Qpid JMS Client for AMQP protocols 0-8, 0-9 and 0-9-1

Qpid JMS Client for AMQP protocols 0-8, 0-9 and 0-9-1

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction

Qpid JMS client is an implementation of JMS specification 1.1 [http://download.oracle.com/otndocs/jcp/7195-jms-1.1-fr-spec-oth-JSpec/]. It utilises an AMQP [http://www.amqp.org] transport layer for the performing of messaging operations. The client is intended to be used for the writing of JMS compatible messaging applications. Such applications can send and receive messages via any AMQP-compatible brokers like RabbitMQ, Qpid Java Broker which support the AMQP protocols 0-8, 0-9, or 0-9-1.

The Qpid JMS client hides the details of AMQP transport implementation behind the JMS API. Thus, the developers need only to be familiar with JMS API in order to use the client. However, the knowledge of the basic concepts of AMQP protocols can help developers in writing reliable and high-performant messaging application.

Using the Qpid JMS client with 0-10

This book documents the behaviour of the Qpid JMS client when used with the AMQP protocols *0-8, 0-9, and 0-9-1* only. For behaviour when using the client with AMQP 0-10 protocol, please refer to Programming in Apache Qpid [../../Programming-In-Apache-Qpid/html/].

Chapter 2. Document Scope And Intended Audience

The intended audience of this document is Java developers who are familiar with the JMS specification. Readers are not required to know all the details of AMQP protocols. However, some knowledge of AMQP basic concepts would be advantageous for reading of this document.

This document only covers the usage of 0-8, 0-9 and 0-9-1 AMQP protocols with Qpid JMS client. The specifications for these protocols are available from the AMQP web site [http://www.amqp.org].

The document covers some specific implementation details of JMS connections, sessions, consumers and producers in Chapter 5, *Understanding the Qpid JMS client*. It also demonstrates how to write a simple point to point and simple publish/subscribe application using Qpid JMS Client in Chapter 4, *Examples*.

The Qpid JMS Client supports various configuration options which can be set via JVM system properties, connection URLs and JNDI configuration file. The setting of system properties is described in Chapter 9, *System Properties*. The details of supported options within the connection URLs are given in Chapter 7, *Connection URLs*. The details of Qpid JMS client JNDI properties format is provided in Chapter 6, *JNDI Properties Format*. The Qpid destination URL format is covered in Chapter 8, *Binding URL*.

The Qpid JMS Client can be used for writing of JMS vendor neutral messaging applications. However, in some cases it might be required to use specific AMQP features. Thus, the Qpid client provides the extended operation set to invoke those features.

Chapter 10, *Logging* provides the details about turning on client logging which can help in debugging of various issues while developing the messaging applications.

The details about Qpid JMS Client Exceptions are provided in Appendix A, *Exceptions*

Chapter 3. Getting the Client And Dependencies

3.1. Getting the Client

The Qpid JMS client is available as a bundle or from Maven repositories.

The bundle (a .tar.gz) includes the Qpid JMS client itself (formed by two JAR: qpid-client and qpid-common) together with slf4j-api, and geronimo-jms_1.1_spec. There is also a qpid-all JAR artifact that, for convenience, includes a manifest classpath that references the other JARs. The bundle is available from the Apache Qpid project web site [http://qpid.apache.org/download.html].

The Qpid JMS client is also available from Maven repositories. Add the following dependency:

```
<dependency>
  <groupId>org.apache.qpid</groupId>
  <artifactId>qpid-client</artifactId>
  <version>0.32-SNAPSHOT</version>
</dependency>
```

Appendix B, *Minimal Maven POM* illustrates a minimal Maven POM required to use the Qpid Client.

3.2. Dependencies

The Qpid JMS client has minimal set of external dependencies.

It requires:

- JDK 1.7 or higher.
- JMS 1.1 specification (such as geronimo-jms_1.1_spec JAR)
- Apache SLF4J [http://www.slf4j.org] (slf4j-api-x.y.z JAR)

The use of SLF4J means that application authors are free to plug in any logging framework for which an SLF4J binding exists.

Chapter 4. Examples

The following programs shows how to send and receive messages using the Qpid JMS client. The first program illustrates a *point to point* example, the second, a pubish/subscribe example.

Both examples show the use JNDI to obtain connection factory and destination objects which the application needs. In this way the configuration is kept separate from the application code itself.

The example code will be straightforward for anyone familiar with Java JMS. Readers in need of an introduction are directed towards Oracle's JMS tutorial [http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html].

4.1. Point to point example

In this example, we illustrate point to point messaging. We create a JNDI context using a properties file, use the context to lookup a connection factory, create and start a connection, create a session, and lookup a destination (a queue) from the JNDI context. Then we create a producer and a consumer, send a message with the producer and receive it with the consumer.

Example 4.1. JMS Example - Point to Point Messaging

```
import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Properties;

public class Hello {

    public Hello() {
    }

    public static void main(String[] args) throws Exception {
        Hello hello = new Hello();
        hello.runTest();
    }

    private void runTest() throws Exception {
      Properties properties = new Properties();
      properties.load(this.getClass().getResourceAsStream("helloworld.properties")
      Context context = new InitialContext(properties);

      ConnectionFactory connectionFactory
          = (ConnectionFactory) context.lookup("qpidConnectionFactory");
      Connection connection = connectionFactory.createConnection();
      connection.start();

      Session session = connection.createSession(true, Session.SESSION_TRANSACTED)
      Queue queue = (Queue) context.lookup("myqueue");

      MessageConsumer messageConsumer = session.createConsumer(queue);
      MessageProducer messageProducer = session.createProducer(queue);
```

```
        TextMessage message = session.createTextMessage("Hello world!");
        messageProducer.send(message);
        session.commit();

        message = (TextMessage)messageConsumer.receive();
        session.commit();
        System.out.println(message.getText());

        connection.close();
        context.close();
    }
}
```

1. Loads the JNDI properties file, which specifies the connection factory, queues and topics. See Chapter 6, *JNDI Properties Format* for details.
2. Creates the JNDI initial context.
3. Looks up a JMS connection factory for Qpid.
4. Creates a JMS connection. Creating the JMS connections establishes the connection to the Broker.
5. Starts the connection, required for the consumption of messages.
6. Creates a transactional session.
7. Looks up a destination for the queue with JNDI name *myqueue*.
8. Creates a consumer that reads messages from the queue[1].
9. Creates a producer that sends messages to the queue.
10. Creates a new message of type *javax.jms.TextMessage*, publishes the message and commits the session.
11. Reads the next available message (awaiting indefinitely if necessary) and commits the session.
12. Closes the Connection. All sessions owned by the Connection along with their MessageConsumers and MessageProducers are automatically closed. The connection to the Broker is closed as this point.
13. Closes the JNDI context.

The contents of the `helloworld.properties` file are shown below.

Example 4.2. JMS Example - Point to Point Messaging - JNDI Properties

```
java.naming.factory.initial = org.apache.qpid.jndi.PropertiesFileInitialContextFac
connectionfactory.qpidConnectionFactory = amqp://guest:guest@clientid/?brokerlist=
queue.myqueue = queue1
```

1. Defines a connection factory from which Connections can be created. The syntax of a ConnectionURL is given in Chapter 7, *Connection URLs*.
2. Defines a queue for which MessageProducers and/or MessageConsumers send and receive messages. The format of these entries is described in Section 6.2, “Queue”.

4.2. Publish/subscribe example

In this second example, we illustrate publish/subscribe messaging. Again, we create a JNDI context using a properties file, use the context to lookup a connection factory, create and start a connection, create a

[1]Creating consumer will automatically create the queue on the Broker and bind it to an exchange. Specifically, in this case as the `queue.` form is used in the JNDI properties the effect will be to create a queue called `queue1` on the Broker, and create a binding between the `amq.direct` exchange and this queue using the queue's name. This process is described in detail in Section 5.6.1, “Consumers have Exchange/Queue Declaration and Binding Side Effect”

session, and lookup a destination (a topic) from the JNDI context. Then we create a producer and two durable subscribers , send a message with the producer. Both subscribers receive the same message.

Example 4.3. JMS Example - Publish/subscribe Messaging

```
import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;

import java.util.Properties;

public class StocksExample {

    public StocksExample() {
    }

    public static void main(String[] args) throws Exception {
      StocksExample stocks = new StocksExample();
      stocks.runTest();
    }

    private void runTest() throws Exception {
      Properties properties = new Properties();
      properties.load(this.getClass().getResourceAsStream("stocks.properties"));
      Context context = new InitialContext(properties);

      ConnectionFactory connectionFactory
          = (ConnectionFactory) context.lookup("qpidConnectionFactory");
      Connection connection = connectionFactory.createConnection();
      connection.start();

      Session session = connection.createSession(true, Session.SESSION_TRANSACTED)
      Topic priceTopic = (Topic) context.lookup("myprices");

      MessageConsumer subscriber1 = session.createDurableSubscriber(priceTopic, "s
      MessageConsumer subscriber2 = session.createDurableSubscriber(priceTopic, "s
      MessageProducer messageProducer = session.createProducer(priceTopic);

      Message message = session.createMessage();
      message.setStringProperty("instrument", "IBM");
      message.setIntProperty("price", 100);
      messageProducer.send(message);
      session.commit();

      message = subscriber1.receive(1000);
      session.commit();
      System.out.println("Subscriber 1 received : " + message);

      message = subscriber2.receive(1000);
      session.commit();
      System.out.println("Subscriber 2 received : " + message);

      session.unsubscribe("sub1");
```

```
      session.unsubscribe("sub2");
      connection.close();
      context.close();
    }
}
```

1 Looks up a destination for the topic with JNDI name myprices.

2 Creates two durable subscribers, `sub1` and `sub2`. Durable subscriptions retain messages for the client even when the client is disconnected, until the subscription is unsubscribed. Subscription 2 has a (commented out) message selector argument so you can conveniently experiement with the effect of those. [2]

3 Unsubscribes the two durable subscribers, permanently removing the knowledge of the subscriptions from the system. An application would normally *NOT* do this. The typical use-case for durable subsciption is one where the subscription exists over an extended period of time.

The contents of the `stocks.properties` file are shown below.

Example 4.4. JMS Example - Publish/subscribe Messaging - JNDI Properties

```
java.naming.factory.initial = org.apache.qpid.jndi.PropertiesFileInitialContextFac
connectionfactory.qpidConnectionFactory = amqp://guest:guest@clientid/?brokerlist=
topic.myprices = prices 1
```

1 Defines a topic for which MessageProducers and/or MessageConsumers send and receive messages. The format of this entry is described in Section 6.3, “Topic”.

[2] Each durable subscription is implemented as a queue on the Broker. See Section 5.6.2, “Topic Subscriptions” for details.

Chapter 5. Understanding the Qpid JMS client

5.1. Overview

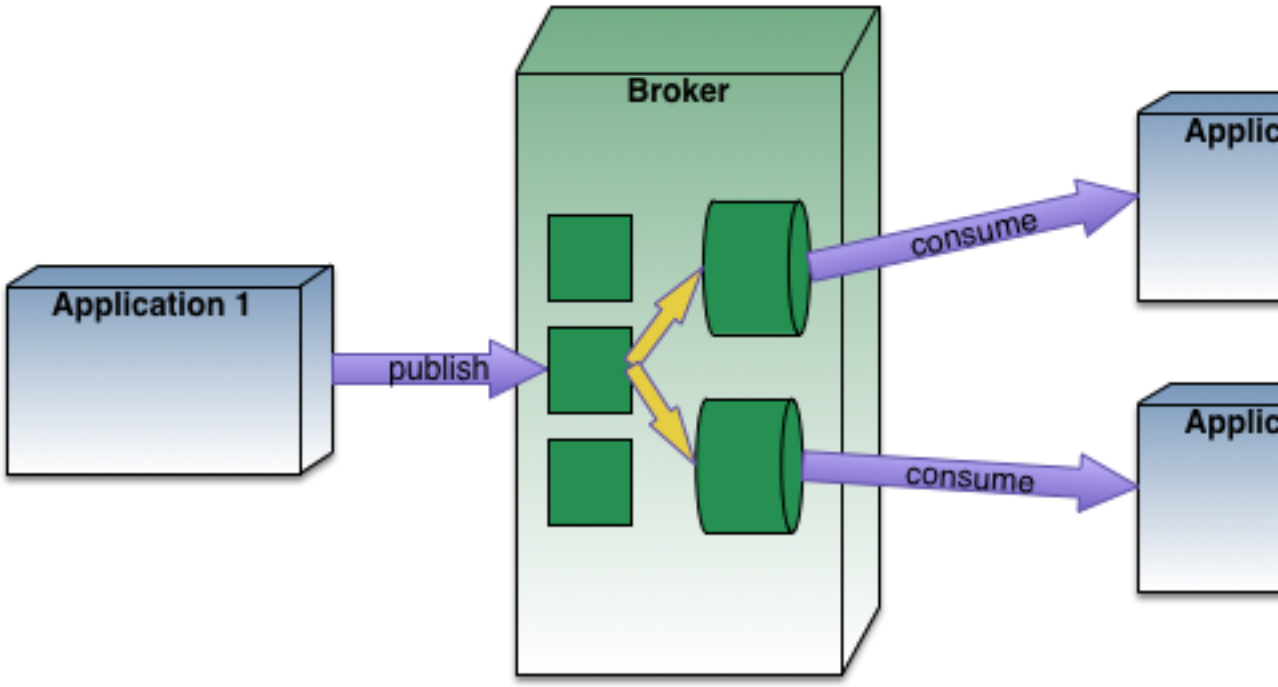
The Qpid JMS client provides a JMS 1.1 compliant implementation. As such, the primary source of documentation is the JMS specification [http://download.oracle.com/otndocs/jcp/7195-jms-1.1-fr-spec-oth-JSpec/] and the JMS javadocs [http://docs.oracle.com/javaee/6/api//javax/jms/package-summary.html]. This documentation assumes the reader has familiarity with these resources.

The remainder of this section describes how the Qpid JMS client behaves and the effect(s) making JMS method calls will have on the Broker.

There areas where the Qpid JMS client provides features beyond those required for JMS compliance. These are described in the sections that follow.

These sections are also used to bring out differences that may surprise those moving from JMS implementations provided by other vendors.

Figure 5.1. Architecture of a typical JMS application



5.2. ConnectionFactory

A ConnectionFactory [http://docs.oracle.com/javaee/6/api/javax/jms/ConnectionFactory.html] allows an application to create a Connection [http://docs.oracle.com/javaee/6/api/javax/jms/Connection.html].

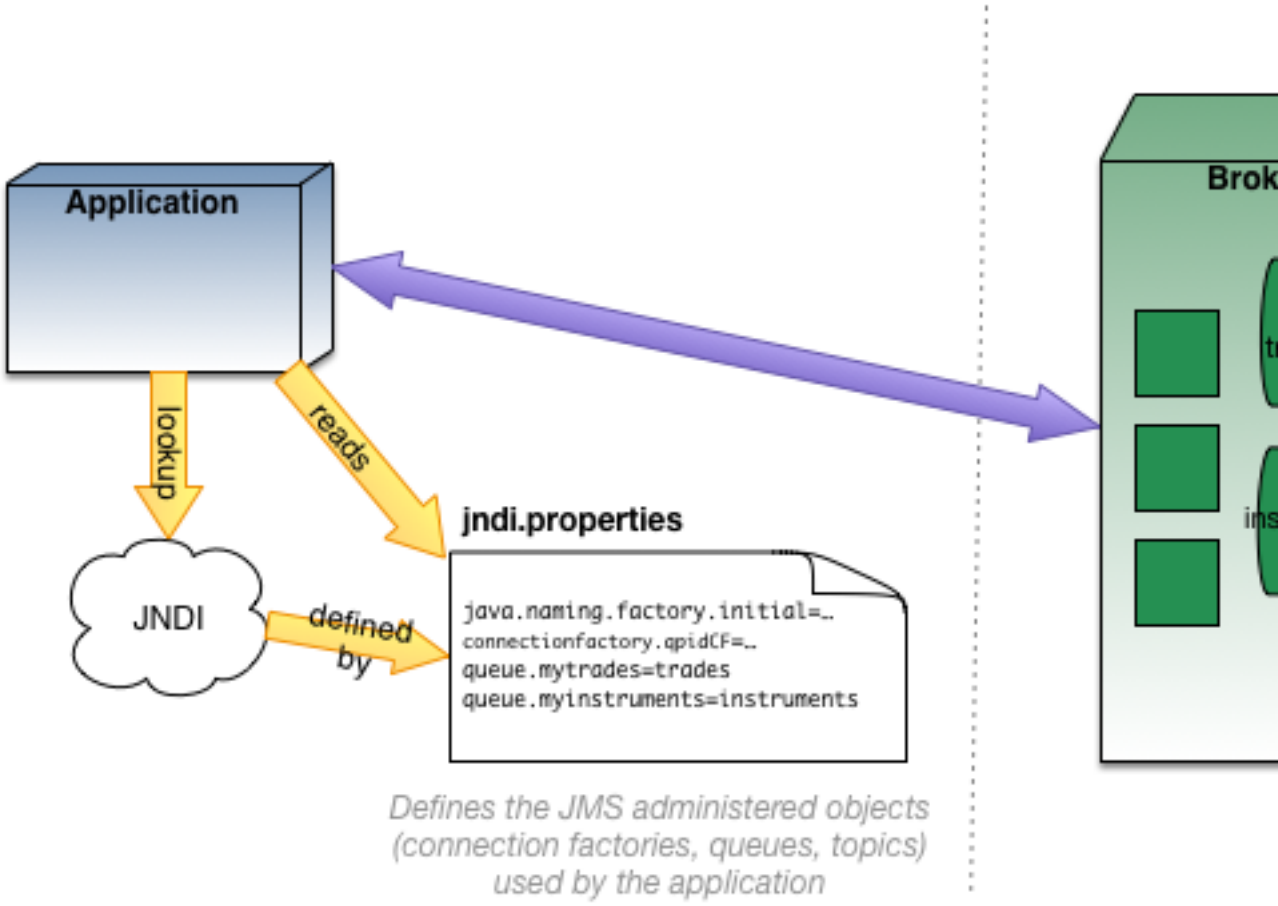
The application obtains the ConnectionFactory from an InitialContext [http://docs.oracle.com/javase/7/docs/api/javax/naming/InitialContext.html]. The InitialContext is itself obtained from an InitialContextFactory.

The Qpid JMS client provides a single implementation of the InitialContextFactory in class `org.apache.qpid.jndi.PropertiesFileInitialContextFactory`. This implementation is backed by a Properties [http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html] object which can of course be loaded from an external properties file, or created programatically.

The examples in the previous chapter illustrated the Java code required to create the InitialContext and an example properties file.

The Qpid JMS client also provides an alternate connection factory implementation providing a connection pool. This can be useful when utilsing frameworks such as Spring. Appendix D, *PooledConnectionFactory*.

Figure 5.2. JNDI overview



Note that the Qpid Broker does not present a JNDI interface to the application.

5.3. Connection

A Connection represents an open communication channel between application and Broker.

Connections are created from the ConnectionFactory [1].

Each connection utilises a single TCP/IP connection between the process of the application and the process of the Broker. The act of establishing a connection is therefore a relatively expensive operation. It is recommended that the same connection is used for a series of message interactions. Patterns utilising a connection per message should not be used.

The underlying TCP/IP connection remains open for the lifetime of the JMS connection. It is closed when the application calls Connection#close() [http://docs.oracle.com/javaee/6/api/javax/jms/Connection.html#close()], but it can also be closed if the connection is closed from the Broker side (via a Management operation or broker shutdown or running into conditions which AMQP specifications treats as errors and mandates closing the connection). The JMS connection will also be closed if the underlying TCP/IP connection is broken.

Qpid connections have failover and heartbeating capabilities. They support SSL and client-auth. These are described in the sub-sections that follow.

5.3.1. Failover

Qpid connections support a failover feature. This is the ability to automatically re-establish a failed connection, either to the same Broker, or the next Broker in the broker list.

This failover process is done in a manner that is mostly transparent to the application. After a successful failover, any existing Connection, Session, MessageConsumer and MessageProducer objects held by the application remain valid.

If a failover occurs during the scope of a JMS Transaction, any work performed by that transaction is lost. The application is made aware of this loss by way of the TransactionRolledBackException [http://docs.oracle.com/javaee/6/api/javax/jms/TransactionRolledBackException.html] from the Session#commit() [http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#commit] call. Applications utilising failover must be prepared to catch this exception and respond by either repeating the work of the transaction, or by propagating a rollback to the originating system.

If, after all retries are exhausted, failover has failed to reconnect the application, the Connection's ExceptionListener [http://docs.oracle.com/javaee/6/api/javax/jms/ExceptionListener.html] will receive a JMSException with a linked exception of AMQDisconnectedException [JMS-Client-0-8-Appendix-Exceptions-AMQDisconnectedException]. Any further use of the JMS objects (Connection, Session etc), will results in a IllegalStateException [http://docs.oracle.com/javaee/6/api/javax/jms/IllegalStateException.html].

Configure failover using the Connection URL. Here's an example Connection URL utilising failover between two brokers. Note the use of the broker options `retries` and `connectdelay` to control the number of connection attempts to each individual broker, and the delay between each connection attempt. Also note the use of the *failover option* `cyclecount` to control the number of times the failover mechanism will traverse the brokerlist.

Example 5.1. Connection URL configured for failover

```
amqp://username:password@clientid/test
            ?brokerlist='tcp://localhost:15672?retries='10'&connectdelay='1000';tc
            &failover='roundrobin?cyclecount='20''
```

[1] Constructors of the AMQConnection class must not be used.

For full details see Chapter 7, *Connection URLs*

Note

Note, that a single broker failover is enabled by default. If the failover behaviour is not desired it can be switched off by setting a failover option to `nofailover` as in the example below

Example 5.2. Connection URL configured with nofailover

```
amqp://username:password@clientid/test
            ?brokerlist='tcp://localhost:15672?failover='nofailover'
```

5.3.2. Heartbeating

Qpid connections support heartbeating. When enabled, the Qpid JMS client and Broker exchange a heartbeat during periods of inactivity. This allows both peers to discover if the TCP/IP connection becomes inoperable in a timely manner.

This feature is sometimes useful in applications that must traverse firewalls as the heartbeat prevents connections from being closed during periods when there is no application traffic.

It is also allows the both the JMS client and the Broker to confirm that the other is *minimally* responsive. (It does nothing however to determine the health of the higher level tiers of application, for this reason, applications may implement an application level heartbeat either in addition to, or instead of the heartbeat.

If the client ever fails to receive two consecutive heartbeats, the Connection will be automatically closed and the Connection's ExceptionListener [http://docs.oracle.com/javaee/6/api/javax/jms/ExceptionListener.html] will receive a JMSException with a linked exception of AMQDisconnectedException. Any further use of the JMS objects (Connection, Session etc), will results in a IllegalStateException [http://docs.oracle.com/javaee/6/api/javax/jms/IllegalStateException.html].

To enable heartbeating either use a Connection URL including the broker option `heartbeat`, or use the system property `qpid.heartbeat`.

Example 5.3. Connection URL configured for heartbeating

```
amqp://guest:guest@clientid/?brokerlist='localhost:5672?heartbeat='5''
```

5.3.3. SSL

The Qpid JMS client supports connections encrypted using Secure Socket Layer (SSL) and SSL-Client Authentication. SSL is configured using Connection URL. To use SSL, SSL must be be configured on the Broker.

Some example Connection URLs using SSL follow:

- Simple SSL when the Broker is secured by a certificate that is signed by a CA which is trusted by the JVM.

Example 5.4. Connection URL configured for SSL - CA trusted by JVM

```
amqp://guest:guest@clientid/?brokerlist='localhost:5671'&ssl='true'
```

- SSL when the Broker is secured by a certificate that is signed by a CA which is NOT trusted by the JVM (such as when a organisation is using a private CA, or self-signed certificates are in use). For this case, we use `trust_store` and `trust_store_password` to specify a path a truststore file (containing the certificate of the private-CA) and the truststore password.

Example 5.5. Connection URL configured for SSL - CA not trusted by JVM

```
amqp://guest:guest@clientid/?brokerlist='localhost:5671?trust_store='/path/to/acr
```

- SSL with SSL client-auth. For this case, we use `key_store` and `key_store_password` to specify a path a keystore file (containing the certificate of the client) and the keystore password.

Example 5.6. Connection URL configured for SSL - SSL client-auth

```
amqp://guest:guest@clientid/?brokerlist='localhost:5671?key_store='/path/to/app1_
```

Alternatively we can use `client_cert_path` and `client_cert_priv_key_ath` to specify a path to a certificate file (in PEM or DER format) and the private key information (again in either PEM or DER format) respectively.

Example 5.7. Connection URL configured for SSL - SSL client-auth (2)

```
amqp://guest:guest@clientid/?brokerlist='localhost:5671?client_cert_path='/path/t
```

5.3.4. Message Compression

The client has the ability to transparently compress message payloads on outgoing messages and decompress them on incoming messages. In some environments and with some payloads this feature might offer performance improvements by reducing the number of bytes transmitted over the connection.

In order to make use of message compression, the Broker must enable the feature too, otherwise the compression options will be ignored.

To enable message compression on the client use the connection url property `compressMessages` (or JVM wide using the system property `qpid.connection_compress_messages`)

It is also possible to control the threshold at which the client will begin to compress message payloads. See connection url property `messageCompressionThresholdSize` (or JVM wide using the system property `qpid.message_compression_threshold_size`)

Note

The Broker, where necessary, takes care of compressing/decompressing messages of the fly so that clients using message compression can exchange messages with clients not supporting message compression transparently, without application intervention.

5.4. Session

A Session object is a single-threaded context for producing and consuming messages.

Session objects are created from the Connection. Whilst Session objects are relatively lightweight, patterns utilising a single Session per message are not recommended.

The number of sessions open per connection at any one time is limited. This value is negotiated when the connection is made. It defaults to 256.

Qpid JMS Sessions have the ability to prefetch messages to improve consumer performance. This feature is described next.

5.4.1. Prefetch

Prefetch specifies how many messages the client will optimistically cache for delivery to a consumer. This is a useful parameter to tune that can improve the throughput of an application. The prefetch buffer is scoped per *Session*.

The size of the prefetch buffer can be tuned per Connection using the connection url option `maxprefetch` (or JVM wide using the system property `max_prefetch`). By default, prefetch defaults to 500.

There are situations when you may wish to consider reducing the size of prefetch:

1. When using a Competing Consumers [http://www.eaipatterns.com/CompetingConsumers.html] pattern, prefetch can give the appearance of unequal division of work. This will be apparent on startup when the queue has messages. The first consumer started will cache prefetch size number of messages, possibly leaving the other consumers with no initial work.

2. When using special queue types (such as LVQs, Sorted Queue and Priority Queues). For these queue types the special delivery rules apply whilst the message resides on the Broker. As soon as the message is sent to the client it delivery order is then fixed. For example, if using a priority queue, and a prefetch of 100, and 100 messages arrive with priority 2, the broker will send these to the client. If then a new message arrives with priority 1, the broker cannot leap frog messages of the lower priority. The priority 1 message will be delivered at the front of the next batch.

3. When message size is large and you do not wish the memory footprint of the application to grow (or suffer an OutOfMemoryError).

Finally, if using multiple MessageConsumers on a single Session, keep in mind that unless you keep polling *all* consumers, it is possible for some traffic patterns to result in consumer starvation and an application level deadlock. For example, if prefetch is 100, and 100 hundred messages arrive suitable for consumer A, those messages will be prefetched by the session, entirely filling the prefetch buffer. Now if the application performs a blocking MessageConsumer#receive() [http://docs.oracle.com/javaee/6/api/javax/jms/MessageConsumer.html#receive()] for Consumer B on the same Session, the application will hang indefinitely as even if messages suitable for B arrive at the Broker. Those messages can never be sent to the Session as no space is available in prefetch.

Note

Please note, when the acknowledgement mode *Session#SESSION_TRANSACTED* or *Session#CLIENT_ACKNOWLEDGE* is set on a consuming session, the prefetched messages are released from the prefetch buffer on transaction commit/rollback (in case of acknowledgement mode *Session#SESSION_TRANSACTED*) or acknowledgement of the messages receipt (in

case of acknowledgement mode *Session#CLIENT_ACKNOWLEDGE*). If the consuming application does not commit/rollback the receiving transaction (for example, due to mistakes in application exception handling logic), the prefetched messages continue to remain in the prefetch buffer preventing the delivery of the following messages. As result, the application might stop the receiving of the messages until the transaction is committed/rolled back (for *Session#SESSION_TRANSACTED*) or received messages are acknowledged (for *Session#CLIENT_ACKNOWLEDGE*).

Settings maxprefetch to 0 (either globally via JVM system property `max_prefetch` or on a connection level as a connection option `maxprefetch`) switches off the pre-fetching functionality. With maxprefetch=0 messages are fetched one by one without caching on the client.

Note

Setting maxprefetch to 0 is recommended in Spring-JMS based applications whenever *DefaultMassgeListenerContainer* is configured with a *CachingConnectionFactory* that has *cacheLevel* set to either *CACHE_CONSUMER* or *CACHE_SESSION*. In these configurations the Qpid JMS *Session* objects remain open in Spring's dynamically scaled pools. If maxprefetch is not 0, any prefetched messages held by the *Session* and any new ones subsequently sent to it (in the background until prefetch is reached) will be effectively by 'stuck' (unavailable to the application) until Spring decides to utilise the cached Session again. This can give the impression that message delivery has stopped even though messages remain of the queue. Setting maxprefetch to 0 prevents this problem from occurring.

If using maxprefetch > 0 *SingleConnectionFactory* must be used. SingleConnectionFactory does not have the same session/consumer caching behaviour so does not exhibit the same problem.

5.4.2. TemporaryQueues

Qpid implements JMS temporary queues as AMQP auto-delete queues. The life cycle of these queues deviates from the JMS specification.

AMQP auto-delete queues are deleted either when the *last* Consumer closes, or the Connection is closed. If no Consumer is ever attached to the queue, the queue will remain until the Connection is closed.

This deviation has no practical impact on the implementation of the request/reply messaging pattern [http://www.eaipatterns.com/RequestReply.html] utilising a per-request temporary reply queue. The reply to queue is deleted as the application closes the Consumer awaiting the response.

Temporary queues are exposed to Management in the same way as normal queues. Temporary queue names take the form string `TempQueue` followed by a random UUID.

Note that TemporaryQueue#delete() [http://docs.oracle.com/javaee/6/api/javax/jms/TemporaryQueue.html#delete()] merely marks the queue as deleted on within the JMS client (and prevents further use of the queue from the application), however, the Queue will remain on the Broker until the Consumer (or Connection) is closed.

5.4.3. CreateQueue

In the Qpid JMS client, Session#createQueue() [http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#createQueue(java.lang.String)] accepts either a queue name, or a Binding URL. If only name is specified the destination will be resolved into binding URL: direct://amq.direct//<queue name>?routingkey='<queue name>'&durable='true'.

Calling Session#createQueue() has no effect on the Broker.

Reiterating the advice from the JMS javadoc, it is suggested that this method is not generally used. Instead, application should lookup Destinations declared within JNDI.

5.4.4. CreateTopic

In the Qpid JMS client, Session#createTopic() [http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#createTopic(java.lang.String)] accepts either a topic name, or a Binding URL. If only name is specified the destination will be resolved into binding URL: topic://amq.topic//<topic name>?routingkey='<topic name>'.

Calling Session#createTopic() has no effect on the Broker.

Reiterating the advice from the JMS javadoc, it is suggested that this method is not generally used. Instead, application should lookup Destinations declared within JNDI.

5.5. MessageProducer

A MessageProducer sends a message an *Exchange*. It is the Exchange (within the Broker) that routes the message to zero or more queue(s). Routing is performed according to rules expressed as *bindings* between the exchange and queues and a *routing key* included with each message.

To understand how this mechanism is used to deliver messages to queues and topics, see Exchanges [../../AMQP-Messaging-Broker-Java-Book/html/Java-Broker-Concepts-Exchanges.html] within the Java Broker book.

It is important to understand that when synchronous publish is not exlicitly enabled, MessageProducer#send() [http://docs.oracle.com/javaee/6/api/javax/jms/MessageProducer.html#send(javax.jms.Message)] is *asynchronous* in nature. When #send() returns to the application, the application cannot be certain if the Broker has received the message. The Qpid JMS client may not have yet started to send the message, the message could residing in a TCP/IP buffer, or the messages could be in some intermediate buffer within the Broker. If the application requires certainty the message has been received by the Broker, a transactional session [http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#SESSION_TRANSACTED] *must* be used, or synchronous publishing must be enabled using either the system property or the connection URL option.

Qpid JMS MessageProducers have a number of features above that required by JMS. These are described in the sub-sections that follow.

5.5.1. Mandatory Messages

With this feature, publishing a message with a routing key for which no binding exists on the exchange will result in the message being returned to the publisher's connection.

The Message is returned to the application in an asynchronous fashion via the Connection's ExceptionListener [http://docs.oracle.com/javaee/6/api/javax/jms/ExceptionListener.html]. When a message is returned, it will be invoked with a JMSException whose linked exception is an AMQNoRouteException [JMS-Client-0-8-Appendix-Exceptions-AMQNoRouteException]. The returned message is available to the application by calling AMQNoRouteException#getUndeliveredMessage(). The ExceptionListener will be invoked exactly once for each returned message.

If synchronous publishing has been enabled, and a mandatory message is returned, the MessageProducer#send() [http://docs.oracle.com/javaee/6/api/javax/jms/MessageProducer.html#send(javax.jms.Message)] method will throw a JMSException.

The mandatory message feature is turned *on* by default for Queue destinations and *off* for Topic destinations. This can be overridden using system properties `qpid.default_mandatory` and `qpid.default_mandatory_topic` for Queues and Topics respectively.

Note

If this the mandatory flag is not set, the Broker will treat the messages as unroutable [../../AMQP-Messaging-Broker-Java-Book/html/Java-Broker-Concepts-Exchanges.html#Java-Broker-Concepts-Exchanges-UnroutableMessage].

5.5.2. Close When No Route

With this feature, if a mandatory message is published with a routing key for which no binding exists on the exchange the Broker will close the connection. This client feature requires support for the corresponding feature by the Broker.

To enable or disable from the client, use the Connection URL option `closeWhenNoRoute`.

See Closing client connections on unroutable mandatory messages [../../AMQP-Messaging-Broker-Java-Book/html/Java-Broker-Close-Connection-When-No-Route.html] within the Java Broker book for full details of the functioning of this feature.

5.5.3. Immediate Messages

This feature is defined in AMQP specifications [http://www.amqp.org].

When this feature is enabled, when publishing a message the Broker ensures that a Consumer is attached to queue. If there is no Consumer attached to the queue, the message is returned to the publisher's connection. The Message is returned to the application in an asynchronous fashion using the Connection's ExceptionListener [http://docs.oracle.com/javaee/6/api/javax/jms/ExceptionListener.html].

The ExceptionListener will be invoked with a JMSException whose linked exception is an AMQNoConsumersException [JMS-Client-0-8-Appendix-Exceptions-AMQNoConsumersException]. The returned message is available to the application by calling AMQNoConsumersException#getUndeliveredMessage(). The ExceptionListener will be invoked exactly once for each returned message.

If synchronous publishing has been enabled, and an immediate message is returned, the MessageProducer#send() [http://docs.oracle.com/javaee/6/api/javax/jms/MessageProducer.html#send(javax.jms.Message)] method will throw a JMSException.

The immediate message feature is turned *off* by default. It can be enabled with system property `qpid.default_immediate`.

5.5.4. Flow Control

With this feature, if a message is sent to a queue that is overflow, the producer's session is blocked until the queue becomes underfull, or a timeout expires. This client feature requires support for the corresponding feature by the Broker.

To control the timeout use System property `qpid.flow_control_wait_failure`. To control the frequency with which warnings are logged whilst a Session is blocked, use System property `qpid.flow_control_wait_notify_period`

See Producer Flow Control [../../AMQP-Messaging-Broker-Java-Book/html/Java-Broker-Runtime-Disk-Space-Management.html#Qpid-Producer-Flow-Control] within the Java Broker book for full details of the functioning of this feature.

5.6. MessageConsumer

A MessageConsumer receives messages from a Queue or Topic.

MessageConsumer objects are created from the Session.

Qpid JMS MessageConsumers have a number of features above that required by JMS. These are described in the sub-sections that follow.

5.6.1. Consumers have Exchange/Queue Declaration and Binding Side Effect

By default, calling Session#createConsumer() [http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#createConsumer(javax.jms.Destination)] will cause:

1. If the exchange does not exist on the Broker, it will be created. The exchange is specified by the Binding URL associated with the Destination.

2. If the queue does not exist on the Broker, it will be created. The queue is specified by the Binding URL associated with the Destination.

3. If there is no binding between the exchange and queue, a binding will be created using the routingkey as a bindingkey. The exchange, queue and routing key are specified by the Binding URL associated with the Destination.

The exchange declare, queue declare and bind side effects can be suppressed using system properties `qpid.declare_exchanges`, `qpid.declare_queues` and `qpid.bind_queues`.

5.6.2. Topic Subscriptions

The Qpid JMS client implements each subscription to a Topic as separate queue on the Broker. From the perspective of the JMS application this implementational detail is irrelevant: the application never needs to directly address these queues. However, these details are important when considering Management and Operational concerns.

Durable topic subscriptions use a *durable* and *exclusive* queue named as follows:

```
clientid: + subscriptionId
```

where `subscriptionId` is that passed to the Session#createDurableSubscriber(javax.jms.Topic,java.lang.String) [http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#createDurableSubscriber(javax.jms.Topic,%20java.lang.String)]

Calling Session#unsubscribe(java.lang.String) [http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#unsubscribe(java.lang.String)] deletes the underlying queue.

Non-durable topic subscriptions use a *non-durable*, *exclusive* and *auto-delete* queue named as follows:

```
tmp + _ + ip + _ + port + _ + sequence
```

where `ip` is the ip address of the client with dots replaced by underscores, `port` is the ephemeral port number assigned to the client's connection, and `sequence` is a sequence number.

Closing the consumer (or closing the connection) will delete the underlying queue.

5.6.3. Maximum Delivery Count

With this feature, the Broker keeps track of a number of times a message has been delivered to a consumer. If the count ever exceeds a threshold value, the Broker moves the message to a dead letter queue (DLQ). This is used to prevent poison messages preventing a system's operation. This client feature requires support for the corresponding feature by the Broker.

When using this feature, the application must either set system property qpid.reject.behaviour or the Binding URL option `rejectbehaviour` to the value `server`.

See Handling Undeliverable Messages [../../AMQP-Messaging-Broker-Java-Book/html/Java-Broker-Runtime-Handling-Undeliverable-Messages.html#Java-Broker-Runtime-Handling-Undeliverable-Messages-Maximum-Delivery-Count] within the Java Broker book for full details of the functioning of this feature.

Note

The optional JMS message header `JMSXDeliveryCount` is *not* supported.

5.7. Destinations

A Destination is either a Queue or Topic. In the Qpid JMS client a Destination encapsulates a Binding URL. In simple terms, the Binding URL comprises of an exchange, queue and a routing key. Binding URLs are described fully by Chapter 8, *Binding URL*.

In many cases, applications do not need to deal directly with Binding URLs, instead they can refer to JMS administered objects declared in the JNDI properties file with the `queue.` and `topic.` prefix to create Queues and Topics objects respectively.

Chapter 6. JNDI Properties Format

The Qpid JMS Client comes with own JNDI context factory `org.apache.qpid.jndi.PropertiesFileInitialContextFactory` which utilises a Java properties file for declaring the JMS administered objects: connection factories, queues, topics and destinations. It uses the following syntax:

```
connectionfactory.<jndi name>=<connection url>
queue.<jndi name>=<queue name>
topic.<jndi name>=<topic name>
destination.<jndi name>=<binding url>
```

An arbitrary number of connection factories, queues, topics, queues or destinations or can be declared in the JNDI properties file. Each JNDI name must be unique.

The application looks up the objects via an InitialContext. This lookup and an example JNDI properties file is provided in Chapter 4, *Examples*

We now consider each JMS administered object type in turn.

6.1. ConnectionFactory

`connectionfactory.`*name* declares a ConnectionFactory [http://docs.oracle.com/javaee/6/api/javax/jms/ConnectionFactory.html] with the given JNDI name. The value must be a legal Connection URL.

See Chapter 7, *Connection URLs* for format of the URL and its permitted options.

6.2. Queue

`queue.`*name* declares a Queue [http://docs.oracle.com/javaee/6/api/javax/jms/Queue.html] with the given JNDI name. The value is simple queue name. This is the name of the queue as known by the Broker.

The `queue.` form is a short hand for declaring a destination:

```
destination.name=direct://amq.direct//<queue name>?routingkey='<queue name>'&durab
```

6.3. Topic

`topic.`*name* declares a Topic [http://docs.oracle.com/javaee/6/api/javax/jms/Topic.html] with the given JNDI name. The value is topic name. This topic name is used on the Broker as a binding key between the `amq.topic` exchange and the queue corresponding to the topic subscriber.

The `topic.` form is a short hand for declaring a destination:

```
destination.name=topic://amq.topic/<topic name>/?routingkey=<topic name>
```

6.4. Destination

`destination.`*name* declares either a Queue [http://docs.oracle.com/javaee/6/api/javax/jms/Queue.html] or Topic [http://docs.oracle.com/javaee/6/api/javax/jms/Topic.html] (depending on the class) with the given JNDI name. The value must be a Binding URL.

See Chapter 8, *Binding URL* for format of the URL and its permitted options.

Chapter 7. Connection URLs

In JNDI properties, a Connection URL specifies options for a connection. The format for a Connection URL is:

```
amqp://[<user>:<pass>@][<clientid>]/[<virtualhost>][?<option>='<value>'[&<option>=
```

For instance, the following Connection URL specifies a user name, a password, a client ID, a virtual host ("test"), a broker list with a single broker: a TCP host with the host name “localhost” using port 5672:

```
amqp://username:password@clientid/test?brokerlist='tcp://localhost:5672'
```

Connection option quoting

Take care with the quoting surrounding option values. Each option value *must* be surrounded with single quotes (').

The Connection URL supports the following options:

Table 7.1. Connection URL Options

Option	Type	Description
brokerlist	see below	List of one or more broker addresses.
maxprefetch	integer	The maximum number of pre-fetched messages per Session. If not specified, default value of 500 is used. Note: You can also set the default per-session prefetch value on a client-wide basis by configuring the client using Java system properties.
sync_publish	String	If the value is 'all' the client library waits for confirmation before returning from a send(), and if the send is unsuccessful the send() will throw a JMSException. (Note this option requires an extension to the AMQP protocol and will only work against a broker of the 0.32 release or later.)
sync_ack	Boolean	A sync command is sent after every acknowledgement to guarantee that it has been received.
use_legacy_map_msg_format	Boolean	If you are using JMS Map messages and deploying a new client with any JMS client older than 0.8 release, you must set this to true to ensure the older clients can understand the map message encoding.
failover	{'singlebroker' | 'roundrobin' , | 'nofailover' | '<class>'}	This option controls failover behaviour. The method `singlebroker` uses only the first broker in the list, `roundrobin`

Option	Type	Description
		will try each broker given in the broker list until a connection is established, `nofailover` disables all retry and failover logic. Any other value is interpreted as a classname which must implement the `org.apache.qpid.jms.failover.FailoverM` interface. The broker list options `retries` and `connectdelay` (described below) determine the number of times a connection to a broker will be retried and the length of time to wait between successive connection attempts before moving on to the next broker in the list. The failover option `cyclecount` controls the number of times to loop through the list of available brokers before finally giving up. Defaults to `roundrobin` if the brokerlist contains multiple brokers, or `singlebroker` otherwise.
closeWhenNoRoute	boolean	See Section 5.5.2, “Close When No Route”.
ssl	boolean	If `ssl='true'`, use SSL for all broker connections. Overrides any per-broker settings in the brokerlist (see below) entries. If not specified, the brokerlist entry for each given broker is used to determine whether SSL is used. Introduced in version 0.22.
compressMessages	Boolean	Controls whether the client will compress messages before they they are sent.
messageCompressionThresholdSize	Integer	The payload size beyond which the client will start to compress message payloads.

Broker lists are specified using a URL in this format:

```
brokerlist='<transport>://<host>[:<port>][?<param>='<value>'[&<param>='<value>']*]
```

For instance, this is a typical broker list:

```
brokerlist='tcp://localhost:5672'
```

A broker list can contain more than one broker address separated by semicolons (;). If so, the connection is made to the first broker in the list that is available.

Example 7.1. Broker Lists

A broker list can specify properties to be used when connecting to the broker. This broker list specifies options for configuring heartbeating

```
amqp://guest:guest@test/test?brokerlist='tcp://ip1:5672?heartbeat='5''
```

This broker list specifies some SSL options

```
amqp://guest:guest@test/test?brokerlist='tcp://ip1:5672?ssl='true'&ssl_cert_alias=
```

This broker list specifies two brokers using the connectdelay and retries broker options. It also illustrates the failover connection URL property.

```
amqp://guest:guest@/test?failover='roundrobin?cyclecount='2''
   &brokerlist='tcp://ip1:5672?retries='5'&connectdelay='2000';tcp://ip2:5672?retr
```

Broker option quoting

Take care with the quoting surrounding broker option values. Each broker option value *must* be surrounded with their own single quotes ('). This is in addition to the quotes surround the connection option value.

The following broker list options are supported.

Table 7.2. Broker List Options

Option	Type	Description
heartbeat	Long	Frequency of heartbeat messages (in seconds). A value of 0 disables heartbeating. For compatibility with old client configuration, option `idle_timeout` (in milliseconds) is also supported.
ssl	Boolean	If `ssl='true'`, the JMS client will encrypt the connection to this broker using SSL. This can also be set/overridden for all brokers using the Connection URL option `ssl`.
trust_store	String	Path to trust store. Used when using SSL and the Broker's certificate is signed by a private-CA (or a self-signed certificate),
trust_store_password	String	Trust store password. Password used to open the trust store.
trusted_certs_path	String	Path to a file containing trusted peer certificates(in PEM or DER format). Used when supplying the trust information for TLS client auth

Option	Type	Description
		using PEM/DER files rather than a Java KeyStore.
key_store	String	Path to key store . Used when using SSL and the client must authenticate using client-auth. If the store contains more than one certificate, `ssl_cert_alias` must be used to identify the certificate that the client must present to the Broker.
key_store_password	String	Key store password. Password used to open the key store.
client_cert_path	String	Path to the client certificate file (in PEM or DER format). Used as an alternative to using a Java KeyStore to hold key information for TLS client auth. When used, the `client_cert_priv_key_path` must also be supplied.
client_cert_priv_key_path	String	Path to the client certificate private key file (in PEM or DER format). Used when supplying the key information for TLS client auth using PEM/DER files rather than a Java KeyStore.
client_cert_intermediary_cert_path	String	Path to a file containing any intermediary certificates (in PEM or DER format). Used when supplying the key information for TLS client auth using PEM/DER files rather than a Java KeyStore. Only required where intermediary certificates are required in the certificate chain.
ssl_cert_alias	String	If multiple certificates are present in the keystore, the alias will be used to extract the correct certificate.
ssl_verify_hostname	Boolean	This option is used for turning on/off hostname verification when using SSL. It is set to 'true' by default. You can disable verification by setting it to 'false': `ssl_verify_hostname='false'`.
retries	Integer	The number of times to retry connection to each broker in the broker list. Defaults to 1.
connectdelay	integer	Length of time (in milliseconds) to wait before attempting to reconnect. Defaults to 0.

Option	Type	Description
connecttimeout	integer	Length of time (in milliseconds) to wait for the socket connection to succeed. A value of 0 represents an infinite timeout, i.e. the connection attempt will block until established or an error occurs. Defaults to 30000.
tcp_nodelay	Boolean	If `tcp_nodelay='true'`, TCP packet batching is disabled. Defaults to true since Qpid 0.14.

Chapter 8. Binding URL

The *Binding URL* syntax for addressing[1]. It allows the specification of the bindings between a queue and an exchange, queue and exchange creation arguments and some ancillary options.

The format for a *Binding URL* is provided below

```
<Exchange Class>://<Exchange Name>/[<Destination>]/[<Queue>][?<option>='<value>'[&
```

where

- *Exchange Class*, specifies the type of the exchange, for example, *direct*,*topic*,*fanout*, etc.
- *Exchange Name*, specifies the name of the exchange, for example, *amq.direct*,*amq.topic*, etc.
- *Destination*, is an optional part of *Binding URL*. It can be used to specify a routing key with the non direct exchanges if an option *routingkey* is not specified. If both *Destination* and option *routingkey* are specified, then option *routingkey* has precedence.
- *Queue*, is an optional part of *Binding URL* to specify a queue name for JMS queue destination. It is ignored in JMS topic destinations. Queue names may consist of any mixture of digits, letters, and underscores
- *Options*, key-value pairs separated by '=' character specifying queue and exchange creation arguments, routing key, client behaviour, etc.

Binding URL option quoting

Take care with the quoting surrounding option values. Each option value *must* be surrounded with single quotes (').

The following *Binding URL* options are currently defined:

Table 8.1. Binding URL options

Option	Type	Description
durable	boolean	Queue durability flag. If it is set to *true*, a durable queue is requested to create. The durable queue should be stored on the Broker and remained there after Broker restarts until it is explicitly deleted. This option has no meaning for JMS topic destinations, as by nature a topic destination only exists when a subscriber is connected. If durability is required for topic destinations, the durable subscription should be created.

[1]The client also supports the ADDR format. This is documented in Programming in Apache Qpid [../../Programming-In-Apache-Qpid/html/].

Option	Type	Description
exclusive	boolean	Queue exclusivity flag. The client cannot use a queue that was declared as exclusive by another still-open connection.
autodelete	boolean	Queue auto-deletion flag. If it is set to *true* on queue creation, the queue is deleted if there are no remaining subscribers.
exchangeautodelete	boolean	Exchange auto-deletion flag.
exchangedurable	boolean	Exchange durability flag. If it is set to *true* when creating a new exchange, the exchange will be marked as durable. Durable exchanges should remain active after Broker restarts. Non-durable exchanges are deleted on following Broker restart.
routingkey	string	Defines the value of the binding key to bind a queue to the exchange. It is always required to specify for JMS topic destinations. If routing key option is not set in *Binding URL* and direct exchange class is specified, the queue name is used as a routing key. *MessagePublisher* uses routing key to publish messages onto exchange.
browse	boolean	If set to *true* on a destination for a message consumer, such consumer can only read messages on the queue but cannot consume them. The consumer behaves like a queue browser in this case.
rejectbehaviour	string	Defines the reject behaviour for the re-delivered messages. If set to 'SERVER' the client delegates the requeue/DLQ decision to the server. If this option is not specified, the messages won't be moved to the DLQ (or dropped) when delivery count exceeds the maximum.

8.1. Binding URL Examples

8.1.1. Binding URLs for declaring of JMS Queues

The Qpid client Binding URLs for JMS queue destinations can be declared using direct exchange (Mostly it is a pre-defined exchange with a name "amq.direct". Also, custom direct exchanges can be used.):

```
direct://amq.direct//<Queue Name>
```

The Binding URLs for destinations created with calls to *Session.createQueue(String)* can be expressed as

```
direct://amq.direct//<Queue Name>?durable='true'
```

The durability flag is set to *true* in such destinations.

Example 8.1. Binding URL examples for JMS queues

```
direct://amq.direct//myNonDurableQueue
direct://amq.direct//myDurableQueue?durable='true'
direct://amq.direct//myAnotherQueue?durable='true'&routingkey='myqueue'
direct://amq.direct//myQueue?durable='true'&routingkey='myqueue'&rejectbehaviour='
direct://custom.direct//yetAnotherQueue
```

8.1.2. Binding URLs for declaring of JMS Topics

The Binding URLs for JMS queue destinations can be declared using topic exchange (A pre-defined exchange having name "amq.topic" is used mainly. However, custom topic exchanges can be used as well):

```
topic://amq.topic//<Queue name>?routingkey='<Topic Name>'&exclusive='true'&autodel
```

The Binding URLs for a topic destination created with calls to *Session.createTopic("hello")* is provided below:

Example 8.2. Binding URL examples for JMS topics

```
topic://amq.topic/hello/tmp_127_0_0_1_36973_1?routingkey='hello'&exclusive='true'&
```

8.1.3. Wildcard characters in routing keys for topic destinations

AMQP exchanges of class *topic* can route messages to the queues using special matches containing wildcard characters (a "#" matches one or more words, a "*" matches a single word). The routing keys words are separated with a "." delimiter to distinguish words for matching. Thus, if a consumer application specifies a routing key in the destination like "usa.#", it should receive all the messages matching to that routing key. For example, "usa.boston", "usa.new-york", etc.

The examples of the *Binding URLs* having routing keys with wildcards characters are provided below:

```
topic://amq.topic?routingkey='stocks.#'
topic://amq.topic?routingkey='stocks.*.ibm'
topic://amq.topic?routingkey='stocks.nyse.ibm'
```

8.1.4. More Examples

Table 8.2. Binding URL examples

Binding URL	Description
fanout://amq.fanout//myQueue	Binding URL binding queue "myQueue" to predefined "amq.fanout" exchange of class "fanout"
topic://custom.topic//anotherQueue?routingkey='aq'	Binding URL binding queue "anotherQueue" to the exchange with name "custom.topic" of class "topic" using binding key "aq".

Chapter 9. System Properties

The following system properties affect the behaviour of the Qpid JMS client. System properties are global in nature so affect all Qpid interactions made from within the same JVM. For many options, there are equivalent Connection URL options allowing the option to be controlled at the level of the Connection.

Table 9.1. System Properties Affecting Connection Behaviour

Property Name	Type	Default Value	Description
qpid.amqp.version	string	0-10	Sets the AMQP version to be used - currently supports one of {0-8,0-9,0-91,0-10}. The client will begin negotiation at the specified version and only negotiate downwards if the Broker does not support the specified version.
qpid.heartbeat	int	Defaults to the heartbeat value suggested by the Broker, if any.	Frequency of heartbeat messages (in seconds). A value of 0 disables heartbeating. Two consecutive missed heartbeats will result in the connection timing out. This can also be set per connection using the Connection URL options. For compatibility with old client configuration, the synonym `amqj.heartbeat.delay` is supported.
ignore_setclientID	boolean	false	If a client ID is specified in the connection URL it's used or else an ID is generated. If an ID is specified after it's been set Qpid will throw an exception. Setting this property to 'true' will disable that check and allow you to set a client ID of your choice later on.
qpid.connection_ssl_verify_hostname	boolean	true	This property is used to turn on/off broker host name verification on SSL negotiation if SSL transport

Property Name	Type	Default Value	Description
			is used. It is set to 'true' by default. Setting this property to 'false' will disable that check and allow you to ignore host name errors.
qpid.connection_compress_messages	boolean	false	Controls whether the client will compress messages before they they are sent.
qpid.message_compression_threshold_size	integer	102400	The payload size beyond which the client will start to compress message payloads.

Table 9.2. Config Options For Session Behaviour

Property Name	Type	Default Value	Description
qpid.dest_syntax	String	ADDR	Addressing syntax: ADDR (Address format) or BURL (Binding URL)
max_prefetch	int	500	Maximum number of pre-fetched messages per Session. This can also be defaulted for sessions created on a particular connection using the Connection URL options.
qpid.use_legacy_map_message	boolean	false	If set will use the old map message encoding. By default the Map messages are encoded using the 0-10 map encoding. This can also be set per connection using the Connection URL options.
qpid.jms.daemon.dispatcher	boolean	false	Controls whether the Session dispatcher thread is a daemon thread or not. If this system property is set to true then the Session dispatcher threads will be created as daemon threads. This setting is introduced in version 0.16.

Table 9.3. Config Options For Consumer Behaviour

Property Name	Type	Default Value	Description
qpid.declare_exchanges	Boolean	true	If true, creating a consumer will also declare the exchange on the Broker (specified within the Binding URL associated with the Destination), creating it if is does not already exist.
qpid.declare_queues	Boolean	true	If true, creating a consumer will also declare the queue on the Broker (specified within the Binding URL associated with the Destination), creating it if is does not already exist.
qpid.bind_queues	Boolean	true	If true, creating a consumer will also bind the queue to the to the exchange using the routing key as a binding key. The exchange name, queue name and routing key are taken from the Binding URL associated with the Destination.
qpid.reject.behaviour	String	NORMAL	Used with the maximum delivery count feature. See Section 5.6.3, “Maximum Delivery Count” for details.

Table 9.4. Config Options For Producer Behaviour

Property Name	Type	Default Value	Description
qpid.default_mandatory	Boolean	True	If true, messages sent to Queue destinations for which cannot be routed to at least one queue on the Broker, will be returned to the application. See Section 5.5.1, “Mandatory Messages” for more details.
qpid.default_mandatory_topic	Boolean	False	If true, messages sent to Topic destinations for which cannot be routed to at least one queue on the Broker, will be returned to the application. See Section 5.5.1, “Mandatory Messages” for more details..

Property Name	Type	Default Value	Description
qpid.default_immediate	Boolean	False	If true, the message will be returned to the application unless the Broker is able to route the message to at least one queue with a consumer attached. See Section 5.5.3, “Immediate Messages” for more details.
qpid.flow_control_wait_failure	long	60000	Used with Section 5.5.4, “Flow Control”. The amount of time (in milliseconds) to wait before timing out.
qpid.flow_control_wait_notify_period	long	5000	Used with Section 5.5.4, “Flow Control”. The frequency at which the log message informing that the producer is flow controlled .
sync_publish	string	"" (disabled)	If 'all' is set then messages will be sent synchronously. This can also be set per connection using the Connection URL options.

Table 9.5. Config Options For Threading

Property Name	Type	Default Value	Description
qpid.thread_factory	string	org.apache.qpid.thread.DefaultThreadFactory	Specifies the thread factory to use. If using a real time JVM, you need to set the above property to `org.apache.qpid.thread.Realtime`
qpid.rt_thread_priority	int	20	Specifies the priority (1-99) for Real time threads created by the real time thread factory.

Table 9.6. Config Options For I/O

Property Name	Type	Default Value	Description
qpid.sync_op_timeout	long	60000	The length of time (in milliseconds) to wait for a synchronous operation to complete. For compatibility with older clients, the synonym

Property Name	Type	Default Value	Description
			`amqj.default_syncwrite_timeout` is supported.
qpid.tcp_nodelay	boolean	true	Sets the TCP_NODELAY property of the underlying socket. The default was changed to true as of Qpid 0.14. This can also be set per connection using the Connection URL broker option`tcp_nodelay`. options. For compatibility with older clients, the synonym `amqj.tcp_nodelay` is supported.
qpid.send_buffer_size	integer	65535	Sets the SO_SNDBUF property of the underlying socket. Added in Qpid 0.16. For compatibility with older clients, the synonym `amqj.sendBufferSize` is supported.
qpid.receive_buffer_size	integer	65535	Sets the SO_RCVBUF property of the underlying socket. Added in Qpid 0.16. For compatibility with older clients, the synonym `amqj.receiveBufferSize` is supported.
qpid.failover_method_timeout	long	60000	During failover, this is the timeout for each attempt to try to re-establish the connection. If a reconnection attempt exceeds the timeout, the entire failover process is aborted. It is only applicable for AMQP 0-8/0-9/0-9-1 clients.

Table 9.7. Config Options For Security - Using SSL for securing connections or using EXTERNAL as the SASL mechanism.

Property Name	Type	Default Value	Description
qpid.ssl_timeout	long	60000	Timeout value used by the Java SSL engine when waiting on operations.
qpid.ssl.KeyManagerFactory.algorithm	string	-	The key manager factory algorithm name. If not set, defaults to the value returned from the Java runtime call `KeyManagerFactory.getDefaultAlg` For compatibility with older clients, the synonym `qpid.ssl.keyStoreCertType` is supported.
qpid.ssl.TrustManagerFactory.algorithm	string	-	The trust manager factory algorithm name. If not set, defaults to the value returned from the Java runtime call `TrustManagerFactory.getDefaultA` For compatibility with older clients, the synonym `qpid.ssl.trustStoreCertType` is supported.

Table 9.8. Config Options For Security - Standard JVM properties needed when Using SSL for securing connections or using EXTERNAL as the SASL mechanism.[a]

Property Name	Type	Default Value	Description
javax.net.ssl.keyStore	string	jvm default	Specifies the key store path. This can also be set per connection using the Connection URL options.
javax.net.ssl.keyStorePassword	string	jvm default	Specifies the key store password. This can also be set per connection using the Connection URL options.
javax.net.ssl.trustStore	string	jvm default	Specifies the trust store path. This can also be set per connection using the Connection URL options.
javax.net.ssl.trustStorePassword	string	jvm default	Specifies the trust store password.

Property Name	Type	Default Value	Description
			This can also be set per connection using the Connection URL options.

[a]Qpid allows you to have per connection key and trust stores if required. If specified per connection, the JVM arguments are ignored.

Chapter 10. Logging

The Qpid JMS client uses the Apache SLF4J [http://www.slf4j.org] logging framework. All logging activity created by the client is directed through the SLF4J API. SLF4J is a is a façade for other common logging frameworks. This makes it easy for application authors to use their prefered logging framework in their application stack, and have the Qpid JMS Client use it too.

SLF4J suppplies bindings for many common logging frameworks (JUL [http://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html], Apache Log4J [http://logging.apache.org/log4j/1.2/], Logback [http://logback.qos.ch].

Include the SLF4J binding corresponding to the logging framework of your choosen logging framework on classpath. For full details, see the SLF4J documentation [http://www.slf4j.org].

10.1. Recommended Production Logging Level

In production, it is recommended that you configure your logging framework is configured with logger `org.apache.qpid` set to `WARN`.

If you are using Apache Log4j with a log4j.properties file, this simply means adding the following line:

```
org.apache.qpid=WARN
```

If you are using another logging framework, or you are using Log4j but configuring in another manner, refer to the documentation accompanying the logging framework for details of how to proceed.

10.2. Enabling Debug

If you are experiencing a problem, it can be informative to enable debug logging to allow the behaviour of the Qpid JMS client to be understood at a deeper level.

To do this, set the `org.apache.qpid` logger to `DEBUG`.

If you are using Apache Log4j with a log4j.properties file, this simply means adding (or changing) the following line:

```
org.apache.qpid=DEBUG
```

If you are using another logging framework, or you are using Log4j but configuring in another manner, refer to the documentation accompanying the logging framework for details of how to proceed.

Appendix A. Exceptions

The methods of Qpid JMS Client throw JMSExceptions [http://docs.oracle.com/javaee/6/api/javax/jms/JMSException.html] in response to error conditions. Typically the exception's message (#getMessage()) summarises the error condition, with contextual information being provided by the messages of linked exception(s). To understand the problem, it is important to read the messages associated with *all* the linked exceptions.

The following table describes some of the more common exceptions linked to JMSException thrown by JMS methods whilst using the client:

Table A.1. Exceptions linked to JMSExceptions thrown by JMS methods

Linked Exception	Message	Explanation/Common Causes
AMQUnresolvedAddressException	*message varies*	Indicates that the hostname included in the Connection URL's brokerlist, could not be resolved, . This could mean that the hostname is mispelt, or there is name resolution problem.
AMQConnectionFailure	Connection refused	Indicates that the host included in the Connection URL's brokerlist, actively refused the connection. This could mean that the hostname and/or port number is incorrect, or the Broker may not be running.
AMQConnectionFailure	connect timed out	Indicates that the host included in the Connection URL's brokerlist, could not be contacted within the connecttimeout. This could mean that the host is shutdown, or a networking routing problem means the host is unreachable.
AMQConnectionFailure	General SSL Problem; PKIX path building failed; unable to find valid certification path to requested target	Indicates that the CA that signed the Broker's certificate is not trusted by the JVM of the client. If the Broker is using a private-CA (or a self signed certificate) check that the client has been properly configured with a truststore. See Section 5.3.3, “SSL”
AMQConnectionFailure / AMQAuthenticationException	not allowed	Indicates that the user cannot be authenticated by the Broker. Check the username and/or password elements within the Connection URL.
AMQConnectionFailure / AMQSecurityException	Permission denied: *virtualhost name*; access refused	Indicates that the user is not authorised to connect to the given virtualhost. The user is recognised by the Broker and is using the correct password but does not have permission. This exception normally indicates that the user (or

Linked Exception	Message	Explanation/Common Causes
		group) has not been permissioned within the Broker's Access Control List (ACL) [../../AMQP-Messaging-Broker-Java-Book/html/Java-Broker-Security-ACLs.html].
AMQTimeoutException	Server did not respond in a timely fashion; Request Timeout	Indicates that the broker did not respond to a request sent by the client in a reasonable length of time. The timeout is governed by `qpid.sync_op_timeout`. This can be a symptom of a heavily loaded broker that cannot respond or the Broker may have failed in unexpected manner. Check the broker and the host on which it runs and performance of its storage.
AMQSecurityException	Permission denied: *message varies*	Indicates that the user is not authorised to use the given resource or perform the given operation. This exception normally indicates that the user (or group) has not been permissioned within the Broker's Access Control List (ACL) [../../AMQP-Messaging-Broker-Java-Book/html/Java-Broker-Security-ACLs.html].

The following table describes some of the more common exceptions linked to JMSException sent to ExceptionListener [http://docs.oracle.com/javaee/6/api/javax/jmx/ExceptionListener.html] instances.

Table A.2. Exceptions linked to JMSExceptions received by ExceptionListeners

Linked Exception	Message	Explanation/Common Causes
AMQNoRouteException	No Route for message [Exchange: *exchange name*, Routing key: *routing key*] [error code 312: no route]	Indicate that the named exchange is unable to route a message to at least one queue. This will occur if a queue has been improperly bound to an exchange. Use the Broker's management interface to check the bindings. See Section 5.5.1, “Mandatory Messages”
AMQNoConsumersException	Immediate delivery is not possible. [error code 313: no consumers]	Immediate delivery was requested by the MessageProducer, but as there are no consumers on any target queue, the message has been returned to the publisher. See Section 5.5.3, “Immediate Messages”
AMQDisconnectedException	Server closed connection and reconnection not permitted	Indicates that the connection was closed by the Broker, and as failover options are not included in the

Linked Exception	Message	Explanation/Common Causes
		Connection URL, the client has been unable to reestablish connection. The Connection is now closed and any attempt to use either Connection object, or any objects created from the Connection will receive an IllegalStateException [http://docs.oracle.com/javaee/6/api/javax/jms/IllegalStateException.html].
AMQDisconnectedException	Server closed connection and no failover was successful	Indicates that the connection was closed by the Broker. The client has tried failover according to the rules of the failover optionswithin the Connection URL, but these attempts were all unsuccessful. The Connection is now closed and any attempt to use either Connection object, or any objects created from the Connection will receive an IllegalStateException [http://docs.oracle.com/javaee/6/api/javax/jms/IllegalStateException.html].

Appendix B. Minimal Maven POM

The following is a minimal Maven POM required to use the Qpid Client. It is suitable for use with the examples included in this book.

Example B.1. Minimal Maven POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/20
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache
  <modelVersion>4.0.0</modelVersion>
  <groupId>test</groupId>
  <artifactId>test</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.apache.qpid</groupId>
      <artifactId>qpid-client</artifactId>
      <version>0.32-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.6.4</version>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-jms_1.1_spec</artifactId>
      <version>1.1.1</version>
    </dependency>
  </dependencies>
</project>
```

Note: We use the SLF4J Binding for Log4J12 here, but any SLF4J Binding could be used instead. Similarly, Geronimo JMS Spec is used, but any dependency that provides the JMS 1.1 specification could be subsituted.

Appendix C. JMS Extensions

This section illustrates using Qpid specific extentions to JMX for the managament of queues, exchanges and bindings.

Important

It is not recommended that these extensions are generally used. These interfaces are subject to change and will not be supported in this form for AMQP 1.0. Instead, the reader is directed towards the Managment interfaces of the Broker.

C.1. Queue Management

These extensions allow queues to be created or removed.

C.1.1. Queue creation

The following example illustrates the creation of the a LVQ queue from a javax.jms.Session object. Note that this utilises a Qpid specific extension to JMS and involves casting the session object back to its Qpid base-class.

Example C.1. Creation of an LVQ using the Qpid extension to JMS

```
Map<String,Object> arguments = new HashMap<String, Object>();
arguments.put("qpid.last_value_queue_key","ISIN");
AMQDestination amqQueue = (AMQDestination) context.lookup("myqueue");
((AMQSession<?,?>) session).createQueue(
        AMQShortString.valueOf(amqQueue.getQueueName()),
        amqQueue.isAutoDelete(),
        amqQueue.isDurable(),
        amqQueue.isExclusive(),
        arguments);
```

C.2. Binding Management

These extensions allow bindings to be created or removed.

C.2.1. Binding creation

The following example illustrates the creation of queue binding to topic exchange with JMS client.

Example C.2. Binding a queue using JMS

```
ConnectionFactory connectionFactory = ...
Connection connection = connectionFactory.createConnection();
AMQSession<?, ?> session = (AMQSession<?,?>)connection.createSession(false, Sessio

...

AMQShortString queueName = new AMQShortString("testQueue");
```

```
AMQShortString routingKey = new AMQShortString("testRoutingKey");
AMQDestination destination = (AMQDestination) session.createQueue(queueName.asStri

...

// binding arguments
Map<String, Object> arguments = new HashMap<String, Object>();
arguments.put("x-filter-jms-selector", "application='app1'");

// create binding
session.bindQueue(queueName, routingKey, FieldTable.convertToFieldTable(arguments)
    new AMQShortString("amq.topic"), destination);
```

Appendix D. PooledConnectionFactory

Qpid client provides `PooledConnectionFactory` which is a special implementation of ConnectionFactory [http://docs.oracle.com/javaee/6/api/javax/jms/ConnectionFactory.html] supporting Connection [http://docs.oracle.com/javaee/6/api/javax/jms/Connection.html] pooling.

The `PooledConnectionFactory` caches a predefined number of connections thus saving an application which connects frequently time. The `Connection` instance is taken from the pool whenever method `PooledConnectionFactory#createConnection()` is invoked and returned into the pool when method `Connection#close()` is called.

A user can configure a maximum allowed number of connections to remain in pool (10 by default) by calling `PooledConnectionFactory#setMaxPoolSize(int)`. When number of connections exceeds the value set for maximum pool size, `PooledConnectionFactory` starts to work as a normal ConnectionFactory [http://docs.oracle.com/javaee/6/api/javax/jms/ConnectionFactory.html] and creates a new connection every time method `PooledConnectionFactory#createConnection()` is invoked.

The Connection URL is set by invoking method `PooledConnectionFactory#setConnectionURLString(String)`.

A user can specify the maximum time a connection may remain idle in pool by calling `PooledConnectionFactory#setConnectionTimeout(long)` passing a value in milliseconds. If connection is not used within the specified interval it is closed automatically.

This implementation can be useful in *Spring JMS* based applications. An example below demonstrates how to configure `PooledConnectionFactory` in the Spring xml configuration.

Example D.1. Example of configuring *PooledConnectionFactory* in spring xml configuration.

```
<bean id="pooledConnectionFactory" class="org.apache.qpid.client.PooledConnectionF
  <!-- set maximum number of pool connections to 20-->
  <property name="maxPoolSize" value="20"></property>
  <!-- set the timeout for connection to remain open in pool without being used --
  <property name="connectionTimeout" value="60000"></property>
  <!-- set connection URL as String -->
  <property name="connectionURLString" value="amqp://guest:guest@clientid/default?
</bean>
```

PooledConnectionFactory spring bean can be configured with such *spring-jms* beans like *DefaultMessageListenerContainer* and *JmsTemplate*. The example below demonstrates how to do that

Example D.2. Examples of configuring *PooledConnectionFactory* with *DefaultMessageListenerContainer* and *JmsTemplate*.

```
<bean id="jmsProducerTemplate" class="org.springframework.jms.core.JmsTemplate">
    <!-- set reference to pooledConnectionFactory bean -->
    <property name="connectionFactory" ref="pooledConnectionFactory"></property>
    <property name="defaultDestination" ref="destination" />
</bean>
```

```
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageList
    <!-- set reference to pooledConnectionFactory bean -->
    <property name="connectionFactory" ref="pooledConnectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener" />
</bean>
```

Note

If using `DefaultMessageListenerContainer` with `cacheLevel` set to `NONE` it is important that `maxConcurrentConsumer` does not exceed the value of maximum pool size set on `PooledConnectionFactory` bean. If this is not the case, once the number of in-use connections reaches the the *PooledConnectionFactory#maxPoolSize* a new connection will be opened for each and every message receipt i.e. a connection per message anti-pattern.

Appendix E. How to bind Qpid destinations and connection factories into Tomcat JNDI

Qpid client destinations and connection factories can be registered in external JNDI containers, for example, Tomcat JNDI implementation.

`org.apache.qpid.jndi.ObjectFactory` implements javax.naming.spi.ObjectFactory [http://docs.oracle.com/javase/7/docs/api/javax/naming/spi/ObjectFactory.html] allowing it to create instances of `AMQConnectionFactory`, `PooledConnectionFactory`, `AMQConnection`, `AMQQueue` and `AMQTopic` in external JNDI container from javax.naming.Reference [http://docs.oracle.com/javase/7/docs/api/javax/naming/Reference.html]s.

Additionally, `AMQConnectionFactory`, `PooledConnectionFactory` and `AMQDestination` (parent of `AMQQueue` and `AMQTopic`) implement javax.naming.Referenceable [http://docs.oracle.com/javase/7/docs/api/javax/naming/Referenceable.html] allowing creation of javax.naming.Reference [http://docs.oracle.com/javase/7/docs/api/javax/naming/Reference.html] objects for binding in external JNDI implementations.

`org.apache.qpid.jndi.ObjectFactory` allows the creation of:

- an instance of `ConnectionFactory` from a `Reference` containing reference address (javax.naming.RefAddr [http://docs.oracle.com/javase/7/docs/api/javax/naming/RefAddr.html]) `connectionURL` with content set to a Connection URL.
- an instance of `PooledConnectionFactory` from a `Reference` containing reference address (javax.naming.RefAddr [http://docs.oracle.com/javase/7/docs/api/javax/naming/RefAddr.html]) `connectionURL` with content set to a Connection URL.
- an instance of `AMQConnection` from a `Reference` containing reference address (javax.naming.RefAddr [http://docs.oracle.com/javase/7/docs/api/javax/naming/RefAddr.html]) `connectionURL` with content set to a Connection URL.
- an instance of `AMQQueue` from a `Reference` containing reference address (javax.naming.RefAddr [http://docs.oracle.com/javase/7/docs/api/javax/naming/RefAddr.html]) `address` with content set to either Address [../../Programming-In-Apache-Qpid/html/] or Binding URL.
- an instance of `AMQTopic` from a `Reference` containing reference address (javax.naming.RefAddr [http://docs.oracle.com/javase/7/docs/api/javax/naming/RefAddr.html]) `address` with content set to either Address [../../Programming-In-Apache-Qpid/html/] or Binding URL.

Note

For `AMQQueue` and `AMQTopic` prefix `BURL:` need to be specified for Binding URL. Otherwise, client will try to parse content using Address [../../Programming-In-Apache-Qpid/html/] format.

An example below demonstrates how to create JNDI resources in the Tomcat container using Resource declarations in context.xml (A Tomcat specific web application configuration file usually added into war under /META-INF/context.xml).

Example E.1. An example of Tomcat context.xml declaring Qpid JNDI resources

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE xml>
<Context>

  <Resource name="jms/connectionFactory" auth="Container"
            type="org.apache.qpid.client.AMQConnectionFactory"
            factory="org.apache.qpid.jndi.ObjectFactory"
            connectionURL="amqp://guest:guest@clientid/?brokerlist='localhost:5672

  <Resource name="jms/pooledConnectionFactory" auth="Container"
            type="org.apache.qpid.client.PooledConnectionFactory"
            factory="org.apache.qpid.jndi.ObjectFactory"
            connectionURL="amqp://guest:guest@clientid/?brokerlist='localhost:5672
            maxPoolSize="20" connectionTimeout="60000"/>

  <Resource name="jms/queue" auth="Container"
            type="org.apache.qpid.client.AMQQueue"
            factory="org.apache.qpid.jndi.ObjectFactory"
            address="BURL:direct://amq.direct//myQueue?durable='true'"/>

  <Resource name="jms/topic" auth="Container"
            type="org.apache.qpid.client.AMQTopic"
            factory="org.apache.qpid.client.AMQConnectionFactory"
            address="BURL:topic://amq.topic//myTopic?routingkey='myTopic'"/>

</Context>
```

In the example above `AMQConnectionFactory` would be registered under JNDI name "jms/connectionFactory", `PooledConnectionFactory` would be registered under JNDI name "jms/pooledConnectionFactory", `Queue` "myQueue" would be registered under JNDI name "jms/queue" and JMS `Topic` destination "myTopic" would be registered under JNDI name "jms/topic". (All resources will be bound under "java:comp/env"). On declaration of `PooledConnectionFactory` optional maxPoolSize and connectionTimeout are set to 20 and 60000 milliseconds accordingly.

The client application can find the resources declared in Tomcat context.xml using the code below:

Example E.2. An example of JNDI lookup for Qpid resources registered in Tomcat JNDI

```
Context context = new InitialContext();
Context environmentContext = (Context)context.lookup("java:comp/env");
...
ConnectionFactory connectionFactory = (ConnectionFactory) environmentContext.l
...
Queue queue = (Queue)environmentContext.lookup("jms/queue");
...
Topic topic = (Topic)environmentContext.lookup("jms/topic");
...
```

Note

In order to support backward compatibility `AMQConnectionFactory` continues to implement javax.naming.spi.ObjectFactory [http://docs.oracle.com/javase/7/docs/api/

javax/naming/spi/ObjectFactory.html] and can be used to instantiate JNDI resources from javax.naming.Reference [http://docs.oracle.com/javase/7/docs/api/javax/naming/Reference.html]s. However, its method `getObjectInstance` is marked as `Deprecated` and will be removed in future version of client. For backward compatibility, Qpid JNDI resources can be declared using fully qualified class names as addresses. That will became unsupported in future version as well. An example of Tomcat context.xml with declarations of JNDI resources using deprecated factory and addresses is provided below.

Example E.3. An example of Tomcat context.xml declaring Qpid JNDI resources using deprecated `ObjectFactory` and deprecated addresses

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE xml>
<Context>

  <Resource name="jms/queue" auth="Container"
            type="org.apache.qpid.client.AMQQueue"
            factory="org.apache.qpid.client.AMQConnectionFactory"
            org.apache.qpid.client.AMQQueue="direct://amq.direct//myDurableQueu

  <Resource name="jms/topic" auth="Container"
            type="org.apache.qpid.client.AMQTopic"
            factory="org.apache.qpid.client.AMQConnectionFactory"
            org.apache.qpid.client.AMQTopic="topic://amq.topic//myTopic?routing

  <Resource name="jms/connectionFactory" auth="Container"
            type="org.apache.qpid.client.AMQConnectionFactory"
            factory="org.apache.qpid.client.AMQConnectionFactory"
            org.apache.qpid.client.AMQConnectionFactory="amqp://guest:guest@cli

</Context>
```