JMS 2.0: Injection of JMS objects

About this document

1. This working document suggests how JMS objects might be injected into applications, written from the perspective of an application developer.
2. It considers both Java EE and Java SE applications.
3. It focuses on how the API might appear to the application, and doesn't worry too much about how this may be implemented.
4. The main purpose of this document is to define the full range of possible annotations what would be needed to allow all JMS objects to be injected. A summary can be found in "Summary of annotations" on page 31.
5. However this leaves several important issues unresolved, and it is must be stressed that this document in its current form does not define a usable API. Amongst other issues, the API presented here contains an undesirable number of duplicate annotations, and it doesn't attempt to define how the relationships between injected objects can be specified. For a discussion of these and other unresolved issues, see "Unresolved issues" on page 33.
6. This next step in developing this API is to discuss how these issues can be resolved, and how it can be implemented.

Annotations to inject individual JMS objects

First of all, let's consider each JMS object in turn and what annotations we would need to allow them to be injected. This discussion focuses on the pieces of information that needs to be supplied to allow each object to be injected, and how

Injecting a Connection:

Let's start by considering you might inject a JMS connection. The current JMS API provides the following two methods to create a `Connection`, both on `ConnectionFactory`:

```
connectionFactory.createConnection()
```

and

```
connectionFactory.createConnection(userName,password)
```

where the `connectionFactory` may be obtained using a `@Resource` annotation.

This suggests that the annotation to inject a Connection could be:

```
@Inject @JMSConnection(lookup="jms/connFactory")
Connection connection;
```

or

```
@Inject @JMSConnection (lookup="jms/connFactory", user="admin", password="secret")
Connection connection;
```

Example 1: Creating a Connection with default credentials (Java EE)

This example considers a use case in which a Java EE application creates a connection using default credentials.

Here's how you might do this using the existing JMS 1.1 API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

public void oldWay1() throws JMSException{
       Connection connection = connectionFactory.createConnection();

       // do stuff with connection

       connection.close();
}
```

Here's how you might do this using the new API.

```
@Inject @JMSConnection(lookup="jms/connFactory")
Connection connection;

public void newWay1(){
       // do stuff with connection
}
```

Note that we don't need to call the close() method when using an injected connection. The connection will be automatically closed when it falls out of scope.

This example illustrates the following unresolved issues:

- The use of injection has made it necessary to change the `Connection` object from a local variable to an instance variable, despite the application only using it in a single method. This does not reflect the true semantics and is potentially misleading. In particular it raises the possibility of the same `Connection` object inadvertently being used concurrently from multiple threads to create a `Session`, which is illegal in Java EE.
- This example does not consider what the scope of the injected object would be. For a discussion of scope, see "Scope of injected variables" on page 36.

Example 2: Creating a connection with user and password (Java EE)

This example considers a use case in which a Java EE application creates a connection, specifying a user and password.

Here's how you might do this using the existing JMS 1.1 API.

```
@Resource(lookup = "jms/connFactory")
ConnectionFactory connFact;

public void oldWay2() throws JMSException{
       Connection connection2 = connFact.createConnection("admin","secret");

       // do stuff with connection

       connection.close();
}
```

Here's how you might do this using the new API.

```
@Inject @JMSConnection (lookup="jms/connFactory", user="admin", password="secret")
Connection connection2;

public void newWay2(){

       // do stuff with connection

}
```

Unresolved issue:

- (Repeated point) The use of injection has made it necessary to change the `Connection` object from a local variable to an instance variable, despite the application only using it in a single method. See Example 1 for more observations on this issue.
- The restriction that that username and password need to be declared in an instance variable prevents the user and password being set at runtime. It is inconvenient and insecure to hardcode passwords in the code. This is probably not an issue with Java EE, where user and password authentication is not particularly useful, but it may be with Java SE.

Injecting a Session

Now let's consider how you might inject a JMS session. The JMS API provides the following method on `Connection` to create a `Session`:

```
createSession(boolean transacted, int acknowledgeMode)
```

However these arguments are mutually exclusive and so JMS 2.0 (issue JMS_Spec-45) proposes a new method which combines these into a single argument:

```
createSession(int sessionMode)
```

This suggests that an appropriate annotation to inject a Session could be:

```
@Inject
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
Session session;
```

As can be seen, the Session is injected using two separate annotations: `@JMSConnection` and `@JMSSession`. The `@JMSConnection`n annotation is used to configure the connection used by this session, and the `@JMSSession` abbotation is used to configure the session itself.

This is considered preferable to defining a single annotation which allows both sets of information to be defined:

```
@Inject // this idea rejected
@JMSSession(lookup="jms/connectionFactory"), sessionMode=Session.AUTO_ACKNOWLEDGE)
Session session;
```

The use of two separate annotations was considered preferable to this because it more accurately reflected the underlying objects (of which users will need to remain aware) and it allowed the same annotation to be used to inject different objects which was considered preferable to having multiple annotations which define the same attribute.

In a Java EE transaction the arguments to `createSession` arguments are ignored altogether. and so JMS 2.0 (issue JMS_Spec-45) proposes a further new method with no arguments:

```
createSession()
```

The suggested meaning of this method would be:

- In a Java EE transaction, the session would be part of a transaction managed by the container.
- In a Java EE undefined transaction context, the session will have a sessionMode of Session.AUTO_ACKNOWLEDGE.
- In a normal Java SE environmentm this is equivalent to calling createSession(Session.AUTO_ACKNOWLEDGE)

The equivalent in annotation to using this method is simply to omit the `@JMSSession` annotation completely:

```
@Inject
@JMSConnection(lookup="jms/connectionFactory")
Session session;
```

Example 3a: Creating a Session (Java SE)

This example considers a use case in which a Java SE application creates a connection (with default credentials) and uses it to create a session with a particular acknowledgement mode.

Here's how you might do this using the existing JMS 1.1 API.

```
public void oldWay3a() throws JMSException{

   InitialContext initialContext = ... // DETAILS OMITTED

   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");

   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

   // do stuff with session

   // close connection after use
   connection.close();
}
```

Here's how you might do this using the new API.

```
@Inject
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
Session session;

public void newWay3a(){

       // do stuff with session

}
```

Note that the injected `Session` is specified using a single attribute, `sessionMode`.

This example illustrates the following unresolved issue:

- These annotations include the JNDI names of administered objects. However, whereas a Java EE environment comes with a ready-configured JNDI provider, a Java SE environment does not. This implies that the use of these annotations in a Java SE environment is dependent on a suitable JNDI provider being available and configured.
- The use of injection has made it necessary to change the `Session` object from a local variable to an instance variable, despite the application only using it in a single method. This does not reflect the true semantics and is potentially misleading. In particular it raises the possibility of the same `Session` object inadvertently being used concurrently from multiple threads, which is illegal in JMS.

Example 3b: Creating a Session in a Java EE container-managed transaction (Java EE)

This example considers a use case in which a Java EE session bean method which uses a container-managed transaction creates a connection (with default credentials) and uses it to create a session.

Here's how you might do this using the existing JMS 1.1 API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void oldWay3b() throws JMSException{
       Connection connection = connectionFactory.createConnection();
       Session session = connection.createSession(true,Session.TRANSACTED);

       // do stuff with session

       connection.close();
}
```

Note that since the session is created within a Java EE transaction, the arguments to `createSession` are ignored so it doesn't actually matter what they are, so the argument supplied are dummies.

Here's how you might do this using the new API.

```
@Inject
@JMSConnection(lookup="jms/connectionFactory")
// No need for @JMSSession annotation in Java EE
Session session;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void newWay3b(){

       // do stuff with session

}
```

Note that since the session is created within a Java EE transaction, there is no need to define a `sessionMode` attribute and therefore no need to specify a `@JMSSession` annotation at all.

This example illustrates the following unresolved issue:

- (Repeated point) The use of injection has made it necessary to change the `Session` object from a local variable to an instance variable, despite the application only using it in a single method. See Example 3a for more observations on this issue.
- The runtime must not create the session until after the container managed transaction has started.

Example 3c: Creating a Session in a Java EE bean-managed transaction (Java EE)

This example considers a use case in which a Java EE session bean method which uses a bean-managed transaction creates a connection (with default credentials) and uses it to create a session.

Here's how you might do this using the existing JMS 1.1 API.

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class ExampleSessionBean implements ExampleSessionBeanLocal {

   @Resource(lookup = "jms/connectionFactory")
   ConnectionFactory connectionFactory;

   @Resource
   UserTransaction userTransaction;

   public void oldWay3c() throws JMSException{
      userTransaction.begin();

      Connection connection = connectionFactory.createConnection();
      Session session = connection.createSession(true,Session.TRANSACTED);

      // do stuff with session

      connection.close();
      userTransaction.commit();
   }
}
```

Note that since the session is created within a Java EE transaction, the arguments to `createSession` are ignored so it doesn't actually matter what they are, so the argument supplied are dummies.

Here's how you might do this using the new API.

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class ExampleSessionBean implements ExampleSessionBeanLocal {

   @Inject
   @JMSConnection(lookup="jms/connectionFactory")
   No need for @JMSSession annotation in Java EE
   Session session;

   @Resource
   UserTransaction userTransaction;

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void newWay3b(){
      userTransaction.begin();

      // do stuff with session

      userTransaction.commit();
   }
}
```

Note that since the session is created within a Java EE transaction, there is no need to define a `sessionMode` attribute and therefore no need to specify a `@JMSSession` annotation at all.

This example illustrates the following unresolved issue:

- (Repeated point) The use of injection has made it necessary to change the `Session` object from a local variable to an instance variable, despite the application only using it in a single method. See Example 3a for more observations on this issue.
- The runtime must not create the session until after `userTransaction.begin()` has been called to start the bean managed transaction.

Example 3d: Creating a Session in a Java EE unspecified transaction context (Java EE)

This example considers a use case in which a Java EE session bean method which executes in an unspecified transaction context and which creates a connection (with default credentials) and uses it to create a session with a session mode of DUPS_OK_ACKNOWLEDGE.

Here's how you might do this using the existing JMS 1.1 API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@TransactionAttribute(TransactionAttributeType.NEVER)
public void oldWay3b() throws JMSException{
   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(false,Session.DUPS_OK_ACKNOWLEDGE);

   // do stuff with session

   connection.close();
}
```

The EJB specification does not explicitly describe how `createSession` should behave when run in an unspecified transaction context. However the interpretation being followed for JMS 2.0 is that can be used to create a non-transacted session with an acknowledgement mode of either `AUTO_ACKNOWLEDGE` or `DUPS_OK_ACKNOWLEDGE`.

Here's how you might do this using the new API.

```
@Inject
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.DUPS_OK_ACKNOWLEDGE)
Session session;

@TransactionAttribute(TransactionAttributeType.NEVER)
public void newWay3b(){

   // do stuff with session

}
```

Note that if we wanted the session mode to be `AUTO_ACKNOWLEDGE` we could have omitted the `@JMSSession` annotation since `AUTO_ACKNOWLEDGE` would be used by default. when there was no transactional context.

This example illustrates the following unresolved issue:

- (Repeated point) The use of injection has made it necessary to change the `Session` object from a local variable to an instance variable, despite the application only using it in a single method. See Example 3a for more observations on this issue.

Injecting a MessageProducer

The current JMS API provides the following method on `Session` to create a `MessageProducer`:

```
session.createProducer(destination)
```

The destination parameter may be null, in which case the producer is referred to as "unidentified".

The annotation to inject an unidentified MessageProducer could be:

```
@Inject
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
MessageProducer producer;
```

Note that no additional annotation is necessary to inject an unidentified MessageProducer beyond the @JMSConnection and @JMSSession annotations needed to define its Session.

The annotation to inject an identified MessageProducer could be:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@JMSDestination(lookup="jms/inboundQueue")
MessageProducer producer;
```

Note that in this case we need to specify the Destination to which the MessageProducer will send messages. This is specified using a `@JMSDestination` annotation.

Example 4: Creating an unidentified MessageProducer (Java EE)

Here's an example using the existing API:

```
@Resource(lookup = "jms/connFactory")
ConnectionFactory connFact;

public void oldWay4() throws JMSException{
       Connection connection = connFact.createConnection();
       Session session = connection.createSession(Session.AUTO_ACKNOWLEDGE);
       MessageProducer messageProducer = session.createProducer(null);
       // do stuff with producer

       connection.close();
}
```

Here's the same example using the new API:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
// No need for @JMSSession annotation in Java EE
MessageProducer producer;

public void newWay4(){
       // do stuff with producer
}
```

Example 5: Creating an identified MessageProducer (Java EE)

Here's an example using the existing API

```
@Resource(lookup = "jms/connFactory")
ConnectionFactory connFact;

@Resource(lookup="jms/inboundQueue")
Destination destination;

public void oldWay5() throws JMSException{
       Connection connection = connFact.createConnection();
       Session session = connection.createSession(Session.AUTO_ACKNOWLEDGE);
       MessageProducer messageProducer = session.createProducer(destination);
       // do stuff with producer

       connection.close();
}
```

Here's the same example using the new API:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
// No need for @JMSSession annotation in Java EE
@JMSDestination(lookup="jms/inboundQueue")
MessageProducer producer;

public void newWay5(){
       // do stuff with producer
}
```

Injecting a Message

The current JMS API provides the following methods on `Session` to create a `Message` and its various subtypes:

```
session.createMessage()
session.createBytesMessage()
session.createMapMessage()
session.createObjectMessage()
session.createObjectMessage(serializableObject)
session.createStreamMessage()
session.createTextMessage()
session.createTextMessage(text)
```

The annotation to inject each type of `Message` could be:

```
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@Inject Message message;

@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@Inject BytesMessage bytesMessage;

@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@Inject MapMessage mapMessage;

@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@Inject ObjectMessage objectMessage;

@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@Inject StreamMessage streamMessage;

@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@Inject TextMessage textMessage;
```

Unresolved issues:

- It might seem excessive to expect users to have to specify the @JMSConnection and @JMSSession annotations when they wish to inject a method, but this is a direct reflection of the JMS API which requires you to create a connection and session before you can create a message.

  However in practice it doesn't matter which session you use (which will be made explicit in JMS 2.0 - issue JMS_SPEC-52). It might therefore be simplest if the default behavior to use any existing `Session` that was available. However in the rather special case where there was a choice of sessions from multiple JMS providers then it would be necessary to allow the application to specify the Session that should be used (since it may be less efficient to use one vendor's message implementation with another vendor's MessageProducer). The simplest way to specify which JMS provider should be used ti create an injected message might be to use the @JMSConnection annotation to specify a

suitable connection factory.

This issue is discussed further in Example 9.

Example 6: Creating a TextMessage

Here's an example using the existing API:

```
@Resource(lookup = "jms/connFactory")
ConnectionFactory connFact;

public void oldWay6() throws JMSException{
       Connection connection = connFact.createConnection();
       Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
       TextMessage textMessage = session.createTextMessage();

       // do stuff with textMessage

       connection.close();
}
```

Here's an example using the new API:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
TextMessage textMessage;

public void newWay6(){
       // do stuff with textMessage
}
```

Injecting a MessageConsumer

The current API provides the following methods on `Session` to create a `MessageConsumer`:

```
// Create a consumer on a destination with no message selector:
createConsumer(destination)

// Create a consumer on a destination with a specified message selector:
createConsumer(destination, messageSelector)

// Create a consumer on a destination
// with a message selector and the noLocal flag set
createConsumer(destination, messageSelector, true)

// Create a durable subscriber on a destination
createDurableSubscriber(topic, subscriptionName)

// Create a durable subscriber on a destination
// with a message selector and the noLocal flag set
createDurableSubscriber(topic, subscriptionName, messageSelector, noLocal)
```

The corresponding annotations to inject a `MessageConsumer` could be as follows:

Creating a consumer on a destination with no message selector:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@JMSDestination(lookup="jms/inboundQueue")
MessageConsumer messageConsumer;
```

Creating a consumer on a destination with a specified message selector:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@JMSDestination(lookup="jms/inboundQueue")
@JMSConsumer(messageSelector="foo")
MessageConsumer messageConsumer;
```

Creating a consumer on a destination with a message selector and the noLocal flag set:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@JMSDestination(lookup="jms/inboundQueue")
@JMSConsumer(messageSelector="foo", noLocal=true)
MessageConsumer messageConsumer;
```

In all the above examples, messages would not be received unless the underlying Conneciton was somehow obtained and its `start()` method called. To avoid the need to do this, we could define an additional annotation on `@JMSConsumer` which would call `start()` on the Connection:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@JMSDestination(lookup="jms/inboundQueue")
@JMSConsumer(start=true)
MessageConsumer messageConsumer;
```

Creating a durable subscriber on a destination:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@JMSDestination(lookup="jms/inboundQueue")
@JMSConsumer(messageSelector="foo" noLocal=true)
@JMSDurableSubscriber(name="bar")
MessageConsumer messageConsumer;
```

Obervations:

- The `@JMSDestination` annotation is used by both `MessageProducer` and `MessageConsumer` objects.
- There is a `@JMSConsumer` annotation but no `@JMSProducer` annotation

- We have an additional `@DurableSubscriber` annotation to allow attributes specific to durable subscriber to be specified. We could merge this with `@JMSConsumer` but that would pollute `@JMSConsumer` with attributes that were not always relevant.

- To destroy a durable subscription the existing JMS 1.1 API on `Session` would need to be used:

```
unsubscribe(java.lang.String name)
```

  This method can't be moved to the `MessageConsumer` object since it can only be used when there is no active consumer on the subscriber.

Example 7: Creating a consumer on a queue (Java EE)

Here's an example of creating a consumer on a queue using the existing API:

```
@Resource(lookup = "jms/connFactory")
ConnectionFactory connFact;

@Resource(lookup="jms/inboundQueue")
Destination destination;

public void oldWay7() throws JMSException{
       Connection connection = connFact.createConnection();
       Session session = connection.createSession(Session.AUTO_ACKNOWLEDGE);
       String messageSelector="color = 'blue'";
       boolean noLocal=true;
       MessageConsumer consumer =
             session.createConsumer(destination,messageSelector,noLocal);
       connection.start();
       // do stuff with consumer

       connection.close();
}
```

Here's the same example using the new API:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
// No need for @JMSSession annotation in Java EE
@JMSDestination(lookup="jms/inboundQueue")
@JMSConsumer(messageSelector="color = 'blue'", noLocal=true, start=true)
MessageConsumer consumer;

public void newWay7(){
       // do stuff with consumer
}
```

In addition, the `@JMSTemporaryQueue` or `@JMSTemporaryTopic` annotations can be used instead of `@JMSDestination` to specify that the consumer will be on a temporary queue or topic. For more information see "Injecting a TemporaryQueue or TemporaryTopic" below.

Example 8: Creating a durable subscriber (Java EE)

Here's an example of creating a durable subscriber using the existing API:

```
@Resource(lookup = "jms/connFactory")
ConnectionFactory connFact;

@Resource(lookup="jms/inboundTopic")
Topic topic;

public void oldWay8() throws JMSException{
       Connection connection = connFact.createConnection();
       Session session = connection.createSession( Session.AUTO_ACKNOWLEDGE);
       String messageSelector="color = 'blue'";
       boolean noLocal=true;
       MessageConsumer consumer =
            session.createConsumer(topic,messageSelector,noLocal);
       String subscriptionName="mySub";
       consumer = session.createDurableSubscriber(
            topic, subscriptionName, messageSelector, noLocal);

       connection.start();
       // do stuff with consumer

       connection.close();
}
```

Here an example using the new API:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
// No need for @JMSSession annotation in Java EE
@JMSDestination(lookup="jms/inboundQueue")
@JMSConsumer(messageSelector="foo", noLocal=true, start=true)
@JMSDurableSubscriber(name="mySub")
MessageConsumer consumer;

public void newWay8(){
       // do stuff with consumer
}
```

Injecting a QueueBrowser

The current API provides the following two methods on `Session` to create a `QueueBrowser`:

```
createBrowser(Queue queue)

createBrowser(Queue queue, java.lang.String messageSelector)
```

The corresponding annotation to inject a `QueueBrowser` could be

```
@Inject
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@JMSDestination(lookup="jms/inboundQueue")
QueueBrowser queueBrowser;
```

or

```
@Inject
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@JMSDestination(lookup="jms/inboundQueue")
@JMSQueueBrowser(messageSelector="foo")
QueueBrowser queueBrowser;
```

Note that the @QueueBrowser annotation is only needed if a message selector is being specified.

- Do we need a separate @QueueBrowser annotation, or can we reuse the @JMSConsumer annotation? This also allows a message selector to be specified but also allows the noLocal and start attributes to be specified. Since these do not apply to queue browsers it might be best to have a separate @QueueBrowser annotation as well.

Injecting a destination (Java EE)

The preceding examples suggest that when injecting a producer or consumer, the information about the destination might be specified using a new @JMSDestination annotation:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
// No need for @JMSSession annotation in Java EE
@JMSDestination(lookup="jms/inboundQueue")
MessageProducer producer;
```

This requires the destination to be available in a JNDI repository. A possible extension might be to allow the destination's provider-specific name to be specified instead:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
// No need for @JMSSession annotation in Java EE
@JMSDestination(name="inboundQueue")
MessageProducer producer;
```

- Do we want to allow destinations to be specified using their provider-specific name, or should this omit this feature since it is contrary to good JMS practice?

The current API allows JMS destinations to be injected using standard @Resource annotations, so long as the destination object can be looked up in a JNDI repository.

```
@Resource(lookup="jms/inboundQueue")
Destination destination;
```

However we may wish to allow JMS destinations to be injected in the same way as other JMS resources, using the @Inject and @JMSDestination annotations, This might be particularly useful if the @JMSDestination annotation allowed a provider-specific JNDI name to be specified.

```
@Inject
@JMSDestination(name="inboundQueue")
Destination destination;
```

- Do we want to allow destinations to be injected using @Inject and @JMSDestination annotations, or should users be expected to use @Resource annotations?

Injecting a TemporaryQueue or TemporaryTopic

The current API provides two methods on `Session` to create temporary destinations:

```
createTemporaryQueue();

createTemporaryTopic();
```

The corresponding annotations to create a TemporaryQueue or TemporaryTopic could be:

```
@Inject
@JMSConnection(lookup="jms/connFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
TemporaryQueue temporaryQueue;
```

and

```
@Inject
@JMSConnection(lookup="jms/connFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
TemporaryTopic temporaryTopic;
```

Having created a temporary queue or topic, how would it be used? JMS specifies that a temporary queue or topic exists for the duration of the Connection used to create it, and can only be consumed by the Connection that created it.

This means that it should be possible to inject a TemporaryQueue, and a MessageConsumer on that temporary queue, and for these to use the same underlying connection. This could be achieved by annotating the injected message consumer with `@JMSTemporaryTopic` or `@TemporaryQueue`:

```
@Inject
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
TemporaryQueue temporaryQueue;

@Inject
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@JMSTemporaryQueue
MessageConsumer consumer;
```

Note that `@JMSTemporaryQueue` and `@JMSTemporaryTopic` would only be needed when injecting a consumer, not when injecting the temporary queue or topic itself. However some

means would be needed to link the two injected objects together to ensure that they relate to the same temporary destination.

This raises the following unresolved issue:

- When injecting both a temporary destination and a consumer on that destination, how does the user specify that these two objects should relate to the same temporary destination?

Use cases

Example 9: Sending a message (Java EE)

This example compares the old and new API for sending a `TextMessage` in a Java EE (EJB or web container) environment.

Here's how you might do this using the existing JMS 1.1 API.

```
@Resource(lookup = "jms/connFactory")
ConnectionFactory connFact;

@Resource(lookup="jms/inboundQueue")
Destination destination;

public void sendMessageOld(String payload) throws JMSException{
       Connection connection = connFact.createConnection();
       Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
       MessageProducer messageProducer = session.createProducer(destination);
       TextMessage textMessage = session.createTextMessage(payload);
       messageProducer.send(textMessage);

       connection.close();
}
```

Here's how you might do this using the new API.

```
@Inject
@JMSConnection(lookup="jms/connFactory")
// No need for @JMSSession annotation in Java EE
@JMSDestination(lookup="jms/inboundQueue")
MessageProducer producer;

@Inject
@JMSConnection(lookup="jms/connFactory")
// No need for @JMSSession annotation in Java EE
TextMessage textMessage;

public void sendMessageNew(String payload) throws JMSException{
       textMessage.setText(payload);
       producer.send(textMessage);
}
```

This example illustrates the following unresolved issues:

- (Repeated point) The use of injection has made it necessary to change the `MessageConsumer` and `TextMessage` objects from a local variable to an instance variable, despite the application only using them in a single method. See Example 3a for more observations on this issue.
- The JMS spec does not require the injected `MessageProducer` and `TextMessage` to be created using the same `Session` (this will be made explicit in issue JMS_SPEC-52). However it would be wasteful of resources to create a second `Session`, and especially a second `Connection`, so applications would need a way of specifying that the injected `MessageProducer` and `TextMessage` must be created using the same `Session`.

  In the special case of injected messages there will never be a requirement to inject messages using a different Session from that used to create the `MessageProducer`, so the default behavior could be to use any existing `Session` that was available. However in the rather special case where there was a choice of sessions from multiple JMS providers then it would be necessary to allow the application to specify the Session that should be used (since it may be less efficient to use one vendor's message implementation with another vendor's MessageProducer).
- (Repeated point) How can we avoid repeating the same annotations on multiple injected objects? For example, how can we avoid repeating identical `@JMSConnection` and `@JMSSession` annotations on every injected object (irrespective of whether the same or multiple `Connection` and `Session` objects will be used).

Example 10: Receiving a message synchronously (Java EE)

This example compares the old and new API for synchronously receiving a `TextMessage` in a Java EE (EJB or web container) environment.

Here's how you might do this using the existing JMS 1.1 API.

```
@Resource(lookup = "jms/connFactory")
ConnectionFactory connFact;

@Resource(lookup="jms/inboundQueue")
Destination destination;

public String receiveMessageOld() throws JMSException{
       Connection connection = connFact.createConnection();
       Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
       MessageConsumer messageProducer = session.createConsumer(destination);
       TextMessage textMessage=(TextMessage) messageProducer.receive();
       String payload = textMessage.getText();
       connection.close();
       return payload;
}
```

Here's how you might do this using the new API.

```
@Inject
@JMSConnection(lookup="jms/connFactory")
// No need for @JMSSession annotation in Java EE
@JMSDestination(lookup="jms/inboundQueue")
@JMSConsumer(start=true)
MessageConsumer consumer;

public String receivedMessageNew() throws JMSException{
       TextMessage textMessage=(TextMessage) consumer.receive();
       String payload = textMessage.getText();
       return payload;
}
```

This example illustrates the following unresolved issues:

- (Repeated point) The use of injection has made it necessary to change the MessageProducer and TextMessage objects from local variables to instance variables, despite the application only using them in a single method. See Example 3a for more observations on this issue.

Example 11: Receiving a message asynchronously (Java SE)

This example compares the old and new API for synchronously receiving `TextMessage` messages asynchronously in a Java SE environment.

Here's how you might do this using the existing JMS 1.1 API:

```
private void consumeAsync() throws Exception {
   InitialContext initialContext = ... // DETAILS OMITTED

   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("inboundQueue");

   initialContext.close();

   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
   MessageConsumer messageConsumer = session.createConsumer(inboundQueue);
   AsyncListener messageListener = new MyListener();
   messageConsumer.setMessageListener(messageListener);

   connection.start();
   synchronized (messageListener){
      messageListener.wait();
   }
   connection.close();
}

class MyListener implements MessageListener {

   int numMessagesReceived = 0;

   public void onMessage(Message message) {
      numMessagesReceived++;
      // PROCESS MESSAGE
       if (numMessagesReceived==10){
          synchronized (this){
          notify();
       }
   }
}
```

Here's how you might do this using the new API:

```
@Inject
// MUST USE SAME CONNECTION AS IS INJECTED BELOW
@JMSConnection(lookup="jms/connFactory")
@JMSSession(sessionMode=Session.AUTO_ACKNOWLEDGE)
@JMSDestination(lookup="jms/inboundQueue")
@JMSConsumer(start=false)
MessageConsumer consumer;

@Inject
@JMSConnection(lookup="jms/connFactory")
Connection connection;
```

```
private void consumeAsync() throws Exception {

   AsyncListener messageListener = new MyListener();
   messageConsumer.setMessageListener(messageListener);

   // note that connection is started after message listener is set
   connection.start();
   synchronized (messageListener){
      messageListener.wait();
   }

}

// MyListener class is the same as above
```

This example illustrates the following unresolved issues:

- (Repeated point) These annotations include the JNDI names of administered objects. However, whereas a Java EE environment comes with a ready-configured JNDI provider, a Java SE environment does not. This implies that the use of these annotations in a Java SE environment is dependent on a suitable JNDI provider being available and configured.
- (Repeated point) The use of injection has made it necessary to change the `MessageConsumer` and `Connection` objects from local variables to instance variables, despite the application only using them in a single method. See Example 3a for more observations on this issue.
- Applications would need a way of specifying that the injected `Connection` is the one that was used to create the injected `MessageConsumer`.
- The only reason we need to inject the Connection in this example is so that we may call `start()` after the call to `setMessageListener()`. Is there a way to avoid having to do this explicitly?
- (Repeated point) How can we avoid repeating the same annotations on multiple injected objects? In this example, how can we avoid repeating the `@JMSConnection` annotation on both injected objects? Although this is a general issue with this API, it is particularly important here since this example requires a single connection,, yet uses the same `@JMSConnection` annotation twice.

Example 12: Receiving a message asynchronously (Java EE)

The Java EE 6 and EJB specifications do not allow applications (other than those running in the application client container) to define their own `MessageListener` objects and register them with a producer using `producer.setMessageListener`. Applications wishing to consume messages asynchronously are expected to use MDBs. MDBs are configured declaratively rather than via Java code, without needing to explicitly create any JMS objects that might be injected such as connections and sessions.

For this reason, this document does not need to consider the case where, in a Java EE application, messages are received asynchronously.

Example 13: Receive synchronously and send a message in the same local transaction (Java SE only)

This example considers the use case in which a Java SE application repeatedly consumes a message from one queue and forwards it to another queue. Each message is received and forwarded in the same local transaction. This means that:

- The `MessageConsumer` and `MessageProducer` must both be created using the same Session
- After each message is sent transaction must be committed by calling the `commit()` method on the `Session`. This means the `Session` used to create the `MessageConsumer` and `MessageProducer` also needs to be available via injection.

Note that local transactions are not permitted in a Java EE EJB or web container, but are allowed in a Java SE environment or in the application client container. This particular example is for a Java SE environment where JMS administered objects must be obtained from JNDI by explicitly configuring a JNDI `InitialContext` and performing a lookup.

Here's how you might do this using the existing JMS 1.1 API. To keep the example short it consumes the incoming message synchronously. However since this is Java SE the message could also be consumed asynchronously using a `MessageListener`.

```
private void execute() throws JMSException, NamingException {

   InitialContext initialContext = ... // DETAILS OMITTED

   ConnectionFactory connectionFactory =
      (ConnectionFactory) initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue = (Queue) initialContext.lookup("jms/inboundQueue");
   Queue outboundQueue = (Queue) initialContext.lookup("jms/outboundQueue");

   initialContext.close();

   Connection connection = connectionFactory.createConnection();
   Session session = connection.createSession(true, Session.SESSION_TRANSACTED);
   MessageConsumer messageConsumer = session.createConsumer(inboundQueue);
   MessageProducer messageProducer = session.createProducer(outboundQueue);

   connection.start();
   TextMessage textMessage = null;
   do {
      textMessage = (TextMessage) messageConsumer.receive(1000);
      if (textMessage!=null){
         messageProducer.send(textMessage);
         session.commit();       }
      } while (textMessage!=null);
      connection.close();
   }
```

Here's how the same example might look when using the new API:

```
@Inject
// MUST USE SAME SESSION AS IS INJECTED BELOW
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.TRANSACTED)
@JMSDestination(lookup="jms/inboundQueue")
@JMSConsumer(start=true)
MessageConsumer messageConsumer;

@Inject
// MUST USE SAME SESSION AS IS INJECTED BELOW
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.TRANSACTED)
@JMSDestination(lookup="jms/outboundQueue")
MessageProducer messageProducer;

@Inject
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.TRANSACTED)
Session session;

public void execute(){

   TextMessage textMessage = null;
   do {
      textMessage = (TextMessage) messageConsumer.receive(1000);
      if (textMessage!=null){
         messageProducer.send(textMessage);
         session.commit();       }
      } while (textMessage!=null);
   }
}
```

This example illustrates the following problems:

- (Repeated point) These annotations include the JNDI names of administered objects. However, whereas a Java EE environment comes with a ready-configured JNDI provider, a java SE environment does not. This implies that the use of these annotations in a Java SE environment is dependent on a suitable JNDI provider being available and configured.
- (Repeated point) The use of injection has made it necessary to change the `MessageConsumer, MessageProducer` and `Session` objects from local variables to instance variables, despite the application only using them in a single method. See Example 3a for more observations on this issue.
- Applications would need a way of specifying that the injected `MessageProducer` and `MessageConsumer` must be created using the same `Session`.
- Applications would need a way of specifying that the injected `Session` is the one that was used to create the injected `MessageProducer` and `MessageConsumer`.
- (Repeated point) How can we avoid repeating the same annotations on multiple injected objects? For example, how can we avoid repeating the `@JMSConnection` and

`@JMSSession` annotations on every injected object? Although this is a general issue with this API, it is particularly important here since this example requires a single connection and session, yet uses the same `@JMSConnection` and `@JMSSession` annotations three times.

Example 14: Request/reply pattern using a TemporaryQueue (Java EE)

This example considers how a request/reply pattern might be implemented in Java EE, using the existing JMS API. In the code below, a method in a session bean (the requestor) sends a request message to some queue (the request queue). The setJMSReplyTo property of the request message is set to a TemporaryQueue, to which the reply should be set. After sending the request, the session bean listens on the temporary queue until it receives the reply.

Since the request message won't actually be sent until the transaction is committed, the request message is sent in a separate transaction from that used to receive the reply.

The details of the responder are omitted here. Typically this will be a MDB which receives the request message, extracts the TemporaryQueue from the setJMSReplyTo property and sends the response to it.

Here's how you might implement the requestor this using the existing JMS 1.1 API.

```
@Stateless
public class RequestorSessionBean implements RequestorSessionBeanLocal {

   @Resource(lookup = "jms/connectionFactory")
   ConnectionFactory connFact;

   @Resource(lookup="jms/requestQueue")
   Destination requestQueue;

   @EJB
   private RequestorSessionBeanLocal requestorSessionBean;

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public String requestReply(String request) throws JMSException {

      Connection connection = connFact.createConnection();
      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

      TemporaryQueue temporaryReplyQueue = session.createTemporaryQueue();

      // send request in a separate transaction
      requestorSessionBean.sendRequest(request,temporaryReplyQueue);

      // now receive the reply,
      // using the same connection as was used to create the temporary reply queue
      MessageConsumer consumer = session.createConsumer(temporaryReplyQueue);
      connection.start();
      TextMessage reply = (TextMessage) consumer.receive();
      String replyString=reply.getText();
      connection.close();
      return replyString;
    }

   @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
   public void sendRequest(String requestString,
         TemporaryQueue temporaryReplyQueue) throws JMSException {

      Connection connection = connFact.createConnection();
      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      TextMessage requestMessage = session.createTextMessage(requestString);
      requestMessage.setJMSReplyTo(temporaryReplyQueue);
      MessageProducer messageProducer = session.createProducer(requestQueue);
      messageProducer.send(requestMessage);
      connection.close();
   }
}
```

When implementing this pattern, the following features of JMS must be borne in mind:

- The same `Connection` object that was used to create the `TemporaryQueue` must also be used to consume the response message from it. (This is a restriction of temporary queues).
- If the request message is sent in a transaction then the response message must be consumed in a separate transaction. That's why the message is sent in a separate business which has the transactional attribute `REQUIRES_NEW`.

Here's how the same example might look when using the new API:

```
@Stateless
public class RequestorSessionBean implements RequestorSessionBeanLocal {

   // temporary reply queue
   // USED BY requestReply METHOD ONLY
   // MUST USE SAME CONNECTION AS MESSAGE CONSUMER BELOW
   @Inject
   @JMSConnection(lookup="jms/connectionFactory")
   // No need for @JMSSession annotation in Java EE
   TemporaryQueue temporaryReplyQueue;

   // consumer used to receive replies
   // USED BY requestReply METHOD ONLY
   // MUST USE SAME CONNECTION AS TEMPORARY QUEUE ABOVE
   // MUST USE SAME TEMPORARY QUEUE AS IS INJECTED ABOVE
   @Inject
   @JMSConnection(lookup="jms/connectionFactory")
   // No need for @JMSSession annotation in Java EE
   @JMSTemporaryQueue
   MessageConsumer messageConsumer;

   // request message
   // USED BY sendRequest METHOD ONLY
   @Inject
   @JMSConnection(lookup="jms/connectionFactory")
   // No need for @JMSSession annotation in Java EE
   TextMessage requestMessage;

   // producer used to send the request
   // USED BY sendRequest METHOD ONLY
   // HAS NO ASSOCIATION WITH ANY OTHER INJECTED OBJECT
   @Inject
   @JMSConnection(lookup="jms/connectionFactory")
   // No need for @JMSSession annotation in Java EE
   @JMSDestination(lookup="jms/requestQueue")
   MessageProducer producer;

   @EJB
   private RequestorSessionBeanLocal requestorSessionBean;

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public String requestReply(String request) throws JMSException {

      // send request in a separate transaction
      requestorSessionBean.sendRequest(request,temporaryReplyQueue);

      // now receive the reply
      TextMessage replyMessage = (TextMessage) messageConsumer.receive();
      return replyMessage.getText();
    }

   @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
   public void sendRequest(String requestString,
         TemporaryQueue temporaryReplyQueue) throws JMSException {

      requestMessage.setText(requestString);
```

```
        requestMessage.setJMSReplyTo(temporaryReplyQueue);
        messageProducer.send(requestMessage);
    }
}
```

This example illustrates the following unresolved issues:

- (Repeated point) The use of injection has made it necessary to change the `TemporaryQueue`, `MessageConsumer`,. `TextMessage` and `MessageProducer` objects from local variables to instance variables, despite the application only using the first two in the `requestReply` method and the last two in the `sendRequest` method. The use of instance variables therefore loses this association of each such object with a specific method. See Example 3a for more observations on this issue.
- (Repeated point) The injected `TextMessage` does not need to be created using a particular `Session` (this will be made explicit in issue JMS_SPEC-52), though to avoid wasting resources is a suitable `Session` is available then this should be used. See Example 9 for more observations on this issue.
- Applications would need a way of specifying that the injected `MessageConsumer` uses the same `TemporaryQueue` as is injected., and that both use the same underlying `Connection` (which is not injected).
- (Repeated point) How can we avoid repeating the same annotations on multiple injected objects? For example, how can we avoid repeating the `@JMSConnection` and `@JMSSession` annotations on every injected object?

Example 15: Sending messages in a bean-managed transaction (Java EE)

This example considers the case in which a session bean sends several messages in a single bean-managed transaction.

Here's how you might implement this using the existing JMS 1.1 API:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class MessageSenderBean implements MessageSenderBeanLocal {

   @Resource(lookup = "jms/connectionFactory")
   ConnectionFactory connFact;

   @Resource(lookup="jms/inboundQueue")
   Destination queue;

   @Resource
   UserTransaction userTransaction;

   public void sendMessages() throws Exception {

      userTransaction.begin();
      Connection connection = connFact.createConnection();
      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      MessageProducer messageProducer = session.createProducer(queue);

      for (int i = 0; i < 10; i++) {
         TextMessage textMessage = session.createTextMessage();
         textMessage.setText("Message "+i);
         messageProducer.send(textMessage);
      }
      userTransaction.commit();;
   }
}
```

Here's how the same example might look when using the new API.

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class MessageSenderBean implements MessageSenderBeanLocal {

   @Inject
   @JMSConnection(lookup="jms/connectionFactory")
   No need for @JMSSession annotation in Java EE
   TextMessage requestMessage;

   @Inject
   @JMSConnection(lookup="jms/connectionFactory")
   No need for @JMSSession annotation in Java EE
   @JMSDestination(lookup="jms/inboundQueue")
   MessageProducer producer;

   @Resource
   UserTransaction userTransaction;
```

```
   public void sendMessages() throws Exception {

      userTransaction.begin();
      for (int i = 0; i < 10; i++) {
         textMessage.setText("Message "+i);
         messageProducer.send(textMessage);
      }
      userTransaction.commit();;
   }
}
```

This example illustrates the following unresolved issues:

- Since this method uses a bean-managed transaction, the underlying Session object can be created using the proposed new method `Connection.createSession()` with no arguments. There is therefore to provide a @JMSSession annotation to specify a sessionMode. However since that method is only valid after the transaction has started, the runtime must not attempt to create the `Session` until after the call to `UserTransaction.begin()`.
- (Repeated point) The use of injection has made it necessary to change the `MessageConsumer` and `TextMessage` objects from a local variable to an instance variable, despite the application only using them in a single method. See Example 3a for more observations on this issue.
- There is an additional issue regarding the injected `TextMessage`. In the JMS 1.1 version, a new `TextMessage` is created for each iteration round the loop. However in the proposed new version, the same `TextMessage` will be used repeatedly unless some way is devised to allow the injected message to be scoped to the `for` block.
- (Repeated point) The injected `TextMessage` does not need to be created using a particular `Session` (this will be made explicit in issue JMS_SPEC-52), though to avoid wasting resources is a suitable `Session` is available then this should be used. See Example 9 for more observations on this issue.
- (Repeated point) How can we avoid repeating the same annotations on multiple injected objects? For example, how can we avoid repeating identical `@JMSConnection` annotations on every injected object (irrespective of whether the same or multiple `Connection` objects will be used).

Summary of annotations

The following table lists the objects that can be injected, and the annotations that may be specified when injecting each type of object:.

Object being injected	Available annotations

Connection	@JMSConnection			
Session	@JMSConnection	@JMSSession		
Message Producer	@JMSConnection	@JMSSession	@JMSDestination	
Message Consumer	@JMSConnection	@JMSSession	@JMSDestination or @JMSTemporaryQueue or @JMSTemporaryTopic	@JMSConsumer and @JMSDurable Subscriber
Queue Browser	@JMSConnection	@JMSSession	@JMSDestination	@JMSQueueBrowser
Temporary Queue	@JMSConnection	@JMSSession		
Temporary Topic	@JMSConnection	@JMSSession		
Message	@JMSConnection	@JMSSession		
Object Message	@JMSConnection	@JMSSession		
Stream Message	@JMSConnection	@JMSSession		
Bytes Message	@JMSConnection	@JMSSession		
Map Message	@JMSConnection	@JMSSession		
Text Message	@JMSConnection	@JMSSession		

The following table lists the various annotations and their parameters

Annotation	Attributes	Meaning
@JMSConnection	lookup	JNDI name of connection factory
	name	User name (optional)
	password	Password (optional)

`@JMSSession`	`sessionMode`	One of `Session.TRANSACTED`, `Session.AUTO_ACKNOWLEDGE`, `Session.CLIENT_ACKNOWLEDGE`, and `Session.DUPS_OK_ACKNOWLEDGE` (all are optional. If sessionMode or @JMSSession omitted, default is auto-ack except in a Java EE transaction, when that transaction is used)
`@JMSDestination`	`lookup`	JNDI name of destination
	`name`	Provider-specific name of destination. Only allowed if lookup not supplied.
`@JMSTemporaryQueue`	no attributes	
`@JMSTemporaryTopic`	no attributes	
`@JMSConsumer`	`messageSelector`	(Optional) Message selector
	`noLocal`	(optional) If set, inhibits the delivery of messages published by its own connection
	`start`	(optional) If set, causes `connection.start()` to be called.
`@QueueBrowser`	`messageSelector`	Message selector
`@JMSDurableSubscription`	`name`	The name used to identify this durable subscription

Unresolved issues

Major unresolved issues

The relationship between injected objects

This document leaves unresolved the issue how the relationship between injected objects (e.g. sessions and connections) will be managed, and what facilities will be offered to applications to control the relationship between such objects.

The simplest case is when we inject two objects, one of which is dependent on the other., or injecting three objects, two of which are dependent on the third.

There's an example of this in "Example 13: Receive synchronously and send a message in the same local transaction (Java SE only)" on page 23. In this example it is essential that the injected producer and consumer objects are created using the injected session object.

```
@Inject
// MUST USE SAME SESSION AS IS INJECTED BELOW
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.TRANSACTED)
@JMSDestination(lookup="jms/inboundQueue")
@JMSConsumer(start=true)
MessageConsumer messageConsumer;

@Inject
// MUST USE SAME SESSION AS IS INJECTED BELOW
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.TRANSACTED)
@JMSDestination(lookup="jms/outboundQueue")
MessageProducer messageProducer;

@Inject
@JMSConnection(lookup="jms/connectionFactory")
@JMSSession(sessionMode=Session.TRANSACTED)
Session session;
```

We need to devise a way:

- To specify that messageConsumer and messageProducer are dependent on session
- To avoid having to repeat the @JMSConnection and @JMSSession annotations on both messageConsumer and messageProducer, since they are already defined on the connection.

A second and more complex case is when we inject two objects, both of which are dependent on a third, which is not injected. There's an example of this in "Example 14: Request/reply pattern using a TemporaryQueue (Java EE)" in page 26. In this example, a temporary queue and a consumer are injected. It is essential that these are created using the same underlying connection (which is not explicitly injected).

```
   // temporary reply queue
   // USED BY requestReply METHOD ONLY
   // MUST USE SAME CONNECTION AS MESSAGE CONSUMER BELOW
   @Inject
   @JMSConnection(lookup="jms/connectionFactory")
   // No need for @JMSSession annotation in Java EE
   TemporaryQueue temporaryReplyQueue;

   // consumer used to receive replies
   // USED BY requestReply METHOD ONLY
   // MUST USE SAME CONNECTION AS TEMPORARY QUEUE ABOVE
   // MUST USE SAME TEMPORARY QUEUE AS IS INJECTED ABOVE
   @Inject
   @JMSConnection(lookup="jms/connectionFactory")
   // No need for @JMSSession annotation in Java EE
   @JMSTemporaryQueue
   MessageConsumer messageConsumer;
```

We need to devise a way:

- To specify that temporaryReplyQueue and messageConsumer are dependent on the same Connection, even though it is not injected.
- To avoid repeating the @JMSConnection annotation on both temporaryReplyQueue and messageConsumer

However this might not be possible to avoid repeating the @JMSConnection annotation without injecting the Connection. Similarly it might not be possible to avoid repeating the @JMSSession annotation injecting the Session.

Finally, there's a third case which is the opposite of the previous two: when we wish to ensure that two injected objects do *not* use the same dependent objects. So can we inject, say, two Session objects and be sure that they will use *different* underlying connections?

Avoiding repetition on annotations

This is a slightly different topic to that discussed immediately above. The issue is: if multiple unrelated objects are injected which have identical @JMSConnection or @JMSSession (or other) annotations, is there a way to avoid repeating the same annotation?

Consider this extract from "Example 15: Sending messages in a bean-managed transaction (Java EE)" on page 29:

```
@Inject
@JMSConnection(lookup="jms/connectionFactory")
TextMessage requestMessage;

@Inject
@JMSConnection(lookup="jms/connectionFactory")
@JMSDestination(lookup="jms/inboundQueue")
MessageProducer producer;
```

In this particular example, the same @JMSConnection annotation is specified for both injected objects. Strictly speaking this is correct: both objects are dependent on a connection factory. However given that the annotations are identical, is there a way to avoid having to repeat it?

Now let us consider for the moment that we don't want these two injected objects to use the same underlying session and connection objects. However we do want them to use the same connection factory. How can we avoid having to specify the same @JMSConnection annotation in both places?

(Note that a connection factory can itself be injected using an old-style @Resource annotation. I don't know whether this helps).

Injected objects cannot be local variables

This document assumes that because of the nature of dependency injection, injected objects must be declared as instance variables even though they are only used by a single method and should really be declared as local variables in a method.

This breaks the encapsulation of the method. A good example of this can be seen in “Example 14: Request/reply pattern using a TemporaryQueue (Java EE)” on page 26, in which some injected variables are used only by one method whilst some other injected variables are used only by another, but since they are all declared as instance variables this encapsulation is lost.

In addition, declaring injected JMS objects as instance variables introduces the danger of these objects being inadvertently used concurrently from multiple threads even though in most cases this is often neither permitted nor safe.

Scope of injected variables

This document also does not attempt to resolve the issue of what scope the injected objects should have.

The examples in this document assume that they come into scope when the method is entered and that they go out of scope when they have it, though it is not clear whether this would be possible to implement.

However, “Example 15: Sending messages in a bean-managed transaction (Java EE)” on page 29 includes an example of an injected object that should really have a lower scope than this: the scope of a `for` block.

Here are some possible scoping options: (this list contributed by Reza):

- @ApplicationScoped - doesn't fit the JMS model very well at all.
- @SessionScoped - HTTP bound and does not fit the JMS model well.
- @Dependent - not scalable, I don't think it is a real candidate. However, maybe we can think about a parallel to the JPA extended persistence context that is only allowed in certain cases.
- @RequestScoped - might work, but the issue is that's it is all-or-nothing. There's no way to have multiple JMS sessions in a given request, for example, even in nested EJB calls.
- @ThreadScoped - Has the same problem as request scoped.
- @ConversationScoped - Might be workable but puts more work on the developer for all cases. Also very similar to the BMT model using transaction scope.
- @TransactionScoped - Works like the JPA transactional persistence context. The down-side is that it is JTA transaction bound, just like the JPA container managed entity manager (vs. the application managed entity manager which can use local transactions). We can think about having a "best effort" mode when a transaction is not present to try to simulate "auto-commit"/stateless mode.
- @MethodLocalScoped - The downside is that life-cycle granularity is limited to a method (unlike the transaction scope). Would also definitely need to be standardized via CDI because it requires changes in the behavior of the EJB, Servlet and all other managed beans to add additional built-in call-backs to CDI on method entry/exit.

Java SE Support for JNDI

The @JMSConnection and @JMSDestination annotations defined in this document allow the JNDI names of administered objects to be specified.

However, whereas a Java EE environment comes with a ready-configured JNDI provider, a Java SE environment does not.

This implies that the use of these annotations in a Java SE environment is dependent on a suitable JNDI provider being available and configured.

Minor unresolved issues

`@JMSConsumer`: what is the default value of the `start` attribute?

It is proposed that the `@JMSConsumer` annotation has an attribute `start`.

- If set to `false`, the underlying connection will not be automatically started. Instead the application must inject the `Connection` and call `start()` explicitly.
- If set to `true`, its underlying `Connection` will be automatically started (if not already started) as soon as the first injected consumer to use that connection is created, or earlier. This avoids the need for the application to call `connection.start()` and probably avoids the need to inject the connection at all

What should be the default value of the `start` attribute?

A connection needs to be started before messages will be delivered to a consumer on that connection. The main reason users might not want `start()` to be called automatically is when setting up an async `MessageListener`. In this case they might wish to defer the call to `connection.start()` until after they have called `messageConsumer.setMessageListener`. (This is discussed in section 4.3.3 "Connection Setup" in the JMS 1.1 specification)..

Since `messageConsumer.setMessageListener` is not permitted in Java EE, it might be approrpiate to have `start=true` as default in Java EE, and `start=false` as default in Java SE. However this might be confusing.

Using user/password authentication with an injected connection

If a connection is injected with the user and password attributes

```
@Inject @JMSConnection (lookup="jms/connFactory", user="admin", password="secret")
Connection connection2;
```

then since the connection must be an instance variable the user and password cannot be set at runtime. This is inflexible and limits the usefulness of this feature.

This is probably not an issue with Java EE, where user and password authentication is not particularly useful, but it may be with Java SE.

Temporary Destinations

There's a specific issue regarding temporary destinations. When injecting both a temporary destination and a consumer on that destination, how does the user specify that these two objects should relate to the same temporary destination?

This is described in more detail in "Injecting a TemporaryQueue or TemporaryTopic" on page 17.

JMS objects which are do not require injection

This section lists some other JMS objects which were considered to not require injection.

Domain-specific interfaces proposed for removal

The following domain-specific interfaces are (in JMS 2.0) proposed for removal and so it is unnecessary and probably inappropriate, to allow them to be injected:

```
QueueConnection
QueueConnectionFactory

QueueReceiver
QueueSender
QueueSession

TopicConnection
TopicConnectionFactory

TopicPublisher
TopicSession
TopicSubscriber
```

Application server interfaces

The following interfaces are not for direct use by applications and so it is not appropriate to allow them to be injected. Six of them are also proposed for removal.

```
ConnectionConsumer
ServerSession
ServerSessionPool

XAConnection
XAConnectionFactory
```

`XAQueueConnection`
`XAQueueConnectionFactory`
`XAQueueSession`
`XASession`
`XATopicConnection`
`XATopicConnectionFactory`
`XATopicSession`

JMS objects which are unsuitable for injection

This section lists some other JMS objects which were considered unsuitable for injection.

ConnectionMetaData

`ConnectionMetaData`: can be created by calling `connection.getMetaData()`. It holds static information about the JMS provider being used. Creating one doesn't use up system resources so instances can be created without limit, and don't need to be closed or cleaned up after use. There is no benefit on allowing these to be injected.

ExceptionListener

`ExceptionListener`: The implementation of this interface is provided by the application. JMS doesn't define how instances are instantiated: it is up to the application to define how an instance is obtained and what its scope is (i.e. whether each connection requires a different `ExceptionListener` instance or whether they all use the same `ExceptionListener` instance). It is also up to the application to define whether instances need closing or cleaning up after use. Because this is an application object, it is not appropriate to attempt to add features to JMS to allow it to be managed.

MessageListener

`MessageListener`: The implementation of this interface is provided by the application.
In Java EE, these are provided as MDBs, whose implementation is provided by the application. The EJB spec already defines a declarative API which defines how they are instantiated, associated with consumers, and cleaned up after use which should be adequate at least for the first version of this new API.

In Java SE, JMS doesn't define how instances are instantiated: it is up to the application to define how an instance is obtained and what its scope is (i.e. whether each consumer requires a different `MessageListener` instance or whether they all use the same `MessageListener` instance). It is also up to the application to define whether instances need closing or cleaning up after use. Because this is an application object, it is not appropriate to attempt to add features to JMS to allow it to be managed.

DeliveryMode

`DeliveryMode` : This interface is never instantiated and so is unsuitable for injection.