();

conn.close();

Note that closing the channel may be considered good practice, but isn’t strictly necessary here - it will be done automatically anyway when the underlying connection is closed.

**[Using Exchanges and Queues](https://www.rabbitmq.com/api-guide.html" \l "exchanges-and-queues)**

Client applications work with exchanges and queues, the high-level building blocks of AMQP. These must be "declared" before they can be used. Declaring either type of object simply ensures that one of that name exists, creating it if necessary.

Continuing the previous example, the following code declares an exchange and a queue, then binds them together.

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

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

channel.queueBind(queueName, exchangeName, routingKey);

This will actively declare the following objects, both of which can be customised by using additional parameters. Here neither of them have any special arguments.

1. a durable, non-autodelete exchange of "direct" type
2. a non-durable, exclusive, autodelete queue with a generated name

The above function calls then bind the queue to the exchange with the given routing key.

Note that this would be a typical way to declare a queue when only one client wants to work with it: it doesn’t need a well-known name, no other client can use it (exclusive) and will be cleaned up automatically (autodelete). If several clients want to share a queue with a well-known name, this code would be appropriate:

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

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

channel.queueBind(queueName, exchangeName, routingKey);

This will actively declare:

1. a durable, non-autodelete exchange of "direct" type
2. a durable, non-exclusive, non-autodelete queue with a well-known name

Note that all of these Channel API methods are overloaded. These convenient short forms of exchangeDeclare, queueDeclare and queueBind use sensible defaults. There are also longer forms with more parameters, to let you override these defaults as necessary, giving full control where needed.

This "short form, long form" pattern is used throughout the client API uses.

**[Publishing messages](https://www.rabbitmq.com/api-guide.html" \l "publishing)**

To publish a message to an exchange, use Channel.basicPublish as follows:

byte[] messageBodyBytes = "Hello, world!".getBytes();

channel.basicPublish(exchangeName, routingKey, null, messageBodyBytes);

For fine control, you can use overloaded variants to specify the mandatory flag, or send messages with pre-set message properties:

channel.basicPublish(exchangeName, routingKey, mandatory,

MessageProperties.PERSISTENT\_TEXT\_PLAIN,

messageBodyBytes);

This sends a message with delivery mode 2 (persistent), priority 1 and content-type "text/plain". You can build your own message properties object, using a Builder class mentioning as many properties as you like, for example:

channel.basicPublish(exchangeName, routingKey,

new AMQP.BasicProperties.Builder()

.contentType("text/plain")

.deliveryMode(2)

.priority(1)

.userId("bob")

.build()),

messageBodyBytes);

This example publishes a message with 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()

.headers(headers)

.build()),

messageBodyBytes);

This example publishes a message with expiration:

channel.basicPublish(exchangeName, routingKey,

new AMQP.BasicProperties.Builder()

.expiration("60000")

.build()),

messageBodyBytes);

We have not illustrated all the possibilities here.

Note that BasicProperties is an inner class of the autogenerated holder class AMQP.

Invocations of Channel#basicPublish will eventually block if a [resource-driven alarm](http://www.rabbitmq.com/alarms.html) is in effect.

**[Channels and Concurrency Considerations (Thread Safety)](https://www.rabbitmq.com/api-guide.html" \l "channel-threads)**

As a rule of thumb, sharing Channel instances between threads is something to be avoided. Applications should prefer using a Channel per thread instead of sharing the same Channelacross multiple threads.

While some operations on channels are safe to invoke concurrently, some are not and will result in incorrect frame interleaving on the wire, double acknowledgements and so on.

Concurrent publishing on a shared channel can result in incorrect frame interleaving on the wire, triggering a connection-level protocol exception and connection closure. It therefore requires explicit synchronization in application code (Channel#basicPublish must be invoked in a critical section). Sharing channels between threads will also interfere with [Publisher Confirms](https://www.rabbitmq.com/confirms.html). We highly recommend avoiding concurrent publishing on a shared channel.

Consuming in one thread and publishing in another thread on a shared channel can be safe.

Server-pushed deliveries (see the section below) are dispatched concurrently with a guarantee that per-channel ordering is preserved. The dispatch mechanism uses a [java.util.concurrent.ExecutorService](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html), one per connection. It is possible to provide a custom executor that will be shared by all connections produced by a single ConnectionFactory using the ConnectionFactory#setSharedExecutor setter.

When [manual acknowledgements](https://www.rabbitmq.com/confirms.html) are used, it is important to consider what thread does the acknowledgement. If it's different from the thread that received the delivery (e.g. Consumer#handleDelivery delegated delivery handling to a different thread), acknowledging with the multiple parameter set to true is unsafe and will result in double-acknowledgements, and therefore a channel-level protocol exception that closes the channel. Acknowledging a single message at a time can be safe.

**[Receiving Messages by Subscription ("Push API")](https://www.rabbitmq.com/api-guide.html" \l "consuming)**

import com.rabbitmq.client.Consumer;

import com.rabbitmq.client.DefaultConsumer;

The most efficient way to receive messages is to set up a subscription using the Consumerinterface. The messages will then be delivered automatically as they arrive, rather than having to be explicitly requested.

When calling the API methods relating to Consumers, individual subscriptions are always referred to by their consumer tags. A consumer tag is a consumer identifier which can be either client- or server-generated. To let RabbitMQ generate a node-wide unique tag, use a Channel#basicConsume override that doesn't take a consumer tag argument or pass an empty string for consumer tag and use the value returned by Channel#basicConsume. Consumer tags are used to cancel consumers.

Distinct Consumer instances must have distinct consumer tags. Duplicate consumer tags on a connection is strongly discouraged and can lead to issues with automatic connection recovery and confusing monitoring data when consumers are monitored.

The easiest way to implement a Consumer is to subclass the convenience class DefaultConsumer. An object of this subclass can be passed on a basicConsume call to set up the 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);

}

});

Here, since we specified autoAck = false, it is necessary to acknowledge messages delivered to the Consumer, most conveniently done in the handleDelivery method, as illustrated.

More sophisticated Consumers will need to override further methods. In particular, handleShutdownSignal is called when channels and connections close, and handleConsumeOk is passed the consumer tag before any other callbacks to that Consumer are called.

Consumers can also implement the handleCancelOk and handleCancel methods to be notified of explicit and implicit cancellations, respectively.

You can explicitly cancel a particular Consumer with Channel.basicCancel:

channel.basicCancel(consumerTag);

passing the consumer tag.

Just like with publishers, it is important to consider concurrency hazard safety for consumers.

Callbacks to Consumers are dispatched in a thread pool separate from the thread that instantiated its Channel. This means that Consumers can safely call blocking methods on the Connection or Channel, such as Channel#queueDeclare or Channel#basicCancel.

Each Channel has its own dispatch thread. For the most common use case of one Consumer per Channel, this means Consumers do not hold up other Consumers. If you have multiple Consumers per Channel be aware that a long-running Consumer may hold up dispatch of callbacks to other Consumers on that Channel.

Please refer to the Concurrency Considerations (Thread Safety) section for other topics related to concurrency and concurrency hazard safety.

**[Retrieving Individual Messages ("Pull API")](https://www.rabbitmq.com/api-guide.html" \l "getting)**

To explicitly retrieve messages, use Channel.basicGet. The returned value is an instance of GetResponse, from which the header information (properties) and message body can be extracted:

boolean autoAck = false;

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

if (response == null) {

*// No message retrieved.*

} else {

AMQP.BasicProperties props = response.getProps();

byte[] body = response.getBody();

long deliveryTag = response.getEnvelope().getDeliveryTag();

...

and since the autoAck = false above, you must also call Channel.basicAck to acknowledge that you have successfully received the message:

...

channel.basicAck(method.deliveryTag, false); *// acknowledge receipt of the message*

}

**[Handling unroutable messages](https://www.rabbitmq.com/api-guide.html" \l "returning)**

If a message is published with the "mandatory" flags set, but cannot be routed, the broker will return it to the sending client (via a AMQP.Basic.Return command).

To be notified of such returns, clients can implement the ReturnListener interface and call Channel.addReturnListener. If the client has not configured a return listener for a particular channel, then the associated returned messages will be silently dropped.

channel.addReturnListener(new ReturnListener() {

public void handleReturn(int replyCode,

String replyText,

String exchange,

String routingKey,

AMQP.BasicProperties properties,

byte[] body)

throws IOException {

...

}

});

A return listener will be called, for example, if the client publishes a message with the "mandatory" flag set to an exchange of "direct" type which is not bound to a queue.

**[Shutdown Protocol](https://www.rabbitmq.com/api-guide.html" \l "shutdown)**

**[Overview of the AMQP client shutdown](https://www.rabbitmq.com/api-guide.html" \l "shutdown-overview)**

The AMQP 0-9-1 connection and channel share the same general approach to managing network failure, internal failure, and explicit local shutdown.

The AMQP 0-9-1 connection and channel have the following lifecycle states:

* open: the object is ready to use
* closing: the object has been explicitly notified to shut down locally, has issued a shutdown request to any supporting lower-layer objects, and is waiting for their shutdown procedures to complete
* closed: the object has received all shutdown-complete notification(s) from any lower-layer objects, and as a consequence has shut itself down

Those objects always end up in the closed state, regardless of the reason that caused the closure, like an application request, an internal client library failure, a remote network request or network failure.

The AMQP connection and channel objects possess the following shutdown-related methods:

* addShutdownListener(ShutdownListener listener) and removeShutdownListener(ShutdownListener listener), to manage any listeners, which will be fired when the object transitions to closedstate. Note that, adding a ShutdownListener to an object that is already closed will fire the listener immediately
* getCloseReason(), to allow the investigation of what was the reason of the object’s shutdown
* isOpen(), useful for testing whether the object is in an open state
* close(int closeCode, String closeMessage), to explicitly notify the object to shut down

Simple usage of listeners would look like:

import com.rabbitmq.client.ShutdownSignalException;

import com.rabbitmq.client.ShutdownListener;

connection.addShutdownListener(new ShutdownListener() {

public void shutdownCompleted(ShutdownSignalException cause)

{

...

}

});

**[Information about the circumstances of a shutdown](https://www.rabbitmq.com/api-guide.html" \l "shutdown-cause)**

One can retrieve the ShutdownSignalException, which contains all the information available about the close reason, either by explicitly calling the getCloseReason() method or by using the causeparameter in the service(ShutdownSignalException cause) method of the ShutdownListener class.

The ShutdownSignalException class provides methods to analyze the reason of the shutdown. By calling the isHardError() method we get information whether it was a connection or a channel error, and getReason() returns information about the cause, in the form an AMQP method - either AMQP.Channel.Close or AMQP.Connection.Close (or null if the cause was some exception in the library, such as a network communication failure, in which case that exception can be retrieved with getCause()).

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();

...

}

}

**[Atomicity and use of the isOpen() method](https://www.rabbitmq.com/api-guide.html" \l "shutdown-atomicity)**

Use of the isOpen() method of channel and connection objects is not recommended for production code, because the value returned by the method is dependent on the existence of the shutdown cause. The following code illustrates the possibility of race conditions:

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);

}

}

Instead, we should normally ignore such checking, and simply attempt the action desired. If during the execution of the code the channel of the connection is closed, a ShutdownSignalException will be thrown indicating that the object is in an invalid state. We should also catch for IOException caused either by SocketException, when broker closes the connection unexpectedly, or ShutdownSignalException, when broker initiated clean close.

public void validMethod(Channel channel)

{

try {

...

channel.basicQos(1);

} catch (ShutdownSignalException sse) {

*// possibly check if channel was closed*

*// by the time we started action and reasons for*

*// closing it*

...

} catch (IOException ioe) {

*// check why connection was closed*

...

}

}

**[Advanced Connection options](https://www.rabbitmq.com/api-guide.html" \l "advanced-connection)**

***[Consumer thread pool](https://www.rabbitmq.com/api-guide.html" \l "consumer-thread-pool)***

Consumer threads (see [Receiving](https://www.rabbitmq.com/api-guide.html#consuming) below) are automatically allocated in a new ExecutorServicethread pool by default. If greater control is required supply an ExecutorService on the newConnection() method, so that this pool of threads is used instead. Here is an example where a larger thread pool is supplied than is normally allocated:

ExecutorService es = Executors.newFixedThreadPool(20);

Connection conn = factory.newConnection(es);

Both Executors and ExecutorService classes are in the java.util.concurrent package.

When the connection is closed a default ExecutorService will be shutdown(), but a user-suppliedExecutorService (like es above) will *not* be shutdown(). Clients that supply a custom ExecutorService must ensure it is shutdown eventually (by calling its shutdown() method), or else the pool’s threads may prevent JVM termination.

The same executor service may be shared between multiple connections, or serially re-used on re-connection but it cannot be used after it is shutdown().

Use of this feature should only be considered if there is evidence that there is a severe bottleneck in the processing of Consumer callbacks. If there are no Consumer callbacks executed, or very few, the default allocation is more than sufficient. The overhead is initially minimal and the total thread resources allocated are bounded, even if a burst of consumer activity may occasionally occur.

**[Using Lists of Hosts](https://www.rabbitmq.com/api-guide.html" \l "address-array)**

It is possible to pass an Address array to newConnection(). An Address is simply a convenience class in the com.rabbitmq.client package with *host* and *port* components. For example:

Address[] addrArr = new Address[]{ new Address(hostname1, portnumber1)

, new Address(hostname2, portnumber2)};

Connection conn = factory.newConnection(addrArr);

will attempt to connect to hostname1:portnumber1, and if that fails to hostname2:portnumber2. The connection returned is the first in the array that succeeds (without throwing IOException). This is entirely equivalent to repeatedly setting host and port on a factory, calling factory.newConnection() each time, until one of them succeeds.

If an ExecutorService is provided as well (using the form factory.newConnection(es, addrArr)) the thread pool is associated with the (first) successful connection.

If you want more control over the host to connect to, see [the support for service discovery](https://www.rabbitmq.com/api-guide.html#service-discovery-with-address-resolver).

**[Service discovery with the AddressResolver interface](https://www.rabbitmq.com/api-guide.html" \l "service-discovery-with-address-resolver)**

As of version 3.6.6, 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;

}

Just like with [a list of hosts](https://www.rabbitmq.com/api-guide.html#address-array), the first Address returned will be tried first, then the second if the client fails to connect to the first, and so on.

If an ExecutorService is provided as well (using the form factory.newConnection(es, addressResolver)) the thread pool is associated with the (first) successful connection.

The AddressResolver is the perfect place to implement custom service discovery logic, which is especially useful in a dynamic infrastructure. Combined with [automatic recovery](https://www.rabbitmq.com/api-guide.html#recovery), the client can automatically connect to nodes that weren't even up when it was first started. Affinity and load balancing are other scenarios where a custom AddressResolver could be useful.

The Java client ships with the following implementations (see the javadoc for details):

1. DnsRecordIpAddressResolver: given the name of a host, returns its IP addresses (resolution against the platform DNS server). This can be useful for simple DNS-based load balancing or failover.
2. DnsSrvRecordAddressResolver: given the name of a service, returns hostname/port pairs. The search is implemented as a DNS SRV request. This can be useful when using a service registry like [HashiCorp Consul](https://www.consul.io/).

**[Heartbeat Timeout](https://www.rabbitmq.com/api-guide.html" \l "heartbeats-timeout)**

See the [Heartbeats guide](https://www.rabbitmq.com/heartbeats.html) for more information about heartbeats and how to configure them in the Java client.

***[Custom Thread Factories](https://www.rabbitmq.com/api-guide.html" \l "thread-factories)***

Environments such as Google App Engine (GAE) can [restrict direct thread instantiation](https://developers.google.com/appengine/docs/java/#Java_The_sandbox). To use RabbitMQ Java client in such environments, it's necessary to configure a custom ThreadFactorythat uses an appropriate method to instantiate threads, e.g. GAE's ThreadManager. Below is an example for Google App Engine.

import com.google.appengine.api.ThreadManager;

ConnectionFactory cf = new ConnectionFactory();

cf.setThreadFactory(ThreadManager.backgroundThreadFactory());

**[Support for Java non-blocking IO](https://www.rabbitmq.com/api-guide.html" \l "java-nio)**

Version 4.0 of the Java client brings *experimental* support for Java non-blocking IO (a.k.a Java NIO). NIO isn't supposed to be faster than blocking IO, it simply allows to control resources (in this case, threads) more easily.

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.

Use the NIO mode if your Java process uses many connections (dozens or hundreds). You should use fewer threads than with the default blocking mode. With the appropriate number of threads set, you shouldn't experiment any decrease in performance, especially if the connections are not so busy.

NIO must be enabled explicitly:

ConnectionFactory connectionFactory = new ConnectionFactory();

connectionFactory.useNio();

The NIO mode can be configured through the NioParams class:

connectionFactory.setNioParams(new NioParams().setNbIoThreads(4));

The NIO mode uses reasonable defaults, but you may need to change them according to your own workload. Some of the settings are: the total number of IO threads used, the size of buffers, a service executor to use for the IO loops, parameters for the in-memory write queue (write requests are enqueued before being sent on the network). Please read the Javadoc for details and defaults.

**[Automatic Recovery From Network Failures](https://www.rabbitmq.com/api-guide.html" \l "recovery)**

**[Connection Recovery](https://www.rabbitmq.com/api-guide.html" \l "connection-recovery)**

Network connection between clients and RabbitMQ nodes can fail. RabbitMQ Java client supports automatic recovery of connections and topology (queues, exchanges, bindings, and consumers). The automatic recovery process for many applications follows the following steps:

1. Reconnect
2. Restore connection listeners
3. Re-open channels
4. Restore channel listeners
5. Restore channel basic.qos setting, publisher confirms and transaction settings

Topology recovery includes the following actions, performed for every channel

1. Re-declare exchanges (except for predefined ones)
2. Re-declare queues
3. Recover all bindings
4. Recover all consumers

*As of version 4.0.0 of the Java client, automatic recovery is enabled by default (and thus topology recovery as well)*.

To disable or enable automatic connection recovery, use the factory.setAutomaticRecoveryEnabled(boolean) method. The following snippet shows how to explicitly enable automatic recovery (e.g. for Java client prior 4.0.0):

ConnectionFactory factory = new ConnectionFactory();

factory.setUsername(userName);

factory.setPassword(password);

factory.setVirtualHost(virtualHost);

factory.setHost(hostName);

factory.setPort(portNumber);

factory.setAutomaticRecoveryEnabled(true);

*// connection that will recover automatically*

Connection conn = factory.newConnection();

If recovery fails due to an exception (e.g. RabbitMQ node is still not reachable), it will be retried after a fixed time interval (default is 5 seconds). The interval can be configured:

ConnectionFactory factory = new ConnectionFactory();

*// attempt recovery every 10 seconds*

factory.setNetworkRecoveryInterval(10000);

When a list of addresses is provided, the list is shuffled and all addresses are tried, one after the next:

ConnectionFactory factory = new ConnectionFactory();

Address[] addresses = {new Address("192.168.1.4"), new Address("192.168.1.5")};

factory.newConnection(addresses);

**[When Will Connection Recovery Be Triggered?](https://www.rabbitmq.com/api-guide.html" \l "connection-recovery-triggers)**

Automatic connection recovery, if enabled, will be triggered by the following events:

* An I/O exception is thrown in connection's I/O loop
* A socket read operation times out
* Missed server [heartbeats](https://www.rabbitmq.com/heartbeats.html) are detected
* Any other unexpected exception is thrown in connection's I/O loop

whichever happens first.

Channel-level exceptions will not trigger any kind of recovery as they usually indicate a semantic issue in the application (e.g. an attempt to consume from a non-existent queue).

**[Recovery Listeners](https://www.rabbitmq.com/api-guide.html" \l "recovery-listeners)**

It is possible to register one or more recovery listeners on recoverable connections and channels. When connection recovery is enabled, connections returned by ConnectionFactory#newConnection and Connection#createChannel implement com.rabbitmq.client.Recoverable, providing two methods with fairly descriptive names:

* addRecoveryListener
* removeRecoveryListener

Note that you currently need to cast connections and channels to Recoverable in order to use those methods.

**[Effects on Publishing](https://www.rabbitmq.com/api-guide.html" \l "publishers)**

Messages that are published using Channel.basicPublish when connection is down will be lost. The client does not enqueue them for delivery after connection has recovered. To ensure that published messages reach RabbitMQ applications need to use [Publisher Confirms](https://www.rabbitmq.com/confirms.html) and account for connection failures.

**[Topology Recovery](https://www.rabbitmq.com/api-guide.html" \l "topology-recovery)**

Topology recovery involves recovery of exchanges, queues, bindings and consumers. It is enabled by default when automatic recovery is enabled. *Hence topology recovery is enabled by default as of Java client 4.0.0*.

Topology recovery can be disabled explicitly if needed:

ConnectionFactory factory = new ConnectionFactory();

Connection conn = factory.newConnection();

*// enable automatic recovery (e.g. Java client prior 4.0.0)*

factory.setAutomaticRecoveryEnabled(true);

*// disable topology recovery*

factory.setTopologyRecoveryEnabled(false);

**[Failure Detection and Recovery Limitations](https://www.rabbitmq.com/api-guide.html" \l "automatic-recovery-limitations)**

Automatic connection recovery has a number of limitations and intentional design decisions that applications developers need to be aware of.

When a connection is down or lost, it [takes time to detect](https://www.rabbitmq.com/heartbeats.html). Therefore there is a window of time in which both the library and the application are unaware of effective connection failure. Any messages published during this time frame are serialised and written to the TCP socket as usual. Their delivery to the broker can only be guaranteed via [publisher confirms](https://www.rabbitmq.com/confirms.html): publishing in AMQP 0-9-1 is entirely asynchronous by design.

When a socket or I/O operation error is detected by a connection with automatic recovery enabled, recovery begins after a configurable delay, 5 seconds by default. This design assumes that even though a lot of network failures are transient and generally short lived, they do not go away in an instant. Connection recovery attempts will continue at identical time intervals until a new connection is successfully opened.

When a connection is in the recovering state, any publishes attempted on its channels will be rejected with an exception. The client currently does not perform any internal buffering of such outgoing messages. It is an application developer's responsibility to keep track of such messages and republish them when recovery succeeds. [Publisher confirms](https://www.rabbitmq.com/confirms.html) is a protocol extension that should be used by publishers that cannot afford message loss.

Connection recovery will not kick in when a channel is closed due to a channel-level exception. Such exceptions often indicate application-level issues. The library cannot make an informed decision about when that's the case.

Closed channels won't be recovered even after connection recovery kicks in. This includes both explicitly closed channels and the channel-level exception case above.

**[Manual Acknowledgements and Automatic Recovery](https://www.rabbitmq.com/api-guide.html" \l "recovery-and-acknowledgements)**

When manual acknowledgements are used, it is possible that network connection to RabbitMQ node fails between message delivery and acknowledgement. After connection recovery, RabbitMQ will reset delivery tags on all channels. This means that *basic.ack*, *basic.nack*, and *basic.reject* with old delivery tags will cause a channel exception. To avoid this, RabbitMQ Java client keeps track of and updates delivery tags to make them monotonically growing between recoveries. Channel.basicAck, Channel.basicNack, and Channel.basicReject then translate adjusted delivery tags into those used by RabbitMQ. Acknowledgements with stale delivery tags will not be sent. Applications that use manual acknowledgements and automatic recovery must be capable of handling redeliveries.

**[Channels Lifecycle and Topology Recovery](https://www.rabbitmq.com/api-guide.html" \l "recovery-channel-lifecycle)**

Automatic connection recovery is meant to be as transparent as possible for the application developer, that's why Channel instances remain the same even if several connections fail and recover behind the scenes. Technically, when automatic recovery is on, Channel instances act as proxies or decorators: they delegate the AMQP business to an actual AMQP channel implementation and implement some recovery machinery around it. That is why you shouldn't close a channel after it has created some resources (queues, exchanges, bindings) or topology recovery for those resources will fail later, as the channel has been closed. Instead, leave creating channels open for the life of the application.

**[Unhandled Exceptions](https://www.rabbitmq.com/api-guide.html" \l "unhandled-exceptions)**

Unhandled exceptions related to connection, channel, recovery, and consumer lifecycle are delegated to the exception handler. Exception handler is any object that implements the ExceptionHandler interface. By default, an instance of DefaultExceptionHandler is used. It prints exception details to the standard output.

It is possible to override the handler using ConnectionFactory#setExceptionHandler. It will be used for all connections created by the factory:

ConnectionFactory factory = new ConnectionFactory();

cf.setExceptionHandler(customHandler);

Exception handlers should be used for exception logging.

**[Metrics and monitoring](https://www.rabbitmq.com/api-guide.html" \l "metrics)**

As of version 4.0.0, the client gathers runtime metrics (e.g. number of published messages). Metrics collection is optional and is set up at the ConnectionFactory level, using the setMetricsCollector(metricsCollector) method. This method expects a MetricsCollectorinstance, which is called in several places of the client code.

The client supports [Micrometer](http://micrometer.io/) (as of version 4.3) and [Dropwizard Metrics](http://metrics.dropwizard.io/) out of the box.

Here are the collected metrics:

* Number of open connections
* Number of open channels
* Number of published messages
* Number of consumed messages
* Number of acknowledged messages
* Number of rejected messages

Both Micrometer and Dropwizard Metrics provide counts, but also mean rate, last five minute rate, etc, for messages-related metrics. They also support common tools for monitoring and reporting (JMX, Graphite, Ganglia, Datadog, etc). See the dedicated sections below for more details.

Please note the following about metrics collection:

* Don't forget to add the appropriate dependencies (in Maven, Gradle, or even as JAR files) to JVM classpath when using Micrometer or Dropwizard Metrics. Those are optional dependencies and will not be pulled automatically with the Java client. You may also need to add other dependencies depending on the reporting backend(s) used.
* Metrics collection is extensible. Implementing a custom MetricsCollector for specific needs is encouraged.
* The MetricsCollector is set at the ConnectionFactory level but can be shared across different instances.
* Metrics collection doesn't support transactions. E.g. if an acknowledgment is sent in a transaction and the transaction is then rolled back, the acknowledgment is counted in the client metrics (but not by the broker obviously). Note the acknowledgment is actually sent to the broker and then cancelled by the transaction rollback, so the client metrics are correct in term of acknowledgments sent. As a summary, don't use client metrics for critical business logic, they're not guaranteed to be perfectly accurate. They are meant to be used to simplify reasoning about a running system and make operations more efficient.

**[Micrometer support](https://www.rabbitmq.com/api-guide.html" \l "metrics-micrometer)**

You can enable metrics collection with [Micrometer](http://micrometer.io/) the following way:

ConnectionFactory connectionFactory = new ConnectionFactory();

MicrometerMetricsCollector metrics = new MicrometerMetricsCollector();

connectionFactory.setMetricsCollector(metrics);

...

metrics.getPublishedMessages(); *// get Micrometer's Counter object*

Micrometer supports [several reporting backends](http://micrometer.io/docs): Netflix Atlas, Prometheus, Datadog, Influx, JMX, etc.

You would typically pass in an instance of MeterRegistry to the MicrometerMetricsCollector. Here is an example with JMX:

JmxMeterRegistry registry = new JmxMeterRegistry();

MicrometerMetricsCollector metrics = new MicrometerMetricsCollector(registry);

ConnectionFactory connectionFactory = new ConnectionFactory();

connectionFactory.setMetricsCollector(metrics);

**[Dropwizard Metrics support](https://www.rabbitmq.com/api-guide.html" \l "metrics-dropwizard-metrics)**

You can enable metrics collection with [Dropwizard](http://metrics.dropwizard.io/) the following way:

ConnectionFactory connectionFactory = new ConnectionFactory();

StandardMetricsCollector metrics = new StandardMetricsCollector();

connectionFactory.setMetricsCollector(metrics);

...

metrics.getPublishedMessages(); *// get Metrics' Meter object*

Dropwizard Metrics supports [several reporting backends](http://metrics.dropwizard.io/3.2.3/getting-started.html): console, JMX, HTTP, Graphite, Ganglia, etc.

You would typically pass in an instance of MetricsRegistry to the StandardMetricsCollector. Here is an example with JMX:

MetricRegistry registry = new MetricRegistry();

StandardMetricsCollector metrics = new StandardMetricsCollector(registry);

ConnectionFactory connectionFactory = new ConnectionFactory();

connectionFactory.setMetricsCollector(metrics);

JmxReporter reporter = JmxReporter

.forRegistry(registry)

.inDomain("com.rabbitmq.client.jmx")

.build();

reporter.start();

**[RabbitMQ Java Client on Google App Engine](https://www.rabbitmq.com/api-guide.html" \l "gae-pitfalls)**

Using RabbitMQ Java client on Google App Engine (GAE) requires using a custom thread factory that instantiates thread using GAE's ThreadManager (see above). In addition, it is necessary to set a low heartbeat interval (4-5 seconds) to avoid running into the low InputStream read timeouts on GAE:

ConnectionFactory factory = new ConnectionFactory();

cf.setRequestedHeartbeat(5);

**[Caveats and Limitations](https://www.rabbitmq.com/api-guide.html" \l "cache-pitfalls)**

To make topology recovery possible, RabbitMQ Java client maintains a cache of declared queues, exchanges, and bindings. The cache is per-connection. Certain RabbitMQ features make it impossible for clients to observe some topology changes, e.g. when a queue is deleted due to TTL. RabbitMQ Java client tries to invalidate cache entries in the most common cases:

* When queue is deleted.
* When exchange is deleted.
* When binding is deleted.
* When consumer is cancelled on an auto-deleted queue.
* When queue or exchange is unbound from an auto-deleted exchange.

However, the client cannot track these topology changes beyond a single connection. Applications that rely on auto-delete queues or exchanges, as well as queue TTL (note: not message TTL!), and use automatic connection recovery, should explicitly delete entities know to be unused or deleted, to purge client-side topology cache. This is facilitated by Channel#queueDelete, Channel#exchangeDelete, Channel#queueUnbind, and Channel#exchangeUnbindbeing idempotent in RabbitMQ 3.3.x (deleting what's not there does not result in an exception).

**[The RPC (Request/Reply) Pattern](https://www.rabbitmq.com/api-guide.html" \l "rpc)**

As a programming convenience, the Java client API offers a class RpcClient which uses a temporary reply queue to provide simple [RPC-style communication](https://www.rabbitmq.com/tutorials/tutorial-six-java.html) facilities via AMQP 0-9-1.

The class doesn’t impose any particular format on the RPC arguments and return values. It simply provides a mechanism for sending a message to a given exchange with a particular routing key, and waiting for a response on a reply queue.

import com.rabbitmq.client.RpcClient;

RpcClient rpc = new RpcClient(channel, exchangeName, routingKey);

(The implementation details of how this class uses AMQP 0-9-1 are as follows: request messages are sent with the basic.correlation\_id field set to a value unique for this RpcClientinstance, and with basic.reply\_to set to the name of the reply queue.)

Once you have created an instance of this class, you can use it to send RPC requests by using any of the following methods:

byte[] primitiveCall(byte[] message);

String stringCall(String message)

Map mapCall(Map message)

Map mapCall(Object[] keyValuePairs)

The primitiveCall method transfers raw byte arrays as the request and response bodies. The method stringCall is a thin convenience wrapper around primitiveCall, treating the message bodies as String instances in the default character encoding.

The mapCall variants are a little more sophisticated: they encode a java.util.Map containing ordinary Java values into an AMQP 0-9-1 binary table representation, and decode the response in the same way. (Note that there are some restrictions on what value types can be used here - see the javadoc for details.)

All the marshalling/unmarshalling convenience methods use primitiveCall as a transport mechanism, and just provide a wrapping layer on top of it.

**[TLS Support](https://www.rabbitmq.com/api-guide.html" \l "tls)**

It's possible to encrypt the communication between the client and the broker [using TLS](https://www.rabbitmq.com/ssl.html). Client and server authentication (a.k.a. peer verification) is also supported. Here is the simplest way to use encryption with the Java client:

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

factory.setPort(5671);

factory.useSslProtocol();

Note the client doesn't enforce any server authentication (peer certificate chain verification) in the above sample as the default, "trust all certificates" TrustManager is used. This is convenient for local development but prone to man-in-the-middle attacks and therefore not recommended for production. To learn more about TLS support in RabbitMQ, see the [TLS guide](https://www.rabbitmq.com/ssl.html). If you only want to configure the Java client (especially the peer verification and trust manager parts), read [the appropriate section](https://www.rabbitmq.com/ssl.html#trust-levels) of the TLS guide.