Using webMethods Integration Server to Build a Client for JMS

Version 9.10

April 2016

WEBMETHODS

This document applies to webMethods Integration Server and Software AG Designer Version 9.10 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: IS-JMS-DG-910-20160415

Table of Contents

About this Guide

Using webMethods Integration Server to Build a Client for JMS is for the developer who is responsible for developing solutions that use webMethods Integration Server to send and receive messages using the Java Message Service (JMS) standard.

This guide explains:

- How to build services that send and receive JMS messages using built-in services.
- How to create and configure JMS triggers for receiving JMS messages
- How Integration Server works with cluster policies when sending JMS messages.
- How to configure JMS triggers to consume messages from a destination in a load-balanced fashion.

This guide assumes that you are familiar with the following:

- Basic concepts of webMethods architecture and terminology.
- Usage of Designer to create elements and build services.
- General knowledge of programming, the Java programming language, and the JMS API.
- How to establish connections to one or more JMS providers by creating JMS connection aliases. For more information about creating a JMS connection alias, see *webMethods Integration Server Administrator's Guide*.

Note: An in-depth treatment of messaging architecture is beyond the scope of this guide but is available elsewhere.

Note: This guide describes features and functionality that may or may not be available with your licensed version of webMethods Integration Server For information about the licensed components for your installation, see the **Settings > License** page in the webMethods Integration Server Administrator.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies storage locations for services on webMethods Integration Server, using the convention *folder.subfolder:service* .

Convention	Description
UPPERCASE	Identifies keyboard keys. Keys you must press simultaneously are joined with a plus sign (+).
Italic	Identifies variables for which you must supply values specific to your own situation or environment. Identifies new terms the first time they occur in the text.
`Monospace font`	Identifies text you must type or messages displayed by the system.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at http://documentation.softwareag.com. The site requires Empower credentials. If you do not have Empower credentials, you must use the TECHcommunity website.

Software AG Empower Product Support Website

You can find product information on the Software AG Empower Product Support website at https://empower.softwareag.com.

To submit feature/enhancement requests, get information about product availability, and download products, go to Products.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the Knowledge Center.

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at http://techcommunity.softwareag.com. You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

1 Introduction to JMS

JMS Messaging

The Java Message Service (JMS) is a Java API that allows applications to communicate with each other using a common set of interfaces. The JMS API provides messaging interfaces, but not the implementations.

A *JMS provider*, such as webMethods Universal Messaging or webMethods Broker, is a messaging system that supports the JMS message interfaces and provides administrative and control features. It supports the routing and delivery of JMS messages.

JMS clients are the programs or components, written in Java, that produce and consume messages.

Messaging Styles

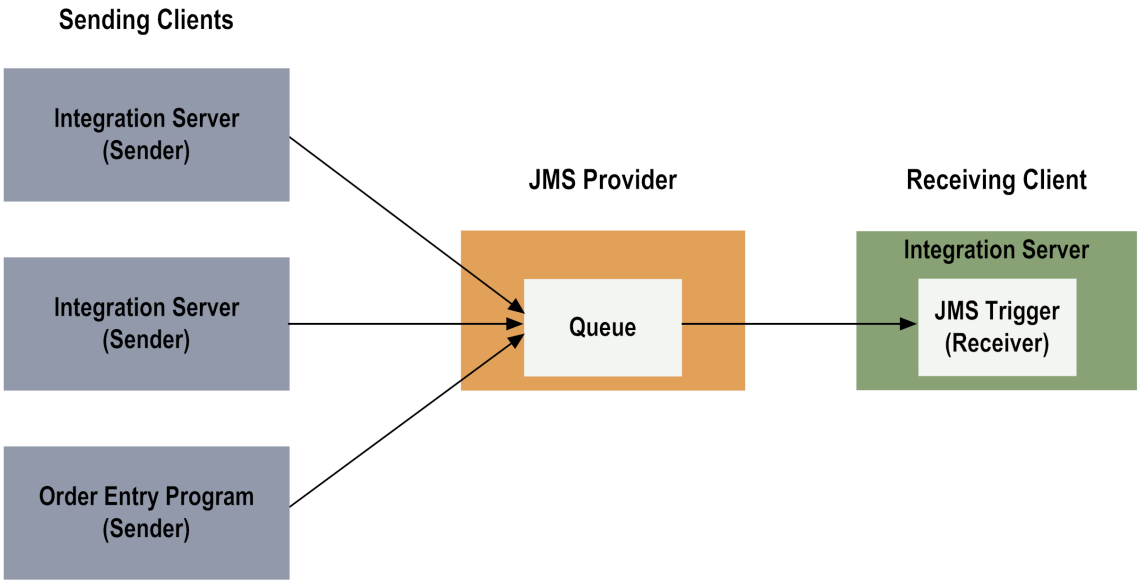
A messaging style refers to how messages are produced and consumed. JMS supports the publish-subscribe (pub-sub) and point-to-point (PTP) messaging styles.

Point-to-point (PTP) Messaging

In point-to-point (PTP) messaging, message producers and consumers are known as *senders* and *receivers*.

The central concept in PTP messaging is a destination called a *queue*. A queue represents a single receiver. Message senders submit messages to a specific queue and another client receives the messages from the queue.

In the PTP model, a queue may receive messages from many different senders and may deliver messages to multiple receivers; however, each message is delivered to only one receiver.

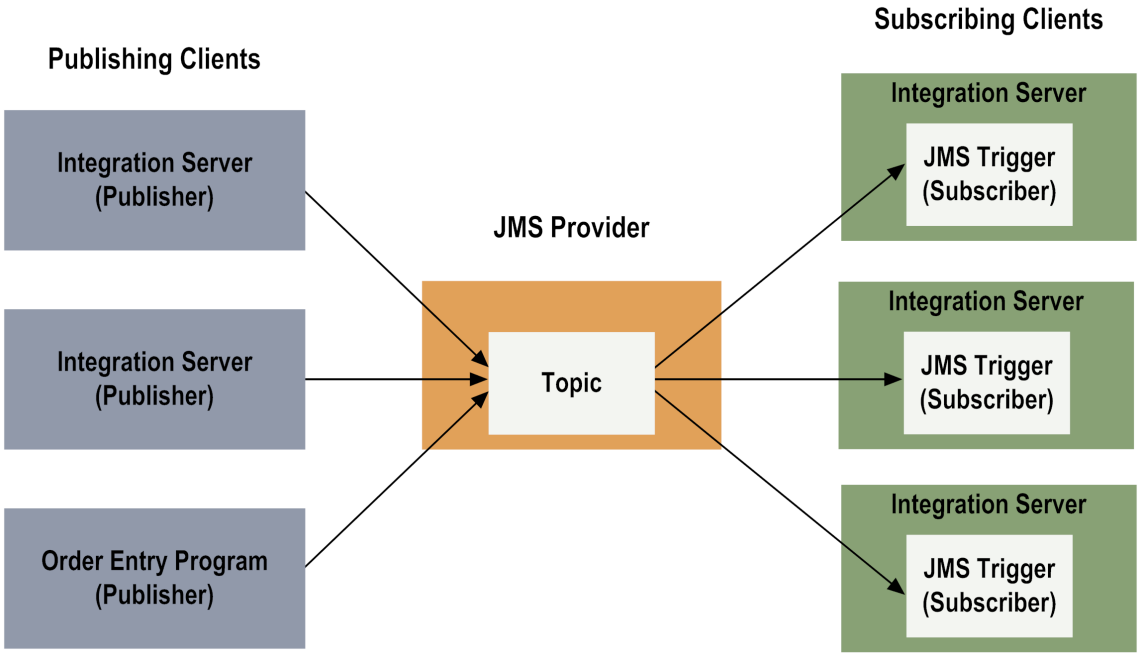


Publish-Subscribe Messaging

In publish-subscribe messaging, message producers and consumers are known as *publishers* and *subscribers*.

The central concept in the publish-subscribe messaging is a destination called a *topic*. Message publishers send messages of specified topics. Clients that want to receive that type of message subscribe to the topic.

The publishers and subscribers never communicate with each other directly. Instead, they communicate by exchanging messages through a JMS provider.



Publishers and subscribers have a timing dependency. Clients that subscribe to a topic can consume only messages published after the client has created a subscription. In addition, the subscriber must continue to be active to consume messages.

The messaging APIs relax this dependency by making a distinction between *durable subscriptions* and *non-durable subscriptions*.

Durable Subscriptions

Durable subscriptions allow subscribers to receive all the messages published on a topic, including those published while the subscriber is inactive. When the subscribing applications are not running, the messaging provider holds the messages in nonvolatile storage. It retains the messages until one of the following occurs:

- The subscribing application becomes active, identifies itself to the provider, and sends an acknowledgment of receipt of the message.
- The expiration time for the messages is reached.

Non-durable Subscriptions

Non-durable subscriptions allow subscribers to receive messages on their chosen topic only if the messages are published while the subscriber is active. You generally use this type of subscription for any kind of data that is time sensitive, such as financial information.

JMS API Programming Model

The following section summarizes the most important components of the JMS API.

The building blocks of a JMS application consist of the following:

- Administered objects (connection factories and destinations)
- Connections
- Sessions
- Message producers
- Message consumers
- Messages

Administered Objects

Administered objects are pre-configured objects that an administrator creates for use with JMS client programs. Administered objects serve as the bridge between the client code and the JMS provider.

By design, the messaging APIs separate the task of configuring administered objects from the client code. This architecture maximizes portability: the provider-specific work is delegated to the administrator rather than to the client code. However, the implementation must supply its own set of administrative tools to configure the administered objects.

JMS administered objects are stored in a standardized namespace called the Java Naming and Directory Interface (JNDI). JNDI is a Java API that provides naming and directory functionality to Java applications. JNDI provides a way to store and retrieve objects by a user supplied name.

Types of Administered Objects

There are two types of administered objects: *connection factories* and *destinations*.

Connection Factories

A *connection factory* is the object a client uses to create a connection with a JMS provider. It encapsulates the set of configuration parameters that a JMS administrator defines for a connection.

The type of connection factory determines whether a connection is made to a topic (in a publish-subscribe application), a connection is make to a queue (in a point-to-point application), or a connection can be made to both (generic connection). The connection factory type also determines whether messages are managed like elements in a distributed transaction in the client application.

You use XA-based connection factories in JMS applications managed by an application server, in the context of a distributed transaction.

Destinations

Destinations are the objects that a client uses to specify the target of messages it produces and the source of messages it consumes. These objects specify the identity of a destination to a JMS API method. Four types of destinations exist; only the first two (queues and topics) are administered objects.

- *Queue*. An object that covers a provider-specific queue name. This object is how a client specifies the identity of a queue to JMS methods.
- *Topic*. An object that covers a provider-specific topic name. This object is how a client specifies the identity of a topic to JMS methods.
- *Temporary Queue*. A queue object created for the duration of a particular connection (or `QueueConnection`). It can only be consumed by the connection from which it was created.
- *Temporary Topic*. A topic object that is created for the duration of a particular connection (or `TopicConnection`). It can only be consumed by the connection from which it was created.

Connections

A *connection* object is an active connection from a client to its JMS provider. In JMS, connections support concurrent use. A connection serves the following purposes:

- A connection encapsulates an open connection with a JMS provider. It typically represents an open TCP/IP socket between a client and the service provider software.
- The creation of a connection object is the point where client authentication takes place.
- A connection object can specify a unique client identifier.
- A connection object supports a user-supplied `ExceptionListener` object.

A connection should always be closed when it is no longer needed.

Sessions

A *session* object is a single-threaded context for producing and consuming messages. If a client uses different threads for different paths of message execution, then a session must be created for each of the threads.

A session is used to create message producers, message consumers, temporary topics, and temporary queues; it also supplies provider-optimized message factories.

In JMS, a session provides the context for grouping a set of send and receive messages into a transactional unit.

Message Producer

A *message producer* is an object that a session creates to send messages to a destination (a topic or a queue).

Message Consumer

A *message consumer* is an object that a session creates to receive messages sent to a destination. A message consumer allows a client to register interest in a destination, which manages the delivery of messages to the registered consumers of that destination.

Message Selector

A client may want to receive subsets of messages. A *message selector* allows a client to filter the messages it wants to receive by use of a SQL92 string expression in the message header. That expression is applied to properties in the message header (not to the message body content) containing the value to be filtered.

If the SQL expression evaluates to true, the message is sent to the client; if the SQL expression evaluates to false, it does not send the message.

Messages

Messages are objects that communicate information between client applications. Following are descriptions of several key concepts related to JMS messages.

Message Structure

Messages are composed of the following parts:

- **Header.** All messages support the same set of header fields. Header fields contain predefined values that allow clients and providers to identify and route messages. Each of the fields supports its own `set` and `get` methods for managing data. Some fields are set automatically by the `send` and `publish` methods, whereas others must be set by the client.

  Examples of header fields include:

  - `JMSDestination`, which holds a `destination` object representing the destination to which the message is to be sent.
  - `JMSMessageID`, which holds a unique message identifier value and is set automatically.
  - `JMSCorrelationID`, which is used to link a reply message with its requesting message. This value is set by the client application.

- `JMSReplyTo`, which is set by the client and takes as a value a `Destination` object representing where the reply is being sent. If no reply is being sent, this field is set to null.

- **Properties (optional).** Properties are used to add optional fields to the message header. Several types of message property fields exist:
  - *Application-specific properties* are typically used to hold message selector values. Message selectors are used to filter and route messages.
  - *Standard properties.* The API provides some predefined property names that a provider may support. Support for the `JMSXGroupID` and `JMSXGroupSeq` is required; however, support for all other standard properties is optional.
  - *Provider-specific properties* are unique to the messaging provider and typically refer to internal values.
- **Body (optional).** The JMS standard defines various types of message body formats that are compatible with most messaging styles. Each form is defined by a message interface.
  - `StreamMessage`. A message whose body contains a stream of Java primitive values. It is filled and read sequentially.
  - `MapMessage`. A message whose body contains a set of name-value pairs where names are Strings and values are Java primitive types. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
  - `TextMessage`. A message whose body contains a `java.lang.String`.
  - `ObjectMessage`. A message that contains a Serializable Java object.
  - `BytesMessage`. A message that contains a stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format. In many cases, it will be possible to use one of the other, self-defining, message types instead.

Both `StreamMessage` and `MapMessage` support the same set of primitive data types. Conversions from one data type to another are possible.

Message Acknowledgment

A message is not considered to be successfully consumed until it is acknowledged. Depending on the session acknowledgment mode, the messaging provider may send a message more than once to the same destination. Several message acknowledgment constants exist.

Value	Description
`AUTO_ACKNOWLEDGE`	Automatically acknowledges the successful receipt of a message.

Value	Description
`CLIENT_ACKNOWLEDGE`	Acknowledges the receipt of a message when the client calls the message's `acknowledge`() method.
`DUPS_OK_ACKNOWLEDGE`	Instructs the session to automatically, lazily acknowledge the receipt of messages, which reduces system overhead but may result in duplicate messages being sent.

2 Working with JMS Triggers

A JMS trigger subscribes to destinations (queues or topics) on a JMS provider and then specifies how Integration Server processes messages the JMS trigger receives from those destinations. Integration Server and Designer support two types of JMS triggers:

- **Standard JMS triggers** use routing rules to specify which services can process messages received by the trigger. The trigger service in the routing rule receives the entire JMS message as an IData.
- **SOAP- JMS triggers** are used to receive JMS messages that contain SOAP messages. When a SOAP-JMS trigger receives a message, Integration Server extracts the SOAP message from the JMS message and passes the SOAP message to the internal web services stack. The web services stack processes the message according to the web service descriptor specified in the SOAP-JMS request.

  Note: WS endpoint triggers are SOAP-JMS triggers. However, WS endpoint triggers can be created and managed using Integration Server Administrator only. For more information about WS endpoint triggers, see *webMethods Integration Server Administrator's Guide*.

Standard JMS triggers and SOAP-JMS triggers can be transacted or non-transacted triggers. The transactionality of a JMS trigger along with the trigger type affect the properties and functionality that can be configured for the trigger.

Note: Information about using Integration Server for JMS is located in *webMethods Integration Server Administrator's Guide*, *webMethods Service Development Help*, and *Using webMethods Integration Server to Build a Client for JMS*.

- *webMethods Integration Server Administrator's Guide* contains information about how to configure Integration Server to work with a JMS provider, how to create a WS endpoint trigger, and how to manage JMS triggers at run time.
- *webMethods Service Development Help* includes this Working with JMS Triggers topic which provides procedures for using Designer to create JMS triggers and set JMS trigger properties.
- *Using webMethods Integration Server to Build a Client for JMS* contains information such as how to build services that send and receive JMS messages, how Integration Server works with cluster policies when sending JMS messages, and detailed information regarding how Integration Server performs exactly-once processing. For completeness, *Using webMethods Integration Server to Build a Client for JMS* also includes the Working with JMS Triggers topic that appears in *webMethods Service Development Help*.

About SOAP-JMS Triggers

A SOAP-JMS trigger is a JMS trigger that receives SOAP over JMS messages and routes the SOAP message to the web services stack for processing. More specifically, the

SOAP-JMS trigger receives JMS messages from a destination (queue or topic) on the JMS provider. Note that a SOAP-JMS trigger can specify a message selector which limits the messages the SOAP-JMS trigger receives from that destination. Integration Server extracts the SOAP message and passes it to the internal web services stack for processing. Integration Server also retrieves JMS message properties that it passes to the web services stack, including targetService, soapAction, contentType, and JMSMessageID. These properties specify the web service descriptor and operation for which the SOAP request is intended. The web services stack then processes the SOAP message according to the web service descriptor (for example, executing request handlers) and invokes the web service operation specified in the SOAP request message.

A SOAP-JMS trigger is associated with one or more provider web service descriptors via a provider web service endpoint alias. The provider web service endpoint alias specifies the SOAP-JMS trigger that receives messages from destinations on the JMS provider. The provider web service endpoint alias is assigned to a JMS binder in a provider web service descriptor. In this way, SOAP-JMS triggers act as listeners for provider web service descriptors.

Note: Even though a SOAP-JMS trigger is associated with one or more provider web service descriptors, the SOAP-JMS trigger can pass any SOAP-JMS message to the web services stack for processing.

The properties assigned to the SOAP-JMS trigger determine how Integration Server acknowledges the message, provides exactly-once processing, or handles transient or fatal errors.

While SOAP-JMS triggers and standard JMS triggers share many properties and characteristics, some properties available to standard JMS triggers are not available to SOAP-JMS triggers, specifically:

- SOAP-JMS triggers can subscribe to one destination only. Consequently, SOAP-JMS triggers do not have joins. Designer does not display the **Join expires** and **Expire after** properties for a SOAP-JMS trigger.
- SOAP-JMS triggers use web services to process the payload of the JMS message. Designer does not display the Message Routing table for SOAP-JMS triggers.
- SOAP-JMS triggers cannot be used to perform ordered service execution. Standard JMS triggers use multiple routing rules and local filters to perform ordered service execution. Because SOAP-JMS triggers do not use routing rules, SOAP-JMS triggers cannot be used to perform ordered service execution.
- A SOAP-JMS trigger, specifically a connection for a SOAP-JMS trigger, can process only one message at a time. Batch processing is not available for SOAP-JMS triggers. Designer does not display the **Max batch processing** property for SOAP-JMS triggers.
- A transacted SOAP-JMS trigger (one that executes as part of a transaction) has additional requirements and limitations when used with web service descriptors. For more information, see the *Web Services Developer's Guide*.

Overview of Building a Non-Transacted JMS Trigger

Building a JMS trigger is a process that involves the following basic stages.

Stage 1 Create a new JMS trigger on Integration Server.

During this stage, you use Designer to create the new JMS trigger on the Integration Server where you will do your development and testing.

Stage 2 Specify a JMS connection alias.

During this stage, you specify the JMS connection alias that Integration Server uses to create connections to the JMS provider. The transaction type of the JMS connection alias determines whether or not the JMS trigger receives and processes messages as part of transaction.

Stage 3 Specify JMS destinations and message selectors.

During this stage, you specify the destinations (queues or topics) on the JMS provider to which the JMS trigger subscribes. That is, the destination is the source of the messages that the JMS trigger consumes. You also specify any message selectors that you want the JMS provider to use to filter the messages it enqueues for the JMS trigger.

Stage 4 Create routing rules (for standard JMS triggers only).

During this stage, you specify the service that Integration Server invokes when the standard JMS trigger receives messages. You can also specify a local filter that Integration Server applies to messages.

Stage 5 Set JMS trigger properties.

During this stage, you determine the type of message processing, the acknowledgement mode, fatal and transient error handling, and exactly-once processing.

Stage 6 Test and debug the JMS trigger.

During this stage, you test and debug the trigger using the tools provided by Integration Server. For more information, see "Debugging a JMS Trigger" on page 60.

Standard JMS Trigger Service Requirements

The service that processes a message received by a standard JMS trigger is called a *trigger service*. Each routing rule in a standard JMS trigger specifies a single trigger service.

Before a JMS trigger can be enabled, the trigger service must already exist on the same Integration Server.

The signature for the trigger service must reference one of the following specifications:

- Use pub.jms:triggerSpec as the specification reference if the trigger service will process one message at a time.
- Use pub.jms:batchTriggerSpec as the specification reference if the trigger service will process multiple messages at one time. That is, the trigger service will receive a batch of messages as input and process all of those messages in a single execution. A trigger that receives and processes a batch of messages is sometimes referred to as a *batch trigger*.

Creating a JMS Trigger

When you create a JMS trigger, keep the following points in mind:

- The JMS connection alias you want Integration Server to use to obtain connections to and receive messages from the JMS provider must already exist. Although a JMS connection alias does not need to be enabled at the time you create the JMS trigger, the JMS connection alias must be enabled for the JMS trigger to execute at run time.

  Note: If you want to manage destinations and durable subscribers on a webMethods Broker that is used as a JMS provider, the JMS connection alias must be enabled when you work with the JMS trigger.

- If you use a JNDI provider to store JMS administered objects, the Connection Factories that you want the JMS trigger to use to consume messages must already exist.
- If you use a JNDI provider to store JMS administered objects and the JMS provider is not webMethods Broker, the destinations (queues and topics) from which this JMS trigger will receive messages must already exist.
- If the JMS provider is webMethods Broker, webMethods Universal Messaging, or webMethods Nirvana the destinations (queues and topics) from which the JMS trigger receives messages do not need to exist before you create the JMS trigger. Instead, you can create destinations using the JMS trigger editor. You can also create, modify, and delete durable subscribers via the JMS trigger. For more information, see "Managing Destinations and Durable Subscribers on the JMS Provider through Designer " on page 34.

- The transaction type of the JMS connection alias determines whether or not the JMS trigger is transacted (that is, it receives and processes messages as part of a transaction). Transacted JMS triggers have slightly different properties and operate differently than non-transacted JMS triggers. For more information about building a transacted JMS trigger, see "Building a Transacted JMS Trigger" on page 61.
- The trigger service that you want to specify in the routing rule must already exist on the same Integration Server on which you create the JMS trigger. For more information, see "Standard JMS Trigger Service Requirements" on page 25.
- A standard JMS trigger can contain multiple routing rules. Each routing rule must have a unique name. For more information about using multiple routing rules, see "Building Standard JMS Triggers with Multiple Routing Rules" on page 36.
- A standard JMS trigger that contains an All (AND) or Only one (XOR) join can only have one routing rule and cannot have a batch processing size (**Max batch messages** property) greater than 1. A JMS trigger with an Any (Or) join can have multiple routing rules. For more information about batch processing, see "About Batch Processing for Standard JMS Triggers" on page 44.
- Integration Server uses a consumer to receive messages for a JMS trigger. This consumer encapsulates the actual javax.jms.MessageConsumer and javax.jms.Session.

To create a JMS trigger

1. In the Package Navigator view of Designer, click **File > New > JMS Trigger**.
2. In the New JMS Trigger dialog box, select the folder in which you want to save the JMS trigger.
3. In the **Element name** field, type a name for the JMS trigger using any combination of letters, numbers, and/or the underscore character.
4. Click **Finish**.
5. In the **JMS connection alias name** field in the Trigger Settings tab, click ....

   Note: A transacted JMS connection alias cannot be assigned to a JMS trigger if a cluster policy is applied to the connection factory used by the JMS connection alias.

6. In the Select a JMS connection alias for *triggerName* dialog box, select the JMS connection alias that you want this JMS trigger to use to receive messages from the JMS provider. Click **OK**.

   Designer sets the **Transaction type** property to match the transaction type specified for the JMS connection alias.

   If a JMS connection alias has not yet been configured on Integration Server, Designer displays a message stating the JMS subsystem has not been configured. For information abut creating a JMS connection alias, see *webMethods Integration Server Administrator's Guide*.

7. In the **JMS trigger type** list, select one of the following:

Select	To...
Standard	Create a standard JMS trigger.
SOAP-JMS	Create a SOAP-JMS trigger.

8. Under **JMS destinations and message selectors**, specify the destinations from which the JMS trigger will receive messages. For more information, see "Adding JMS Destinations and Message Selectors to a JMS Trigger" on page 28.

> **Note:** For SOAP-JMS triggers, you can specify one destination only.

9. If you selected multiple destinations, select the join type. The join type determines whether Integration Server needs to receive messages from all, any, or only one of destinations to execute the trigger service.

Select...	If you want...
All (AND)	Integration Server to invoke the trigger service when the trigger receives a message from every destination within the join time-out period. The messages must have the same activation.
Any (OR)	Integration Server to invoke the trigger service when the trigger receives a message from any of the specified destinations. This is the default join type. **Note:** Using an Any (OR) join is similar to creating multiple JMS triggers that listen to different destinations. While a JMS trigger with an Any (OR) join will use fewer resources (a single thread will poll each destination for messages), it may cause a decrease in performance (it may take longer for one thread to poll multiple destinations).
Only one (XOR)	Integration Server to invoke the trigger service when it receives a message from any of the specified destinations. For the duration of the join time-out period, the Integration Server discards any messages with the same activation that the trigger receives from the specified destinations.

10. If this is a standard JMS trigger, under **Message routing**, add routing rules. For more information, see "Adding Routing Rules to a Standard JMS Trigger" on page 33.
11. In the Properties view, set properties for the JMS trigger.
12. Enter comments or notes, if any, in the **Comments** tab.
13. Click **File > Save**.

Adding JMS Destinations and Message Selectors to a JMS Trigger

The destination is the queue or topic to which the JMS trigger subscribes on the JMS provider. When a JMS trigger subscribes to a topic, you can also indicate whether Integration Server creates a durable subscriber or a non-durable subscriber for the topic.

To add a JMS destination to a JMS trigger

1. In the Package Navigator view of Designer, open the JMS trigger.
2. In the Trigger Settings tab, under **JMS destinations and message selectors**, click  .
3. In the **Destination Name** column, do one of the following to specify the destination from which you want the JMS trigger to receive messages.
   - If the JMS connection alias uses JNDI to retrieve administered objects, specify the lookup name of the Destination object.
   - If the JMS connection alias uses the native webMethods API to connect directly to Broker, specify the provider-specific name of the destination.
   - If the JMS connection alias creates a connection on Broker, Universal Messaging, or Nirvana, click  to select from a list of existing destinations. You can also create a destination and then select it. After you select the destination, click **OK**.

     If the **Order By** mode for the selected destination does not match the existing message processing mode, Designer prompts you to change the processing mode. This situation can occur only when the JMS provider is Broker.

     For instructions for creating a destination, see "Creating a Destination on the JMS Provider" on page 30.
4. In the **Destination Name** column, in the **Destination Type** column, select the type of destination:

Select...	If...
Queue	The destination is a queue. This is the default.
Topic	The destination is a topic.

Select...	If...
Topic (Durable Subscriber)	The destination is a topic for which there is a durable subscriber.

Note: Designer populates **Destination Type** automatically if you selected a destination from the list of existing destinations on the JMS provider.

5. In the **JMS Message Selector** column, click ... . In the Enter JMS Message Selector dialog box, enter the expression that you want to use to receive a subset of messages from this destination and click **OK**.

   For more information about creating a JMS message selector, see "Creating a Message Selector" on page 33.

6. If you specified the destination type as **Topic (Durable Subscriber)**, in the **Durable Subscriber Name** column, do one of the following:

   - Enter a name for the durable subscriber.
   - If the JMS connection alias creates a connection on Broker, Universal Messaging, or Nirvana click ... to select from a list of existing durable subscribers for the topic. In the Durable Subscriber List dialog box select the durable subscriber and click **OK**.

     If the durable subscriber that you want this JMS trigger to use does not exist, you can create it by entering in the name in the **Durable Subscriber Name** column. The name must be unique for the connection where the connection name is the client ID of the JMS connection alias. Broker, Universal Messaging, or Nirvana will create the durable subscriber name using the client ID of the JMS connection alias and the specified durable subscriber name.

Note: Designer populates **Durable Subscriber Name** automatically if you selected a Topic (Durable Subscriber) destination from the list of existing destinations on Broker or Universal Messaging.

7. If you want the JMS trigger to ignore messages sent using the same JMS connection alias as the JMS trigger, select the check box in the **Ignore Locally Published** column. This property applies only when the Destination Type is **Topic** or **Topic (Durable Subscriber)**.

Note: If the JMS connection alias specified for this trigger has the **Create New Connection per Trigger** option enabled, then **Ignore Locally Published** will not work. For the JMS trigger to ignore locally published messages, the publisher and subscriber must share the same connection. When the JMS connection alias uses multiple connections per trigger, the publisher and subscriber will not share the same connection.

8. Repeat this procedure for each destination from which you want the JMS trigger to receive messages.

9. Click **File > Save**.

Notes:

- If you specify a new durable subscriber name and the JMS connection alias that the JMS trigger uses to retrieve messages is configured to manage destinations, Integration Server creates a durable subscriber for the topic when the JMS trigger is first enabled.
- If you specify a destination type of **Topic (Durable Subscriber)** but do not specify a durable subscriber name, Designer changes the destination type to **Topic** when you save the JMS trigger.

Creating a Destination on the JMS Provider

If the JMS connection alias that the JMS trigger uses to retrieve messages is configured to manage destinations, you can create a destination on the JMS provider while using the JMS trigger editor.

Keep the following points in mind when creating destinations using Designer:

- The JMS connection alias used by the JMS trigger must use Universal Messaging, Nirvana, or Broker as the JMS provider.

  Note: Prior to version 9.5 SP1, webMethods Universal Messaging was named webMethods Nirvana.

- The JMS connection alias used by the JMS trigger must be configured to manage destinations.
- The JMS connection alias must be enabled when you work with the JMS trigger.
- If the JMS connection alias creates a connection on a Broker in a Broker cluster, you will not be able to create a destination at the Broker.

To create a destination on the JMS provider

1. In the Package Navigator view of Designer, open the JMS trigger that uses a JMS connection alias that connects to the JMS provider on which you want to create the destinations.
2. In the Trigger Settings tab, under **JMS destinations and message selectors**, click  .
3. In the **Destination Name** column, click  .
4. In the Destination List dialog box, click **Create New Destination**.
5. In the Create New Destination dialog box, provide the following information:

In this field...	Specify...
Destination Name	A name for the destination.

In this field...	Specify...
Destination Key	A name for the destination key. If you do not specify a destination key, Integration Server uses the destination name as the destination key. In the Destination List, when a destination has a destination key Designer displays the destination name using this format: *destinationKey* (*destinationName*)
Destination Type	The type of destination. Select one of the following:

Select...	To...
Queue	The destination is a queue. This is the default.
Topic	The destination is a topic.
Topic (Durable Subscriber)	The destination is a topic for which you want to create a durable subscriber.

Durable Subscriber Name	A name for the durable subscriber. The name must be unique for the connection, where the connection name is the client ID of the JMS connection alias. The JMS provider (Broker, Universal Messaging, or Nirvana) will create the durable subscriber name using the client ID of the JMS connection alias and the specified durable subscriber name. This field only applies if the destination is **Topic (Durable Subscriber)**.
Order By	How Broker distributes messages received by this destination. This field only apples if the JMS provider used y the trigger JMS connection alias is the Broker and the destination is **Queue**.

Select...	To...
Publisher	Distribute messages received by this destination one at a time in the

In this field...	Specify...
	order in which they were received from the publisher.
	None — Distribute the messages received by this destination in any order. This is the default.

Note: An order mode of publisher maps to a serial message processing mode. An order mode of none maps to a concurrent message processing mode.

6. Click **OK** to create the destination.
7. If you want the current JMS trigger to retrieve messages from the new destination, select the destination and click **OK**.

   Designer adds the destination to the **JMS destinations and message selectors** list. If the **Order By** mode for the new destination does not match the existing message processing mode, Designer prompts you to change the processing mode.

Notes:

- Integration Server adds the new destination to the Broker as a shared-state client.
- If you specify a destination type of **Topic (Durable Subscriber)** but do not specify a durable subscriber name, Designer changes the destination type to **Topic** when you save the JMS trigger.

About Durable and Non-Durable Subscribers

When a JMS trigger receives messages from a topic, you can specify whether or not the JMS trigger is a durable subscriber.

A *durable subscriber* establishes a durable subscription with a unique identity on the JMS provider. A *durable subscription* allows subscribers to receive all the messages published on a topic, including those published while the subscriber is inactive (for example, if the JMS trigger is disabled). When the associated JMS trigger is disabled, the JMS provider holds the messages in guaranteed storage. If a durable subscription already exists for the specified durable subscriber on the JMS provider, this service resumes the subscription.

A *non-durable subscription* allows subscribers to receive messages on their chosen topic only if the messages are published while the subscriber is active. A non-durable subscription lasts the lifetime of its message consumer. Note that non-durable subscribers cannot receive messages in a load-balanced fashion.

Creating a Message Selector

If you want the JMS trigger to receive a subset of messages from a specified destination, create a message selector. A message selector is an expression that specifies the criteria for the messages in which the JMS trigger is interested.

The JMS provider applies the message selector to messages it receives. If the selector evaluates to true, the message is sent to the JMS trigger. If the selector evaluates to false, the message is not sent to the JMS trigger.

By creating message selectors, you can delegate some filtering work to the JMS provider. This can preserve Integration Server resources that otherwise would have been spent receiving and processing unwanted messages.

The message selector must use the message selector syntax specified in the Java Message Service standard. The message selector can reference header and property fields in the JMS message only.

Note: If you want to filter on the contents of the JMS message body, write a local filter. Integration Server evaluates a local filter after the JMS trigger receives the message from the JMS provider. Only standard JMS triggers can use local filters.

Adding Routing Rules to a Standard JMS Trigger

The routing rule specifies the service that Integration Server invokes when the standard JMS trigger receive a message from a destination.

To add a routing rule to a standard JMS trigger

1. In the Package Navigator view of Designer, open the JMS trigger.
2. In the Trigger Settings tab, under **Message routing**, click  to add a new routing rule.
3. In the **Name** column, type a name for the routing rule. By default Designer assigns the first rule the name “Rule 1”.
4. In the **Service** column, click  to navigate to and select the service that you want to invoke when Integration Server receives messages from the specified destinations.
5. In the **Local Filter** column, click  to enter the filter that you want Integration Server to apply to messages this JMS trigger receives. For more information about creating a local filter, see "Creating a Local Filter" on page 33.
6. Click **File > Save**.

Creating a Local Filter

You can further refine the messages received and processed by a standard JMS trigger by creating local filters. A local filter specifies criteria for the contents of the message

body. Integration Server applies a local filter to a message after the JMS trigger receives the message from the JMS provider. If the message meets the filter criteria, Integration Server executes the trigger service specified in the routing rule. If the message does not meet the filter criteria, Integration Server discards the message and acknowledges the message to the JMS provider.

If a JMS trigger contains multiple routing rules to support ordered service execution, you can use local filters to process a series of messages in a particular order. For more information about ordered service execution, see "Building Standard JMS Triggers with Multiple Routing Rules" on page 36.

When creating a local filter, you can omit the *JMSMessage* document from the filter expression even though it is part of the pipeline provided to the JMS trigger service. For example, a filter that matches those messages where the value of the *myField* field is "XYZ" would look like the following:

```
%properties/myField% == "XYZ"
```

Note that even though the properties field is a child of the *JMSMessage* document, the *JMSMessage* document does not need to appear in the filter expression.

The following filter matches those messages where the *data* document within the *JMSMessage /body* document contains a field named *myField* whose value is "A":

```
%body/data/myField% == "A"
```

Note: When receiving a batch of messages, Integration Server evaluates the local filter against the first message in the batch only. Integration Server does not apply the filter to subsequent messages in the batch. For more information about batch processing, see "About Batch Processing for Standard JMS Triggers" on page 44.

Managing Destinations and Durable Subscribers on the JMS Provider through Designer

When editing a JMS trigger in Designer, you can create and manage destinations and durable subscribers on webMethods Universal Messaging, webMethods Nirvana, or webMethods Broker. Specifically, you can do the following:

- Create a destination.
- Create and delete a durable subscriber.
- Select the destination from which you want the JMS trigger to receive messages from a list of existing destinations.
- Select a durable subscriber that you want the JMS trigger to use from a list of existing durable subscribers for a specified topic.
- Change the Shared State or Order By mode for a queue or durable subscriber by changing the message processing mode of the JMS trigger. You can do this only when Broker is the JMS provider only.

Designer uses the JMS connection alias specified by the JMS trigger to make the changes on the JMS provider. To manage destinations on the JMS provider, the JMS connection alias that the JMS trigger uses must be

- Configured to manage destinations
- Enabled when you create and edit the JMS trigger.
- To manage destinations on Broker, Integration Server must be version 8.0 SP1 or higher.
- To manage destinations on Universal Messaging, Integration Server must be version 9.0 SP1 or higher.

> **Note:** Prior to version 9.5 SP1, webMethods Universal Messaging was named webMethods Nirvana.

For a complete list of the requirements for using Designer to manage destinations and durable subscribers on the JMS provider, see *webMethods Integration Server Administrator's Guide*.

> **Note:** The ability to use Designer to manage JMS destinations on Broker, Nirvana, and Universal Messaging is a design-time feature. In a production environment, this functionality should be disabled.

Modifying Destinations or Durable Subscribers via a JMS Trigger in Designer

If a JMS trigger uses a JMS connection alias that is configured to manage destinations, you can modify the destination or durable subscribers while editing a JMS trigger. Changes to destinations or durable subscriptions can result in unused durable subscriptions on the JMS provider. Changing destinations can make the JMS trigger out of sync with the destination. For example, when using the Broker, modifying the destination could result in out of sync Shared State or Order By mode settings.

When you make a change that results in a change to a destination or durable subscriber, Designer informs you about the necessary change and then prompts you to confirm making change to the destination or durable subscriber on the JMS provider.

For example, if you change the name of the durable subscriber for a Topic (Durable Subscriber) destination, Designer displays a message stating, "By making this change the trigger will no longer subscribe to durable subscriber *oldDurableSubscriberName* . Would you like to remove this durable subscriber from the JMS provider?" If you confirm the change, Integration Server removes the durable subscriber from Broker. If you do not confirm the change, the durable subscriber will remain on Broker. You will need to use the Broker interface in My webMethods to remove the durable subscriber.

> **Note:** If another client, such as another JMS trigger, currently connects to the queue or durable subscriber that you want to modify or remove, then Integration Server cannot update or remove the queue or durable subscriber. If the JMS

provider is Broker, updates must be made through My webMethods. If the JMS provider is Universal Messaging, updates must be made through Universal Messaging Enterprise Manager. If the JMS provider is Nirvana, updates must be made through Nirvana Enterprise Manager.

For more information about managing destinations and durable subscriptions on the JMS provider, see "Managing Destinations and Durable Subscribers on the JMS Provider through Designer " on page 34.

Building Standard JMS Triggers with Multiple Routing Rules

A JMS trigger can contain more than one routing rule. Each routing rule can specify a different local filter and a different service to invoke.

You might create multiple routing rules so that a JMS trigger processes a group of messages in a specific order. Each routing rule might execute a different trigger service based on the contents or type of message received. When a JMS trigger receives a message, Integration Server determines which service to invoke by evaluating the local filters for each routing rule.

Integration Server evaluates the routing rules in the same order in which the rules appear in the editor. It is possible that a message could satisfy more than one routing rule. However, Integration Server executes only the service associated with the first satisfied routing rule and ignores the remaining routing rules. Therefore, the order in which you list routing rules on the editor is important.

You might want to use multiple routing rules to control service execution when a service that processes a message depends on successful execution of another service. For example, to process a purchase order, you might create one service that adds a new customer record to a database, another that adds a customer order, and a third that bills the customer. The service that adds a customer order can only execute successfully if the new customer record has been added to the database. Likewise, the service that bills the customer can only execute successfully if the order has been added. You can ensure that the services execute in the necessary order by creating a trigger that contains one routing rule for each expected message.

Note: SOAP-JMS triggers do not have routing rules.

Guidelines for Building a JMS Trigger that Performs Ordered Service Execution

Use the following general guidelines to build a JMS trigger that performs ordered service execution.

- Because the JMS provider cannot guarantee message order across destinations, the JMS trigger must specify a single destination. That is, the JMS trigger cannot include a join.

- Each routing rule, except the last one, must contain a local filter. For example, you might create a filter based on a custom property that the sending client adds to the message. Integration Server uses the local filters to differentiate between the messages. Without a local filter, only the first routing rule would ever execute.
- Routing rules must appear in the order in which you want the messages to be processed. Each routing rule must have a unique name.
- Set the **Processing mode** property to serial to ensure that the Integration Server processes the messages in the same order in which the JMS trigger receives them. Serial processing ensures that the services that process the messages do not execute at the same time.
- Set **Max batch messages** to 1 (the default). When a trigger service processes a batch of messages, Integration Server only applies the filter to the first message in the batch.

Important: Messages must be sent to JMS provider in the same order in which you want the messages to be processed.

Enabling or Disabling a JMS Trigger

You can enable or disable a JMS trigger.

Note: If you disable a SOAP-JMS trigger that acts as a listener for one or more provider web service descriptors, Integration Server will not retrieve any messages for those web service descriptors.

To enable or disable a JMS trigger

1. In the Package Navigator view of Designer, open the JMS trigger that you want to enable or disable.
2. In the Properties view, under **General**, set the **Enabled** property to one of the following:

Select...	To...
True	Enable a JMS trigger that is currently disabled.
False	Disable a JMS trigger that is currently enabled.

3. Click **File > Save**.

Notes:

- When you disable a JMS trigger, Integration Server interrupts any server threads that are processing messages. If the JMS trigger is currently processing messages, Integration Server waits 3 seconds before forcing the JMS trigger to stop processing

messages. If it does not complete within 3 seconds, Integration Server stops the message consumer used to receive messages for the JMS trigger and closes the JMS consumer. At this point the server thread for the JMS trigger may continue to run to completion. However, the JMS trigger will not be able to acknowledge the message when processing completes. If the message is persistent, this can lead to duplicate messages.

- You can disable one or more JMS triggers using the pub.triggers:disableJMSTriggers service.
- You can enable one or more JMS triggers using the pub.triggers:enableJMSTriggers service.
- You can enable, disable, and suspend one or more JMS triggers using Integration Server Administrator.

JMS Trigger States

A JMS trigger can have one of the following states:

Trigger State	Description
Enabled	The JMS trigger is running and connected to the JMS provider. Integration Server retrieves and processes messages for the JMS trigger.
Disabled	The JMS trigger is stopped. Integration Server neither retrieves nor processes messages for the JMS trigger. The JMS trigger remains in this state until you enable the trigger.
Suspended	The JMS trigger is running and connected to the JMS provider. Integration Server has stopped message retrieval, but continues processing any messages it has already retrieved. Integration Server enables the JMS trigger automatically upon server restart or when the package containing the JMS trigger reloads. **Note:** You can suspend a JMS trigger using Integration Server Administrator or the pub.triggers:suspendJMSTriggers service.

Setting an Acknowledgement Mode

Acknowledgment mode indicates how Integration Server acknowledges messages received on behalf of a JMS trigger. A message is not considered to be successfully consumed until it is acknowledged.

Note: The **Acknowledgement mode** property is not available for transacted JMS triggers. That is, if the JMS connection alias is of type XA_TRANSACTION or LOCAL_TRANSACATION, Designer does not display the **Acknowledgement mode** property.

To set an acknowledgment mode

1. In the Package Navigator view of Designer, open the JMS trigger for which you want to set the acknowledgment mode.
2. In the Properties view, under **General**, select one of the following for **Acknowledgement mode**:

Select...	To...
CLIENT_ACKNOWLEDGE	Acknowledge or recover the message only after the JMS trigger processes the message completely. This is the default.
DUPS_OK_ACKNOWLEDGE	Lazily acknowledge the delivery of messages. This may result in the delivery of duplicate messages.
AUTO_ACKNOWLEDGE	Automatically acknowledge the message when it is received by the JMS trigger. Integration Server will acknowledge the message before the trigger completes processing. The JMS provider cannot redeliver the message if Integration Server becomes unavailable before message processing completes.

3. Click **File > Save**.

About Join Time-Outs

When you create a standard JMS trigger that receives messages from two or more destinations), you create a join. Consequently, you need to specify a join time-out. A *join time-out* specifies how long Integration Server waits for additional messages to fulfill the join. Integration Server starts the join time-out period when it receives the first message that satisfies the join.

The implications of a join time-out are different depending on the join type.

Note: You need to specify a join time-out only when the join type is **All (AND)** or **Only one (XOR)**. You do not need to specify a join time-out for an **Any (OR)** join.

Join Time-Outs for All (AND) Joins

A join time-out for an **All (AND)** join specifies how long Integration Server waits for messages from all of the destinations specified in the join.

When a JMS trigger receives a message that satisfies part of an All (AND) join, Integration Server stores the message. Integration Server waits for the JMS trigger to receive messages from the remaining destinations specified in the join. Only messages with the same activation ID as the first received message will satisfy the join.

If Integration Server receives messages from all of the destinations specified in the join before the time-out period elapses, Integration Server executes the service specified in the routing rule. If Integration Server doe not receive messages from all of the destinations before the time-out period elapses, Integration Server discards the messages and writes a log entry.

When the time-out period elapses, the next message that satisfies the **All (AND)** join causes the time-out period to start again.

Join Time-Outs for Only One (XOR) Joins

A join time-out for an **Only one (XOR)** join specifies how long Integration Server discards instances of the other messages received from the specified destinations.

When a JMS trigger receives a message that satisfies part of an **Only one (XOR)** join, Integration Server executes the service specified in the routing rule. Integration Server starts the join time-out when the JMS trigger receives the message. For the duration of the time-out period, Integration Server discards any messages the JMS trigger receives from a destination specified in the JMS trigger. Integration Server only discards those messages with the same activation ID as the first message.

When the time-out period elapses, the next message that the JMS trigger receives that satisfies the **Only one (XOR)** join causes the trigger service to execute and the time-out period to start again.

Setting a Join Time-Out

When configuring JMS trigger properties, you can specify whether a join times out and if it does, what the time-out period should be. The time-out period indicates how long Integration Server waits for messages from the other destinations specified in the join after Integration Server receives the first message.

Note: You need to specify a join time-out only when the join type is **All (AND)** or **Only one (XOR)**. You do not need to specify a join time-out for an **Any (OR)** join.

To set a join time-out

1. In the Package Navigator view of Designer, open the JMS trigger for which you want to set the join time-out.
2. In the Properties view, under **General**, next to **Join expires**, select one of the following:

Select...	To...
True	Specify that Integration Server should stop waiting for messages from other destinations in the join after the time-out period elapses. In the **Expire after** property, specify the length of the join time-out period. The default join time-out period is 1 day.
False	Specify that the join does not expire. Integration Server should wait indefinitely for messages from the additional destinations specified in the join condition. Set the **Join expires** property to **False** only if you are confident that all of the messages will be received eventually. **Important:** A join is persisted across server restarts.

3. Click **File > Save**.

About Execution Users for JMS Triggers

For a JMS trigger, the execution user indicates which credentials Integration Server should use when invoking services associated with the JMS trigger. When a client invokes a service via an HTTP request, Integration Server checks the credentials and user group membership of the client against the Execute ACL assigned to the service. Integration Server performs this check to make sure that the client is allowed to invoke that service. When a JMS trigger executes, however, Integration Server invokes the service when it receives a message rather than as a result of a client request. Integration Server does not associate user credentials with a message. You can specify which credentials Integration Server should supply when invoking a JMS trigger service by setting an execution user for a JMS trigger.

You can instruct Integration Server to invoke a service using the credentials of one of the predefined user accounts (Administrator, Default, Developer, Replicator). You can also specify a user account that you or another server administrator defined. When Integration Server receives a message for the JMS trigger, Integration Server uses the credentials for the specified user account to invoke the service specified in the routing rule.

Assigning an Execution User to a JMS Trigger

Make sure that the user account you select includes the credentials required by the execute ACL assigned to the services associated with the JMS triggers.

To assign an execution user for a JMS trigger

1. In the Package Navigator view of Designer, open the JMS trigger for which you want to assign the execution user.
2. In the Properties view, under **General**, in the **Execution user** property, type the name of the user account whose credentials Integration Server uses to execute a service associated with the JMS trigger. You can specify a locally defined user account or a user account defined in a central or external directory.
3. Click **File > Save**.

About Message Processing

Message processing determines how Integration Server processes the messages received by the JMS trigger. You can specify serial processing or concurrent processing.

- In serial processing, Integration Server processes messages received by a JMS trigger one after the other in the order in which the messages were received from the JMS provider.
- In concurrent processing, Integration Server processes messages received from the JMS provider in parallel.

Serial Processing

In serial processing, Integration Server processes messages received by a JMS trigger one after the other in the order in which the messages were received from the JMS provider. Integration Server uses a single thread for receiving and processing a message for a serial JMS trigger. Integration Server evaluates the first message it receives, determines which routing rule the message satisfies, and executes the service specified in the routing rule. Integration Server waits for the service to finish executing before processing the next message received from the JMS provider.

If you want to process messages in the same order in which JMS clients sent the messages to the JMS provider, you will need to configure the JMS provider to ensure that messages are received by the JMS trigger in the same order in which the messages are published.

For information about using serial JMS triggers in a cluster to process messages from a single destination in publishing order, see the *Using webMethods Integration Server to Build a Client for JMS*.

Tip: If your trigger contains multiple routing rules to handle a group of messages that must be processed in a specific order, use serial processing.

Concurrent Processing

In concurrent processing, Integration Server processes messages received from the JMS provider in parallel. That is, Integration Server processes as many messages for the JMS triggers as it can at the same time, using a separate server thread to process each message. Integration Server does not wait for the service specified in the routing rule to finish executing before it begins processing the next message. You can specify the maximum number of messages Integration Server can process concurrently. This equates to specifying the maximum number of server threads that can process messages for the JMS trigger at one time.

Concurrent processing provides faster performance than serial processing. Integration Server processes the received messages more quickly because it can process more than one message for the trigger at a time. However, the more messages Integration Server processes concurrently, the more server threads it dispatches, and the more memory the message processing consumes.

Additionally, for JMS triggers with concurrent processing, Integration Server does not guarantee that messages are processed in the order in which they are received.

A concurrent trigger can connect to the JMS provider through multiple connections, which can increase trigger throughout. For more information about multiple connections, refer to "Using Multiple Connections to Retrieve Messages for a Concurrent JMS Trigger" on page 45.

Message Processing and Message Consumers

Integration Server uses a consumer to receive messages for a JMS trigger. This consumer encapsulates the actual javax.jms.MessageConsumer and javax.jms.Session. The type of message processing affects how Integration Server uses consumers to receive messages.

Serial JMS triggers have one consumer and will use one thread from the server thread pool to receive and process a message.

Concurrent JMS triggers use a pool of consumers to receive and process messages. Each consumer uses one thread from the server thread pool to receive and process a message. For a concurrent JMS trigger, the **Max execution threads** property specifies how many threads can be used to process messages for the trigger at one time. For concurrent JMS triggers, Integration Server also dedicates a thread to managing the pool of consumers. Consequently, the maximum number of threads that can be used by a JMS trigger is equal to the **Max execution threads** value plus 1. For example, a concurrent JMS trigger configured to use 10 threads at a time can use a maximum of 11 server threads.

When there are multiple connections to the Broker, the threads are divided among the connections. Therefore, if a trigger is configured so that **Connection count** is 2 and **Max**

execution threads is set to 10, each connection will have 5 threads plus 1, for a total of 12 threads.

Message Processing and Load Balancing

Load balancing allows multiple consumers on one or more Integration Servers to retrieve and process messages concurrently. Load balancing is necessary for concurrent JMS triggers regardless of whether or not they are running in a cluster of Integration Servers. This is because concurrent JMS triggers use multiple consumers. Each consumer receives a message from the JMS provider, processes the message, and acknowledges the message to the JMS provider. Each consumer needs to consume a message from the same destination, but not process any duplicate message. For information about configuring load-balancing, see *webMethods Integration Server Administrator's Guide*.

About Batch Processing for Standard JMS Triggers

You can configure a standard JMS trigger and its associated trigger service to process a group or "batch" of messages at one time. Batch processing can be an effective way of handling a high volume of small messages for the purposes of persisting them or delivering them to another back-end resource. For example, you might want to take a batch of messages, create a packet of SAP IDocs, and send the packet to SAP with a single call. Alternatively, you might want to insert multiple messages into a database at one time using only one insert. The trigger service processes the messages as a unit as opposed to in a series.

The **Max batch messages** property indicates the maximum number of messages that the trigger service can receive at one time. For example, if the **Max batch messages** property is set to 5, Integration Server passes the trigger service up to 5 messages received by the JMS trigger to process during a single execution.

Integration Server uses one consumer to receive and process a batch of messages. During pre-processing, Integration Server checks the maximum delivery count for each message and, if exactly-once processing is configured, determines whether or not the message is a duplicate. Integration Server then bundles the message into a single IData and passes it to the trigger service. If the message has exceeded the maximum delivery count or is a duplicate message, Integration Server does not include it in the message batch sent to the trigger service.

Note: The watt.server.jms.trigger.maxDeliveryCount property determines the maximum number of times the JMS provider can deliver a message to a JMS trigger.

Integration Server acknowledges all the messages received in a batch from the JMS provider at one time. This includes messages that failed pre-processing. As described by the Java Message Service standard, when a client acknowledges one message, the client acknowledges all of the messages received by the session. Because Integration Server uses a consumer that includes a javax.jms.MessageConsumer and a javax.jms.Session, when Integration Server acknowledges one message in the batch, it effectively acknowledges all the messages received in the batch.

If a batch of messages is not acknowledged or they are recovered back to the JMS provider, the JMS provider can redeliver all of the messages in the batch to the JMS trigger. However, when using webMethods Broker, Integration Server can acknowledge individual messages that fail pre-processing.

Guidelines for Configuring Batch Processing

When configuring JMS trigger for batch processing, keep the following in mind:

- The trigger service must be coded to handle multiple messages as input. That is, the trigger service must use the pub.jms.batchTriggerSpec as the service signature.
- When receiving a batch of messages, Integration Server evaluates the local filter in the routing rule against the first message in the batch only.
- A transacted JMS trigger can be used for batch processing if the JMS connection alias used by the trigger connects to a JMS provider that supports reuse of transacted JMS sessions. If the JMS provider does not support reuse of transacted JMS sessions, set **Max batch processing** to 1.

  Consult the documentation for your JMS provider to determine whether or not the JMS provider supports the reuse of transacted JMS sessions. Note that webMethods Broker version 8.2 and higher, webMethods Universal Messaging version 9.5 SP1 and higher, and webMethods Nirvana version 7 and higher support the reuse of transacted JMS sessions.
- A JMS trigger that contains an All (AND) or Only one (XOR) join cannot use batch processing.
- SOAP-JMS triggers cannot process messages in batches.

Using Multiple Connections to Retrieve Messages for a Concurrent JMS Trigger

You can configure a concurrent JMS trigger to obtain multiple connections to the JMS provider. Multiple connections can improve trigger throughput. Keep in mind, however, that each connection used by the JMS trigger requires a dedicated Integration Server thread, regardless of the current throughput.

For a JMS trigger to have multiple connections to the JMS provider, the JMS connection alias used by the trigger must be configured to create a new connection for each trigger. For more information about JMS connection aliases, refer to *webMethods Integration Server Administrator's Guide*.

A concurrent JMS trigger can use multiple connections to retrieve messages from a JMS provider. For a trigger to use multiple connections, the following must be true:

- The JMS trigger must be configured for concurrent processing. Serial JMS triggers cannot use multiple connections.

- The JMS trigger must receive messages from a queue or from a topic with a durable subscriber. JMS triggers that receive messages from non-durable subscribers (topics) cannot use multiple connections.
- The JMS trigger must not have the **Ignore locally published** option selected when the JMS connection alias is configured to use the **Create New Connection per Trigger** option. For the JMS trigger to ignore locally published messages, the publisher and subscriber must share the same connection. When the JMS connection alias uses multiple connections per trigger, the publisher and subscriber will not share the same connection.
- The JMS connection alias used by the JMS trigger must be configured to create an individual connection for each trigger. To configure a JMS alias to create individual connections for each JMS trigger, select the **Create New Connection per Trigger** option on the **Settings > Messaging > JMS Settings > JMS Connection Alias** screen on Integration Server Administrator.

Note: When using multiple connections to the Broker, Integration Server uses a different client ID for each JMS trigger that uses the JMS connection alias. However, when Integration Server connects to other JMS providers, it uses the same client ID for each connection. Some JMS providers do not permit multiple connections to use the same client ID to retrieve messages from a Topic with a durable subscriber. Review the JMS provider documentation before configuring the use of multiple connections for a JMS connection alias and any concurrent JMS triggers that use the JMS connection alias.

Retrieving Multiple Messages for a JMS Trigger with Each Request

You can instruct Integration Server to retrieve multiple messages for a JMS trigger with each request by using the prefetch cache. When a JMS trigger is configured to use the prefetch cache, Integration Server retrieves multiple messages for the trigger each time Integration Server requests more messages from the Broker. When the JMS trigger needs a new message to process, the JMS trigger retrieves the message from the local, prefetched cache instead of requesting a new message from the Broker. Use of the prefetch cache may improve performance of the JMS trigger because it reduces the time spent retrieving messages for the JMS trigger.

Using the prefetch cache is most likely to improve performance for JMS triggers that process many small messages and have trigger services that execute quickly. If the JMS trigger receives large messages or the JMS trigger has long-running trigger services, using the prefetch cache may increase the overall time needed to retrieve and process a message. For JMS triggers that fit this use case, including concurrent JMS triggers, reducing the number of prefetched messages may actually decrease the time needed to retrieve and process a message. You may need to set the number of prefetched messages to 1 (one).

Note: This prefetch cache can be used with JMS triggers that receive messages from Broker only.

The use of the prefetch cache for a JMS trigger and the number of messages Integration Server might retrieve with each request are determined by the **Max prefetch size** property for the JMS trigger and the value of the watt.server.jms.trigger.maxPrefetchSize parameter.

- When the **Max prefetch size** property is greater than 0, Integration Server uses the prefetch cache with the JMS trigger. The **Max prefetch size** property value specifies the number of messages that Integration Server might retrieve and cache for the trigger. The default is 10.
- When the **Max prefetch size** property is set to -1, Integration Server uses the prefetch cache with the JMS trigger. The watt.server.jms.trigger.maxPrefetchSize parameter value determines how many messages Integration Server might retrieves and cache for the JMS trigger.
- When the **Max prefetch size** property is set to 0, Integration Server does not use the prefetch cache with the JMS trigger.

When the prefetch cache is in use and the number of messages retrieved by Integration Server is greater than one, the same server thread might process all of the messages retrieved by the prefetch request. This is true even for concurrent JMS triggers. The first thread for the concurrent JMS trigger processes the first set of prefetched messages. The second thread for the concurrent JMS trigger processes the second set of prefetched messages.

For example, suppose that the number of available messages is 22, **Max execution threads** is 4, and **Max prefetch size** is 10. In the initial request for messages, the first server thread may retrieve 10 messages. The same server thread will process these first 10 messages. The second server thread may retrieve 10 messages, all of which will be processed by the second server thread. The third server thread may retrieve the remaining 2 messages, both of which will be processed by the third server thread. While the concurrent JMS trigger can use up to 4 server threads, Integration Server might use only 3 server threads to retrieve and process messages due to the way in which a JMS trigger processes prefetched messages. A concurrent JMS trigger will use all of the configured execution threads to process messages only when the number of messages on the Broker is greater than the number of messages that can be prefetched.

Note: When you are working with a cluster of Integration Servers, the prefetch behavior might appear at first to be misleading. For example, suppose that you have a cluster of two Integration Servers. Each Integration Server contains the same JMS trigger. Twenty messages are sent to a destination from which JMS trigger receives messages. It might be expected the JMS trigger on Integration Server 1 will receive the first message, the JMS trigger on Integration Server 2 will receive the second message, and so forth. However, what may happen is that the JMS trigger on Integration Server 1 will receive the first 10 messages and the JMS trigger on Integration Server 2 will receive the second 10 messages.

Configuring Message Processing

Keep the following points in mind when configuring message processing for a JMS trigger:

- You can configure a standard JMS trigger and its associated trigger service to process a group or "batch" of messages at one time. For information about batch processing, see "About Batch Processing for Standard JMS Triggers" on page 44 and "Guidelines for Configuring Batch Processing" on page 45.
- If the JMS provider from which the JMS trigger retrieves messages does not support concurrent access by durable subscribers, you must set the **Max execution threads** property to 1 for the concurrent JMS trigger. Consult the documentation for your JMS provider for more information.
- Non-durable subscribers, i.e., JMS triggers that subscribe to topics but do not specify a durable subscriber, cannot receive messages in a load-balanced fashion. Because it is possible for a JMS trigger using a non-durable subscriber to process duplicates of a message, set **Max execution threads** to 1.
- For a destination that acts as a shared state client, the serial processing mode corresponds to a shared state order mode of publisher; a concurrent processing mode corresponds to a shared state order mode of none.
- If you use webMethods Broker as the JMS provider, changing the message processing mode for a JMS trigger can create a mismatch with the corresponding destination on the Broker. If you do not use Designer to make the changes, you need to use the Broker interface of My webMethods to update the destination.
- A concurrent JMS trigger can use multiple connections to retrieve messages from the JMS provider. For information about requirements for using multiple connections, see "Using Multiple Connections to Retrieve Messages for a Concurrent JMS Trigger" on page 45.
- You can only use the **Max prefetch** property with webMethods Broker.

To configure message processing for a JMS trigger

1. In the Package Navigator view of Designer, open the JMS trigger for which you want to specify message processing.
2. In the Properties view, under **Messaging processing**, next to **Processing mode**, select one of the following:

Select...	To...
Serial	Specify that Integration Server should process messages received by the trigger one after the other.

Select...	To...
Concurrent	Specify that Integration Server should process multiple messages for this trigger at one time. In the **Max execution threads** property, specify the maximum number of messages that Integration Server can process concurrently.

3. If you want this trigger to perform batch processing, next to **Max batch messages**, specify the maximum number of messages that the trigger service can receive at one time. If you do not want the trigger to perform batch processing, leave this property set to 1. The default is 1.
4. If you want this trigger to use multiple connections to receive messages from the JMS provider, next to **Connection count**, specify the number of connections you want the JMS trigger to make to the JMS provider. The default is 1.
5. If you want Integration Server to use the prefetch cache with this JMS trigger, in the Properties view, under **webMethods Broker** do one of the following for **Max prefetch size**:
   - Specify the number of messages you want Integration Server to retrieve and cache for this JMS trigger. The default is 10 messages.
   - Specify -1 if you want the value of watt.server.jms.trigger.maxPrefetchSize parameter to determine how many messages Integration Server retrieves and caches for the JMS trigger.
   - Specify 0 if you do not want to use the prefetch cache with this JMS trigger.
6. Click **File > Save**.

   If the destination is Queue or Topic (Durable Subscriber) and the JMS trigger is connected to the queue or durable subscriber, Designer prompts you to update the corresponding destination on the Broker with the changed shared state order mode, click **Yes** to update the destination. Click **No** to skip the destination update. Note that messages might be lost while Designer and Integration Server make the update because Integration Server deletes and recreates the subscription as part of the update.

   Note: A JMS trigger is connected to the Broker when the specified JMS connection alias is enabled and connected to the Broker.

Fatal Error Handling for Non-Transacted JMS Triggers

You can specify that Integration Server suspend a JMS trigger automatically if a fatal error occurs during trigger service execution. A fatal error occurs when the trigger service ends because of an exception.

If a trigger service ends because of an exception, and you configured the JMS trigger to suspend on fatal errors, Integration Server suspends the trigger and acknowledges the message to the JMS provider. The JMS trigger remains suspended until one of the following occurs:

- You enable the trigger using the pub.trigger:enableJMSTriggers service.
- You enable the trigger using Integration Server Administrator.
- Integration Server restarts or the package containing the trigger reloads. (When Integration Server suspends a trigger because of a fatal error, Integration Server considers the change to be temporary. For more information about temporary vs. permanent state changes for triggers, see *webMethods Integration Server Administrator's Guide*.)

Automatic suspension of a trigger can be especially useful for serial triggers that are designed to process a group of messages in a particular order. If the trigger service ends in error while processing the first message, you might not want the trigger to proceed with processing the subsequent messages in the group. If Integration Server automatically suspends the trigger, you have an opportunity to determine why the trigger service did not execute successfully.

Important: If you disable or suspend a SOAP-JMS trigger that acts as a listener for one or more provider web service descriptors, Integration Server will not retrieve any messages for those web service descriptors until the trigger is enabled.

You can handle the exception that causes the fatal error by configuring Integration Server to generate JMS retrieval failure events for fatal errors and by creating an event handler that subscribes to JMS retrieval failure events. Integration Server passes the event handler the contents of the JMS message as well as information about the exception.

Integration Server handles fatal errors for transacted JMS differently than for non-transacted JMS triggers. For information about fatal error handling for transacted JMS triggers, see "Fatal Error Handling for Transacted JMS Triggers" on page 65.

Configuring Fatal Error Handling for Non-Transacted JMS Triggers

To configure fatal error handling for a non-transacted JMS trigger

1. In the Package Navigator view of Designer, open the JMS trigger for which you want to specify document processing.
2. In the Properties view, under **Fatal error handling**, set the **Suspend on error** property to **True** if you want Integration Server to suspend the trigger when a trigger service ends with an error. Otherwise, select **False**. The default is **False**.
3. Click **File > Save**.

Transient Error Handling for Non-Transacted JMS Triggers

When building a JMS trigger, you can specify what action Integration Server takes when the trigger service fails because of a transient error caused by a run-time exception. A *transient error* is an error that arises from a temporary condition that might be resolved or corrected quickly, such as the unavailability of a resource due to network issues or failure to connect to a database. Because the condition that caused the trigger service to fail is temporary, the trigger service might execute successfully if Integration Server waits and then re-executes the service.

A *run-time exception* (specifically, an ISRuntimeException) occurs in the following situations:

- The trigger service catches and wraps a transient error and then re-throws it as an ISRuntimeException.
- The web service operation that processes the message received by a SOAP-JMS trigger catches and wraps a transient error and then re-throws it as an ISRuntimeException.

  Note: For a service handler invoked by a SOAP-JMS trigger, Integration Server treats all errors as fatal. Service handlers invoked by SOAP-JMS triggers cannot be retried.

- A pub.jms:send, pub.jms:sendAndWait, or pub.jms:reply service fails because a resource (such as the JNDI provider or JMS provider) is not available.
- If the JMS provider is not available, and the settings for the pub.jms* service indicate that Integration Server should write messages to the client side queue, Integration Server does not throw an ISRuntimeException.
- A transient error occurs on the back-end resource for an adapter service. Adapter services built on Integration Server 6.0 or later, and based on the ART framework, detect and propagate exceptions that signal a retry automatically if a transient error is detected on their back-end resource.

Note: A web service connector that sends a JMS message can throw an ISRuntimeException, such as when the JMS provider is not available. However, Integration Server automatically places the ISRuntimeException in the *fault* document returned by the web service connector. If you want the parent flow service to catch the transient error and re-throw it as an ISRuntimeException, you must code the parent flow service to check the *fault* document for an ISRuntimeException and then throw an ISRuntimeException explicitly.

You can also configure Integration Server and/or a JMS trigger to handle transient errors that occur during trigger preprocessing. The trigger preprocessing phase encompasses

the time from when a trigger first receives a message from it's local queue on Integration Server to the time the trigger service executes.

For more information about transient error handling for trigger preprocessing, see "Transient Error Handling During Trigger Preprocessing" on page 111.

About Retry Behavior for Trigger Services

When you configure transient error handling for a non-transacted JMS trigger, you specify the following retry behavior:

- Whether Integration Server should retry trigger services for the standard JMS trigger. Keep in mind that a trigger service can retry only if it is coded to throw ISRuntimeExceptions. For more information, see "Service Requirements for Retrying a Trigger Service" on page 52.
- For a SOAP-JMS trigger, whether Integration Server should retry web service operation that throw and an ISRuntimeException.

  Note: Integration Server does not apply the SOAP-JMS trigger transient error handling behavior to service handlers executed as part of processing web services. Integration Server treats all errors thrown by service handler as fatal errors.

- The maximum number of retry attempts Integration Server should make for each trigger service.
- The time interval between retry attempts.
- How to handle a retry failure. That is, you can specify what action Integration Server takes if all the retry attempts are made and the trigger service or web service operation still fails because of an ISRuntimeException. For more information about handling retry failures, see "Handling Retry Failure" on page 53.

Service Requirements for Retrying a Trigger Service

To be eligible for retry, the trigger service or web service operation must do one of the following to catch a transient error and re-throw it as an ISRuntimeException:

- If the trigger service or web service operation is a flow service, the trigger service must invoke pub.flow:throwExceptionForRetry. For more information about the pub.flow:throwExceptionForRetry, see the *webMethods Integration Server Built-In Services Reference*.
- If the trigger service or web service operation is written in Java, the service can use com.wm.app.b2b.server.ISRuntimeException(). For more information about constructing ISRuntimeExceptions in Java services, see the *webMethods Integration Server Java API Reference* for the com.wm.app.b2b.server.ISRuntimeException class.

When a service invokes a pub.jms* service that sends a JMS message and the service fails because a resource needed by the pub.jms* service is not available, Integration Server automatically detects and propagates an ISRuntimeException.

Adapter services built on Integration Server 6.0 or later, and based on the ART framework, detect and propagate exceptions that signal a retry if a transient error is detected on their back-end resource. This behavior allows for the automatic retry when the service functions as a trigger service.

Note: Integration Server does not retry a trigger service that fails because a ServiceException occurred. A ServiceException indicates that there is something functionally wrong with the service. A service can throw a ServiceException using the EXIT step.

Handling Retry Failure

Retry failure occurs for a standard JMS trigger when Integration Server makes the maximum number of retry attempts and the trigger service still fails because of an ISRuntimeException. Retry failure occurs for a SOAP-JMS trigger when Integration Server makes the maximum number of retry attempts to process a web service operation and the operation still fails because of an ISRuntimeException.

When you configure retry properties, you can specify one of the following actions to determine how Integration Server handles retry failure for a trigger.

- **Throw exception.** When Integration Server exhausts the maximum number of retry attempts, Integration Server treats the last trigger service or web service operation failure as a service error. This is the default behavior.
- **Suspend and retry later.** When Integration Server reaches the maximum number of retry attempts, Integration Server suspends the trigger and then retries the trigger service or web service operation at a later time.

Overview of Throw Exception for Retry Failure

The following table provides an overview of how Integration Server handles retry failure when the **Throw exception** option is selected.

Step	Description
1	Integration Server makes the final retry attempt and the trigger service or web service operation fails because of an ISRuntimeException.
2	Integration Server treats the last trigger service or web service operation failure as a ServiceException.
3	Integration Server rejects the message.

Step	Description
	If the message is persistent, Integration Server returns an acknowledgement to the JMS provider.
4	Integration Server generates a JMS retrieval failure event if the watt.server.jms.trigger.raiseEventOnRetryFailure property is set to true (the default).
5	If the JMS trigger is configured to suspend on error when a fatal error occurs, Integration Server suspends the JMS trigger. Otherwise, Integration Server processes the next message for the JMS trigger.

In summary, the default retry failure behavior (**Throw exception**) rejects the message and allows the trigger to continue with message processing when retry failure occurs for a trigger service.

Overview of Suspend and Retry Later for Retry Failure

The following table provides more information about how the **Suspend and retry later** option works.

Step	Description
1	Integration Server makes the final retry attempt and the trigger service or web service operation fails because of an ISRuntimeException.
2	Integration Server suspends the JMS trigger temporarily. **Note:** The change to the trigger state is temporary. Message processing will resume for the trigger if Integration Server restarts, the trigger is enabled or disabled, or the package containing the trigger reloads. You can also enable triggers manually using Integration Server Administrator or by invoking the pub.trigger:enableJMSTriggers service. **Important:** If you disable or suspend a SOAP-JMS trigger that acts as a listener for one or more provider web service descriptors, Integration Server will not retrieve any messages for those web service descriptors until the SOAP-JMS trigger is enabled.
3	Integration Server recovers the message back to the JMS provider. This indicates that the required resources are not ready to process the message and makes the message available for processing at a later time. For serial triggers, it also ensures that the message maintains its position at the top of trigger queue.

Step	Description
4	Optionally, Integration Server schedules and executes a resource monitoring service. A *resource monitoring service* is a service that you create to determine whether the resources associated with a trigger service are available. A resource monitoring service returns a single output parameter named *isAvailable* .
5	If the resource monitoring service indicates that the resources are available (that is, the value of *isAvailable* is true), Integration Server enables the trigger. Message processing and message retrieval resume for the JMS trigger. If the resource monitoring service indicates that the resources are not available (that is, the value of *isAvailable* is false), Integration Server waits a short time interval (by default, 60 seconds) and then re-executes the resource monitoring service. Integration Server continues executing the resource monitoring service periodically until the service indicates the resources are available. **Tip:** You can change the frequency with which the resource monitoring service executes by modifying the value of the watt.server.jms.trigger.monitoringInterval property.
6	After Integration Server resumes the JMS trigger, Integration Server passes the message to the trigger. The trigger and trigger service (or web service operation) process the message just as they would any message received by the JMS trigger. **Note:** At this point, the retry count is set to 0 (zero).

In summary, the **Suspend and retry later** option provides a way to resubmit the message programmatically. It also prevents the trigger from retrieving and processing other messages until the cause of the transient error condition has been remedied.

Configuring Transient Error Handling for a Non-Transacted JMS Trigger

The transient error handling and retry behavior that you specify for a non-transacted JMS trigger determines how Integration Server handles retry failure caused by transient errors during trigger service execution. The selected behavior also determines how Integration Server handles transient errors that occur during trigger preprocessing.

For more information about transient error handling for trigger preprocessing, see "Transient Error Handling During Trigger Preprocessing" on page 111.

Note: If you do not configure service retry for a trigger, set the **Max retry attempts** property to 0. Because managing service retries creates extra overhead, setting this property to 0 can improve the performance of services invoked by the trigger.

To configure transient error handling for a non-transacted JMS trigger

1. In the Package Navigator view of Designer, open the JMS trigger for which you want to configure retry behavior.
2. In the Properties view, under **Transient error handling**, in the **Max retry attempts field**, specify the maximum number of times Integration Server should attempt to re-execute the trigger service. The default is 0 retries (the trigger service does not retry).
3. In the **Retry interval** property, specify the time period the Integration Server waits between retry attempts. The default is 10 seconds.
4. Set the **On retry failure** property to one of the following:

Select...	To...
Throw exception	Specify that Integration Server should throw a service exception when the last allowed retry attempt ends because of an ISRuntimeException. This is the default.
Suspend and retry later	Specify that Integration Server should recover the message back to the JMS provider and suspend the trigger when the last allowed retry attempt ends because of an ISRuntimeException. **Note:** If you want Integration Server to automatically enable the trigger when the trigger's resources become available, you must provide a resource monitoring service that Integration Server can execute to determine when to resume the trigger.

5. If you selected **Suspend and retry later**, then in the **Resource monitoring service** property specify the service that Integration Server should execute to determine the availability of resources associated with the trigger service. Multiple triggers can use the same resource monitoring service. For information about building a resource monitoring service, see *Using webMethods Integration Server to Build a Client for JMS*.
6. Click **File > Save**.

Notes:

- Standard JMS triggers and services can both be configured to retry. When a trigger invokes a service (that is, the service functions as a trigger service), Integration Server uses the trigger retry properties instead of the service retry properties.
- SOAP-JMS triggers and services used as operations in provider web service descriptors can both be configured to retry. When a web service operation processes a message received by a SOAP-JMS trigger, Integration Server uses the trigger retry properties instead of the service (operation) retry properties.
- Integration Server does not retry service handlers invoked by a SOAP-JMS trigger.
- When Integration Server retries a trigger service and the trigger service is configured to generate audit data on error, Integration Server adds an entry to the audit log for each failed retry attempt. Each of these entries will have a status of "Retried" and an error message of "Null". However, if Integration Server makes the maximum retry attempts and the trigger service still fails, the final audit log entry for the service will have a status of "Failed" and will display the actual error message. Integration Server makes the audit log entry regardless of which retry failure option the trigger uses.
- Integration Server generates the following journal log message between retry attempts:

  [ISS.0014.0031D] Service *serviceName* failed with ISRuntimeException. Retry *x* of *y* will begin in *retryInterval* milliseconds.
- You can invoke the pub.flow:getRetryCount service within a trigger service to determine the current number of retry attempts made by Integration Server and the maximum number of retry attempts allowed for the trigger service. For more information about the pub.flow:getRetryCount service, see the *webMethods Integration Server Built-In Services Reference*.

Exactly-Once Processing for JMS Triggers

Within Integration Server, exactly-once processing is a facility that ensures one-time processing of a persistent message by a JMS trigger. The trigger does not process duplicates of the message. Integration Server provides exactly-once processing when all of the following are true:

- The message is persistent.
- The JMS trigger has an acknowledgement mode set to `CLIENT_ACKNOWLEDGE`.
- Exactly-once properties are configured for the JMS trigger.

Note: Software AG recommends that if you want to use exactly-once processing for JMS triggers subscribing to topics, make sure the topic uses a durable subscriber.

Duplicate Detection Methods for JMS Triggers

Integration Server ensures exactly-once processing by performing duplicate detection and by providing the ability to retry trigger services. *Duplicate detection* determines whether the current message is a copy of one previously processed by the trigger.

Duplicate messages can be introduced in to the webMethods system in the following situations:

- The sending client sends the same message more than once.
- When receiving persistent messages from the JMS provider, Integration Server and the JMS provider lose connectivity before the JMS trigger processes and acknowledges the message. The JMS trigger will receive the message again when the connection is restored.

Integration Server uses duplicate detection to determine the message's status. The message status can be one of the following:

- **New.** The message is new and has not been processed by the trigger.
- **Duplicate.** The message is a copy of one already processed the trigger.
- **In Doubt.**Integration Server cannot determine the status of the message. The trigger may or may not have processed the message before.

To resolve the message status, Integration Server evaluates, in order, one or more of the following:

- **Delivery count** indicates how many times the JMS provider has delivered the message to the JMS trigger.
- **Document history database** maintains a record of all persistent message IDs processed by JMS triggers that have an acknowledgment mode of `CLIENT_ACKNOWLEDGE` and for which exactly-once processing is configured.
- **Document resolver service** is a service created by a user to determine the message status. The document resolver service can be used instead of or in addition to the document history database.

The steps that Integration Server takes to determine a message's status depend on the exactly-once properties configured for the JMS trigger.

Note: For detailed information about exactly-once processing for messages received by JMS triggers, see *Using webMethods Integration Server to Build a Client for JMS*.

Configuring Exactly-Once Processing for a JMS Trigger

Configure exactly-once processing for a JMS trigger when you want the trigger to process persistent messages once and only once. If it is acceptable for a trigger service to

process duplicates of a message, you should not configure exactly-once processing for the trigger.

Keep the following points in mind when configuring exactly-once processing:

- Integration Server can perform exactly-once processing for persistent messages only. The sending client must set the *JMSDeliveryMode* to persistent.
- The JMS trigger must specify `CLIENT_ACKNOWLEDGE` for the acknowledgement mode.
- You do not need to configure all three methods of duplicate detection. However, if you want to ensure exactly-once processing, you must use a document history database or implement a custom solution using the document resolver service.

  A document history database offers a simpler approach than building a custom solution and will typically catch all duplicate messages. There may be exceptions depending on your implementation. For more information about these exceptions, see "Building a Transacted JMS Trigger" on page 61. To minimize these exceptions, it is recommended that you use a history database and a document resolver service.
- Stand-alone Integration Servers cannot share a document history database. Only a cluster of Integration Servers or a non-clustered group of Integration Servers can (and must) share a document history database.
- Make sure the duplicate detection window set by the **History time to live** property is long enough to catch duplicate messages but does not cause the document history database to consume too many server resources. If sending JMS clients reliably send messages once, you might use a smaller duplicate detection window. If the JMS clients are prone to sending duplicate messages, consider setting a longer duplicate detection window.
- If you intend to use a document history database as part of duplicate detection, you must first install the document history database component and associate it with a JDBC connection pool. For instructions, see *Installing Software AG Products*.

Note: For detailed information about exactly-once processing for messages received by JMS triggers, see *Using webMethods Integration Server to Build a Client for JMS*.

To configure exactly-once processing for a JMS trigger

1. In the Package Navigator view of Designer, open the JMS trigger for which you want to configure exactly-once processing.
2. In the Properties view, under **Exactly Once**, set the **Detect duplicates** property to **True**.
3. To use a document history database as part of duplicate detection, do the following:
   a. Set the **Use history** property to **True**.
   b. In the **History time to live** property, specify how long the document history database maintains an entry for a message processed by this trigger. This value determines the length of the duplicate detection window.

4. To use a service that you create to resolve the status of In Doubt messages, specify that service in the **Document resolver service** property.
5. Click **File > Save**.

Disabling Exactly-Once Processing for a JMS Trigger

If you later determine that exactly-once processing is not necessary for a JMS trigger, you can disable it.

To disable exactly-once processing for a JMS trigger

1. In the Package Navigator view of Designer, open the trigger for which you want to configure exactly-once processing.
2. In the Properties view, under **Exactly Once**, set the **Detect duplicates** property to **False**.

   Designer disables the remaining exactly-once properties.
3. Click **File > Save**.

Debugging a JMS Trigger

To debug and test a JMS trigger you can:

- Instruct Integration Server to produce an extra level of verbose logging. You can enable debug trace logging for all JMS triggers or for individual JMS triggers
- Send messages to which the JMS trigger subscribes to the JMS provider. You can create a service that sends the messages. Alternatively, you can create a launch configuration that publishes a JMS message that contains an instance of a specified IS document type to the JMS provider.

Enabling Trace Logging for All JMS Triggers

To enable debug trace logging for all JMS triggers

1. Open Integration Server Administrator if it is not already open.
2. In the **Settings** menu of the Navigation panel, click **Extended**.
3. Click **Edit Extended Settings**.
4. Under **Extended Settings**, type the following:

   ```
   watt.server.jms.debugTrace=true
   ```
5. Click **Save Changes**.
6. Suspend and then enable all JMS triggers.

   For information about suspending and enabling all JMS triggers at one time, see *webMethods Integration Server Administrator's Guide*.

Enabling Trace Logging for a Specific JMS Trigger

To enable debug trace logging for a specific JMS trigger

1. Open Integration Server Administrator if it is not already open.
2. In the **Settings** menu of the Navigation panel, click **Extended**.
3. Click **Edit Extended Settings**.
4. Under **Extended Settings**, type the following:

   ```
   watt.server.jms.debugTrace.triggerName=true
   ```

   Where *triggerName* is the fully qualified name of the trigger in the format *folder*.*subfolder*:*triggerName*.
5. Click **Save Changes**.
6. Disable and then enable the trigger.

Building a Transacted JMS Trigger

A *transacted JMS trigger* is a JMS trigger that executes within a transaction. A *transaction* is a logical unit of work composed of one or more interactions with one or more resources. The interactions within a transaction are either all committed or all rolled back. A transaction either entirely succeeds or has no effect at all.

For a transacted JMS trigger, Integration Server uses a transacted JMS connection alias to receive messages from the JMS provider and to process the messages. A JMS connection alias is considered to be transacted when it has a transaction type of XA TRANSACTION or LOCAL TRANSACTION.

The execution of a transacted JMS trigger is an implicit transaction. In an implicit transaction, Integration Server starts and completes the transaction automatically, without the need for executing any of the transaction management services.

Integration Server starts the implicit transaction when it uses the specified transacted JMS connection alias to connect to the JMS provider and receive messages for the transacted JMS trigger. Integration Server implicitly commits or rolls back the transaction based on the success or failure of the trigger service.

- Integration Server commits the transaction if the trigger service executes successfully.
- Integration Server rolls back the transaction if the trigger service fails with an ISRuntimeException (a transient error). For detailed information about how Integration Server handles a transient error within a transaction, see "Transient Error Handling for Transacted JMS Triggers" on page 67.
- Integration Server rolls back the transaction if the trigger service fails with a Service Exception (a fatal error). For detailed information about how Integration Server

handles a fatal error within a transaction, see "Fatal Error Handling for Transacted JMS Triggers" on page 65.

Because Integration Server handles the transaction implicitly, you do not need to use any of the transaction management services, such as pub.art.transaction:startTransaction, in the trigger service. However, if the trigger service includes a nested transaction, you can use the transaction management services to explicitly manage the nested transaction.

Like a non-transacted JMS trigger, a transacted JMS trigger specifies a destination from which it would like to receive documents and specifies routing rules to process messages it receives. However, a transacted JMS trigger has some prerequisites as well as some properties that are different from a non-transacted JMS trigger.

Prerequisites for a Transacted JMS Trigger

Before you build a transacted JMS trigger, make sure the following points are true:

- A transacted JMS connection alias exists. A JMS connection alias is considered to be transacted when it has a transaction type of XA TRANSACTION or LOCAL TRANSACTION.

  Note: A transacted JMS connection alias cannot be assigned to a JMS trigger if a cluster policy is applied to the connection factory used by the JMS connection alias.

- The WmART package is installed and enabled.

Properties for Transacted JMS Triggers

Integration Server and Designer provide different properties for a transacted JMS trigger than for a non-transacted JMS trigger. The following list identifies properties that are specific to transacted JMS triggers, specific to non-transacted JMS triggers, or apply to both but must be set to a particular value for transacted JMS triggers.

- For transacted JMS triggers, message acknowledgement is handled by the transaction; the acknowledgement mode does not apply. Consequently, Designer does not display the **Acknowledgement mode** property for a transacted JMS trigger.
- A transacted JMS trigger can only use Any (OR) joins, for which you do not need to specify a join time-out. Because All (AND) and Only one (XOR) joins cannot be used, Designer does not display the **Join expires** and **Expire after** properties for a transacted JMS trigger.
- A transacted JMS trigger can be used for batch processing if the JMS connection alias used by the trigger connects to a JMS provider that supports reuse of transacted JMS sessions. If the JMS provider does not support reuse of transacted JMS sessions, set **Max batch processing** to 1.

  Consult the documentation for your JMS provider to determine whether or not the JMS provider supports the reuse of transacted JMS sessions. Note that webMethods Broker version 8.2 and higher, webMethods Universal Messaging version 9.5 SP1

and higher, and webMethods Nirvana version 7 and higher support the reuse of transacted JMS sessions.

- Because a transaction is an all or nothing situation, a trigger service cannot retry a message if a trigger service ends because of a transient error. Designer does not display the retry properties (**Max retry attempts**, **Retry interval**, and **On retry failure**) for a transacted JMS trigger.
- You can specify how Integration Server handles a transient error that causes the transaction to be rolled back. Designer displays an **On transaction rollback** property that you can use to specify whether Integration Server simply recovers the message from the JMS provider or whether it suspends the JMS trigger in addition to recovering the message. For more information about transient error handling for transacted JMS triggers, see "Transient Error Handling for Transacted JMS Triggers" on page 67.

Steps for Building a Transacted JMS Trigger

Building a transacted JMS trigger is a process that involves the following basic stages.

Stage 1 Create a new JMS trigger on Integration Server.

Stage 2 Specify a JMS connection alias with a transaction type of XA TRANSACTION or LOCAL TRANSACTION.

Stage 3 Specify the destination (queues or topics) on the JMS provider from which you want to receive messages. You also specify any message selectors that you want the JMS provider to use to filter messages for the JMS trigger.

If this a SOAP-JMS trigger, you can specify one destination only.

Stage 4 For a standard JMS trigger, create routing rules and specify the services that Integration Server invokes when the JMS trigger receives messages.

SOAP-JMS triggers do not use routing rules.

Stage 5 Set the following JMS trigger properties:

Property name...	Description
Enabled	Enables or disables a JMS trigger as follows: ■ If set to **True**, enables a JMS trigger that is currently disabled. ■ If set to **False**, disables a JMS trigger that is currently enabled.

Execution user	Name of the user account whose credentials Integration Server uses to execute a service associated with the JMS trigger.
Message processing	Specifies whether Integration Server should process messages serially or concurrently. When set to: ■ **Serial**, Integration Server processes messages received by the trigger one after the other. ■ **Concurrent**, Integration Server processes multiple messages for the trigger at one time.
Fatal error handling > Suspend on error	Specifies whether you want Integration Server to suspend the trigger when a trigger service ends with an error. Select **True** or **False**.
Transient error handling	Specifies how Integration Server responds when a transaction is rolled back due to a transient error that occurs during processing of a transacted JMS trigger. When the **On transaction rollback** property is set to: ■ **Recover only**, Integration Server recovers the message after a transaction is rolled back due to a transient error. This is the default. ■ **Suspend and recover**, Integration Server suspends the JMS trigger and recovers the message after a resource monitoring service indicates that the resources needed by the trigger service are available.
Exactly once	Specifies whether you want the trigger to process persistent messages once and only once. Set **Detect duplicates** to **True** to configure exactly once processing.
Permissions	In Designer, select the ACLs that you want to assign for each level of access as follows: ■ For the **List ACL** permission, specify the ACL whose allowed member can see that the element exists and view the element's metadata (input, output, etc.).

- For the **Read ACL**, specify the ACL whose allowed member can view the source code and metadata of the element.
- For the **Write ACL**, specify the ACL whose allowed member can lock, check out, edit, rename, and delete the element.
- For the **Execute ACL**, specify the ACL whose allowed member can execute the service. This level of access only applies to services and web service descriptors.

Stage 6 Test and debug the JMS trigger. For more information, see "Debugging a JMS Trigger" on page 60.

Fatal Error Handling for Transacted JMS Triggers

You can specify that Integration Server suspend a transacted JMS trigger automatically if a fatal error occurs during trigger service execution. For a standard JMS trigger, a fatal error occurs when the trigger service ends because of a ServiceException. For a SOAP-JMS trigger, a fatal error occurs when the web service operation ends because of a ServiceException.

When a transacted JMS trigger is configured to suspend when a fatal error occurs, Integration Server does the following when the trigger service or web service operation ends with a ServiceException:

Step	Description
1	The trigger service for a transacted JMS trigger fails because of a ServiceException. Or, a web service operation invoked via a transacted SOAP-JMS trigger fails because of a ServiceException.
2	Integration Server rolls back the entire transaction and Integration Server recovers the message back to the JMS provider. The JMS provider marks the message as redelivered and increments the value of the *JMSXDeliveryCount* property in the JMS message.
3	If the JMS trigger is configured to use a document history database for exactly-once processing, Integration Server adds an entry with a status of “completed” for the message to the document history database. Because Integration Server does not acknowledge the message when it is rolled back, the JMS provider makes the message available for redelivery to the JMS trigger. However, a message that causes a trigger service to end because of a Service Exception typically does not process successfully

Step	Description
	upon redelivery. Integration Server adds the "completed" entry so that the message is treated as a duplicate when it is received from the JMS provider. The message is rejected after it is resent. If the JMS trigger does not use a document history database, Integration Server continues to receive and attempt message processing until the message processes successfully or the maximum delivery count has been met. The maximum delivery count determines the maximum number of time the JMS provider can deliver the message to the JMS trigger. It is controlled by the watt.server.jms.trigger.maxDeliveryCount property.
4	Integration Server suspends the JMS trigger. **Important:** If you disable or suspend a SOAP-JMS trigger that acts as a listener for one or more provider web service descriptors, Integration Server will not retrieve any messages for those web service descriptors until the trigger is enabled.
5	The JMS trigger remains suspended until one of the following occurs: ■ You enable the trigger using the pub.trigger:enableJMSTriggers service. ■ You enable the trigger using Integration Server Administrator. ■ Integration Server restarts or the package containing the trigger reloads. (When Integration Server suspends a trigger because of a fatal error, Integration Server considers the change to be temporary. For more information about temporary vs. permanent state changes for triggers, see *webMethods Integration Server Administrator's Guide*.)

You can handle the exception that causes the fatal error by configuring Integration Server to generate JMS retrieval failure events for fatal errors and by creating an event handler that subscribes to JMS retrieval failure events. Integration Server passes the contents of the JMS message and exception information to the event handler.

Configuring Fatal Error Handling for Transacted JMS Triggers

To configure fatal error handling for a transacted JMS trigger

1. In the Package Navigator view of Designer, open the JMS trigger for which you want to specify document processing.
2. In the Properties view, under **Fatal error handling**, set the **Suspend on error** property to **True** if you want Integration Server to suspend the trigger when a trigger service ends with an error. Otherwise, select **False**. The default is **False**.

3. Configure exactly-once processing for the JMS trigger. For more information about configuring exactly-once processing, see "Configuring Exactly-Once Processing for a JMS Trigger" on page 58.
4. Click **File > Save**.

Transient Error Handling for Transacted JMS Triggers

When building a transacted JMS trigger, you can specify what action Integration Server takes when a transient error causes a trigger service or a web service operation to fail and the entire transaction is rolled back.

A *transient error* is an error that arises from a temporary condition that might be resolved or corrected quickly, such as the unavailability of a resource due to network issues or failure to connect to a database. A transient error is caused by a run-time exception. A *run-time exception* (specifically, an ISRuntimeException) occurs in the following situations.

- The trigger service catches and wraps a transient error and then re-throws it as an ISRuntimeException.
- The web service operation that processes the message received by a SOAP-JMS trigger catches and wraps a transient error and then re-throws it as an ISRuntimeException.

  Note: For a service handler invoked by a SOAP-JMS trigger, Integration Server treats all errors as fatal. Service handlers invoked by SOAP-JMS triggers cannot be retried.

- The pub.jms:send, pub.jms:sendAndWait, or pub.jms:reply service fails because a resource (such as the JNDI provider or JMS provider) is not available.

  If the JMS provider is not available, and the settings for the pub.jms* service indicate that Integration Server should write messages to the client side queue, Integration Server does not throw an ISRuntimeException.

- A transient error occurs on the back-end resource for an adapter service. Adapter services built on Integration Server 6.0 or later, and based on the ART framework, detect and propagate exceptions that signal a retry automatically if a transient error is detected on their back-end resource.

Note: A web service connector that sends a JMS message can throw an ISRUntimeException, such as when the JMS provider is not available. However, Integration Server automatically places the ISRuntimeException in the *fault* document returned by the web service connector. If you want the parent flow service to catch the transient error and re-throw it as an ISRuntimeException, you must code the parent flow service to check the *fault* document for an ISRuntimeException and then throw an ISRuntimeException explicitly.

You can specify one of the following transient error handling options for a transacted JMS trigger:

- **Recover only.** After a transaction is rolled back, Integration Server receives the message from the JMS provider almost immediately. This is the default.
- **Suspend and recover.** After a transaction is rolled back, Integration Server suspends the JMS trigger and receives the message from the JMS provider at a later time.

You can also configure Integration Server and/or a JMS trigger to handle transient errors that occur during trigger preprocessing. The trigger preprocessing phase encompasses the time from when a trigger first receives a message from it's local queue on Integration Server b to the time the trigger service executes.

For more information about transient error handling for trigger preprocessing, see "Transient Error Handling During Trigger Preprocessing" on page 111.

Overview of Recover Only for Transaction Rollback

The following table provides an overview of how Integration Server handles transaction rollback when the **Recover Only** option is selected for a transacted JMS trigger.

Step	Description
1	The trigger service web service operation fails because of an ISRuntimeException.
2	Integration Server rolls back the entire transaction. When the transaction is rolled back, Integration Server recovers the message back to the JMS provider automatically. The JMS provider marks the message as redelivered and increments the delivery count (*JMSXDeliveryCount* field in the JMS message). At this point, a JMS provider typically makes the message available for immediate redelivery.
3	Integration Server receives the same message from the JMS provider and processes the message. Because Integration Server receives the message almost immediately after transaction roll back, it is likely that the temporary condition that caused the ISRuntimeException has not resolved and the trigger service will end with a transient error again. Consequently, setting **On transaction rollback** to **Recover only** could result in wasted processing. **Note:** Integration Server enforces a maximum delivery count, which determines the maximum number of time the JMS provider can deliver the message to the JMS trigger. If the maximum delivery count has been met, the JMS provider will not deliver the message

Step	Description
	to the JMS trigger. Instead, the JMS provider will acknowledge and remove the message. The maximum delivery count is controlled by the watt.server.jms.trigger.maxDeliveryCount property.

Overview of Suspend and Recover for Transaction Rollback

The following table provides an overview of how Integration Server handles transaction rollback when the **Suspend and recover** option is selected for a transacted JMS trigger.

Step	Description
1	The trigger service or web service operation fails because of an ISRuntimeException.
2	Integration Server rolls back the entire transaction. When the transaction is rolled back, Integration Server recovers the message back to the JMS provider automatically. The JMS provider marks the message as redelivered and increments the delivery count (*JMSXDeliveryCount* field in the JMS message).
3	Integration Server suspends the JMS trigger temporarily. The JMS trigger is suspended on this Integration Server only. If the Integration Server is part of a cluster, other servers in the cluster can retrieve and process messages for the trigger. **Important:** If you disable or suspend a SOAP-JMS trigger that acts as a listener for one or more provider web service descriptors, Integration Server will not retrieve any messages for those web service descriptors until the SOAP-JMS trigger is enabled. **Note:** The change to the trigger state is temporary. Message processing will resume for the trigger if Integration Server restarts, the trigger is enabled or disabled, or the package containing the trigger reloads. You can also enable triggers manually using Integration Server Administrator or by invoking the pub.trigger:enableJMSTriggers service.
4	Optionally, Integration Server schedules and executes a resource monitoring service. A *resource monitoring service* is a service that you create to determine whether the resources associated with a trigger service are available. A resource monitoring service returns a single output parameter named *isAvailable* .

Step	Description
5	If the resource monitoring service indicates that the resources are available (that is, the value of *isAvailable* is true), Integration Server enables the trigger. Message processing and message retrieval resume for the JMS trigger. If the resource monitoring service indicates that the resources are not available (that is, the value of *isAvailable* is false), Integration Server waits a short time interval (by default, 60 seconds) and then re-executes the resource monitoring service. Integration Server continues executing the resource monitoring service periodically until the service indicates the resources are available. **Tip:** You can change the frequency at which the resource monitoring service executes by modifying the value of the watt.server.jms.trigger.monitoringInterval property.
6	After Integration Server resumes the JMS trigger, Integration Server receives the message from the JMS provider and processes the message. **Note:** If the maximum delivery count has been met, the JMS provider will not deliver the message to the JMS trigger. The maximum delivery count determines the maximum number of time the JMS provider can deliver the message to the JMS trigger. It is controlled by the watt.server.jms.trigger.maxDeliveryCount property.

Configuring Transient Error Handling for Transacted JMS Triggers

The transient error handling and transaction rollback behavior that you specify for a transacted JMS trigger determines how Integration Server handles transaction rollback caused by transient errors during trigger service execution. The selected behavior also determines how Integration Server handles transient errors that occur during trigger preprocessing.

For more information about transient error handling for trigger preprocessing, see "Transient Error Handling During Trigger Preprocessing" on page 111.

Use the following procedure to configure how Integration Server responds when a transaction is rolled back due to a transient error that occurs during processing of a transacted JMS trigger.

To configure transient error handling for a transacted JMS trigger

1. In the Package Navigator view of Designer, open the trigger for which you want to configure transient error handling.
2. In the Properties view, under **Transient error handling**, in the **On transaction rollback** property, select one of the following:

Select...	To...
Recover only	Specify that Integration Server recovers the message after a transaction is rolled back due to a transient error. This is the default.
Suspend and recover	Specify that Integration Server does the following after a transaction is rolled back due to a transient error: ■ Suspends the JMS trigger ■ Recovers the message after a resource monitoring service indicates that the resources needed by the trigger service are available.

3. If you selected **Suspend and recover**, in the **Resource monitoring service** property, specify the service that Integration Server should execute to determine the availability of resources associated with the trigger service or web service operation. Multiple triggers can use the same resource monitoring service.
4. Click **File > Save**.

3 Sending and Receiving JMS Messages

The JMS Services

Using the following JMS services, you can create services that send and/or receive JMS messages. The JMS services are located in the WmPublic package.

Service	Description
pub.jms:acknowledge	Sends an acknowledgement for a message to the JMS provider.
pub.jms:createConsumer	Creates a message consumer to receive messages from destinations on the JMS provider.
pub.jms:receive	Synchronously receives a message from a queue or topic on the JMS provider.
pub.jms:reply	Sends a reply message to a requesting client.
pub.jms:send	Sends a JMS message to the JMS provider.
pub.jms:sendAnd Wait	Sends a request in the form of a JMS message to the JMS provider and, optionally, waits for a reply.
pub.jms:sendBatch	Sends multiple JMS messages to the same destination on the JMS provider.
pub.jms.waitForReply	Retrieves the reply message for an asynchronous request.

Sending a JMS Message

When you build a service that sends a JMS message, you specify how Integration Server connects to the JMS provider, the message destination, and whether or not a client side queue should be used.

How to Send a JMS Message

The following describes the general steps you take to send a JMS message to a JMS provider.

1. Create an empty flow service.

2. Create the message body.

   How you build the message body depends on the format that you want to use for the message. For example, if you want to use a String as the message body, create a field of type String and then add content to the String field. If you want to use a Document (IData) as the message body, create a document and then add content to the document. Note that a Document (IData) should only be used when sending a JMS message from one Integration Server to another.

   If you want more control over the actual javax.jms.Message that Integration Server sends to the JMS provider, you can create a Java service that calls the com.wm.app.b2b.server.jms.producer.ProducerFacade class, and then invoke one of the following methods to create the desired javax.jms.Message:

   createBytesMessage(String)

   createMapMessage(String)

   createObjectMessage(String)

   createStreamMessage(String)

   createTextMessage(String)

   The Java service calling this API must return an Object of type javax.jms.Message, which can then be mapped to the *JMSMessage/body/message* input parameter of the pub.jms:send service.

   Important: If you want to send a StreamMessage or a MapMessage, you need to use the appropriate com.wm.app.b2b.server.jms.producer.ProducerFacade API to create the javax.jms.Message object.

   When you create the javax.jms.Message with the com.wm.app.b2b.server.jms.producer.ProducerFacade, you can use the javax.jms.Message setter methods to set the values of the message headers and properties directly. You can also set the value of message headers and properties using the input parameters of the pub.jms* service that you use to send the message. If you set the message headers and properties both ways, the values provided to the pub.jms* service take precedence.

   Software AG recommends that you use a pub.jms* service to create and send the JMS message. This may provide better performance on average.

3. Invoke pub.jms:send.

   This service creates a JMS message (javax.jms.Message) based on input provided to the service or takes an existing JMS message and sends it to the JMS provider.

   Note: If you want to send multiple JMS messages to the same destination, use pub.jms:sendBatch. For more information about the pub.jms:sendBatch services, see the *webMethods Integration Server Built-In Services Reference*.

4. Specify the JMS connection alias.

The JMS connection alias indicates how Integration Server connects to the JMS provider. The alias also specifies whether the alias uses a client side queue and if Integration Server will retry the pub.jms:send service automatically if the service fails because of a transient error.

Name	Description
connectionAliasName	Name of the JMS connection alias that you want to use to send the message.

5. Specify the destination to which you want to send the message.

   If the JMS connection alias you specified in step 4 uses the native webMethods API to create the connection directly on the webMethods Broker, you need to specify the *destinationName* as well as the *destinationType* .

Name	Description
destinationName	Name or lookup name of the Destination to which you want to send the message. ■ Specify the lookup name of the Destination object when the JMS connection alias uses JNDI to retrieve administered objects. ■ Specify the provider-specific name of the Destination when the JMS connection alias uses the native webMethods API to connect directly to the webMethods Broker.
destinationType	Specifies whether the Destination is a queue or a topic. The default is queue.

6. Set values for the header fields in the JMS message.

   All of the header fields are optional.

Name	Description
deliveryMode	Specifies the message delivery mode for the message. Specify one of the following: ■ `PERSISTENT` provides once-and-only-once delivery for the message. The message will not be lost if a JMS provider failure occurs.

Name	Description
	■ `NON_PERSISTENT` provides at-most-once delivery for the message. The message has no guarantee of being saved if a JMS provider failure occurs. The default is `PERSISTENT`.
priority	Specifies the message priority. The JMS standard defines priority levels from 0 to 9, with 0 as the lowest priority and 9 as the highest. The default is 4. Message priority is not supported by Universal Messaging. Any priority assigned to a JMS message sent to Universal Messaging will be ignored.
timeToLive	Specifies the length of time, in milliseconds, that the JMS provider retains the message. The default is 0, meaning that the message does not expire.
JMSType	Message type identifier for the message.

If you created a javax.jms.Message and you set the message header fields using the javax.jms.Message setter methods, you do not need to provide inputs to the fields in *JMSMessage/header* . If you do set message header fields using both approaches, Integration Server uses the values provided as input to the pub.jms:send service.

7. Set values for the Integration Server-specific properties.

   The properties fields are optional fields added to the message header and are often used to hold message selector values. Integration Server adds the following properties to JMS messages it sends. You can set these values as follows.

Name	Description
activation	Specifies the activation ID for the message. A JMS trigger uses the activation ID to join together messages it receives. For more information about setting the activation, see "Assigning an Activation to a JMS Message" on page 95.
uuid	Specifies a universally unique identifier for a message. For more information about setting a UUID, see "Setting the UUID" on page 95.

If you created a javax.jms.Message and you set the message property fields using the javax.jms.Message setter methods, you do not need to provide inputs to the fields in *JMSMessage/properties* . If you do set message property fields using both approaches, Integration Server uses the values provided as input to the pub.jms:send service.

For information about setting the *JMS_WMClusterNodes* property, see "Overriding the Cluster Policy when Sending JMS Messages" on page 125.

8. Add any custom properties to the JMS message.

   To add a new property to *JMSMessage/properties* , click  on the Pipeline view. Select a data type for the property and assign it a name. Make sure to place the new property in the *JMSMessage/properties* field.

   Assign a value to any custom properties that you add.

9. Map data to the body of the JMSMessage document.

   Specifically, map the field that contains the data you want to included in the message body to the field in *JMSMessage/body* with the appropriate data type.

Map to this field...	If you...
string	Used a field of type String for the message body content.
bytes	Used a one-dimensional byte array for the message body content.
object	Used a Serializable Java object for the message body content.
data	Used a Document (IData) for the JMS message body content. Keep in mind that the IData message format can only be used when sending a JMS message from one Integration Server to another.
message	Used a Java service to create an object of type javax.jms.Message. **Note:** If you created a Java service that used one of the com.wm.app.b2b.server.jms.producer.ProducerFacade methods to create a javax.jms.Message object, map the javax.jms.Message object produced by the Java service to *message* .

10. Specify whether the client side queue should be used.

    When the client side queue is in use, Integration Server places messages in the client side queue if the JMS provider is not available at the time the pub.jms:send service

executes. If you want to use the client side queue with this implementation of the pub.jms:send service, the JMS connection alias specified for *connectionAliasName* must be configured to have a client side queue. A JMS connection alias has a client side queue if the **Maximum CSQ Size** property for the alias is set to a value other than 0 (zero).

Name	Description
useCSQ	Indicates whether Integration Server places the sent message in the client side queue if the JMS provider is not available at the time the message is sent. ■ `True` specifies that Integration Server writes messages to the client side queue if the JMS provider is not available at the time this service executes. When the JMS provider becomes available, Integration Server sends messages from the client side queue to the JMS provider. ■ `False` indicates that Integration Server throws an ISRuntimeException if the JMS provider is not available at the time this service executes. The default is `False`. **Note:** If the specified *connectionAliasName* uses a cluster connection factory to which the multisend guaranteed policy is applied, set *useCSQ* to `False`. For more information about the multisend guaranteed policy, see "Working with the Multisend Guaranteed Policy" on page 122.

Sending a JMS Message and Waiting for a Reply

Integration Server provides the pub.jms:sendAndWait service, which you can use to send a message and wait for a reply.

You can use the pub.jms:sendAndWait service to issue a request/reply in a synchronous or asynchronous manner.

- In a synchronous request/reply, the service that sends the request stops executing while it waits for a reply. When the service receives a reply message, the service resumes execution. If the *timeout* elapses before the service receives a reply, Integration Server ends the request, and the service returns a null message that indicates that the request timed out. Integration Server then executes the next step in the flow service.
- In an asynchronous request/reply, the service that sends the request continues executing the steps in the service after sending the message. To retrieve the reply,

the requesting flow service must invoke the pub.jms:waitForReply service. If the *timeout* elapses before the pub.jms:waitForReply service receives a reply, the pub.jms:waitForReply service returns a null document indicating that the request timed out.

A service that contains multiple asynchronous send and wait invocations allows the service to send all the requests before collecting the replies. This approach can be more efficient than sending a request, waiting for a reply, and then sending the next request.

How to Send a Request Message and Wait for a Reply

The following describes the general steps you take to build a service that sends a request message and then waits for a reply.

1. Create an empty flow service.
2. Create the message body.

   For more information about creating content for the body of a JMS message, see step 2 in the section "How to Send a JMS Message" on page 74.
3. Invoke pub.jms:sendAndWait.

   This service creates a JMS message (javax.jms.Message) based on input provided to the service or takes an existing JMS message and sends it to the JMS provider.
4. Specify the JMS connection alias.

   The JMS connection alias indicates how Integration Server connects to the JMS provider.

Name	Description
connectionAliasName	Name of the JMS connection alias that you want to use to send the message.

5. Specify the destination to which you want to send the message.

   If the JMS connection alias you specified in step 4 uses the native webMethods API to create the connection directly on the webMethods Broker, you need to specify the *destinationName* as well as the *destinationType* .

Name	Description
destinationName	Name or lookup name of the Destination to which you want to send the message. ■ Specify the lookup name of the Destination object when the JMS connection alias uses JNDI to retrieve administered objects. ■ Specify the provider-specific name of the Destination when the JMS connection alias uses the native

Name	Description
	webMethods API to connect directly to the webMethods Broker.
destinationType	Specifies whether the Destination is a queue or a topic. The default is queue.

6. Specify the destination to which message recipients should send the reply message. (Optional)

   If you do not specify a destination for reply messages, Integration Server uses a temporaryQueue to receive the reply. A *temporaryQueue* is a queue object created for the duration of a particular connection.

   When using pub.jms:sendAndWait to issue a request/reply, you must specify a queue as the value of the *destinationNameReplyTo* parameter. In a request/reply scenario, it is possible that the message consumer created to receive the reply might be created after the reply message is sent. (In a synchronous request/reply, the pub.jms:sendAndWait service creates the message consumer. In an asynchronous request/reply, the pub.jms:waitForReply service or a custom solution, such as a JMS trigger, creates the message consumer.) If the reply destination is a queue, a message consumer can receive messages published to the queue regardless of whether the message consumer was active at the time the message was published. If the destination is a topic, a message consumer can receive only messages published when the message consumer was active. If the reply is sent to a topic before the message consumer is created, the message consumer will not receive the reply. Consequently, when creating a request/reply, the *destinationNameReplyTo* parameter should specify the name or lookup name of a queue.

Name	Description
destinationNameReplyTo	Name or lookup name of the Destination to which you want the reply message sent. ■ Specify the lookup name of the Destination object when the JMS connection alias uses JNDI to retrieve administered objects. ■ Specify the provider-specific name of the Destination when the JMS connection alias uses the native webMethods API to connect directly to the webMethods Broker.
destinationTypeReplyTo	Specifies whether the Destination is a queue or a topic. The default is queue.

7. If this is a synchronous request/reply, set the request timeout.

The timeout indicates how long Integration Server waits for a reply message. The *timeout* parameter only applies to synchronous send and wait requests.

Name	Description
timeout	Time to wait (in milliseconds) for the response to arrive. You must set this to a value greater than 0 (zero). If no *timeout* value is specified, the service does not wait and returns a null document.

8. Populate the JMS message.

   To populate the JMS message header, properties, and body, follow steps 5–9 in the section "How to Send a JMS Message" on page 74.

9. Indicate whether the request is synchronous or asynchronous.

   The pub.jms:sendAndWait provides a parameter that you can set to indicate whether the request is synchronous or asynchronous. By default, the request is synchronous.

Name	Description
async	Flag specifying whether this is an asynchronous or synchronous request/reply. ■ `True` indicates that this is an asynchronous request/reply. After sending the message, Integration Server executes the next step in the flow service immediately. Integration Server does not wait for a reply before continuing service execution. ■ `False` indicates that this is a synchronous request/reply. After sending the message, Integration Server waits for a reply before executing the next step in the flow service. The default is `False`.

10. Specify whether the client side queue should be used.

    When the client side queuing is in use, Integration Server places messages in the client side queue if the JMS provider is not available at the time the pub.jms:sendAndWait service executes. If you want to use the client side queue with this implementation of the pub.jms:sendAndWait service, the JMS connection alias specified for *connectionAliasName* must be configured to have a client side queue. A JMS connection alias has a client side queue if the **Maximum CSQ Size** property for the alias is set to a value other than 0 (zero).

    The client side queue can be used with asynchronous requests only.

If the client side queue is in use, the reply destination must be a queue that is not temporary. Consequently, if *useCSQ* is set to true, values must be specified for the *destinationNameReplyTo* and *destinationTypeReplyTo* input parameters.

Name	Description
useCSQ	Indicates whether Integration Server places the sent message in the client side queue if the JMS provider is not available at the time the message is sent. ■ `True` specifies that Integration Server writes messages to the client side queue if the JMS provider is not available at the time this service executes. When the JMS provider becomes available, Integration Server sends messages from the client side queue to the JMS provider. ■ `False` indicates that Integration Server throws an ISRuntimeException if the JMS provider is not available at the time this service executes. The default is `False`. **Note:** If the specified *connectionAliasName* uses a cluster connection factory to which the multisend guaranteed policy is applied, set *useCSQ* to `False`. For more information about the multisend guaranteed policy, see "Working with the Multisend Guaranteed Policy" on page 122.

11. Invoke pub.jms:waitForReply.

    If you configured pub.jms:sendAndWait as an asynchronous request/reply, you need to invoke the pub.jms:waitForReply service to retrieve the reply message.

    Specify the following input values for the pub.jms:waitForReply service.

Name	Description
correlationID	A unique identifier used to associate the reply message with the initial request. Integration Server uses the value of the *uuid* or *JMSMessageID* fields in the requesting JMS message to correlate the response to the request. ■ If you set the *uuid* in the JMS message request, you can link the value of the *uuid* field from the *JMSMessage* produced by pub.jms:sendAndWait to the *correlationID* .

Name	Description
	■ If you did not specify a *uuid* , you can link the *JMSMessageID* field from the *JMSMessage* produced by pub.jms:sendAndWait to the *correlationID* .
timeout	Optional. Time to wait (in milliseconds) for the reply to arrive. If *timeout* is greater than 0 (zero) and a reply is not available at the time the pub.jms:waitForReply service executes, the service continues to wait for the document until the time specified in the *timeout* parameter elapses. If the service does not receive a reply by the time the timeout interval elapses, the pub.jms:waitForReply service returns a null document. If *timeout* is set to 0 (zero), the pub.jms:waitForReply service waits indefinitely for a response. Software AG does not recommend setting *timeout* to 0 (zero). If *timeout* is not set, the pub.jms:waitForReply service does not wait for a reply. The pub.jms:waitForReply service always returns a null document. The service returns a null document even if the reply queue contains a response for the request.

Note: The pub.jms:waitForReply service cannot be used to retrieve a response to requests that were routed through the client side queue. To retrieve the response, create a JMS trigger that subscribes to the reply queue.

Note: If the pub.jms:sendAndWait service executes and the message is sent directly to the JMS provider (i.e., it is not sent to the client side queue), the *JMSMessage\header\JMSMessageID* contains a unique identifier assigned by the JMS provider. If the *JMSMessageID* field is null, after the service executes, the JMS provider was not available at the time the service executed. Integration Server wrote the message to the client side queue.

12. Process the reply message.

    The pub.jms:sendAndWait (or pub.jms:waitForReply) service produces the output parameter *JMSReplyMessage* , which contains the JMS message received as a reply.

    If Integration Server does not receive a reply before the specified *timeout* value elapses or if a *timeout* value was not specified, the *JMSReplyMessage* is null. Be sure to code your service to handle this situation.

How to Send a Request that Uses a Dedicated Listener to Retrieve Replies

Integration Server provides the ability to use a dedicated MessageConsumer to retrieve all the replies for a published request. By default Integration Server creates a new JMS MessageConsumer for each reply message. In many cases, this does not impact performance. However, in situations where many threads invoke pub.jms:sendAndWait concurrently, creating a MessageConsumer for every expected response can impact performance.

To send a request that uses a dedicated listener to retrieve the reply, the following must be true:

- The JMS connection alias specified in *connectionAliasName* must be configured to use a dedicated listener to retrieve replies. Specifically:
  - The **Create Temporary Queue** check box must be selected for the alias.
  - The **Enable Request-Reply Listener for Temporary Queue** check box must be selected for the alias.
- The pub.jms:sendAndWait invocation must be synchronous. The *async* input parameter must be set to false.
- The pub.jms:sendAndWait invocation must not specify value for the *destinationNameReplyTo* input parameter.

How to Send a JMS Message and Specify a Reply Destination without Waiting for a Reply

Integration Server provides the ability to send a message, specify a destination for replies, but not wait for or retrieve any reply messages. The act of waiting comes with extra overhead for Integration Server that is unnecessary if you do not want the sending service to wait for a reply.

The pub.jms:send service includes an input parameter for setting the JMSReplyTo header, specifically the *JMSMessage/header/replyTo* field. When executing the pub.jms:send service with a valid value for the *JMSMessage/header/replyTo* parameter, Integration Server creates the javax.jms.Destination and maps it to the JMSReplyTo field within the message header. Integration Server sends the message and returns immediately. The service does not wait for a response message. If *JMSMessage/header/replyTo* parameter is empty, then Integration Server does not set the JMSReplyTo header for the JMS message. If *JMSMessage/header/replyTo* is invalid, then Integration Server throws a ServiceException.

The following procedure describes the general steps you take to build a service that sends a request message, specifies a reply destination, but does not wait for a reply:

1. Follow the steps for sending a JMS message as described in "How to Send a JMS Message" on page 74.
2. When setting the values for the header fields in the JMS message, specify one of the following for the *replyTo* field.
   - If the JMS connection alias used by the pub.jms:send service connects to the JMS provider using JNDI, set *replyTo* to be the lookup name of the destination lookup object name.
   - If the JMS connection alias used by the pub.jms:send service connects to the JMS provider using a native Broker connection, set JMSMessage/header/replyTo to the Broker queue name. That is, if the JMS connection alias specifies the webMethods Broker as the JMS provider and uses the native webMethods API to connect directly to the Broker, specify the name of the queue on the Broker that should receive replies to the message.

     Note: When using the native webMethods API to connect to the Broker, the *JMSMessage/header/replyTo* destination must be a queue. Topics are not supported.

Replying to a JMS Message

You can create a service that sends a reply message in response to a received request message. The reply message might be a simple acknowledgement or might contain information requested by the sender.

When you send a reply message, Integration Server uses information in the request message to determine the reply destination. The *JMSReplyTo* field in the request message is set by the sending client and indicates the destination to which the reply will be sent. The replying Integration Server automatically sets this value when it executes the pub.jms:reply service.

When replying to a message, Integration Server also automatically sets the *JMSCorrelationID* in the reply message. Integration Server, and many JMS clients, use the *JMSCorrelationID* to correlate the reply message with the request message. Integration Server uses the value of the *wm_tag*, *uuid* or *JMSMessageID* fields in the requesting JMS message to correlate the request and the response.Integration Server determines which field to use as the *JMSCorrelationID* using the following order.

- If the request message contains a *wm_tag* value in the *JMSMessage/properties* , Integration Server uses the *wm_tag* value as the *JMSCorrelationID* of the reply message. The *wm_tag* field is used to correlate the reply with a dedicated listener created for a particular JMS connection alias.
- If the sender of the request message specified the *uuid* , the replying Integration Server uses the *uuid* as the *JMSCorrelationID* of the reply message.

- If the sender of the request message did not specify a *uuid* , the replying Integration Server uses the *JMSMessageID* from the request message as the *JMSCorrelationID* of the reply message.

How to Send a Reply Message

The following describes the general steps you take to build a service that sends a reply message.

1. Open or create the service that will send the reply.

   If a JMS trigger received the message, this might be the trigger service or a service invoked by the trigger service. If you used the pub.jms:receive service to retrieve the message from the JMS provider, you might reply to the message within the same service or in another service invoked by the same top-level service.

2. Invoke pub.jms:reply.

   This service takes the reply message you created and delivers it to the destination specified in the *JMSReplyTo* field in the header of the request message.

3. Populate the JMS message.

   If a JMS trigger received the message, populate the JMS message header, properties, and body, follow steps 5–9 in the section "How to Send a JMS Message" on page 74.

4. Specify the consumer and message.

   If you received the message using the pub.jms:receive service, you must specify the message consumer used to receive the message and the request message. You do not need to specify this information if a JMS trigger received the message.

Name	Description
consumer	The message consumer object used to receive the request message from the JMS provider. Integration Server uses information from the *consumer* to create a message producer that will send the reply message.
message	A javax.jms.Message object that contains the request message. You can map the *JMSMessage/body/message* field in the request message to the pub.jms:reply*message* input parameter. The pub.jms:reply service uses the request message to determine the *replyTo* destination.

Receiving a JMS Message Using Built-In Services

At times, you might not want to wait for a JMS trigger to execute to receive a message. Instead, you might want to receive a message from the JMS provider on demand. Receiving a message on demand provides more control over when and how Integration Server receives a message; however, it may not be as efficient or practical as using a JMS trigger to listen for and then receive the message. You can use the pub.jms:receive service to retrieve messages on demand from the JMS provider.

To listen for messages and receive them when they are available, create a JMS trigger that listens to the destination. For more information about creating a JMS trigger, see *webMethods Service Development Help*.

How to Actively Receive a JMS Message

The following describes the general steps you take to build a service that receives a message from the JMS provider.

1. Create a new service.
2. Invoke a pub.jms:createConsumer.

   This service creates a message consumer that receives messages sent to a particular destination.

   Use the following steps to create the message consumer.

   a. Specify the JMS connection alias.

      The JMS connection alias indicates how Integration Server connects to the JMS provider.

Name	Description
connectionAliasName	Name of the JMS connection alias that you want to use to receive the message.

   b. Specify the destination from which you want to receive the message.

      Specify the messages that you want the consumer to receive by selecting a destination and by creating a message selector. A *message selector* is a filter that the JMS provider evaluates. If a message does not meet the criteria specified in the filter, the consumer does not receive the message. Use a message selector to receive a subset of messages from a destination.

Name	Description
destinationName	Name or lookup name of the Destination from which you want to receive the message. ■ Specify the lookup name of the Destination object when the JMS connection alias uses JNDI to retrieve administered objects. ■ Specify the provider-specific name of the Destination when the JMS connection alias uses the native webMethods API to connect directly to the webMethods Broker.
destinationType	Specifies whether the Destination is a queue or a topic. The default is queue. If the JMS connection alias you specified in step 2a uses the native webMethods API to create the connection directly on the webMethods Broker, you need to specify the *destinationName* as well as the *destinationType* .
messageSelector	Optional. Specifies a filter used to receive a subset of messages from the specified destination. The message selector must use the message selector syntax specified in the Java Message Service standard. For more information about message selectors, see *webMethods Service Development Help*.
durableSubscriberName	Optional. Name of the durable subscriber that you want this service to use on the JMS provider. A durable subscriber creates a durable subscription on the JMS provider. If a durable subscriber of this name already exists on the JMS provider, this service resumes the previously established subscription. **Note:** This parameter only applies when the *destinationType* is set to `TOPIC`. If you select `TOPIC`, but do not specify a *durableSubscriberName* , this service creates a nondurable subscriber. If *destinationType* is set to `QUEUE`, this parameter is ignored.

c. Determine the acknowledgment mode.

Acknowledgment mode indicates how Integration Server acknowledges messages received by a message consumer.

Use the following parameter to specify the acknowledgment mode.

Name	Description
acknowledgmentMode	■ `AUTO_ACKNOWLEDGE` Automatically acknowledge the message when it is received by the message consumer. The message consumer will acknowledge the message before the message processing completes. The JMS provider cannot redeliver the message if Integration Server becomes unavailable before message processing completes. ■ `CLIENT_ACKNOWLEDGE` Acknowledge the receipt of a message when the JMS client explicitly acknowledges it. In this case, acknowledge the message when Integration Server invokes the pub.jms:acknowledge service. ■ `DUPS_OK_ACKNOWLEDGE` Automatically, lazily acknowledge the receipt of messages, which reduces system overhead but may result in duplicate messages being sent. The default is `AUTO_ACKNOWLEDGE`.

d. Indicate whether locally published messages are ignored.

If you specified `TOPIC` as the *destinationType* , you can configure a consumer to ignore messages published using the same JMS connection alias used by the consumer.

Integration Server considers a message to be local if it is:

- Sent by the same Integration Server, and
- Sent using the same JMS connection alias.

Name	Description
noLocal	Indicates whether the consumer ignores locally published messages: ■ `True` indicates the consumer will not receive locally published messages.

Name	Description
	■ `False` indicates the consumer can receive locally published messages. The default is `False`.

3. Invoke pub.jms:receive.

   This service uses the consumer created by the pub.jms:createConsumer service to receive messages from the specified destination.

   In the Pipeline view, make sure the *consumer* created by the pub.jms:createConsumer service is linked to the pub.jms:receive service input parameter *consumer* . Designer should link these automatically.

   Specify how long the consumer should wait to receive a message from the JMS provider.

Name	Description
timeout	Specifies the time to wait, in milliseconds, for a message to be received from the JMS provider. If you specify 0 (zero), the consumer will not wait. The default is 0 (zero).

4. Process the received JMS message.

   Invoke a service or a sequence of services to process the message received from the JMS provider. In the Pipeline view, link the *JMSMessage* returned by pub.jms:receive to the input for the service that processes the message.

   If the *timeout* period elapses before a message is received, the value of *JMSMessage* is null. Make sure to code your service to handle this situation.

5. Invoke pub.jms:acknowledge.

   If the acknowledgment mode of the *consumer* that received the message is set to `CLIENT_ACKNOWLEDGE` use the pub.jms:acknowledge service to acknowledge the message to the JMS provider. A message is not considered to be successfully consumed until it is acknowledged.

   Provide the following input parameter.

Name	Description
message	A javax.jms.Message object that identifies the message for which you want Integration Server to send an acknowledgement to the JMS provider.

Name	Description
	You can map the value of the *JMSMessage/body/message* field in the JMS message retrieved by the pub.jms:receive service to this field.

If you use the *consumer* created by the pub.jms:createConsumer service to receive multiple messages, keep in mind that acknowledging a message automatically acknowledges the receipt of all messages received in the same session. That is, all messages received by the same *consumer* will be acknowledged when just one of the received messages is acknowledged. Therefore, if the *consumer* receives multiple messages, invoke the pub.jms:acknowledge service after processing all of the received messages.

Any message consumers created during the execution of a service will be closed automatically when the service completes. If the *consumer* closes without acknowledging messages, messages are implicitly recovered back to the JMS provider.

Sending a JMS Message as Part of a Transaction

A transaction is a logical unit of work, composed of many different processes and involving one or more resources, that either entirely succeeds or has no effect at all. Transactions can either be implicit or explicit.

In an *implicit transaction*, the transaction manager in Integration Server automatically manages the transactions without requiring any additional services or input. In an *explicit transaction*, you control the transactional units of work by defining the start and completion boundaries of the transaction. The WmART package on Integration Server provides built-in services that you can use to start and complete transactions.

In some situations, you might use the built-in service pub.art.transaction:startTransaction to start a transaction explicitly, but then allow Integration Server to commit or roll back the transaction implicitly based on the success or failure of the service.

For more information about transactions see "Transaction Management" on page 133.

You can create a service that sends or receives JMS messages within an explicit transaction. The service must do the following:

- Use pub.art.transaction:startTransaction to start the transaction.
- Create a connection to the JMS provider using a JMS connection alias with a transaction type of LOCAL_TRANSACTION or XA_TRANSACTION, depending on the kind of transaction.
- Use pub.art.transaction:commitTransaction to commit the transaction.
- Use pub.art.transaction:rollbackTransaction to roll back the transaction.

Keep the following points in mind when building services that send or receive JMS messages within a transaction:

- To send or receive JMS messages within a transaction, you must install and enable the WmART package. (This is true even if you intend to use Integration Server to manage all transactions implicitly.)
- To use pub.jms:send or pub.jms:sendAndWait within a transaction, the client side queue cannot be used (the *useCSQ* parameter must be set to false).
- To use pub.jms:sendAndWait within a transaction, the request/reply must be asynchronous (the *async* parameter must be set to true). If *async* is set to false, Integration Server throws a JMSSubsystemException when the service executes.
- If you do not specifically invoke pub.art.transaction:commitTransaction or pub.art.transaction:rollbackTransaction, Integration Server implicitly commits the transaction when the services within the transaction are successful. Integration Server implicitly rolls back the transaction when one of the services within the transaction fails with any type of exception.

How to Send a JMS Message within a Transaction

The following describes the general steps you take to send a JMS message to a JMS provider as part of a transaction (XA or Local).

1. Create an empty flow service.
2. Create the message body.

   For more information about creating content for the body of a JMS message, see step 2 in the section "How to Send a JMS Message" on page 74.

3. Invoke pub.art.transaction:startTransaction.

   This service starts an explicit transaction. This service is located in the WmART package.

   In the *startTransactionInput* document list, you can provide the following optional parameter.

Name	Description
transactionName	A String that specifies the name of the transaction to be started. If this field is blank, Integration Server will generate a name for you.

   If you do not use pub.art.transaction:startTransaction to start an explicit transaction, Integration Server starts an implicit transaction when it executes a pub.jms:send service that specifies a transacted JMS connection alias.

4. Invoke pub.jms:send.

This service takes the JMS message you created and sends it to the JMS provider.

5. Specify the JMS connection alias.

   The JMS connection alias indicates how Integration Server connects to the JMS provider.

Name	Description
connectionAliasName	Name of the JMS connection alias that you want to use to send the message. The specified JMS connection alias *must* have a transaction type of LOCAL_TRANSACTION or XA_TRANSACTION, depending on the kind of transaction.

6. Finish supplying inputs to the pub.jms:send service.

   Follow steps 5-9 under "How to Send a JMS Message" on page 74.

7. Add any additional services to the transaction.

   For example, you might want to invoke another built-in JMS service or an adapter service.

8. Insert logic to commit and/or rollback the transaction explicitly.

   You may build your service to commit the transaction if all services execute successfully and to rollback the transaction if all services do not execute successfully.

   - Invoke pub.art.transaction:commitTransaction to commit the transaction and send the JMS message. In the Pipeline view, map the contents of *startTransactionOutput/transactionName* to *commitTransactionInput/transactionName* .
   - Invoke pub.art.transaction:rollbackTransaction to roll back the transaction. The message will not be sent to the JMS provider. In the Pipeline view, map the contents of *startTransactionOutput/transactionName* to *rollbackTransactionInput/transactionName* .

   If you do not specifically invoke pub.art.transaction:commitTransaction or pub.art.transaction:rollbackTransaction, Integration Server implicitly commits the transaction when the services within the transaction are successful. Integration Server implicitly rolls back the transaction when one of the services within the transaction fails with any type of exception.

Setting Properties in a JMS Message

Properties are an optional part of a JMS message that enable you to add fields to the message header. Properties usually hold message selector values and are application-specific, standard, or provider-specific.

When building a service that sends a JMS message, you can:

- Add your own custom properties to the *JMSMessage/properties* document using the pipeline in Designer.
- Assign values to the application-specific properties *activation* and *uuid* included by Integration Server.

The following sections provide information about setting the *activation* and *uuid* . For information about adding fields to the pipeline in Designer, see *webMethods Service Development Help*.

Assigning an Activation to a JMS Message

An *activation* is a unique identifier assigned to a message that will be processed by a JMS trigger that contains a join. A join specifies that a JMS trigger handles messages received from two or more destinations as a single unit and with a single routing rule. The JMS trigger needs to receive messages from all, only one, or any of the destinations before it executes the associated routing rule.

Because a JMS trigger can receive multiple messages from the destinations, Integration Server uses the *activation* value to identify the set of messages processed by an instance of a join.

- For an All (AND) join, Integration Server waits until it receives messages with the same *activation* from each destination before executing the routing rule.
- For an Only one (XOR) join, Integration Server executes the routing rule after it receives a message from any destination in the join; however, the JMS trigger discards messages with the same *activation* received from the other destinations for the duration of the join time-out.
- For an Any (OR) join, Integration Server executes the routing rule when it receives messages from any destination in the join. Integration Server does not use the *activation* value when processing JMS triggers with an Any (OR) join.

When the JMS trigger receives messages with a different activation from one the destinations, Integration Server treats it as another instance of the join.

Integration Server stores the activation in the *activation* field of a JMS message, specifically, *JMSMessage/properties/activation* . The *activation* field is of type String.You assign an activation to a message manually. Integration Server does not assign an activation automatically.

Setting the UUID

The *UUID* is a universally unique identifier for a message. Integration Server uses the UUID to provide duplicate detection for exactly-once processing. Integration Server stores the UUID in the *JMSMessage/properties/uuid* field.

You might want to assign a UUID in the following situation:

- The JMS message originated in a back-end system that assigned a unique identifier to the message data. You can map the value assigned by the system to the *JMSMessage/properties/uuid* field. A JMS trigger that receives the message can use the assigned UUID to filter out duplicate messages from a back-end system.
- A JMS message is part of a request/reply. If you specify the *uuid* when sending the request, the replying Integration Server will use the *uuid* as the *JMSCorrelationID* of the reply message. If you do not specify a *uuid*, the replying Integration Server uses the *JMSMessageID* of the request message as the *JMSCorrelationID* of the reply message.

The maximum length of a UUID is 96 characters. Integration Server does not assign a UUID automatically.

4 Exactly-Once Processing for JMS Triggers

Overview of Exactly-Once Processing for JMS Triggers

Within Integration Server, exactly-once processing is a facility that ensures one-time processing of a persistent message by a JMS trigger. The trigger does not process duplicates of the message. Integration Server provides exactly-once processing when all of the following are true:

- The message is persistent.
- The JMS trigger has an acknowledgement mode set to `CLIENT_ACKNOWLEDGE`.
- Exactly-once properties are configured for the JMS trigger.

Note: Exactly-once processing typically only provides value for JMS triggers that receive messages from queues or JMS triggers that receive messages from topics using durable subscribers.

Duplicate Detection Methods for JMS Triggers

Integration Server ensures exactly-once processing by performing duplicate detection and by providing the ability to retry trigger services. *Duplicate detection* determines whether the current message is a copy of one previously processed by the trigger.

Duplicate messages can be introduced into the webMethods system in the following situations:

- The sending client sends the same message more than once.
- When receiving persistent messages from the JMS provider, Integration Server and the JMS provider lose connectivity before the JMS trigger processes and acknowledges the message. The JMS trigger will receive the message again when the connection is restored.

Integration Server uses duplicate detection to determine the message's status. The message status can be one of the following:

- **New.** The message is new and has not been processed by the trigger.
- **Duplicate.** The message is a copy of one already processed the trigger.
- **In Doubt.**Integration Server cannot determine the status of the message. The trigger may or may not have processed the message before.

To resolve the message status, Integration Server evaluates, in order, one or more of the following:

- **Delivery count** indicates how many times the JMS provider has delivered the message to the JMS trigger.

- **Document history database** maintains a record of all persistent message IDs processed by JMS triggers that have an acknowledgment mode of `CLIENT_ACKNOWLEDGE` and for which exactly-once processing is configured.
- **Document resolver service** is a service created by a user to determine the message status. The document resolver service can be used instead of or in addition to the document history database.

The steps that Integration Server takes to determine a message's status depend on the exactly-once properties configured for the JMS trigger. For more information about configuring exactly-once properties, see *webMethods Service Development Help*.

Summary of Duplicate Detection Process for JMS Triggers

The following table summarizes the process Integration Server follows to determine a message's status and the action the server takes for each duplicate detection method.

Step 1 Check Delivery Count

When a JMS trigger is configured to detect duplicates, Integration Server checks the message's delivery count to determine if the JMS trigger processed the message previously. Specifically, Integration Server checks the value of the *JMSXDeliveryCount* property in the message.

Delivery Count	Action
1	If using document history, Integration Server proceeds to Step 2 to check the document history database. If document history is not used, Integration Server considers the message to be New. Integration Server executes the trigger service.
>1	If using document history, Integration Server proceeds to Step 2 to check the document history database. If document history is not used, Integration Server proceeds to Step 3 to execute the document resolver service. If neither document history nor a document resolver service are used, Integration Server considers the message to be In Doubt.

-1 (Undefined)	If using document history, Integration Server proceeds to Step 2 to check the document history database. If document history is not used, Integration Server proceeds to Step 3 to execute the document resolver service. Otherwise, Integration Server considers the message to be New. Integration Server executes the trigger service.

Step 2 Check Document History

If a document history database is configured and the trigger uses it to maintain a record of processed messages, Integration Server checks for the message's UUID in the document history database. If the message does not have a UUID, Integration Server uses the value of the *JMSMessageID* field from the message header.

UUID or JMSMessageID Exists?	Action
No	Message is New. Integration Server executes the trigger service.
Yes	Integration Server checks the processing status of the entry. ■ If the processing status indicates that message processing completed, then Integration Server considers the message to be a Duplicate. Integration Server acknowledges the message but does not execute the trigger service. ■ If the processing status indicates that processing started but did not complete, Integration Server considers the message to be In Doubt. If provided, Integration Server proceeds to Step 3 to invoke the document resolver service. Otherwise, Integration Server considers the message to be In Doubt. Integration Server acknowledges the message but does not execute the trigger service.

Step 3 Execute Document Resolver Service

If a document resolver service is specified, Integration Server executes the document resolver service assigned to the trigger.

Returned Status	Action
NEW	Integration Server executes the trigger service and acknowledges the message.
DUPLICATE	Integration Server acknowledges the message but does not execute the trigger service.
IN_DOUBT	Integration Server acknowledges the message but does not execute the trigger service.

Notes:

- When a transacted JMS trigger fails because of a transient error and the document history database is configured and enabled, Integration Server considers the message to be NEW the next time it is received.
- When a transacted JMS trigger fails because of a fatal error and the document history database is configured and enabled, Integration Server rejects the message the next time it is received and generates a JMS retrieval failure event.
- When a transacted JMS trigger fails because of a fatal or transient error and the document history database is neither configured nor enabled, Integration Server does not send a JMS retrieval failure event the next time the message is received. Integration Server sends a JMS retrieval failure event if the maximum delivery count is eventually reached.

Delivery Count for JMS Messages

The delivery count indicates the number of times the JMS provider has delivered or attempted to deliver a message to the JMS trigger. Most JMS providers track the message delivery count in the JMS-defined property *JMSXDeliveryCount* . The initial delivery is 1, the second delivery is 2, and so on. A delivery count other than 1 indicates that the trigger might have received and processed (or partially processed) the message before.

For example, when Integration Server first retrieves a message for a JMS trigger, the *JMSXDeliveryCount* count is 1 (one). If a resource (JMS provider or Integration Server) shuts down before the trigger processes and acknowledges the message, Integration Server retrieves the message again when the connection is re-established. The second time Integration Server retrieves the message, the *JMSXDeliveryCount* is 2. A delivery count greater than 1 indicates that the JMS provider may have delivered the message to the trigger before.

The following table identifies the possible delivery count values and the message status associated with each value.

A delivery count of...	Indicates...
-1	The JMS provider that delivered the message does not maintain a *JMSXDeliveryCount* or an error occurred when retrieving the *JMSXDeliveryCount* . As a result, the delivery count is undefined. Integration Server uses a value of -1 to indicate that the delivery count is absent. If other methods of duplicate detection are configured for this trigger (document history database or document resolver service), Integration Server uses these methods to determine the message status. If no other methods of duplicate detection are configured, Integration Server assigns the message a status of New and executes the trigger service.
1	This is the first time the JMS trigger received the message. If the JMS trigger uses a document history to perform duplicate detection, Integration Server checks the document history database to determine the message status. If no other methods of duplicate detection are configured, Integration Server assigns the message a status of New and executes the trigger service.
>1	The JMS provider has delivered the message more than once. The trigger might or might not have processed the message before. The delivery count does not provide enough information to determine whether the trigger processed the message before. If other methods of duplicate detection are configured for this trigger (document history database or document resolver service), Integration Server uses these methods to determine the message status. If no other methods of duplicate detection are configured, Integration Server assigns the message a status of In Doubt and acknowledges the message.

Integration Server uses delivery count to determine message status whenever you enable exactly-once processing for a JMS trigger. That is, setting the **Detect duplicates** property to true indicates delivery count will be used as part of duplicate detection.

Note: webMethods messaging triggers use a redelivery count instead of a delivery count for exactly-once processing.

Document History Database for Use with JMS Triggers

The document history database maintains a history of the persistent messages processed by JMS triggers. Integration Server adds an entry to the document history database when a trigger service begins executing and when it executes to completion (whether it ends in success or failure). The document history database contains message processing information only for triggers for which the **Use history** property is set to true.

The database saves the following information about each message:

- **Trigger ID.** Universally unique identifier for the JMS trigger processing the message.
- **UUID or JMSMessageID.** Universally unique identifier for the message. If the sending client assigned a value to the *uuid* field in the message, Integration Server uses the *uuid* value to identify the message. If the *uuid* field is empty, Integration Server uses the value of the *JMSMessageID* field in the message header to identify the message.
- **Processing Status.** Indicates whether the trigger service executed to completion or is still processing the message. An entry in the document history database has either a status of "processing" or a status of "completed." Integration Server adds an entry with a "processing" status immediately before executing the trigger service. When the trigger service executes to completion, Integration Server adds an entry with a status of "completed" to the document history database.
- **Time.** The time the trigger service began executing. The document history database uses the same time stamp for both entries it makes for a message. This allows Integration Server to remove both entries for a specific message at the same time.

To determine whether a message is a duplicate of one already processed by the JMS trigger, Integration Server checks for the message's UUID (or *JMSMessageID*) in the document history database. The existence or absence of the message's UUID (JMSMessageID) can indicate whether the message is new or a duplicate.

If the UUID or JMSMessageID...	Then Integration Server...
Does not exist	Assigns the message a status of New and executes the trigger service. The absence of the UUID (*JMSMessageID*) indicates that the trigger has not processed the message before.
Exists in a "processing" entry and a "completed" entry	Assigns the message a status of Duplicate. The existence of the "processing" and "completed" entries for the message's UUID (*JMSMessageID*) indicate the trigger processed the message successfully already. Integration Server acknowledges the message, discards it, and writes a journal log entry indicating that a duplicate message was received.

If the UUID or JMSMessageID...	Then Integration Server...
Existing in a "processing" entry only	Cannot determine the status of the message conclusively. The absence of an entry with a "completed" status for the UUID (*JMSMessageID*) indicates that the trigger service started to process the message but did not finish. The trigger service might still be executing or the server might have unexpectedly shut down during service execution. If a document resolver service is specified for the JMS trigger, Integration Server invokes it. If a document resolver service is not specified, Integration Server assigns the message a status of In Doubt, acknowledges the message, and writes a journal log entry stating that an In Doubt message was received.
Exists in a "completed" entry only	Determines the message is a Duplicate. The existence of the "completed" entry indicates the JMS trigger processed the message successfully already. Integration Server acknowledges the message, discards it, and writes a journal log entry indicating that a Duplicate message was received.

Note: Integration Server also considers a message to be In Doubt when the value of the message's UUID (or *JMSMessageID*) exceeds 96 characters. If specified, Integration Server executes the document resolver service to determine the message's status. Otherwise, the Integration Server logs the message as In Doubt.

If you want to use document history to ensure exactly-once processing for some or all of your JMS triggers, you or the server administrator must create the Document History database component and connect it to a JDBC connection pool. For information about configuring the document history database, refer to *Installing Software AG Products*.

What Happens when the Document History Database Is Not Available for a JMS Trigger?

If the connection to the document history database is unavailable when Integration Server attempts to query the database, Integration Server considers the lack of availability to be a transient error in the preprocessing phase of trigger execution. How Integration Server responds to the transient error depends on the configured transient error handling for trigger preprocessing.

For more information about transient error handling during trigger preprocessing, see "Transient Error Handling During Trigger Preprocessing" on page 111.

Managing the Size of the Document History Database

To keep the size of the document history database manageable, Integration Server periodically removes expired rows from the database. The length of time the document history database maintains information about a UUID varies for each trigger (JMS trigger or webMethods messaging trigger) and depends on the value of the **History time to live** property for the trigger.

Integration Server provides a scheduled service, namely the Message History Sweeper, that removes expired entries from the database. By default, the Message History Sweeper task executes every 10 minutes. You can change the frequency with which the task executes. For information about editing scheduled services, see *webMethods Integration Server Administrator's Guide*.

Note: The watt.server.idr.reaperInterval property determines the initial execution frequency for the Message History Sweeper task. After you define a JDBC connection pool for Integration Server to use to communicate with the document history database, change the execution interval by editing the scheduled service.

You can also use Integration Server Administrator to clear expired document history entries from the database immediately.

Clearing Expired Entries from the Document History Database

To clear expired entries from the document history database

1. Open Integration Server Administrator.
2. From the **Settings** menu in the Navigation panel, click **Resources**.
3. Click **Exactly Once Statistics**.
4. Click **Remove Expired Document History Entries**.

Document Resolver Service for a JMS Trigger

The document resolver service is a service that you build to determine whether a message's status is New, Duplicate, or In Doubt. Integration Server passes the document resolver service some basic information that the service will use to determine message status, such as the trigger name and the JMS message. The document resolver service must return one of the following for the message status: NEW, DUPLICATE, or IN_DOUBT.

By using the delivery count and the document history database, Integration Server can assign most messages a status of New or Duplicate. However, a small window of time

exists where checking the delivery count and the message history database might not conclusively determine whether a trigger processed a message before. For example:

- If a duplicate message arrives before the trigger finishes processing the original message, the document history database does not yet contain an entry that indicates processing completed. Integration Server assigns the second message a status of In Doubt. Typically, this is only an issue for long-running trigger services.
- If Integration Server fails before completing message processing, the JMS provider redelivers the message. However, the document history database contains only an entry that indicates message processing started. Integration Server assigns the redelivered message a status of In Doubt.

You can write a document resolver service to determine the status of messages received during these windows. How the document resolver service determines the message status is up to the developer of the service. Ideally, the writer of the document resolver service understands the semantics of all the applications involved and can use the message to determine the message status conclusively. If processing an earlier copy of the message left some application resources in an indeterminate state, the document resolver service can also issue compensating transactions.

If provided, the document resolver service is the final method of duplicate detection.

Document Resolver Service and Exceptions for a JMS Trigger

At run time, a document resolver service might end because of an exception. How Integration Server proceeds depends on the type of exception and how the JMS trigger is configured to handle transient errors.

- If the document resolver service ends with an ISRuntimeException, and transient error handling for the JMS trigger is configured to **Suspend and retry later** (non-transacted JMS trigger) or **Suspend and recover** (transacted JMS trigger), Integration Server suspends the trigger and schedules a system task to execute the trigger's resource monitoring service (if one is specified). Integration Server enables the trigger and retries trigger execution when the resource monitoring service indicates that the resources used by the trigger are available.

  If a resource monitoring service is not specified, you will need to resume the trigger manually (via the Integration Server Administrator or the pub.trigger:enableJMSTriggers service). For more information about configuring a resource monitoring service, see "Building a Resource Monitoring Service" on page 129.
- If the document resolver service ends with an ISRuntimeException, and transient error handling for the JMS trigger is configured to **Throw exception** (non-transacted JMS trigger) or **Recover only** (transacted JMS trigger), Integration Server assigns the document a status of In Doubt, acknowledges the document, and generates a JMS retrieval failure event.
- If the document resolver service ends with an exception other than an ISRuntimeException, Integration Server assigns the message a status of In Doubt, acknowledges the message, and generates a JMS retrieval failure event.

Note: The watt.server.jms.trigger.raiseEventOnRetryFailure property must be set to true (the default) for Integration Server to generate JMS retrieval failure events.

Extenuating Circumstances for Exactly-Once Processing

Although Integration Server provides robust duplicate detection capabilities, activity outside of the scope or control of the Integration Server retrieving the message might cause a trigger to process a message (document) more than once. Alternatively, situations can occur where Integration Server might determine a message is a duplicate when it is actually a new message.

Circumstances in which Duplicate Messages Can Be Processed

In the following situations, a trigger with exactly-once processing configured might process a duplicate message (document).

- If the client sends a message twice and assigns a different UUID each time, Integration Server does not detect the second message as a duplicate. Because the messages have different UUIDs, Integration Server processes both messages.
- If the document resolver service incorrectly determines the status of a message to be new (when it is, in fact, a duplicate), Integration Server processes the message a second time.
- If a client sends a message twice, and the second message is sent after Integration Server removes the expired message UUID entries from the document history table, Integration Server determines the second message is new and processes it. Because the second message arrives after the first message's entries have been removed from the document history database, Integration Server does not detect the second message as a duplicate.
- If the time drift between the computers hosting a cluster of Integration Servers is greater than the duplicate detection window for the trigger, one of the Integration Servers in the cluster might process a duplicate message. (The size of the duplicate detection window is determined by the **History time to live** property under **Exactly Once**.)

  For example, suppose the duplicate detection window is 15 minutes and that the clock on the computer hosting one Integration Server in the cluster is 20 minutes ahead of the clocks on the computers hosting the other Integration Servers. A trigger on one of the Integration Servers with a slower clock processes a message successfully at 10:00 GMT.

  The Integration Server adds two entries to the document history database. Both entries use the same time stamp and both entries expire at 10:15 GMT. However, the Integration Server with the faster clock is 20 minutes ahead of the others and

might reap the entries from the document history database before one of the other Integration Servers in the cluster does.

If the Integration Server with the faster clock removes the entries before 15 minutes have elapsed and a duplicate of the message arrives, the Integration Servers in the cluster will treat the message as a new message.

Note: Time drift occurs when the computers that host the clustered servers gradually develop different date/time values. Even if the administrator synchronizes the computer date/time when configuring the cluster, the time maintained by each computer can gradually differ as time passes. To alleviate time drift, synchronize the cluster node times regularly.

Circumstances in which New Messages Are Treated as Duplicates

In some circumstances Integration Server might not process a new, unique message (document) because duplicate detection determines the message is duplicate. For example:

- If the sending client assigns two different messages the same UUID, the Integration Server detects the second message as a duplicate and does not process it.
- If the document resolver service incorrectly determines the status of a message to be duplicate (when it is, in fact, new), Integration Server discards the message without processing it.

Important: In the previous examples, Integration Server functions correctly when determining the message status. However, factors outside of the control of Integration Server create situations in which duplicate messages are processed or new messages are marked as duplicates. The designers and developers of the solution must make sure that clients properly send messages, exactly-once properties are optimally configured, and that document resolver services correctly determine a message's status.

Exactly-Once Processing and Performance

Exactly-once processing for a trigger consumes server resources and can introduce latency into message processing by triggers. For example, when Integration Server maintains a history of persistent messages processed by a JMS trigger (or guaranteed documents for a webMethods messaging trigger), each trigger service execution causes two inserts into the document history database. This requires Integration Server to obtain a connection from the JDBC pool, traverse the network to access the database, and then insert entries into the database.

Additionally, when the delivery count cannot conclusively determine the status of a message or document, Integration Server must obtain a database connection from the

JDBC pool, traverse the network, and query the database to determine whether the trigger processed the message.

If querying the document history database is inconclusive or if the server does not maintain a document history for the trigger, invocation of the document resolver service will also consume resources, including a server thread and memory.

The more duplicate detection methods that are configured for a trigger, the higher the quality of service. However, each duplicate detection method can lead to a decrease in performance.

If a trigger does not need exactly-once processing (for example, the trigger service simply requests or retrieves data), consider leaving exactly-once processing disabled for the trigger. However, if you want to ensure exactly-once processing, you must use a document history database or implement a custom solution using the document resolver service.

5 Transient Error Handling During Trigger Preprocessing

Trigger preprocessing encompasses the time from when a trigger first receives a message (document) from its local queue on Integration Server to the time Integration Server invokes the trigger service. Transient errors can occur during this time. A transient error is an error that arises from a temporary condition that might be resolved or corrected quickly, such as the unavailability of a resource due to network issues or failure to connect to a database. For example, if a document history database is used for exactly-once processing, the unavailability of the database may cause a transient error. Because the condition that caused the trigger preprocessing to fail is temporary, the trigger preprocessing might complete successfully if Integration Server waits and then re-attempts trigger preprocessing. To allow the preprocessing to complete successfully, Integration Server provides some properties and settings for transient error handling.

Server and Trigger Properties that Affect Transient Error Handling During Trigger Preprocessing

Integration Server and Designer provide properties that you can use to configure how Integration Server handles transient errors that occur during the preprocessing phase of trigger execution.

- The watt.server.trigger.preprocess.suspendAndRetryOnError server configuration property. This property determines if Integration Server suspends a trigger if an error occurs during trigger preprocessing. This server configuration parameter acts as a global on/off switch. When set to true, Integration Server suspends any trigger that experiences an error during preprocessing. When set to false, Integration Server uses the individual trigger properties to determine whether or not to suspend the trigger.
- The watt.server.trigger.preprocess.monitorDatabaseOnConnectionException server configuration property. This property determines how Integration Server handles a ConnectionException that causes a transient error. A ConnectionException occurs when the document history database is not enabled or is configured incorrectly.
- The **On Retry Failure** trigger property for webMethods messaging triggers and non-transacted JMS triggers. When set to **Suspend and retry later**, Integration Server suspends a trigger that encounters a transient error during trigger preprocessing.

  Note: The **On Retry Failure** trigger property also determines how Integration Server handles retry failure for a trigger service.

- The **On Transaction Rollback** property for a transacted JMS trigger. When set to Suspend and recover, Integration Server suspends a transected JMS trigger that encounters a transient error during trigger preprocessing.

  Note: The **On Transaction Rollback** property also determines how Integration Server handles a transaction rollback caused by a transient error that occurs during trigger execution.

For a detailed explanation about how Integration Server uses these property settings when a transient error occurs during trigger preprocessing, see "Overview of Transient Error Handling During Trigger Preprocessing" on page 113.

Overview of Transient Error Handling During Trigger Preprocessing

Following is an overview of how Integration Server performs transient error handling for an ISRuntimeException that occurs during trigger preprocessing. Typically, transient errors that occur during preprocessing occur during exactly-once processing. For example, the document history database might not be available of the document resolver service fails because of an ISRuntimeException.

Step	Description
1	A transient error, specifically an ISRuntimeException, occurs during the preprocessing phase of trigger execution.
2	Integration Server checks the values of watt.server.trigger.preprocess.suspendAndRetryOnError server configuration property and the **On Retry Failure** trigger property. If this is a transacted JMS trigger, Integration Serverchecks the value of the **On Transaction Rollback** property instead of the **On Retry Failure** property. If one of the following is true, Integration Server suspends the trigger, rolls the message back to the messaging provider, and proceeds as described in step **3**: ■ watt.server.trigger.preprocess.suspendAndRetryOnError is set to true. ■ **On Retry Failure** property is set to **Suspend and retry later** or **On Transaction Rollback** property is set to **Suspend and recover**. If none of the above are true, then Integration Server does not suspend the trigger if a transient error occurs during trigger preprocessing. Instead, Integration Server does one of the following: ■ If the trigger specifies a document resolver service, Integration Server executes the document resolver service to determine the status of the document. If the document resolver service ends because of an ISRuntimeException, Integration Server assigns the document a status of In Doubt, acknowledges the document, and uses the audit subsystem to log the document. ■ If the trigger does not specify a document resolver service, Integration Server assigns the document a status of In Doubt. Integration Server throws an exception, acknowledges the document to the messaging

Step	Description
	provider, and uses the audit subsystem to log the document. This may result in message loss. **Note:** If the trigger is a webMethods messaging trigger, Integration Server uses the audit subsystem to log the document. You can use webMethods Monitor to resubmit the document.
3	Integration Server does one of the following once the trigger is suspended: ■ If the transient error (ISRuntimeException) is caused by a SQLException (which indicates that an error occurred while reading to or writing from the database), Integration Server suspends the trigger and schedules a system task that executes an internal service that monitors the connection to the document history database. Integration Server resumes the trigger and re-executes it when the internal service indicates that the connection to the document history database is available. ■ If the transient error (ISRuntimeException) is caused by a ConnectionException (which indicates that document history database is not enabled or is not properly configured), and the watt.server.trigger.preprocess.monitorDatabaseOnConnectionException property is set to true, Integration Server schedules a system task that executes an internal service that monitors the connection to the document history database. Integration Server resumes the trigger and re-executes it when the internal service indicates that the connection to the document history database is available. ■ If the transient error (ISRuntimeException) is caused by a ConnectionException and the watt.server.trigger.preprocess.monitorDatabaseOnConnectionException property is set to false, Integration Server does not schedule a system task to check for the database's availability and will not resume the trigger automatically. You must manually resume the trigger after configuring the document history database properly. ■ If the transient error (ISRuntimeException) is caused by some other type of exception, Integration Server suspends the trigger and schedules a system task to execute the trigger's resource monitoring service (if one is specified). When the resource monitoring service indicates that the resources used by the trigger are available, Integration Server resumes the trigger and again receives the message from the messaging provider. If a resource monitoring service is not specified, you will need to resume the trigger manually (via Integration Server Administrator or the pub.trigger* services).

6 Consuming JMS Messages Concurrently in a Load-Balanced Fashion

Introduction

You may want to your JMS triggers to consume messages from a destination in a load-balanced fashion. To load balance message consumption, you can use multiple consumers on one or more Integration Servers to retrieve and process messages concurrently.

Within Integration Server, the ability to receive messages from a destination in a load-balanced fashion is important in two situations:

- Concurrent JMS triggers

  When a concurrent JMS trigger receives messages from the JMS provider, it creates multiple consumers. Each consumer receives a message from the JMS provider, processes the message, and acknowledges the message to the JMS provider. Each consumer needs to consume a message from the same destination but not process any duplicate messages.

- A cluster of Integration Servers

  The same JMS trigger, running on multiple Integration Servers, needs to consume messages from the same destination without processing any duplicate messages.

  Note: Load balancing is necessary for concurrent JMS triggers regardless of whether or not they are running in a cluster of Integration Servers.

The Java Message Service standard does not supply semantics for consuming messages from a destination in a load-balanced fashion. However, it does state that a client can have multiple sessions in which each session is an independent consumer and producer of messages. Regarding the type of destination used by each messaging style, the Java Message Service standard makes the following provisions:

- **Queues (point-to-point messaging).** While the Java Message Service standard does not supply the semantics for multiple consumers receiving messages concurrently, it does not prohibit a JMS provider from supporting it. Most JMS providers support load balancing of messages from a queue across multiple consumers. However, review your JMS provider's documentation to determine how to consume messages from a queue concurrently. For information about using the webMethods Broker to consume messages from a queue concurrently, see "Consuming JMS Messages Concurrently from the webMethods Broker " on page 117.

- **Topics (publish-subscribe messaging).** The Java Message Service standard specifies that each subscriber to the same topic receives each message. The standard does not provide semantics regarding how to concurrently consume messages published to a topic in a load-balanced fashion. Some JMS providers work around this limitation by offering a proprietary extension to the JMS API. Review your JMS provider's documentation to determine how to consume messages from a topic concurrently. For information about how to configure JMS triggers and the webMethods Broker

to consume messages from a topic in a load-balanced fashion, see "Consuming JMS Messages Concurrently from the webMethods Broker " on page 117.

Consuming JMS Messages Concurrently from the webMethods Broker

The webMethods Broker supports the following load-balancing behavior for JMS destinations.

- **Queues.** Multiple clients can connect to and receive messages from the same queue if the queue is configured to share state and all the clients use the same client ID.
- **Topics.** Multiple clients can consume messages in a load-balanced fashion if the clients are connecting to a durable subscriber, state sharing is enabled for the durable subscriber, and all the clients use the same client ID.

> **Note:** Non-durable subscribers (i.e., JMS triggers that subscribe to topics but do not specify a durable subscriber) cannot receive messages in a load-balanced fashion. A JMS trigger using a non-durable subscriber will process duplicates. Therefore, make sure to set **Max execution threads** to 1 when setting message processing properties for a JMS trigger that specifies a non-durable subscriber. This behavior may vary with other JMS providers. For more information about configuring message processing, see *webMethods Service Development Help*.

Configuring JMS Triggers, Integration Server, and webMethods Broker for Load-Balancing

To perform load-balancing while consuming messages concurrently from destinations on the webMethods Broker, the following must be true:

- The JMS trigger must receive messages from a topic using a durable subscriber or from a queue.
- The JMS trigger must specify a JMS connection alias that configures a connection to the webMethods Broker.
- The JMS trigger must process messages concurrently.
- The JMS trigger must be configured identically on all of the Integration Servers across which you are load-balancing message consumption.
- The queue or durable subscriber must be configured to share state. Sharing client state allows multiple clients, each using its own session, to process messages from a single destination in parallel, on a first-come, first-serve basis. To configure state sharing for a queue or durable subscriber, use the Broker user interface in My webMethods. You can also configure state sharing as part of creating the destination

or durable subscriber in Designer. For more information about configuring queues and durable subscribers, see *Administering webMethods Broker*.

If the JMS trigger specifies a JMS connection alias that is configured to manage destinations on the webMethods Broker, Integration Server and Designer can configure state sharing for the durable subscriber automatically.

- The JMS connection alias must be configured identically on all of the Integration Servers across which you are load-balancing message consumption.

Automatic Load Balancing Configuration for Durable Subscribers when Using the webMethods Broker

When the JMS connection alias specified by a JMS trigger connects to the webMethods Broker and is configured to manage destinations, Integration Server can automatically configure load balancing for a JMS trigger that specifies a durable subscriber.

If the durable subscriber specified by the JMS trigger does not exist, when you save a JMS trigger, Integration Server creates the durable subscriber at the Broker. By default, Integration Server enables state sharing for the durable subscriber. Integration Server uses the message processing mode specified for the JMS trigger to set the shared state order mode for the durable subscriber. (A message processing mode of serial maps to a shared state order mode of publisher; a message processing mode of concurrent maps to a shared state order mode of none.)

If the durable subscriber specified by the JMS trigger exists already, Integration Server can update the shared state order mode of the durable subscriber when you save the JMS trigger. To change the shared state order mode for a durable subscriber, change the message processing mode for the JMS trigger and confirm making the change on Broker when prompted by Designer.

If a JMS trigger specifies a durable subscriber that already exists and you want to change the state sharing property of the durable subscriber to true, you need to use the Broker interface in My webMethods to delete the durable subscription. Then, you can either allow Integration Server to re-create the durable subscription by saving the JMS trigger or you can use the Broker interface in My webMethods to re-create the durable subscription with the correct shared state and shared state order values.

Note: When Integration Server creates a destination or durable subscriber on the Broker, Integration Server sets the shared state to true.

Important: If the JMS connection alias is not configured to manage destinations, you must use the Broker interface in My webMethods to manage the destinations and durable subscribers used with JMS triggers.

Consuming JMS Messages in Order with Multiple Consumers

The Java Message Service standard states that messages sent by a session to a destination must be received by consumers in the same order in which the messages were sent. However, the Java Message Service standard does not specify how the JMS provider should distribute messages when multiple consumers receive messages from the same destination. Because each JMS provider is different, it is advisable to review the documentation from your JMS provider to determine how to use load-balanced consumers to receive messages in the same order in which the messages were sent to the destination.

Consuming JMS Messages in Order Using the webMethods Broker

You can configure the webMethods Broker to distribute messages to multiple consumers in the same order in which the webMethods Broker received the messages. To do this, when creating destinations in Designer, set the **Order By** property to **Publisher**. When using the Broker interface in My webMethods to create destinations (queue or durable subscriber), set the **Shared State** property to "publisher".

When a destination has a shared state order of publisher, the webMethods Broker distributes messages to consumers in a serial fashion. This occurs even if multiple load-balanced consumers share the same destination. For example, suppose that Server1 and Server2 request messages from QueueA on behalf of JMS triggers. The webMethods Broker gives Server1 the first message in the queue. The request from Server2 for a message is blocked until Server1 acknowledges the message. Server2 then receives the next message.

Note: By default, Integration Server retrieves and caches up to 10 messages per request for a JMS trigger. The watt.server.jms.trigger.maxPreFetchSize server parameter determines the number of messages retrieved for each request. For more information about this parameter, see *webMethods Integration Server Administrator's Guide*.

7 Working with Cluster Policies

Introduction

When using the webMethods Broker as a JMS provider, Integration Server can send and receive JMS messages in accordance with a cluster policy. The cluster policy, which is applied to the cluster connection factory used by a JMS connection alias, determines the Broker to which the message is sent. Integration Server automatically handles sending and receiving JMS messages using the cluster connection factory. However, the multisend guaranteed and multisend best effort cluster policies have specific requirements for the JMS client sending the message. The following sections provide more information about how Integration Server acts as the JMS client for these policies and explain how to override a cluster policy when sending a JMS message.

Working with the Multisend Guaranteed Policy

The multisend guaranteed policy specifies that the JMS client should send the same JMS message to exactly *n* out of *m*Brokers in the Broker cluster. The publishing operation is successful only if the JMS message is sent to precisely *n*Brokers. The policy specifies that if the JMS client cannot send the JMS message to the precise number of Brokers successfully, the JMS client should not send the JMS message to any Brokers.

To ensure that Integration Server sends the JMS message to the exact number of Brokers, Integration Server uses an XA transaction to perform a two-phase commit. Integration Server manages the entire transaction as part of executing the pub.jms:* sending service. Consequently, regardless of the connection type of the JMS connection alias, the multisend guaranteed policy does not require special configuration or set up in the sending service or the JMS connection alias.

When sending a multisend guaranteed JMS message using a connection with a transaction type of NO_TRANSACTION, Integration Server starts the transaction when it begins executing the pub.jms* service. Integration Server dynamically adds the *n* participating Brokers to the transaction. (Integration Server treats each Broker enlisted in the transaction as an XA resource.) After *n*Brokers receive the message, Integration Server commits the transaction and the pub.jms* service completes execution.

However, if you want more control over the transaction or if you want to enlist other resources in the transaction, use a JMS connection alias with a transaction type set to XA_TRANSACTION or LOCAL_TRANSACTION. Integration Server can then use an implicit or explicit transaction to send the message.

When sending a multisend guaranteed JMS message in an implicit transaction, Integration Server starts the transaction when executing the pub.jms* service that specifies a JMS connection alias of type XA_TRANSACTION or LOCAL_TRANSACTION. Integration Server commits or rolls back the transaction when the top-level service executes to completion. When the transaction type is XA_TRANSACTION, Integration Server logs the state of each transaction. This XA

transaction logging allows Integration Server to recover any transactions that did not complete due to Integration Server failure.

When sending a multisend guaranteed JMS message in an explicit transaction, you control the transactional units of work by defining the start and completion boundaries of the transaction. The parent service that sends the JMS message must use the pub.art.transaction* services to start, commit, and roll back the transaction. Integration Server enlists the Brokers as XA resources when it executes the pub.jms:* service.

In some situations, you might use the built-in service pub.art.transaction:startTransaction to start a transaction explicitly, but then allow Integration Server to commit or roll back the transaction implicitly based on the success or failure of the top-level service.

By default, Integration Server performs transaction logging only when the JMS connection alias has a transaction type of XA_TRANSACTION. However, you can configure Integration Server to perform transaction logging whenever it sends a JMS message as part of a multisend guaranteed policy. For more information about transaction logging, see "Transaction Logging with the Multisend Guaranteed Policy" on page 124.

For more information about sending a JMS message within a transaction, see "Sending a JMS Message as Part of a Transaction" on page 92.

Note: When sending a JMS message as part of a transaction, client side queuing cannot be used. The *useCSQ* input parameter for the pub.jms:send and pub.jms:sendAndWait services must be set to false when sending JMS messages in a transaction. If the *useCSQ* input parameter is set to true and the sending service executes within an explicit or implicit transaction, Integration Server throws a ServiceException.

Error Handling with the Multisend Guaranteed Policy

When sending JMS messages using a connection with a multisend guaranteed policy, how Integration Server handles errors depends on the transaction type of the connection used to send the JMS message.

Error Handling for Transaction Type of NO_TRANSACTION

When sending JMS messages using a multisend guaranteed policy with a connection of type of NO_TRANSACTION, the following error handling may occur:

- If the minimum number of Brokers required by the multisend guaranteed policy are not available, Integration Server will try various combinations of Brokers in the Broker cluster to ensure that the JMS message is sent to the minimum number of Brokers. For example, if the multisend guaranteed policy specifies that the JMS message must be sent to 2 of 4 Brokers in a Broker cluster that consists of BrokerA, BrokerB, BrokerC, and BrokerD. Integration Server might first try to send the JMS message to BrokerA and BrokerB. If BrokerA is not available, Integration Server retries with a different combination of Brokers, such as BrokerB and BrokerC. Integration Server will retry up to two times to send the message using different

combinations of Brokers. If the minimum number of Brokers is not available after the final retry attempt, Integration Server throws an ISRuntimeException.

- If a fatal error occurs while Integration Server is sending messages to multiple Brokers, Integration Server throws a ServiceException and the sending service fails. For example, an invalid destination lookup name or invalid connection factory name results in a ServiceException and thus a fatal error.

Note: When overriding a multisend guaranteed policy and using a connection transaction type of NO_TRANSACTION, if one of the Brokers is not available while Integration Server is sending the message, Integration Server does not retry sending the message with a different combination of Brokers. Instead, Integration Server throws an ISRuntimeException.

Error Handling for Transaction Type of XA_TRANSACTION or LOCAL_TRANSACTION

When sending JMS messages using a multisend guaranteed policy with a connection of type of XA_TRANSACTION or LOCAL_TRANSACTION, the following error handling may occur:

- If the minimum number of Brokers required by the multisend guaranteed policy are not available and the transaction type is XA_TRANSACTION or LOCAL_TRANSACTION, Integration Server throws an ISRuntimeException and the service fails. Integration Server considers this a transient error. Consequently, you can configure service retry to instruct Integration Server to retry the top-level service automatically.
- If a fatal error occurs while Integration Server is sending messages to multiple Brokers, Integration Server throws a ServiceException and the sending service fails. For example, an invalid destination lookup name or invalid connection factory name results in a ServiceException and thus a fatal error. Integration Server rolls back the transaction after a fatal error.
- If a transient error occurs while Integration Server is sending messages to multiple Brokers and the connection transaction type is XA_TRANSACTION or LOCAL_TRANSACTION, Integration Server throws an ISRuntimeException and rolls back the transaction.

You can configure a service to re-execute automatically after an ISRuntimeException occurs. For more information about configuring service retry, see *webMethods Service Development Help*.

Transaction Logging with the Multisend Guaranteed Policy

When executing an XA transaction, Integration Server logs the state of each transaction. This transaction logging allows Integration Server to recover any transactions that did not complete due to Integration Server failure. While this is the most reliable way to

ensure the integrity of a transaction, it may be expensive in terms of performance and it may not always be necessary.

When sending a message using a connection from a cluster connection factory that specifies a multisend guaranteed policy, Integration Server performs transaction logging only if the connection transaction type is XA_TRANSACTION.

However, you might want Integration Server to perform XA transaction logging and XA transaction recovery for all transactions that involve the multisend guaranteed policy, regardless of the connection transaction type. To do this, set the watt.server.jms.guaranteedMultisend.alwaysUseTXLogging parameter to true. For more information about this parameter, see *webMethods Integration Server Administrator's Guide*.

Working with the Multisend Best Effort Policy

The multisend best effort policy specifies that a JMS client send the same JMS message to all, or as many Brokers in the Broker cluster as possible. The publish operation is considered to be successful if even one of the Brokers receives the message. The multisend best effort policy requires the connection to be non-transacted. When sending JMS messages in conjunction with the multisend best effort policy, the connection transaction type must be NO_TRANSACTION. If the connection transaction type is XA_TRANSACTION or LOCAL_TRANSACTION, Integration Server throws a JMSSubsystemException when attempting to enable the connection and the sending service fails.

For information about specifying a transaction type for a JMS connection alias, see *webMethods Integration Server Administrator's Guide*.

Overriding the Cluster Policy when Sending JMS Messages

When Integration Server sends a JMS message using a connection from a cluster connection factory, the policy applied to the cluster connection factory determines the Broker (or Brokers in the case of a multisend best effort or multisend guaranteed policy) to which the message is sent. When a series of JMS messages are sent using the same connection factory, different Brokers might receive the messages. In some situations, you might want the same Broker to receive all of the messages in the series.

For example, suppose that Integration Server sends a group of three JMS messages using a cluster connection factory to which the "random" policy is applied. It might not matter which Broker in the Broker cluster receives the first JMS message, but you might want the same Broker to receive the two remaining messages. For example, you might want the JMS messages to be processed in the same order in which the messages were sent. Or, if the Brokers in the cluster have different receivers that can process the message, you might want the same receiver to process all of the messages.

You can instruct the Broker Server to route the messages to the same Broker (or Brokers in the case of a multisend best effort or multisend guaranteed policy) by overriding the cluster policy.

Overriding the policy consists of two basic tasks:

- Determining the Broker (or Brokers) to which Integration Server sent the initial message.
- Specifying the Broker (or Brokers) to which Integration Server sends a subsequent message.

To accomplish both tasks, most built-in services that send and receive JMS messages (pub.jms*) include a parameter named *JMS_WMClusterNodes* . This parameter is a child of the *JMSMessage/properties* document and *JMSReplyMessage/properties* documents. The *JMS_WMClusterNodes* parameter can be in the input and/or output signatures of the services.

- In the service output, the *JMS_WMClusterNodes* parameter contains the names of the Broker that received the JMS message. For a cluster connection factory with a multisend guaranteed or multisend best effort policy, the *JMS_WMClusterNodes* parameter lists multiple Brokers. The sending Integration Server supplies the service with this information.
- In the service input, the *JMS_WMClusterNodes* parameter specifies the Broker (or Brokers in the case of a multisend guaranteed or multisend best effort policy) to which you want the message sent.

How to Override the Cluster Policy when Sending a JMS Message

You can instruct Broker Server to override the policy applied to a cluster connection factory only when the following conditions are met:

- JMS messages are sent using a JMS connection alias that uses a cluster connection factory.
- JMS messages are sent using the same cluster connection factory. Note that multiple JMS connection aliases can use the same cluster connection factory.
- The cluster connection factory configuration allows the policy to be overridden. (In My webMethods Server, the **Cluster Policy Override** option is selected for the cluster connection factory.)

The following steps describe how to build a service that overrides a cluster policy to specify that the same Broker (or Brokers in the case of a multisend policy) processes a series of JMS messages.

Note: When overriding the policy for a series of JMS messages, the messages do not need to be sent within the same flow service. Information about the Broker that received the initial message needs to be captured after the initial message is sent and then used when sending subsequent messages. This can be done across multiple services as long as each sending service uses the same cluster connection factory. For the sake of simplicity, the following steps explain how to send the messages in a single flow service.

1. Create the flow service that will send the JMS messages.
2. Insert a service to send the first JMS message.

   Send the JMS message by invoking the pub.jms:send or pub.jms:sendAndWait service. For more information about sending JMS messages, see "Sending and Receiving JMS Messages" on page 73.

   If you use the pub.jms:sendAndWait service to perform an asynchronous request-reply, you also need to invoke pub.jms:waitForReply to retrieve the reply message.
3. In the pipeline, add a new String variable to Pipeline Out.
4. Map the value of the *JMS_WMClusterNodes* output parameter to a new String variable in the pipeline.

If you sent the message using...	Then map this to the new String...
pub.jms:send	The value of the service output parameter *JMSMessage/properties/JMS_WMClusterNodes* produced by the pub.jms:send service
pub.jms:sendAndWait (synchronous)	The value of the *JMSMessageReply/properties/ JMS_WMClusterNodes* output parameter produced by the pub.jms:sendAndWait service
pub.jms:sendAndWait (asynchronous)	The value of the *JMSMessageReply/properties/ JMS_WMClusterNodes* output parameter produced by the pub.jms:waitForReply service

   Do not edit the contents of the *JMS_WMClusterNodes* output parameter.
5. Insert a service to send the next JMS message.

   Send the JMS message by invoking the pub.jms:send or pub.jms:sendAndWait service.
6. Map the value of the String field added in step 3 to the *JMSMessage/properties/ JMS_WMClusterNodes* input parameter for the service invoked in step 5.
7. Repeat steps 4–6 for each JMS message that you want to be received by the same Broker (or Brokers).

Notes:

- Make sure to code the service to handle any ISRuntimeExceptions thrown as a result of a Broker Server exception for invalid data or as the result of unavailable Brokers. For more information, see "Exceptions when Overriding Cluster Policies" on page 128.
- When overriding a multisend guaranteed policy and using a connection transaction type of NO_TRANSACTION, if one of the Brokers is not available while Integration Server is sending the message, Integration Server does not retry sending the message

with a different combination of Brokers. Instead, Integration Server throws an ISRuntimeException.

Exceptions when Overriding Cluster Policies

In addition to handling the exceptions that may occur when sending a JMS message, a service that overrides a cluster policy must handle ISRuntimeExceptions that result when policy requirements are not or cannot be met. Integration Server throws an ISRuntimeException after attempting to override a policy for the following general reasons:

- The service sending the JMS message provided invalid data and the Broker Server throws an exception. Integration Server wraps the Broker Server exception and rethrows it as an ISRuntimeException.
- The Brokers specified in the *JMS_WMClusterNodes* input parameter are not available.

Following is a list of situations in which Integration Server throws an ISRuntimeException while attempting to override the connection factory policy when sending a JMS message.

- The *JMS_WMClusterNodes* specifies a single Broker and that Broker is not available. This applies to policies such as round robin, round robin weighted, random, and sticky.
- The *JMS_WMClusterNodes* specifies multiple Brokers and the policy requires that the JMS message be sent to one Broker. This applies to the round robin, round robin weighted, random, and sticky policies.
- The cluster policy is multisend best effort and none of the Brokers specified in *JMS_WMClusterNodes* are available.
- The cluster policy is multisend guaranteed and one or more of the Brokers specified in *JMS_WMClusterNodes* are not available.
- The *JMS_WMClusterNodes* specifies a Broker that is no longer part of the Broker cluster for the cluster connection factory.

A Building a Resource Monitoring Service

About a Resource Monitoring Service

A *resource monitoring service* is a service that you create to check the availability of resources that a trigger uses. Integration Server schedules a system task to execute a resource monitoring service after it suspends a trigger. Specifically, Integration Server suspends a trigger and invokes the associated resource monitoring service when one of the following occurs:

- During exactly-once processing, the document resolver service ends because of an ISRuntimeException and the watt.server.trigger.preprocess.suspendAndRetryOnError property is set to true (the default).
- A retry failure occurs for a non-transacted trigger and the configured retry behavior is "suspend and retry later."
- A transient error occurs for a transacted JMS trigger and the configured behavior when transaction roll back occurs is to suspend the JMS trigger and recover the message.

The same resource monitoring service can be used for multiple triggers. When the service indicates that resources are available, Integration Server resumes all the triggers that use the resource monitoring service.

Service Requirements

A resource monitoring service must do the following:

- Use the pub.trigger:resourceMonitoringSpec as the service signature.
- Check the availability of the resources used by the document resolver service and all the trigger services associated with a trigger. Keep in mind that each condition in a trigger can be associated with a different trigger service. However, you can only specify one resource monitoring service per trigger.
- Return a value of "true" or "false" for the *isAvailable* output parameter. The author of the resource monitoring service determines what criteria makes a resource available.
- Catch and handle any exceptions that might occur. If the resource monitoring service ends because of an exception, Integration Server logs the exception and continues as if the resource monitoring service returned a value of "false" for the *isAvailable* output parameter.

B Building a Document Resolver Service

About a Document Resolver Service

A document resolver service is a service that you create to perform duplicate detection for messages received by a JMS trigger or documents received by a webMethods messaging trigger. Integration Server uses the document resolver service as the final method of duplicate detection.

Service Requirements

The document resolver service must do the following:

- Use pub.jms:documentResolverSpec as the service signature if the service is for a JMS trigger. Use pub.publish:documentResolverSpec as the service signature if the service is for a webMethods messaging trigger. Integration Server passes the document resolver service values for each of the variables declared in the input signature. Integration Server passes the document resolver service values for each of the variables declared in the input signature.
- Return a status of NEW, DUPLICATE, or IN_DOUBT. Integration Server uses the status to determine whether or not to process the message.
- Catch and handle any exceptions that might occur, including an ISRuntimeException. For information about how Integration Server proceeds with duplicate detection when an exception occurs, see "Document Resolver Service for a JMS Trigger" on page 105 and "Document Resolver Service and Exceptions for a JMS Trigger" on page 106.
- Determine how far message processing progressed. If necessary, the document resolver service can issue compensating transactions to reverse the effects of a partially completed transaction.

C Transaction Management

Transaction Management Overview

This appendix provides an overview of transaction management, including transaction types and implicit vs. explicit transactions. It also describes how Integration Server supports the built-in services used to manage explicit transactions. For descriptions of each of the specific built-in transaction management services, see "Built-In Transaction Management Services" on page 137.

Transactions

Integration Server considers a transaction to be one or more interactions with one or more resources that are treated as a single logical unit of work. The interactions within a transaction are either all committed or all rolled back. For example, if a transaction includes multiple database inserts, and one or more inserts fail, all inserts are rolled back.

Transaction Types

Integration Server supports the following kinds of transactions:

- A *local transaction* (LOCAL_TRANSACTION), which is a transaction to a resource's local transaction mechanism
- An *XAResource transaction* (XA_TRANSACTION), which is a transaction to a resource's XAResource transaction mechanism

Integration Server can automatically manage both kinds of transactions without requiring the user to do anything. For more information about implicit transactions, see "Implicit and Explicit Transactions" on page 135.

However, in some cases, users need to explicitly control the transactional units of work. Examples of these cases are provided in "Implicit and Explicit Transactions" on page 135.

To support transactions, Integration Server relies on a built-in Java EE transaction manager. The transaction manager is responsible for beginning and ending transactions, maintaining a transaction context, enlisting newly connected resources into existing transactions, and ensuring that local and XAResource transactions are not combined in illegal ways.

The transaction manager *only* manages operations performed by adapter services, a transacted JMS trigger, or a built-in JMS service that uses a transacted JMS connection alias.

Important: You cannot step or trace a flow that contains a transacted adapter service or a transacted JMS service.

XA Transactions

If an XA transactional connection throws an exception during a service transaction and the exception results in an inconsistent state, you may need to resolve the transaction using the tools provided with the database.

For information about using Integration Server to manage XA transactions, see *webMethods Integration Server Administrator's Guide*.

Implicit and Explicit Transactions

Implicit transactions are automatically handled by the Integration Server transaction manager. When you define an explicit transaction, you define the start-on-completion boundaries of the transaction. As such, implicit and explicit transactions need to be created and managed differently.

The following sections describe implicit and explicit transactions and how to manage them.

Implicit Transactions

With implicit transactions, Integration Server automatically manages both local and XAResource transactions without requiring you to explicitly do anything. That is, the Integration Server starts and completes an implicit transaction with no additional service calls required by the user.

A transaction context, which the transaction manager uses to define a unit of work, starts when one of the following occurs:

- An adapter service is encountered during flow service execution. The connection required by the adapter service is registered with the newly created context and used by the adapter service. If another adapter service is encountered, the transaction context is searched to see if the connection is already registered. If the connection is already registered, the adapter service uses this connection. If the connection is not registered, a new connection instance is retrieved and registered with the transaction.
- Integration Server uses a transacted JMS connection alias to receive messages from the JMS provider for a JMS trigger. A JMS connection alias is considered to be transacted when it has a transaction type of XA TRANSACTION or LOCAL TRANSACTION.
- A built-in JMS service that uses a transacted JMS connection alias to connect to the JMS provider is encountered during flow service execution.

Note that if the top-level flow service invokes another flow, services in the child flow use the same transaction context.

When the top-level flow service completes, the transaction is completed and is either committed or rolled back, depending on the status (success or failure) of the top-level flow service or the JMS trigger service.

A single transaction context can contain any number of XA_TRANSACTION connections but no more than one LOCAL_TRANSACTION connection.

For more information about designing and using flows, see *webMethods Service Development Help*.

Explicit Transactions

You use explicit transactions when you need to explicitly control the transactional units of work. To do this, you use additional services, known as built-in services, in your flow.

A transaction context starts when the pub.art.transaction:startTransaction service is executed. The transaction context is completed when either the pub.art.transaction:commitTransaction or pub.art.transaction:rollbackTransaction service is executed. As with implicit transactions, a single transaction context can contain any number of XA_TRANSACTION connections but no more than one LOCAL_TRANSACTION connection.

Note: With explicit transactions, you must be sure to call either pub.art.transaction:commitTransaction or pub.art.transaction:rollbackTransaction for each pub.art.transaction:startTransaction; otherwise, you will have dangling transactions that will require you to reboot Integration Server. You must also ensure that the startTransaction is outside the SEQUENCE.

A new explicit transaction context can be started within a transaction context, provided that you ensure that the transactions within the context are completed in the reverse order they were started. That is, the last transaction to start should be the first transaction to complete, and so on.

The following example shows a *valid* construct:

```
pub.art.transaction:startTransaction
    pub.art.transaction:startTransaction
        pub.art.transaction:startTransaction
        pub.art.transaction:commitTransaction
    pub.art.transaction:commitTransaction
pub.art.transaction:commitTransaction
```

The following example shows an *invalid* construct:

```
pub.art.transaction:startTransaction
    pub.art.transaction:startTransaction
pub.art.transaction:commitTransaction
    pub.art.transaction:commitTransaction
```

Note: You can use the pub.flow:getLastError service in the SEQUENCE to retrieve the error information when a sequence fails. For more information about using the pub.flow:getLastError service, see *webMethods Integration Server Built-In Services Reference*.

For more information about designing and using flows, see *webMethods Service Development Help*.

Built-In Transaction Management Services

The following table identifies each of the built-in services you can use for transaction management.

Service	Description
pub.art.transaction:commitTransaction	Commits an explicit transaction. It must be used in conjunction with the pub.art.transaction:startTransaction service. If it does not have this corresponding service, your flow service will receive a run time error.
pub.art.transaction:rollbackTransaction	Rolls back an explicit transaction. It must be used in conjunction with the pub.art.transaction:startTransaction service. If it does not have this corresponding service, your flow service will receive a run time error.
pub.art.transaction:setTransactionTimeout	Manually sets a transaction timeout interval for implicit and explicit transactions. When you use this service, you are temporarily overriding the Integration Server transaction timeout interval.
pub.art.transaction:startTransaction	Starts an explicit transaction. It must be used in conjunction with either a pub.art.transaction:commitTransaction service or pub.art.transaction:rollbackTransaction service. If it does not have one of these corresponding services, your flow service will receive an run time error.

For more information about the transaction management services, including detailed descriptions of the service signatures, see *webMethods Integration Server Built-In Services Reference*.