JMS 2.0: Support for Multi-tenancy

About this document

This document contains proposals on how multi-tenancy might be supported in JMS 2.0.

- It reviews the Java EE 7 proposals for resource configuration metadata in respect of JMS resources.
- It reviews the Java EE 7 proposals for multi-tenancy and discusses the three ways in which the destinations used by different instances of a multi-tenant application might be isolated from one another.
- It proposes extensions to the `@JMSDestinationDefinition` and `@JMSConnectionFactoryDefinition` annotations (and their XML equivalents) to allow the application to specify the tenant isolation level of a given destination to be either "shared" or "isolated".
- It proposes extensions to the `javax.jms.Destination` and `javax.jms.ConnectionFactory` interfaces to allow a deployer to specify the isolation level to be used with these administered objects.
- If suggests four alternative means by which an application server could pass the tenantId of a running application to the JMS client. Two of these options would require changes to the JCA spec

Multi-tenancy in Java EE

Java EE 7 extends the existing Java EE application model to support PaaS and multi-tenancy. This is discussed in the document "The Java EE 7 Platform and Support for the PaaS Model" at http://java.net/downloads/javaee-spec/PaaS.pdf

The basic idea described in that document is to allow the same PaaS application to be made available in a PaaS environment for use by multiple tenants. Each tenant will then use a different instance (or set of instances) of the application.

There are two stages in making a PaaS application available to a particular tenant:

- The application is submitted to the PaaS environment (this step only has to be done once)
- The application is customized and deployed for a specific tenant.

Instances of the application used by different tenants are isolated from one another. The resources of the application (including JMS resources) may also need to also be isolated from one another so that each tenant uses a separate set of resources. Not all resources may need to be isolated in this way: some resources may be common to all tenants and so can be thought of as shared.

Resource configuration in Java EE 6 and 7

Before discussing multi-tenancy any further, let's review the new feature which is being added to the Java EE platform to assist in the resource provisioning.

Let's look at a simple example of a session bean which sends a message to a queue:

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

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

public void sendMessage (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();
}
```

See Figure 1 "Resource configuration for a simple JMS application" at the end of this section.

In **Java EE 6**, the deployer has the responsibility of provisioning a suitable connection factory resource and binding it to `jms/connFactory`, and provisioning a suitable destination resource and binding it to `jms/inboundQueue`.

In more detail, the deployer has the following responsibilities:

1. Ensuring that the physical resources that are needed exist and are available. In this particular case this means that a JMS server needs to be available which contains the required queue. The deployer will not necessarily be the person who provisions these physical resources: it may be necessary for the deployer to delegate this task to the administrator of the server in question. However the deployer still needs to be sure that the required physical resources exist before they can proceed with the deployment.

2. Creating connection factory and destination objects in the application server's JNDI store which represent these physical resources. In this case this means
    a. creating a ConnectionFactory object, configuring it to use the chosen JMS server, and binding it in the application server's JNDI store at `jms/connFactory`
    b. creating a Queue object, configuring it to refer to the chosen physical queue, and binding it in the application server's JNDI store at `jms/inboundQueue`.

The deployer is given no additional information to help the deployer know what resources are needed by the application and how they should be configured.

In **Java EE 7** the application will be able to provide additional metadata which makes it easier for the deployer to know what is required. The goal is to allow the creation of PaaS providers

which can perform this activity automatically, at least in the case when the required resources are provided by the PaaS environment.

An application will be able to specify the resources it requires by providing **resource definition metadata**, either by adding annotations to the source code or by providing an XML equivalent. For JMS the annotations will be `@JMSConnectionFactoryDefinition` and `@JMSDestinatonDefinition`.

Essentially, for every connection factory or destination JNDI resource referenced by the application, there may be a `@JMSConnectionFactoryDefinition` or `@JMSDestinatonDefinition` annotation, or an XML equivalent. These contain information which can be used at the deployment stage to provision the required resources. The deployer may be able to ignore or override this information.

So what will this metadata look like? For the example code above, it would consist of a `@JMSConnectionFactoryDefinition` annotation and a `@JMSDestinationDefinition` annotation.

Here's an example of the resource metadata for defining the JMS connection factory used in the above code example:

```
@JMSConnectionFactoryDefinition {
    name="jms/connFactory",
    className="javax.jms.ConnectionFactory”,
    initialPoolSize="32",
    maxPoolSize="64"
}
```

(There's also an XML equivalent not discussed here).

This annotation specifies that a connection factory needs to be provisioned at `jms/connfact`. In this particular case the annotation doesn't define what JMS provider will be used or how to connect to it, so the container's built-in JMS server will be used. However it does define information about how the connections will be pooled: the initial connection pool size will be 32 and the maximum connection pool size will be 64.

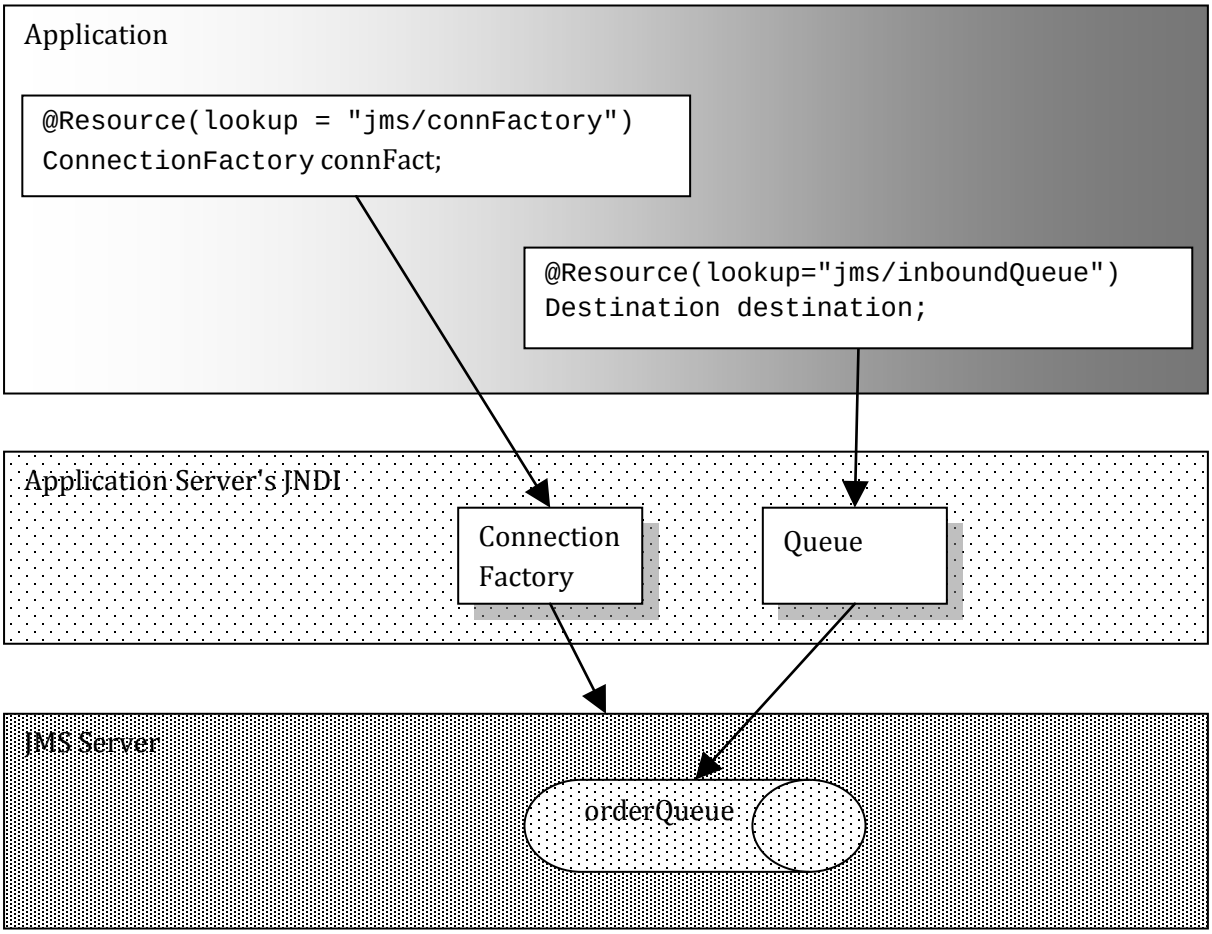
Here's an example of the resource metadata for defining the JMS destination used in the above code example:

```
@JMSDestinationDefinition {
    description=“MDB input queue”
    name="ims/inboundQueue",
    className="javax.jms.Queue",
    resourceName="orderQueue"
}
```

This annotation specifies that the destination needs to be provided at `jms/inboundQueue`. In this particular case the destination is a queue, and should have the name `orderQueue`.

The following diagram summarises the resulting configuration. This diagram will be extended in future sections where we discuss how this application might be used by multiple-tenants.

Resource configuration for a simple JMS application



Resource configuration for a multi-tenant application

Now let's consider how JMS connection factories and destinations might need to be configured differently when a multi-tenant application is for a specific tenant.

A tenant is essentially a unit of isolation. The first thing to consider is the degree of isolation required for each destination used by an application. There are two options.

1. All tenants see the same destination. An example might be a portfolio management application where all tenants subscribe to the same centralized market data feed. We will refer to these as "**shared**" destinations.

2. Each tenant sees a completely separate destination, so tenants do not receive one another's messages. An example may be a queue used to dispatch work asynchronously between two components of the same application. We will refer to these as "**isolated**" destinations.

A "shared" destination is essentially the same as a destination in a non-multi-tenant environment. Just a normal destination.

A "isolated" destination is something new. It could be achieved in several ways:

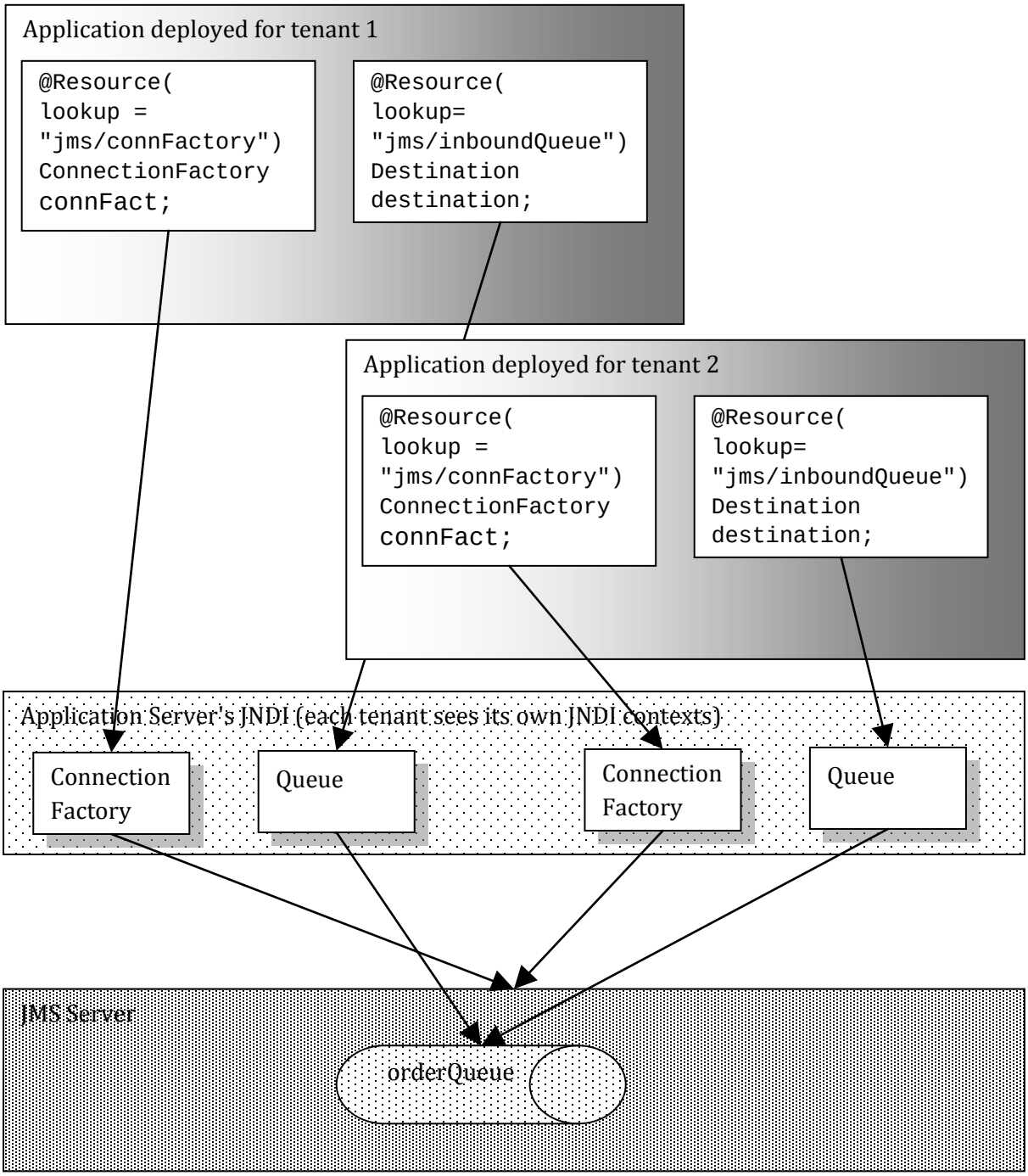
1. Each tenant application uses a completely different JMS server. This gives the highest degree of isolation but requires the highest amount of resources. It does not require the JMS server to be aware that the application is multi-tenant. Let's call this "**isolation using multiple providers**".
2. Each tenant application uses the same JMS server but different destinations for each tenant. This gives an intermediate level of isolation with an intermediate use of resources. The deployer will be responsible for provisioning administered objects to ensure that that each tenant uses a different destination. It does not require the JMS server to be aware that the application is multi-tenant. Let's call this "**isolation using multiple destinations**".
3. Each tenant application uses the same JMS server and the same destinations for each tenant., with the JMS provider isolating each the different tenant applications using the same destination. Te degree of isolation this provides would depend on the JMS provider, but it might potentially require the lowest level of resources. Inevitably the JMS server would need to be aware that the application is multi-tenant, and would need to be able to identify which tenant application is using it for a given operation. Let's call this "**isolation using a multi-tenant destination**".

Let's consider JMS destinations (queues and topics) initially. The application itself will refer to a particular destination using a particular JNDI name. When this application is provisioned for a specific tenant, an appropriate destination resource needs to be created and bound to that JNDI name in the JNDI namespace for that tenant.

These four options are represented by the following diagrams:

Shared destination

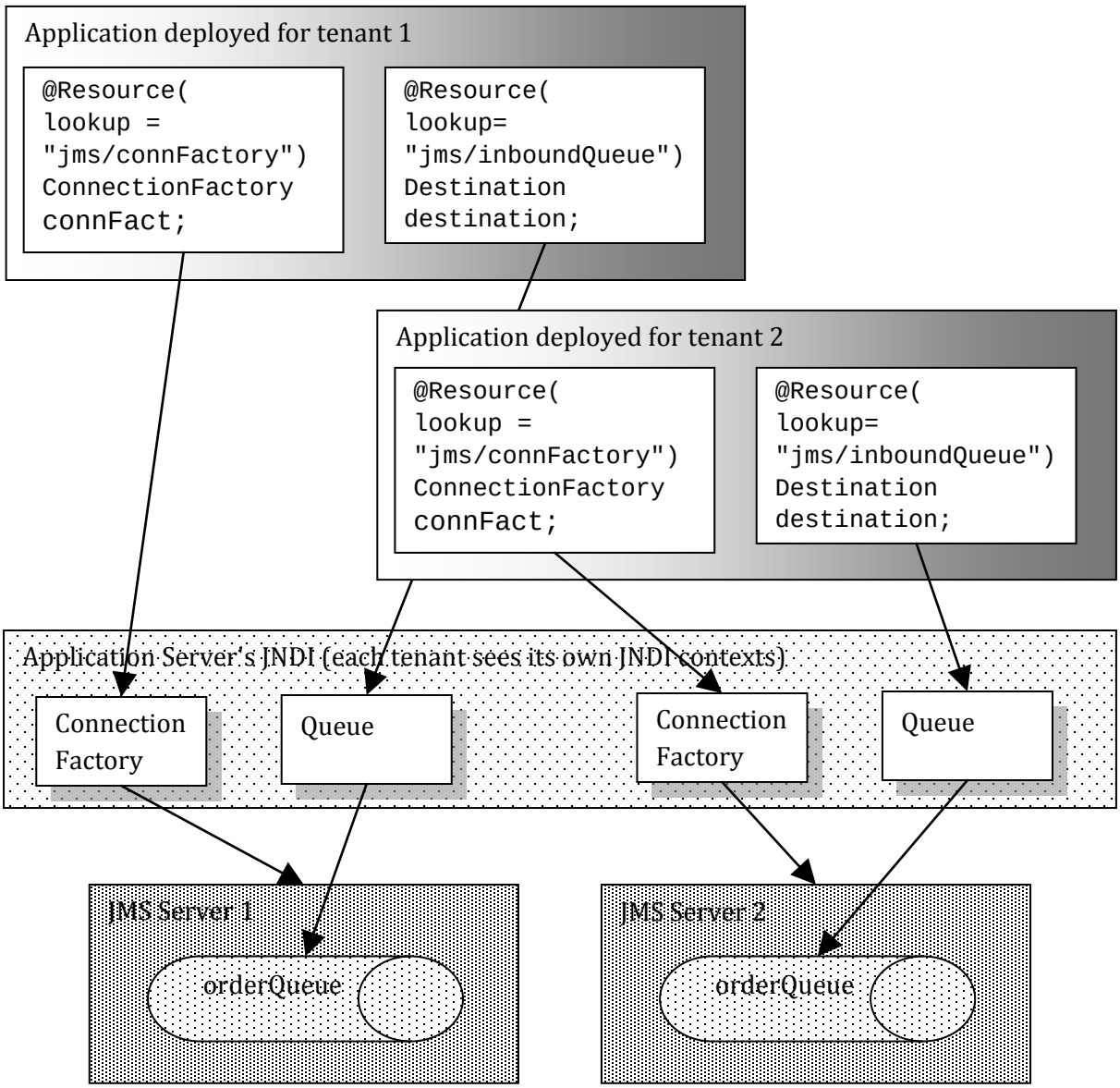
All tenants share the same destination:



In this example, both application instances see exactly the same destination.

Isolation using multiple providers

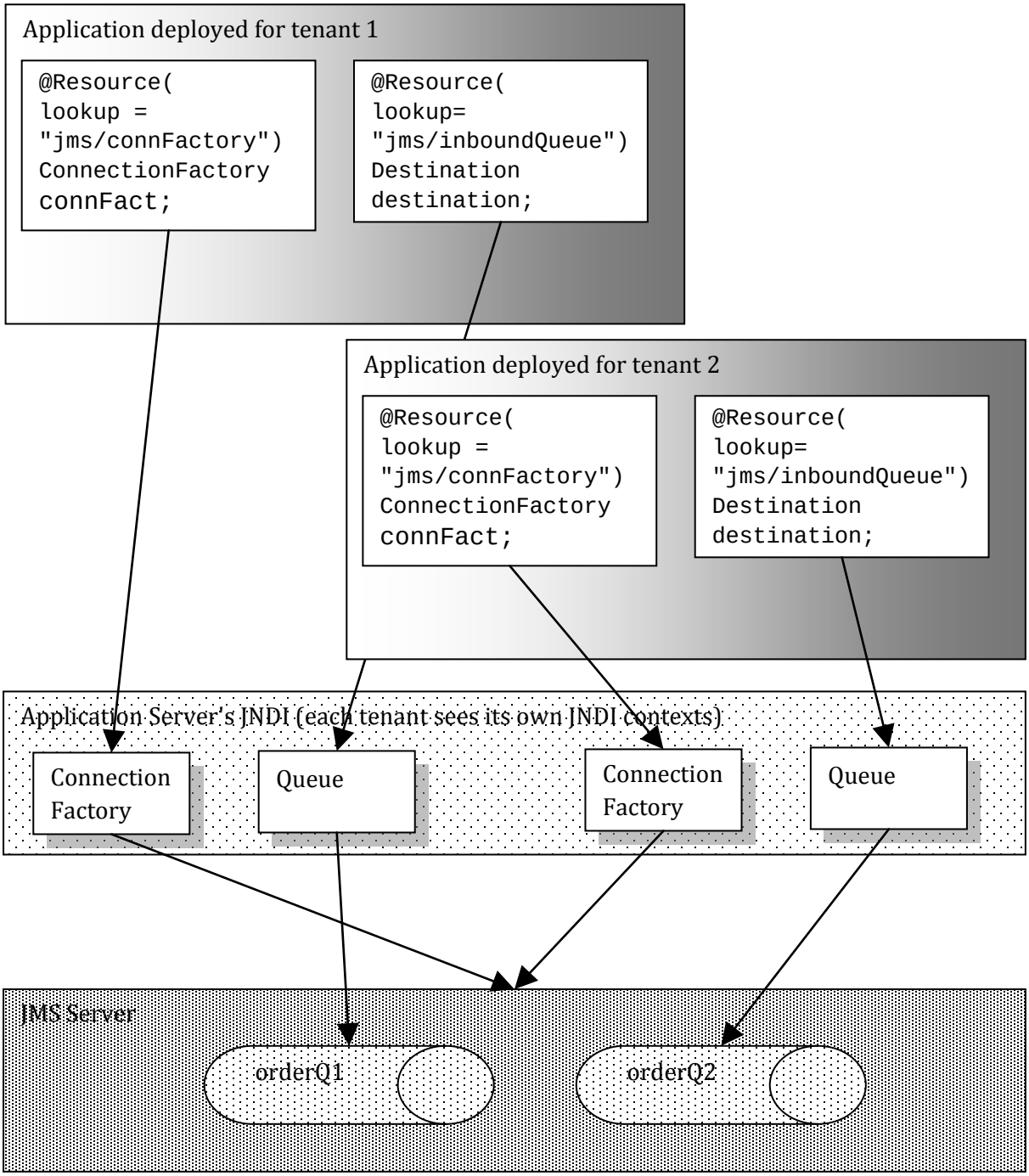
Each tenant application uses a separate destination, which is achieved by using separate JMS servers for each tenant application. The deployer configures the connection factory object used by each tenant application to use a different JMS server.



The JMS servers do not need to be aware that they are being used by different tenants.

Isolation using multiple destinations

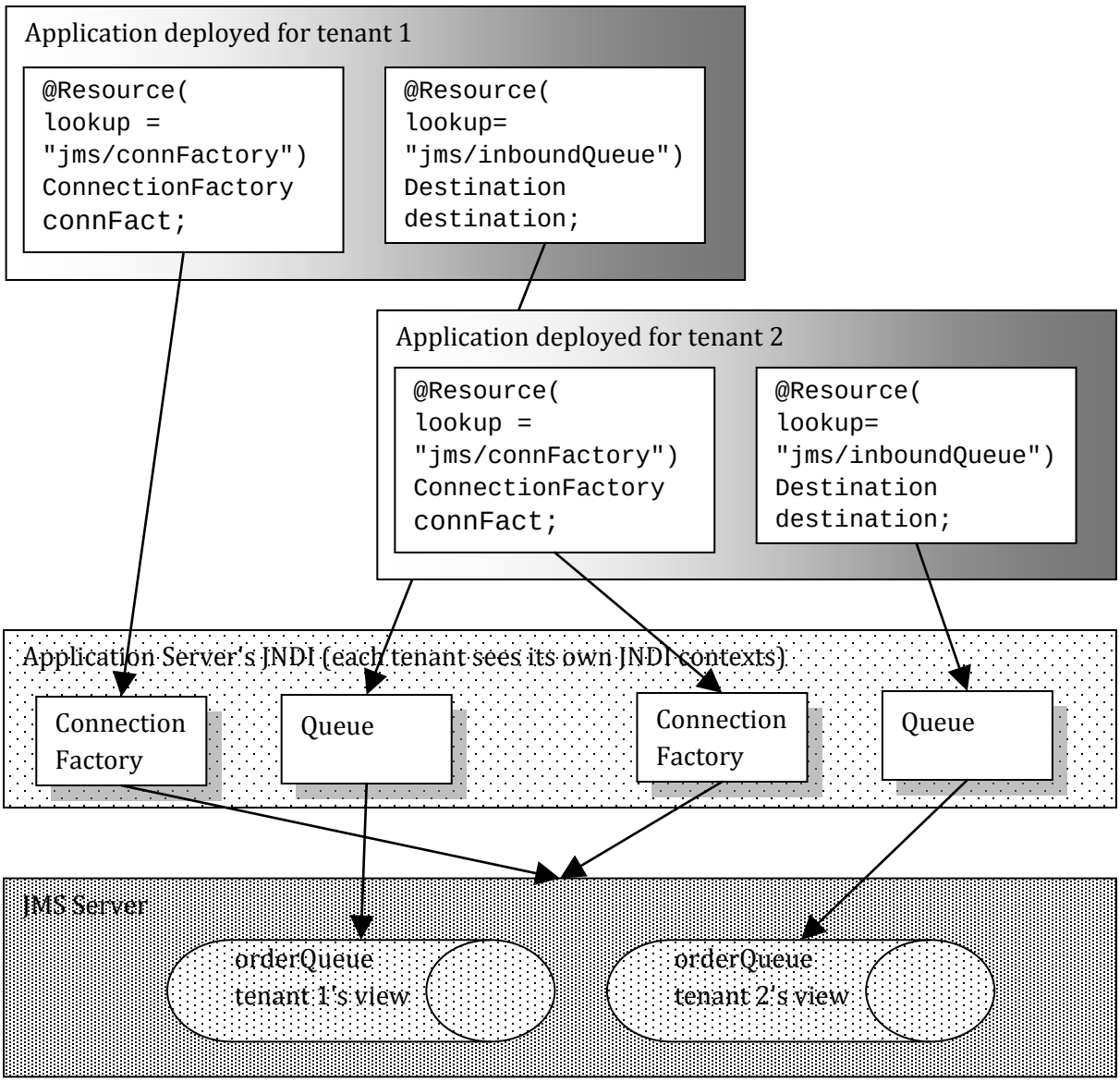
Each tenant application uses a separate destination on the same JMS server. The deployer configures the destination object used by each tenant application to use different destinations.



The JMS server does not need to be aware that these destinations are being used by different tenants. However the provisioning process will need to generate a different destination name for each tenant, possibly automatically.

Isolation using a multi-tenant destination

Each tenant application uses the same destination on the same JMS server, but the JMS provider is tenant-aware and can isolate each tenant from one another so from *their* perspective they see different destinations



The JMS server needs to be aware that the destination is being used by different tenants, and needs to be able to distinguish one tenant from another. There are several ways this might be achieved:

- by setting the tenantId on the connection factory,
- by the JMS provider looking up the tenantId from a well-known location in JNDI, or
- by the application server passing in the tenantId using the JCA API.

Specifying the isolation level

The resource configuration metadata could be used to allow the application developer to specify whether a destination is shared or isolated.

Specifying isolation level on a destination

The most obvious way to do this would be to extend the `@JMSDestinationDefinition` annotation to allow the required isolation level to be specified:

```
@JMSDestinationDefinition {
    description=“MDB input queue”
    name="ims/inboundQueue",
    className="javax.jms.Queue",
    resourceName="orderQueue",
    tenantIsolation="SHARED"
}
```

```
@JMSDestinationDefinition {
    description=“MDB input queue”
    name="ims/inboundQueue",
    className="javax.jms.Queue",
    resourceName="orderQueue",
    tenantIsolation="ISOLATED"
}
```

If `tenantIsolation="SHARED"` , the provisioning process will operate in exactly the same way as if the application were not multi-tenant.

If, `tenantIsolation="ISOLATED"` the provisioning process would provision the necessary degree of isolation using one of the three approaches described :

A. provisioning multiple providers,
B. provisioning multiple destinations or
C. provisioning a single multi-tenant destination

The resource provisioning process could be left to decide for itself which method is used, or the `tenantIsolation` property could be extended to allow the application to specify which of these methods should be used.

If method (c) is chosen, the provisioning process should not only provision a physical destination capable of supporting multiple tenants, but also notify the JMS provider at runtime that the queue or topic needs to be specific to a particular tenant id. This can be achieved using a new property on a `javax.jms.Destination` which specifies the isolation level:

```
public interface Destination {

  /**
   * Specifies the isolation level of this Destination object
   *
   *
   * @param isolationLevel the isolation level
   * @throws JMSException
   */
   void setIsolationLevel(IsolationLevel level) throws JMSException;

  /** Returns the isolation level of this Destination
   *
   * @return the isolation level
   * @Throws JMSException
   */
   IsolationLevel getisolationLevel() throws JMSException;

}
```

`IsolationLevel` could be an emumeration.

This property would not be needed if isolation was being provided by methods A or B.

If the isolationLevel has been specified on a destination then all messages sent to that destination would be identified with the tenant id. Conversely, a consumer created on that destination would only receive messages identified with that tenant id. This means that the JMS client would need to know the tenantId. The possible ways to achieve this are described in "Connection configuration for a multi-tenant application" below.

Although the JMS specification might define a minimum degree of isolation in that tenants will not be able to receive each other's messages, JMS vendors will be free to decide for themselves exactly how isolated the tenants will be. They might choose to implement the isolation using completely separate destinations, or by using the same destination with tenant id as a kind of built-in message selector.

Specifying isolation level on a connection factory

Application developers may also wish to specify that all destinations used with a particular connection factory should have a certain level of isolation.

```
@JMSConnectionFactoryDefinition {
    className="javax.jms.ConnectionFactory",
    tenantIsolation="ISOLATED"
}
```

However the deployment descriptor of an application does not specify which connection factory will be used with a particular destination. JMS 2.0 may change this for MDBs, but in general only the runtime code defines this information.

Therefore even if the `@JMSConnectionFactoryDefinition` specifies `tenantIsolation="ISOLATED"` the deployer cannot tell what destinations this relates to. The deployer can therefore only achieve the required isolation levels using method A, provisioning multiple providers. This information alone is insufficient information to allow the

deployer to offer method B (provisioning multiple destinations) unless the destinations themselves are also annotated with an isolation level. It is also insufficient information to allow the deployer to offer method C (provisioning a single multi-tenant destination) unless the JMS provider allows the multi-tenant nature of a destination to be configured at runtime.

In this last case only it may be necessary for the deployer to set the required isolation level on the connection factory using the following API.

```
public interface ConnectionFactory {

  /**
   * Specifies the isolation level of this ConnectionFactory
   *
   *
   * @param isolationLevel the isolation level
   * @throws JMSException
   */
   void setIsolationLevel(IsolationLevel level) throws JMSException;

  /** Returns the isolation level of this ConnectionFactory
   *
   * @return the isolation level
   * @Throws JMSException
   */
   IsolationLevel getisolationLevel() throws JMSException;

   ...
```

How to pass tenantId to the JMS client

If destination isolation is being achieved using method C (provisioning a single multi-tenant destination) then the JMS client needs to be able to know the tenantId of the application that is using it.

There are four possible ways where this may be achieved:

1. The deployer could set the tenantId on the connection factory or destination administered objects used by a tenant application instance. This would require new setter and getter methods on ConnectionFactory to allow this:
   - void `setTenantId(String tenantId)`
   - `String getTenantId()`

   It would be necessary to ensure that only the deployer could call `setTenantId`, not the application itself, to avoid one tenant application adopting a different tenantId.

2. The JMS client could use JNDI to lookup the tenantId from a well-known JNDI context, such as `java:comp/tenantId`. This would make the JMS client dependent on JNDI, which is probably acceptable as JNDI is part of Java SE. The use of JNDI is potentially expensive, though this is probably acceptable if the lookup is only performed when a JMS connection is first created.

3. The JCA API could be extended to allow the application server to pass the tenantId to the resource adapter, which could then use a private API to pass it to the JMS client itself. (this would only work with provider-specific RAs).

   The JCA specification already defines how the application server may pass "identity" information to the resource adapter by means of a `javax.security.auth.Subject` object. This typically contains a single `java.security.Principal` object which contains the user and password to be used by the JMS client.

   This works as follows: when a new connection needs to be added to the connection pool, the `ConnectionManager` class (provided by the application server) calls the resource adapter method `ManagedConnectionFactory.createManagedConnection`, with the `javax.security.auth.Subject` as an argument. Similarly when an existing connection is fetched from the pool, the `ConnectionManager` calls `ManagedConnection.getConnection()` which again has the `javax.security.auth.Subject` as an argument.

   If we considered tenantId to be a kind of "identity" information, the JCA spec could require that `javax.security.auth.Subject` object should also contain an additional `java.security.Principal` object which holds the tenantId. Perhaps a new `TenantPrincipal` class.

   This would allow each connection to "know" the tenantId that is using it.

4. If it were considered inappropriate to "pollute" the `javax.security.auth.Subject` object with tenant-related information, the methods on `ManagedConnectionFactory and ManagedConnection` mentioned above could simply be extended to add `tenantId` as an argument.

Note that both (3) and (4) would require changes to the JCA spec.

Notes from a review of version 3

The following notes were made following a review of version 3 of this document with the Java EE platform specification leads.

Specifying the required isolation level in resource metadata

The use of resource metadata to specify the required isolation level was endorsed.

```
@JMSDestinationDefinition {
    description="MDB input queue"
    name="ims/inboundQueue",
    className="javax.jms.Queue",
    resourceName="orderQueue",
    tenantIsolation="ISOLATED"
}
```

```
@JMSConnectionFactoryDefinition {
    className="javax.jms.ConnectionFactory",
    tenantIsolation="ISOLATED"
}
```

To avoid any possible confusion with database technology, the term "isolation level" should be avoided and the term "tenant isolation" used.

Passing tenantId to the connection used by a MDB to consume messages

In the discussion "How to pass tenantId to the JMS client", option 3 discussed how the JCA spec allows the application server to pass a `javax.security.auth.Subject` object to the resource adapter when creating a new connection or fetching one from the pool. This Subject object could be extended to hold a new Principal object that contained the tenantId..

However this doesn't cover the connection used by a MDB to receive messages.

JCA defines a security inflow contract, which, is a mechanism for allowing the RA to pass a caller java.security.Principal to the MDB application. The application can obtain this by calling EJBContext.getCallerPrincipal(). (Can it also call Subject.getPrincipals() to obtain this Principal?).

However this doesn't help with Java EE 7 multi-tenancy, where we need a way for the *application server* to pass the tenant information to the RA when the endpoint is activated.

JCA 1.6 doesn't specify a way to do this currently. One possibility is to add a third argument to ResourceAdapter.endpointActivation(MessageEndpointFactory endpointFactory, ActivationSpec spec) as follows:

```
ResourceAdapter.endpointActivation(
   MessageEndpointFactory endpointFactory, ActivationSpec spec,
   javax.security.auth.Subject subject)
```

The application server would need to set an appropriate `javax.security.auth.Subject`. This would require a change to the JCA specification.

An alternative to passing this in via the JCA API would be for the app server to associate the tenantId with the current thread (how would it do this?), so that the endpointActivation() implementation could call Subject.getPrincipals() and obtain, say, a TenantPrincipal.

Note that I don't see tenantId as flowing in with each message: it's the other way round: the call to endpointActivation() would pass the tenantId to the JMS server when creating the consumer.

Passing tenantId in a Java EE 8 SaaS application

The document makes four proposals for passing tenantId to the JMS client when a connection was created. These proposals were reviewed to assess which of them could be easily extended to support the needs of Java EE 8. This will provide full support for SaaS, which will mean the ability of a single deployed application to work on behalf of multiple tenants.

Although the details of how this would work do not need to be decided for Java EE 7, the basic idea would be that tenantId would be associated not with the deployed application but with the current thread.

We reviewed which of the four options for passing tenantId to the JMS client discussed above, could be extended to handle SaaS multi-tenancy:

Mechanism for passing tenantId to the JMS client	Would it work in a SaaS application?
1. The deployer would set the tenantId on the connection factory or destination administered objects used by a tenant application instance.	No, since all tenants using a given deployed application would see the same administered objects.
2. The JMS client could use JNDI to lookup the tenantId from a well-known JNDI context, such as `java:comp/tenantId`.	Yes, JNDI could be made to return a different tenantId depending on the thread that called it. However since the tenantId would need to be checked every time a JMS client method was used, and JNDI may be relatively slow, this is potentially a slow solution.
3. The JCA API could be extended to allow the application server to pass the tenantId to the resource adapter by means of a `javax.security.auth.Subject` object when a connection was created or fetched from the pool (and when an endpoint was activated).	No, this would allow per-connection tenantId but not per-thread tenantId.
4. Extend JCA API to allow tenantId to be passed as an argument when a connection was created or fetched from the pool (and when an endpoint was activated).	No, this would allow per-connection tenantId but not per-thread tenantId.

However it was considered that the use of a `javax.security.auth.Subject` object to pass tenantId might be a good way to pass a per-thread tenantId, since Java SE already has the concept of per-thread security credentials. This would by-pass any need to pass this information using the JCA API.

The Java EE 7 expert group would take on the issue of how best to pass tenantId to a resource manager in a way which would work on a per-thread basis for Java EE 8. There is therefore no need for a JMS-specific solution.