Publish-Subscribe Developer's Guide

Version 10.11

October 2021

WEBMETHODS

This document applies to webMethods Integration Server 10.11 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.

Copyright © 2007-2021 Software AG, Darmstadt, Germany and/or Software AG USA, Inc., Reston, VA, USA, and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA Inc. and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at https://softwareag.com/licenses/.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at https://softwareag.com/licenses/ and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software AG Products / Copyright and Trademark Notices of Software AG Products". These documents are part of the product documentation, located at https://softwareag.com/licenses/ and/or in the root installation directory of the licensed product(s).

Document ID: IS-PS-DG-1011-20211015

Table of Contents

About this GuideAbout this Guide....7
Document Conventions....8
Online Information and Support....9
Data Protection....10

1 An Introduction to the Publish-and-Subscribe Model....11
Introduction....12
What Is the Publish-and-Subscribe Model?....12
webMethods Components....13
Basic Elements in the Publish-and-Subscribe Model....14

2 Overview of Publishing and Subscribing....17
Introduction....18
Overview of Publishing to a webMethods Messaging Provider....18
Overview of Subscribing from webMethods Messaging Provider....27
Overview of Local Publishing....33

3 Steps for Building a Publish-and-Subscribe Solution....37
Introduction....38
Step 1: Research the Integration Problem and Determine Solution....38
Step 2: Determine the Production Configuration....39
Step 3: Create the Messaging Connection Aliases....39
Step 4: Create the Publishable Document Types....39
Step 5: Make the Publishable Document Types Available....39
Step 6: Create the Services that Publish the Documents....40
Step 7: Create Trigger Services to Process the Documents....40
Step 8: Define the Triggers....41

4 Configuring the Integration Server to Publish and Subscribe to Documents....43
Introduction....44
Configuring the Connection to the Messaging Provider....44
Configuring Document Stores....44
Specifying a User Account for Invoking Services Specified in webMethods messaging triggers....45
Configuring Settings for a Document History Database....45
Configuring Integration Server for Key Cross-Reference and Echo Suppression....46
Configuring Integration Server to Handle Native Broker Events....46

5 Publishing Documents....49
The Publishing Services....50
Setting Fields in the Document Envelope....50
Publishing a Document....54

Publishing a Document and Waiting for a Reply....57
Delivering a Document....62
Delivering a Document and Waiting for a Reply....65
Impact of Universal Messaging Changes to Shared Durables on Document Delivery....69
Replying to a Published or Delivered Document....70

6 Working with webMethods Messaging Triggers....75
Overview of Building a webMethods Messaging Trigger....76
Creating a webMethods Messaging Trigger....78
Disabling and Enabling a webMethods Messaging Trigger....91
About Join Time-Outs....93
About Priority Message Processing....95
About Execution Users for webMethods Messaging Triggers....97
About Capacity and Refill Level for the webMethods Messaging Trigger Queue....98
About Document Acknowledgements for a webMethods Messaging Trigger....100
About Message Processing....101
Fatal Error Handling for a webMethods Messaging Trigger....111
About Transient Error Handling for a webMethods Messaging Trigger....112
Exactly-Once Processing for webMethods Messaging Triggers....120
Modifying a webMethods Messaging Trigger....123
Deleting webMethods Messaging Triggers....124
Running a webMethods Messaging Trigger with a Launch Configuration....125
Debugging a webMethods Messaging Trigger....130

7 Exactly-Once Processing for Documents Received by webMethods Messaging Triggers....133
Introduction....134
What Is Document Processing?....134
Overview of Exactly-Once Processing....134
Extenuating Circumstances for Exactly-Once Processing....142
Exactly-Once Processing and Performance....144
Configuring Exactly-Once Processing....144
Building a Document Resolver Service....144
Viewing Exactly-Once Processing Statistics....145
Clearing Exactly-Once Processing Statistics....146

8 Transient Error Handling During Trigger Preprocessing....147
Server and Trigger Properties that Affect Transient Error Handling During Trigger Preprocessing....148
Overview of Transient Error Handling During Trigger Preprocessing....149

9 How Triggers Correspond to Durable Subscribers on Universal Messaging....151

10 Understanding Join Conditions....153
Introduction....154
Join Types....154
Subscribe Path for Documents that Satisfy a Join Condition....155
Join Conditions in Clusters....161

11 Synchronizing Data Between Multiple Resources.....163
Data Synchronization Overview.....164
Data Synchronization with webMethods.....164
Tasks to Perform to Set Up Data Synchronization.....174
Defining How a Source Resource Sends Notification of a Data Change.....175
Defining the Structure of the Canonical Document.....176
Setting Up Key Cross-Referencing in the Source Integration Server.....177
Setting Up Key Cross-Referencing in the Target Integration Server.....181
For N-Way Synchronizations Add Echo Suppression to Services.....184

A Naming Guidelines.....191
Naming Rules for Integration Server Elements.....192
Naming Rules for webMethods Broker Document Fields.....192

B Building a Resource Monitoring Service.....195
Overview.....196
Service Requirements.....196

About this GuideAbout this Guide

- Document Conventions 8
- Online Information and Support 9
- Data Protection 10

Software AG Designer enables users to build integration solutions locally within one webMethods Integration Server or across multiple Integration Servers all exchanging information via Software AG Universal Messaging or webMethods Broker. This guide is for developers and administrators who want to make use of this capability.

With Software AG Designer, you can create webMethods messaging triggers and JMS triggers. A webMethods messaging trigger is trigger that subscribes to and processes documents published/delivered locally or to the Broker or Universal Messaging. A JMS trigger is a trigger that receives messages from a destination (queue or topic) on a JMS provider and then processes those messages. This guide discusses development and use of webMethods messaging triggers only. Where the term triggers appears in this guide, it refers to webMethods messaging triggers.

Note:
Prior to Integration Server 9.5 SP1, a webMethods messaging trigger was called a Broker/local trigger.

Note:webMethods Broker is deprecated.

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

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format *folder.subfolder.service*, APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	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.

Convention	Description
[ ]	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 and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at http://documentation.softwareag.com.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at https://empower.softwareag.com/.

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.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at https://empower.softwareag.com/public_directory.aspx and give us a call.

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.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 An Introduction to the Publish-and-Subscribe Model

- Introduction 12
- What Is the Publish-and-Subscribe Model? 12
- webMethods Components 13
- Basic Elements in the Publish-and-Subscribe Model 14

Introduction

Companies today are tasked with implementing solutions for many types of integration challenges within the enterprise. Many of these challenges revolve around application integration (between software applications and other systems) and fall into common patterns, such as:

- **Propagation.** Propagation of similar business objects from one system to multiple other systems, for example, an order status change or a product price change.
- **Synchronization.** Synchronization of similar business objects between two or more systems to obtain a single view, for example, real-time synchronization of customer, product registration, product order, and product SKU information among several applications. This is the most common issue requiring an integration solution.
  - In a one-way synchronization, there is one system (resource) that acts as a data source and one or more resources that are targets of the synchronization.
  - In a two-way synchronization, every resource is both a potential source and target of a synchronization. There is not a single resource that acts as the primary data resource. A change to any resource should be reflected in all other resources. This is called a two-way synchronization.
- **Aggregation.** Information joined from multiple sources into a common destination system, for example, communicating pharmacy customer records and prescription transactions and website data into a central application and database.

The webMethods product suite provides tools that you can use to design and deploy solutions that address these challenges using a publish-and-subscribe model.

What Is the Publish-and-Subscribe Model?

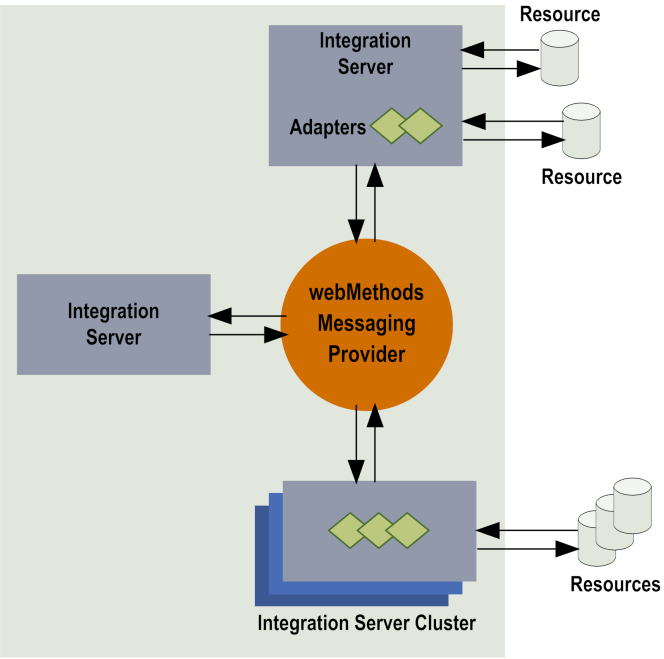
The publish-and-subscribe model is a specific type of message-based solution in which messages are exchanged anonymously through a message broker. Applications that produce information that needs to be shared will make this information available in specific types of recognizable documents that they publish to the message broker. Applications that require information subscribe to the document types they need.

At run time, the message broker receives documents from publishers and then distributes the documents to subscribers. The subscribing application processes or performs work using the document and may or may not send a response to the publishing application.

In a webMethods system, webMethods Integration Server or applications running on Integration Server publish documents to a messaging provider. You can use Universal Messagingand/or Broker as messaging providers. The messaging provider then routes the documents to subscribers (webMethods messaging trigger) located on other Integration Servers. The following sections provide more detail about these components.

webMethods Components

Integration Server and the webMethods messaging providers (Universal Messaging and Broker) share a fast, efficient process for exchanging documents across the entire webMethods system.



Integration Server

Integration Server is the central run-time component of a publish-subscribe solution. It serves as the entry point for the systems and applications that you want to integrate, and is the system's primary engine for the execution of integration logic. It also provides the underlying handlers and facilities that manage the orderly processing of information from resources inside and outside the enterprise. Integration Server publishes documents to and receives documents from Universal Messaging or Broker. For more information about Integration Server, see *webMethods Integration Server Administrator's Guide*.

webMethods Messaging Providers

The messaging provider servers as the intermediary that routes documents between publishers and subscribers. The messaging provider receives, queues, and delivers documents. It provides the infrastructure for implementing synchronous or asynchronous, message-based solutions that are built on the publish-and-subscribe model or one of its variants, request/reply or publish-and-wait.

The webMethods suite includes two messaging providers that you can use:

- Universal Messaging
- Broker

Note:webMethods Broker is deprecated.

For more information about these products, see the respective product documentation.

Basic Elements in the Publish-and-Subscribe Model

The following sections describe the basic building blocks of an integration solution that uses the publish-and-subscribe model.

Documents

In an integration solution built on the publish-and-subscribe model, applications publish and subscribe to *documents*. Documents are objects that webMethods components use to encapsulate and exchange data. A document represents the body of data that a resource passes to webMethods components. Often it represents a business event such as placing an order (purchase order document), shipping goods (shipping notice), or adding a new employee (new employee record).

Each published document includes an envelope. The envelope is much like a header in an email message. The envelope records information such as the sender's address, the time the document was sent, sequence numbers, and other useful information for routing and control. It contains information about the document and its transit through your webMethods system.

Publishable Document Types

Every published document is associated with a publishable document type. A *publishable document type* is a named schema-like definition that describes the structure of a particular kind of document that can be published and subscribed to. An instance of a publishable document type can either be published locally within an Integration Server, or can be published to a to a messaging provider. In a publication environment that includes a messaging provider, each publishable document type is bound to a provider definition.

For more information about publishable document types, see *webMethods Service Development Help*.

Triggers (webMethods Messaging Triggers)

Triggers, specifically webMethods messaging triggers, establish subscriptions to publishable document types. Triggers also specify the services that will process documents received by the subscription. Within a trigger, a condition associates one or more publishable document types with a service.

Prior to Integration Server 9.5 SP1, a webMethods messaging trigger was called a Broker/local trigger.

For more information about triggers, see *webMethods Service Development Help*.

Note:
This guide discusses development and use of webMethods messaging triggers only. Where the terms "trigger" or "triggers" appear in this guide, they refer to webMethods messaging triggers.

Services

Services are method-like units of work. They contain logic that Integration Server executes. You build services to carry out work such as extracting data from documents, interacting with back-end resources, and publishing documents to the messaging provider. When you build a trigger, you specify the service that you want to use to process the documents that you subscribe to.

For more information about building services, see *webMethods Service Development Help*.

Adapter Notifications

Adapter notifications notify your webMethods system whenever a specific event occurs on an adapter's resource. The adapter notification publishes a document when the specified event occurs on the resource. For example, if you are using the Adapter for JDBC and a change occurs in a database table that an adapter notification is monitoring, the adapter notification publishes a document containing data from the event and sends it to Integration Server. Each adapter notification has an associated publishable document type. Integration Server assigns this document type the same name as the adapter notification but appends "PublishDocument" to the name.

You can use triggers to subscribe to the publishable document types associated with adapter notifications. The service associated with the publishable document type in the trigger condition might perform some additional processing, updating, or synchronization based on the contents of the adapter notification.

Canonical Documents

A *canonical document* is a standardized representation that a document might assume while it is passing through your webMethods system. A canonical document acts as the intermediary data format between resources.

For example, in an implementation that accepts purchase orders from companies, one of the steps in the process converts the purchase order document to a company's standard purchase order format. This format is called the 'canonical' form of the purchase order document. The canonical document is published, delivered, and passed to services that process purchase orders.

By converting a document to a neutral intermediate format, subscribers (such as adapter services) only need to know how to convert the canonical document to the required application format. If canonical documents were not used, every subscriber would have to be able to decode the native document format of every publisher.

A canonical document is a publishable document type. The canonical document is used when building publishing services and subscribed to when building triggers. In flow services, you can map documents from the native format of an application to the canonical format.

2 Overview of Publishing and Subscribing

- Introduction 18
- Overview of Publishing to a webMethods Messaging Provider 18
- Overview of Subscribing from webMethods Messaging Provider 27
- Overview of Local Publishing 33

Introduction

In the webMethods system, Integration Servers exchange documents via publication and subscription. One Integration Server publishes a document and one or more Integration Servers subscribe to and process that document.

This chapter provides overviews of how Integration Server interacts with the webMethods messaging provider to publish and subscribe to documents, specifically:

- How Integration Server publishes documents to the webMethods messaging provider.
- How Integration Server retrieves documents from the webMethods messaging provider.
- How Integration Server publishes and subscribes to documents locally.

Overview of Publishing to a webMethods Messaging Provider

Integration Server publishes documents to a webMethods messaging provider, which then routes the documents to all of the subscribers. Integration Server can publish documents to Universal Messaging, Broker, or locally within the Integration Server.

The following sections describe how Integration Server interacts with the webMethods messaging provider in these publishing scenarios:

- Publishing a document to the webMethods messaging provider.
- Publishing a document to the webMethods messaging provider when the webMethods messaging provider is not available.
- Publishing a document to the webMethods messaging provider and waiting for a reply (request/reply).

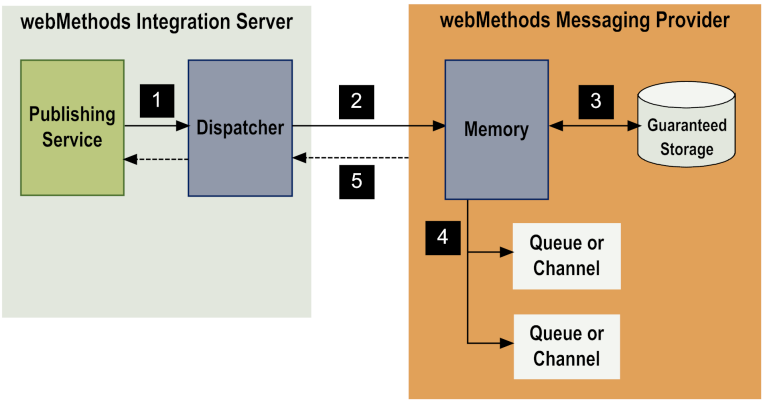
Publishing Documents to the webMethods Messaging Provider

When Integration Server sends documents to a configured webMethods Messaging Provider, Integration Server either publishes or delivers the document.

- When the Integration Server publishes a document, it is broadcast to all subscribers. The webMethods messaging provider routes the document to all clients subscribed to that document.
- When Integration Server delivers a document, the delivery request identifies the document recipient. The webMethods messaging provider enqueues the document for the specified client only.

The following diagram illustrates how Integration Server publishes or delivers documents to webMethods messaging provider when the webMethods messaging provider is available.

Publishing to the webMethods messaging provider



Step	Description
1	A publishing service sends a document to the dispatcher (or an adapter notification publishes a document when an event occurs on the resource the adapter monitors). If validation is configured for the publishable document type, Integration Server validates the document against its publishable document type before sending the document to the dispatcher. If the document is not valid, the service returns an exception specifying the validation error.
2	The dispatcher sends the document to the webMethods messaging provider. **Note:** If the webMethods messaging provider is not available, the document is guaranteed, and use of the client side queue is configured, the dispatcher routes the document to the outbound document store. For more information, see "Publishing Documents When the webMethods Messaging Provider Is Not Available" on page 20.
3	The webMethods messaging provider examines the storage type for the document to determine how to store the document. ■ If the document is volatile, the webMethods messaging provider stores the document in memory. ■ If the document is guaranteed, the webMethods messaging provider stores the document in memory and on disk.
4	The webMethods messaging provider routes the document to subscribers by doing one of the following: ■ If the document was published (broadcast), the webMethods messaging provider identifies subscribers and enqueues a copy of the document for each subscriber. ■ If the document was delivered, the webMethods messaging provider enqueues the document for the client specified in the delivery request. The Broker places documents in a queue for a subscriber. Universal Messaging places documents on a channel.

Step	Description
	A document remains enqueued on the webMethods messaging provider until it is picked up by the subscriber. If the time-to-live for the document elapses, the webMethods messaging provider discards the document. For more information about setting time-to-live for a publishable document type, see *webMethods Service Development Help*. If there are no subscribers for the document and the messaging provider has been configured to handle dead letters (sometimes called dead events), the webMethods messaging provider routes the document to the dead letter queue or dead event store. For more information about configuring the messaging provider to handle documents for which there are no subscribers, refer to the documentation for the webMethods messaging provider that is in use. **Note:** If the Broker is the webMethods messaging provider and a deadletter subscription exists for the document, the Broker deposits the document in the queue containing the deadletter subscription. For more information about creating deadletter subscriptions, see *webMethods Service Development Help*.
5	Integration Server returns control to the publishing service, which executes the next step.

Note:
You can configure publishable document types and Integration Server so that Integration Server does not validate documents when they are published. For more information about validating publishable document types, see *webMethods Service Development Help*.

Publishing Documents When the webMethods Messaging Provider Is Not Available

Integration Server constantly monitors its connection to the webMethods messaging provider and will alter the publishing path if it determines that the webMethods messaging provider used by a publishing service is not available. How Integration Server responds when the webMethods messaging provider is not available depends on:

- The webMethods messaging provider used by the publishing service.

  Note:
  The messaging connection alias assigned to the publishable document type of which the publishing service is publishing an instances determines which webMethods messaging provider the publishing service uses.

- Whether or not the client side queue is configured.

  - For Universal Messaging, you configure the client side queue on a per connection alias basis. The value of the **Enable CSQ** check box for the Universal Messaging connection alias used by the publishing service determines whether or not a client side queue is used when the Universal Messaging server is not available.

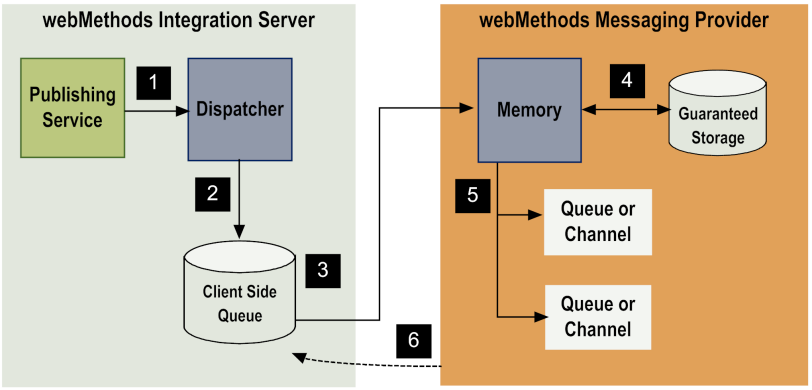
- For Broker, the value of the watt.server.publish.useCSQ parameter determines whether or not the client side queue is used when the Broker is not available. When the watt.server.publish.useCSQ parameter is set to always, the default, Integration Server places documents in the client side queue. When the watt.server.publish.useCSQ parameter is set to never, Integration Server throws a ServiceException.

  Note:
  The client side queue used to store documents published to the Broker is also called the outbound document store.

- For information about configuring the client side queues for Universal Messaging and Broker, see the section About the *Outbound Document Store* in the *webMethods Integration Server Administrator's Guide*.the section About the *Outbound Document Store* in the *webMethods Integration Server Administrator's Guide*.
- The storage type for the publishable document type of which an instance is being published.

The following diagram illustrates how Integration Server publishes documents when the webMethods messaging provider is unavailable.

Publishing when the webMethods messaging provider is unavailable



Step	Description
1	A publishing service sends a document to the dispatcher (or an adapter notification publishes a document when an event occurs on the resource the adapter monitors). If validation is configured for the publishable document type, Integration Server validates the document against its publishable document type before sending the document to the dispatcher. If the document is not valid, the service returns an exception specifying the validation error.
2	The dispatcher detects that the webMethods messaging provider is not available, Integration Server, specifically the publishing service, waits for the connection to the messaging provider to be re-established.

Step	Description
	■ When Universal Messaging is the webMethods messaging provider, Integration Server waits up to the length of time specified for the **Publish Wait Time While Reconnecting** field for the Universal Messaging connection alias used by the publishing service. ■ When Broker is the webMethods messaging provider, Integration Server waits 400 milliseconds for the Broker to become available. If the webMethods messaging provider is not available after the specified wait time elapses, Integration Server does one of the following depending on the storage type of the document: ■ If the document is guaranteed and the client side queue is configured, the dispatcher routes the document to the client side queue. ■ If the document is guaranteed and the client side queue is not configured, Integration Server throws an ISRuntimeException. ■ If the document is volatile, the dispatcher discards the document and the publishing service throws an exception. Integration Server executes the next step in the publishing service.
3	When Integration Server re-establishes a connection to the webMethods messaging provider, Integration Server automatically sends the documents from the client side queue to the messaging provider When Universal Messaging is the messaging provider, Integration Server determines how to drain the client side queue based on the value of the **Drain CSQ in Order** check box for the Universal Messaging connection alias used by the publishing service. ■ When the **Drain CSQ in Order** check box is selected, Integration Server continues to write new messages to the client side queue until the client side queue is completely drained. ■ When the **Drain CSQ in Order** check box is not selected, Integration Server sends new messages directly to the Universal Messaging server while it drains the client side queue. When Broker is the messaging provider, Integration Server determines how to drain the client side queue based on the value of the watt.server.publish.drainCSQInOrder parameter determines how the outbound store is emptied. ■ When watt.server.publish.drainCSQInOrder is set to true (the default), Integration Server sends all newly published documents (guaranteed and volatile) to the client side queue until the outbound store has been emptied. This allows Integration Server to maintain publication order. ■ When watt.server.publish.drainCSQInOrder is set to false, Integration Serversends new messages directly to the Broker while it drains the client side queue.

Step	Description
4	The webMethods messaging provider examines the storage type for the document, determines that it is guaranteed and stores the document in memory and on disk.
5	The webMethods messaging provider routes the document to subscribers by doing one of the following: ■ If the document was published (broadcast), the webMethods messaging provider identifies subscribers and enqueues a copy of the document for each subscriber. ■ If the document was delivered, the webMethods messaging provider enqueues the document for the destination specified in the delivery request. The Broker places documents in a queue for a subscriber. Universal Messaging places documents on a channel. A document remains enqueued on the webMethods messaging provider until it is picked up by the subscriber. If the time-to-live for the document elapses, the webMethods messaging provider discards the document. For more information about setting time-to-live for a publishable document type, see *webMethods Service Development Help*. If there are no subscribers for the document and the messaging provider has been configured to handle dead letters (sometimes called dead events), the webMethods messaging provider routes the document to the dead letter queue or dead event store. For more information about configuring the messaging provider to handle documents for which there are no subscribers, refer to the documentation for the webMethods messaging provider that is in use. **Note:** If the Broker is the webMethods messaging provider and a deadletter subscription exists for the document, the Broker deposits the document in the queue containing the deadletter subscription. For more information about creating deadletter subscriptions, see *webMethods Service Development Help*.
6	The webMethods messaging provider enqueues the document for subscribers. Integration Server removes the document from the client side queue.

Notes:

- You can set the capacity of the client side queue and the outbound document store.
  - For the client side queue for a Universal Messaging connection alias, the **Maximum CSQ Size** field determines the maximum number of documents that can be in the queue. When the client side queue is at capacity, publishing services that use this connection alias will end with an ISRuntimeException.
  - For the outbound document store used with Broker, the watt.server.control.maxPersist server configuration parameter determines the maximum number of documents that can be in the store while the Broker is unavailable. After the outbound document store reaches capacity, the server "blocks" or "pauses" any threads that are executing services that publish

documents. The threads remain blocked until Integration Server begins draining the outbound document store.

- To empty the outbound document store more rapidly, Integration Server sends the documents in batches instead of one at a time. You can use the **Maximum Documents to Send per Transaction** field to specify the maximum number of documents that Integration Server sends from the outbound document store to the Broker. The **Maximum Documents to Send per Transaction** field is located on the **Settings > Resources > Store Settings > Edit Document Store Settings** page in Integration Server Administrator.
- If the initial attempt to publish the document to Universal Messaging from the CSQ fails, Integration Server makes subsequent attempts until the document is published successfully or Integration Server makes the maximum attempts specified in watt.server.messaging.csq.maxRedeliveryCount. If Integration Server makes the maximum redelivery attempts and all attempts have failed, Integration Server discards the document.
- If the initial attempt to publish the document to Broker from the CSQ fails, Integration Server makes subsequent attempts until the document is published successfully or Integration Server makes the maximum attempts specified in watt.server.publish.maxCSQRedeliveryCount. Each attempt to publish to Broker from the CSQ is considered a redelivery attempt Integration Server. After Integration Server makes the specified number of attempts to transmit a document from the CSQ to the Broker and all attempts fail, the audit subsystem logs the document and assigns it a status of STATUS_TOO_MANY_TRIES.
- If Integration Server publishes documents to a Universal Messaging server on which the PauseServerPublishing configuration parameter is set to true, publishing fails with an nPublishPauseException. Integration Server handles the nPublishPauseException by proceeding as if the messaging connection alias is unavailable.
- You can configure publishable document types and Integration Server so that Integration Server does not validate documents when they are published. For more information about validating publishable document types, see *webMethods Service Development Help*.

Tip:
You can use webMethods Monitor to find and resubmit documents with a status of STATUS_TOO_MANY_TRIES or FAILED if the document was to be published to Broker. For more information about using webMethods Monitor, see the webMethods Monitor documentation.

Publishing Documents to a webMethods Messaging Provider and Waiting for a Reply

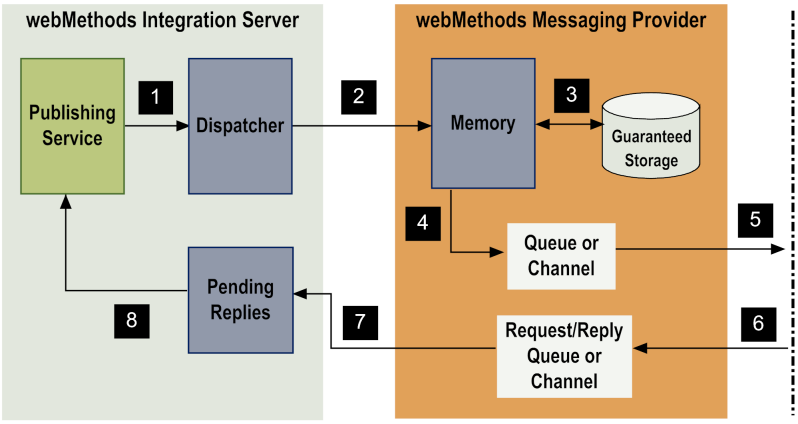
In a publish-and-wait scenario, a service publishes a document (a request) and then waits for a reply document. This is sometimes called the request/reply model. A request/reply can be synchronous or asynchronous.

- In a *synchronous* request/reply, the publishing flow service stops executing while it waits for a response. When the service receives a reply document from the specified client, the service resumes execution.

- In an *asynchronous* request/reply, the publishing flow service continues executing after publishing the request document. That is, the publishing service does not wait for a reply before executing the next step in the flow service. The publishing flow service must invoke a separate service to retrieve the reply document.

The following diagram illustrates how Integration Server and the webMethods messaging provider handle a synchronous request/reply. The webMethods messaging provider can be Broker or Universal Messaging.

Publishing a document to the webMethods messaging provider and waiting for a reply



Step	Description
1	A publishing service sends a document (the request) to the dispatcher. Integration Server populates the *tag* field in the document envelope with a unique identifier that will be used to match up the reply document with this request. The publishing service enters into a waiting state. The service will not resume execution until it receives a reply from a subscriber or the wait time elapses. Integration Server begins tracking the wait time as soon as it publishes the document. If validation is configured for the publishable document type, Integration Server validates the document against its publishable document type before sending the document to the dispatcher. If the document is not valid, the service returns an exception specifying the validation error. The service unblocks, but with an exception.
2	The dispatcher sends the document to the webMethods messaging provider. **Note:** If the webMethods messaging provider is not available, the document is guaranteed, and use of the client side queue is configured, the dispatcher routes the document to the outbound document store. For more information, see "Publishing Documents When the webMethods Messaging Provider Is Not Available" on page 20.
3	The webMethods messaging provider examines the storage type for the document to determine how to store the document.

Step	Description
	■ If the document is volatile, the webMethods messaging provider stores the document in memory. ■ If the document is guaranteed, the webMethods messaging provider stores the document in memory and on disk.
4	The webMethods messaging provider routes the document to subscribers by doing one of the following: ■ If the document was published (broadcast), the webMethods messaging provider identifies subscribers and enqueues a copy of the document for each subscriber. ■ If the document was delivered, the webMethods messaging provider enqueues the document for the destination specified in the delivery request. The Broker places documents in a queue for a subscriber. Universal Messaging places documents on a channel. A document remains enqueued on the webMethods messaging provider until it is picked up by the subscriber. If the time-to-live for the document elapses, the webMethods messaging provider discards the document. For more information about setting time-to-live for a publishable document type, see *webMethods Service Development Help*. If there are no subscribers for the document and the messaging provider has been configured to handle dead letters (sometimes called dead events), the webMethods messaging provider routes the document to the dead letter queue or dead event store. For more information about configuring the messaging provider to handle documents for which there are no subscribers, refer to the documentation for the webMethods messaging provider that is in use. **Note:** If the Broker is the webMethods messaging provider and a deadletter subscription exists for the document, the Broker deposits the document in the queue containing the deadletter subscription. For more information about creating deadletter subscriptions, see *webMethods Service Development Help*.
5	Subscribers retrieve and process the document. A subscriber uses the pub.publish:reply service to compose and publish a reply document. This service automatically populates the *tag* field of the reply document envelope with the same value used in the *tag* field of the request document envelope. The pub.publish:reply service also automatically specifies the requesting client as the recipient of the reply document.
6	One or more subscribers send reply documents to the webMethods messaging provider. The webMethods messaging provider stores the reply documents in memory. The webMethods messaging provider enqueues the reply documents in the request/reply queue or channel for the Integration Server that initiated the request.

Step	Description
7	The Integration Server that initiated the request retrieves the reply documents from the webMethods messaging provider. Integration Server uses the *tag* value of the reply document to match up the reply with the original request.
8	Integration Server places the reply document in the pipeline of the waiting service. The waiting service resumes execution.

Notes:

- If the requesting service specified a publishable document type for the reply document, the reply document must conform to the specified type. Otherwise, the reply document can be an instance of any publishable document type.
- A single request might receive many replies. The Integration Server that initiated the request uses only the first reply document it retrieves from the webMethods messaging provider. Integration Server discards all other replies. *First* is arbitrarily defined. There is no guarantee provided for the order in which the webMethods messaging provider processes incoming replies.
- All reply documents are treated as volatile documents. Volatile documents are stored in memory and will be lost if resource on which the reply document is located shuts down or if a connection is lost while the reply document is in transit.
- When Universal Messaging is the webMethods messaging provider, reply documents use an encoding type of IData.
- If the wait time elapses before the service receives a reply, Integration Server ends the request, and the service returns a null document that indicates the request timed out. Integration Server then executes the next step in the flow service. If a reply document arrives after the flow service resumes execution, Integration Server rejects the document and creates a journal log message stating that the document was rejected because there is no thread waiting for the document.
- You can configure publishable document types and Integration Server so that Integration Server does not validate documents when they are published. For more information about validating publishable document types, see *webMethods Service Development Help*.

Overview of Subscribing from webMethods Messaging Provider

When webMethods messaging trigger receives and processes a document, the path a document follows on the subscriber side includes retrieving the document from the webMethods messaging provider, storing the document on Integration Server, and processing the document. The subscription path for a document depends on whether the document was published to all subscribers (broadcast) or delivered to Integration Server directly.

The following sections describe how Integration Server interacts with the webMethods messaging provider to retrieve published and delivered documents.

Note:

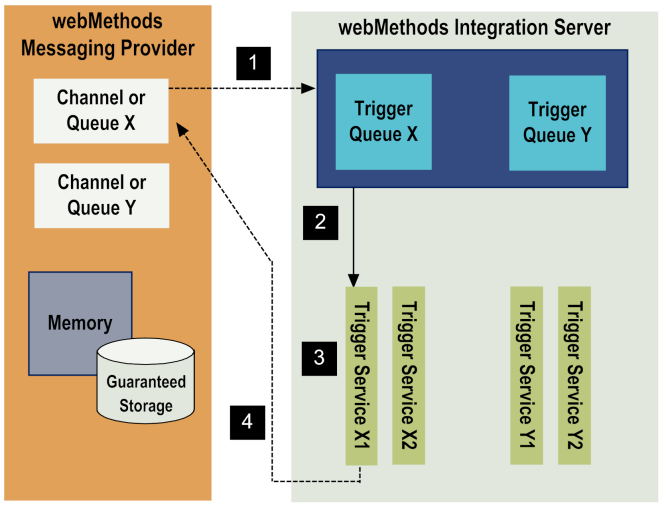
For information about the subscribe path for documents that match a join condition, see "The Subscribe Path for Documents that Satisfy an All (AND) Join Condition" on page 155.

The Subscribe Path for Published Documents

When a document is published or broadcast, the webMethods messaging provider enqueues a copy of the document for each subscriber.

The following diagram illustrates the path of a document to a subscriber (webMethods messaging trigger) on Integration Server.

Subscribe path for published documents



Step	Description
1	Integration Server retrieves documents for a webMethods messaging trigger from the webMethods messaging provider. Integration Server places the documents in the trigger's queue.
2	Integration Server pulls a document from the trigger queue and evaluates the document against the conditions in the trigger. **Note:** If exactly-once processing is configured for the trigger, Integration Server first determines whether the document is a duplicate of one that has already been processed by the trigger. Integration Server continues processing the document only if the document is new.
3	If the document matches a trigger condition, Integration Server executes the trigger service associated with that condition. If the document does not match a trigger condition, Integration Server discards the document and returns an acknowledgement to the webMethods messaging provider.

Step	Description
	Integration Server also generates a journal log message stating that the document did not match a condition.
4	After the trigger service executes to completion (success or error), one of the following occurs: ■ If the trigger service executed successfully, Integration Server returns an acknowledgement to the webMethods messaging provider. ■ If a service exception occurs, the trigger service ends in error and Integration Server rejects the document. If the document is guaranteed, Integration Server returns an acknowledgement to the webMethods messaging provider. Integration Server sends an error document to indicate that an error has occurred. ■ If a transient error occurs during trigger service execution and the service catches the error, wraps it and re-throws it as an ISRuntimeException, then Integration Server waits for the length of the retry interval and re-executes the service using the original document as input. If Integration Server reaches the maximum number of retries and the trigger service still fails because of a transient error, Integration Server treats the last failure as a service error. For more information about retrying a trigger service, see *webMethods Service Development Help*.

Notes:

- After receiving an acknowledgement, the webMethods messaging provider removes its copy of the document from guaranteed storage. Integration Server returns an acknowledgement for guaranteed documents only.
- If Integration Server shuts down or reconnects to the webMethods messaging provider before acknowledging a guaranteed document, Integration Server recovers the document from the webMethods messaging provider when the Integration Server restarts or the connection is re-established. That is, the documents are redelivered. For more information about guaranteed documents, see *webMethods Service Development Help*.
- If a trigger service generates audit data on error and includes a copy of the input pipeline in the service log, you can use webMethods Monitor to re-invoke the trigger service at a later time. For more information about configuring services to generate audit data, see *webMethods Service Development Help*.
- It is possible that a document could satisfy more than one condition in a trigger. However, Integration Server executes only the service associated with the first satisfied condition.
- The processing mode for a trigger determines whether Integration Server processes documents in a trigger queue serially or concurrently. In serial processing, Integration Server processes the documents one at a time in the order in which the documents were placed in the trigger queue. In concurrent processing, Integration Server processes as many documents as it can at one time, but not necessarily in the same order in which the documents were placed in the queue. For more information about document processing, see *webMethods Service Development Help*.

- If a transient error occurs during document retrieval or storage, the audit subsystem logs the document and assigns it a status of FAILED. A *transient error* is an error that arises from a condition that might be resolved later, such as the unavailability of a resource due to network issues or failure to connect to a database. You can use webMethods Monitor to find and resubmit documents with a FAILED status if the document was published locally or received from Broker. For more information about using webMethods Monitor, see the webMethods Monitor documentation.
- You can configure a trigger to suspend and retry at a later time if retry failure occurs. *Retry failure* occurs when Integration Server makes the maximum number of retry attempts and the trigger service still fails because of an ISRuntimeException. For more information about handling retry failure, see *webMethods Service Development Help*.

The Subscribe Path for Delivered Documents When Using webMethods Broker

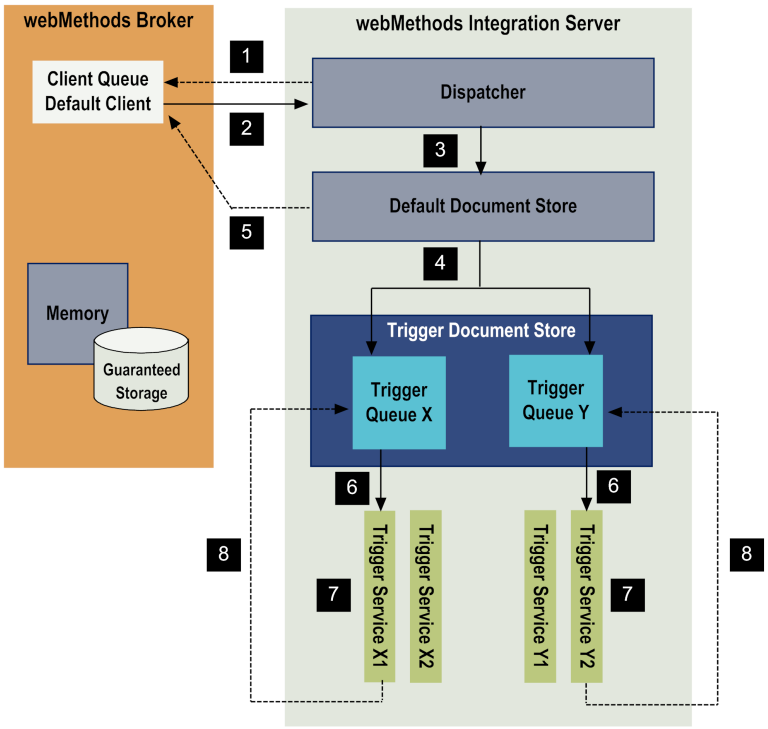
A publishing service can deliver a document by specifying the destination of the document. That is, the publishing service specifies the Broker client that is to receive the document. When the Broker receives a delivered document, it places a copy of the document in the queue for the specified client only.

Typically, documents are delivered to the default client. The default client is the Broker client created for Integration Server when Integration Server first configures its connection to the Broker.

Note:
If a publishing service specifies an individual trigger as the destination of the document (the publishing service specifies a trigger client ID as the destination ID), the subscribe path the document follows is the same as the path followed by a published document.

The following diagram illustrates the subscription path for a document delivered to the default client when Broker is the messaging provider.

Subscribe path for documents delivered to the default client when using Broker



Step	Description
1	The dispatcher on Integration Server requests documents from the default client's queue on the Broker. **Note:** The default client is the Broker client created for Integration Server. Broker places documents in the default client's Broker queue only if the publisher delivered the document to Integration Server's client ID.
2	The thread retrieves documents delivered to the default client in batches. The number of documents the thread retrieves at one time is determined by the capacity and refill level of the default document store and the number of documents available for the default client on the Broker. For more information about configuring the default document store, see the section *Configuring the Default Document Store* in the *webMethods Integration Server Administrator's Guide*. the section *Configuring the Default Document Store* in the *webMethods Integration Server Administrator's Guide*.
3	The dispatcher places a copy of the documents in memory in the default document store.
4	The dispatcher identifies subscribers to the document and routes a copy of the document to each subscriber's trigger queue. In the case of delivered documents, Integration Server saves the documents to a trigger queue. The trigger queue is located within a trigger document store that is saved on disk.

Step	Description
5	Integration Server removes the copy of the document from the default document store and, if the document is guaranteed, returns an acknowledgement to the Broker. The Broker removes the document from the default client's queue.
6	The dispatcher obtains a thread from the server thread pool, pulls the document from the trigger queue, and evaluates the document against the conditions in the trigger. **Note:** If exactly-once processing is configured for the trigger, the Integration Server first determines whether the document is a duplicate of one already processed by the trigger. Integration Server continues processing the document only if the document is new.
7	If the document matches a trigger condition, Integration Server executes the trigger service associated with that condition. If the document does not match a trigger condition, Integration Server, sends an acknowledgement to the trigger queue, discards the document (removes it from the trigger queue), and returns the server thread to the server thread pool. Integration Server also generates a journal log message stating that the document did not match a condition.
8	After the trigger service executes to completion (success or error), one of the following occurs: ■ If the trigger service executed successfully, Integration Server returns an acknowledgement to the trigger queue (if this is a guaranteed document), removes the document from the trigger queue, and returns the server thread to the thread pool. ■ If a service exception occurs, the trigger service ends in error and Integration Server rejects the document, removes the document from the trigger queue, returns the server thread to the thread pool, and sends an error document to indicate that an error has occurred. If the document is guaranteed, Integration Server returns an acknowledgement to the trigger queue. The trigger queue removes its copy of the guaranteed document from storage. ■ If a transient error occurs during trigger service execution and the service catches the error, wraps it and re-throws it as an ISRuntimeException, then Integration Server waits for the length of the retry interval and re-executes the service using the original document as input. If Integration Server reaches the maximum number of retries and the trigger service still fails because of a transient error, Integration Server treats the last failure as a service error. For more information about retrying a trigger service, see *webMethods Service Development Help*.

Notes:

- Integration Server saves delivered documents in a trigger document store located on disk. Integration Server saves published documents in a trigger document store located in memory.
- If Integration Server shuts down before processing a guaranteed document saved in a trigger document store on disk, Integration Server recovers the document from the trigger document

store when it restarts. Volatile documents are saved in memory and are not recovered up restart.

- If a service generates audit data on error and includes a copy of the input pipeline in the service log, you can use webMethods Monitor to re-invoke the trigger service at a later time. For more information about configuring services to generate audit data, see *webMethods Service Development Help*.
- It is possible that a document could match more than one condition in a trigger. However, Integration Server executes only the service associated with the first matched condition.
- The processing mode for a trigger determines whether Integration Server processes documents in a trigger queue serially or concurrently. In serial processing, Integration Server processes the documents one at a time in the order in which the documents were placed in the trigger queue. In concurrent processing, Integration Server processes as many documents as it can at one time, but not necessarily in the same order in which the documents were placed in the queue. For more information about document processing, see *webMethods Service Development Help*.
- If a transient error occurs during document retrieval or storage, the audit subsystem logs the document and assigns it a status of FAILED. You can use webMethods Monitor to find and resubmit documents with a FAILED status if the document was published locally or received from Broker. For more information about using webMethods Monitor, see the webMethods Monitor documentation.
- You can configure a trigger to suspend and retry at a later time if retry failure occurs. *Retry failure* occurs when Integration Server makes the maximum number of retry attempts and the trigger service still fails because of an ISRuntimeException. For more information about handling retry failure, see *webMethods Service Development Help*.

Overview of Local Publishing

Local publishing refers to the process of publishing a document within the Integration Server. Only subscribers located on the same Integration Server can receive and process the document. In local publishing, the document remains within Integration Server. There is no involvement with the webMethods messaging provider.

Note:
While it might be convenient to use local publishing, this solution does not scale. Software AG recommends the use of a messaging provider in production.

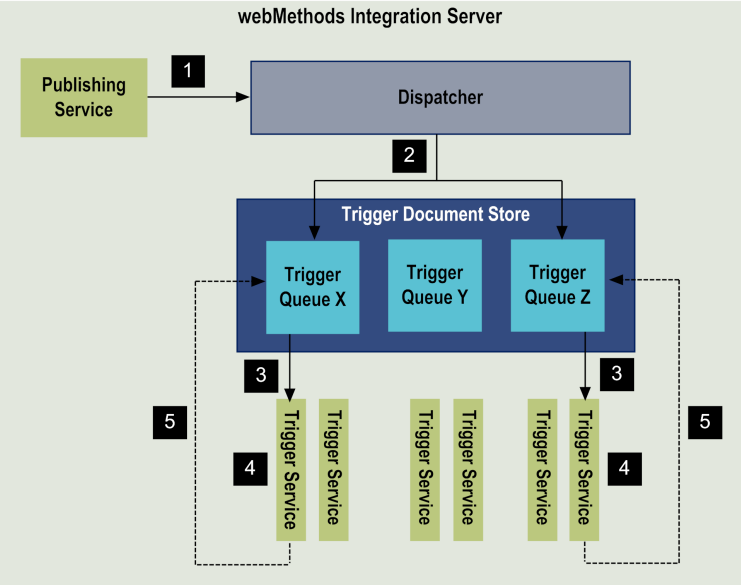
An instance of a publishable document type is published locally in the following situations:

- When the Broker connection alias is assigned to the publishable document type and the service that publishes the document specifies that the document should be published locally.
- When the messaging connection alias assigned to the publishable document type is set to IS_LOCAL_CONNECTION.
- The default messaging connection alias is IS_LOCAL_CONNECTION.

Note:
A publishable document type that specifies Universal Messaging as the messaging provider cannot be published locally.

The following diagram illustrates how the publish and subscribe paths for a locally published document.

Publishing a document locally



Step	Description
1	A publishing service publishes a document, sending the document to the dispatcher. If validation is configured for the publishable document type, Integration Server validates the document against its publishable document type before sending the document to the dispatcher. If the document is not valid, the service returns an exception specifying the validation error.
2	The dispatcher does one of the following: ■ The dispatcher determines which triggers subscribe to the document and places a copy of the document in each subscriber's trigger queue. The dispatcher saves locally published documents in a trigger document store located on disk. ■ If there are no subscribers for the document, the dispatcher discards the document.
3	The dispatcher obtains a thread from the server thread pool, pulls the document from the trigger queue, and evaluates the document against the conditions in the trigger. **Note:**

Step	Description
	If exactly-once processing is configured for the trigger, Integration Server first determines whether the document is a duplicate of one already processed by the trigger. Integration Server continues processing the document only if the document is new.
4	If the document matches a trigger condition, the dispatcher executes the trigger service associated with that condition. If the document does not match a trigger condition, Integration Server sends an acknowledgement to the trigger queue, discards the document (removes it from the trigger queue), and returns the server thread to the server thread pool.
5	After the trigger service executes to completion (success or error), one of the following occurs: ■ If the trigger service executed successfully, Integration Server sends an acknowledgement to the trigger queue (if this is a guaranteed document), removes the document from the trigger queue, and returns the server thread to the thread pool. ■ If a service exception occurs, the trigger service ends in error and Integration Server rejects the document, removes the document from the trigger queue, and returns the server thread to the thread pool. If the document is guaranteed, Integration Server sends an acknowledgement to the trigger queue. ■ If a transient error occurs during trigger service execution and the service catches the error, wraps it and re-throws it as an ISRuntimeException, then Integration Server waits for the length of the retry interval and re-executes the service using the original document as input. If Integration Server reaches the maximum number of retries and the trigger service still fails because of a transient error, Integration Server treats the last failure as a service error. For more information about retrying a trigger service, see *webMethods Service Development Help*.

Notes:

- You can configure publishable document types and Integration Server so that Integration Server does not validate documents when they are published. For more information about validating publishable document types, see *webMethods Service Development Help*.
- Integration Server saves locally published documents in a trigger document store located on disk. If Integration Server shuts down before processing a locally published guaranteed document, Integration Server recovers the document from the trigger document store when it restarts. Integration Server does not recover volatile documents when it restarts.
- If a subscribing trigger queue reaches its maximum capacity, you can configure Integration Server to reject locally published documents for that trigger queue. For more information about this feature, see the description of the watt.server.publish.local.rejectOOS parameter in *webMethods Integration Server Administrator's Guide*.
- If a service generates audit data on error and includes a copy of the input pipeline in the service log, you can use webMethods Monitor to re-invoke the trigger service at a later time. For more

information about configuring services to generate audit data, see *webMethods Service Development Help*.

- It is possible that a document could match more than one condition in a trigger. However, Integration Server executes only the service associated with the first matched condition.
- The processing mode for a trigger determines whether Integration Server processes documents in a trigger queue serially or concurrently. In serial processing, Integration Server processes the documents one at a time in the order in which the documents were placed in the trigger queue. In concurrent processing, Integration Server processes as many documents as it can at one time, but not necessarily in the same order in which the documents were placed in the queue. For more information about document processing, see *webMethods Service Development Help*.
- You can configure a trigger to suspend and retry at a later time if retry failure occurs. *Retry failure* occurs when Integration Server makes the maximum number of retry attempts and the trigger service still fails because of an ISRuntimeException. For more information about handling retry failure, see *webMethods Service Development Help*.
- You can configure Integration Server to strictly enforce a locally published document's time-to-live and discard the document before processing it if the document has expired. For more information about this feature, see the description of the watt.server.trigger.local.checkTTL parameter in *webMethods Integration Server Administrator's Guide*.

3 Steps for Building a Publish-and-Subscribe Solution

- Introduction 38
- Step 1: Research the Integration Problem and Determine Solution 38
- Step 2: Determine the Production Configuration 39
- Step 3: Create the Messaging Connection Aliases 39
- Step 4: Create the Publishable Document Types 39
- Step 5: Make the Publishable Document Types Available 39
- Step 6: Create the Services that Publish the Documents 40
- Step 7: Create Trigger Services to Process the Documents 40
- Step 8: Define the Triggers 41

Introduction

There are two sides of a publish-and-subscribe model integration solution. One side is the publishing side and the other is the subscribing side. The table below lists what you must create for each side of the integration solution.

On the publishing side, create:	On the subscribing side, create:
■ Publishable document types for the documents that are to be published ■ Services that publish the documents	■ Services to process the incoming documents that are published by the publishing side ■ Triggers that associate the incoming documents with services that process the documents

The following table lists the tasks that you need to perform to build an integration solution and whether the publishing side or the subscribing side is responsible for the task.

Step	Task	Publishing	Subscribing
1	Research the integration problem and determine how you want to resolve it.	**X**	**X**
2	Determine the development environment.	**X**	
3	Create the messaging connection aliases.	**X**	
4	Create the publishable document types for the documents to be published.	**X**	**X**
5	Make the publishable document types and messaging connection aliases available to the subscribing side.	**X**	**X**
6	Create the services that publish the documents.	**X**	
7	Create the services that process the published documents.		**X**
8	Define the triggers that associate the published documents to the services that processes the document.		**X**

Step 1: Research the Integration Problem and Determine Solution

The first step to building an integration solution is to define the problem and determine how to solve the problem using the publish-and-subscribe model. When designing the solution, determine:

- **Documents that you are going to need to publish/subscribe.** You will use this information when creating the publishable document types.

- **How you need to publish the documents.** You will use this information when creating the services that publish the documents.
- **How you need to process the documents.** You will use this information when creating the services that process the documents.

Step 2: Determine the Production Configuration

Determine what your production configuration will be like. You might want your development environment to mirror your production environment. Questions to answer are:

- Will all the document publishing and subscribing be performed on a single Integration Server or will you use multiple Integration Servers?
- If you use multiple Integration Servers, will you configure a cluster or will you use a non-clustered group?
- If you use a non-clustered group of Integration Servers, will you configure them to receive messages from the messaging provider in a load balanced manner?
- Will you use Broker or Universal Messaging as the messaging provider in the production environment?

> **Note:**webMethods Broker is deprecated.

Step 3: Create the Messaging Connection Aliases

Once you decide which messaging providers you will be using, create messaging connection aliases. A messaging connection alias defines the configuration needed to establish a connection between Integration Server and a webMethods messaging provider. You can use Universal Messaging and/or Broker as the messaging provider. For each messaging provider that you want to use, you need to create at least one messaging connection alias. For more information about configuring Integration Server for webMethods messaging, see *webMethods Integration Server Administrator's Guide* and "Configuring the Integration Server to Publish and Subscribe to Documents" on page 43.

Step 4: Create the Publishable Document Types

After you determine the documents that you are going to publish in your solution, on the publishing side, use Software AG Designer to create the publishable document types. For more information about how to create publishable document types, see *webMethods Service Development Help*.

Step 5: Make the Publishable Document Types Available

To create services that process documents and triggers that subscribe to documents, the subscribing side needs the publishable document types that define the documents that will be published. The following table describes how to make the publishable document types available based on your development environment.

Development Environment	Action to Take
One Integration Server. Publishing side and subscribing side are being developed on one single Integration Server.	You do not have to take any actions to make the publishable document types available to other developers. After you create the publishable document type for the publishing side, the publishable document type is immediately available for the subscribing side to use.
Multiple Integration Servers with a Broker. Publishing side and subscribing side are each being developed on separate Integration Servers connected by a Broker.	When you create the publishable document type, a corresponding provider definition (Broker document type) is automatically created on the Broker. You can make publishable document types available to other developers one of the following ways: ■ Use Designer to create a publishable document type from the Broker document type. For instructions for how to create a publishable document type from a Broker document type, see *webMethods Service Development Help*. ■ Use package replication to distribute publishable document types to developers working with other Integration Servers. When other developers receive the package, they should install the package and then use Designer to synchronize the document types by pulling them from the Broker.
Multiple Integration Servers with Universal Messaging. Publishing side and subscribing side are each being developed on separate Integration Servers connected by Universal Messaging.	When you create the publishable document type, a corresponding provider definition (channel) is automatically created on the Universal Messaging. You can make publishable document types available to other developers by using package replication or Deployer to distribute document types to developers working with other Integration Servers.

Step 6: Create the Services that Publish the Documents

On the publishing side, you need to create the services that will publish the documents to the messaging provider or locally on the same Integration Server. Use Designer or your own development environment to create these services. For more information about creating publishable services, see *webMethods Service Development Help*.

Step 7: Create Trigger Services to Process the Documents

On the subscribing side, you need to create trigger services that will process the incoming documents. Use Designer or your own development environment to create these services. When creating a service to process a document, include in the input signature a document reference to

the publishable document type for the published document. In this way, you can reference the data in the document using the fields defined in the publishable document type.

For more information about requirements for services that process published documents, or creating services and using document references in input signatures, see *webMethods Service Development Help*.

Step 8: Define the Triggers

On the subscribing side, create triggers to associate one or more publishable document types with the service that processes the published documents. To associate a publishable document type with the service, you create a condition in the trigger that identifies the publishable document type you are subscribing to and the service to invoke when a document of that type arrives. You can further refine the condition by adding filters that specifies criteria for the contents of a published document. When you save the trigger, Integration Server uses the conditions in the trigger to define subscriptions to publishable document types.

For more information about how to define triggers, see *webMethods Service Development Help*.

4 Configuring the Integration Server to Publish and Subscribe to Documents

- Introduction 44
- Configuring the Connection to the Messaging Provider 44
- Configuring Document Stores 44
- Specifying a User Account for Invoking Services Specified in webMethods messaging triggers 45
- Configuring Settings for a Document History Database 45
- Configuring Integration Server for Key Cross-Reference and Echo Suppression 46
- Configuring Integration Server to Handle Native Broker Events 46

Introduction

Before you can begin to publish and subscribe to documents, whether locally or using a webMethods messaging provider, you need to configure Integration Server for webMethods messaging. This configuration consists of using the Integration Server Administrator to:

- Configure one or more connections to one or more messaging providers (Universal Messaging or Broker).
- Configure document stores where Integration Server will save documents until they can be published or processed.
- Specify a user account for executing services specified in webMethods messaging triggers that receive messages from Broker.
- Configure a document history database if you intend to use exactly-once processing.
- Configure a key cross-referencing and echo suppression database.
- Configure settings for handling native Broker events.
- Configure other Integration Server parameters that can affect a publish-and-subscribe solution.

Configuring the Connection to the Messaging Provider

If you want to use one or more webMethods messaging providers (Universal Messaging and/or Broker) as the messaging facility for distributing documents you need to configure at least one messaging connection alias for each messaging provider. If you do not configure a connection to a messaging provider, Integration Server publishes all documents locally.

For detailed information about configuring a connection to the messaging provider (Broker or Universal Messaging), see the section *Authenticating Connections to the Universal Messaging Server* in the *webMethods Integration Server Administrator's Guide*.the section *Authenticating Connections to the Universal Messaging Server* in the *webMethods Integration Server Administrator's Guide*. For more information about the Broker, see *Administering webMethods Broker*.

Configuring Document Stores

When you use Broker as the messaging provider, Integration Server uses *document stores* to save published documents to disk or to memory while the documents are in transit or waiting to be processed. Integration Server maintains three document stores for published documents.

- **Default document store.** The default document store contains guaranteed documents delivered to the default Broker client of Integration Server
- **Trigger document store.** The trigger document store contains locally published guaranteed documents delivered to specific webMethods messaging triggers.
- **Outbound document store.** The outbound document store, sometimes called the client-side queue, contains guaranteed documents waiting to be sent to the Broker. Integration Server places documents in the outbound document store when the configured Broker is not available.

When the connection to the Broker is restored, the server empties the outbound document store by sending the saved documents to the Broker.

Using Integration Server Administrator, you can configure properties for each document store. For example, you can determine the store locations and the initial store sizes. For detailed information about configuring document stores, see the section *Configuring Document Stores* in the *webMethods Integration Server Administrator's Guide*.the section *Configuring Document Stores* in the *webMethods Integration Server Administrator's Guide*.

Specifying a User Account for Invoking Services Specified in webMethods messaging triggers

If a Integration Server receives locally published documents or documents routed through the Broker, you need to specify the user account that Integration Server uses to invoke trigger services. 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 the client is allowed to invoke that service. In a publish-and-subscribe situation, however, Integration Server invokes the service when it receives a document rather than as a result of a client request. Because Integration Server does not associate user credentials with a published document, you can specify the user account for Integration Server to use when invoking services associated with triggers.

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

For more information about setting the **Run Trigger Service As User** property, see the section *Removing References to a User Account* in the *webMethods Integration Server Administrator's Guide*.the section *Removing References to a User Account* in the *webMethods Integration Server Administrator's Guide*.

Note:
If a webMethods messaging trigger receives documents from Universal Messaging, you set the execution user using the **Execution user** property for the trigger.

Configuring Settings for a Document History Database

To provide exactly-once processing for one or more triggers, you need to use a document history database to maintain a record of all the documents processed by those triggers. You or the server administrator must create the Document History database component and connect it to a JDBC connection pool. For instructions, see *Installing Software AG Products*.

Configuring Integration Server for Key Cross-Reference and Echo Suppression

If you intend to use the key cross-reference and echo suppression services to perform data synchronizations, you must store the cross-reference keys and the latching status information in the embedded internal database or in an external RDBMS. Integration Server writes cross-reference data to the embedded internal database by default. If you want to store the key cross-reference information in an external RDBMS, you must create the Cross Reference database component and connect it to a JDBC connection pool. For instructions, see *Installing Software AG Products*.

For more information about the key cross-reference and echo suppression services, see "Synchronizing Data Between Multiple Resources" on page 163.

Configuring Integration Server to Handle Native Broker Events

By default, Integration Server encodes and decodes data it passes to and from the Broker as follows:

- When Integration Server sends a document to the Broker, it first encodes the document (IData object) into a Broker event.
- When Integration Server receives a document from the Broker, it decodes the Broker event into an IData object.

In some situations, you may want to bypass this encoding or decoding step on Integration Server and instead send and receive "native" Broker events to and from the Broker. These situations are when you:

- Migrate Enterprise business logic to Integration Server.
- Use custom Broker clients written in Java, C, or COM/ActiveX.

You configure Integration Server to handle native Broker events by setting server parameters.

To configure Integration Server to handle native Broker events

1. Open the Integration Server Administrator if it is not already open.
2. Go to **Settings > Extended**.
3. Locate the **watt.server.publish.usePipelineBrokerEvent** parameter and change its value to `true`.

   If the **watt.server.publish.usePipelineBrokerEvent** parameter is not displayed, see *webMethods Integration Server Administrator's Guide* for instructions on displaying extended settings.
4. Locate the **watt.server.publish.validateOnIS** parameter and change its value to `never`.
5. If Integration Server is retrieving documents from the Broker on behalf of a trigger, locate the **watt.server.trigger.keepAsBrokerEvent** parameter and change its value to `true`.

6. Click **Save Changes**.

7. Restart Integration Server.

Note:
If you set the **watt.server.trigger.keepAsBrokerEvent** parameter to `true` and the **watt.server.publish.validateOnIS** parameter to `always` or `perDoc`, you will receive validation errors.

5 Publishing Documents

- The Publishing Services 50
- Setting Fields in the Document Envelope 50
- Publishing a Document 54
- Publishing a Document and Waiting for a Reply 57
- Delivering a Document 62
- Delivering a Document and Waiting for a Reply 65
- Impact of Universal Messaging Changes to Shared Durables on Document Delivery 69
- Replying to a Published or Delivered Document 70

The Publishing Services

Using the publishing services, you can create services that publish or deliver documents locally or to the messaging provider. The publishing services are located in the WmPublic package.

The following table describes the services you can use to publish documents.

Service	Description
pub.publish:deliver	Delivers a document to a specified destination.
pub.publish:deliverAndWait	Delivers a document to a specified destination and waits for a response.
pub.publish:publish	Publishes a document locally or to a configured messaging provider (Universal Messaging or Broker). Any clients (triggers) with subscriptions to documents of this type will receive the document.
pub.publish:publishAndWait	Publishes a document locally or to a configured messaging provider (Universal Messaging or Broker) and waits for a response. Any clients (triggers) with subscriptions for the published document will receive the document.
pub.publish:reply	Delivers a reply document in answer to a document received by the client.
pub.publish:waitForReply	Retrieves the reply document for a request published asynchronously.

Setting Fields in the Document Envelope

The document envelope contains information about the published document, such as the publisher's client ID, the client to which error notifications should be sent, a universally unique identification number, and the route the document has taken through the system. The document envelope contains read/write fields as well as read-only fields.

The following table identifies the read/write envelope fields that you might want to set when building a service that publishes documents.

Field name	Description
activation	A String that specifies the activation ID for the published document. If a document does not have an activation ID, Integration Server automatically assigns an activation ID when it publishes the document. Specify an activation ID when you want a trigger to join together documents published by different services. In this case, assign the same activation ID to the documents in the services that publish the documents. For more information about how Integration Server uses activation IDs to satisfy join conditions, see "Understanding Join Conditions" on page 153.

Field name	Description
errorsTo	A String that specifies the client ID to which Integration Server sends an error notification document (an instance of pub.publish.notify:error) if errors occur during document processing by subscribers. If you do not specify a value for *errorsTo*, error notifications are sent to the document publisher. **Note:** The *errorsTo* field is not supported by Universal Messaging.
priority	Specifies the priority level of the message. The priority level indicates how quickly the document should be published and processed. A value of 0 is the lowest processing priority; a value of 9 indicates expedited processing. Set a message priority level in the document envelope when publishing the document. The default priority is 4. **Note:** Priority messaging does not apply to locally published documents. For more information about priority messaging, see *webMethods Service Development Help*.
replyTo	A String that specifies which client ID replies to the published document should be sent to. If you do not specify a *replyTo* destination, responses are sent to the document publisher. **Important:** When you create a service that publishes a document and waits for a reply, do not set the value of the *replyTo* field in the document envelope. By default, Integration Server uses the publisher ID as the *replyTo* value. If you change the *replyTo* value, responses will not be delivered to the waiting service.
uuuid	Universally unique identifier for the document. Integration Server assigns a UUID when publishing the document. However, if a service publishes a document to Universal Messaging using pub.publish:publish, you can specify a value for *uuid*. The *uuid* value that you assign must meet the following criteria: - Must be a minimum of 1 character in length. - Must be a maximum of 96 characters in length. - Must not begin with the sequence "wm:" in any case combination. The "wm:" prefix is reserved for UUIDs generated by Integration Server. If you do not specify a valid *uuid* value, Integration Server assigns one. Integration Server accepts a user-defined UUID only for invocations of pub.publish:publish that publish a document to Universal Messaging. That is,

Field name	Description
	the document being published must be an instance of a publishable document type to which a Universal Messaging connection alias is assigned. For any other publishing service in pub.publish or invocations of pub.publish:publish that publish a document locally or to Broker, Integration Server ignores any assigned *uuid* value. For information about how UUIDs are generated for documents published to Digital Event Services, see "About UUID, EventIds, and Digital Event Services " on page 52.

For more information about the fields in the document envelope, see the description of the pub.publish:envelope document type in the the section *pub.publish:envelope* in the *webMethods Integration Server Built-In Services Reference*. the section *pub.publish:envelope* in the *webMethods Integration Server Built-In Services Reference*.

About the Activation ID

An activation ID is a unique identifier assigned to a published document. Subscribing triggers use the activation ID to determine whether a document satisfies a join condition. Integration Server stores the activation ID in the *activation* field of a document envelope.

By default, Integration Server assigns the same activation ID to each document published within a single top-level service. For example, suppose the processPO service publishes a newCustomer document, a checkInventory document, and a confirmOrder document. Because all three documents are published within the processPO service, Integration Server assigns all three documents the same activation ID.

You can override the default behavior by assigning an activation ID to a document manually. For example, in the pipeline, you can map a variable to the *activation* field of the document. If you want to explicitly set a document's activation ID, you must set it before publishing the document. When publishing the document, the Integration Server will not overwrite an explicitly set value for the *activation* field.

You need to set the activation ID for a document only when you want a trigger to join together documents published by different services. If a trigger will join together documents published within the same execution of a service, you do not need to set the activation ID. Integration Server automatically assigns all the documents the same activation ID.

Tip:
If a service publishes a new document as a result of receiving a document, and you want to correlate the new document with the received document, consider assigning the activation ID of the received document to the new document.

About UUID, EventIds, and Digital Event Services

The UUID is the universally unique identifier for a document (event). Applications rely on the UUID to determine whether or not the application has already received or processed the document.

Due to this reliance, when publishing a document to Universal Messaging and Broker, Integration Server ensures that a UUID is assigned to a document, including automatically assigning one if the UUID is not provided. However, for documents published to Digital Event Services, Integration Server does not automatically generate and assign a UUID to the document. Instead, Integration Server relies on Digital Event Services and on the field used as the EventId annotation to provide a unique ID for the document.

The EventId annotation specifies the name of the field that contains the UUID within the document. At publication time, Digital Event Services ensures that a published document contains a value in the field assigned to the EventId annotation.

At design time, you can:

- Assign a field to the EventId annotation to indicate which field in the document or document reference contains the UUID value. You can identify any String field contained in the publishable document type as the EventId annotation with the exception of a field within the *_env* field. Because the *_env* field is not mapped to a digital event type definition, you cannot specify *_env/uuid* as the EventId field.
- Leave the EventId annotation unassigned. That is, do not assign a field to the EventId annotation. Integration Server updates the digital event type definition to add an internal field named *_guid* that will be used as the EventId annotation.

  Note:
  The *_guid* field is an internal field added to the digital event type definition by Integration Server. The *_guid* field does not appear in the publishable document type.

When building the service that publishes an instance of the publishable document type, you can do one of the following to provide a UUID for the document:

- If you assigned a field to the EventId annotation, provide a value for that field.
- If you did not assign a field to the EventId annotation, you can provide a value for the *_env/uuid* field. Integration Serveruses the *_env/uuid* value as the *_guid* value in the published event.
- Rely on Digital Event Services to generate a UUID by not specifying a value for the field used as the EventId annotation or the *_env/uuid* field

At publication time, to ensure that a document published to Digital Event Services contains a value for EventId, one of the following occurs on Integration Serveror Digital Event Services

- If the publishable document type specifies a field for the EventId annotation and that field is not null, Integration Server publishes the event, making no changes to the event for the EventId.
- If the publishable document type specifies a field for the EventId annotation and that field is null, Digital Event Services generates a UUID and uses the UUID as the value of the field assigned to the EventId annotation.
- If the field used as the EventId annotation contains an empty string “” and Digital Event Services considers the document to be persistent, Digital Event Services rejects the publish and throws an error stating that a non-empty value must be supplied as the EventId. Integration Serverwraps and rethrows the exception and the publish fails.

Note:
Whether or not Digital Event Services considers a document persistent is determined by the delivery mode assigned to the event type that corresponds to the publishable document type.

- If the publishable document type did not specify a field for the EventId annotation and the *_env/uuid* field is not null, Integration Server uses the value of the *_env/uuid* field to populate the *_guid* field.
- If the publishable document type did not specify a field for the EventId annotation and the *_env/uuid* field is null, Digital Event Services generates a UUID and uses the UUID as the value of the *_guid* field in the event.

When a trigger receives an event from Digital Event Services and decodes the event to IData, Integration Server updates the contents of the *_env/uuid* field with the value of the EventId. This ensures that the document has a UUID that can use for duplicate Integration Server detection.

Publishing a Document

When you publish a document using the pub.publish:publish service, the document is broadcast. The service publishes the document for any interested subscribers. Any subscribers to the document can receive and process the document.

The messaging connection alias assigned to the publishable document type determines which messaging provider receives and routes the document. Integration Server publishes the document using the messaging connection alias specified in the **Connection alias name** property of the publishable document type.

Integration Server can publish a document locally instead of sending it to the messaging provider. An instance of a publishable document type can be published locally when:

- The Broker connection alias is assigned to the publishable document type and the *local* input parameter in the pub.publish:publish service to true.
- The IS_LOCAL_CONNCECTION messaging connection alias is assigned to the publishable document type.

How to Publish a Document

The following describes the general steps you take to create a service that publishes a document.

1. Create a document reference to the publishable document type that you want to publish. You can accomplish this by:

   - Declaring a document reference in the input signature of the publishing service

     -OR-

   - Inserting a MAP step in the publishing service and adding the document reference to **Pipeline Out**. You must *immediately* link or assign a value to the document reference. If

you do not, the document reference is automatically cleared the next time the Pipeline view is refreshed.

2. Add content to the document reference. You can add content by linking fields to the document reference or by using the **Set Value** modifier to assign values to the fields in the document reference.

3. Assign values to fields in the envelope (_env field) of the document reference. When a service or adapter notification publishes a document, Integration Server and the messaging provider (Broker or Universal Messaging) automatically assign values to many fields in the document envelope. However, you can manually set some of these fields. Integration Server, Broker, and Universal Messaging do not overwrite fields that you set manually. For more information about assigning values to fields in the document envelope, see "Setting Fields in the Document Envelope" on page 50.

   Note:
   If you assign a value to the *uuid* field in the document envelope, Integration Server uses that value only if the pub.publish:publish service sends a document to Universal Messaging. If the pub.publish:publish service publishes a document to Broker or locally, Integration Server overwrites the assigned *uuid* value with one that Integration Server generates upon publish.

4. Set values for custom header fields. When publishing a document to Universal Messaging, you can add custom header fields to the document by assigning values to a `_properties` document variable in the publishable document type. Integration Server adds the contents of `_properties` as name=value pairs to the header. Subscribers of the publishable document type can register filters that indicate which documents the subscriber wants to receive based on the custom header field contents. When Universal Messaging receives the published document, it applies the filter and only routes the document to the subscriber if the filter criteria is met.

5. Invoke pub.publish:publish to publish the document. This service takes the document you created and publishes it.

   Note: Integration Server issues a ServiceException when the dispatcher is shut down during the execution of this service. Make sure to code your service to handle this situation.

   The pub.publish:publish service expects to find a document (IData object) named *document* in the pipeline. If you are building a flow service, you will need to use the Pipeline view to map the document you want to publish to *document*.

   In addition to the document reference you map into *document*, you must provide the following parameter to pub.publish:publish.

Name	Description
documentTypeName	A String specifying the fully qualified name of the publishable document type that you want to publish. The publishable document type must exist on Integration Server.

   You may also provide the following optional parameters:

Name	Description
local	A String indicating whether you want to publish the document locally. When you publish a document locally, Integration Server does not send the document to the messaging provider. The document remains on the publishing Integration Server. Only subscribers on the same Integration Server can receive and process the document. The *local* parameter applies only when the publishable document type specified for *documentTypeName* uses the Broker connection alias. A publishable document type that specifies Universal Messaging as the messaging provider cannot be published locally. A publishable document type that specifies the IS_LOCAL_CONNECTION alias can be published locally only. Integration Server uses the *local* value only if the publishable document type uses a Broker connection alias. **Set to... / To...** `true` — Publish the document locally. `false` — Publish the document to the configured messaging provider. This is the default.
delayUntilServiceSuccess	A String specifying that Integration Server will delay publishing the document until the top-level service executes successfully. If the top-level service fails, Integration Server will not publish the document. **Set to... / To...** `true` — Delay publishing until after the top-level service executes successfully. **Note:**Integration Server does not return a status when this parameter is set to `true`. `false` — Publish the document when the publish service executes.

Note:
The watt.server.control.maxPublishOnSuccess parameter controls the maximum number of documents that Integration Server can publish on success at one time. You can use this parameter to prevent the server from running out of memory when a service publishes many, large documents on success. By default, this parameter is set to 50,000 documents. Decrease the number of documents that can be published on success to help prevent an out of memory error. For more information about this parameter, see *webMethods Integration Server Administrator's Guide*.

Publishing a Document and Waiting for a Reply

A service can request information from other resources in the webMethods system by publishing a document that contains a query for the information. Services that publish a request document, wait for and then process a reply document follow the *request/reply* model. The request/reply model is a variation of the publish-and-subscribe model. In the request/reply model, a publishing client broadcasts a request for information. Subscribers retrieve the broadcast document, process it, and send a reply document that contains the requested information to the publisher.

A service can implement a synchronous or asynchronous request/reply.

- In a synchronous request/reply, the publishing service stops executing while it waits for a response to a published request. The publishing service resumes execution when a reply document is received or the specified waiting time elapses.
- In an asynchronous request/reply, the publishing service continues to execute after publishing the request document. The publishing service must invoke another service to wait for and retrieve the reply document.

If you plan to build a service that publishes multiple requests and retrieves multiple replies, consider making the requests asynchronous. You can construct the service to publish all the requests first and then collect the replies. This approach can be more efficient than publishing a request, waiting for a reply, and then publishing the next request.

You can use the pub.publish:publishAndWait service to build a service that performs a synchronous or an asynchronous request/reply. If you need a specific client to respond to the request for information, use the pub.publish:deliverAndWait service instead. For more information about using the pub.publish:deliverAndWait service, see "Publishing a Document and Waiting for a Reply" on page 57.

For information about how a Integration Server sends a request and reply, see "Publishing a Document and Waiting for a Reply" on page 57.

How to Publish a Request Document and Wait for a Reply

The following describes the general steps you take to code a service that publishes a request document and waits for a reply.

1. Create a document reference to the publishable document type that you want to publish. You can accomplish this by:

   - Declaring a document reference in the input signature of the publishing service.

     -OR-

   - Inserting a MAP step in the publishing service and adding the document reference to **Pipeline Out**. You must link or assign a value to the document reference *immediately*. If you do not, the document reference is automatically cleared the next time the Pipeline view is refreshed.

2. Add content to the document reference. You can add content by linking fields to the document reference or by using the **Set Value** modifier to assign values to the fields in the document reference.

3. Assign values to fields in the envelope (_env field) of the document reference. When a service or adapter notification publishes a document, Integration Server and the messaging provider (Broker or Universal Messaging) automatically assign values to fields in the document envelope. However, you can manually set some of these fields. Integration Server, Broker, and Universal Messaging do not overwrite fields that you set manually. For more information about assigning values to the document envelope, see "Setting Fields in the Document Envelope" on page 50.

   Important:
   When you create a service that publishes a document and waits for a reply, do not set the value of the *replyTo* field in the document envelope. By default, Integration Server uses the publisher ID as the *replyTo* value. If you change the *replyTo* value, responses will not be delivered to the waiting service.

4. Set values for custom header fields. When publishing a document to Universal Messaging, you can add custom header fields to the document by assigning values to a `_properties` document variable in the publishable document type. Integration Server adds the contents of `_properties` as name=value pairs to the header. Subscribers of the publishable document type can register filters that indicate which documents the subscriber wants to receive based on the custom header field contents. When Universal Messaging receives the published document, it applies the filter and only routes the document to the subscriber if the filter criteria is met.

5. Invoke pub.publish:publishAndWait to publish the document. This service takes the document you created and publishes it.

   The pub.publish:publishAndWait service expects to find a document (IData object) named *document* in the pipeline. If you are building a flow service, you will need to use the Pipeline view to map the document you want to publish to *document*.

   In addition to the document reference you map into *document*, you must provide the following parameters to pub.publish:publishAndWait.

Name	Description
documentTypeName	A String specifying the fully qualified name of the publishable document type that you want to publish an instance of. The publishable document type must exist on Integration Server. If the publishable document type in *documentTypeName* is associated with a Universal Messaging connection alias, the **Enable Request/Reply Channel and Listener** check box must be selected for the alias. When this check box is selected, Integration Server ensures that a request/reply channel exists for the Universal Messaging connection alias on the Universal Messaging server and that Integration Server has a listener that subscribes to the

Name	Description
	alias-specific request/reply channel. If the check box is cleared, there will be no channel in which Universal Messaging can collect replies and no listener with which Integration Server can retrieve replies. The pub.publish:publishAndWait service will end with a ServiceException if the he **Enable Request/Reply Channel and Listener** check box is not selected for the Universal Messaging connection alias.

You may also provide the following optional parameters:

Name	Description
receiveDocument TypeName	A String specifying the fully qualified name of the publishable document type expected as a reply. This publishable document type must exist on your Integration Server. If you do not specify a *receiveDocumentTypeName* value, the service uses the first reply that it receives for this request. **Important:** If you specify a document type, you need to work closely with the developer of the subscribing trigger and the reply service to make sure that the reply service sends a reply document of the correct type.
local	A String indicating whether you want to publish the document locally. When you publish a document locally, the document remains on the publishing Integration Server. Integration Server does not publish the document to the Broker. Only subscribers on the same Integration Server can receive and process the document. The *local* parameter applies only when the publishable document type specified for *documentTypeName* uses the Broker connection alias. A publishable document type that specifies Universal Messaging as the messaging provider cannot be published locally. A publishable document type that specifies the IS_LOCAL_CONNECTION alias can be published locally only. Integration Server uses the *local* value only if the publishable document type uses a Broker connection alias.

Set to...	To...
`true`	Publish the document locally.
`false`	Publish the document to the configured Broker. This is the default.

Name	Description
waitTime	A String specifying how long the publishing service waits (in milliseconds) for a reply document. If you do not specify a *waitTime* value, the service waits until it receives a reply. Integration Server begins tracking the *waitTime* as soon as it publishes the document.
async	A String indicating whether this is a synchronous or asynchronous request.

Set to...	To...
`true`	Indicate that this is an asynchronous request. Integration Server publishes the document and then executes the next step in the service.
`false`	Indicate that this is a synchronous request. Integration Server publishes the document and then waits for the reply. Integration Server executes the next step in the service only after it receives the reply document or the wait time elapses. This is the default.

6. Map the tag field to another pipeline variable. If this service performs a publish and wait request in an asynchronous manner (`async` is set to `true`), the pub.publish:publishAndWait service produces a field named `tag` as output. The `tag` field contains a unique identifier that Integration Server uses to match the request document with a reply document.

   If you create a service that contains multiple asynchronous requests, make sure to link the *tag* output to another field in the pipeline. Each asynchronously published request produces a *tag* field. If the *tag* field is not linked to another field, the next asynchronously published request (that is, the next execution of the pub.publish:publishAndWait service or the pub.publish:deliverAndWait service) will overwrite the first *tag* value.

   Note:
   The *tag* value produced by the pub.publish:publishAndWait service is the same value that Integration Server places in the *tag* field of the request document envelope.

7. Invoke pub.publish:waitForReply to retrieve the reply document. If you configured the pub.publish:publishAndWait service to publish and wait for the document asynchronously, you need to invoke the pub.publish:waitForReply service. This service retrieves the reply document for a specific request.

   The pub.publish:waitForReply service expects to find a String named *tag* in the pipeline. (Integration Server retrieves the correct reply by matching the *tag* value provided to the waitForReply service to the *tag* value in the reply document envelope.) If you are building a flow service, you will need to use the Pipeline view to map the field containing the *tag* value of the asynchronously published request to *tag*.

8. Process the reply document. The pub.publish:publishAndWait (or pub.publish:waitForReply) service produces an output parameter named `receivedDocument` that contains the reply document (an IData object) delivered by a subscriber.

   If the *waitTime* interval elapses before Integration Server receives a reply, the *receivedDocument* parameter contains a null document.

   If the pub.publish:publishAndWait service provides a value for *receiveDocumentTypeName*, the reply document must be of the type specified in the *receiveDocumentTypeName* field.

   Note:
   A single publish and wait request might receive many response documents. The publishing Integration Server uses only the first reply document it receives from the messaging provider. Integration Server discards all other replies. *First* is arbitrarily defined. There is no guarantee provided for the order in which the messaging provider processes incoming replies. If you need a reply document from a specific client, use the pub.publish:deliverAndWait service instead.

Debugging a Flow Service that Performs an Asynchronous Request/Reply with Universal Messaging

Debugging a flow service that performs an asynchronous request/reply where Universal Messaging is the messaging provider has some unique considerations due to the nature of debugging in Designer and the request/reply channel that the publishing service uses to retrieve an asynchronous reply.

- When debugging a flow service using the debug tools in Designer, Designer considers every **Run**, **Step Into**, and **Step Over** action to be a new service execution. When running a service in debug mode, the service execution ends when Designer encounters a breakpoint or the flow terminates. For a step command such as **Step Into**and **Step Over**, the service execution starts and end with each step command.
- The Universal Messaging connection alias used for an asynchronous request/reply must have the **Enable Request/Reply Channel and Listener** option selected. When the Universal Messaging connection alias starts, Integration Server creates a request/reply channel for this Universal Messaging connection alias if one does not yet exist. Integration Server also starts a listener on Integration Server that subscribes to the alias-specific request/reply channel.

In combination, the above behaviors result in the following limitations when debugging a flow service that performs an asynchronous request/reply:

- The step commands cannot be used to step through the flow service. Because Designer considers every **Step Into** or **Step Over** action to be a new service execution, the request/reply channel created by stepping into or over thepub.publish:publishAndWait or pub.publish:deliverAndWait publishing service is also removed as part of the step command. Consequently, there is no channel from which the pub.publish:waitForReply service can retrieve a reply document.
- When using breakpoints, insert the breakpoint at or before the service that initiates the request and/or after the service that retrieves the request. That is, insert the breakpoint at or before the invocation of pub.publish:publishAndWait or pub.publish:deliverAndWait and/or after the invocation of

pub.publish:waitForReply. Do not set a breakpoint on a step that occurs after the initiating the request but before retrieving the reply.

Designer considers every **Run** command to be a new service execution. The request/reply channel created by the publishing service is removed when Designer encounters the breakpoint. Setting a breakpoint after the publish but before retrieving the reply results in the removal of the request reply/channel which means that there is no channel from which the pub.publish:waitForReply service can retrieve a reply document.

Delivering a Document

Delivering a document is much like publishing a document, except that you specify the destination that you want to receive the document. The destination can be a webMethods messaging trigger or the default client/subject of an Integration Server. For a trigger that receives messages from Universal Messaging, you specify the subject that corresponds to the trigger or theIntegration Server. For a trigger that receives messages from the Broker, you specify the client ID that corresponds to the trigger or the Integration Server.

The messaging connection alias assigned to the publishable document type determines which messaging provider receives and routes the document. Integration Server publishes the document using the messaging connection alias specified in the **Connection alias name** property of the publishable document type.

Note:
Instances of publishable document types that use the IS_LOCAL_CONNECTION messaging connection alias cannot be delivered.

How to Deliver a Document

To deliver a document, you invoke the pub.publish:deliver service. The following describes the general steps you take to create a service that delivers a document to a specific destination.

1. Create a document reference to the publishable document type that you want to deliver. You can accomplish this by:

   - Declaring a document reference in the input signature of the publishing service

     -OR-

   - Inserting a MAP step in the publishing service and adding the document reference to **Pipeline Out**. You must *immediately* link or assign a value to the document reference. If you do not, the document reference is automatically cleared the next time the Pipeline view is refreshed.

2. Add content to the document reference. You can add content by linking fields to the document reference or by using the **Set Value** modifier to assign values to the fields in the document reference.

3. Assign values to fields in the envelope (_env field) of the document reference. When a service or adapter notification publishes a document, Integration Server and the messaging provider automatically assign values to fields in the document envelope. However, you can manually set some of these fields. Integration Server and the messaging provider do not overwrite fields that you set manually. For more information about assigning values to fields in the document envelope, see "Setting Fields in the Document Envelope" on page 50.

   Note:
   In the Pipeline view, you can assign values only to fields under **Pipeline Out** or **Service In**.

4. Set values for custom header fields. When publishing a document to Universal Messaging, you can add custom header fields to the document by assigning values to a `_properties` document variable in the publishable document type. Integration Server adds the contents of `_properties` as name=value pairs to the header. Subscribers of the publishable document type can register filters that indicate which documents the subscriber wants to receive based on the custom header field contents. When Universal Messaging receives the published document, it applies the filter and only routes the document to the subscriber if the filter criteria is met.

5. Invoke pub.publish:deliver to deliver the document. This service takes the document you created and publishes it.

   The pub.publish:deliver service expects to find a document (IData object) named *document* in the pipeline. If you are building a flow service, you will need to use the Pipeline view to map the document you want to deliver to *document*.

   In addition to the document reference you map into *document*, you must provide the following parameters to pub.publish:deliver.

Name	Description
documentTypeName	A String specifying the fully qualified name of the publishable document type that you want to publish. The publishable document type must exist on the Integration Server. **Note:** Instances of publishable document types that use the IS_LOCAL_CONNECTION messaging connection alias cannot be delivered.
destID	String specifying the ID for the destination to which you want to deliver the document. You can specify the ID for an individual trigger or the default client of an Integration Server as the destination. When you deliver a document to the default client of Integration Server, Integration Server routes the delivered document to any subscribers on that Integration Server. If you specify an invalid client ID, the Integration Server delivers the document to the messaging provider, but the messaging

Name	Description
	provider never delivers the document to the intended recipient and no error is produced.

You may also provide the following optional parameters:

Name	Description
delayUntilServiceSuccess	A String specifying that Integration Server will delay publishing the document until the top-level service executes successfully. If the top-level service fails, the Integration Server will not publish the document.

Set to...	To...
true	Delay publishing until after the top-level service executes successfully.
false	Publish the document when the pub.publish:deliver service executes. This is the default.

Note:
The watt.server.control.maxPublishOnSuccess parameter controls the maximum number of documents that Integration Server can publish on success at one time. You can use this parameter to prevent the server from running out of memory when a service publishes many, large documents on success. By default, this parameter is set to 50,000 documents. Decrease the number of documents that can be published on success to help prevent an out of memory error. For more information about this parameter, see *webMethods Integration Server Administrator's Guide*.

Cluster Failover and Document Delivery with webMethods Broker

Cluster failover will not occur for a guaranteed document delivered to the shared default client for a cluster of Integration Servers that use Broker as the messaging provider. When the shared default client receives the document, it immediately acknowledges the document to the Broker and places the document in a subscribing trigger queue on one of the Integration Server nodes. If the receiving Integration Server fails before it processes the document, another server in the cluster cannot process the document because the document is stored locally on the receiving server. Additionally, the Broker will not redeliver the document to the cluster because the default client already acknowledged the document to the Broker. The receiving server will process the guaranteed document after the server restarts. (Volatile documents will be lost if the resource on which they are located fails.)

Delivering a Document and Waiting for a Reply

You can initiate and continue a private conversation between two clients by creating a service that delivers a document and waits for a reply. This is a variation of the *request/reply* model. The publishing client executes a service that delivers a document requesting information to a specific client. The subscribing client processes the document and sends the publisher a reply document that contains the requested information.

A service can implement a synchronous or asynchronous request/reply.

- In a synchronous request/reply, the publishing service stops executing while it waits for a response to a published request. The publishing service resumes execution when a reply document is received or the specified waiting time elapses.
- In an asynchronous request/reply, the publishing service continues to execute after publishing the request document. The publishing service must invoke another service to wait for and retrieve the reply document.

If you plan to build a service that publishes multiple requests and retrieves multiple replies, consider making the requests asynchronous. You can construct the service to publish all the requests first and then collect the replies. This approach can be more efficient than publishing a request, waiting for a reply, and then publishing the next request.

You can use the pub.publish:deliverAndWait service to build a service that performs a synchronous or an asynchronous request/reply. This service delivers the request document to a specific client. If multiple clients can supply the requested information, consider using the pub.publish:publishAndWait service instead. For more information about using the pub.publish:publishAndWait service, see "Publishing a Document and Waiting for a Reply" on page 57.

The messaging connection alias assigned to the publishable document type determines which messaging provider receives and routes the document. Integration Server publishes the document using the messaging connection alias specified in the **Connection alias name** property of the publishable document type.

Note:
Instances of publishable document types that use the IS_LOCAL_CONNECTION messaging connection alias cannot be delivered.

If the publishable document type in *documentTypeName* is associated with a Universal Messaging connection alias, the **Enable Request/Reply Channel and Listener** check box must be selected for the alias. When this check box is selected, Integration Server ensures that a request/reply channel exists for the Universal Messaging connection alias on the Universal Messaging server and that Integration Server has a listener that subscribes to the alias-specific request/reply channel. If the check box is cleared, there will be no channel in which Universal Messaging can collect replies and no listener with which Integration Server can retrieve replies. The pub.publish:deliverAndWait service will end with a ServiceException if the **Enable Request/Reply Channel and Listener** check box is not selected for the Universal Messaging connection alias.

Note:

For considerations regarding debugging an asynchronous request/reply when Universal Messaging is the messaging provider, see "Debugging a Flow Service that Performs an Asynchronous Request/Reply with Universal Messaging " on page 61.

How to Deliver a Document and Wait for a Reply

The following describes the general steps that you take to code a service that delivers a document to a specific destination and waits for a reply.

1. Create a document reference to the publishable document type that you want to deliver. You can accomplish this by:

   - Declaring a document reference in the input signature of the publishing service

     -OR-

   - Inserting a MAP step in the publishing service and adding the document reference to **Pipeline Out**. You must *immediately* link or assign a value to the document reference. If you do not, the document reference is automatically cleared the next time the Pipeline view is refreshed.

2. Add content to the document reference. You can add content by linking fields to the document reference or by using the **Set Value** modifier to assign values to the fields in the document reference.

3. Assign values to fields in the envelope (_envfield) of the document reference. When a service or adapter notification publishes a document, Integration Server and the messaging provider automatically assign values to fields in the document envelope. However, you can manually set some of these fields. Integration Server and the messaging provider do not overwrite fields that you set manually. For more information about assigning values to fields in the document envelope, see "Setting Fields in the Document Envelope" on page 50.

   Important:
   When you create a service that delivers a document and waits for a reply, do not set the value of the *replyTo* field in the document envelope. By default, Integration Server uses the publisher ID as the *replyTo* value. If you set the *replyTo* value, responses may not be delivered to the waiting service.

4. Set values for custom header fields. When publishing a document to Universal Messaging, you can add custom header fields to the document by assigning values to a `_properties` document variable in the publishable document type. Integration Server adds the contents of `_properties` as name=value pairs to the header. Subscribers of the publishable document type can register filters that indicate which documents the subscriber wants to receive based on the custom header field contents. When Universal Messaging receives the published document, it applies the filter and only routes the document to the subscriber if the filter criteria is met.

5. Invoke pub.publish:deliverAndWait to publish the document. This service takes the document you created and publishes it to the messaging provider.

The pub.publish:deliverAndWait service expects to find a document (IData object) named *document* in the pipeline. If you are building a flow service, you will need to use the Pipeline view to map the document you want to publish to *document*.

In addition to the document reference you map into *document*, you must provide the following parameters to pub.publish:deliverAndWait.

Name	Description
documentTypeName	A String specifying the fully qualified name of the publishable document type that you want to publish. The publishable document type must exist on Integration Server. **Note:** Instances of publishable document types that use the IS_LOCAL_CONNECTION messaging connection alias cannot be delivered.
destID	A String specifying the ID for the destination to which you want to deliver the document. You can specify the ID for an individual trigger or the default client of an Integration Server as the destination. When you deliver a document to the default client of Integration Server, Integration Server routes the delivered document to any subscribers on that Integration Server. If you specify an invalid client ID, the Integration Server delivers the document to the messaging provider, but the messaging provider never delivers the document to the intended recipient and no error is produced.

You may also provide the following optional parameters:

Name	Description
receiveDocumentTypeName	A String specifying the fully qualified name of the publishable document type expected as a reply. This publishable document type must exist on your Integration Server. If you do not specify a *receiveDocumentTypeName* value, the service uses the first reply document it receives from the client specified in *destID*. **Important:** If you specify a document type, you need to work closely with the developer of the subscribing trigger and the reply service to make sure that the reply service sends a reply document of the correct type.

Name	Description
waitTime	A String specifying how long the publishing service waits (in milliseconds) for a reply document. If you do not specify a *waitTime* value, the service waits until it receives a reply. Integration Server begins tracking the *waitTime* as soon as it publishes the document.
async	A String indicating whether this is a synchronous or asynchronous request.

Set to...	To...
`true`	Indicate that this is an asynchronous request. Integration Server publishes the document and then executes the next step in the service.
`false`	Indicate that this is a synchronous request. Integration Server publishes the document and then waits for the reply. Integration Server executes the next step in the service only after it receives the reply document or the wait time elapses. This is the default.

6. Map the tag field to another pipeline variable. If this service performs a publish and wait request in an asynchronous manner (`async` is set to `true`), the pub.publish:deliverAndWait service produces a field named `tag` as output. The `tag` field contains a unique identifier that Integration Server uses to match the request document with a reply document.

   If you create a service that contains multiple asynchronous requests, make sure to link the *tag* output to another field in the pipeline. Each asynchronously published request produces a *tag* field. If the *tag* field is not linked to another field, the next asynchronously published request (that is, the next execution of the pub.publish:publishAndWait service or the pub.publish:deliverAndWait service) will overwrite the first *tag* value.

   Note:
   The *tag* value produced by the pub.publish:deliverAndWait service is the same value that Integration Server places in the *tag* field of the request document's envelope.

7. Invoke pub.publish:waitForReply to retrieve the reply document. If you configured the pub.publish:deliverAndWait service to publish and wait for the document asynchronously, you need to invoke the pub.publish:waitForReply service. This service retrieves the reply document for a specific request.

   The pub.publish:waitForReply service expects to find a String named *tag* in the pipeline. (Integration Server retrieves the correct reply by matching the *tag* value provided to the waitForReply service to the *tag* value in the reply document envelope.) If you are building a flow service, you will need to use the Pipeline view to map the field.

8. **Process the reply document. The** pub.publish:deliverAndWait **(or** pub.publish:waitForReply**) service produces an output parameter named** receivedDocument **that contains the reply document (an IData object) delivered by a subscriber.**

   If the *waitTime* interval elapses before Integration Server receives a reply, the *receivedDocument* parameter contains a null document.

Impact of Universal Messaging Changes to Shared Durables on Document Delivery

In Universal Messaging version 10.0 the implementation of shared durables was rewritten to improve stability and performance. However, these changes were not compatible with the pub.publish:deliver* services which relied on the previous implementation of shared durables to ensure that delivered documents were routed by Universal Messaging to the intended destination. As of version 10.3 (and in fixes delivered on earlier releases), Universal Messagingimplemented subscriber name filtering for its durable subscriptions. In subscriber name filtering, when a publisher designates a message for a specific subscriber, Universal Messaging routes the message to a consumer whose durable subscription name matches the designated subscriber name in the message. For Integration Server, the subscription name filtering requires some modifications, specifically:

- Enable the subscription name filtering feature on Universal Messaging To enable the feature through Universal Messaging Enterprise Manager, navigate to the realm and select the Config tab. In the realm server configuration panel, click Show Advanced Config. Expand Advanced Configuration - Durable Config, and set Durable Name Filtering to true.
- Change the *destID* input parameter value in invocations of pub.publish:deliver and pub.publish:deliverAndWait, if necessary. This may only be necessary when the trigger to which the message is to be delivered contains an underscore in its name. For more information about client IDs for triggers and how this change may impact existing services that invoke pub.publish:deliver*, see "Client IDs for Triggers and the Destination ID in pub.publish:deliver* Services" on page 69.

Client IDs for Triggers and the Destination ID in pub.publish:deliver* Services

For Integration Server, adopting the Universal Messaging subscription name filtering feature required a change in the naming conventions for the client ID of the trigger. The client ID of the trigger is now the trigger's shared durable name which is as follows:

clientPrefix##triggerName

Where *clientPrefix* is the Universal Messaging connection alias used by the trigger and *triggerName* is the fully qualified name of the trigger where periods and colons are replaced by double underscores.

For example, a trigger that uses a Universal Messaging connection alias with a client prefix of myPrefix and a trigger name of myFolder.mySubfolder:myTrigger the trigger client ID, and therefore the destination ID, would be: myPrefix##myFolder__mySubFolder__myTrigger

Note:
The change to shared durables does not affect the destination ID for the default client of an Integration Serverwhich remains: *clientPrefix*_DefaultClient Where *clientPrefix* is the client prefix for the messaging connection alias used by subscribers to the publishable document type on Integration Server.

The new naming format applies to webMethods messaging trigger created in Integration Server version 10.3 and later.

When calling pub.publish:deliver or pub.publish:deliverAndWait, Integration Server attempts to automatically convert trigger client IDs that are in the old format to the new format during the publishing process. A majority of the time this conversion will work and no changes will be needed to the *destID* input parameter for existing invocations of pub.publish:deliver*. However, if the service delivers a message to a client ID that has an underscore it its name, then the client ID is converted to the wrong value which can result in the message not getting delivered at all or being delivered to the wrong trigger. The previous naming format for trigger client IDs used underscores between the clientID, folder names, and trigger name (specifically *clientPrefix_folderName_subfolderName_triggerName*). When converting the client ID to the new format, Integration Server changes the first underscore (_) to a # and makes all other single underscores (_) into double underscores (__). If a folder, subfolder, or trigger name includes an underscore character, Integration Server replaces that character with a double underscore as well, which results in the incorrect client name.

For example, suppose that a trigger uses a connection alias with a client prefix of myPrefix and the trigger name is my_folder.mySubfolder:myTrigger. Under the previous client ID naming format, the client ID would be: myPrefix_my_folder_mySubfolder_myTrigger. The new format would be myPrefix#my_folder__mySubfolder__myTrigger. However, when executing a pub.publish:deliver* service that includes the old client ID as the destination ID, Integration Server converts the client ID to: myPrefix#my__folder__mySubfolder__myTrigger. This converted client ID does not match the correct client ID for the new format because of the additional underscore character in the folder name portion of the client ID. In this situation, the *destID* field needs to be updated manually with the new client ID format.

The **Messaging > webMethods messaging trigger management >** *triggerName* page in Integration Server Administrator displays the client ID in the new format in the **Trigger Client ID** field.

Replying to a Published or Delivered Document

You can create a service that sends a reply document in response to a published or delivered request document. The reply document might be a simple acknowledgement or might contain information requested by the publisher.

You can build services that send reply documents in response to one or more received documents. For example, if receiving documentA and documentB satisfies an **All (AND)** join condition, you might create a service that sends a reply document to the publisher of documentA and the same reply document to the publisher of documentB.

To send a reply document in response to a document that you receive, you create a service that invokes the pub.publish:reply service.

Keep the following details in mind when creating a service that sends a reply document in response to a received request.

- All reply documents are treated as volatile documents even if the publishable document type used for the reply document has a storage type of guaranteed. Volatile documents are stored in memory. If the resource on which the reply document is stored shuts down before processing the reply document, the reply document is lost. The resource will not recover it upon restart.
- Integration Server always encodes reply messages as IData.
- Integration Server sends the reply document using the same connection that the trigger used to retrieve the request document. Integration Server ignores the messaging connection alias assigned to the publishable document type in *documentTypeName*.
- Because Integration Server sends the reply document using the same connection that the trigger used to retrieve the request document, when Universal Messaging is the messaging provider, the publishable document types to which the trigger subscribes must be associated with a Universal Messaging connection alias for which the **Enable Request/Reply Channel and Listener** check box is selected.

Specifying the Envelope of the Received Document

The pub.publish:reply service contains an input parameter named *receivedDocumentEnvelope*. This parameter identifies the envelope of the request document for which this service creates a reply. Integration Server uses the information in the received document envelope to make sure it delivers the reply document to the correct client.

To determine where to send the reply, Integration Server first checks the value *replyTo* field in the received document envelope. If the *replyTo* field specifies a client ID to which to send responses, Integration Server delivers the reply document to that client. If the *replyTo* field contains no value, Integration Server sends reply documents to the publisher (which is specified in the envelope's *pubID* field).

When you code a service that replies to a document, setting the *receivedDocumentEnvelope* parameter is optional. This field is optional because Integration Server uses the information in the received documents envelope to determine where to send the reply document.

If the service executes because two or more documents satisfied an **All (AND)** join condition, Integration Server uses the envelope of the last document that satisfied the join condition as the value of the *receivedDocumentEnvelope* parameter. For example, suppose that documentA and documentB satisfied an **All (AND)** join condition.

If Integration Server first receives documentA and then receives documentB, Integration Server uses the envelope of documentB as the value of *receivedDocumentEnvelope*. Integration Server sends the reply document only to the client identified in the envelope of documentB. If you want Integration Server to always use the envelope of documentA, link the envelope of documentA to *receivedDocumentEnvelope*.

Tip:
If you want a reply service to send documents to the publisher of documentA and the publisher of documentB, invoke the pub.publish:reply service once for each document. That is, you need to

code your service to contain one pub.publish:reply service that responds to the publisher of documentA and a second pub.publish:reply service that responds to the sender of documentB.

How to Create a Service that Sends a Reply Document

The following describes the general steps you take to code a service that sends a reply document in response to a received document.

1. Declare a document reference to the publishable document type. In the input signature of the service, declare a document reference to the publishable document type for the received document. The name of the document reference must be the fully qualified name of the publishable document type.

   If you intend to use the service to reply to documents that satisfy a join condition (a condition that associates multiple publishable document types with a service), the service's input signature must have a document reference for each publishable document type. The names of these document reference fields must be the fully qualified names of the publishable document type they reference.

2. Create a document reference to the publishable document type that you want to use as the reply document. You can accomplish this by:

   - Declaring a document reference in the input signature of the replying service

     -OR-

   - Inserting a MAP step in the replying service and adding the document reference to **Pipeline Out**. You must *immediately* link or assign a value to the document reference. If you do not, the document reference is automatically cleared the next time the Pipeline view is refreshed.

   Note:
   If the publishing service requires that the reply document be an instance of a specific publishable document type, make sure that the document reference variable refers to this publishable document type.

3. Add content to the reply document. You can add content to the reply document by linking fields to the document reference or by using the **Set Value** modifier to assign values to the fields in the document reference.

4. Assign values to fields in the envelope (_env field) of the reply document. When a service or adapter notification publishes a document, Integration Server and Broker automatically assign values to fields in the document envelope. When you create a service that sends a reply document, Integration Server uses the field in the envelope of the received document to populate the reply document envelope. However, you can manually set some of these fields. Integration Server and Broker do not overwrite fields that you set manually. For more information, see "Setting Fields in the Document Envelope" on page 50.

5. Set values for custom header fields. When publishing a document to Universal Messaging, you can add custom header fields to the document by assigning values to a `_properties` document variable in the publishable document type. Integration Server adds the contents of `_properties` as name=value pairs to the header. Subscribers of the publishable document type can register filters that indicate which documents the subscriber wants to receive based on the custom header field contents. When Universal Messaging receives the published document, it applies the filter and only routes the document to the subscriber if the filter criteria is met.

6. Invoke pub.publish:reply to publish the reply document. This service takes the reply document you created and delivers it to the client specified in the envelope of the received document.

   The pub.publish:reply service expects to find a document (IData object) named *document* in the pipeline. If you are building a flow service, you will need to use the Pipeline view to map the document reference for the document you want to publish to *document*.

   In addition to the document reference that you map into *document*, you must provide the following parameters to the pub.publish:reply service.

Name	Description
documentTypeName	A String specifying the fully qualified name of the publishable document type for the reply document. The publishable document type must exist on Integration Server.

Important:
Services that publish or deliver a request and wait for a reply can specify a publishable document type to which reply documents must conform. If the reply document is not of the type specified in *receiveDocumentTypeName* parameter of the pub.publish:publishAndWait or pub.publish:deliverAndWait service, the publishing service will not receive the reply. You need to work closely with the developer of the publishing service to make sure that your reply document is an instance of the correct publishable document type.

You may also provide the following optional parameters.

Name	Description
receivedDocumentEnvelope	A document (IData object) containing the envelope of the received document. By default, Integration Server uses the information in the received document's envelope to determine where to send the reply document. If the service executes because two or more documents satisfied an **All (AND)** join condition, Integration Server uses the envelope of the last document that satisfied the join condition. If you want Integration Server to always use the envelope from the same document type, link the envelope of that publishable document type to *receivedDocumentEnvelope*. If you want each document publisher to receive a reply document, you must invoke the pub.publish:reply service for each document in the join.

Important:
If the replying service executes because a document satisfied an **Any (OR)** or **Only one (XOR)** join condition, do not map or assign a value to the *receivedDocumentEnvelope*. It is impossible to know which document in the **Any (OR)** or **Only one (XOR)** join will be received first. For example, suppose that an **Only one (XOR)** join condition specified document types documentA and documentB. Integration Server uses the envelope of whichever document it received first as the *receivedDocumentEnvelope* value. If you map the envelope of documentA to *receivedDocumentEnvelope*, but Integration Server receives documentB first, the reply service will fail.

Name	Description
delayUntilServiceSuccess	A String specifying that Integration Server will delay publishing the reply document until the top-level service executes successfully. If the top-level service fails, Integration Server will not publish the reply document.

Set to...	To...
`true`	Delay publishing until after the top-level service executes successfully.
`false`	Publish the document when the pub.publish:reply service executes. This is the default.

7. Build a trigger. For this service to execute when Integration Server receives documents of a specified type, you need to create a trigger. The trigger needs to contain a condition that associates the publishable document type used for the request document with this reply service. For more information about creating a trigger, see *webMethods Service Development Help*.

6 Working with webMethods Messaging Triggers

- Overview of Building a webMethods Messaging Trigger 76
- Creating a webMethods Messaging Trigger 78
- Disabling and Enabling a webMethods Messaging Trigger 91
- About Join Time-Outs 93
- About Priority Message Processing 95
- About Execution Users for webMethods Messaging Triggers 97
- About Capacity and Refill Level for the webMethods Messaging Trigger Queue 98
- About Document Acknowledgements for a webMethods Messaging Trigger 100
- About Message Processing 101
- Fatal Error Handling for a webMethods Messaging Trigger 111
- About Transient Error Handling for a webMethods Messaging Trigger 112
- Exactly-Once Processing for webMethods Messaging Triggers 120
- Modifying a webMethods Messaging Trigger 123
- Deleting webMethods Messaging Triggers 124
- Running a webMethods Messaging Trigger with a Launch Configuration 125
- Debugging a webMethods Messaging Trigger 130

A webMethods messaging trigger subscribes to one or more publishable document types and processes instances of those document types. A webMethods messaging trigger can receive documents published to a webMethods messaging provider (Software AG Universal Messaging or webMethods Broker) or documents published locally by the Integration Server on which the trigger resides.

Each webMethods messaging trigger is composed of two basic pieces:

- A subscription to one or more publishable document types
- A service that processes instances of those publishable document types.

When a webMethods messaging trigger receives a document to which it subscribes from the messaging provider, Integration Server passes the document to the specified service and then invokes the service.

Note:
Prior to Integration Server and Software AG Designer versions 9.5 SP1, a webMethods messaging trigger was called a webMethods Broker/local trigger.

Note:
Information about webMethods messaging triggers is located in *webMethods Service Development Help* and *Publish-Subscribe Developer's Guide*. Both documents include the "Working with webMethods Messaging Triggers" on page 75 topic. *Publish-Subscribe Developer's Guide* contains information such as how webMethods messaging triggers work, how Integration Server receives documents from webMethods messaging triggers, how webMethods messaging triggers with join conditions work, and how Integration Server performs exactly-once processing.

Overview of Building a webMethods Messaging Trigger

Building a webMethods messaging trigger is a process that involves the following basic stages:

Stage 1 Create a new webMethods messaging trigger on Integration Server.

During this stage, you create the new webMethods messaging trigger on the Integration Server where you will do your development and testing. For more information, see "Creating a webMethods Messaging Trigger " on page 78.

Stage 2 Create one or more conditions for the webMethods messaging trigger.

During this stage, you create a trigger condition which associates a subscription to a publishable document types with a service that processes instances of that document types. You can also create filters to apply to incoming documents and select join types.

Stage 3 Set webMethods messaging trigger properties.

During this stage, you set parameters that configure the run-time environment of this webMethods messaging trigger, such as trigger queue capacity, document processing mode, fatal and transient error handling, exactly-once processing, and priority level.

Stage 4 Run the webMethods messaging trigger.

During this stage you can use the tools provided by Designer to run and debug the webMethods messaging trigger. For more information, see "Running a webMethods Messaging Trigger with a Launch Configuration" on page 125.

webMethods Messaging Trigger Requirements

AwebMethods messaging trigger must meet the following requirements:

- The webMethods messaging trigger contains at least one condition.
- Each condition in the webMethods messaging trigger specifies a unique name.
- Each condition in the webMethods messaging trigger specifies a service.
- Each condition in the webMethods messaging trigger specifies at least one publishable document type.
- If more than one condition in the webMethods messaging trigger specifies the same publishable document type and the trigger receives messages from the webMethods Broker, the filters in the conditions must be the same. Specifically, the contents of the **Filter** column must be identical for each condition that subscribes to the publishable document type. Software AG does not recommend using the same publishable document type in more than one condition in the same trigger when receiving messages from webMethods Broker.

  Note:
  Provider filters must be identical if multiple conditions in the same trigger specify the same publishable document type.

- If more than one condition in the webMethods messaging trigger specifies the same publishable document type and the trigger receives messages from Universal Messaging, the provider filters must be identical in each condition but the local filters can be different. Specifically, the contents of the **Provider Filter (UM)** column must be identical for each condition that subscribes to the publishable document type. The contents of the **Filter** column can be different.
- The webMethods messaging trigger contains no more than one join condition.
- The webMethods messaging trigger subscribes to publishable document types that use the same messaging connection alias. For the publishable document types to which the trigger subscribes, the value of the **Connection alias name** property can be:
  - The name of a specific messaging connection alias.
  - DEFAULT where the default messaging connection alias is the same as the alias specified for the other publishable document types to which the trigger subscribes.
- The syntax of a filter applied to a publishable document type is correct. That is, the filter in the **Filter** column must be valid. Integration Server does not validate the provider filter in the **Provider Filter** column.

When you save a webMethods messaging trigger, Integration Server evaluates the webMethods messaging trigger to make sure the webMethods messaging trigger is valid. If Integration Server determines that the webMethods messaging trigger or a condition in the webMethods messaging trigger is not valid, Designer displays an error message and prompts you to cancel the save or continue the save with a disabled webMethods messaging trigger.

Trigger Service Requirements

The service that processes a document received by a webMethods messaging trigger is called a *trigger service*. A condition specifies a single trigger service.

A trigger service for a webMethods messaging trigger must meet the following requirements:

- Before you can enable a webMethods messaging trigger, the trigger service must already exist on the same Integration Server.
- The input signature for the trigger service needs to have a document reference to the publishable document type.
- The name for this document reference must be the fully qualified name of the publishable document type. The fully qualified name of a publishable document type conforms to the following format: folder.subfolder:PublishableDocumentTypeName

  For example, suppose that you want a webMethods messaging trigger to associate the Customers:customerInfo publishable document type with the Customers:addToCustomerStore service. On the **Input/Output** tab of the service, the input signature must contain a document reference named Customers:customerInfo.
- If you intend to use the service in a join condition (a condition that associates multiple publishable document types with a service), the service's input signature must have a document reference for each publishable document type. The names of these document reference fields must be the fully qualified names of the publishable document type they reference.

Note:
An XSLT service cannot be used as a trigger service.

Creating a webMethods Messaging Trigger

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

- The publishable document types and services that you want to use in conditions must already existwebMethods messaging trigger.
- A webMethods messaging trigger can subscribe to publishable document types only. A webMethods messaging trigger cannot subscribe to ordinary IS document types.
- A webMethods messaging trigger must meet the requirements specified in " webMethods Messaging Trigger Requirements" on page 77.

Important:

When you create webMethods messaging triggers, work on a stand-alone Integration Server instead of an Integration Server in a cluster or non-clustered group. Creating, modifying, disabling, and enabling webMethods messaging triggers on an Integration Server in a cluster or non-clustered group can create inconsistencies in the object that corresponds to the trigger on the messaging provider.

To create a webMethods messaging trigger

1. In the Package Navigator view of Designer, click **File > New > webMethods Messaging Trigger**.

2. In the Create a New webMethods Messaging Trigger dialog box, select the folder in which you want to save the webMethods messaging trigger.

3. In the **Element Name** field, type a name for the webMethods messaging trigger using any combination of letters, and/or the underscore character.

   For information about restricted characters, see the *webMethods Service Development Help*.

4. Click **Finish**.

   Designer generates the new webMethods messaging trigger and displays it in the Designer window.

5. Under **Condition detail**, build a condition to specify the document types to which the webMethods messaging trigger subscribes and the trigger services that execute when instances of those document types are received. For more information about creating conditions, see "Creating Conditions" on page 80.

6. In the Properties view, set properties for the webMethods messaging trigger.

7. Click **File > Save**.

Notes:

- Integration Server validates the webMethods messaging trigger before saving it. If Integration Server determines that the webMethods messaging trigger is invalid, Designer prompts you to save the webMethods messaging trigger in a disabled state. For more information about valid webMethods messaging trigger, see " webMethods Messaging Trigger Requirements" on page 77.
- You can also use the pub.trigger:createTrigger service to create a webMethods messaging trigger. For more information about this service, see the *webMethods Integration Server Built-In Services Reference*.

Creating Conditions

A condition associates one or more publishable document types with a single service. A webMethods messaging trigger subscribes to the publishable document type in a subscription. The service, called a trigger services, processes instance of the document type received by the trigger.

A condition can be a simple condition or a join condition. A simple a condition associates one publishable document type with a service. A join associates more than one publishable document types with a service and specifies how the trigger handles the documents as a unit.

A webMethods messaging trigger must have at least one condition.

Keep the following points in mind when you create a condition for a webMethods messaging trigger:

- The publishable document types and services that you want to use in a condition must already exist.
- A webMethods messaging trigger can subscribe to publishable document types only. A webMethods messaging trigger cannot subscribe to ordinary IS document types.
- An XSLT service cannot be used as a trigger service.
- Conditions must meet additional requirements identified in " webMethods Messaging Trigger Requirements" on page 77.
- Trigger services must meet additional requirements identified in "Trigger Service Requirements" on page 78.
- If a webMethods messaging trigger subscribes to a publishable document type that is not in the same package as the trigger, create a package dependency on the package containing the publishable document type from the package containing the trigger. This ensures that Integration Server loads the package containing the publishable document type before loading the trigger.
- If a webMethods messaging trigger uses a trigger service that is not in the same package as the trigger, create a package dependency on the package containing the trigger service from the package containing the trigger. This ensures that Integration Server loads the package containing the service before loading the trigger.

To create a condition for a webMethods messaging trigger

1. In the Package Navigator view of the Service Development perspective, open the webMethods messaging trigger for which you want to set a condition.

2. Under **Conditions**, click to add a new condition.

3. Under **Condition detail**, in the **Name** field, type the name you want to assign to the condition. Designer automatically assigns each condition a default name such as `Condition1` or `Condition2`. You can keep this name or change it to a more descriptive one.

4. In the **Service** field, enter the fully qualified service name that you want to associate with the publishable document types in the condition. You can type in the service name, or click to navigate to and select the service.

5. Click under **Condition detail** to add a new document type subscription for this webMethods messaging trigger .

6. In the Select dialog box, select the publishable document types to which you want to subscribe. You can select more than one publishable document type by using the CTRL or SHIFT keys.

   Designer creates a row for each selected publishable document type. Designer enters the name of the messaging connection alias used by each publishable document type in the **Connection Alias** column.

7. In the **Filter** column next to each publishable document type, do the following:

   - If the publishable document type uses webMethods Broker as the messaging provider, specify a filter that you want Integration Server and/or webMethods Broker to apply to each instance of this publishable document type. For more information, see "Creating Filters for Use with webMethods Broker " on page 87.
   - If the publishable document type uses Universal Messaging as the messaging provider, specify the local filter that you want Integration Server to apply to each instance of the publishable document type received by the trigger. For more information, see "Creating Filters for Use with Universal Messaging " on page 84.

   Create the filter in the **Filter** column using the conditional expression syntax described in *webMethods Service Development Help*.

   Filters are optional for a trigger condition. For more information about filters, see "Using Filters with a Subscription" on page 83.

8. If the publishable document type uses Universal Messaging as the messaging provider, in the **Provider Filter (UM only)** column, enter the filter that you want Universal Messaging to apply to each instance of the publishable document type. Universal Messaging enqueues the document for the trigger only if the filter criteria is met. For information about the syntax for provider filters for Universal Messaging, see the Universal Messaging documentation. For more information about using filters in trigger conditions, see "Creating Filters for Use with Universal Messaging " on page 84.

9. If you specified more than one publishable document type in the condition, select a join type.

Join Type	Description
All (AND)	Integration Server invokes the trigger service when the server receives an instance of each specified publishable document type within the join time-out period. The instance documents must have the same activation ID. This is the default join type.
Any (OR)	Integration Server invokes the trigger service when it receives an instance of any one of the specified publishable document types.
Only one (XOR)	Integration Server invokes the trigger service when it receives an instance of any of the specified document types. For the duration of the join time-out period, Integration Server discards any instances of the specified publishable document types with the same activation ID.

10. Repeat this procedure for each condition that you want to add to the webMethods messaging trigger .

11. Click **File > Save**.

Notes:

- Integration Server validates the webMethods messaging trigger before saving it. If Integration Server determines that the webMethods messaging trigger is invalid, Designer prompts you to save the webMethods messaging trigger in a disabled state. For more information about valid webMethods messaging triggers, see " webMethods Messaging Trigger Requirements" on page 77.
- Integration Server establishes the subscription locally by creating a trigger queue for the webMethods messaging trigger .
- If the trigger subscribes to one or more publishable document types that use webMethods Broker as the messaging provider, one of the following happens upon saving the trigger.
  - If Integration Server is currently connected to the webMethods Broker, Integration Server registers the trigger subscription with the webMethods Broker by creating a client for the trigger on the webMethods Broker. Integration Server also creates a subscription for each publishable document type specified in the webMethods messaging trigger conditions and saves the subscriptions with the webMethods messaging trigger client. webMethods Broker validates the filters in the webMethods messaging trigger conditions when Integration Server creates the subscriptions.
  - If Integration Server is not currently connected to a webMethods Broker, the webMethods messaging trigger will only receive documents published locally. When Integration Server reconnects to a webMethods Broker, the next time Integration Server restarts Integration Server will create a client for the webMethods messaging trigger on the webMethods Broker and create subscriptions for the publishable document types identified in the webMethods messaging trigger conditions. webMethods Broker validates the filters in the webMethods messaging trigger conditions when Integration Server creates the subscriptions.

- If the trigger subscribes to a publishable document type that uses Universal Messaging as the messaging provider, one of the following happens upon saving the trigger.
  - If Integration Server is currently connected to Universal Messaging, Integration Server creates a durable subscription on the channel that corresponds to the publishable document type. A durable subscription can also be referred to as a "durable".
  - If Integration Server is not currently connected to Universal Messaging, you need to synchronize the publishable document type with the provider when the connection becomes available. Synchronizing creates the durable subscription on the channel that corresponds to the publishable document type.
- If a publishable document type specified in a webMethods messaging trigger condition does not exist on the webMethods Broker (that is, there is no associated webMethods Broker document type), Integration Server still creates the trigger client on the webMethods Broker, but does not create any subscriptions. Integration Server creates the subscriptions when you synchronize (push) the publishable document type with the webMethods Broker.
- If a publishable document type specified in a webMethods messaging trigger condition does not exist on Universal Messaging, Designer displays an error stating that a channel does not exist for the specified document type.
- When creating a condition, you can specify the trigger service by dragging a service from Package Navigator view and dropping it in the **Service** field. Additionally, you can specify the document types to which the webMethods messaging trigger subscribes by dragging one or more document types from Package Navigator view and dropping them in the table in **Condition detail**.
- If you need to specify nested fields in the **Filter** field, you can copy a path to the **Filter** field from the document type. Select the field in the document type, right-click and select **Copy**. You can then paste into the **Filter** field. However, you must add % as a preface and suffix to the copied path.

Using Filters with a Subscription

You can further specify the documents that you want a trigger to receive and process by creating a filter for the condition. A filter specifies criteria that the published document must meet before the webMethods messaging trigger receives and/or processes the document. You can use the following types of filters:

- **Provider filter.** A provider filter is saved on the messaging provider. The messaging provider applies the filter when it receives the document from the publisher. If the document meets the filter criteria, the messaging provider enqueues the document for the subscribing trigger.
- **Local filter.** A local filter is saved on Integration Server. After a trigger receives a document, Integration Server applies the filter to the document. If the document meets the filter criteria, Integration Server executes the trigger.

How you create filters for a condition depends on the following:

- The messaging provider used by the publishable document type.

- If the messaging provider is Universal Messaging, the encoding type for the publishable document type.

Creating Filters for Use with Universal Messaging

If a webMethods messaging trigger subscribes to publishable document types associated with a Universal Messaging connection alias, you can create:

- A provider filter that Universal Messaging applies to the documents that it receives. Universal Messaging saves the filter along with the subscription to the document type. When Universal Messaging receives an instance of the publishable document type, Universal Messaging applies the filter. Universal Messaging enqueues the document for the trigger only if the filter criteria is met.

  Use the **Provider Filter (UM only)** column in the Condition detail table to specify a provider filter. For information about the syntax for provider filters for Universal Messaging, see the Universal Messaging documentation.

  When you save a trigger, Integration Server does not evaluate the syntax of the provider filter. Integration Server passes the filter directly to Universal Messaging.

  Note:
  If the trigger contains multiple conditions that subscribe to the same publishable document type, Integration Server does verify that the provider filters are identical upon save. If the supplied provider filters are identical, Integration Server saves the trigger. If the provider filters are not identical, Integration Server throws an exception and considers the trigger to be invalid.

- A local filter that Integration Server applies to the published document header or document body after the trigger receives the document. Use the **Filter** column in the Condition detail table to specify a local filter.

  Create the local filter using the conditional expression syntax described in *webMethods Service Development Help*.

  When you save a trigger, Integration Server evaluates the local filter to make sure it uses the proper syntax. If the syntax is correct, Integration Server saves the webMethods messaging trigger in an enabled state. If the syntax is incorrect, Integration Server saves the webMethods messaging trigger in a disabled state

Universal Messaging Provider Filters and Encoding Type

The encoding type specified for the publishable document type to which the webMethods messaging trigger subscribes determines the scope of the published document to which Universal Messaging applies the filter.

- When IData is the encoding type, Universal Messaging applies the filter to the custom header fields added to a published document via the *_properties* field. The provider filter allows the webMethods messaging trigger to indicate which documents it wants to receive based on the header contents.

- When protocol buffers is the encoding type, Universal Messaging applies the filter to the body of the document only.

  Because Integration Server includes the headers in the body of the published document as well as in the document header, you can still filter on the document headers when the encoding type is protocol buffers.

When using protocol buffers to encode messages, there are additional considerations for creating provider filters for Universal Messaging. Specifically, Universal Messaging *cannot* filter on:

- Fields that cannot be encoded as protocol buffers, including fields whose names contain characters that are not valid for protocol buffers, data types that are not supported by protocol buffers, and duplicate fields.

  For information about fields that cannot be represented as protocol buffers, see the information about setting an encoding type for a publishable document type in the *webMethods Service Development Help*.

- Undeclared fields that are not defined in the publishable document type but are in the published document.
- Fields that contain null values. Even if the field can be represented in protocol buffers, at run time, a null value cannot be included in a protocol buffer message. Consequently, Universal Messaging cannot apply a provider filter that checks for a null value.
- Any list data type where one of the elements in the list contains a null value.
- Fields whose values are references to another field.

While Universal Messaging cannot filter on the contents of the fields identified above, these fields and their contents will be passed through as an UnknownFieldSet which is represented as an IData byte array. A webMethods messaging trigger that receives the document will be able to decode the fields and include them in the pipeline.

For Universal Messaging to filter on protocol buffers, the configuration property **Global Values > ExtendedMessageSelector** must be set to true on Universal Messaging. True is the default.

Use Universal Messaging Enterprise Manager to view and edit the configuration properties for the realm to which Integration Server connects.

Note:
When the encoding type is IData, it is optional to include *_properties* in the provider filter. For example, if you want Universal Messaging to filter for messages where the contents of the *_properties/color* field is equal to "blue", the provider filter would be: color='blue'. However, when the encoding type is protocol buffers, you need to include *_properties* in the provider filter. For example, _properties.color='blue'. If you want a provider filter that operates on the contents of *_properties* to work regardless of the encoding type, always include *_properties* in the filter expression.

Examples of Universal Messaging Provider Filters for Use with Protocol Buffers

Following are examples of provider filters for protocol buffer encoded documents. The Universal Messaging server applies the filter when it receives an instance of a document in the trigger condition that contains the provider filter.

Filter	Evaluates to true when...
stringField = 'abc'	The value of *stringField* is "abc".
stringField < 'B'	The value of the *stringField* is less than B
stringField < '5'	The value of the *stringField* is less than 5.
stringField = '1'	The value of the *stringField* is "1".
document.stringField='abc'	In the Document field named *document*, the value of the *stringField* is "abc".
stringList[1] = 'b'	The value of the second element in *stringList* is "b"
documentList[0].stringField = 'a'	In the first element of the Document list named *documentList*, the value of *stringField* is "a".
stringField1 = 'a' and stringField2 = 'b'	The value of *stringField1* is "a" and the value of *stringField2* is "b".
stringField1 = 'a' or stringField2 = 'b'	The value of *stringField1* is "a" or the value of *stringField2* is "b".
stringField1 = stringField2	The value of *stringField1* is the same as the value of *stringField2*.
integerField = 5 Where *integerField* is an Object field with the Java wrapper type java.lang.Integer applied.	The value of *integerField* is "5".
integerField > 1 Where *integerField* is an Object field with the Java wrapper type java.lang.Integer applied.	The value of *integerField* is greater than 1.
boolean1 = true and boolean2 = false Where *boolean1* and *boolean2* are Object fields with the Java wrapper type java.lang.Boolean applied.	The value of *boolean1* is true and the value of *boolean2* is false.

For more information about server-side filtering on Universal Messaging, including syntax, see the Universal Messaging documentation.

Creating Filters for Use with webMethods Broker

If a webMethods messaging trigger subscribes to publishable document types associated with a webMethods Broker connection alias, you can specify a single filter that can be used by webMethods Broker and/or Integration Server. Use the **Filter** column in the Condition detail table to specify the filter.

Create the filter using the syntax defined in *webMethods Service Development Help*.

The filter can be saved with the subscription on the webMethods Broker and with the webMethods messaging trigger on the Integration Server. This is because some filter syntax that is valid on Integration Server is not valid on webMethods Broker. For example, webMethods Broker prohibits the use of certain words or characters in field names, such as Java keywords, @, *, and names containing white spaces. The location of the filter and whether webMethods Broker and/or Integration Server applies the filter, depends on the filter syntax, which is evaluated at design time.

When you save a webMethods messaging trigger, Integration Server and webMethods Broker evaluate the filter in the **Filter** column.

- Integration Server evaluates the filter to make sure it uses the proper syntax. If the syntax is correct, Integration Server saves the webMethods messaging trigger in an enabled state. If the syntax is incorrect, Integration Server saves the webMethods messaging trigger in a disabled state.
- webMethods Broker evaluates the filter syntax to determine if the filter syntax is valid on the webMethods Broker. If webMethods Broker determines that the syntax is valid for the webMethods Broker, it saves the filter with the document type subscription. If the webMethods Broker determines that the filter syntax is not valid on the webMethods Broker or if attempting to save the filter on the webMethods Broker would cause an error, webMethods Broker saves the subscription without the filter.

  webMethods Broker saves as much of a filter as possible with the subscription. For example, suppose that a filter consists of more than one expression, and only one of the expressions contains the syntax that the webMethods Broker considers invalid. webMethods Broker saves the expressions it considers valid with the subscription on the webMethods Broker. (Integration Server saves all the expressions.)

When a filter is saved only on Integration Server and not on webMethods Broker, the performance of Integration Server can be affected. When the webMethods Broker applies the filter to incoming documents, it discards documents that do not meet filter criteria. Integration Server only receives documents that meet the filter criteria. If the subscription filter resides only on Integration Server, webMethods Broker automatically places the document in the subscriber's queue. webMethods Broker routes all the documents to the subscriber, creating greater network traffic between the webMethods Broker and the Integration Server and requiring more processing by the Integration Server.

For more information about the syntax for local filters, see the *webMethods Service Development Help*.

Note:webMethods Broker is deprecated.

Using Hints in Filters

Hints are used to further define a filter. You add hints to the end of a subscription in the **Filter** field.

Hints use the following syntax:

```
{hint: HintName=Value}
```

The table below identifies the HintNames that you can use with a document subscription.

Hint	Description
`IncludeDeliver`	When set to `true`, the filter applies to documents that are delivered to the client and documents that are delivered to the subscription queue. By default, filters are only applied to documents that are delivered to the subscription queue.
`LocalOnly`	When set to `true`, the filter is applied only to documents that originate from the webMethods Broker to which the Integration Server is connected. Documents originating from a different webMethods Broker are discarded.
`DeadLetterOnly`	When set to `true`, a deadletter subscription is created for the document type specified in the webMethods messaging trigger. The webMethods messaging trigger that subscribes to this hint receives messages that do not have subscribers. To learn more about detecting deadletters, see the *webMethods Broker Java Client API Reference*.

Keep the following points in mind when you add hints to filters:

- Hints must be added at the end of the filter string in the **Filter** field.
- Hints must be in the following format:

  ```
  {hint: HintName=Value}
  ```

  For example, the following filter will match only those documents that originate from the webMethods Broker to which the Integration Server is connected and the value of *city* is equal to `Fairfax`.

  ```
  %city% L_EQUALS "Fairfax" {hint:LocalOnly=true}
  ```

- A filter can also contain a combination of subscription hints. For example, the following filter will match only those documents that do not have a subscriber and that originate from the webMethods Broker to which the Integration Server is connected.

  ```
  {hint:DeadLetterOnly=true} {hint:LocalOnly=true}
  ```

Detecting Deadletters with Hints

A deadletter is an unclaimed published document. If there are no subscribers for a document that is published, the webMethods Broker returns an acknowledgement to the publisher and then discards the document. If, however, a deadletter subscription exists for the document, the webMethods Broker deposits the document in the queue containing the deadletter subscription.

A deadletter subscription allows you to trap unclaimed documents. Detecting and trapping deadletters is a valuable way to quickly identify and resolve discrepancies between a published document and the filtering criteria specified by a subscriber.

You create a deadletter subscription by inserting the `DeadLetterOnly` hint to the subscription filter. For more information about creating deadletter subscriptions using the `DeadLetterOnly` hint, see "Using Hints in Filters" on page 88.

When using the `DeadLetterOnly` hint, keep the following points in mind:

- If the `DeadLetterOnly` hint is used in a filter that contains other expressions, the other expressions are ignored by the webMethods Broker. Consider the following example:

  ```
  %city% L_EQUALS "Fairfax" {hint:DeadLetterOnly=true} {hint:LocalOnly=true}
  ```

  webMethods Broker will ignore the expression `%city% L_EQUALS "Fairfax"` in the filter and will trap only those documents that do not have a subscriber and that originate from the webMethods Broker to which the Integration Server is connected.

- If the filter is registered on Integration Server but not on the webMethods Broker, the `DeadLetterOnly` subscription traps only the documents that are rejected by Integration Server. The filter does not trap the documents rejected by the webMethods Broker unless the filter is registered on the webMethods messaging trigger.

- Both the `LocalOnly` and `IncludeDeliver` hints are implied.

Note:
If you are using Universal Messaging you can configure a dead events store. For more information, see the Universal Messaging documentation.

Using Multiple Conditions in a webMethods Messaging Trigger

You can build webMethods messaging triggers that can contain more than one condition. Each condition can associate one or more documents with a service. You can use the same service or different services for each condition. You can create only one join condition in a webMethods messaging trigger, but a webMethods messaging trigger can contain any number of simple conditions.

When a webMethods messaging trigger receives a document, Integration Server determines which service to invoke by evaluating the webMethods messaging trigger conditions. Integration Server evaluates the webMethods messaging trigger conditions in the same order in which the conditions appear in the editor. It is possible that a document could satisfy more than one condition in a webMethods messaging trigger. However, Integration Server executes only the service associated with the first satisfied condition and ignores the remaining conditions. Therefore, the order in which you list conditions is important.

When you build a webMethods messaging trigger with multiple conditions, each condition can specify the same service. However, you should avoid creating conditions that specify the same publishable document type. If the conditions in a webMethods messaging trigger specify the same publishable document type, Integration Server always executes the condition that appears first. For example, if a webMethods messaging trigger contained the following conditions:

Condition Name	Service	Document Types
ConditionAB	serviceAB	documentA or documentB
ConditionA	serviceA	documentA

Integration Server will never execute serviceA. Whenever Integration Server receives documentA, the document satisfies ConditionAB, and Integration Server executes serviceAB.

Using Multiple Conditions for Ordered Service Execution

You might create a webMethods messaging trigger with multiple conditions to handle a group of published documents that must be processed in a specific order. For each condition, associate one publishable document type with a service. Place your conditions in the order in which you want the services to execute. In the **Processing mode** property, specify serial document processing so that the webMethods messaging trigger will process the documents one at a time, in the order in which they are received. The serial dispatching ensures that the services that process the documents do not execute at the same time. (This assumes that the documents are published and therefore received in the proper order.)

Note:
Using multiple conditions to achieve ordered service execution is only supported for webMethods messaging triggers that receive messages from the webMethods Broker. webMethods Broker is deprecated.

You might want to use multiple conditions to control the service execution when a service that processes a document depends on another service successfully executing. 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 webMethods messaging trigger that contains one condition for each expected publishable document type. You might create a webMethods messaging trigger with the following conditions:

Condition Name	Service	Document Types
Condition1	addCustomer	customerName
Condition2	addCustomerOrder	customerOrder
Condition3	billCustomer	customerBill

If you create one webMethods messaging trigger for each of these conditions, you could not guarantee that the Integration Server would invoke services in the required order even if publishing occurred in that order. Specifying serial dispatching for the webMethods messaging trigger ensures that a service will finish executing before the next document is processed. For example, Integration Server could still be executing addCustomer, when it receives the documents customerOrder and customerBill. If you specified concurrent dispatching instead of serial dispatching, the Integration Server might execute the services addCustomerOrder and billCustomer before it finished executing addCustomer. In that case, the addCustomerOrder and billCustomer services would fail.

Important:
An ordered scenario assumes that documents are published in the correct order and that you set up the webMethods messaging trigger to process documents serially. For more information about specifying the document processing for a webMethods messaging trigger, see "Selecting Message Processing" on page 107.

Ordering Conditions in a webMethods Messaging Trigger

The order in which you list conditions in the editor is important because it indicates the order in which the Integration Server evaluates the conditions at run time. When the Integration Server receives a document, it invokes the service specified in the first condition that is satisfied by the document. The remaining conditions are ignored.

To change the order of conditions in a webMethods messaging trigger

1. In the Package Navigator view of Designer, open the webMethods messaging trigger.
2. Under **Conditions**, select the condition to be moved.
3. Click or to move the condition up or down.
4. Click **File > Save** to save the webMethods messaging trigger.

Disabling and Enabling a webMethods Messaging Trigger

You can use the **Enabled** property to disable or enable a webMethods messaging trigger.

- When you disable a webMethods messaging trigger that receives messages from webMethods Broker, Integration Server disconnects the trigger client on the webMethods Broker. The webMethods Broker removes the document subscriptions created by the trigger client. The webMethods Broker does not place published documents in client queues for disabled webMethods messaging triggers. When you enable a disabled webMethods messaging trigger, Integration Server connects the trigger client to the webMethods Broker and re-establishes the document subscriptions on the webMethods Broker.

  Note:
  You can also suspend document retrieval and document processing for a webMethods messaging trigger. Unlike disabling a webMethods messaging trigger, suspending retrieval

and processing does not destroy the webMethods Broker client queue. The webMethods Broker continues to enqueue documents for suspended webMethods messaging triggers. However, Integration Server does not retrieve or process documents for suspended webMethods messaging triggers. For more information about suspending webMethods messaging triggers, see *webMethods Integration Server Administrator's Guide*.

- When you disable a webMethods messaging trigger that receives messages from Universal Messaging, the corresponding durable subscription remains intact. Universal Messaging enqueues any messages to which the trigger subscribes. The trigger will receive the messages when it is re-enabled.

You cannot disable a webMethods messaging trigger during trigger service execution.

To disable or enable a webMethods messaging trigger

1. In the Package Navigator view of Designer, open the webMethods messaging trigger that you want to disable or enable.

2. In the Properties view, under **General**, set **Enabled** to one of the following:

Select...	To...
False	Disable the webMethods messaging trigger.
True	Enable the webMethods messaging trigger.

3. Click **File > Save**.

   You can enable only valid webMethods messaging triggers. If the webMethods messaging trigger is not valid, Designer displays an error message when you save the webMethods messaging trigger prompting you to cancel the save or continue the save with a disabled webMethods messaging trigger. For more information about requirements for a valid webMethods messaging trigger, see " webMethods Messaging Trigger Requirements" on page 77.

Disabling and Enabling a webMethods Messaging Trigger in a Cluster or Non-Clustered Group

When a webMethods messaging trigger exists on multiple Integration Servers in a cluster or in a non-clustered group, the subscriptions created by the webMethods messaging trigger remain active even if you disable the webMethods messaging trigger from one of the Integration Servers. This is because the client created for the webMethods messaging trigger on the webMethods Broker is shared. The client on the webMethods Broker becomes disconnected when you disable the webMethods messaging trigger on all the servers in the cluster or non-clustered group of servers. Even when the shared webMethods messaging trigger client becomes disconnected, the subscriptions established by the webMethods messaging trigger remain active. The webMethods Broker continues to enqueue documents for the webMethods messaging trigger. When you

re-enable the webMethods messaging trigger on any server in the cluster or non-clustered group, all the queued documents that did not expire will be processed.

To disable a webMethods messaging trigger in a cluster or non-clustered group of Integration Servers, disable the webMethods messaging trigger on each Integration Server, and then manually remove the document subscriptions created by the webMethods messaging trigger from the webMethods Broker.

About Join Time-Outs

When you create a join condition using an All (AND) join or an Only one (XOR), you need to specify a join time-out. A *join time-out* specifies how long Integration Server waits for the other documents in the join condition. Integration Server uses the join time-out period to avoid deadlock situations (such as waiting for a document that never arrives) and to avoid duplicate service invocation.

How the join time-out affects document processing by the webMethods messaging trigger is different for each join type.

- For an All (AND) join, the join time-out determines how long Integration Server waits to receive an instance of each publishable document type in the condition.
- For an Only one (XOR) join, the join time-out determines how long Integration Server discards instances of publishable document types in the condition after it receives an instance document of one of the publishable document types.
- An Any (OR) join condition does not need a join time-out. Integration Server treats an Any (OR) join condition like a webMethods messaging trigger with multiple simple conditions that all use the same trigger service.

Join Time-Outs for All (AND) Join Conditions

A join time-out for an All (AND) join condition specifies how long the Integration Server waits for all of the documents specified in the join condition. When Integration Server pulls a document from the webMethods messaging trigger queue, it determines which condition the document satisfies. If the document satisfies an All (AND) join condition, the Integration Server starts the join time-out period and moves the document from the webMethods messaging trigger queue to the ISInternal database. Integration Server assigns the document a status of "pending." Integration Server then waits for the remaining documents in the join condition. Only documents with the same activation ID as the first received document will satisfy the join condition.

If Integration Server receives all of the documents specified in the join condition (and processes the documents from the trigger queue) before the time-out period elapses, it executes the service specified in the condition. If Integration Server does not receive all of the documents before the time-out period elapses, Integration Server removes the pending documents from the database and generates a journal log message.

When the time-out period elapses, the next document in the webMethods messaging trigger queue that satisfies the All (AND) condition causes the time-out period to start again. Integration Server places the document in the database and assigns a status of "pending" even if the document has

the same activation ID as an earlier document that satisfied the join condition. Integration Server then waits for the remaining documents in the join condition.

Join Time-Outs for Only One (XOR) Join Conditions

A join time-out for an Only one (XOR) join condition specifies how long Integration Server discards instances of the other documents in the condition. When Integration Server pulls the document from the webMethods messaging trigger queue, it determines which condition the document satisfies. If that condition is an Only one (XOR) condition, the Integration Server executes the service specified in the condition. When it pulls the document from the webMethods messaging trigger queue, Integration Server starts the time-out period. For the duration of the time-out period, Integration Server discards any documents of the type specified in the join condition. Integration Server discards only those documents with same activation ID as the first document.

When the time-out period elapses, the next document in the webMethods messaging trigger queue that satisfies the Only one (XOR) condition causes the trigger service to execute and the time-out period to start again. Integration Server executes the service even if the document has the same activation ID as an earlier document that satisfied the join condition. Integration Server generates a journal log message when the time-out period elapses for an Only one (XOR) condition.

Setting a Join Time-Out

When configuring webMethods messaging trigger properties, you can specify whether a join condition times out and if it does, what the time-out period should be. The time-out period indicates how long Integration Server waits for additional documents after receiving the first document specified in the join condition.

To set a join time-out

1. In the Package Navigator view of Designer, open the webMethods messaging trigger that you want to set the join time-out.

2. In the Properties view, under **General**, set **Join expires** to one of the following:

Select...	To...
True	Indicate that Integration Server stops waiting for the other documents in the join condition once the time-out period elapses. In the **Expire after** property, specify the length of the join time-out period. The default time period is 1 day.
False	Indicate that the join condition does not expire. Integration Server waits indefinitely for the additional documents specified in the join condition. Set the **Join expires** property to **False** only if you are confident that all of the documents will be received.

Select...	To...
	Important: A join condition is persisted across server restarts. To remove a waiting join condition that does not expire, disable, then re-enable and save the webMethods messaging trigger. Re-enabling the webMethods messaging trigger effectively recreates the webMethods messaging trigger.

3. Click **File > Save** to save the webMethods messaging trigger.

About Priority Message Processing

Priority messaging determines the order in which a webMethods messaging trigger receives and subsequently processes the documents from the messaging provider. How priority messaging works depends on the messaging provider from which the trigger receives documents.

When priority messaging is enabled for a webMethods messaging trigger that receives documents from webMethods Broker, webMethods Broker places documents in the trigger client queue using the following criteria:

- **Message priority.** webMethods Broker inserts documents with the highest priority at the top of the client queue. A value of 0 is the lowest processing priority; a value of 9 indicates expedited processing.
- **Message publication time.** webMethods Broker orders documents with the same priority in the client queue according to the time at which the document was published. Within documents of the same priority, webMethods Broker inserts the most recently published document after the earlier documents. This ensures that the webMethods messaging trigger receives and processes the documents with the same priority in publication order.

Integration Server receives and processes the documents following the order in which the documents appear in the client queue on the webMethods Broker.

When priority messaging is enabled for a webMethods messaging trigger that receives documents from a Universal Messaging server, Universal Messaging and Integration Server adapt to expedite processing of higher priority documents over lower priority documents.

Note:
Priority messaging applies only to documents that are routed through the webMethods Broker and Universal Messaging. Priority messaging does not apply to locally published documents.

To use priority messaging, you configure both the publishing side and the subscribing side.

- On the publishing side, set a message priority level in the document envelope. The priority level indicates how quickly the document should be processed once it is published. A value of 0 is the lowest processing priority; a value of 9 indicates expedited processing. The default priority is 4.

- On the subscribing side, enable priority messaging for the webMethods messaging trigger. This is necessary only for triggers that receive documents from webMethods Broker.

  Note:
  All webMethods messaging triggers that receive documents from Universal Messaging receive and process documents in priority fashion. Priority messaging cannot be disabled for webMethods messaging triggers that receive documents from Universal Messaging.

Enabling and Disabling Priority Message Processing for a webMethods Messaging Trigger

When enabling or disabling priority message processing for a webMethods messaging trigger, keep the following points in mind:

- Priority messaging only applies to documents that are published to the webMethods Broker and Universal Messaging. It does not apply to locally published documents.
- All webMethods messaging triggers that receive documents from Universal Messaging receive and process documents in priority fashion. Priority messaging cannot be disabled for webMethods messaging triggers that receive documents from Universal Messaging. That is, Integration Server ignores the value of the **Priority** property if the webMethods messaging trigger subscribes to one or more publishable document types associated with a Universal Messaging connection alias.
- When you enable or disable priority messaging for a webMethods messaging trigger that receives documents from webMethods Broker, Integration Server disconnects the trigger client on the webMethods Broker and recreates the associated webMethods messaging trigger client queue. Any documents that existed in the webMethods messaging trigger client queue before you enabled or disabled priority messaging will be lost. For this reason, you may want to ensure the webMethods messaging trigger queue on Integration Server is empty before enabling or disabling priority messaging.
- Priority messaging may consume webMethods Broker resources and can introduce latency into document processing by webMethods messaging triggers. For more information about how priority messaging may impact performance, refer to the *webMethods Broker Java Client API Reference*.

To enable or disable priority message processing for a webMethods messaging trigger

1. In the Package Navigator view of Designer, open the webMethods messaging trigger for which you want to enable priority processing of documents.

2. In the Properties view, under **General**, set **Priority** enabled to one of the following:

Select...	To...
True	Enable priority processing.

Select...	To...
False	Disable priority processing.

3. Click **File > Save** to save the webMethods messaging trigger.

About Execution Users for webMethods Messaging Triggers

For a webMethods messaging trigger that receives documents from Universal Messaging, the **Execution user** property indicates which credentials Integration Server should use when invoking services associated with the 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 webMethods messaging 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 webMethods messaging trigger service after receiving a document from Universal Messaging by setting the **Execution user** property for the 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 webMethods messaging trigger receives a message from Universal Messaging, Integration Server uses the credentials for the specified user account to invoke the service specified in the trigger condition.

Note:
For a webMethods messaging trigger that receives locally published messages or messages from the webMethods Broker, Integration Server, uses the user account specified in the **Run Trigger Service As User** property on the **Settings > Resources > Store Settings** page in Integration Server Administrator. For more information about the **Run Trigger Service As User** property, see *webMethods Integration Server Administrator's Guide*.

Assigning an Execution User to a webMethods Messaging Trigger

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

Note:
The **Execution user** property only applies to webMethods messaging triggers that receive documents from Universal Messaging. The publishable document type to which a trigger subscribes determine the messaging provider from which the trigger receives documents. The **Execution user** property is display only if a webMethods messaging trigger receives locally published documents or documents published to webMethods Broker.

To assign an execution user for a webMethods messaging trigger

1. In the Package Navigator view of Designer, open the webMethods messaging trigger for which you want to set 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 webMethods messaging 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 Capacity and Refill Level for the webMethods Messaging Trigger Queue

On Integration Server, the webMethods messaging trigger queue contains documents waiting for processing. Integration Server assigns each webMethods messaging trigger a queue. A document remains in the webMethods messaging trigger queue until it is processed by the webMethods messaging trigger.

You can determine the capacity of each webMethods messaging trigger queue. The capacity indicates the maximum number of documents that Integration Server can store for that webMethods messaging trigger.

For a webMethods messaging trigger that receives documents from the webMethods Broker, you can also specify a refill level to indicate when Integration Server should retrieve more documents for the webMethods messaging trigger. The difference between the capacity and the refill level determines up to how many documents the Integration Server retrieves for the webMethods messaging trigger from the webMethods Broker. For example, if you assign the webMethods messaging trigger queue a capacity of 10 and a refill level of 4, the Integration Server initially retrieves 10 documents for the webMethods messaging trigger. When only 4 documents remain to be processed in the webMethods messaging trigger queue, Integration Server retrieves up to 6 more documents. If 6 documents are not available, Integration Server retrieves as many as possible.

For a webMethods messaging trigger that receives documents from the webMethods Broker, the capacity and refill level also determine how frequently Integration Server retrieves documents for the webMethods messaging trigger and the combined size of the retrieved documents, specifically:

- The greater the difference between capacity and refill level, the less frequently Integration Server retrieves documents from the webMethods Broker. However, the combined size of the retrieved documents will be larger.
- The smaller the difference between capacity and refill level, the more frequently Integration Server retrieves documents. However, the combined size of the retrieved documents will be smaller.

Note:

A refill level can be set for webMethods messaging triggers that receive documents from the webMethods Broker only. Refill level does not apply to webMethods messaging triggers that receive documents from Universal Messaging.

For a webMethods messaging trigger that receives messages from Universal Messaging, Integration Server receives documents for the trigger one at a time until the trigger queue is at capacity. After the number of documents in the trigger queue equals the configured capacity, Integration Server stops receiving documents. When the number of documents awaiting processing in the trigger queue is less than the configured capacity, the trigger resumes receiving messages from Universal Messaging.

Guidelines for Setting Capacity and Refill Levels for webMethods Messaging Triggers

When you set values for capacity and refill level, you need to balance the frequency of document retrieval with the combined size of the retrieved documents. Use the following guidelines to set values for capacity and refill level for a webMethods messaging trigger that retrieves messages from webMethods Broker.

- If the webMethods messaging trigger subscribes to small documents, set a high capacity. Then, set refill level to be 30% to 40% of the capacity. Integration Server retrieves documents for this webMethods messaging trigger less frequently, however, the small size of the documents indicates that the combined size of the retrieved documents will be manageable. Additionally, setting the refill level to 30% to 40% ensures that the webMethods messaging trigger queue does not empty before the Integration Server retrieves more documents. This can improve performance for high-volume and high-speed processing.
- If the webMethods messaging trigger subscribes to large documents, set a low capacity. Then, set the refill level to just below slightly less than the capacity. Integration Server retrieves documents more frequently, however, the combined size of the retrieved documents will be manageable and will not overwhelm Integration Server.

Setting Capacity and Refill Level for a webMethods Messaging Trigger

To set trigger queue capacity and refill level

1. In the Package Navigator view of Designer, open the webMethods messaging trigger for which you want to specify the webMethods messaging trigger queue capacity.

2. In the Properties view, under **Trigger queue**, in the **Capacity** property, specify the maximum number of documents that the trigger queue can contain. The default is 10.

3. In the **Refill level** property, specify the number of unprocessed documents that must remain in this webMethods messaging trigger queue before Integration Server retrieves more documents for the queue from the webMethods Broker. The default is 4.

The **Refill level** value must be less than or equal to the **Capacity** value.

Note:
The **Refill level** property applies to webMethods messaging triggers that receive documents from the webMethods Broker only. The **Refill level** property does not apply to webMethods messaging triggers that receive documents from Universal Messaging.

4. Click **File > Save** to save the webMethods messaging trigger.

Notes:

- The server administrator can use the Integration Server Administrator to gradually decrease the capacity and refill levels of all webMethods messaging trigger queues. The server administrator can also use the Integration Server Administrator to change the **Capacity** or **Refill leve**l values for a webMethods messaging trigger. For more information, see *webMethods Integration Server Administrator's Guide*.
- You can specify whether Integration Server should reject locally published documents when the queue for the subscribing webMethods messaging trigger is at maximum capacity. For more information about this feature, see the description for the watt.server.publish.local.rejectOOS parameter in *webMethods Integration Server Administrator's Guide*.

About Document Acknowledgements for a webMethods Messaging Trigger

When a trigger service finishes processing a guaranteed document, Integration Server returns an acknowledgement to the messaging provider. Upon receipt of the acknowledgement, the messaging provider removes its copy of the document from storage. By default, Integration Server returns an acknowledgement for a guaranteed document as soon as it finishes processing the document.

Note: Integration Server returns acknowledgements for guaranteed documents only. Integration Server does not return acknowledgements for volatile documents.

You can increase the number of document acknowledgements returned at one time by changing the value of the **Acknowledgement queue size** property. The acknowledgement queue is a queue that contains pending acknowledgements for guaranteed documents processed by the webMethods messaging trigger. When the acknowledgement queue size is greater than one, a server thread places a document acknowledgement into the acknowledgement queue after it finishes executing the trigger service. Acknowledgements collect in the queue until a background thread returns them as a group to the sending resource.

If the **Acknowledgement queue size** is set to one, acknowledgements will not collect in the acknowledgement queue. Instead, Integration Server returns an acknowledgement to the sending resource immediately after the trigger service finishes executing.

If a resource or connection failure occurs before acknowledgements are sent or processed, the transport redelivers the previously processed, but unacknowledged documents. The number of documents redelivered to a webMethods messaging trigger depends on the size on the number of guaranteed documents that were processed but not acknowledged before failure occurred. If

exactly-once processing is configured for the webMethods messaging trigger, Integration Server detects the redelivered, guaranteed documents as duplicates and discards them without re-processing them. For more information about exactly-once processing, see "Exactly-Once Processing for webMethods Messaging Triggers" on page 120.

Increasing the size of a webMethods messaging trigger's acknowledgement queue can provide the following benefits:

- **Reduces network traffic.** Returning acknowledgements one at a time for each guaranteed document that is processed can result in a high volume of network traffic. Configuring the webMethods messaging trigger so that Integration Server returns several document acknowledgements at once can reduce the amount of network traffic.
- **Increases server thread availability.** If the size of the acknowledgement queue is set to 1 (the default), Integration Server releases the server thread used to process the document only after returning the acknowledgement. If the size of the acknowledgement queue is greater than 1, Integration Server releases the server thread used to process the document immediately after the thread places the acknowledgement into the acknowledgement queue. When acknowledgements collect in the queue, server threads can be returned to the thread pool more quickly.

Setting the Size of the Acknowledgement Queue

To set the size of the acknowledgement queue for a webMethods messaging trigger

1. In the Package Navigator view of Designer, open the webMethods messaging trigger for which you want to specify the size of the acknowledgement queue.
2. In the Properties view, under **Trigger queue**, in the **Acknowledgement queue size** property, specify the maximum number of pending document acknowledgements for the webMethods messaging trigger.

   The value must be greater than zero. The default is 1.
3. Click **File > Save** to save the webMethods messaging trigger.

About Message Processing

Message processing determines the order in which Integration Server processes the documents received by a webMethods messaging trigger.

- In serial processing, Integration Server processes the documents received by a webMethods messaging trigger one after the other, in the order in which the documents were received.
- In concurrent processing, Integration Server processes the documents by a webMethods messaging trigger in parallel.

Serial Processing

In serial processing, Integration Server processes the documents received by a webMethods messaging trigger one after the other. Integration Server retrieves the first document received by the webMethods messaging trigger, determines which condition the document satisfies, and executes the service specified in the webMethods messaging trigger condition. Integration Server waits for the service to finish executing before retrieving the next document received by the webMethods messaging trigger.

For a webMethods messaging trigger with serial processing, Integration Server processes documents in the same order in which the trigger retrieves the documents from the messaging provider. However, Integration Server processes documents for a serial trigger more slowly than it processes documents for a concurrent trigger.

If your webMethods messaging trigger contains multiple conditions to handle a group of published documents that must be processed in a specific order, use serial processing. This is sometimes called ordered service execution. Only triggers that receive messages from webMethods Broker can perform ordered service execution.

When a webMethods messaging trigger receives documents from webMethods Broker, the queue for the serial trigger on the webMethods Broker has a Shared Document Order mode of "Publisher".

When a webMethods messaging trigger receives documents from Universal Messaging, the durable subscription for a trigger with serial processing is a serial durable subscription. That is, in Universal Messaging Enterprise Manager, the durable subscription for the trigger has the **Serial** check box selected.

Serial Processing in a Cluster or Non-Clustered Group of Integration Servers

Serial document processing determines how the messaging provider distributes guaranteed documents to the individual servers within a cluster or non-clustered group. In a cluster or non-clustered group, the individual Integration Servers share the same client. For example, if the messaging provider is the webMethods Broker, the servers act as a single webMethods Broker client and share the same trigger client queues and document subscriptions. With serial processing, servers in a cluster or non-clustered group can process documents from a publisher in the same order in which the documents were published.

Note:
In addition to the term "non-clustered group," the terms "stateless cluster" and "external cluster" are sometimes used to describe the situation in which a group of Integration Servers function in a manner similar to a cluster but are not part of a configured cluster.

For each webMethods messaging trigger, each server in the cluster or non-clustered group maintains a trigger queue in memory. This allows multiple servers to process documents for a single webMethods messaging trigger. The messaging provider manages the distribution of documents to the individual webMethods messaging triggers in the cluster or non-clustered group.

How the messaging provider distributes documents for a serial trigger on the Integration Servers in the cluster or group to ensure that documents from a single publisher are processed in publication order varies:

- webMethods Broker distributes documents so that the Integration Servers in the cluster or non-clustered group process guaranteed documents from a single publisher in the same order in which the documents were published. Multiple Integration Servers can process documents for a single trigger, but only one Integration Server in the cluster or non-clustered group processes documents for a particular publisher. For more information, see "Serial Processing with the webMethods Broker in a Clustered or a Non-Clustered Group of Integration Servers" on page 103
- Universal Messaging distributes the documents to which a particular serial trigger subscribes across all the Integration Servers in a cluster or non-clustered group but allows processing by only one Integration Server in the cluster or group at a time. Because a serial trigger processes only one document at a time across the cluster or group, this distribution approach ensures that documents are processed in the same order in which they were published. For more information, see "Serial Processing with Universal Messaging in a Clustered or a Non-Clustered Group of Integration Servers" on page 105.

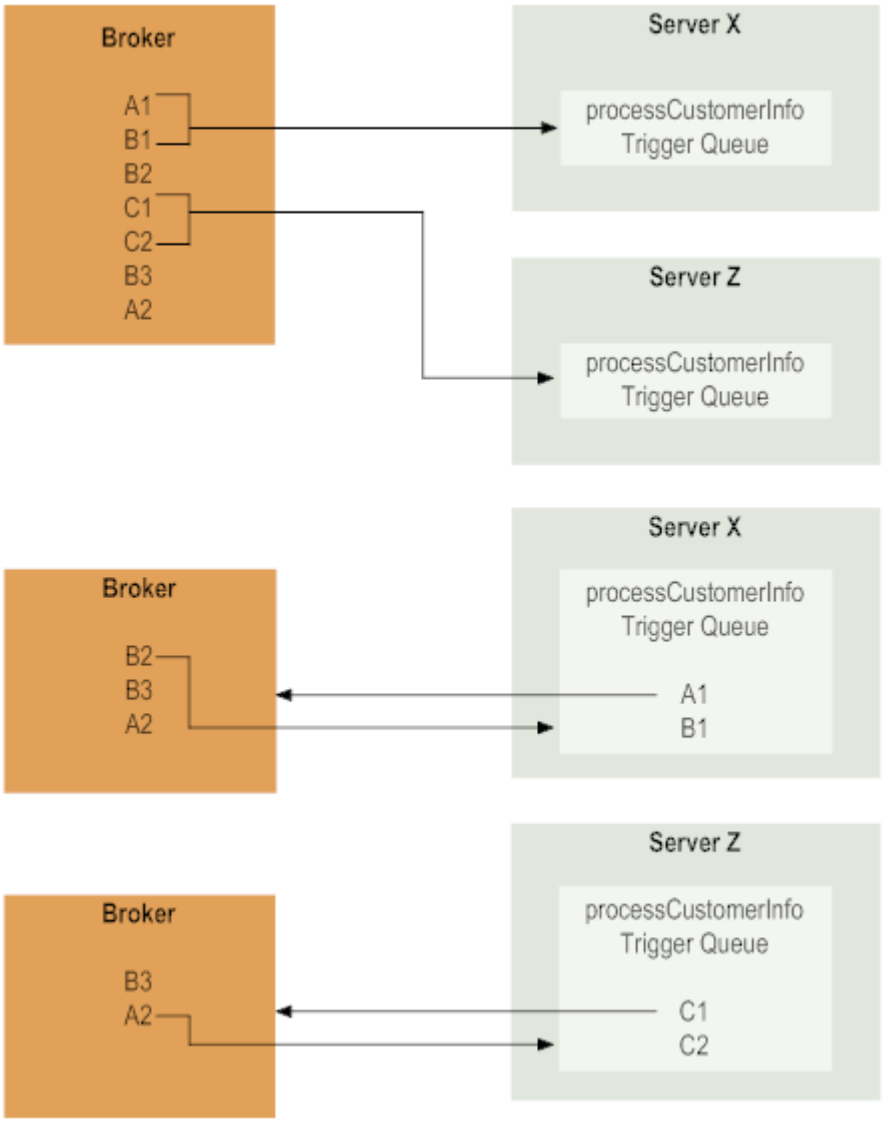
Serial Processing with the webMethods Broker in a Clustered or a Non-Clustered Group of Integration Servers

To ensure that a serial webMethods messaging trigger processes guaranteed documents from individual publishers in publication order, the webMethods Broker distributes documents from one publisher to a single server in a cluster or non-clustered group. The webMethods Broker continues distributing documents from the publisher to the same server as long as the server contains unacknowledged documents from that publisher in the trigger queue. Once the server acknowledges all of the documents from the publisher to the webMethods Broker, other servers in the cluster or non-clustered group can process future documents from the publisher.

For example, suppose that a cluster contains two servers: ServerX and ServerZ. Each of these servers contains the webMethods messaging trigger processCustomerInfo. The processCustomerInfo webMethods messaging trigger specifies serial document processing with a capacity of 2 and a refill level of 1. For each publisher, the cluster must process documents for this webMethods messaging trigger in the publication order. In this example, the processCustomerInfo trigger client queue on the webMethods Broker contains documents from PublisherA, PublisherB, and PublisherC. PublisherA published documents A1 and A2, PublisherB published documents B1, B2, and B3, and PublisherC published documents C1and C2.

The following illustration and explanation describe how serial document processing works in a clustered environment that uses webMethods Broker as the messaging provider.

Serial processing in a cluster of Integration Servers



Step	Description
1	ServerX retrieves the first two documents in the queue (documents A1 and B1) to fill its processCustomerInfo trigger queue to capacity. ServerX begins processing document A1.
2	ServerZ retrieves the documents C1 and C2 to fill its processCustomerInfo trigger queue to capacity. ServerZ begins processing the document C1. Even though document B2 is the next document in the queue, the webMethods Broker does not distribute document B2 from PublisherB to ServerZ because ServerX contains unacknowledged documents from PublisherB.
3	ServerX finishes processing document A1 and acknowledges document A1 to the webMethods Broker.

Step	Description
4	ServerX requests 1 more document from the webMethods Broker. (The processCustomerInfo webMethods messaging trigger has refill level of 1.) The webMethods Broker distributes document B2 from PublisherB to ServerX.
5	ServerZ finishes processing document C1 and acknowledges document C1to the webMethods Broker
6	ServerZ requests 1 more document from the webMethods Broker. The webMethods Broker distributes document A2to ServerZ. ServerZ can process a document from PublisherA because the other server in the cluster (ServerX) does not have any unacknowledged documents from PublisherA. Even though document B3 is the next document in the queue, the webMethods Broker does not distribute document B3to ServerZ because ServerX contains unacknowledged documents from PublisherB.

Notes:

- The webMethods Broker and Integration Servers in a cluster cannot ensure that serial webMethods messaging triggers process volatile documents from the same publisher in the order in which the documents were published.
- When documents are delivered to the default client in a cluster, the webMethods Broker and Integration Servers cannot ensure that documents from the same publisher are processed in publication order. This is because the Integration Server acknowledges documents delivered to the default client as soon as they are retrieved from the webMethods Broker.

Serial Processing with Universal Messaging in a Clustered or a Non-Clustered Group of Integration Servers

To process documents received by a serial trigger in publishing order across a cluster or non-clustered group of Integration Servers , Universal Messaging delivers only one document at a time to an instance of the trigger. When an instance of serial trigger in the cluster or non-clustered group of Integration Servers receives a document, instances of that serial trigger that reside on other Integration Servers will not receive a document until the distributed document is processed and acknowledged. Because the serial trigger processes only one document at a time across the cluster or group, this routing approach ensures that documents from the same publisher are processed in the same order in which the documents were published.

A serial webMethods messaging trigger in Integration Server corresponds to a serial durable subscription on Universal Messaging. In Universal Messaging Enterprise Manager, the durable subscription for the trigger has the **Serial** check box selected. With a serial durable subscription, multiple consumers (triggers) can connect to the durable subscription. However, Universal Messaging delivers one document at a time across all the consumers for the durable subscription. Universal Messaging does not necessarily distribute documents for a serial trigger in a strict round-robin fashion. Rather, Universal Messaging may distribute several documents in a row to one consumer, several more documents in a row to another consumer, and so forth. This approach provides processing in publication order by publisher and also allows documents to be distributed

across the cluster or non-clustered group of Integration Servers in a load-balanced fashion over time.

Note:
If you do not need serial processing of documents by publisher, but you want a trigger to process documents one at a time, select concurrent processing and set **Max execution threads** to 1. This configuration allows the trigger on each Integration Server in the cluster or group to process one document at a time.

Serial Triggers Migrated to Integration Server 10.3 or Later from Earlier Versions

As of Integration Server version 10.3, when using Universal Messaging as the messaging provider, a webMethods messaging trigger with serial processing corresponds to a serial durable subscription on Universal Messaging. For Integration Server version 9.9 to 10.2, a webMethods messaging trigger with serial processing corresponded to a priority durable subscription on Universal Messaging. Prior toIntegration Server 9.9, a webMethods messaging trigger with serial processing corresponded to a shared named object on Universal Messaging.

All serial webMethods messaging triggers created on Integration Server 10.3 or later will correspond to a serial durable subscription. However, serial triggers migrated from Integration Server versions 9.9 to 10.2 will still correspond to a priority named object (durable subscription). Serial triggers migrated from an Integration Server prior to version 9.9 will still correspond to a shared named object (durable subscription). As a result, the serial trigger and the durable subscription will be out of sync. To synchronize the migrated serial trigger and the durable subscription, you must do one of the following:

- If you are using a fresh install of Universal Messaging 10.3 or later (that is, the Universal Messaging server was not migrated), when you start Integration Server, synchronize the publishable document types with the provider using Designer or the built-in service pub.publish:syncToProvider. Synchronizing the publishable document types causes Integration Server to reload the webMethods messaging triggers. Integration Server creates a serial durable subscription for each serial trigger.
- If you are using an installation of Universal Messaging 9.9 or later that was migrated from an earlier version, you must delete and recreate the durable subscription that corresponds to a serial trigger. For more information about deleting and recreating a durable subscription associated with a trigger, see “Synchronizing the webMethods Messaging Trigger and Durable Subscription on Universal Messaging ” on page 110.

Concurrent Processing

In concurrent processing, Integration Server processes the documents received by a webMethods messaging trigger in parallel. Integration Server processes as many documents in the webMethods messaging trigger queue as it can at the same time. Integration Server does not wait for the service specified in the webMethods messaging trigger condition to finish executing before it begins processing the next document in the trigger queue. You can specify the maximum number of documents Integration Server can process concurrently.

Concurrent processing provides faster performance than serial processing. The Integration Server process the documents in the trigger queue more quickly because the Integration Server can process more than one document at a time. However, the more documents Integration Server processes concurrently, the more server threads Integration Server dispatches, and the more memory the document processing consumes.

Additionally, for concurrent webMethods messaging triggers, Integration Server does not guarantee that documents are processed in the order in which they are received.

Concurrent document processing is equivalent to the Shared Document Order mode of "None" on the webMethods Broker.

When receiving messages from Universal Messaging, the Universal Messaging window size limits the number of documents that can be processed at one time by an individual trigger. By default, the window size of a client queue for the trigger is set to the sum of the **Capacity** and **Max execution threads** properties plus 5. For example, if the **Capacity** property is set to 10 and **Max execution threads** is set to 5, the client queue window size is 20. The window size set for a trigger overrides the default value specified in Universal Messaging.

Selecting Message Processing

To select message processing

1. In the Package Navigator view of Designer, open the webMethods messaging trigger for which you want to specify the message processing mode.

2. In the Properties view, under **Message processing**, select one of the following for **Processing mode**:

Select...	To...
Serial	Specify that Integration Server should process documents received by the webMethods messaging trigger one after the other.
Concurrent	Specify that Integration Server should process as many documents received by the webMethods messaging trigger as it can at once.

3. If you selected concurrent processing, in the **Max execution threads** property, specify the maximum number of documents that Integration Server can process concurrently. Integration Server uses one server thread to process each document in the trigger queue.

4. If you selected serial processing and you want Integration Server to suspend document processing and document retrieval automatically when a trigger service ends with an error, under **Fatal error handling**, select **True** for the **Suspend on error** property.

   For more information about fatal error handling, see "Fatal Error Handling for a webMethods Messaging Trigger " on page 111.

5. Click **File > Save** to save the webMethods messaging trigger.

Notes:

- If you selected serial processing, Integration Server creates a serial durable subscription on the channels that correspond to the publishable document types to which the trigger subscribes.
- If you selected concurrent processing, Integration Server creates a shared durable subscription on the channels that correspond to the publishable document types to which the trigger subscribes
- Integration Server Administrator can be used to change the number of concurrent execution threads for a webMethods messaging trigger temporarily or permanently. For more information, see *webMethods Integration Server Administrator's Guide*.

Changing Message Processing When webMethods Broker Is the Messaging Provider

After you perform capacity planning and testing for your integration solution, you might want to modify the processing mode for a webMethods messaging trigger. Keep the following points in mind before you change the processing mode for a webMethods messaging trigger that receives documents from webMethods Broker:

- When you change the processing mode for a webMethods messaging trigger, Integration Server recreates the associated trigger client queue on the webMethods Broker.

  Important:
  Any documents that existed in the trigger client queue before you changed the message process mode will be lost.

- If you created the webMethods messaging trigger on an Integration Server connected to a configured webMethods Broker, you can only change the processing mode if Integration Server is currently connected to the webMethods Broker.
- If you change the document processing mode when Integration Server is not connected to the configured webMethods Broker, Designer displays a message stating that the operation cannot be completed.
- Integration Server does not change the processing mode if the webMethods Broker connection alias shares a client prefix.

  Note:
  A webMethods Broker connection alias shares a client prefix if the **Client Prefix Is Shared** property for the connection alias is set to **Yes**.

Changing Message Processing When Universal Messaging Is the Messaging Provider

You can change the message processing after you create a webMethods messaging trigger. For example, capacity planning might indicate that a concurrent trigger should be changed to serial. Keep the following points in mind when changing the processing mode from serial to concurrent or vice versa for a webMethods messaging trigger that receives documents from Universal Messaging:

- When you change the processing mode for a webMethods messaging trigger that uses a Universal Messaging connection alias that does not share a client prefix, Integration Server deletes and recreates the durable subscription that corresponds to the trigger on Universal Messaging. The trigger and its associated durable subscription on Universal Messaging remain in sync.

  Note:
  A Universal Messaging connection alias does not share a client prefix if the **Client Prefix Is Shared** property for the connection alias is set to **No**.

- When you change the processing mode for a webMethods messaging trigger that uses a Universal Messaging connection alias that shares a client prefix, Integration Server *does not* delete and recreate the durable subscription that corresponds to the trigger on Universal Messaging. As a result, the trigger on Integration Server will be out of sync with the associated durable subscription on Universal Messaging. If the same trigger exists on other Integration Server, such as in a cluster or a non-clustered group of Integration Server, the changed trigger will also be out of sync with the trigger on other Integration Server. This affects document processing. One of the following situations occurs:

  - If you changed the processing mode from serial to concurrent, the corresponding durable subscription on Universal Messaging remains a serial durable subscription. The trigger continues to process documents concurrently. However, if the trigger exists on more than one Integration Server, such as in a cluster or a non-clustered group of Integration Servers, Universal Messaging distributes documents to only one trigger on the clustered or non-clustered group of Integration Servers at a time. Universal Messaging does not distribute documents in a way that allows for concurrent processing across the Integration Servers.

  - If you changed the processing mode from concurrent to serial, the corresponding durable subscription on Universal Messaging remains a shared durable subscription. Integration Server does not change the durable subscription to be a serial durable subscription. Consequently, if the trigger exists on more than one Integration Server, such as in a cluster or a non-clustered group of Integration Servers, Universal Messaging distributes documents to the trigger on each connected Integration Server. Universal Messaging does not distribute documents in a way that ensures that processing order matches publication order.

  For information about how to synchronize the trigger and the durable subscription when the processing mode is out of sync, see "Synchronizing the webMethods Messaging Trigger and Durable Subscription on Universal Messaging " on page 110.

Note:
A Universal Messaging connection alias shares a client prefix if the **Client Prefix Is Shared** property for the connection alias is set to **Yes**.

- Software AG does not recommend changing the processing mode for a trigger when more than one Integration Server connects to the same durable subscription that corresponds to the trigger. For example, if the trigger is on an Integration Server that is part of a cluster or a non-clustered group, more than one Integration Server can share the same durable subscription.

Synchronizing the webMethods Messaging Trigger and Durable Subscription on Universal Messaging

A webMethods messaging trigger and the associated durable subscription on Universal Messaging can get out of sync. For example, when you change the processing mode for a webMethods messaging trigger that uses a Universal Messaging connection alias that shares a client prefix, Integration Server *does not* delete and recreate the durable subscription that corresponds to the trigger on Universal Messaging. As a result, the trigger on Integration Server is out of sync with the durable subscription on Universal Messaging. To synchronize the webMethods messaging trigger and the associated durable subscription, you must delete and recreate the durable subscription.

Note:
A webMethods messaging trigger with a serial processing mode corresponds to a *serial* durable subscription on Universal Messaging. A webMethods messaging trigger with a concurrent processing mode corresponds to a *shared* durable subscription on Universal Messaging. For information about the type of durable subscription on Universal Messaging to which a serial trigger created on versions of Integration Server prior to 10.3 corresponds, see "Serial Triggers Migrated to Integration Server 10.3 or Later from Earlier Versions" on page 106

To synchronize the webMethods messaging trigger and the durable subscription on Universal Messaging

- Do one of the following:
  - If the webMethods messaging trigger resides on the only Integration Server connected to Universal Messaging and the **Client Prefix Is Shared** property for the Universal Messaging connection alias is set to **No**, start the trigger to delete and recreate the corresponding durable subscription. You can start a trigger by disabling and then enabling the Universal Messaging connection alias used by the trigger.

    Note: Integration Server starts triggers upon server restart.

  - If more than one Integration Server connects to Universal Messaging or the **Client Prefix Is Shared** property for the Universal Messaging connection alias is set to **Yes**, you must use Universal Messaging Enterprise Manager to delete the durable subscription. Make sure to delete the durable subscription when the durable subscription is fully drained and no new documents will be sent to it. You many need to quiesce document publishers before deleting the durable subscription. Then create the durable subscription for the trigger by disabling and then enabling the Universal Messaging connection alias used by the trigger.

Fatal Error Handling for a webMethods Messaging Trigger

If a webMethods messaging trigger processes documents serially, you can configure fatal error handling for the webMethods messaging trigger. A fatal error occurs when the trigger service ends because of an exception. You can specify that Integration Server suspend the webMethods messaging trigger automatically if a fatal error occurs during trigger service execution. Specifically, Integration Server suspends document retrieval and document processing for the webMethods messaging trigger if the associated trigger service ends because of an exception.

When Integration Server suspends document processing and document retrieval for a webMethods messaging trigger, Integration Server writes the following message to the journal log:

```
Serial trigger triggerName has been automatically suspended due
to an exception.
```

Document processing and document retrieval remain suspended until one of the following occurs:

- You specifically resume document retrieval or document processing for the webMethods messaging trigger. You can resume document retrieval and document processing using Integration Server Administrator, built-in services (pub.trigger:resumeProcessing or pub.trigger:resumeRetrieval), or by calling methods in the Java API (com.wm.app.b2b.server.dispatcher.trigger.TriggerFacade.setProcessingSuspended() and com.wm.app.b2b.server.dispatcher.trigger.TriggerFacade.setRetrievalSuspended()).
- Integration Server restarts, the webMethods messaging trigger is enabled or disabled (and then re-enabled), the package containing the webMethods messaging trigger reloads. (When Integration Server suspends document retrieval and document processing for a webMethods messaging trigger because of an error, Integration Server considers the change to be temporary. For more information about temporary vs. permanent state changes for webMethods messaging triggers, see *webMethods Integration Server Administrator's Guide*.)

For more information about resuming document processing and document retrieval, see *webMethods Integration Server Administrator's Guide* and the *webMethods Integration Server Built-In Services Reference*.

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

By automatically suspending document retrieval as well, Integration Server prevents the webMethods messaging trigger from retrieving more documents. Because Integration Server already suspended document processing, new documents would just sit in the trigger queue. If Integration Server does not retrieve more documents for the webMethods messaging trigger and Integration Server is in a cluster, the documents might be processed more quickly by another Integration Server in the cluster.

Configuring Fatal Error Handling for a webMethods Messaging Trigger

Keep the following points in mind when configuring fatal error handling for a webMethods messaging trigger.

- You can configure fatal error handling for serial webMethods messaging triggers only.
- Integration Server does not automatically suspend webMethods messaging triggers because of transient errors that occur during trigger service execution. For more information about transient error handling, see "About Transient Error Handling for a webMethods Messaging Trigger " on page 112.

To configure fatal error handling for a webMethods messaging trigger

1. In the Package Navigator view of Designer, open the webMethods messaging trigger for which you want to specify the fatal error handling setting.

2. In the Properties view, under **Fatal error handling**, set the **Suspend on error** property to **True** if you want Integration Server to suspend document processing and document retrieval automatically when a trigger service ends with an error. Otherwise, select **False**. The default is **False**.

3. Click **File > Save** to save the webMethods messaging trigger.

About Transient Error Handling for a webMethods Messaging Trigger

When building a webMethods messaging trigger, you can specify whether or not Integration Server retries a trigger service when the trigger service fails because of a transient error caused by a run-time exception.

A *run-time exception* (specifically, an ISRuntimeException) occurs when the trigger service catches and wraps a transient error and then reissues it as an ISRuntimeException. 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 the Integration Server waits and then re-executes the service.

You can configure transient error handling for a webMethods messaging trigger to instruct Integration Server to wait an specified time interval and then re-execute a trigger service automatically when an ISRuntimeException occurs. Integration Server re-executes the trigger service using the original input document.

When you configure transient error handling for a webMethods messaging trigger, you specify the following retry behavior:

- Whether Integration Server should retry trigger services for the webMethods messaging trigger. Keep in mind that a trigger service can retry only if it is coded to throw ISRuntimeExceptions.
- 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 still fails because of an ISRuntimeException.

You can also configure Integration Server and/or a webMethods messaging 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 webMethods messaging trigger 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 147.

Service Requirements for Retrying a Trigger Service for a webMethods Messaging Trigger

To be eligible for retry, the trigger service must do one of the following to catch a transient error and reissue it as an ISRuntimeException:

- If the trigger service 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 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.

If a transient error occurs and the trigger service does not use pub.flow:throwExceptionForRetry or ISRuntimeException() to catch the error and throw an ISRuntimeException, the trigger service ends in error. Integration Server will not retry the trigger 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 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.

For more information about the EXIT step, see *webMethods Service Development Help*.

Handling Retry Failure

Retry failure occurs when Integration Server makes the maximum number of retry attempts and the trigger service 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 webMethods messaging trigger.

- **Throw exception.** When Integration Server exhausts the maximum number of retry attempts, Integration Server treats the last trigger service 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 webMethods messaging trigger and then retries the trigger service at a later time.

Overview of Throw Exception for Retry Failure

Throwing an exception when retry failure occurs allows the webMethods messaging trigger to continue with document processing when retry failure occurs for a trigger service. You can configure audit logging in such a way that you can use webMethods Monitor to submit the document at a later time (ideally, after the condition that caused the transient error has been remedied).

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 fails because of an ISRuntimeException.
2	Integration Server treats the last trigger service failure as a service exception.
3	Integration Server rejects the document. If the document is guaranteed, Integration Server returns an acknowledgement to the webMethods Broker. If a trigger service generates audit data on error and includes a copy of the input pipeline in the service log, you can use webMethods Monitor to re-invoke the trigger service manually at a later time. Note that when you use webMethods Monitor to process the document, it is processed out of order. That is, the document is not processed in the same order in which it was received (or published) because the document was acknowledged to its transport when the retry failure occurred.
4	Integration Server processes the next document in the webMethods messaging trigger queue.

Overview of Suspend and Retry Later for Retry Failure

Suspending a webMethods messaging trigger and retrying the trigger service later when retry failure occurs provides a way to resubmit the document programmatically. It also prevents the webMethods messaging trigger from retrieving and processing other documents until the cause of the transient error condition has been remedied. This preserves the publishing order, which can be especially important for serial webMethods messaging triggers.

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 fails because of an ISRuntimeException.
2	Integration Server suspends document processing and document retrieval for the webMethods messaging trigger temporarily. The webMethods messaging 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 documents for the webMethods messaging trigger. **Note:** The change to the webMethods messaging trigger state is temporary. Document retrieval and document processing will resume for the webMethods messaging trigger if Integration Server restarts, the webMethods messaging trigger is enabled or disabled, or the package containing the webMethods messaging trigger reloads. You can also resume document retrieval and document processing manually using Integration Server Administrator or by invoking the pub.trigger:resumeRetrieval and pub.trigger:resumeProcessing public services.
3	Integration Server rolls back the document to the webMethods messaging trigger document store. This indicates that the required resources are not ready to process the document and makes the document available for processing at a later time. For serial webMethods messaging triggers, it also ensures that the document maintains its position at the top of webMethods messaging trigger queue. **Note:** When the watt.server.dispatcher.messageStore.redeliverOriginalMessage parameter is set to true,Integration Server stores and resubmits the original message after retry failure. If the parameter is set to false, Integration Server stores the message as it is at that point in trigger service execution. If the trigger service modified the message, Integration Server stores the modified message and uses that as input for subsequent trigger service execution. The default value of the watt.server.dispatcher.messageStore.redeliverOriginalMessage parameter is false.
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 resumes document retrieval and document processing for the webMethods messaging 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.trigger.monitoringInterval property.
6	After Integration Server resumes the webMethods messaging trigger, Integration Server passes the document to the webMethods messaging trigger. The webMethods messaging trigger and trigger service process the document just as they would any document in the trigger queue. **Note:** At this point, the retry count is set to 0 (zero).

Configuring Transient Error Handling for a webMethods Messaging Trigger

The transient error handling behavior that you specify for a webMethods messaging trigger determines how Integration Server handles transient errors that occur 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 147.

To configure transient error handling for a webMethods messaging trigger

1. In the Package Navigator view of Designer, open the webMethods messaging trigger for which you want to configure retry behavior.

2. In the Properties view, under **Transient error handling**, select one of the following for **Retry until** property:

Select...	To...
Max attempts reached	Specify that Integration Server retries the trigger service a limited number of times.

Select...	To...
	In the **Max retry attempts** property, enter the maximum number of times Integration Server should attempt to re-execute the trigger service. The default is 0 retries.
Successful	Specify that the Integration Server retries the trigger service until the service executes to completion. **Note:** If a webMethods messaging trigger is configured to retry until successful and a transient error condition is never remedied, a trigger service enters into an infinite retry situation in which it continually re-executes the service at the specified retry interval. Because you cannot disable a webMethods messaging trigger during trigger service execution and you cannot shut down the server during trigger service execution, an infinite retry situation can cause the Integration Server to become unresponsive to a shutdown request. For information about escaping an infinite retry loop, see "About Retrying Trigger Services and Shutdown Requests" on page 118.

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	Indicate that Integration Server throws a service exception when the last allowed retry attempt ends because of an ISRuntimeException. This is the default. For more information about the **Throw exception** option, see "Overview of Throw Exception for Retry Failure" on page 114.
Suspend and retry later	Indicate that Integration Server suspends the webMethods messaging trigger when the last allowed retry attempt ends because of an ISRuntimeException. Integration Server retries the trigger service at a later time. For more information about the **Suspend and retry later** option, see "Overview of Suspend and Retry Later for Retry Failure" on page 114. **Note:** If you want Integration Server to suspend the webMethods messaging trigger and retry it later, you must provide a resource monitoring service that Integration Server can execute to determine when to resume the webMethods messaging trigger. For more information about

Select...	To...
	building a resource monitoring service, see *Publish-Subscribe Developer's Guide*.

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 webMethods messaging triggers can use the same resource monitoring service.

6. Click **File > Save**.

Notes:

- webMethods messaging triggers and services can both be configured to retry. When a webMethods messaging trigger invokes a service (that is, the service functions as a trigger service), the Integration Server uses the webMethods messaging trigger retry properties instead of the service retry properties.
- 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 service 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 service log entry for the service will have a status of "Failed" and will display the actual error message. This occurs regardless of which retry failure option the webMethods messaging 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.

- If you do not configure service retry for a webMethods messaging trigger, set the **Max retry attempts** property to 0. This can improve the performance of services invoked by the webMethods messaging trigger.
- You can invoke the pub.flow:getRetryCount service within a trigger service to determine the current number of retry attempts made by the 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*.

About Retrying Trigger Services and Shutdown Requests

While Integration Server retries a trigger service, Integration Server ignores requests to shut down the server until the trigger service executes successfully or all retry attempts are made. This allows Integration Server to process a document to completion before shutting down.

Sometimes, however, you might want Integration Server to shut down without completing all retries for trigger services. Integration Server provides a server parameter that you can use to

indicate that a request to shut down the Integration Server should interrupt the retry process for trigger services. The `watt.server.trigger.interruptRetryOnShutdown` parameter can be set to one of the following:

Set to...	To...
`false`	Indicate that Integration Server should not interrupt the trigger service retry process to respond to a shutdown request. The Integration Server shuts down only after it makes all the retry attempts or the trigger service executes successfully. This is the default value. **Important:** If watt.server.trigger.interruptRetryOnShutdown is set to "false" and a webMethods messaging trigger is set to retry until successful, a trigger service can enter into an infinite retry situation. If the transient error condition that causes the retry is not resolved, Integration Server continually re-executes the service at the specified retry interval. Because you cannot disable a webMethods messaging trigger during trigger service execution and you cannot shut down the server during trigger service execution, an infinite retry situation can cause Integration Server to become unresponsive to a shutdown request. To escape an infinite retry situation, set the watt.server.trigger.interruptRetryOnShutdown to "true". The change takes effect immediately.
`true`	Indicate that Integration Server should interrupt the trigger service retry process if a shutdown request occurs. Specifically, after the shutdown request occurs, Integration Server waits for the current service retry to complete. If the trigger service needs to be retried again (the service ends because of an ISRuntimeException), the Integration Server stops the retry process and shuts down. Upon restart, the transport (the webMethods Broker or, for a local publish, the transient store) redelivers the document to the webMethods messaging trigger for processing. **Note:** If the trigger service retry process is interrupted and the transport redelivers the document to the webMethods messaging trigger, the transport increases the redelivery count for the document. If the webMethods messaging trigger is configured to detect duplicates but does not use a document history database or a document resolver service to perform duplicate detection, Integration Server considers the redelivered document to be "In Doubt" and will not process the document. For more information about duplicate detection and exactly-once processing, see "Exactly-Once Processing for webMethods Messaging Triggers" on page 120.

Note:
When you change the value of the `watt.server.trigger.interruptRetryOnShutdown` parameter, the change takes effect immediately.

Exactly-Once Processing for webMethods Messaging Triggers

Within Integration Server, exactly-once processing is a facility that ensures one-time processing of a guaranteed document by a webMethods messaging trigger. The webMethods messaging trigger does not process duplicates of the document.

Integration Server provides exactly-once processing for documents received by a webMethods messaging trigger when all of the following are true:

- The document is guaranteed.
- Exactly once properties are configured for the webMethods messaging trigger.

Duplicate Detection Methods for a webMethods Messaging Trigger

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 document is a copy of one previously processed by the webMethods messaging trigger. Duplicate documents can be introduced in to the webMethods system when:

- The publishing client publishes the same document more than once.
- During publishing or retrieval of guaranteed documents, the sending resource loses connectivity to the destination resource before receiving a positive acknowledgement for the document. The sending resource will redeliver the document when the connection is restored.

Note:
Exactly-once processing and duplicate detection are performed for guaranteed documents only.

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

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

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

- **Redelivery count** indicates how many times the transport has redelivered the document to the webMethods messaging trigger.
- **Document history database** maintains a record of all guaranteed documents processed by webMethods messaging triggers for which exactly-once processing is configured.

- **Document resolver service** is a service created by a user to determine the document status. The document resolver service can be used instead of or in addition to the document history database.

The steps that Integration Server performs to determine a document's status depend on the exactly-once properties configured for the subscribing trigger. For more information about configuring exactly-once properties, see "Configuring Exactly-Once Processing for a webMethods Messaging Trigger " on page 121.

Note:
For detailed information about exactly-once processing for webMethods messaging triggers, see *Publish-Subscribe Developer's Guide*.

Configuring Exactly-Once Processing for a webMethods Messaging Trigger

Configure exactly-once processing for a webMethods messaging trigger when you want the webMethods messaging trigger to process guaranteed documents once and only once. If it is acceptable for a trigger service to process duplicates of a document, you should not configure exactly-once processing for the webMethods messaging trigger.

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

- Integration Server can perform exactly-once processing for guaranteed documents only.
- 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 *Publish-Subscribe Developer's Guide*. To minimize these exceptions, it is recommended that you use a history database and a document resolver service.
- If Integration Server connects to an 6.0 or 6.0.1 version of the webMethods Broker, you must use a document history database and/or a document resolver service to perform duplicate detection. Earlier versions of the webMethods Broker do not maintain a redelivery count. Integration Server will assign documents received from these webMethods Brokers a redelivery count of -1. If you do not enable another method of duplicate detection, Integration Server assigns the document a New status and executes the trigger service.
- Stand-alone Integration Servers cannot share a document history database. Only a cluster or a non-clustered group of Integration Servers can 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 documents but does not cause the document history database to consume too many server resources. If external applications reliably publish documents once, you might use a smaller duplicate detection window. If the external applications are prone to publishing duplicate documents, 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*.

To configure exactly-once processing for a webMethods messaging trigger

1. In the Package Navigator view of Designer, open the webMethods messaging 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 webMethods messaging 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 webMethods Messaging Trigger

If you later determine that exactly-once processing is not necessary for a webMethods messaging trigger, you can disable it. When you disable exactly-once processing, the Integration Server provides at-least-once processing for all guaranteed documents received by the webMethods messaging trigger.

To disable exactly-once processing for a webMethods messaging trigger

1. In the Package Navigator view of Designer, open the webMethods messaging 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**.

Modifying a webMethods Messaging Trigger

After you create a webMethods messaging trigger, you can modify it by changing or renaming the condition, specifying different publishable document types, specifying different trigger services, or changing webMethods messaging trigger properties. To modify a webMethods messaging trigger, you need to lock the webMethods messaging trigger and have write access to the webMethods messaging trigger.

If your integration solution includes a messaging provider (webMethods Broker or Universal Messaging), the messaging provider needs to be available when editing a webMethods messaging trigger. Editing a webMethods messaging trigger when the messaging provider is unavailable can cause the webMethods messaging trigger to become out of sync with the associated object on the provider. Do not edit any of the following webMethods messaging trigger components when the messaging provider used by the publishable document types in the trigger are not available.

- Any publishable document types specified in the webMethods messaging trigger. That is, do not change the subscriptions established by the webMethods messaging trigger.
- Any filters specified in the webMethods messaging trigger.
- The webMethods messaging trigger state (enabled or disabled).
- The document processing mode (serial or concurrent processing).
- Priority messaging state (enabled or disabled). This applies to webMethods messaging triggers that receive documents from the webMethods Broker only.

If you edit any of these webMethods messaging trigger components when messaging provider is unavailable and save the changes, the webMethods messaging trigger will become out of sync with its associated object on the messaging provider. You will need to synchronize the webMethods messaging trigger with its associated provider object when the messaging provider becomes available. To synchronize, use Designer to disable the webMethods messaging trigger, re-enable the webMethods messaging trigger, and save. This effectively recreates the object on the messaging provider that is associated with the webMethods messaging trigger. Alternatively, you can disable and then enable the messaging connection alias used by the trigger. However, this restarts all the triggers that use the messaging connection alias and may consume more time and resources.

Note:
If you changed the message processing mode for a webMethods messaging trigger that uses a Universal Messaging connection alias with a shared client prefix, you might need to use Universal Messaging Enterprise Manager to delete and recreate the durable subscription. For more information, see "Synchronizing the webMethods Messaging Trigger and Durable Subscription on Universal Messaging " on page 110.

Modifying a webMethods Messaging Trigger in a Cluster or Non-Clustered Group

Once you set up a cluster or a non-clustered group of Integration Servers, avoid editing any of the webMethods messaging triggers in the cluster or non-clustered group.

Important:
Modifying triggers on an Integration Server in a cluster or in a non-clustered group of servers can cause the triggers to be out of sync with the other servers in the cluster or non-clustered group. It can also create inconsistencies with the associated object on the messaging provider.

In a clustered environment, you can modify selected webMethods messaging trigger properties (capacity, refill level, maximum and execution threads) using the Integration Server Administrator. For more information about editing webMethods messaging trigger properties using the Integration Server Administrator, see *webMethods Integration Server Administrator's Guide*.

Deleting webMethods Messaging Triggers

Keep the following points in mind when deleting a webMethods messaging trigger:

- When you delete a webMethods messaging trigger, Integration Server deletes the queue for the webMethods messaging trigger on Integration Server.
- If the messaging connection alias used by the publishable document type to which the trigger subscribes does not share a client prefix, deleting the webMethods messaging trigger causes the messaging provider to delete the associated provider object (webMethods Broker client queue or Universal Messaging durable subscription on a channel).
- If the messaging connection alias used by the publishable document type to which the trigger subscribes shares a client prefix, deleting the webMethods messaging trigger does not cause the messaging provider to delete the associated provider object (webMethods Broker client queue or Universal Messaging durable subscription on a channel).
- To delete a webMethods messaging trigger, you must lock it and have write access to it.
- You can also use the pub.trigger:deleteTrigger service to delete a webMethods messaging trigger. For more information about this service, see the *webMethods Integration Server Built-In Services Reference*.

To delete a webMethods messaging trigger

1. In the Package Navigator view of Designer, select the webMethods messaging trigger that you want to delete.
2. Click **Edit > Delete**.
3. In the Delete Confirmation dialog box, click **OK**.

Deleting webMethods Messaging Triggers in a Cluster or Non-Clustered Group

When a webMethods messaging trigger exists on multiple Integration Servers in a cluster or non-clustered group of Integration Servers, the subscriptions created by the webMethods messaging trigger remain active even if you delete the webMethods messaging trigger from one of the

Integration Servers. When deleting webMethods messaging triggers from the servers in a cluster or non-clustered group, the associated provider object remains connected to the cluster or non-clustered group until you delete the webMethods messaging trigger on all of the servers. If you do not delete the webMethods messaging trigger on all of the servers, the provider object for the webMethods messaging trigger remains connected and the messaging provider continues to enqueue documents for the webMethods messaging trigger.

To delete a webMethods messaging trigger from a cluster or non-clustered group of Integration Servers, delete the webMethods messaging trigger from each Integration Server in the cluster, and then manually delete the provider object associated with webMethods messaging trigger from the messaging provider.

Note:
In addition to the term "non-clustered group," the terms "stateless cluster" and "external cluster" are sometimes used to describe the situation in which a group of Integration Servers function in a manner similar to a cluster but are not part of a configured cluster.

Running a webMethods Messaging Trigger with a Launch Configuration

In Designer, you can run a webMethods messaging trigger to verify that the subscription, filters, and trigger service work as expected. Designer requires launch configurations to run webMethods messaging triggers. However, if a webMethods messaging trigger does not have an associated launch configuration and you bypass the Run Configurations dialog boxes when running the webMethods messaging trigger, Designer creates a launch configuration on the fly and saves it in your workspace. You can use this configuration from one session to the next. In fact, Designer reuses this configuration every time you run the webMethods messaging trigger without creating another launch configuration.

By default, Designer saves launch configurations locally in an unexposed location in your workspace. However, you might want to share launch configurations with other developers. You can specify that Designer save a launch configuration to a shared file within your workspace; this location will be exposed. On the **Common** tab in the Run Configurations dialog box, select the **Shared file** option and provide a workspace location in which to save the file.

In a launch configuration for a webMethods messaging trigger, you specify:

- The condition that you want Designer to test. Each launch configuration can specify only one condition in the webMethods messaging trigger.
- The document type whose subscription you want to test. For an Any (OR) or Only one (XOR) join condition, you specify the document type for which you want to supply input.
- Input data that Designer uses to build a document. Designer evaluates the filter using the data in the document and provides the document as input to the trigger service.

You can create multiple launch configurations for each webMethods messaging trigger.

Creating a Launch Configuration for a webMethods Messaging Trigger

Use the following procedure to create a launch configuration for running a webMethods messaging trigger.

To create a launch configuration for running a webMethods messaging trigger

1. In the Service Development perspective, select **Run > Run Configurations**
2. In the **Run Configuration** dialog box, select **webMethods Messaging Trigger** and click to add a new launch configuration.
3. In the **Name** field, specify a name for the launch configuration.
4. On the **webMethods Messaging Trigger** tab, in the **Integration Server** list, select the Integration Server on which the webMethods messaging trigger for which you are creating a launch configuration resides.
5. In the **webMethods Messaging Trigger** field click **Browse** to navigate to and select the trigger.
6. On the **Input** tab, in the **Condition** list, select the condition that you want to test using the launch configuration.
7. If the condition is a join condition with an Any (OR) or Only one (XOR) join, do the following:
   a. Next to **Document Type**, click **Select**.
   b. In the Select a Document Type dialog box, select the document type for which you want to provide input data in this launch configuration.
      - For an Any (OR) join, select one document type.
      - For an Only one (XOR) join, select the document or document types that you want to use to test the join condition.
   c. Click **OK**.
8. On the **Input** tab, select the tab with the name of the IS document type for which you want to provide input data.

   If the selected condition uses an All (AND) join, Designer displays one tab for each document type in the join condition. If the condition is an Only one (XOR) join and you selected multiple document types for which to supply input data, Designer displays one tab for each selected document type.

a. Select or clear the **Include empty values for String Types** check box to indicate how to handle variables that have no value.

   - If you want to use an empty String (i.e., a String with a zero-length), select the **Include empty values for String Types** check box. Also note that Document Lists that have defined elements will be part of the input, but they will be empty.
   - If you want to use a null value for the empty Strings, clear the check box. String-type variables will not included in the input document.

   Note:
   The setting applies to all String-type variables in the root document of the input signature. The setting does not apply to String-type variables within Document Lists. You define how you want to handle String-type variables within Document Lists separately when you assign values to Document Lists variables.

b. Specify the values to save with the launch configuration for the webMethods messaging trigger by doing one of the following:

   - Type the input value for each field in the document type.
   - To load the input values from a file, click **Load** to locate and select the file containing the input values. If Designer cannot parse the input data, it displays an error message indicating why it cannot load the data.

   Designer validates the provided input values. If provided values do not match the input parameter data type, Designer displays a message to that effect. You cannot use the launch configuration for the webMethods messaging trigger if the provided input does not match the defined data type.

9. Repeat the preceding step for each IS document type displayed on the **Input** tab.

10. If you want to save the input values that you have entered, click **Save**.

11. Click **Apply**.

12. If you want to execute the launch configuration, click **Run**. Otherwise, click **Close**.

Running a webMethods Messaging Trigger

Keep the following points in mind when running a webMethods messaging trigger:

- When you run a webMethods messaging trigger, you can select the launch configuration that Designer uses to run the webMethods messaging trigger. If a launch configuration does not exist for a webMethods messaging trigger, Designer creates a launch configuration and immediately prompts you for input values and then runs the webMethods messaging trigger. Designer saves the launch configuration in your workspace.
- When you run a webMethods messaging trigger, you can only test one condition at a time.

- When you run a webMethods messaging trigger with a join condition Designer treats the activation IDs for the documents as identical. Designer ignores the value of the *activation* field in the document envelope.
- When you run a webMethods messaging trigger by running a launch configuration, the webMethods messaging trigger is tested locally. That is, a document is not routed through the messaging provider.

  Note:
  To test a webMethods messaging trigger by publishing a document to the messaging provider, test a publishable document type. You test a publishable document type by creating and running a launch configuration for the publishable document type.
- Designer displays results for running the webMethods messaging trigger in the Results view.

To run a webMethods messaging trigger

1. In Package Navigator view of the Service Development perspective, select the webMethods messaging trigger you want to run.
2. Select **Run > Run As > webMethods Messaging Trigger**
3. If multiple launch configurations exist for the service, use the **Select Launch Configuration** dialog box to select the launch configuration that you want Designer to use to run the webMethods messaging trigger.
4. If the launch configuration is set up to prompt the user for input values or there is no launch configuration, in the **Enter Input for *triggerName*** dialog box, in the **Condition** list, select the condition that you want to test using the launch configuration.
5. If the condition is a join condition with an Any (OR) or Only one (XOR) join, do the following:
   a. Next to **Document type**, click **Select**.
   b. In the Select a Document Type dialog box, select the document type for which you want to provide input data.
      - For an Any (OR) join, select one document type.
      - For an Only one (XOR) join, select one or more document types to use to test the join condition. Note that at run time, the trigger service processes only one of the documents. The trigger discards the other document.
   c. Click **OK**.
6. In the **Enter Input for *triggerName*** dialog box, select the tab with the name of the IS document type for which you want to provide input data.

7. Select or clear the **Include empty values for String Types** check box to indicate how to handle variables that have no value.

   - If you want to use an empty String (i.e., a String with a zero-length), select the **Include empty values for String Types** check box. Also note that Document Lists that have defined elements will be part of the input, but they will be empty.
   - If you want to use a null value for the empty Strings, clear the check box. String-type variables will not included in the input document.

   Note:
   The setting applies to all String-type variables in the root document of the input signature. The setting does not apply to String-type variables within Document Lists. You define how you want to handle String-type variables within Document Lists separately when you assign values to Document Lists variables. For more information, see *webMethods Service Development Help*.

8. Specify the values to save with the launch configuration for the webMethods messaging trigger by doing one of the following:

   - Type the input value for each field in the document type.
   - To load the input values from a file, click **Load** to locate and select the file containing the input values. If Designer cannot parse the input data, it displays an error message indicating why it cannot load the data.

   Note:
   If you type in input values, Designer discards the values you specified after the run. If you want to save input values, create a launch configuration. For instructions, see "Running a webMethods Messaging Trigger " on page 127.

9. Click **OK**.

   Designer runs the trigger and displays the results in the Results view.

Testing Join Conditions

While running a launch configuration for a webMethods messaging trigger provides verification of filters and the trigger service execution, it does not test all aspects of a join condition. For example, running a webMethods messaging trigger does not test the join expiration. In addition to running a launch configuration to test a join condition, consider testing the join condition in the following ways:

- Publish documents from Designer using a launch configuration.

  You can publish documents by creating and running a launch configuration for a publishable document type.

  To test a join condition by publishing documents via a launch configuration, you must use the same activation ID for all the documents specified in the join. If you re-use an activation ID

from one test to the next, make sure that the documents sent in the first test are processed before starting the next test.

- Create a service that publishes the documents.

  You can also test join processing for a join condition by creating a flow service that invokes a publish service for each of the document types specified in the join condition. Integration Server automatically assigns an activation ID and uses that activation ID for all the documents published in the same service.

During trigger processing and join processing, Integration Server writes messages to the journal log. You can use the contents of the journal log to test and debug the join conditions in the webMethods messaging trigger.

Debugging a webMethods Messaging Trigger

To debug and test a webMethods messaging trigger, you can:

- Run the webMethods messaging trigger using a launch configuration. You can use the launch configuration to verify that the subscription, filters, and trigger service work as expected.
- Publish a document to which the webMethods messaging trigger subscribes. You can do this by creating a launch configuration for a publishable document type and then running the launch configuration. Alternatively, you can create a service that uses one of the pub.publish* services to publish a document to which the trigger subscribes and then run the service to publish the document.
- Instruct Integration Server to produce an extra level of verbose logging. You can enable debug trace logging for all webMethods messaging triggers or an individual webMethods messaging trigger.

  Note: Integration Server generates additional logging for triggers that receive messages from Universal Messaging only or through Digital Event Services.

Enabling Trace Logging for All webMethods Messaging Triggers

You can configure Integration Server to generate additional debug and trace logging for all the webMethods messaging triggers that receive messages from Universal Messaging or from Digital Event Services.

Note:
For the increased logging to appear in the server log, you must set the logging level for server facility 0153 Dispatcher (Universal Messaging) to Trace.

To enable debug trace logging for all webMethods messaging triggers

1. Open Integration Server Administrator if it is not already open.
2. Go to **Settings > Extended**.

3. Click **Edit Extended Settings**.

4. Under **Extended Settings**, type the following:

```
watt.server.messaging.debugTrace=true
```

5. Click **Save Changes**.

6. Disable and then enable the messaging connection aliases used by the webMethods messaging triggers.

Enabling Trace Logging for a Specific webMethods Messaging Trigger

You can configure Integration Server to generate additional debug and trace logging for a specific webMethods messaging trigger that receives messages from Universal Messaging or Digital Event Services.

Note:
For the increased logging to appear in the server log, you must set the logging level for server facility 0153 Dispatcher (Universal Messaging) to Trace.

To enable debug trace logging for a specific webMethods messaging trigger

1. Open Integration Server Administrator if it is not already open.

2. Go to **Settings > Extended**.

3. Click **Edit Extended Settings**.

4. Under **Extended Settings**, type the following:

```
watt.server.messaging.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.

7 Exactly-Once Processing for Documents Received by webMethods Messaging Triggers

- Introduction 134
- What Is Document Processing? 134
- Overview of Exactly-Once Processing 134
- Extenuating Circumstances for Exactly-Once Processing 142
- Exactly-Once Processing and Performance 144
- Configuring Exactly-Once Processing 144
- Building a Document Resolver Service 144
- Viewing Exactly-Once Processing Statistics 145
- Clearing Exactly-Once Processing Statistics 146

Introduction

This chapter explains what exactly-once processing with regards to webMethods messaging trigger, how Integration Server performs exactly-once processing, and how to configure exactly-once processing for a webMethods messaging trigger.

What Is Document Processing?

Within the publish-and-subscribe model, *document processing* is the process of evaluating documents against trigger conditions and executing the appropriate trigger services to act on those documents. The processing used by Integration Server depends on the document storage type and the trigger settings. Integration Server offers three types of document processing.

- **At-least-once processing** indicates that a trigger processes a document one or more times. The trigger might process duplicates of the document. Integration Server provides at-least-once processing for guaranteed documents.
- **At-most-once processing** indicates that a trigger processes a document once or not at all. Once the trigger receives the document, processing is attempted but not guaranteed. Integration Server provides at-most-once processing for volatile documents (which are neither redelivered nor acknowledged). Integration Server might process multiple instances of a volatile document, but only if the document was published more than once.
- **Exactly-once processing** indicates that a trigger processes a document once and only once. The trigger does not process duplicates of the document. Integration Server provides exactly-once processing for guaranteed documents received by triggers for which exactly-once properties are configured.

At-least-once processing and exactly-once processing are types of guaranteed processing. In guaranteed processing, Integration Server ensures that the trigger processes the document once it arrives in the trigger queue. Integration Server provides guaranteed processing for documents with a guaranteed storage type.

Note:
Guaranteed document delivery and guaranteed document processing are not the same thing. *Guaranteed document delivery* ensures that a document, once published, is delivered at least once to the subscribing triggers. *Guaranteed document processing* ensures that a trigger makes one or more attempts to process the document once the trigger receives the document.

The following section provides more information about how Integration Server ensures exactly-once processing.

Overview of Exactly-Once Processing

Within Integration Server, exactly-once processing is a facility that ensures one-time processing of a guaranteed document received by a trigger. Integration Server provides exactly-once processing by performing duplicate detection and by supplying the ability to retry trigger services.

Duplicate detection determines whether the current document is a copy of one previously processed by the trigger. Duplicate documents can be introduced in to the webMethods system when:

- The publishing client publishes the same document more than once.
- During publishing or retrieval of guaranteed documents, the sending resource loses connectivity to the destination resource before receiving a positive acknowledgement for the document. The sending resource will redeliver the document when the connection is restored.

Note:
Exactly-once processing and duplicate detection are performed for guaranteed documents only.

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

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

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

- **Redelivery count.** Indicates how many times the transport has redelivered the document to the trigger.
- **Document history database.** Maintains a record of all guaranteed documents processed by triggers for which exactly-once processing is configured.
- **Document resolver service.** Service created by a user to determine the document status. The document resolver service can be used instead of or in addition to the document history database.

The steps that Integration Server performs to determine a document's status depend on the exactly-once properties configured for the subscribing trigger. For more information about configuring exactly-once properties, see "Configuring Exactly-Once Processing" on page 144.

The table below summarizes the process thatIntegration Server follows to determine a document's status and the action the server takes for each duplicate detection method.

Step 1 Check Redelivery Count

When the trigger is configured to detect duplicates, Integration Server will check the document's redelivery count to determine if the trigger processed the document before.

Redelivery Count	Action
0	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 document to be NEW. Integration Server executes the trigger service.
> 0	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 document to be IN DOUBT.
-1 (Undefined)	If using document history, proceed to Step 2 to check the document history database. If document history is not used, proceed to Step 3 to execute the document resolver service. Otherwise, document is NEW. Execute 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 documents, Integration Server checks for the document's UUID in the document history database.

UUID Exists?	Action
No.	Document is NEW. Execute trigger service.
Yes. Processing completed.	Document is a DUPLICATE. Acknowledge document and discard.
Yes. Processing started.	If provided, proceed to Step 3 to invoke the document resolver service. Otherwise, document is IN DOUBT.

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	Execute trigger service.
DUPLICATE	Acknowledge document and discard.
IN DOUBT	Acknowledge and log document.

Note: Integration Server sends In Doubt documents to the audit subsystem for logging. If the messaging provider is Broker or the document is published locally, you can resubmit In Doubt documents using webMethods Monitor. Integration Server discards Duplicate documents.

Duplicate documents cannot be resubmitted. For more information about webMethods Monitor, see the webMethods Monitor documentation.

The following sections provide more information about each method of duplicate detection.

Redelivery Count

The redelivery count indicates the number of times the transport (the Universal Messaging server, Broker or, for local publishing, the transient store) has redelivered a document to the trigger. The transport that delivers the document to the trigger maintains the document redelivery count. The transport updates the redelivery count immediately after the trigger receives the document. A redelivery count other than zero indicates that the trigger might have received and processed (or partially processed) the document before.

For example, suppose that your integration solution consists of an Integration Server and a Broker. When the server first retrieves the document for the trigger, the document redelivery count is zero. After the server retrieves the document, the Broker increments the redelivery count to 1. If a resource (Broker or Integration Server) shuts down before the trigger processes and acknowledges the document, the Broker will redeliver the document when the connection is re-established. The redelivery count of 1 indicates that the Broker delivered the document to the trigger once before.

The following table identifies the possible redelivery count values and the document status associated with each value.

A redelivery count of...	Indicates...
-1	The resource that delivered the document does not maintain a document redelivery count. The redelivery count is undefined. Integration Server uses a value of -1 to indicate that the redelivery count is absent. For example: ■ A document received from a Broker version 6.0 or 6.0.1 does not contain a redelivery count. (Brokers version 6.0.1 and earlier do not maintain document redelivery counts.) ■ Any document received via a serial durable subscription on a Universal Messaging server does not have a redelivery count. Integration Server assigns the document a redelivery count of -1. 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 document status. If no other methods of duplicate detection are configured, Integration Server assigns the document a status of New and executes the trigger service.
0	This is most likely the first time the trigger received the document. If the trigger uses a document history to perform duplicate detection, Integration Server checks the document history database to determine the document status. If no other methods of duplicate detection are

A redelivery count of...	Indicates...
	configured, the server assigns the document a status of New and executes the trigger service.
> 0	The number of times the resource redelivered the document to the trigger. The trigger might or might not have processed the document before. For example, the server might have shut down before or during processing. Or, the connection between Integration Server and messaging provider was lost before the server could acknowledge the document. The redelivery count does not provide enough information to determine whether the trigger processed the document 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 document status. If no other methods of duplicate detection are configured, the server assigns the document a status of In Doubt, acknowledges the document, uses the audit subsystem to log the document, and writes a journal log entry stating that an In Doubt document was received.

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

Note:
You can retrieve a redelivery count for a document at any point during trigger service execution by invoking the pub.publish:getRedeliveryCount service. For more information about this service, see the section *pub.publish:getRedeliveryCount* in the *webMethods Integration Server Built-In Services Reference*. the section *pub.publish:getRedeliveryCount* in the *webMethods Integration Server Built-In Services Reference*.

Document History Database

The document history database maintains a history of the guaranteed documents processed by 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 document processing information only for triggers for which the **Use history** property is set to true.

The database saves the following information about each document:

- **Trigger ID.** Universally unique identifier for the trigger processing the document.
- **Document UUID.** Universally unique identifier for the document. The publisher is responsible for generating and assigning this number.

Note: Integration Server automatically assigns a UUID to all the documents that it publishes. However, if a service publishes a document to Universal Messaging using pub.publish:publish, you can specify a value for *uuid* that Integration Server uses instead of generating one.

- **Processing Status.** Indicates whether the trigger service executed to completion or is still processing the document. 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 document. This allows Integration Server to remove both entries for a specific document at the same time.

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

If the UUID...	Then Integration Server...
Does not exist.	Assigns the document a status of New and executes the trigger service. The absence of the UUID indicates that the trigger has not processed the document before.
Exists in a "processing" entry and a "completed" entry.	Assigns the document a status of Duplicate. The existence of the "processing" and "completed" entries for the document's UUID indicate the trigger processed the document successfully already. Integration Server acknowledges the document, discards it, and writes a journal log entry indicating that a duplicate document was received.
Exists in a "processing" entry only.	Cannot determine the status of the document conclusively. The absence of an entry with a "completed" status for the UUID indicates that the trigger service started to process the document, 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, Integration Server invokes it. If a document resolver service is not specified for this trigger, Integration Server assigns the document a status of In Doubt, acknowledges the document, uses the audit subsystem to log the document, and writes a journal log entry stating that an In Doubt document was received.
Exists in a "completed" entry only	Determines the document is a Duplicate. The existence of the "completed" entry indicates the trigger processed the document successfully already. Integration Server acknowledges the document, discards it, and writes a journal log entry indicating that a Duplicate document was received.

Note: Integration Server also considers a document to be In Doubt when the document's UUID (or, in the absence of a UUID the value of *trackID* or *eventID*) exceeds 96 characters. Integration

Server then uses the document resolver service, if provided, to determine the status of the document. For more information about how Integration Server handles a document missing a UUID, see "Documents without UUIDs" on page 140.

For information about configuring the document history database, refer to *Installing Software AG Products*.

What Happens When the Document History Database Is Not Available?

If the connection to the document history database is down 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 handles the 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 147.

Documents without UUIDs

The UUID is the universally unique identifier that distinguishes a document from other documents. The publisher is responsible for assigning a UUID to a document. However, some publishing clients might not assign a UUID to a document. For example, the 6.0.1 version of Integration Server does not assign a UUID when publishing a document.

Integration Server requires the UUID to create and find entries in the document history database. Therefore, if the server receives a document that does not have a UUID, it creates a UUID using one of the following values from the document envelope:

- If the *trackID* field contains a value, the server uses the *trackID* value as the UUID.
- If the *trackID* field is empty, the server uses the *eventID* as the UUID.

The maximum length of the UUID field is 96 characters. If the *trackID* (or *eventID*) is greater than 96 characters, the server does not assign a UUID and cannot conclusively determine the document's status. If specified, Integration Server executes the document resolver service to determine the document's status. Otherwise, Integration Server logs the document as In Doubt.

Note: Integration Server versions later than 6.01. ensure that a UUID is always assigned to a document before publishing the document to the messaging provider.

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 per trigger and depends on the value of the trigger's **History time to live** property.

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 the section *Scheduling a User Task* in the *webMethods Integration Server Administrator's Guide*. the section *Scheduling a User Task* in the *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.

To clear expired entries from the document history database

1. Open Integration Server Administrator.
2. Go to **Settings > Resources**.
3. Click **Exactly Once Statistics**.
4. Click **Remove Expired Document History Entries**.

Document Resolver Service

The document resolver service is a service that you build to determine whether a document'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 document status, such as whether or not the transport sent the document previously, the document UUID, the transport used to route the document, and the actual document. The document resolver service must return one of the following for the document status: NEW, DUPLICATE, or IN DOUBT.

By using the redelivery count and the document history database, Integration Server can assign most documents a status of New or Duplicate. However, a small window of time exists where checking the redelivery count and the document history database does not conclusively determine whether a trigger processed a document before. For example:

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

You can write a document resolver service to determine the status of documents received during these windows. How the document resolver service determines the document 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 document to determine the document status conclusively. If processing an earlier copy of the document 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.

For more information about building a document resolver service, see "Building a Document Resolver Service" on page 144.

Document Resolver Service and Exceptions

At run time, a document resolver service might end because of an exception. How Integration Server proceeds depends on the type of exception and the value of the watt.server.trigger.preprocess.suspendAndRetryOnError property.

- If the document resolver service ends with an ISRuntimeException, and the watt.server.trigger.preprocess.suspendAndRetryOnError property is set to true, Integration Server suspends the trigger and schedules a system task to execute the trigger's resource monitoring service (if one is specified). Integration Server resumes 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:resumeProcessing and pub.trigger:resumeRetrieval services). For more information about configuring a resource monitoring service, see "Building a Resource Monitoring Service" on page 195.

- If the document resolver service ends with an ISRuntimeException, and the watt.server.trigger.preprocess.suspendAndRetryOnError property is set to false, Integration Server assigns the document a status of In Doubt, acknowledges the document, and uses the audit subsystem to log the document.

- If the document resolver service ends with an exception other than an ISRuntimeException, Integration Server assigns the document a status of In Doubt, acknowledges the document, and uses the audit subsystem to log the document.

Extenuating Circumstances for Exactly-Once Processing

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

For example, in the following situations a trigger with exactly-once processing configured might process a duplicate document.

- If the client publishes a document twice and assigns a different UUID each time, Integration Server does not detect the second document as a duplicate. Because the documents have different UUIDs, Integration Server processes both documents.

- If the document resolver service incorrectly determines the status of a document to be new (when it is, in fact, a duplicate), the server processes the document a second time.
- If a client publishes a document twice and the second publish occurs after the server removes the expired document UUID entries from the document history table, Integration Server determines the second document is new and processes it. Because the second document arrives after the first document's entries have been removed from the document history database, Integration Server does not detect the second document 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 document. (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 slower Integration Servers processes a document 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 fast Integration Server 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 fast Integration Server removes the entries before 15 minutes have elapsed and a duplicate of the document arrives, the Integration Servers in the cluster will treat the document as a new document.

  Note:
  Time drift occurs when the computers that host the clustered servers gradually develop different date/time values. Even if the Integration Server 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.

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

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

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

Exactly-Once Processing and Performance

Exactly-once processing for a trigger consumes server resources and can introduce latency into document processing by triggers. For example, when Integration Server maintains a history of guaranteed documents processed by a 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 redelivery count cannot conclusively determine a document's status, the server must obtain a database connection from the JDBC pool, traverse the network, and query the database to determine whether the trigger processed the document.

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.

Configuring Exactly-Once Processing

For information about using Designer to configure and disable exactly-once processing for webMethods messaging triggers, see "Exactly-Once Processing for webMethods Messaging Triggers" on page 120.

Building a Document Resolver Service

A document resolver service is a service that you create to perform duplicate detection. Integration Server uses the document resolver service as the final method of duplicate detection.

The document resolver service must do the following:

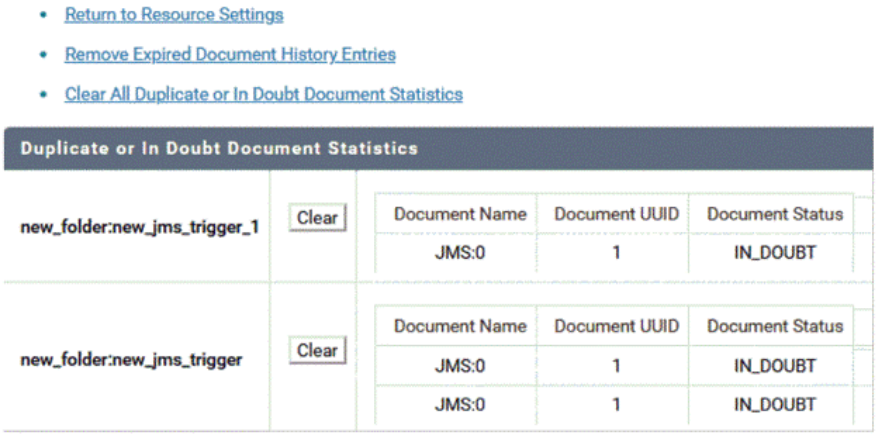
- Use the pub.publish:documentResolverSpec as the service signature. Integration Server passes the document resolver service values for each of the variables declared in the input signature.
- Return a status of NEW, IN DOUBT, or DUPLICATE. Integration Server uses the status to determine whether or not to process the document.
- 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 and Exceptions" on page 142. For information about building services that throw a retry exception, see *webMethods Service Development Help*.

- Determine how far document processing progressed. If necessary, the document resolver service can issue compensating transactions to reverse the effects of a partially completed transaction.

Viewing Exactly-Once Processing Statistics

You can use the Integration Server Administrator to view a history of the In Doubt or Duplicate documents received by triggers. The Integration Server Administrator displays the name, UUID (universally unique identifier), and status for the Duplicate or In Doubt documents received by triggers for which exactly-once processing is configured.

Exactly-Once Statistics



Integration Server saves exactly-once statistics in memory. When the server restarts, the statistics will be removed from memory.

Note:
The exactly-once statistics table might not completely reflect all the duplicate documents received via the following methods: delivery to the default client, local publishing, and from a 6.0.1 Broker. In each of these cases, Integration Server saves documents in a trigger queue located on disk. When a trigger queue is stored on disk, the trigger queue rejects immediately any documents that are copies of documents currently saved in the trigger queue. Integration Server does not perform duplicate detection for these documents. Consequently, the exactly-once statistics table will not list duplicate documents that were rejected by the trigger queue.

To view exactly-once processing statistics

1. Start Integration Server and open the Integration Server Administrator.

2. Go to **Settings > Resources**.

3. Click **Exactly-Once Statistics**.

Clearing Exactly-Once Processing Statistics

1. Start Integration Server and open the Integration Server Administrator.

2. Go to **Settings > Resources**.

3. Click **Exactly-Once Statistics**.

4. Click **Clear All Duplicate or In Doubt Document Statistics**.

8 Transient Error Handling During Trigger Preprocessing

- Server and Trigger Properties that Affect Transient Error Handling During Trigger Preprocessing 148
- Overview of Transient Error Handling During Trigger Preprocessing 149

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.

Note:
The transient error handling for trigger preprocessing applies to webMethods messaging triggers and JMS triggers. It does not apply to MQTT triggers.

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 transacted 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 149.

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 provider, and uses the audit subsystem to log the document. This may result in message loss. **Note:**

Step	Description
	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).

9 How Triggers Correspond to Durable Subscribers on Universal Messaging

When Universal Messaging is used as the provider for webMethods messaging or JMS, triggers on Integration Server may map to a durable on Universal Messaging, specifically:

- When Universal Messaging is used as the webMethods messaging provider, the webMethods messaging triggers map to durable subscribers on Universal Messaging. Integration Server configures this automatically for webMethods messaging triggers.
- When Universal Messaging is the JMS provider, the JMS triggers that receive messages from a topic with a JMS durable subscription do use a durable on Universal Messaging. The JMS triggers receive messages from a channel on Universal Messaging with a durable, either shared or serial.

  For a JMS trigger that uses a durable subscriber to retrieve messages from a topic on Universal Messaging, the type of durable is set on the connection factory. The type of durable is not set on or by Integration Server. You must use Universal Messaging Enterprise Manager to set the durable on the connection factory. Software AG recommends that you set the type of durable based on the processing mode for the JMS trigger.

Note:
JMS triggers that receive messages from a queue do *not* use a durable on Universal Messaging. The triggers receive messages from a queue on Universal Messaging. JMS triggers that receive messages from a topic without a durable subscription do *not* use a durable on Universal Messaging either. The triggers receive messages from a channel on Universal Messaging

Knowing which durable corresponds to a particular type of trigger can help you locate information in the Universal Messaging documentation about the durable. This may provide a more complete picture about the behavior of the trigger and its durable as well as make it easier to troubleshoot issues that may occur.

The following table identifies the durable type used with a specific combination of the trigger type and message processing mode.

Trigger Type	Processing Mode	Durable Type
webMethods messaging trigger	Serial	Serial

Trigger Type	Processing Mode	Durable Type
webMethods messaging trigger	Concurrent	Shared
JMS trigger that receives messages from a topic for which there is a durable subscription	Serial	Serial **Note:** This is the recommended durable type for the serial processing mode. You must set this on the connection factory.
JMS trigger that receives messages from a topic for which there is a durable subscription	Concurrent	Shared **Note:** This is the recommended durable type for the concurrent processing mode. You must set this on the connection factory.

Note:
For a JMS trigger that uses a durable subscriber to retrieve messages from a topic on Universal Messaging, the type of durable is controlled by the connection factory. The type of durable is not set automatically Integration Server. Software AG recommends that you set the durable type on a connection factory to match the processing mode identified in the above table.

10 Understanding Join Conditions

- Introduction 154
- Join Types 154
- Subscribe Path for Documents that Satisfy a Join Condition 155
- Join Conditions in Clusters 161

Introduction

Join conditions are conditions that associate two or more document types with a single trigger service. Typically, join conditions are used to combine data published by different sources and process it with one service.

Join Types

The join type that you specify for a join condition determines whether Integration Server needs to receive all, any, or only one of the documents to execute the trigger service. The following table describes the join types that you can specify for a condition.

Join Type	Description
All (AND)	Integration Server invokes the associated trigger service when the server receives an instance of each specified publishable document type within the join time-out period. The instance documents must have the same activation ID. This is the default join type. For example, suppose that a join condition specifies document types documentA and documentB and documentC. Instances of all the document types must be received to satisfy the join condition. Additionally, all documents must have the same activation ID and must be received before the specified join time-out elapses.
Any (OR)	Integration Server invokes the associated trigger service when it receives an instance of any one of the specified publishable document types. For example, suppose that the join condition specifies document types documentA or documentB or documentC. Only one of these documents is required to satisfy the join condition. Integration Server invokes the associated trigger service every time it receives a document of type documentA, documentB, or documentC. The activation ID does not matter. No time-out is necessary.
Only one (XOR)	Integration Server invokes the associated trigger service when it receives an instance of any of the specified document types. For the duration of the join time-out period, Integration Server discards (blocks) any instances of the specified publishable document types with the same activation ID. For example, suppose that the join condition specifies document types documentA or documentB or documentC. Only one of these documents is required to satisfy the join condition. It does not matter which one. Integration Server invokes the associated trigger service after it receives an instance of one of the specified document types. Integration Server continues to discard instances of any qualified document types with the same activation ID until the specified join time-out elapses. **Tip:**

Join Type	Description
	You can create an **Only one (XOR)** join condition that specifies only one publishable document type. For example, you can create a condition that specified documentA and documentA. This condition indicates that Integration Server should process one and only one documentA with a particular activation ID during the join time-out period. Integration Server discards any other documentA documents with the same activation ID as the first one received.

Subscribe Path for Documents that Satisfy a Join Condition

Integration Server processes documents that satisfy join conditions in almost the same way in which it processes documents for simple conditions. When Integration Server determines that a document satisfies an **All (AND)** join condition or an **Only one (XOR)** join condition, it uses a join manager and the ISInternal database to process and store the individual documents in the join condition.

The following sections provide more information about how Integration Server processes documents for join conditions.

Note: Integration Server processes documents that satisfy an **Any (OR)** condition in the same way that it processes documents that satisfy simple conditions.

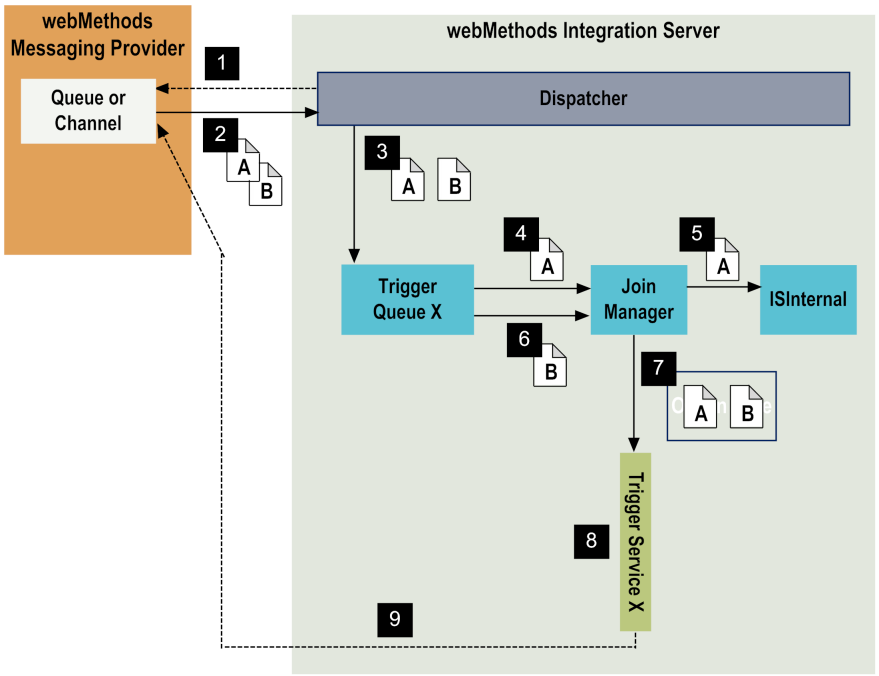
The Subscribe Path for Documents that Satisfy an All (AND) Join Condition

When Integration Server receives a document that satisfies an **All (AND)** join condition, it stores the document and then waits for the remaining documents specified in the join condition. Integration Server invokes the trigger service if each of the following occurs:

- The trigger receives an instance of each document specified in the join condition.
- The documents have the same activation ID.
- The documents arrive within the specified join time-out period.

The following diagram illustrates how Integration Server receives and processes documents for **All (AND)** join conditions. In the following example, trigger X contains an **All (AND)** join condition that specifies that documentA and documentB must be received for the trigger service to execute.

Subscribe path for documents that satisfy an All (AND) join condition



Step	Description
1	Integration Server requests documents for a trigger from the messaging provider.
2	Integration Server receives documents for the trigger, including documentA and documentB. Both documentA and documentB have the same activation ID.
3	Integration Server places documentA and documentB in the trigger's queue on Integration Server.
4	Integration Server pulls documentA from the trigger queue and evaluates the document against the conditions in the trigger. Integration Server determines that documentA partially satisfies an **All (AND)** join condition. Integration Server moves documentA from the trigger queue to the join manager. Integration Server starts the join time-out period. **Note:** If exactly-once processing is configured for the trigger, Integration Server first determines whether the document is a copy of one already processed by the trigger. Integration Server continues processing the document only if the document is new.
5	The join manager saves documentA to the ISInternal database. Integration Server assigns documentA a status of "pending." Integration Server returns an acknowledgement for the document to the messaging provider.
6	Integration Server pulls documentB from the trigger queue and evaluates the document against the conditions in the trigger. Integration Server determines that documentB partially satisfies an **All (AND)** join condition. Integration Server sends documentB from the trigger queue to the join manager.

Step	Description
7	The join manager determines that documentB has the same activation ID as documentA. Because the join time-out period has not elapsed, the **All (AND)** join condition is completed. The join manager delivers a join document containing documentA and documentB to the trigger service specified in the condition.
8	Integration Server executes the trigger service.
9	After the trigger service executes to completion (success or error), one of the following occurs: ■ If the service executes successfully and documentB is guaranteed, Integration Server acknowledges receipt of documentB to the messaging provider. Integration Server then removes the copy of the documentA from the ISInternal database and removes the copy of documentB from the trigger queue. ■ If a service exception occurs, the service ends in error and Integration Server rejects the join document. If documentB is guaranteed, Integration Server acknowledges receipt of documentB to the messaging provider. Integration Server then removes the copy of the documentA from the ISInternal database and removes the copy of documentB from the trigger queue. Integration Server sends an error notification document to the publisher. ■ If the trigger service catches a transient error, wraps it, and re-throws it as an ISRuntimeException, then Integration Server waits for the length of the retry interval and re-executes the service using the original document as input. If Integration Server reaches the maximum number of retries and the trigger service still fails because of a transient error, Integration Server treats the last failure as a service error. For more information about retrying a trigger service, see *webMethods Service Development Help*. **Note:** A transient error is an error that arises from a condition that might correct itself later, such as a network issue or an inability to connect to a database.

Notes:

- If the join time-out period elapses before the other documents specified in the join condition (in this case, documentB) arrive, the ISInternal database drops documentA.
- If documentB had a different activation ID, the join manager would move documentB to the ISInternal database, where it would wait for a documentA with a matching activation ID.
- If documentB arrived after the join time-out period started by the receipt of documentA had elapsed, documentB would not complete the join condition. The ISInternal database would have already discarded documentA when the join time-out period elapsed. The join manager would send documentB to the ISInternal database and wait for another documentA with the same activation ID. Integration Server would restart the join time-out period.
- Integration Server returns acknowledgements for guaranteed documents only.

- If a transient error occurs during document retrieval or storage, the audit subsystem sends the document to the logging database and assigns it a status of FAILED. You can use webMethods Monitor to find and resubmit documents with a FAILED status if the documents were published locally or received from Broker. For more information about using webMethods Monitor, see the webMethods Monitor documentation.
- If a trigger service generates audit data on error and includes a copy of the input pipeline in the service log, you can use webMethods Monitor to re-invoke the trigger service at a later time. For more information about configuring services to generate audit data, see *webMethods Service Development Help*.
- You can configure a trigger to suspend and retry at a later time if retry failure occurs. *Retry failure* occurs when Integration Server makes the maximum number of retry attempts and the trigger service still fails because of an ISRuntimeException. For more information about handling retry failure, see *webMethods Service Development Help*.

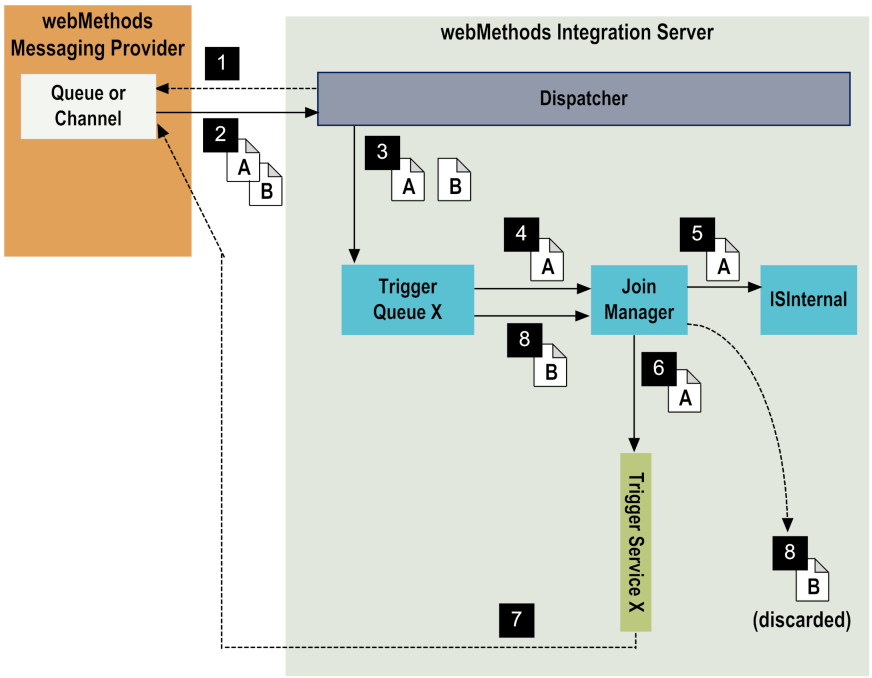
The Subscribe Path for Documents that Satisfy an Only one (XOR) Join Condition

When Integration Server receives a document that satisfies an **Only one (XOR)** condition, it executes the trigger service specified in the join condition. For the duration of the join time-out period, Integration Server discards documents if:

- The documents are of the type specified in the join condition, and
- The documents have the same activation ID as the first document that satisfied the join condition.

The following diagram illustrates how Integration Server receives and processes documents for **Only one (XOR)** join conditions. In the following example, trigger X contains an **Only one (XOR)** join condition that specifies that either documentA or documentB must be received for the trigger service to execute. Integration Server uses whichever document it receives first to execute the service. When the other document specified in the join condition arrives, Integration Server discards it.

Subscribe path for documents that satisfy an Only one (XOR) condition



Step	Description
1	Integration Server requests documents for the trigger from the messaging provider.
2	Integration Server receives documents for the trigger, including documentA and documentB. Both documentA and documentB have the same activation ID.
3	Integration Server places documentA and documentB in the trigger's queue on Integration Server.
4	Integration Server pulls documentA from the trigger queue and evaluates the document against the conditions in the trigger. Integration Server determines that documentA satisfies an **Only one (XOR)** join condition. Integration Server moves documentA from trigger queue to the join manager. Integration Server starts the join time-out period. **Note:** If exactly-once processing is configured for the trigger, Integration Server first determines whether the document is a copy of one already processed by the trigger. Integration Server continues processing the document only if the document is new.
5	The join manager saves the state of the join for this activation in the ISInternal database. The state information includes a status of "complete".
6	Integration Server completes the processing of documentA by executing the trigger service specified in the **Only one (XOR)** condition.
7	After the trigger service executes to completion (success or error), one of the following occurs:

Step	Description
	■ If the service executes successfully, Integration Server returns the server thread to the thread pool. If the documentA is guaranteed, Integration Server returns an acknowledgement to the messaging provider. Integration Server removes the copy of the document from the trigger queue. ■ If a service exception occurs, the service ends in error and Integration Server rejects the document. If documentA is guaranteed, Integration Server returns an acknowledgement to the messaging provider. Integration Server removes the copy of the document from the trigger queue and sends the publisher an error document to indicate that an error has occurred. ■ If the trigger service catches a transient error, wraps it, and re-throws it as an ISRuntimeException, Integration Server waits for the length of the retry interval and re-executes the service. If Integration Server reaches the maximum number of retries and the trigger service still fails because of a transient error, Integration Server treats the last failure as a service error. For more information about retrying a trigger service, see *webMethods Service Development Help*. **Note:** A transient error is an error that arises from a condition that might correct itself later, such as a network issue or an inability to connect to a database.
8	Integration Server pulls documentB from the trigger queue, and evaluates the document against the conditions in the trigger. Integration Server determines that documentB satisfies the **Only one (XOR)** join condition. Integration Server sends documentB from the trigger queue to the join manager.
9	The join manager determines that documentB has the same activation ID as documentA. Because the join time-out period has not elapsed, the Integration Server discards documentB. Integration Server returns an acknowledgement for documentB to the messaging provider.

Notes:

- If documentB had a different activation ID, the join manager would move documentB to the ISInternal database and execute the trigger service specified in the **Only one (XOR)** join condition.
- If documentB arrived after the join time-out period started by the receipt of documentA had elapsed, Integration Server would invoke the trigger service specified in the **Only one (XOR)** join condition and start a new time-out period.
- Integration Server returns acknowledgements for guaranteed documents only.
- If a transient error occurs during document retrieval or storage, the audit subsystem sends the document to the logging database and assigns it a status of FAILED. You can use webMethods Monitor to find and resubmit documents with a FAILED status if the documents were published

locally or received from Broker. For more information about using webMethods Monitor, see the webMethods Monitor documentation.

- If a trigger service generates audit data on error and includes a copy of the input pipeline in the service log, you can use webMethods Monitor to re-invoke the trigger service at a later time. For more information about configuring services to generate audit data, see *webMethods Service Development Help*.
- You can configure a trigger to suspend and retry at a later time if retry failure occurs. *Retry failure* occurs when Integration Server makes the maximum number of retry attempts and the trigger service still fails because of an ISRuntimeException. For more information about handling retry failure, see *webMethods Service Development Help*.

Join Conditions in Clusters

A cluster is treated as an individual Integration Server and acts as such with the exception of a failover. Any Integration Server in a cluster can act as the recipient of a document that fulfills a join condition. If there is more than one document required to fulfill the join, any members of the cluster can receive the documents as long as the documents are received within the allocated time-out period.

A cluster failover occurs if a document that completes a join condition is received by an Integration Server, which then experiences a hardware failure. In such cases, if the document is guaranteed, the messaging provider will redeliver the document to another Integration Server within the cluster and the join condition will be fulfilled.

Each member of a cluster shares the same database for storing the join state.

11 Synchronizing Data Between Multiple Resources

- Data Synchronization Overview 164
- Data Synchronization with webMethods 164
- Tasks to Perform to Set Up Data Synchronization 174
- Defining How a Source Resource Sends Notification of a Data Change 175
- Defining the Structure of the Canonical Document 176
- Setting Up Key Cross-Referencing in the Source Integration Server 177
- Setting Up Key Cross-Referencing in the Target Integration Server 181
- For N-Way Synchronizations Add Echo Suppression to Services 184

Data Synchronization Overview

Often, multiple applications within an enterprise use equivalent data. For example, a Customer Relationship Management (CRM) system and a Billing system both contain information about customers. If the address for a customer changes, the change should be updated in both systems. *Data synchronization* keeps equivalent data across multiple systems consistent; that is, if data is changed in one application, a similar change is made to the equivalent data in all other applications.

Applications can be thought of as resources of data. (The remainder of this chapter will refer to applications as resources.) There are two methods for keeping data synchronized between resources:

- *One-way synchronization*, in which one resource is the source of all data changes. When a data change is required, it is always made in the source. After a change is made, it is propagated to all other resources, which are referred to as the *targets*.

  For example, for data synchronization between a CRM system, Billing system, and Order Management system, the CRM might be the source of all changes and the Billing and Order Management systems receive changes made to the CRM system

- *N-way synchronization*, in which every resource can be a source and a target as well. Changes to data can be made in any resource. After the change is made, it is propagated to all other resources.

  For example, for data synchronization between a CRM system, Billing system, and Order Management system, any of the systems can initiate a change, and then the change is propagated to the other two systems.

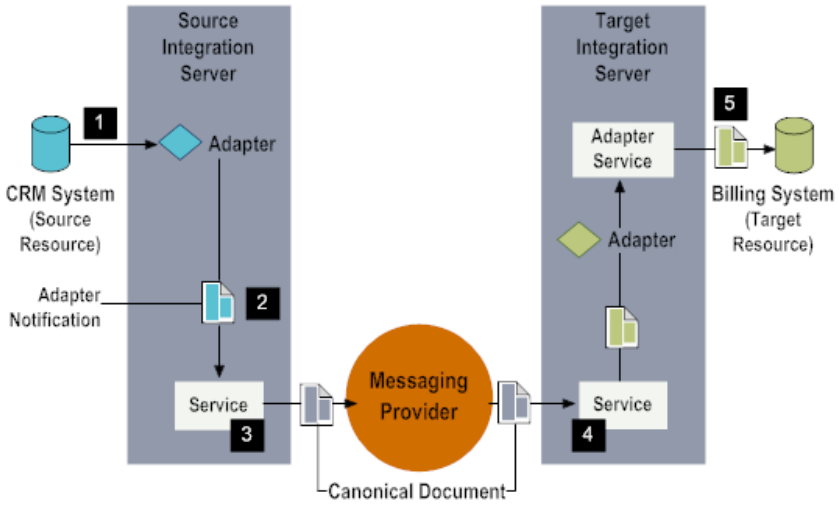
  N-way synchronizations are more complex than one-way synchronizations because multiple applications can change corresponding data concurrently.

Data Synchronization with webMethods

To perform data synchronization with webMethods, when a resource makes a data change, it notifies Integration Server by sending a document that describes the data change. If the source resource is using a webMethods adapter, the adapter can send an adapter notification that describes the data change.

The data from the source is typically in a layout or structure that is native to the source. You set up processing in Integration Server that maps values from the notification document to build a common-structure document, referred to as a *canonical document*. Each target receives the canonical document, which they use to update the equivalent data on their systems.

The following diagram illustrates the basics of performing data synchronization with webMethods software.



Step	Description
1	The source resource makes a data change.
2	Integration Server receives notification of the change on the source. For example, in the above illustration, an adapter checks for changes on the source. When the adapter recognizes a change on the source, it sends an adapter notification that contains information about the change that was made. The adapter might either publish the adapter notification or directly invoke a service passing it the adapter notification. For more information about adapter notifications, see the guide for the specific adapter that you are using with a source resource.
3	A service that you create receives the notification of change on the source. For example, it receives the adapter notification. This service maps data from the change notification document to a canonical document. A *canonical document* is a common business document that contains the information that all target resources will require to incorporate data changes. The canonical document has a neutral structure among the resources. For more information, see "Canonical Documents" on page 166. After forming the canonical document, the service publishes the canonical document. The targets subscribe to the canonical document.
4	On a target, the trigger that subscribes to the canonical document invokes a service that you create. This service maps data from the canonical document into a document that has a structure that is native to the target resource. The target resource uses the information in this document to make the equivalent change.
5	The target resource makes a data change that is equivalent to the change that the source initiated. If the target resource is using an adapter, an adapter service can be used to make the data change. For more information about adapter services, see the guide for the specific adapter that you are using with a target resource.

Equivalent Data and Native IDs

As stated above, data synchronization keeps equivalent data across multiple systems consistent. Equivalent data in resources does not necessarily use the same layout or structure. The data structure is based on the requirements of the resource. The following example shows different data structures for customer data that a CRM system and a Billing system might use:

Structure of customer data in a CRM system

```
Customer ID
Customer Name
       First
       Surname
Customer Address
       Line1
       Line2
       City
       State
       ZipCode
       Country
Customer Payment Information
Customer Account
```

Structure of customer data in a Billing system

```
Account ID
Account Type
Billing Account Owner
       Last
       First
Billing Address
       Number
       Street
       AptNumber
       CityOrTown
       State
       Code
       Country
Billing Preferences
```

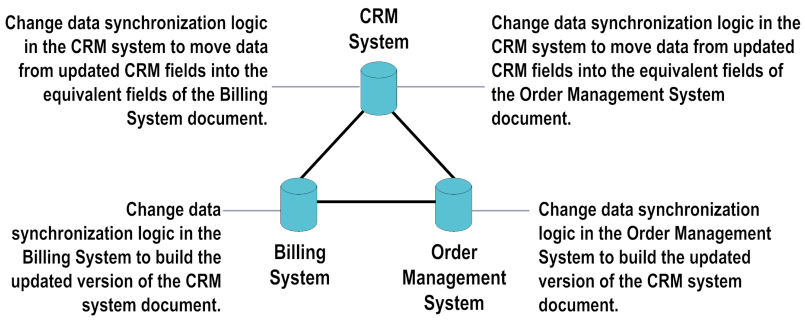
Data in an application contains a *key* value that uniquely identifies an object within the application. In the example above, the key value that uniquely identifies a customer within the CRM system is the Customer ID; similarly, the key value in the Billing system is the Account ID. The key value in a specific application is referred to as that application's *native ID*. In other words, the native ID for the CRM system is the Customer ID, and the native ID for the Billing system is the Account ID.

Canonical Documents

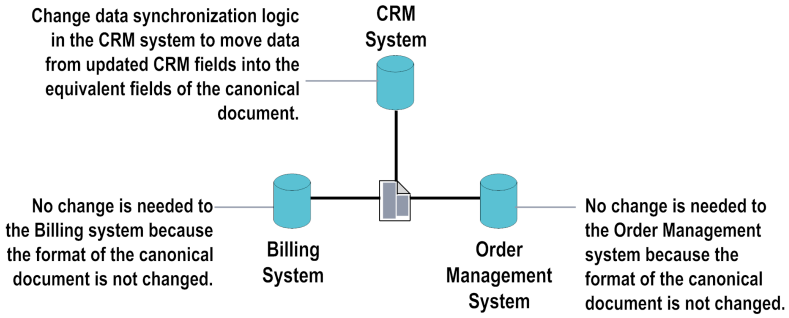
Source resources send documents to notify other resources of data changes. You set up processing to map the data from these notification documents into a canonical document. A canonical document is a document with a neutral structure, and it encompasses all information that resources will require to incorporate data changes. Use of canonical documents:

- **Simplifies data synchronization between multiple resources** by eliminating the need for every resource to understand the document structure of every other resource. Resources need only include logic for mapping to and from the canonical document. *Without* a canonical document, each resource needs to understand the document structures of the other resources. If a change is made to the document structure of one of the resources, logic for data synchronization would need to change in all resources.

  For example, consider keeping data synchronized between three resources: a CRM system, Billing system, and Order Management system. Because the systems map directly to and from each others native structures, a change in one resource's structure affects all resources. If the address structure was changed for the CRM system, you would need to update data synchronization logic in all resources.



When you use a canonical document, you limit the number of data synchronization logic changes that you need to make. Using the same example where the address structure changes for the CRM system, you would only need to update data synchronization logic in the CRM system.



- **Makes adding new resources to the data synchronization easier.** With canonical documents, you only need to add data synchronization logic to the new resource to use the canonical document. *Without* a canonical document, you would need to update the data synchronization logic of all resources so they understand the structure of the newly added resource, and the new resource would need to understand the document structures of all the existing resources.

Structure of Canonical Documents and Canonical IDs

You define the structure of the canonical documents to include a superset of all the fields that are required to keep data synchronized between the resources. For more information, see "Defining the Structure of the Canonical Document" on page 176.

One field that you must include in the structure of the canonical document is a key value called the *canonical ID*. The canonical ID uniquely identifies the object to which the canonical document refers. In other words, the canonical ID in a canonical document serves the same purpose that the native ID does for a native document from one of your resources.

For example, in a CRM system document, the native ID might be a Customer ID that uniquely identifies a customer's account in the CRM system. Similarly, a document from the Billing system might use an Account ID for the native ID to uniquely identify an account. When the CRM system document or Billing system documents get mapped to a canonical document, the document must contain a canonical ID that uniquely identifies the object (customer or account).

Integration Server provides *key cross-referencing* to allow you to create and manage the values of canonical IDs and their mappings to native IDs. The key cross-referencing tools that Integration Server provides are:

- Cross-reference database component that you use to store relationships between canonical IDs and native IDs.
- Built-in services that you use to manipulate the cross-reference table.

For more information, see "Key Cross-Referencing and the Cross-Reference Table" on page 168.

Key Cross-Referencing and the Cross-Reference Table

Key cross-referencing allows you to use a common key (i.e., the canonical ID) to build relationships between equivalent business objects from different resources. You build the relationships by mapping key values (i.e., the native IDs) from the resources to a common canonical ID. You maintain these relationships in the cross-reference database component (if using an external RDBMS) or the cross-reference table (if using the embedded internal database). For simplicity, in this chapter the term cross-reference table is used to encompass both. For information about configuring the cross-reference table, see "Configuring Integration Server for Key Cross-Reference and Echo Suppression" on page 46.

Note:
The field names listed in the table below are not the actual column names used in the cross-reference database component. They are the input variable names used by the key cross-referencing built-in services that correspond to the columns of the cross-reference database component.

The cross-reference table includes the fields described in the following table:

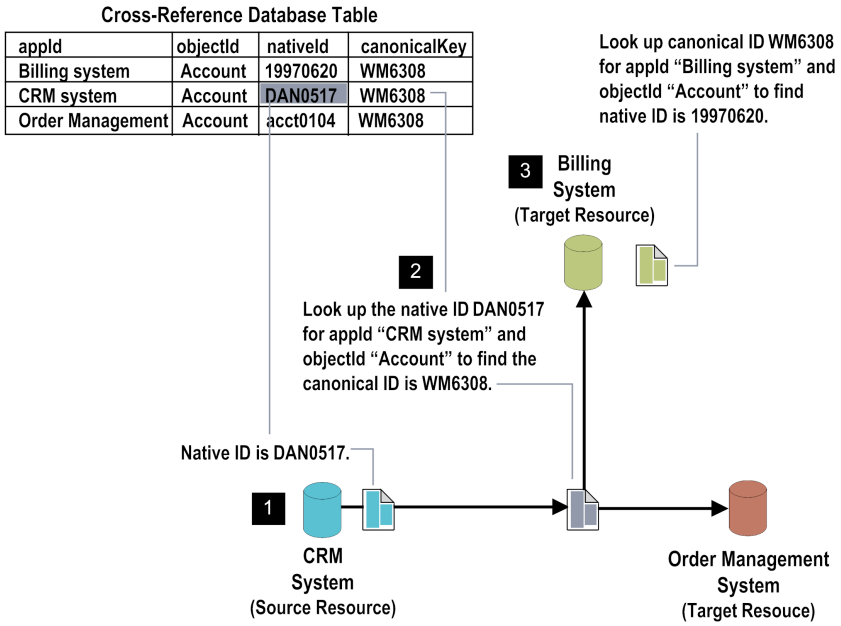
Fields	Description
appId	A string that contains the identification of a resource, for example, "CRM System". You assign this value.
objectId	A string that contains the identification of the of the object that you want to keep synchronized, for example, "Account". You assign this value.
nativeId	The native ID of the object for the specific resource specified by appId. You obtain this value from the resource.
canonicalKey	The canonical ID that is common to all resources for identifying the specific object. You can assign this value, or you can allow a built-in service to generate the value for you.

For example, to synchronize data between a CRM system, Billing system, and Order Management system, you might have the following rows in the cross-reference table:

appId	objectId	nativeId	canonicalKey
CRM system	Account	DAN0517	WM6308
Billing system	Account	19970620	WM6308
Order Management	Account	acct0104	WM6308

How the Cross-Reference Table Is Used for Key Cross-Referencing

The following diagram illustrates how to use the cross-reference table for key cross-referencing during data synchronization.



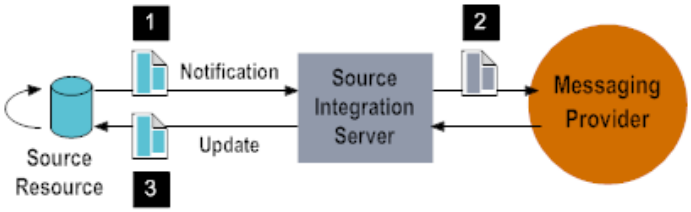
Step	Description
1	As described in "Data Synchronization with webMethods " on page 164, when a source makes a data change, the source sends a document to notify other resources of the change. A service that you create receives this document. Your service builds the canonical document that describes the change.
2	When forming the canonical document, to determine the value to use for the canonical ID, your service invokes a built-in service. This built-in service inspects the cross-reference table to locate the row that contains the native ID from the source document. The built-in service then returns the corresponding canonical ID from the cross-reference table. For more information about the built-in services you use, see "Setting Up Key Cross-Referencing in the Source Integration Server " on page 177.
3	A service that you create on the target receives the canonical document. When a target receives the canonical document, it needs to determine the native ID of the object that the change affects. To determine the native ID, your service on the target invokes a built-in

Step	Description
	service. This built-in service inspects the cross-reference table to locate the row that contains the canonical ID and the appropriate resource identified by the appId and object identified by objectId. The built-in service then returns the corresponding native ID from the cross-reference table. For more information about the built-in services you use, see "Setting Up Key Cross-Referencing in the Target Integration Server " on page 181.

Echo Suppression for N-Way Synchronizations

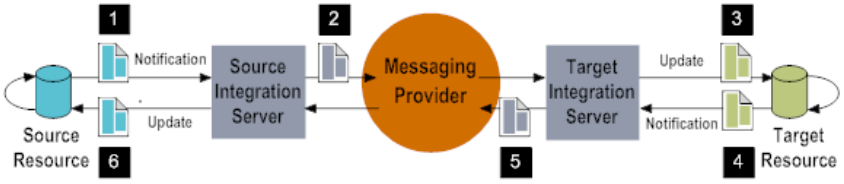
One other feature that Integration Server provides for data synchronization is echo suppression, which is also called latching. *Echo suppression* (or *latching*) is the process of preventing circular updating from occurring. Circular updating can occur when performing n-way synchronizations (data synchronizations where every resource can be a source *and* a target as well).

Circular updating occurs when the source subscribes to the canonical document that it publishes as illustrated in the diagram below.



Step	Description
1	A data change occurs on a source, and the source resource sends a notification document.
2	The source Integration Server builds and publishes a canonical document.
3	Because the source is also a target, it (as well as all other targets) subscribes to the canonical document via a trigger. As a result, the source receives the canonical document it just published. Logic on the source Integration Server uses the canonical document to build an update document to send to the source. The source receives the update document and makes the data change again. Because the source made this data change, it once again acts as a source to notify targets of the data change. The process starts again with step 1.

In addition to the source immediately receiving the canonical that it formed, it can also receive the canonical document many more times because other targets build and publish the canonical document after making the data change that the source initiated. See below for an illustration of this circular updating.



Step	Description
1	A data change occurs on a source, and the source sends a notification document.
2	The source Integration Server builds and publishes a canonical document.
3	A target receives the canonical document and makes the equivalent change.
4	Because the target made a data change, it sends a notification document for the data change.
5	The target Integration Server builds and publishes a canonical document.
6	The source receives the notification of the change that was made by the target and makes the change, again. This results in the process starting again with step 1.

To avoid the circular updating, Integration Server provides you with the following tools to perform echo suppression:

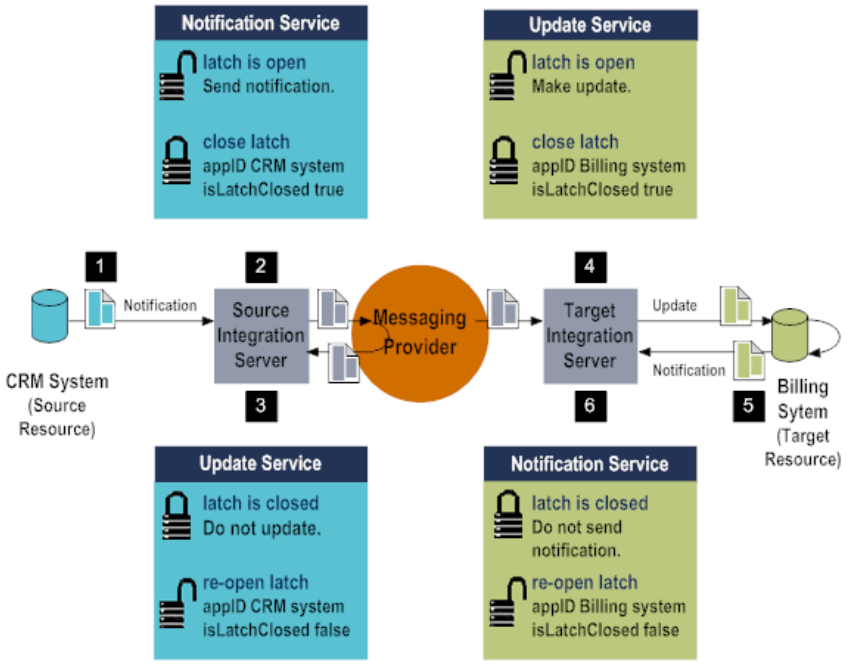
- The isLatchClosed field in the cross-reference table that you use to keep track of whether a resource has already made a data change or not.
- Built-in services that you use to manipulate the isLatchClosed field in the cross-reference table to:
  - Determine the value of the isLatchClosed field, and
  - Set the value of the isLatchClosed column.

How the isLatchClosed Field Is Used for Echo Suppression

In addition to the appID, objectId, nativeId, and canonicalKey fields that are described in "Key Cross-Referencing and the Cross-Reference Table" on page 168, the cross-reference table also includes the isLatchClosed field. The isLatchClosed field acts as a flag that indicates whether an object managed by a resource (e.g., account information) is allowed to be updated or whether a data change has already been made to the object and therefore should not be made again.

- **When the isLatchClosed field is false,**this indicates that the latch is open and therefore updates can be made to the object.
- **When the isLatchClosed field is true,** this indicates that the latch is closed and therefore updates should not be made to the object.

The following diagram illustrates how to use the isLatchClosed field for echo suppression during data synchronization.



Step	Description
1	A data change occurs on a source, and the source sends a notification document.
2	A notification service that you create to notify targets of a data change invokes the pub.synchronization.latch:isLatchClosed built-in service to determine whether the latch for the object is open or closed. Initially for the source, the latch for the object that was changed is open. This indicates that updates *can* be made to this object. The object is identified in the cross-reference table by the following cross-reference table fields and the latch is considered open because isLatchClosed is false:

appId	objectId	canonicalKey	isLatchClosed
CRM system	Account	WM6308	false

Finding that the latch is currently open, the notification service builds the canonical document and publishes it. The notification service invokes the pub.synchronization.latch:closeLatch built-in service to close the latch. This sets the isLatchClosed field to true and indicates that updates *cannot* be made to this object and prevents a circular update. After the latch is closed, the cross-reference table fields are as follows:

appId	objectId	canonicalKey	isLatchClosed
CRM system	Account	WM6308	true

3	Because the source is also a target, it subscribes to the canonical document it just published. The trigger passes the canonical document to a service you create to update the resource when a data change occurs. This update service invokes the pub.synchronization.latch:isLatchClosed built-in service to determine whether the latch is open or closed for the object.

Step	Description

Finding that the latch is currently closed, which indicates that the change has already been made, the update service does *not* make the update to the object. The update service invokes the pub.synchronization.latch:openLatch built-in service to re-open the latch to allow future updates to the object. After the latch is open, the cross-reference table fields are as follows:

appId	objectId	canonicalKey	isLatchClosed
CRM system	Account	WM6308	false

4 A service you create to update the target when a data change occurs receives the canonical document. This update service invokes the pub.synchronization.latch:isLatchClosed built-in service to determine whether the latch is open or closed for the object. Initially, the cross-reference table fields for the target object are as follows:

appId	objectId	canonicalKey	isLatchClosed
Billing system	Account	WM6308	false

Because initially the isLatchClosed column is set to false, this means the latch is open and updates *can* be made to this object. To make the update, the update service maps information from the canonical document to a native document for the target resource and sends the document to the target. The target resource uses this document to make the equivalent change.

The update service uses the pub.synchronization.latch:closeLatch built-in service to close the latch. This indicates that updates *cannot* be made to this object and prevents a circular update. After the latch is closed, the cross-reference table fields are as follows:

appId	objectId	canonicalKey	isLatchClosed
Billing system	Account	WM6308	true

5 Because the target made a data change, it sends notification of a change.

6 Because the target is also a source, when it receives notification of a data change, it attempts to notify other targets of the data change. A notification service that you create to notify targets of a data change invokes the pub.synchronization.latch:isLatchClosed built-in service to determine whether the latch for the object is open or closed.

Finding that the latch is closed, which indicates that the change has already been made, the notification service does *not* build the canonical document. The notification service simply invokes the pub.synchronization.latch:openLatch built-in service to re-open the latch. Because the latch is now open, future updates can be made to the object. After the latch is open, the cross-reference table fields are as follows:

appId	objectId	canonicalKey	isLatchClosed
Billing system	Account	WM6308	false

Tasks to Perform to Set Up Data Synchronization

The following table lists the tasks you need to perform to synchronize data changes to an object that is maintained in several of your resources (e.g., account information across all resources). Additionally, the table lists the section in this chapter where you can find more information about each task.

Task	For more information, see...
Ensure the cross-reference table is set up.	"Defining How a Source Resource Sends Notification of a Data Change" on page 175
Define publishable document type for the notification documents that each source sends when the object being synchronized is changed.	"Defining How a Source Resource Sends Notification of a Data Change" on page 175
Define an publishable document type for the canonical document, which describes the data change for all target resources.	"Defining the Structure of the Canonical Document" on page 176
Define logic on the source Integration Server to receive the source resource's notification document, build the canonical document, and publish the canonical document. This includes the following tasks: ■ Create a trigger to subscribe to the source's notification document. Note that you only need to create this trigger if the source publishes its notification document. ■ Create a service that builds the canonical document from the fields in the source's notification document and publishes the canonical document to notify targets of the data change.	"Setting Up Key Cross-Referencing in the Source Integration Server " on page 177
Define logic on the target Integration Server that receives the canonical document and interacts with a target resource to make the equivalent change to the object on the target. This includes the following tasks: ■ Create a trigger that subscribes to the canonical document that the source Integration Server publishes. ■ Create an IS document type for the native document that the target Integration Server sends the target resource, so the target can make the equivalent change. ■ Create a service that builds the target native document from fields in the canonical document.	"Setting Up Key Cross-Referencing in the Target Integration Server " on page 181

Task	For more information, see...
If you are doing n-way synchronizations, add logic to perform echo suppression to the services you created for key cross-referencing.	"For N-Way Synchronizations Add Echo Suppression to Services" on page 184

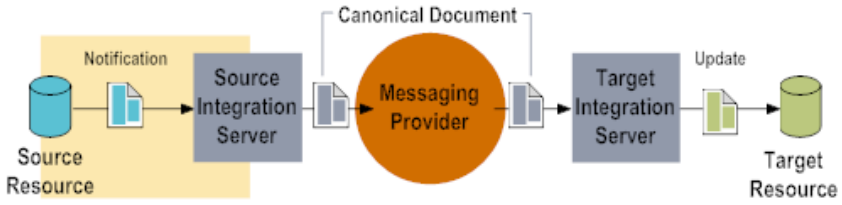
Defining How a Source Resource Sends Notification of a Data Change

Before you create logic that receives a notification document from a source resource and builds the canonical document, you need to determine how the source resource is to notify the source Integration Server when a data change occurs.

The source resource notifies other resources of a data change by sending a document that describes the change. The type of document the source sends depends on whether you are:

- Using a webMethods adapter with the source, or
- Developing your own interaction with the source.

The following diagram highlights the part of data synchronization that this section addresses.



When Using an Adapter with the Source

When you use an adapter to manage the source resource, configure the adapter to send an adapter notification when a change occurs on the resource. For information about how to configure the adapter, see the documentation for the adapter.

If the adapter does *not* create an publishable document type for the adapter notification, use Software AG Designer to define an publishable document type that defines the structure of the adapter notification.

Based on the adapter, the adapter does one of the following to send the adapter notification:

- **Publishes the adapter notification.** A trigger on the source Integration Server subscribes to this publishable document type. When the trigger receives a document that matches the publishable document type for the adapter notification, it invokes the trigger service that builds the canonical document.
- **Directly invokes the service that builds the canonical document.** When the service is directly invoked, the adapter notification is sent to the service as input.

For more information about creating the service that builds the canonical document, see "Setting Up Key Cross-Referencing in the Source Integration Server " on page 177.

When Developing Your Own Interaction with the Source

When you develop your own logic to interact with the source resource, the logic should include sending a document when a data change occurs within the resource. You define the document fields that you require for the notification. Be sure to include a field for the native ID to identify the changed object on the source.

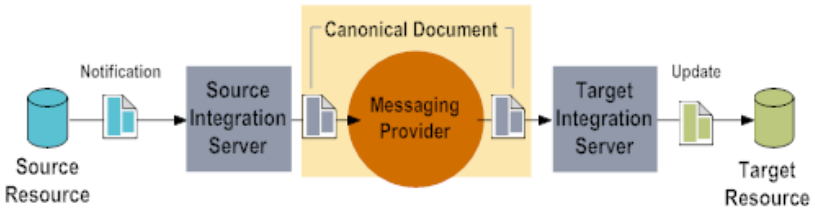
After determining the fields that you need in the source native document (i.e., the notification), use Software AG Designer to define an IS document type for the native document. The logic you create to interact with the source resource can do one of the following to send the source native document:

- **Publish the source native document.** If your logic publishes the source's native document, define a publishable document type for the source's native document. A trigger on the source Integration Server subscribes to this publishable document type. When the trigger receives a document that matches the publishable document type for the source's native document, it invokes the trigger service that builds the canonical document.
- **Directly invoke the service that builds the canonical document,** When the service is directly invoked, the source's native document is sent to the service as input. If your logic passes the native document to the service as input, the IS document type does not need to be publishable.

For more information about creating the service that your logic should invoke, see "Setting Up Key Cross-Referencing in the Source Integration Server " on page 177.

Defining the Structure of the Canonical Document

The following diagram highlights the part of data synchronization that uses the canonical document.



To define the structure for canonical documents, you include a superset of all the fields that are required to keep data synchronized between the resources. Additionally, you must include a field for the canonical ID.

The following table lists options for defining the structure of a canonical document:

Use a...	Benefit
Standard format (e.g., cXML, CBL, RosettaNet)	A standards committee has already decided the structure, and you can leverage their thought and effort.

Use a...	Benefit
Complete custom format	You define a unique structure that you tailor for your organization. The document structure might be smaller, and therefore easier to maintain because it only contains the fields your enterprise requires. Also, the smaller size has a positive effect on performance.
Custom format based on a standard	You define a unique structure by starting with a structure that a standards committee has defined. You can take advantage of the thought and effort already put into deciding the standards-based format. However, you can delete fields that your enterprise might not need and add additional fields that are specific to your enterprise.

After determining the fields that you need in the canonical document, use Software AG Designer to define an publishable document type for the canonical document. For more information about how to create publishable document types, see *webMethods Service Development Help*.

Setting Up Key Cross-Referencing in the Source Integration Server

The source resource sends a notification document to the source Integration Server when a data change occurs in the source resource. This section describes how to define the logic for the source Integration Server to:

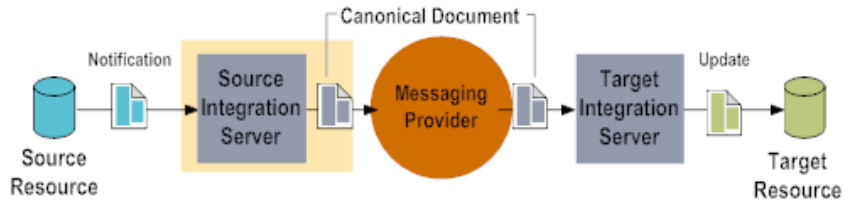
- Receive the notification document,
- Use notification document to build the canonical document, and
- Publish the canonical document.

You create a service to build and publish the canonical document. The logic to build the canonical document uses built-in services for key cross-referencing, which are described in "Built-In Services for Key Cross-Referencing" on page 178. In this chapter, the service that builds the canonical document is referred to as a notification service.

You should implement key cross-referencing for both one-way and n-way synchronizations.

Note:
For an overview of key cross-referencing, including the problem key cross-referencing solves and how key cross-referencing works, see "Key Cross-Referencing and the Cross-Reference Table" on page 168.

The following diagram highlights the part of data synchronization that this section addresses.



Built-In Services for Key Cross-Referencing

The following table lists the built-services that webMethods provides for key cross-referencing. The key cross-referencing services are located in the pub.synchronization.xref folder. For more information about these services, see the the section *pub.synchronization.xref:createXReference* in the *webMethods Integration Server Built-In Services Reference*.the section *pub.synchronization.xref:createXReference* in the *webMethods Integration Server Built-In Services Reference*.

Service	Description
createXReference	Used by the source to assign a canonical ID and add a row to the cross-reference table to create a cross-reference between the canonical ID and the source's native ID. To assign the value for the canonical ID, you can either specify the value as input to the createXReference service or have the createXReference service generate the value.
insertXReference	Used by a target to add a row to the cross-reference table to add the target's cross-reference between an existing canonical ID (which the source added) and the target's native ID.
getCanonicalKey	Retrieves the value of the canonical ID from the cross-reference table that corresponds to the native ID that you specify as input to the getCanonicalKey service.
getNativeId	Retrieves the value of the native ID from the cross-reference table that corresponds to the canonical ID that you specify as input to the getNativeId service.

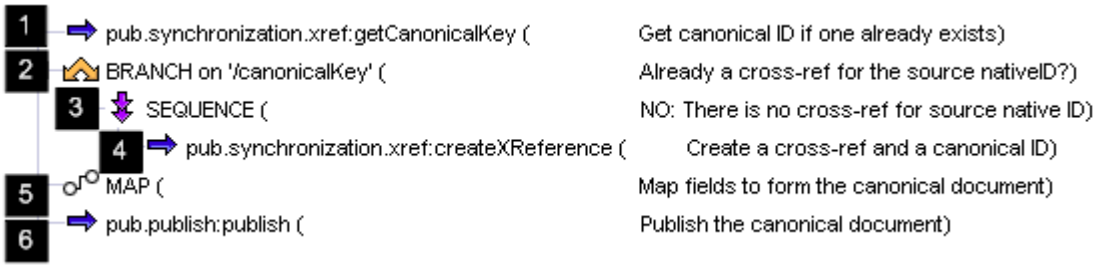
Setting up the Source Integration Server

To set up key cross-referencing in the source Integration Server:

- **Create a trigger to subscribe to the source resource's notification document, if applicable.** You only need to define a trigger that subscribes to the notification document if the source resource publishes the notification document (i.e., either an adapter notification or other native document). If the source directly passes the notification document to a service as input, you do not need to define a trigger. If you need to define a trigger, define the trigger that:
  - Subscribes to the publishable document type that defines the notification document (i.e., either an adapter notification or other native document).
  - Defines the trigger service to be the service that builds the canonical document from the notification document.

  For more information, see *webMethods Service Development Help*.
- **Create the trigger service that builds the canonical document** and publishes the canonical document to propagate the data change to all target resources. You need to create a service that puts the data change information into a neutral format that all targets understand. The neutral format is the canonical document.

The service builds the canonical document by mapping information from the notification document to the canonical document. To obtain the canonical ID for the canonical document, the service uses the built-in key cross-referencing services pub.synchronization.xref:getCanonicalKey and/or pub.synchronization.xref:createXReference, as shown in the sample logic below. After forming the canonical document, your service publishes the canonical document.



Step	Description

1 **Determine whether there is already a canonical ID.** Invoke the pub.synchronization.xref:getCanonicalKey service to locate a row in the cross-reference table for the source object. If the row already exists, a canonical ID already exists for the source object. Pass the getCanonicalKey service the following inputs that identify the source object:

In this input variable...	Specify...
appId	The identification of the application (e.g., CRM system).
objectId	The string that you assigned to identify the object (e.g., Account). This string is referred to as the object ID.
nativeId	The native ID from the notification document (e.g., adapter notification), which was received as input to your service.

If the getCanonicalKey service finds a row in the cross-reference table that matches the input information, it returns the value of the canonical ID in the *canonicalKey* output variable. If no row is found, the value of the *canonicalKey* output variable is blank (i.e., an empty string). For more information about the getCanonicalKey service, see the the section *pub.synchronization.xref:getCanonicalKey* in the *webMethods Integration Server Built-In Services Reference*.the *webMethods Integration Server Built-In Services Reference* section *pub.synchronization.xref:getCanonicalKey* in the .

2 **Split logic based on whether the canonical ID already exists.** Use a BRANCH flow step to split the logic. Set the **Switch** property of the BRANCH flow step to `canonicalKey`.

3 **Build a sequence of steps to execute when the canonical ID does not already exist.** Under the BRANCH flow step is a single sequence of steps that should be executed only if a canonical ID was *not* found. Note that the **Label** property for the SEQUENCE flow step is set to blank. At run time, the server matches the value of the *canonicalKey* variable to the **Label** field to determine whether to execute the sequence. Because

Step	Description

the *canonicalKey* variable is set to blank (i.e., an empty string), the label field must also be blank.

Important:
Do not use **$null** for the **Label** property. An empty string is not considered a null.

4 **If there is no canonical ID, define one.** If a row for the source object is not found in the cross-reference table, there is no canonical ID for the source object. Define a canonical ID by adding a row to the cross-reference table to cross-reference the source native ID with a canonical ID. You add the row by invoking the pub.synchronization.xref:createXReference service. Pass the createXReference service the following:

In this input variable...	Specify...
appId	The identification of the application (e.g., CRM system).
objectId	The object type (e.g., Account).
nativeId	The native ID from the notification document (e.g., adapter notification), which was received as input to your service.
canonicalKey	(optional) The value you want to assign the canonical ID.

If you do not specify a value for the *canonicalKey* input variable, the createXReference service generates a canonical ID for you. For more information about the createXReference service, see the the section *pub.synchronization.xref:createXReference* in the *webMethods Integration Server Built-In Services Reference*.the section *pub.synchronization.xref:createXReference* in the *webMethods Integration Server Built-In Services Reference*.

5 **Build the canonical document.** Map fields from the notification document (e.g., adapter notification) to the fields of the canonical document. Make sure you map the canonical ID generated in the last step to the canonical ID field of the canonical document.

The notification document has the structure that you previously defined with a the publishable document type. See "Defining How a Source Resource Sends Notification of a Data Change" on page 175. Similarly, the canonical document has the structure that you previously defined with a publishable document type. See "Defining the Structure of the Canonical Document" on page 176.

Note:
Although this sample logic shows only a single MAP flow step, you might need to use additional flow steps or possibly create a separate service to build the canonical document.

Step	Description
6	**Publish the canonical document.** After the service has formed the canonical document, invoke the pub.publish:publish service to publish the canonical document to the messaging provider.

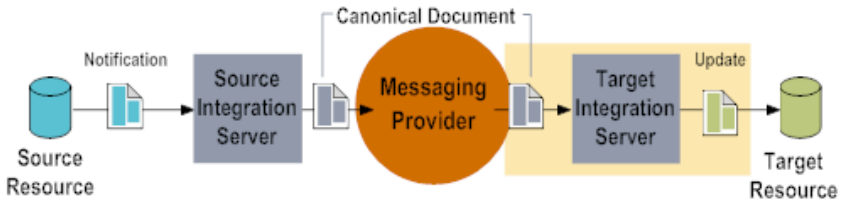
Setting Up Key Cross-Referencing in the Target Integration Server

The canonical document is published to the target Integration Servers. This section describes how to define the logic for a target Integration Server to:

- Receive the canonical document.
- Use the canonical document to build a document to inform the target resource of the data change. This document has a structure that is native to the target resource.
- Send the native document to the target resource, so the target resource can make the equivalent data change.

You create a service to build the native document and send it to the target resource. The logic to build the native document uses built-in services for key cross-referencing, which are described in "Built-In Services for Key Cross-Referencing" on page 178. In this chapter, the service that receives the canonical document and builds a native document is referred to as an update service.

The following diagram highlights the part of data synchronization that this section addresses.



To set up key cross-referencing in the target Integration Server:

- **Create a trigger that subscribes to the canonical document that the source server publishes.** On the target Integration Servers, define a trigger that:
  - Subscribes to the publishable document type that defines the canonical document.
  - Defines the trigger service to be the service that builds the builds a native document for the target resource.

  For more information, see *webMethods Service Development Help*.
- **Create an IS document type** that defines the structure of the document that the target Integration Server needs to send to the target resource to notify it of a data change. For more information about how to create IS document types, see *webMethods Service Development Help*.

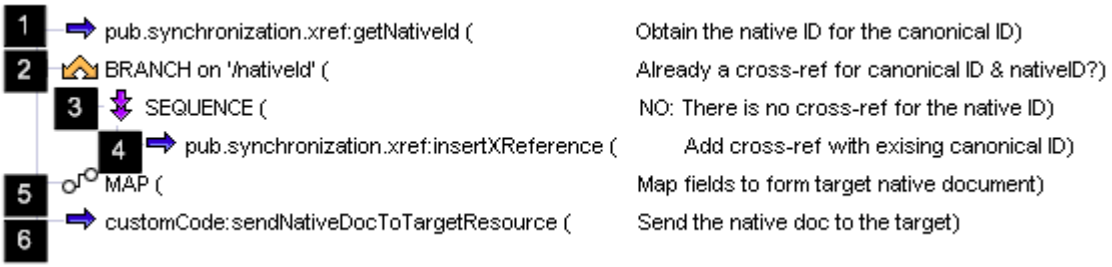
- **Create the trigger service that uses the canonical document to build the target native document and sends the native document to the target resource.** The service receives the canonical document, which contains the description of the data change to make. However, typically the target resource will not understand the canonical document. Rather, the target resource requires a document in its own native format.

  The service can build the native document for the target resource by mapping information from the canonical document to the target resource's native document format. Make sure you include the native ID in this document. To obtain the native ID, invoke the pub.synchronization.xref:getNativeId built-in service. If the native ID is cross-referenced with the canonical ID in the cross-reference table, this service returns the native ID. If no cross-reference has been set up for the object, you will need to determine the best way to obtain the native ID.

  After forming the native document, the trigger service interacts with the target resource to make the data change.

  Note:
  For a description of the built-in services that webMethods provides for key cross-referencing, see "Built-In Services for Key Cross-Referencing" on page 178.

The following shows sample logic for the update service. See the table after the diagram for more information.



Step	Description
1	**Obtain the native ID for the target object if there is an entry for the target object in the cross-reference table.** Invoke the pub.synchronization.xref:getNativeId service to locate a row in the cross-reference table for the target object. If the row already exists, the row contains the native ID for the target object. Pass the getNativeId service the following inputs that identify the target object:

In this input variable...	Specify...
appId	The identification of the application (e.g., Billing system).
objectId	The object type (e.g., Account).
canonicalKey	The canonical ID from the canonical document, which was received as input to your service.

Step	Description
	If the getNativeId service finds a row that matches the input information, it returns the value of the native ID in the *nativeId* output variable. If no row is found, the value of the *nativeId* output variable is blank (i.e., an empty string). For more information about the getNativeId service, see the the section *pub.synchronization.xref:getNativeId* in the *webMethods Integration Server Built-In Services Reference*.the section *pub.synchronization.xref:getNativeId* in the *webMethods Integration Server Built-In Services Reference*.
2	**Split logic based on whether a native ID was obtained for the target resource.** Use a BRANCH flow step to split the logic. Set the **Switch** property of the BRANCH flow step to `nativeId`, to indicate that you want to split logic based on the value of the *nativeId* pipeline variable.
3	**Build a sequence of steps to execute when the native ID is not obtained.** Under the BRANCH flow step is a single sequence of steps to perform only if a native ID was *not* found. Note that the **Label** property for the SEQUENCE flow step is set to blank. At run time, the server matches the value of the *nativeID* variable to the label field to determine whether to execute the sequence. Because the *nativeId* variable is set to blank (i.e., an empty string), the **Label** field must also be blank.
	Important: Do not use **$null** for the **Label** property. An empty string is not considered null.
4	**If no native ID was obtained, specify one.** If a native ID was not found, add a row to the cross-reference table for the target object to cross-reference the target native ID with the canonical ID by invoking the pub.synchronization.xref:insertXReference service. Pass the insertXReference service:

In this input variable...	Specify...
appId	The identification of the application (e.g., Billing system).
objectId	The object type (e.g., Account).
nativeId	The native ID for the object in the target resource. You must determine what the native ID should be.
canonicalKey	The canonical ID from the canonical document, which was received as input to your service.

For more information about the insertXReference service, see the the section *pub.synchronization.xref:insertXReference* in the *webMethods Integration Server Built-In Services Reference*.the section *pub.synchronization.xref:insertXReference* in the *webMethods Integration Server Built-In Services Reference*.

5 **Build the native document for the target resource.** To build the native document, map fields from the canonical document to the fields of native document. Also map the native ID to the native document.

Step	Description
	The canonical document has the structure that you previously defined with a publishable document type. See "Defining the Structure of the Canonical Document" on page 176. Similarly, the native document has the structure that you previously defined with an IS document type. **Note:** Although this sample logic shows only a single MAP flow step, you might need to use additional flow steps or possibly create a separate service to build the native document for the target resource.
6	**Invoke a service to send the native document to the target resource, so the target resource can make the equivalent change.** Create a service that sends the native document to the target. If you use an adapter with your target resource, you can use an adapter service to update the target resource. For more information about adapter services, see the documentation for your adapter.

For N-Way Synchronizations Add Echo Suppression to Services

When you use n-way synchronization, you need to include logic that performs echo suppression to the:

- Notification services that run on source Integration Servers. For a description of notification services, see "Setting Up Key Cross-Referencing in the Source Integration Server " on page 177.
- Update services that run on target Integration Servers. For a description of update services, see "Setting Up Key Cross-Referencing in the Target Integration Server " on page 181.

Echo suppression logic blocks circular updating of data changes from occurring. Echo suppression is not needed when you use one-way synchronization.

Note:
For an overview of echo suppression, including information about how echo suppression solves the problem of circular updating, see "Echo Suppression for N-Way Synchronizations" on page 170.

Built-in Services for Echo Suppression

The following table lists the built-services that webMethods provides for echo suppression. The echo suppression services are located in the pub.synchronization.latch folder.

Service	Description
closeLatch	Closes the latch for the specified canonical ID, application ID (appId), and object type (objectId). To close the latch, the isLatchClosed field of the cross-reference

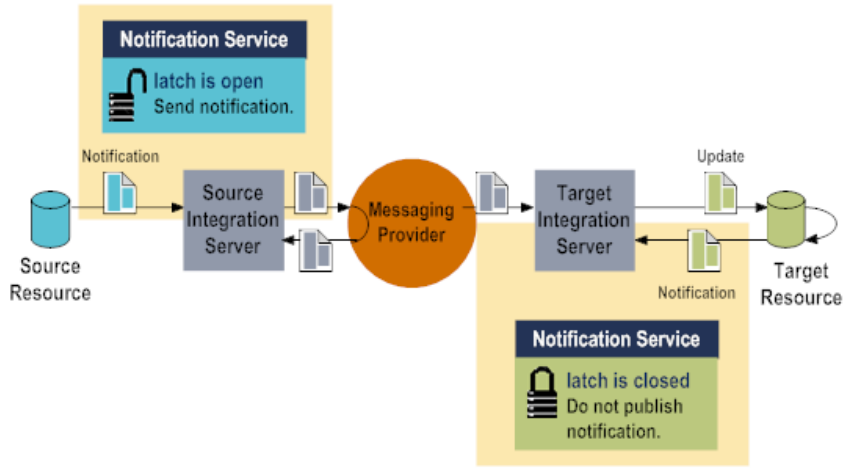
Service	Description
	table is set to `true`. A closed latch indicates that the resource described in the cross-reference row cannot be acted upon until the latch is open using the openLatch service.
isLatchClosed	Determines whether the latch is open or closed for the specified canonical ID, application ID (appId), and object type (objectId). To check the status of the latch, the service uses the isLatchClosed field of the cross-reference table. The output provides a status of true (the latch is closed) or false (the latch is open).
openLatch	Opens the latch for the specified canonical ID, application ID (appId), and object type (objectId). To open the latch, the isLatchClosed field of the cross-reference table is set to `false`. An open latch indicates that the resource described in the cross-reference row can be acted upon.

Adding Echo Suppression to Notification Services

The echo suppression logic in a notification service determines whether a latch is open or closed before it attempts to build and publish the canonical document.

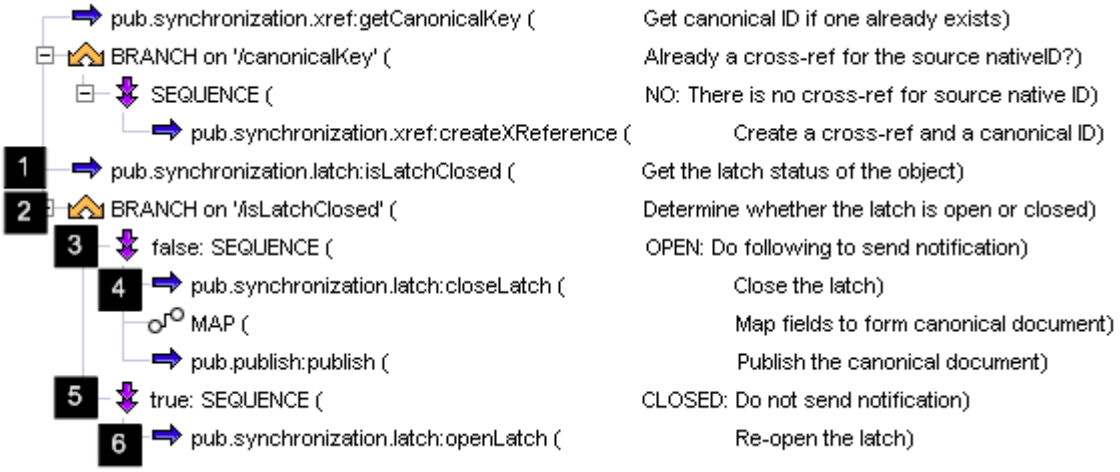
- **If the latch is open,** the resource is the source of the data change. In this case, the notification service on the source Integration Server builds the canonical document and publishes it. The notification service should include logic that closes the latch to prevent a circular update.
- **If the latch is closed,** the resource has already made the data change. In this case, the notification service does not need to build the canonical document to notify resources about the data change because the notification service on the source Integration Server has already done so. The notification service should simply re-open the latch and terminate processing.

The following diagram highlights the part of data synchronization that this section addresses.



Incorporating Echo Suppression Logic into a Notification Service

The following shows the sample notification service with echo suppression logic added to it. The sample notification service was presented in "Setting Up Key Cross-Referencing in the Source Integration Server " on page 177 along with a description of its flow steps, which are unnumbered in the sample below. The numbered flow steps in the sample below are the flow steps added for echo suppression. For more information about the numbered flow steps, see the table after the diagram.



Step	Description
1	**Determine whether the latch is open or closed for the changed object.** Invoke the pub.synchronization.latch:isLatchClosed service to locate a row in the cross-reference table for the object that has changed and for which you want to send notification. Pass the isLatchClosed service the following inputs that identify the object:

In this input variable...	Specify...
appId	The identification of the application (e.g., Billing system).
objectId	The object type (e.g., Account).
canonicalKey	The canonical ID.

The isLatchClosed service uses the isLatchClosed field from the matching row to determine whether the latch is open or closed. If the isLatchClosed field is 'false', the latch is open, and the isLatchClosed service returns 'false' in the *isLatchClosed* output variable. If the isLatchClosed field is 'true', the latch is closed, and the service returns 'true'. For more information about the isLatchClosed service, see the section *pub.synchronization.latch:isLatchClosed* in the *webMethods Integration Server Built-In Services Reference*.the section *pub.synchronization.latch:isLatchClosed* in the *webMethods Integration Server Built-In Services Reference*.

2	**Split logic based on whether the latch is open or closed.** Use a BRANCH flow step to split the logic. Set the **Switch** property of the BRANCH to `isLatchClosed`, to indicate that you want to split logic based on the value of the *isLatchClosed* pipeline variable.

Step	Description
3	**Build a sequence of steps to execute when the latch is open.** Because the **Label** property for the SEQUENCE flow step is set to false, this sequence of operations is executed when the *isLatchClosed* variable is false, meaning the latch is open. When the latch is open, the target resources have not yet been notified of the data change. This sequence of steps builds and publishes the canonical document.
4	**Close the latch.** When the latch is open, the first step is to close the latch. By closing the latch before publishing the canonical document, you remove any chance that Integration Server will receive and act on the published canonical document. To close the latch, invoke the pub.synchronization.latch:closeLatch service. Pass the closeLatch service the same input variables that were passed to the pub.sychronization.latch:isLatchClosed service in step 1 above. For more information about the closeLatch service, see the section *pub.synchronization.latch:closeLatch* in the *webMethods Integration Server Built-In Services Reference*.the section *pub.synchronization.latch:closeLatch* in the *webMethods Integration Server Built-In Services Reference*.
5	**Build a sequence of steps to execute when the latch is closed.** Because the **Label** property for the SEQUENCE flow operation is set to true, this sequence of steps is executed when the *isLatchClosed* variable is true, meaning the latch is closed. When the latch is closed, notification of the data change has already been published. As a result, the notification service does *not* need to build or publish the canonical document. This sequence of steps simply re-opens the latch.
6	**Re-open the latch.** Re-open the latch to reset the latch for future data changes. To re-open the latch, invoke the pub.synchronization.latch:openLatch service. Pass the openLatch service the same input variables that were passed to the pub.sychronization.latch:isLatchClosed service in step 1 above. For more information about the openLatch service, see the the section *pub.synchronization.latch:openLatch* in the *webMethods Integration Server Built-In Services Reference*.the section *pub.synchronization.latch:openLatch* in the *webMethods Integration Server Built-In Services Reference*..

Important:
If multiple resources will make changes to the same object simultaneously or near simultaneously, echo suppression cannot guarantee successful updating. If you expect simultaneous or near simultaneous updates, you must take additional steps:

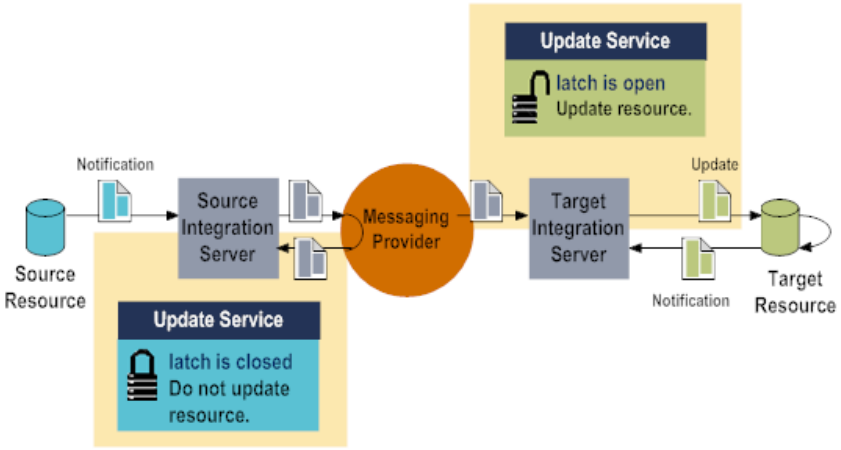
1. When defining the structure of a canonical document, include a tracking field that identifies the source of the change to the canonical document structure.
2. In the notification service include a filter or BRANCH to test on the source field to determine whether to send the notification.

Adding Echo Suppression to Update Trigger Services

The update trigger service receives the canonical document that describes a data change. The echo suppression logic in an update service determines whether a latch is open or closed before it attempts to use the information in the canonical document to update a resource with the data change.

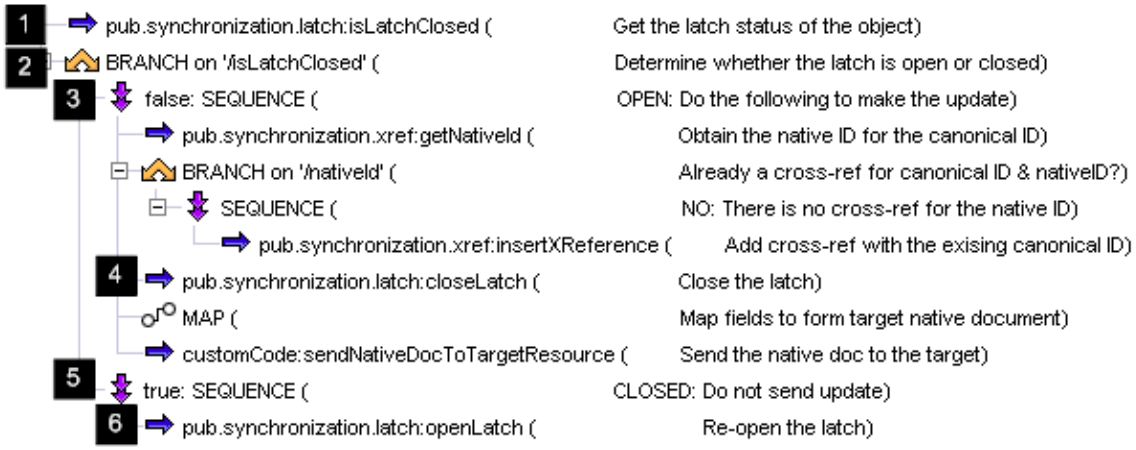
- **If the latch is open,** the data change has not yet been made to the resource. In this case, the update service builds and sends the native document that informs the target resource of the data change. The update service closes the latch to prevent a circular update.
- **If the latch is closed,** the resource was the source of the data change and has already made the data change. In this case, the update service does not need build and send the native document. The update service should simply re-open the latch and terminate processing.

The following diagram highlights the part of data synchronization that this section addresses.



Incorporating Echo Suppression Logic into an Update Service

The following shows the sample update service with echo suppression logic added to it. The sample update service was presented in "Setting Up Key Cross-Referencing in the Target Integration Server " on page 181 along with a description of its flow steps, which are unnumbered in the sample below. The numbered flow steps in the sample below are the flow steps added for echo suppression. For more information about the numbered flow steps, see the table after the diagram.



Step	Description
1	**Determine whether the latch is open or closed for the changed object.** Invoke the pub.synchronization.latch:isLatchClosed service to locate a row in the cross-reference table for the changed object. Pass the isLatchClosed service the following inputs that identify the object:

In this input variable...	Specify...
appId	The identification of the application (e.g., Billing system).
objectId	The object type (e.g., Account).
canonicalKey	The canonical ID.

The isLatchClosed service uses the isLatchClosed field from the matching row to determine whether the latch is open or closed. If the isLatchClosed field is 'false', the latch is open, and the isLatchClosed service returns 'false' in the *isLatchClosed* output variable. If the isLatchClosed field is 'true', the latch is closed, and the service returns 'true'. For more information about the isLatchClosed service, see the the section *pub.synchronization.latch:isLatchClosed* in the *webMethods Integration Server Built-In Services Reference*.the section *pub.synchronization.latch:isLatchClosed* in the *webMethods Integration Server Built-In Services Reference*..

2	**Split logic based on whether the latch is open or closed.** Use a BRANCH flow step to split the logic. Set the **Switch** property of the BRANCH to `isLatchClosed`, to indicate that you want to split logic based on the value of the *isLatchClosed* pipeline variable.
3	**Build a sequence of steps to execute when the latch is open.** Because the **Label** property for the SEQUENCE flow step is set to false, this sequence of operations is executed when the *isLatchClosed* variable is false, meaning the latch is open. When the latch is open, the target resource has not yet made the equivalent data change. This sequence of steps builds and sends a native document that the target resource uses to make the equivalent change.
4	**Close the latch.** When the latch is open, close the latch before sending the native document to the target resource. For n-way synchronizations because a target is also a source, when the resource receives and makes the equivalent data change, the resource then sends notification of a data change. By closing the latch before sending the native document to the target resource, you remove any chance that Integration Server will receive and act on a notification document being sent by the resource. To close the latch, invoke the pub.synchronization.latch:closeLatch service. Pass the closeLatch service the same input variables that were passed to the pub.sychronization.latch:isLatchClosed service in step 1 above. For more information about the closeLatch service, see the the section *pub.synchronization.latch:closeLatch* in the *webMethods Integration Server Built-In Services Reference*.the section *pub.synchronization.latch:closeLatch* in the *webMethods Integration Server Built-In Services Reference*..
5	**Build a sequence of steps to execute when the latch is closed.** Because the **Label** property for the SEQUENCE flow step is set to true, this sequence of steps is executed when the

Step	Description
	isLatchClosed variable is true, meaning the latch is closed. When the latch is closed, the resource has already made the equivalent data change. As a result, the update service does *not* need to build or send a native document to the target resource.
6	**Re-open the latch.** Re-open the latch to reset the latch for future data changes. To re-open the latch, invoke the pub.synchronization.latch:openLatch service. Pass the openLatch service the same input variables that were passed to the pub.sychronization.latch:isLatchClosed service in step 1 above. For more information about the openLatch service, see the the section *pub.synchronization.latch:openLatch* in the *webMethods Integration Server Built-In Services Reference*.the section *pub.synchronization.latch:openLatch* in the *webMethods Integration Server Built-In Services Reference*..

Important:
If multiple resources will make changes to the same object simultaneously or near simultaneously, echo suppression cannot guarantee successful updating. If you expect simultaneous or near simultaneous updates, you must take additional steps:

1. When defining the structure of a canonical document, include a tracking field that identifies the source of the change to the canonical document structure.
2. In the update service include a filter or BRANCH to test on the source field to determine whether to update the object.

A Naming Guidelines

- Naming Rules for Integration Server Elements 192
- Naming Rules for webMethods Broker Document Fields 192

Naming Rules for Integration Server Elements

Software AG Designer place some restrictions on the characters that can be used in element, package, and folder names. Specifically, element and package names cannot contain:

- Reserved words and characters that are used in Java or C/C++ (such as *for*, *while*, and *if*)
- Digits as their first character
- Spaces
- Control characters and special characters like periods (.), including:

?	'	-	#	=	)	(	.	/	\	;
&	@	^	!	|	}	{	`	>	<	
%	*	:	$	]	[	"	+	,	~	

- Characters outside of the basic ASCII character set, such as multi-byte characters

If you specify a name that disregards these restrictions, Designer displays an error message. When this happens, use a different name or try adding a letter or number to the name to make it valid.

Naming Rules for webMethods Broker Document Fields

When you save a trigger, Integration Server evaluates the filter to make sure it uses the proper syntax. Some field names that are valid on Integration Server are not valid on the Broker. If you want a filter to be saved on the Broker, you need to make sure that fields in filters conform to the following rules:

- Names must be unicode values.
- Characters must be alphanumeric, underscore, and plus codes over `\u009F`.
- The first character cannot be a numeric character (0–9) or an underscore (_).
- Names cannot contain symbols, spaces, or non-printable-ANSI.
- Following is a list of reserved words:

acl	any	boolean	broker
byte	char	client	clientgroup
const	date	double	enum
event	eventtype	extends	false
family	final	float	host
import	infoset	int	long

nal	null	server	short
string	struct	territory	true
typedef	unicode_char	unicode_string	typedef
unicode_char	unicode_string	union	unsigned

If Integration Server determines that the syntax is valid for the Broker, it saves the filter with the subscription on the Broker. If Integration Server determines that the filter syntax is not valid on the Broker or if attempting to save the filter on the Broker would cause an error, Integration Server saves the subscription on the Broker without the filter. The filter will be saved only on Integration Server.

Note:webMethods Broker is deprecated.

B Building a Resource Monitoring Service

- Overview 196
- Service Requirements 196

Overview

A *resource monitoring service* is a service that you create to check the availability of resources used by a trigger. 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 and the configured retry behavior is suspend and retry later.

When the resource monitoring service indicates that the resources used by the trigger are available, Integration Server resumes the trigger.

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.

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.