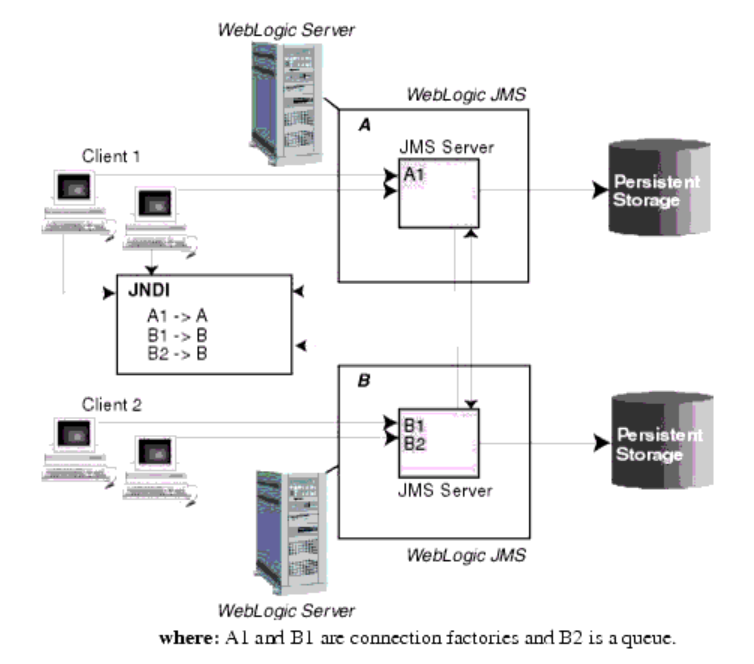
**JMS (JAVA Messaging Services)**

**WebLogic JMS Architecture**



* A JMS server is an environment-related configuration entity that acts as management container for JMS queue and topic resources defined within JMS modules that are targeted to specific that JMS server. A JMS server's primary responsibility for its targeted destinations is to maintain information on what persistent store is used for any persistent messages that arrive on the destinations, and to maintain the states of durable subscribers created on the destinations. You can configure one or more JMS servers per domain, and a JMS server can manage one or more JMS modules.
* JMS modules contain configuration resources, such as standalone queue and topic destinations, distributed destinations, and connection factories, and are defined by XML documents that conform to the **weblogic-jms.xsd** schema.
* Client JMS applications that either produce messages to destinations or consume messages from destinations.
* JNDI (Java Naming and Directory Interface), which provides a server lookup facility.
* WebLogic persistent storage (a server instance's default store, a user-defined file store, or a user-defined JDBC-accessible store) for storing persistent message data.

**JMS Servers:**

JMS server's primary responsibility for its targeted destinations is to maintain information on what persistent store is used for any persistent messages that arrive on the destinations, and to maintain the states of durable subscribers created on the destinations. JMS servers are persisted in the domain's config.xml file and multiple JMS servers can be configured on the various WebLogic Server instances in a cluster, as long as they are uniquely named.

**Create JMS servers**

JMS servers are environment-related configuration entities that act as management containers for JMS queue and topic destinations in JMS modules that are targeted to them. Multiple JMS modules can be targeted to each JMS server in a domain.

Steps to create a JMS server:

1. If you have not already done so, in the Change Center of the Administration Console, click Lock & Edit (see Use the Change Center).
2. In the Administration Console, expand Services > Messaging and select JMS Servers.
3. On the Summary of JMS Servers page, click New.

***Note****: Once you create a JMS server, you cannot rename it. Instead, you must delete it and create another one that uses the new name.*

1. On the Create a JMS Server page:
2. In Name, enter a name for the JMS server.
3. In Persistent Store, select a pre-configured custom file or JDBC store that will be used by the JMS server or click the Create a New Store button to create a store on the fly. If you leave this field set to none, then the JMS server will use the default file store that is automatically configured on each targeted server instance. For more information about configuring stores, see Configure custom persistent stores.

***Note****: When a JMS server is targeted to a migratable target, it cannot use the default store, so a custom store must be configured and targeted to the same migratable target.*

1. Click Next to proceed to the targeting page.
2. On the Select Targets page, select the server instance or migratable server target on which to deploy the JMS server. Migratable targets define a set of WebLogic Server instances in a cluster that can potentially host a pinned service, such as a JMS server.

***Note****: In a clustered server environment, a recommended best practice is to target a JMS server to a migratable target, so that a member server will not be a single point of failure. A JMS server can also be automatically migrated from an unhealthy server instance to a healthy server instance, with the help of the server health monitoring services.*

1. Click Finish.
2. On the Summary of JMS Servers page, click the new JMS server to open it.

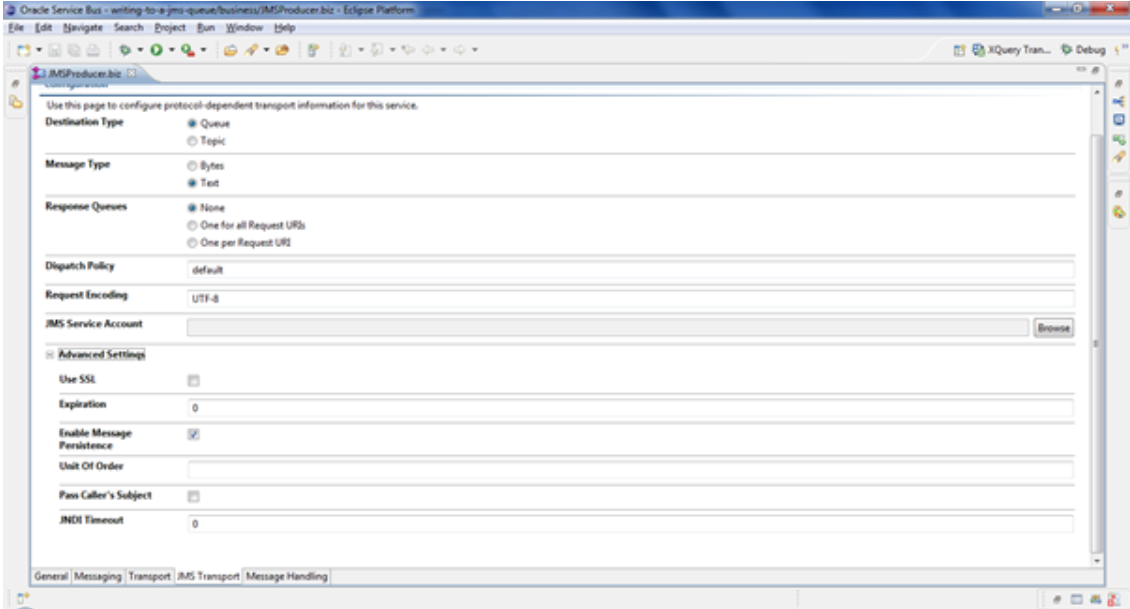
***Note*** *that there are many optional parameters that can be set on the JMS server configuration tabs, including General configuration parameters, Thresholds and Quotas, Logging, and Server Session Pools. For more information about configuring these parameters, see Configure JMS servers.*

1. After modifying any values, click Save.
2. To activate these changes, in the Change Center of the Administration Console, click Activate Changes.

Not all changes take effect immediately—some require a restart

**USING JMS TRANSPORT WITH OSB**

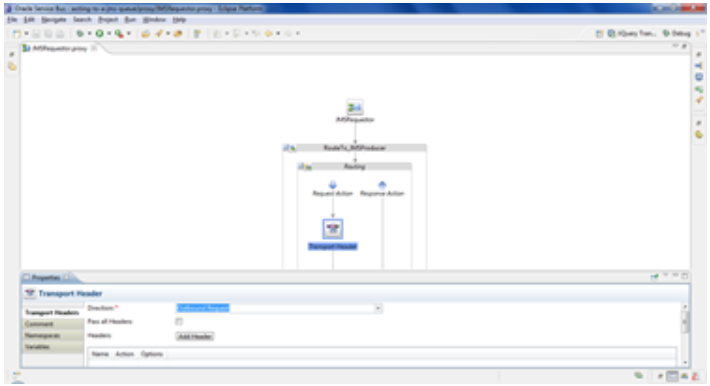
1. JMS Transport – Advanced settings



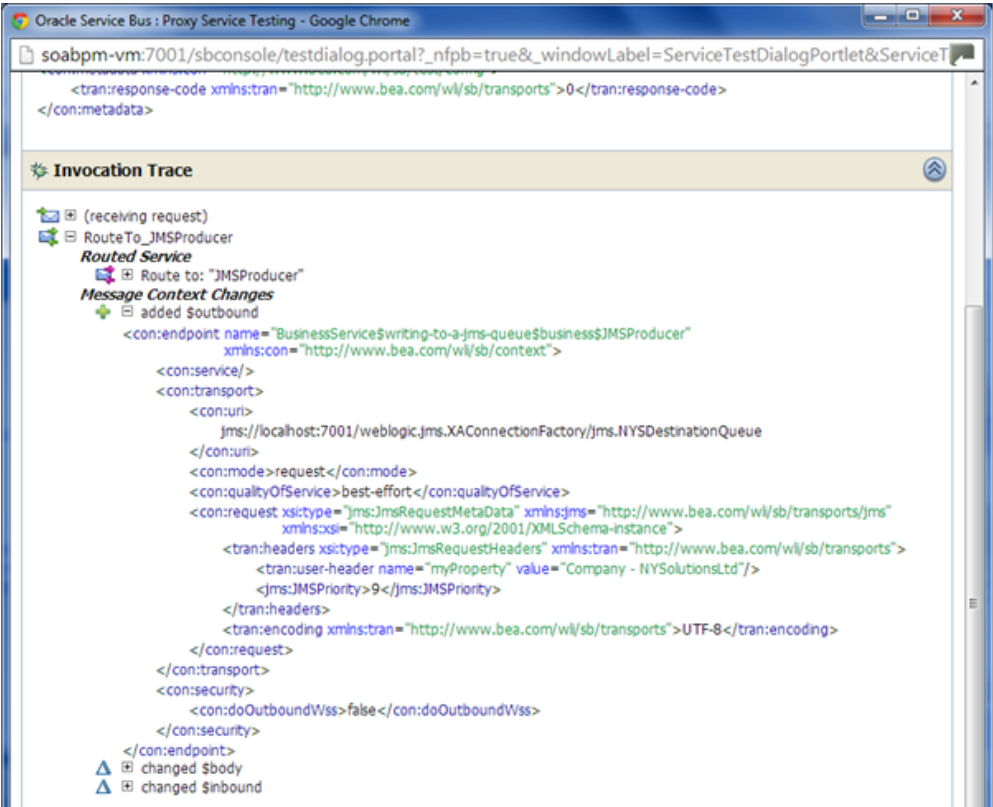
* The option Enable Message Persistence is by default enabled, which guarantees message delivery because the messages persist and will survive server shutdowns and failures.
  + To improve throughput, deselect this option if the occasional loss of a message is tolerable.
* Set a time interval in milliseconds in the Expiration field to specify the message time-to-live.
  + After the time-to-live is passed, the message will automatically be treated according to the Expiration Policy defined on the JMS destination (that is, the queue) and either discarded, logged, or redirected to another JMS destination.
  + The default value of 0 for Expiration means that a message never expires and therefore waits to be consumed forever or until the server is shutdown, if the message is not persistent.
* Any JMS destination can override the Expiration set on the business service by setting the Time-to-Live Override setting through the WebLogic console when configuring the JMS destination.

1. Changing JMS transport Headers

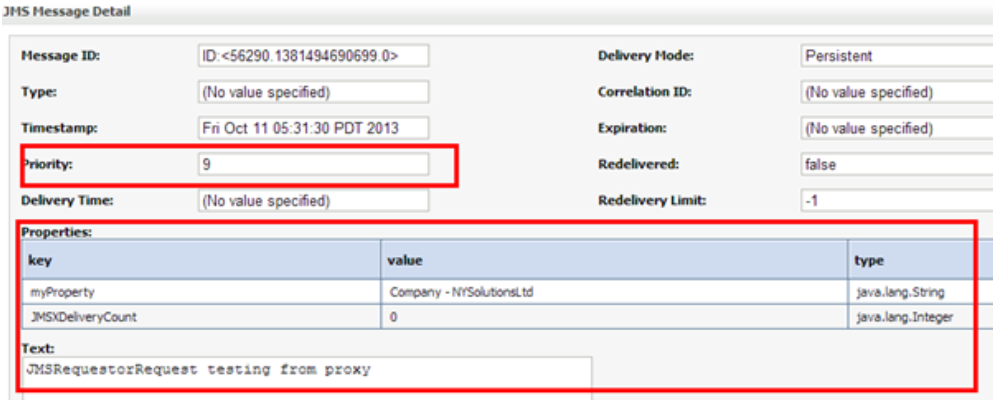
* When sending a message to JMS queue or topic through a business service, default values for several JMS Transport message headers are used. The message headers as well as the message properties can be overwritten by using a Transport Header action in a proxy service when routing to the business service.



* Set the properties and deploy the project. On testing you will see these



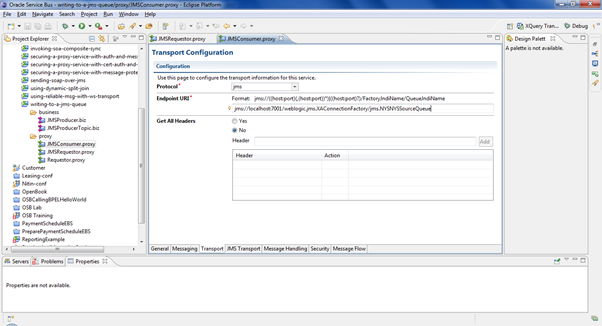
* Check the messages on the destination queue.



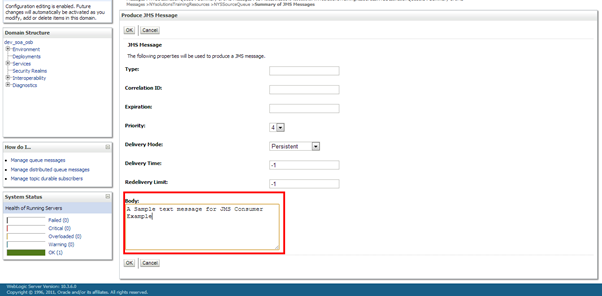
* We have successfully overwritten the JMS header values configured in business service using the transport header action.

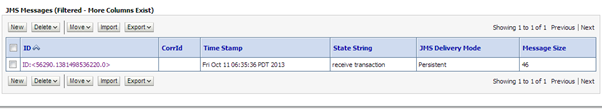
***Note****: The Transport Header action can be placed into the Request Action of the Routing action or Publish action in order to override the jms header values.*

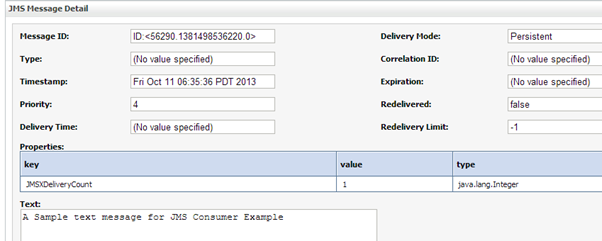
1. Consuming from a JMS Queue



* Testing the JMS Consumer



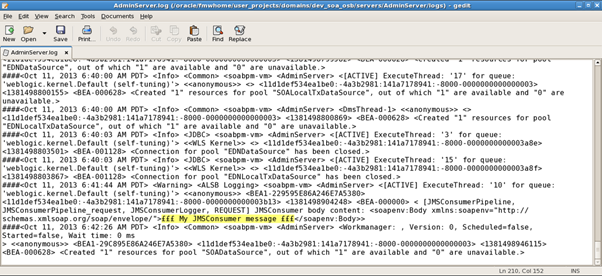




* Deploy the JMSConsumer proxy and the message is gone.



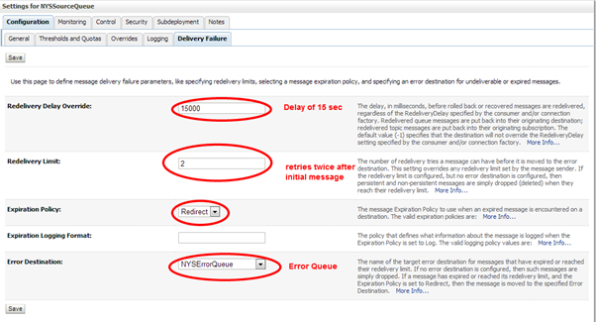
* You can now check the OSB server logs and you should see the logged body element [highlighted].



* Key points
  + On a proxy service, the JMS protocol always works inbound, that is, the proxy service is consuming messages from a queue.
  + On a business service in contrast, the JMS protocol always works outbound, that is, the business service is putting the message to a queue.

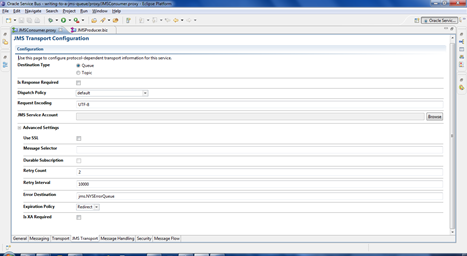
1. Configuring Retry handling in JMS

Configure the redelivery limit and expiration policy on a Queue that will take part in a reliable communication. This option prevents an infinite loop: when the redelivery limit is reached, the message will be moved to the error queue. The redelivery limit on the queue will always override the value set on the message itself.



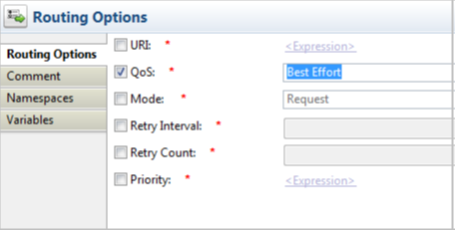
When creating a message from the weblogic console make sure to overwrite the -1 in the Redelivery Limit field to the same value as specified on the queue Otherwise the -1 will overwrite the setting on the queue and it will again retry forever.[Bug in console]

These settings can also be done from the Advanced Settings section on the JMS Transport tab of the proxy service. In that case, they will overwrite any settings specified directly on the JMS queue:



**QOS – Using Best effort**

When we use XA-enabled resources, enable Same Transaction for Response, and use Exactly Once QoS on the outbound transport, every resource will be rolled-back in case of an error in the Request or Response lane. To make this work, we should check if we are using a XA or non-XA resource and what the value of the QoS is, otherwise the resource won’t take part in the global transaction.



In this recipe, we don’t need to set QoS to Exactly Once because this is the default value when we use JMS with a XA Connection Factory as inbound transport. To know the QoS values of your proxy service, we can read the following transport variables:

**$inbound/ctx:transport/ctx:qualityOfService**

**$outbound/ctx:transport/ctx:qualityOfService**

The inbound variable value can’t be changed and OSB uses the inbound value as a default value for the outbound transport. We can change the outbound QoS value with a Routing Options action.

***Note: Only use the Exactly Once of QoS when there is a transaction involved.***

In case of a Publish or a Service Callout action, the default value of QoS is always **Best Effort**. This means in case of a JCA resource such as a DB or an AQ adapter we need to set the dataSourceName property of the resource adapter and not using the XADataSourcename property, otherwise we will get a runtime error. If the Publish or Service Callout action should be part of the global transaction, then use the Routing Options action and set the QoS to **Exactly Once**. In the case of a JCA adapter, we need to use the XADataSourceName property of the resource adapter.

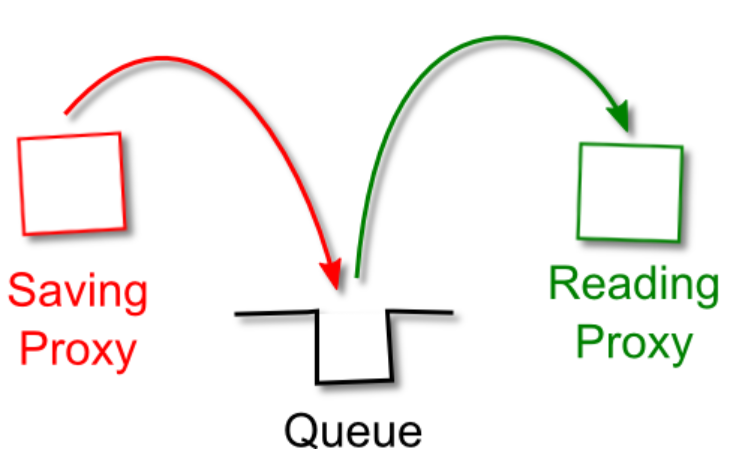
If our business service Publish action or Service Callout does not support XA, then with each retry of the proxy service, the service will be invoked. This can lead to many duplicate messages or invokes.

We can also force this behavior in a XA-enabled business service by adding a Routing Option in the Request Action of the Route Node. In the Routing Options, you need to enable QoS and set it to Best Effort. With Best Effort, the resource won’t take part in the global transaction even when XA is enabled.

**Transactions**

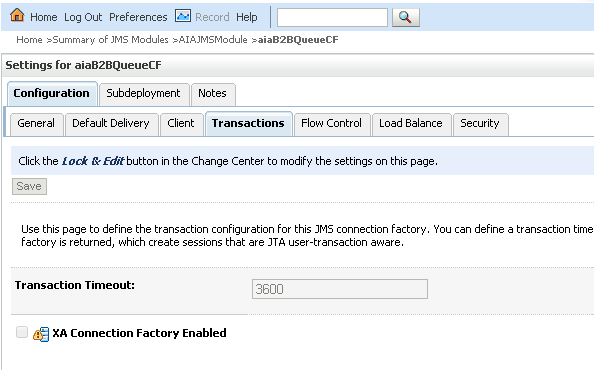
Reliable messaging in OSB is implemented over the JMS layer. It already has everything needed, we only have to configure it properly and provide the adapter proxies.

A queue allows to place a message into it, and then retrieve it from another end. What’s important for reliable messaging, a queue can be persistent, meaning the message is saved in a store that survives the JVM reboot. Typically that store is a special file, but could also be a database.



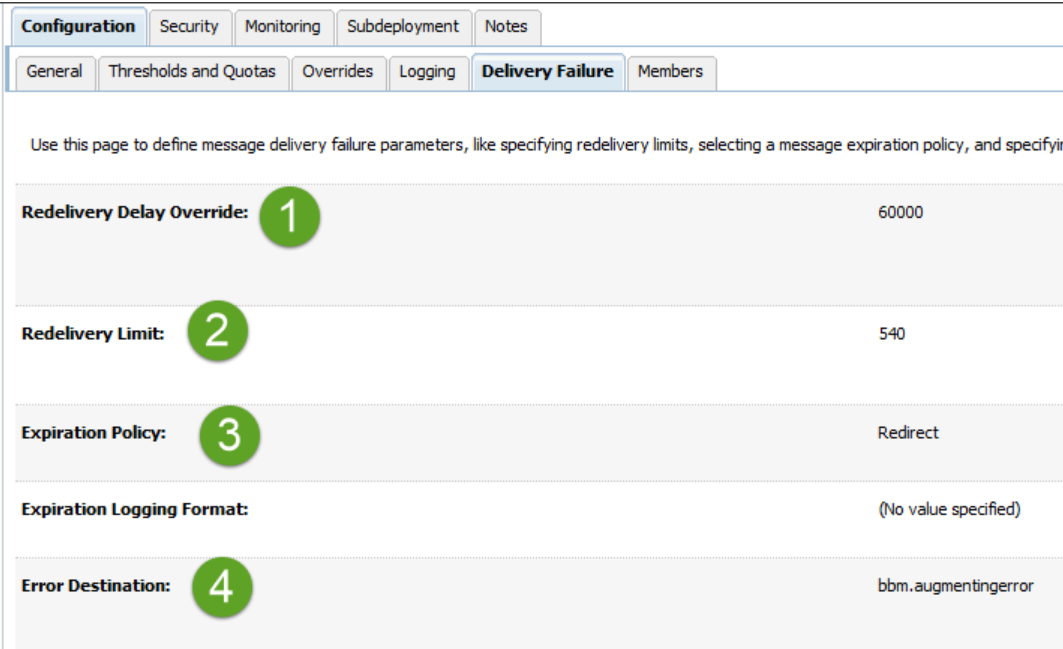
Only when the reading proxy confirms the message was delivered successfully, the message will be removed from the disk storage. In case of any recoverable error (a fault, even the whole box crash) the message remains in the queue and will be re-processed when the cause of error is corrected.

To use a queue in a transactional context, we should connect to it using a connection factory with transactions enabled (so-called XA):

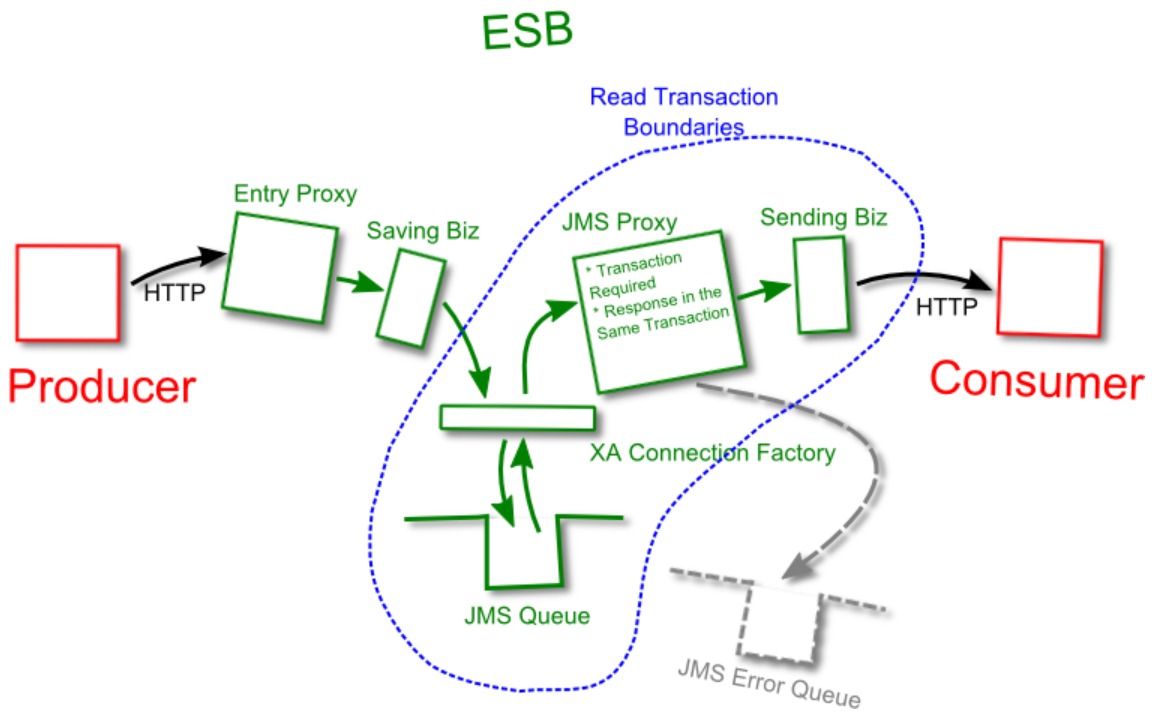


**Error Handling:**

JMS queues allow to configure the retry interval (say, we want to retry every 60 seconds), maximum number of retries and what to do with the message when it is tried for too long.



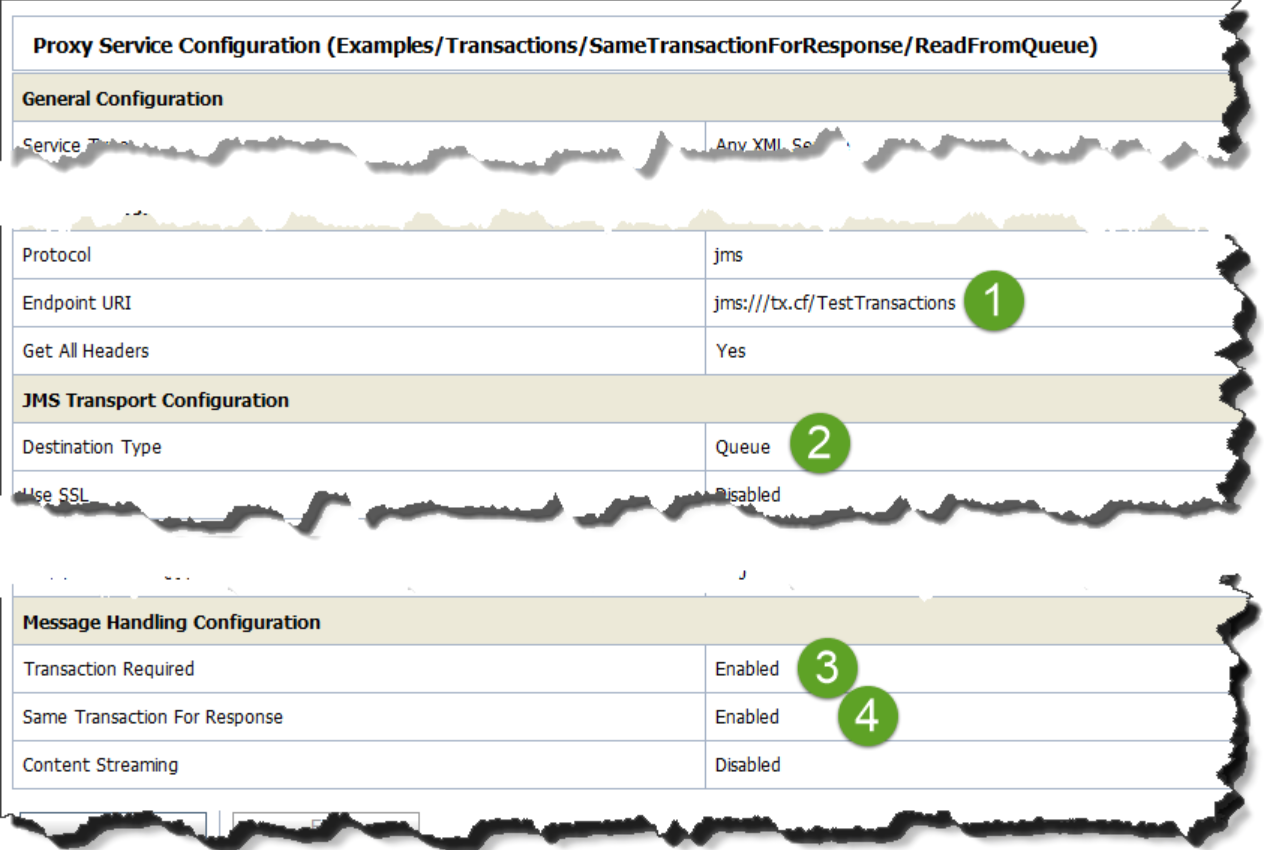
**POC on Transaction (OSB):**



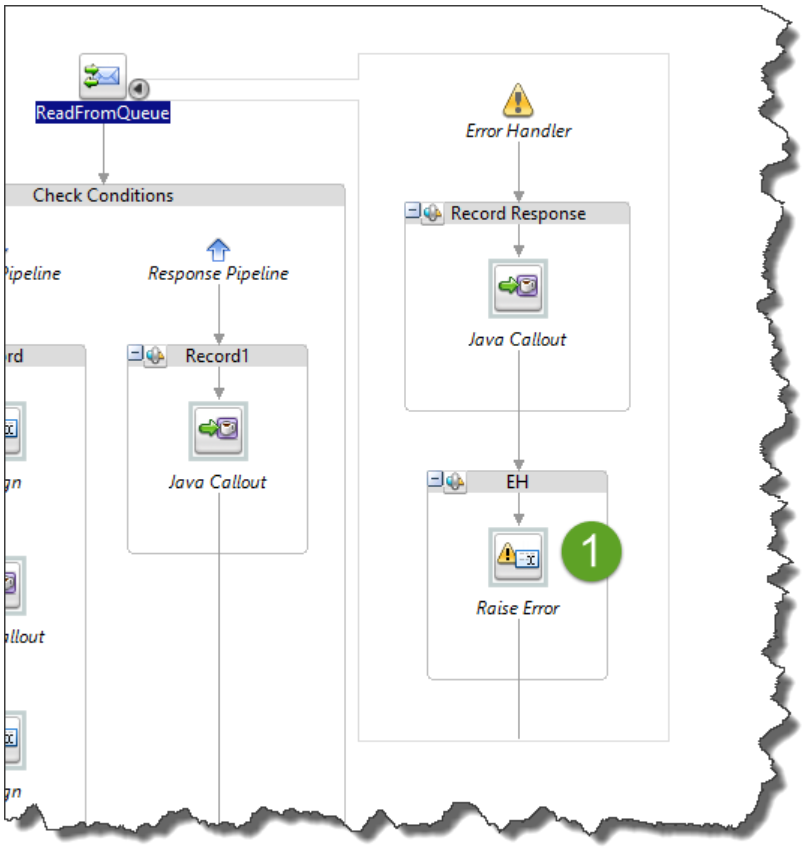
* We need to have an entry proxy that receives HTTP requests from the producer. That proxy saves the message into the queue by calling a Biz service (2), and responds to the consumer with HTTP 200. In case of any fault it responds with HTTP 500.
* A JMS Biz service places the message into a JMS queue. The service uses a connection factory with XA flag (transactional) set. If the store is not available for any reason (could happen with JDBC-based stores), the transaction will fail and the entry proxy will get a fault.

*The URL should look like jms:///tx.cf/TestTransactions, where tx.cf is an XA-enabled connection factory, and TestTransaction is the JNDI name of the queue.*

* A JMS reading proxy. It uses the same XA connection factory. In addition, it has “Transaction Required” and “Response in the Same Transaction” flags (see below for details).



* Important implementation detail – it should raise an error in the error handler, if it has one. Reply-with-Error step is not generating a fault and hence does not roll back the transaction:



* A HTTP Biz service that delivers the messages to the consumer. In case of any error it raises a fault and propagates it to the JMS reading proxy (3). The proxy rollbacks the transaction, keeping the message in the queue.

**Why Response in the Same Transaction?**

The answer is in the OSB threading model. The thread that reads the message from the queue and then sends it to the consumer is not the same thread that waits for the response.

Now imagine this scenario:

1. Thread1 reads the message from the JMS queue.
2. Thread1 calls the Biz service and sends the message over HTTP.
3. Now the Thread1’s work is done, it can complete.
4. Thread1’s transaction commits, too, because Thread1 raised no errors.
5. Hence, the message is removed from the queue.
6. Thread2 is waiting for the consumer response.
7. Thread2 gets a timeout and raises a fault.

But that fault cannot rollback the transaction because it has been already committed!

At this point we do not know if the consumer has received the message (timeout doesn’t give us any information), and we do not have the message in the queue to retry.

To prevent this from happening, Thread2 should be placed into the same transaction as Thread1.

*From “****At Least Once****” to “****Exactly Once****” by Eliminating Duplicates*

As I said already, the HTTP links in our design are not reliable. Producer has to retry when it is not sure the message has reached OSB, and OSB has to retry when it is not sure the message has reached the consumer. From time to time, the message does reach the recipient, but the sender cannot tell (due to timeout errors or dropped connections), so the sender performs a retry, and now the recipient has 2 identical messages. We’re lucky if the messages are idempotent, i.e. their duplicate processing doesn’t cause any problems. For instance, a request to change a user’s phone number can be considered idempotent if received in a short timespan (as it is usually a case with temporal failures). But what if the messages are not idempotent? What if the message is the proverbial bank transfer note, for instance? We cannot allow for duplicate transfers!

**Ques**: How to deal with it?

**Answer**: The answer is to eliminate the duplicates based on message id.

*Every message entering the system must have a unique id. The id is generated by the producer application. It is important that any retry attempt by the producer uses the same id for a message! Otherwise the same message may enter OSB under different ids, and these duplicates cannot be eliminated.*

Sometimes this id could be a natural value from the business domain (say, account# plus the invoice#), but more often the id is generated as a random artificial value.

**Eliminating Duplicates on the Consumer**

The consumer is expected to have some shared storage, usually a database, where it can check if the message has been processed already. If it was, the duplicate message is just dropped.

**Eliminating Duplicates on OSB**

In the default install, OSB’s managed servers do not have a shared storage. One message instance can be delivered to one managed server, while the second one - to another, and hence OSB cannot detect and eliminate the duplicates. At the same time, in many configurations OSB is deployed with access to either Coherence or third-party shared in-memory storage. OSB’s managed servers can place the ids of the messages they see into such storage, and consult it again to check if the message is a duplicate. The best place to check-and-drop is right before the message is placed into the queue. If the check is moved into JMS reading proxy, every time a message is retried the same id will have to be checked, and the proxy will not know why there is a duplicate – is it because it was already processed by another node or because of the retry?

Using retry count in the Biz service configuration is an easy way to make your service unreliable. Biz-based retries are holding a thread, potentially causing the resource exhausting!

Imagine again that the message is retrieved from the queue and sent to the consumer. Consumer is not available for one reason or another, and Biz service retries the message a few time, but then what? What should it do with the message? Any JVM-wide error that happens during Biz retries immediately loses the message, too.