

Web Services Developer's Guide

Version 10.7

October 2020

This document applies to webMethods Integration Server 10.7 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 © 2007-2023 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-WS-DG-107-20230517

Table of Contents

About this Guide	9
Document Conventions	10
Online Information and Support	11
Data Protection	12
1 Working with Web Services	13
What Are Web Service Descriptors?	15
About Provider Web Service Descriptors	16
About Consumer Web Service Descriptors	30
About Refreshing a Web Service Descriptor	41
Viewing the WSDL Document for a Web Service Descriptor	50
WS-I Compliance for Web Service Descriptors	52
Changing the Target Namespace for a Web Service Descriptor	53
Viewing the Namespaces Used within a WSDL Document	54
Enabling MTOM/XOP Support for a Web Service Descriptor	54
Adding SOAP Headers to the Pipeline	55
Validating SOAP Response	56
Validating Schemas Associated with a Web Service Descriptor	57
Working with Binders	59
Working with Operations	68
Adding Headers to an Operation	74
About SOAP Fault Processing	77
Viewing Document Types for a Header or Fault Element	82
Working with Handlers	83
Working with Policies	85
About Pre-8.2 Compatibility Mode	88
2 SOAP Message Exchange Patterns	95
Message Exchange Patterns that Integration Server Supports	96
How Integration Server Determines the MEP Type to Use	96
How the MEP Type Affects the SOAP Response a Provider Returns	98
How <wsdl:output> and <wsdl:fault> Elements Affect a Consumer	99
How Adding Response Headers or Faults Affect In-Only MEP Operations	100
How the MEP Affects the Execution of Handlers	100
Considerations When Changing a Provider's Compatibility Mode	100
Considerations When Changing a Consumer's Compatibility Mode	102
3 Using SOAP over JMS with Web Services	103
Introduction	104
Pre-Requisites for Using SOAP/JMS	104
Using SOAP/JMS with Provider Web Service Descriptors	104
Using SOAP/JMS with Consumer Web Service Descriptors	108
Using SOAP/JMS with Web Services with Transactions	112

Asynchronously Invoking an In-Out Operation.....	114
Consumer Support for Robust In-Only Operations with SOAP/JMS Bindings.....	115
4 Working with Web Service Connectors.....	117
About Web Service Connectors.....	118
Signature for a Web Service Connector.....	118
How a SOAP Fault is Mapped to the Generic Fault Output Structure.....	144
Setting Transport Headers for HTTP/S.....	145
Setting Transport Headers for JMS.....	146
Passing Message-Level Security Information to a Web Service Connector.....	150
5 Working with Response Services.....	153
About Response Services.....	154
Signature for a Response Service.....	155
Signature for a genericFault_Response Service.....	155
6 About Handlers and Handler Services.....	157
What Are Handlers and Handler Services?.....	158
Setting Up a Handler.....	158
Registering a Handler.....	159
About Request Handler Services.....	159
About Response Handler Services.....	162
About Fault Handler Services.....	167
7 About Outbound Callback Services.....	171
What Are Outbound Callback Services?.....	172
Usage of Outbound Callback Services.....	172
Invoking Outbound Callback Services.....	173
8 MTOM Streaming.....	175
Configuring MTOM Streaming for a Web Service Descriptor.....	176
Integration Server Parameters for MTOM Streaming.....	176
Using MTOM Streaming for Service First Provider Web Service Descriptors.....	177
Using MTOM Streaming for WSDL First Provider Web Service Descriptors.....	177
Using MTOM Streaming for Consumer Web Service Descriptors.....	178
How MTOM Streaming Affects Saved Pipelines.....	179
9 Including SOAP Headers in the Pipeline.....	181
Anatomy of a SOAP Header in the Pipeline.....	182
Example of a SOAP Header in the Pipeline.....	183
10 Web Service Authentication and Authorization.....	185
Introduction.....	186
Authentication and Authorization for Consumer Web Service Descriptors.....	186
ACL Checking Scenarios for Consumer Web Service Descriptors.....	188
Authentication and Authorization for Provider Web Service Descriptors.....	190

ACL Checking Scenarios for Provider Web Service Descriptors.....	194
Authentication and Authorization for Consumer Web Service Descriptors While Processing Asynchronous Responses.....	195
ACL Checking Scenarios for Consumer Web Service Descriptors While Processing Asynchronous Responses.....	198
Authentication and Authorization for Provider Web Service Descriptors on an Earlier Web Services Implementation.....	199
ACL Checking Scenarios for Provider Web Service Descriptors on an Earlier Web Services Implementation.....	203
11 Transient Error Handling for Provider Web Service Descriptors.....	205
Introduction.....	206
Transient Error Handling for an Operation Invoked via SOAP over HTTP.....	206
Transient Error Handling for an Operation Invoked by a Non-Transacted SOAP-JMS Trigger.....	208
Transient Error Handling for an Operation Invoked by a Transacted SOAP-JMS Trigger.....	212
12 How Integration Server Builds Consumer and Provider Endpoint URLs.....	217
How Integration Server Builds the Consumer Endpoint URL.....	218
How Integration Server Builds the Provider Endpoint URL.....	219
13 How Integration Server Determines which Operation to Invoke.....	221
Determining the Operation for an HTTP/S Request.....	222
Determining the Operation for a SOAP/JMS Request.....	223
Fallback Mechanisms for Determining the Operation.....	223
14 Array Handling for Document/Literal and RPC/Literal.....	227
How Integration Server Represents Arrays for Document/Literal and RPC/Literal.....	228
Backward Compatibility for Web Service Descriptors Created in Integration Server 7.x.....	229
XML Namespace Decoding for Array Elements.....	229
15 Defining Policies for Web Services (WS-Policy).....	231
About WS-Policy.....	232
WS-Policy Files.....	232
About Updating WS-Policies.....	235
About Deleting WS-Policies.....	235
16 Securing Web Services (WS-Security).....	237
WS-Security in Integration Server.....	238
Transport-Based vs. Message-Based Security.....	238
WS-Security and the SOAP Message Header.....	239
WS-Security and the Message Direction.....	239
How You Can Secure SOAP Messages with WS-Security.....	240
17 WS-Security Certificate and Key Requirements.....	243
Overview.....	244

Certificate and Key Requirements for WS-Security.....	244
About Certificate and Key Resolution Order.....	246
WS-Security Key Resolution Order: Web Services Consumer.....	246
WS-Security Key Resolution Order: Web Services Provider.....	252
18 Securing Web Services Using WS-SecurityPolicy.....	261
About Implementing WS-SecurityPolicy.....	262
Securing Web Services Using Policies Based on WS-SecurityPolicy.....	263
WS-SecurityPolicy Files.....	267
WS-SecurityPolicy Assertions Reference.....	268
Policies Based on WS-SecurityPolicy that Integration Server Provides.....	281
19 Securing Web Services Using the WS-Security Facility.....	301
About the Integration Server WS-Security Facility.....	302
Configuring the WS-Security Facility.....	305
WS-Security Facility Policy Reference.....	308
Sample Policy File.....	316
Policy Files Supplied with the WS-Security Facility.....	318
20 Web Services Addressing (WS-Addressing).....	321
About WS-Addressing in Integration Server.....	322
Applying WS-Addressing to Web Service Descriptors.....	324
WS-Addressing Behavior of Web Service Descriptors.....	325
WS-Addressing Policies Provided by Integration Server.....	328
Accessing WS-Addressing Headers of a SOAP Message.....	328
Processing Responses Asynchronously.....	329
WS-Addressing and WSDL.....	330
Generation of the WS-Addressing Headers: Resolution Order and Usage.....	330
21 Web Services Reliable Messaging (WS-ReliableMessaging).....	335
About Web Services Reliable Messaging in Integration Server.....	336
Using Reliable Messaging in Integration Server.....	336
A Provided WS-SecurityPolicies vs. WS-Security Facility Policies.....	339
Overview.....	340
Policies that Provide Username Authentication.....	340
Policies that Provide Authentication Using X.509 Certificates.....	341
Policies that Provide SAML Authentication.....	341
Policies that Provide Signature and Encryption Without Authentication.....	342
B CDATA Blocks in Inbound and Outbound SOAP Messages.....	343
Support for Preserving CDATA Tag Delimiters in Inbound SOAP Messages.....	344
Support for Processing CDATA Blocks in Outbound SOAP Messages.....	344
Encoding CDATA Blocks in Outbound SOAP Messages.....	345
C Omitting Well-Known Schema Locations from Generated WSDL.....	347

D Preserving Namespace Declarations when Decoding xsd:any Elements.....349

About this GuideAbout this Guide

- Document Conventions 10
- Online Information and Support 11
- Data Protection 12

This guide is for developers who want to use Software AG Designer to create web services and incorporate web services into the integration solutions they develop.

This guide contains the following:

- An introduction to web service descriptors and web service connectors—the fundamental elements for web services in Integration Server.
- Conceptual and procedural information for creating and configuring web service descriptors.
- A description of the web service connector signature, including how Integration Server represents the elements of a SOAP fault in the pipeline.
- Detailed information about Integration Server processing, including authentication and authorization for web services, construction of endpoint URLs, use of SOAPAction to locate an operation, and array handling.
- Steps to configure MTOM streaming when sending and receiving SOAP messages using web services.
- In depth information about securing web services with WS-Security and WS-SecurityPolicy.

Note:

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

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format <i>folder.subfolder.service</i> , APIs, Java classes, methods, properties.
<i>Italic</i>	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.

Convention	Description
	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the symbol.
[]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information 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 Working with Web Services

■ What Are Web Service Descriptors?	15
■ About Provider Web Service Descriptors	16
■ About Consumer Web Service Descriptors	30
■ About Refreshing a Web Service Descriptor	41
■ Viewing the WSDL Document for a Web Service Descriptor	50
■ WS-I Compliance for Web Service Descriptors	52
■ Changing the Target Namespace for a Web Service Descriptor	53
■ Viewing the Namespaces Used within a WSDL Document	54
■ Enabling MTOM/XOP Support for a Web Service Descriptor	54
■ Adding SOAP Headers to the Pipeline	55
■ Validating SOAP Response	56
■ Validating Schemas Associated with a Web Service Descriptor	57
■ Working with Binders	59
■ Working with Operations	68
■ Adding Headers to an Operation	74
■ About SOAP Fault Processing	77
■ Viewing Document Types for a Header or Fault Element	82
■ Working with Handlers	83
■ Working with Policies	85

■ About Pre-8.2 Compatibility Mode	88
------------------------------------------	----

Web services are building blocks for creating open, distributed systems. A web service is a collection of functions that are packaged as a single unit and published to a network for use by other software programs. For example, you could create a web service that checks a customer's credit or tracks delivery of a package. If you want to provide higher-level functionality, such as a complete order management system, you could create a web service that maps to many different IS flow services, each performing a separate order management function.

Designer uses web service descriptors to encapsulate information about web services and uses web service connectors to invoke web services.

Note:

Information about web services is located in *webMethods Service Development Help*, *Web Services Developer's Guide*, and *webMethods Integration Server Administrator's Guide*.

- *webMethods Service Development Help* includes this “Working with Web Services” on page 13 topic which provides procedures for using Designer to create web service descriptors, adding operations, binders, handlers, and policies to a web service descriptor; and setting web service descriptor properties.
- *Web Services Developer's Guide* contains information such as how Integration Server processes web services, how a SOAP fault is represented in the pipeline, steps to configure MTOM streaming when sending and receiving SOAP messages using web services, and how to secure web services with WS-Security and WSSecurityPolicy. For completeness, *Web Services Developer's Guide* also contains the “Working with Web Services” on page 13 topic that appears in *webMethods Service Development Help*.
- *webMethods Integration Server Administrator's Guide* contains information about creating web service endpoint alias and configuring Integration Server to use web services reliable messaging.

What Are Web Service Descriptors?

A *web service descriptor (WSD)* is an element on Integration Server that defines a web service in IS terms. The WSD encapsulates all the information required by the provider or the consumer (requester) of a web service. The WSD contains the message formats, data types, transport protocols, and transport serialization formats that should be used between the consumer (requester) and the provider of the web service. It also specifies one or more network locations at which a web service can be invoked. In essence, the WSD represents an agreement governing the mechanics of interacting with that service.

- A *provider web service descriptor* defines a web service that is hosted on the Integration Server, that is, a service “provided” to external users. A provider web service descriptor will expose one or more IS services as *operations*, which can be published to a registry as a single web service. External users can access the web service through the registry and invoke the IS services remotely.
- A *consumer web service descriptor* defines an external web service, allowing Integration Server to create a *web service connector (WSC)* for each operation in the web service. The web service connector(s) can be used in Designer just like any other IS flow service; when a connector is invoked it calls a specific operation of a web service. In version 9.0 and later, Integration Server also creates a response service for each operation in the web service. Response services are flow services to which you can add custom logic to process asynchronous SOAP responses.

About Provider Web Service Descriptors

You can turn any service in any Integration Server package into a web service by using the IS service as an operation in a provider web service descriptor. Integration Server provides an environment for executing services efficiently and securely. It receives and decodes requests from clients, calls the requested services, and encodes and returns the output to the clients.

A *provider web service descriptor* (WSD) is created from one or more IS services or from a single WSDL document, and is designed to allow the IS services to be invoked as web services over a network. The provider web service descriptor contains all the data required to create a WSDL document for the IS web service, as well as the data needed at run time to process the request and response.

You can create a provider web service descriptor from a service that exists on Integration Server or from a WSDL document.

- A *service first provider web service descriptor* refers to provider web service descriptors created from an existing service on Integration Server. In this case, you specify the protocol, binding style/use, and host server when creating the WSD. The IS service becomes an operation in the provider web service descriptor. Integration Server uses the existing service signature as the input and output messages for the operation. You can add operations and bindings to a service first provider web service descriptor.
- A *WSDL first provider web service descriptor* refers to a provider web service descriptor created from an existing WSDL document, from a service asset in CentraSite, or from a web service acquired from a UDDI registry. In this case, Designer uses the message and operation definitions from the WSDL to generate a “placeholder” flow service for each operation encountered in the WSDL, along with IS document types defining the input and output signature of the generated flow services. You can then implement any required logic within the placeholder flow service. Note that you cannot add operations or bindings to a WSDL first provider WSD.

The provider web service descriptor can be published to a UDDI registry (or other publicly accessible server) as a web service, which can be invoked remotely by an external user. A web service provider can also distribute WSDL files directly to consumers.

Service Signature Requirements for Service First Provider Web Service Descriptors

When you create a service first provider web service descriptor, you select one or more services to use as operations. The service signature becomes the input and output messages for the operations in the WSDL document. However, Integration Server allows constructs within service signatures that cannot be represented in certain web service style/use combinations.

When adding a service to or creating a service first provider web service descriptor, Integration Server verifies that the service signature can be represented in the style/use specified for the web service descriptor. If a service signature does not meet the style/use signature requirements, Integration Server will not add the service as an operation. Or, in the case of creating a service first provider WSD, Integration Server will not create the WSD.

Following, is a list of service signature restrictions and requirements for each style/use. Note that this list may not be exhaustive.

Signature Restrictions for Document/Literal

*body fields are not allowed at the top level

@attribute fields (fields starting with the “@” symbol) are not allowed at the top level

String table fields are not allowed

Signature Restrictions for RPC/Encoded

* body fields are not allowed

@attribute fields are not allowed (fields starting with the “@” symbol)

Top-level fields cannot be namespace qualified

Top-level field names cannot be in the format *prefix:localName*

Signature Restrictions for RPC/Literal

*body fields are not allowed at the top level

@attribute fields (fields starting with the “@” symbol) are not allowed at the top level

String table fields are not allowed

List fields (String List, Document List, Document Reference List, and Object List) are not allowed at the top level

Duplicate field names (identically named fields) are not allowed at the top level

Top-level fields cannot be namespace qualified

Top-level field names cannot be in the format *prefix:localName*

Using XML Namespaces with Prefixes with Fields in Service Signatures

You can associate the name of an Integration Server field (such as an IS document variable) with an XML namespace. When you do this, the local name is the name of the field and the XML namespace name is the URI that identifies the namespace. You can also include a prefix as part of the name.

Assign XML namespaces and prefixes to Integration Server fields as follows:

- To assign an XML namespace to an Integration Server field, complete the **XML Namespace** property in the **General** category of the field’s Properties view.
- To assign a prefix to an Integration Server field, precede the field name with the prefix followed by a colon (for example, *prefix:variableName*).

Note:

The style/use combinations RPC/Literal and RPC/Encoded prohibit top-level field from being namespace qualified.

Handling Incomplete Service Signatures Using Wrapper Services

When you use a service as an operation in a web service descriptor, the service signature must accurately and completely reflect the expected service input and output.

If the signature is not accurate or complete, the WSDL document created for the web service descriptor will contain incorrect signature information. Clients generated from the WSDL document may not execute as expected.

However, sometimes it may not be possible to make the service signature complete before using it in a web service descriptor or you may not want to alter the service signature. In these situations, you can expose the service as a web service by creating a wrapper service. The wrapper service needs to declare the complete service signature and invoke the service that you want to expose as a web service. You can then use the wrapper service as an operation in a provider web service descriptor.

For example, suppose that you want to expose an XSLT service as a web service on one Integration Server and invoke it from another. However, the XSLT source contains an optional run-time property that is added to the pipeline at run time. This optional property is not reflected in the input signature of the XSLT service. If you added the XSLT service to a provider web service descriptor, the resulting WSDL document would not list the property as part of the input message. Consequently, a consumer web service descriptor and a web service connector created from the WSDL document would not account for the property and invocation will fail.

To successfully use the XSLT service as a web service, you can do the following:

1. Create a wrapper flow service that:
 - Defines all of the input parameters of the XSLT service in its input signature.
 - Defines the run-time property of the XSLT source in its input signature.
 - Invokes the XSLT service.
2. On the Integration Server that hosts the wrapper flow service and the XSLT service, create a provider web service descriptor from the wrapper flow service.

On the Integration Server from which you will invoke the web service, create a consumer web service descriptor from the WSDL of the provider web service descriptor. The web service connector that corresponds to the operation for the XSLT service will display the complete input signature.

Creating a Service First Provider Web Service Descriptor

Keep the following points in mind when creating a service first provider web service descriptor:

- You must have Write access to the folder in which you want to store the provider web service descriptor.

- The style and use selected for a provider web service descriptor determines what types of fields and field names are allowed in the service signature. Designer will not create a provider web service descriptor if the signature of the service does not meet the requirements of the selected binding style/use. For more information, see [“Service Signature Requirements for Service First Provider Web Service Descriptors”](#) on page 16.
- Depending on the use and style that you specify, you may have to either rename certain fields in the IS service or assign an XML namespace to them.
- You must have at least one web service endpoint alias that specifies the JMS transport before you can create a provider web service descriptor with a JMS binder. For more information about creating a web service endpoint alias, see the section *Configuring Endpoint Aliases for Web Services* in the *webMethods Integration Server Administrator's Guide*.
- When using an adapter service to create a provider web service descriptor, if the service returns values in the pipeline that do not match the output signature, you must change those variable properties to optional fields (where applicable), or else wrap the service in a flow to add or drop variables to match the output signature.
- Web service descriptors that are not running in compatibility mode can stream MTOM attachments for both inbound and outbound SOAP messages. To stream MTOM attachments, the object that represents the field to be streamed should be of type `com.wm.util.XOPObject` Java class.
- You can quickly create a service first provider web service descriptor by right-clicking the service, selecting **Generate Provider WSD**. Enter a name for the web service descriptor in the Provide a Name dialog box and click **OK**. Designer automatically creates a provider web service descriptor in the same folder as the selected IS service, using all the default options.
- Integration Server generates invalid WSDL for Axis and .Net clients if the provider web service descriptor contains a C service for that takes a document specification as input. Axis and .Net clients cannot handle the resulting Java stub classes and throw an error. Do not use a C service with a document specification in the input in a server first provider web service descriptor if you know that the resulting WSDL will be used by Axis and .Net clients.

➤ To create a service first provider web service descriptor

1. In the Package Navigator view of Designer, click **File > New > Web Service Descriptor**.
2. In the New Web Service Descriptor dialog box, select the folder in which you want to save the provider web service descriptor. Click **Next**.
3. In the **Element Name** field, specify a name for the provider web service descriptor using any combination of letters, numbers, and/or the underscore character. Click **Next**.
4. Under **Create web service descriptor as**, select **Provider (Inbound Request)**.
5. Under **Web service source**, select **Existing IS service(s)**.

6. Click **Next**.
7. Select one or more services to include as operations in the provider web service descriptor. Click **Next**.
8. Provide the following information:

In this field...	Specify...
SOAP version	Whether SOAP messages for this web service should use SOAP 1.1 or SOAP 1.2 message format.
Transport	<p>The transport protocol used to access the web service. Select one of the following:</p> <ul style="list-style-type: none">■ HTTP■ HTTPS■ JMS
Use and style for operations	<p>The style/use for operations in the provider web service descriptor. Select one of the following:</p> <ul style="list-style-type: none">■ Document - Literal■ RPC - Literal■ RPC - Encoded
Endpoint	<p>The address at which the web service can be invoked. Do one of the following:</p> <ul style="list-style-type: none">■ To use a provider web service endpoint alias to specify the address, select the Alias option. Then, in the Alias list, select the provider web service endpoint alias. Select <code>DEFAULT(aliasName)</code> if you want to use the information in the default provider web service endpoint alias for the address. If the Alias list includes a blank row, the Integration Server does not have a default provider web service endpoint alias for the protocol.<div data-bbox="500 1514 1365 1724"><p>Note:</p><p>If you select the blank row and a default provider endpoint alias is later set for the selected protocol, Integration Server then uses the information from the alias when constructing the WSDL document and during run-time processing.</p></div>■ To specify a host and port as the address, select the Host option. Then, in the Host field specify the host name for the Integration Server on which the web service resides. In the Port field, specify an active

In this field...	Specify...
	<p>HTTP or HTTPS listener port defined on the Integration Server specified in the Host field.</p> <p>Note: You can only specify Host and Port for the endpoint if a default provider endpoint alias does not exist for the selected protocol. When a default alias exists, Designer populates the Host and Port fields with the host and port from the default provider endpoint alias.</p> <p>Note: If you selected JMS as the transport, you must specify an alias. After you select a provider web service endpoint alias, Designer displays the initial portion of the JMS URI that will be used as the address in the Port address (prefix) field.</p>
Directive	The SOAP processor used to process the SOAP messages received by the operation in the provider web service descriptor. The Directive list displays all of the SOAP processors registered on the Integration Server. The default processor is ws - Web Services SOAP Processor .
Target namespace	The URL that you want to use as the target namespace for the provider web service descriptor. In a WSDL document generated for this provider web service descriptor the elements, attributes, and type definitions will belong to this namespace.

Note:
If you specify a transport, but do not specify a host, port, or endpoint alias, Integration Server uses the primary port as the port in the endpoint URL. If the selected transport and the protocol of the primary port do not match, web service clients will not execute successfully. For more information see [“Protocol Mismatch Between Transport and Primary Port” on page 22](#).

9. Under **Enforce WS-I Basic Profile 1.1 compliance** do one of the following:

- Select **Yes** if you want Designer to validate all the web service descriptor objects and properties against the WS-I requirements before creating the web service descriptor.
- Select **No** if you do not want Designer to enforce compliance for WS-I Basic Profile 1.1.

Note:
WS-I compliance cannot be enforced if the WSDL contains a SOAP over JMS binding.

10. If you want Integration Server to use the Xerces Java parser to validate the schema elements that represent the signatures of the services used as operations, select the **Validate schema using Xerces** check box.

11. Click **Finish**.

If Designer cannot create or cannot completely generate a web service descriptor, Designer displays error messages or warning messages.

Notes:

- If you selected the **Validate schema using Xerces** check box, when creating a service first provider web service descriptor, Integration Server converts the signatures of the services used as operations to XML schema elements. Then Integration Server uses the Xerces Java parser to validate the schema elements. If the schema element does not conform syntactically to the schema for XML Schemas defined in *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures* (which is located at <https://www.w3.org/TR/xmlschema11-1/>, Integration Server does not create the web service descriptor. Instead, Designer displays an error message that lists the number, title, location, and description of the validation errors.

Note: Integration Server uses Xerces Java parser version Xerces-J 2.12.1-xml-schema-1.1. Limitations for this version are listed at <http://xerces.apache.org/xerces2-j/xml-schema.html>.

- Set up a package dependency if an IS service uses a document type from a different package as the input or output signature.
- The Message Exchange Pattern (MEP) that Integration Server uses for the operation it creates from the service can be In-Out MEP or In-Only MEP. Integration Server always uses In-Out MEP when the web service descriptor's **Pre-8.2 compatibility mode** property is true. When this property is false, Integration Server uses:
 - In-Out MEP when the service signature contains both input and output parameters.
 - In-Only MEP when the service signature contains no output parameters.

Note:

If you want to use Robust In-Only MEP rather than In-Only MEP, after creating the web service descriptor for a service with no output parameters, add a fault to the operation.

For more information about Integration Server MEP support, see the section *How Integration Server Determines the MEP Type to Use* in the *Web Services Developer's Guide*.

Protocol Mismatch Between Transport and Primary Port

A protocol mismatch between the transport for a binder in a provide web service descriptor and the primary port can occur in the following situations:

- When creating a service first web service descriptor, you specify a transport, but do not specify a host, port, or endpoint alias and there is not a default provider endpoint alias for the transport protocol, Integration Server uses the primary port as the port in the endpoint URL.
- When creating a WSDL first provider web service descriptor and default provider endpoint alias is not specified for the protocol used by the binding in the WSDL document. Integration Server uses the primary port as the port in the endpoint URL.

If the selected transport and the protocol of the primary port do not match, Designer displays the following warning when you save the provider web service descriptor:

Selected transport protocol does not match that of the primary port on Integration Server.

For example, suppose that you specify a transport of HTTPS when creating the provider web service descriptor, but do not specify a host, port, or endpoint alias. Additionally, Integration Server does not identify a default web service provider endpoint alias for HTTPS. Furthermore, the primary port is an HTTP port. In this situation, Designer displays the above message.

You must resolve this mismatch before making a WSDL document for this provider web service descriptor available to web service consumers. Otherwise, the web service clients will not execute successfully.

Creating a WSDL First Provider Web Service Descriptor

You can create a WSDL first provider web service descriptor from a WSDL document accessed via a URL, from a UDDI registry, or from a service asset in CentraSite.

Keep the following points in mind when creating a WSDL first provider web service descriptor:

- You must have Write access to the folder in which you want to store the provider web service descriptor.
- If the URL for the WSDL contains special characters that need to be encoded, specify the encoding using the **Encoding for WSDL URL** option in the Web Service Descriptor Editor Preferences page.
- Before you can create a provider web service descriptor from a WSDL document that contains a JMS binding, you must have at least one valid web service endpoint alias that specifies the JMS transport. When you create a provider web service descriptor from a WSDL document that specifies a SOAP over JMS binding, Designer automatically assigns the first valid provider web service endpoint alias for JMS to the web service descriptor binder. If there is not valid endpoint alias for JMS, the web service descriptor cannot be created. For example, if the only web service endpoint alias that exists for JMS specifies a SOAP-JMS trigger that no longer exists, Integration Server does not consider the endpoint to be valid and does not create the web service descriptor.
- To create a WSDL first provider web service descriptor from a web service in a UDDI registry, Designer must be configured to connect to that UDDI registry.
- To create a WSDL first provider web service descriptor from a service asset in CentraSite, Designer must be configured to connect to CentraSite.
- You can also create a provider web service descriptor from a service asset in CentraSite by dragging and dropping the service asset from the Registry Explorer view into Package Navigator view. Designer prompts you for a name for the web service descriptor and prompts you to indicate whether you want to create a consumer or provider web service descriptor.
- You can specify whether Integration Server enforces strict, lax, or no content model compliance when generating IS document types from the XML Schema definition contained or referenced

in the WSDL document. Content models provide a formal description of the structure and allowed content for a complex type. The type of compliance that you specify can affect whether Integration Server generates an IS document type from a particular XML Schema definition successfully.

- Do not create a WSDL first provider web service descriptor from a WSDL that specifies RPC-Encoded, contains attributes in its operation signature, and/or has complex type definitions with mixed content. Integration Server might successfully create a web service descriptor from such WSDLs. However, the web service descriptor may exhibit unexpected runtime behavior.

➤ **To create a WSDL first provider web service descriptor**

1. In Package Navigator view, click **File > New > Web Service Descriptor**.
2. In the New Web Service Descriptor dialog box, select the folder in which you want to save the provider web service descriptor. Click **Next**.
3. In the **Element Name** field, specify a name for the provider web service descriptor using any combination of letters, numbers, and/or the underscore character. Click **Next**.
4. Under **Create web service descriptor as**, select **Provider (Inbound Request)**.
5. Under **Web service source**, select **WSDL**. Click **Next**.
6. Under **Source location**, do one of the following:

Select...	To generate a provider web service descriptor from...
CentraSite	A service asset in CentraSite
File/URL	A WSDL document that resides on the file system or on the Internet.
UDDI	A WSDL document in a UDDI registry

7. Click **Next**.
8. If you selected **CentraSite** as the source, under **Select Web Service from CentraSite**, select the service asset in CentraSite that you want to use to create the web service descriptor. Click **Next**.

Designer filters the contents of the Services folder to display only service assets that are web services.

If Designer is not configured to connect to CentraSite, Designer displays the **CentraSite > Connections** preference page and prompts you to configure a connection to CentraSite.

9. If you selected **File/URL** as the source, do one of the following:

- Enter the URL for the WSDL document. The URL should begin with http:// or https://. Click **Next**.
 - Click **Browse** to navigate to and select a WSDL document on your local file system. Click **Next**.
10. If you selected **UDDI** as the source, under **Select Web Service from UDDI Registry**, select the web service from the UDDI registry. Click **Next**.
- If Designer is not currently connected to a UDDI registry, the Open UDDI Registry Session dialog box appears. Enter the details to connect to the UDDI registry and click **Finish**.
11. Under **Content model compliance**, select one of the following to indicate how strictly Integration Server enforces content model compliance when creating IS document types from the XML Schema definition in the WSDL document.

Select...	To...
Strict	<p>Generate the IS document type only if Integration Server can represent the content models defined in the XML Schema definition correctly. Document type generation fails if Integration Server cannot accurately represent the content models in the source XML Schema definition.</p> <p>Currently, Integration Server does not support repeating model groups, nested model groups, or the any attribute. If you select strict compliance, Integration Server does not generate an IS document type from any XML schema definition that contains those items.</p> <p>Note: If Integration Server cannot generate an IS document type that complies with the content model in the XML schema definition in the WSDL document, Integration Server will not generate the provider web service descriptor.</p>
Lax	<p>When possible, generate an IS document type that correctly represents the content models for the complex types defined in the XML schema definition from the WSDL document. If Integration Server cannot correctly represent the content model in the XML Schema definition in the resulting IS document type, Integration Server generates the IS document type using a compliance mode of None.</p> <p>When you select lax compliance, Integration Server will generate the IS document type even if the content models in the XML schema definition cannot be represented correctly.</p>
None	<p>Generate an IS document type that does not necessarily represent or maintain the content models in the source XML Schema definition.</p>

Select...	To...
	When compliance is set to none, Integration Server generates IS document types the same way they were generated in Integration Server releases prior to version 8.2.

12. Select the **Enable MTOM streaming for elements of type base64Binary** check box if you want elements declared to be of type base64Binary in the WSDL or schema to be enabled for streaming of MTOM attachments. For more information about MTOM streaming for web services, see the *Web Services Developer's Guide*.
13. If you want Integration Server to use the Xerces Java parser to validate any schema elements in the WSDL document or any referenced XML Schema definitions before creating the web service descriptor, select the **Validate schema using Xerces** check box.

Note: Integration Server automatically uses the internal schema parser to validate the schemas in or referenced by a WSDL document. However, the Xerces Java parser provides stricter validation than the Integration Server internal schema parser. As a result, some schemas that the internal schema parser considers to be valid might be considered invalid by the Xerces Java parser. While validation by the Xerces Java parser can increase the time it takes to create a web service descriptor and its associated elements, using stricter validation can help ensure interoperability with other web service vendors.

14. Under **Enforce WS-I Basic Profile 1.1 compliance** do one of the following:
 - Select **Yes** if you want Designer to validate all the WSD objects and properties against the WS-I requirements before creating the WSD.
 - Select **No** if you do not want Designer to enforce compliance for WS-I Basic Profile 1.1.

Note:

WS-I Basic Profile 1.0 supports only HTTP or HTTPS bindings. Consequently, WS-I compliance cannot be enforced if the WSDL contains a SOAP over JMS binding.

15. Click **Next** if you want to specify different prefixes than those specified in the XML schema definition. If you want to use the prefixes specified in the XML schema definition itself, click **Finish**.
16. On the Assign Prefixes panel, if you want the web service descriptor to use different prefixes than those specified in the XML schema definition, select the prefix you want to change and enter a new prefix. Repeat this step for each namespace prefix that you want to change.

Note:

The prefix you assign must be unique and must be a valid XML NCName as defined by the specification <http://www.w3.org/TR/REC-xml-names/#NT-NCName>.

17. Click **Finish**.

Designer creates the provider web service descriptor and saves it to the folder you specified. Designer also creates supporting IS elements, such as flow services, IS document types, and IS schemas.

If Designer cannot create or cannot completely generate a provider web service descriptor, Designer displays error messages or warning messages.

18. If Integration Server determines that an XML Schema definition included in or referenced by the WSDL document is invalid or cannot be generated according to the selected content model compliance option, Designer displays the validation error message at the top of the Select Document Type Generation Options panel. Click **Cancel** to abandon this attempt to create a consumer web service descriptor. Alternatively, click **Back** to navigate to previous panels and change your selections.
19. If the WSDL document contains constructs that the current web services stack does not support, Designer displays a message identifying the reasons the web service descriptor cannot be created on the current web services stack. Designer then prompts you to create the web service descriptor using an earlier version of the web services stack. If you want to create the web service descriptor using the earlier version of the web services stack, click **OK**. Otherwise, click **Cancel**.

Notes:

- If the WSDL document contains a construct supported on the web services implementation introduced in Integration Server 7.1 but not on the current Web Services Stack, Designer gives you the option of creating the web service descriptor using the earlier web services implementation. If the WSDL document contains any of the following, Designer prompts you to use the web services implementation introduced in 7.1:
 - Mixed “use” values across bindings and operations referenced by services in the WSDL document.
 - Mixed “style” values across bindings referenced by services in the WSDL.
 - More than one operation with the same name in the same port type.
 - Bindings that do not contain all of the operations declared in the port type.
 - Services with multiple bindings that reference different port types.

If you create the web services descriptor using the earlier version of the web services stack, the **Pre-8.2 compatibility mode** property will be set to true for the resulting web service descriptor.

Note:

The **Pre-8.2 compatibility mode** property and the ability to run in pre-8.2 compatibility mode are deprecated as of Integration Server 10.4 due to the deprecation of the web services implementation introduced in Integration Server version 7.1.

- Integration Server does not create a provider web service descriptor if the WSDL document contains any bindings that are not supported by Integration Server.

- Integration Server will create duplicate operations in case the WSDL document has multiple port names for the same binding. To ensure that duplicate operations are not created, modify the WSDL to make the port name unique for each binding.
- When creating the binders for a WSDL first provider web service descriptor generated from a WSDL document with an HTTP or HTTPS binding, Integration Server assigns the default provider endpoint alias for HTTP or HTTPS to the binder. Integration Server uses the information from the default provider endpoint alias during WSDL generation and run-time processing. Integration Server determines whether to use the HTTP or HTTPS default provider endpoint alias by selecting the default alias for the protocol specified in the soap:addressLocation attribute of the wsdl:port element. If a default provider endpoint alias is not specified for the protocol used by the binding in the WSDL document, Integration Server uses its own hostname as the host and the primary port as the port. If the binding transport protocol is not the same as the primary port protocol, the web service descriptor has a protocol mismatch that you must resolve before making a WSDL generated from the descriptor available to consumers. For more information about a protocol mismatch, see “Protocol Mismatch Between Transport and Primary Port” on page 22.

Note:

The default provider endpoint alias also determines security, WS-Addressing, and WS-Reliable Messaging information for the web service descriptor and resulting WSDL document.

- Integration Server uses the internal schema parser to validate the XML schema definition associated with the WSDL document. If you selected the **Validate schema using Xerces** check box, Integration Server also uses the Xerces Java parser to validate the XML Schema definition. With either parser, if the XML Schema does not conform syntactically to the schema for XML Schemas defined in *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures* (which is located at <https://www.w3.org/TR/xmlschema11-1/>), Integration Server does not create an IS schema or an IS document type for the web service descriptor. Instead, Designer displays an error message that lists the number, title, location, and description of the validation errors within the XML Schema definition.

Note: Integration Server uses Xerces Java parser version Xerces-J 2.12.1-xml-schema-1.1. Limitations for this version are listed at <http://xerces.apache.org/xerces2-j/xml-schema.html>.

- When validating XML schema definitions, Integration Server uses the Perl5 regular expression compiler instead of the XML regular expression syntax defined by the World Wide Web Consortium for the XML Schema standard. As a result, in XML schema definitions consumed by Integration Server, the pattern constraining facet must use valid Perl regular expression syntax. If the supplied pattern does not use proper Perl regular expression syntax, Integration Server considers the pattern to be invalid.

Note:

If the `watt.core.datatype.usejavaregex` configuration parameter is set to true, Integration Server uses the Java regular expression compiler instead of the Perl5 regular expression compiler. When the parameter is true, the pattern constraining facet in XML schema definitions must use valid syntax as defined by the Java regular expression.

- When creating the document types for the provider web service descriptor, Integration Server registers each document type with the complex type definition from which it was created in the schema. This enables Integration Server to provide derived type support for document creation and validation.
- If you selected strict compliance and Integration Server cannot represent the content model in the complex type accurately, Integration Server does not generate any IS document types or the web service descriptor.
- The contents of an IS document type with a **Model type** property value other than “Unordered” cannot be modified.
- For an IS document type from a WSDL document, Designer displays the location of the WSDL in the **Source URI** property. Designer also sets the **Linked to source** property to true which prevents any editing of the document type contents. To edit the document type contents, you first need to make the document type editable by breaking the link to the source. However, Software AG does not recommend editing the contents of document types created from WSDL documents.
- If the source WSDL document is annotated with WS-Policy:
 - Integration Server enforces the annotated policy at run time. However, if you attach a policy from the policy repository to the web service descriptor, the attached policy will override the original annotated policy.
 - Integration Server will only enforce supported policy assertions in the annotated policy. For information about supported assertions, see the *Web Services Developer's Guide*.
 - Integration Server does not save the annotated policy in the policy repository.
- The Message Exchange Pattern (MEP) that Integration Server uses for an operation defined in the WSDL can be In-Out MEP, In-Only MEP, or Robust In-Only MEP. Integration Server always uses In-Out MEP when the web service descriptor's **Pre-8.2 compatibility mode** property is set to true. When this property is set to false, Integration Server uses:
 - In-Out MEP when an operation has defined input and output.
 - In-Only MEP when an operation has no defined output and no defined fault.
 - Robust In-Only MEP when an operation has no defined output, but does have a defined fault.

For more information about Integration Server MEP support, see the section *How Integration Server Determines the MEP Type to Use* in the *Web Services Developer's Guide* .

- If the WSDL is annotated with WS-Policy, Integration Server will only enforce supported policy assertions. Currently Integration Server supports only WS-Security policies. Also be aware that Integration Server does not save the WS-Policy that is in the WSDL in the policy repository. Integration Server will enforce the annotated policy unless a policy that resides in the Integration Server policy repository is specifically attached to the web service descriptor. If you attach a policy to the web service descriptor, the attached policy will override the original annotated policy.

- Integration Server creates the docTypes and services folders to store the IS document types, IS schemas, and skeleton services generated from the WSDL document. These folders are reserved for elements created by Integration Server for the web service descriptor only. Do not place any custom IS elements in these folders. During refresh of a web service descriptor, the contents of these folders will be deleted and recreated.
- If an XML Schema definition referenced in the WSDL document contains the `<!DOCTYPE` declaration, Integration Server issues a `java.io.FileNotFoundException`. To work around this issue, remove the `<!DOCTYPE` declaration from the XML Schema definition.
- When creating a WSDL first provider web service descriptor from an XML Schema definition that imports multiple schemas from the same target namespace, Integration Server throws Xerces validation errors indicating that the element declaration, attribute declaration, or type definition cannot be found. The Xerces Java parser honors the first `<import>` and ignores the others. To work around this issue, you can do one of the following:
 - Combine the schemas from the same target namespace into a single XML Schema definition. Then change the XML schema definition to import the merged schema only.
 - When creating the WSDL first provider web service descriptor, clear the **Validate schema using Xerces** check box to disable schema validation by the Xerces Java parser. When generating the web service descriptor, Integration Server will not use the Xerces Java parser to validate the schemas associated with the XML Schema definition.

About Consumer Web Service Descriptors

To use Integration Server and Designer to invoke web services located on remote servers, create a consumer web service descriptor from a WSDL document, from a service asset in CentraSite, or from the web service in the UDDI registry. A *consumer web service descriptor (WSD)* defines an external web service. It contains all the data from the WSDL document that defines the web service, as well as data needed for certain Integration Server run-time properties.

Integration Server creates a *web service connector (WSC)* for each operation in the web service. A web service connector is a flow service with an input and output signature that corresponds to the input and output messages of the web service operation.

The web service connector is a flow service that:

- Uses an input and output signature that corresponds to the input and output messages of the web service operation.
- Contains flow steps that create and send a message to the web service using the transport, protocol, and location information specified in the web service.
- Contains flow steps that extract data from the output message returned by the web service.

When Integration Server executes a web service connector, the web service connector calls a specific operation of a web service.

In versions 9.0 and later, Integration Server also creates a response service for each operation in the WSDL document. Response services are flow services to which you can add custom logic to

process asynchronous SOAP responses. For more information about response services, see [“About Response Services” on page 41](#).

Creating a Consumer Web Service Descriptor

You can create a consumer WSD from a WSDL document accessible via a URL, a WSDL document in a UDDI registry, or a service asset in CentraSite.

- You must have Write access to the folder in which you want to store the consumer web service descriptor.
- If you are creating a consumer web service descriptor from a WSDL located on a website, if the website on which the document resides is password protected, you must download the WSDL document to your local file system and then create the consumer web service descriptor.
- If the URL for the WSDL contains special characters that need to be encoded, specify the encoding using the **Encoding for WSDL URL** option in the Web Service Descriptor Editor Preferences page.
- To create a consumer web service descriptor from a service asset in CentraSite, Designer must be configured to connect to CentraSite.
- You can also create a consumer web service descriptor from a service asset in CentraSite by dragging and dropping the service asset from the Registry Explorer view into Package Navigator view. Designer prompts you for a name for the web service descriptor and prompts you to indicate whether you want to create a consumer or provider web service descriptor.
- To create a consumer web service descriptor from a web service in a UDDI registry, Designer must be configured to connect to that UDDI registry.
- You can specify whether Integration Server enforces strict, lax, or no content model compliance when generating IS document types from the XML Schema definition contained or referenced in the WSDL document. Content models provide a formal description of the structure and allowed content for a complex type. The type of compliance that you specify can affect whether Integration Server generates an IS document type from a particular XML Schema definition successfully.
- Do not create a consumer web service descriptor from a WSDL that specifies RPC-Encoded, contains attributes in its operation signature, and/or has complex type definitions with mixed content. Integration Server might successfully create a web service descriptor from such WSDLs. However, the web service descriptor may exhibit unexpected runtime behavior.

➤ To create a consumer web service descriptor

1. In Package Navigator view, click **File > New > Web Service Descriptor**.
2. In the New Web Service Descriptor dialog box, select the folder in which you want to save the consumer web service descriptor. Click **Next**.

3. In the **Element Name** field, specify a name for the consumer WSD using any combination of letters, numbers, and/or the underscore character. Click **Next**.
4. Under **Create web service descriptor as**, select **Consumer (Outbound Request)**.
5. Under **Web service source**, select **WSDL**. Click **Next**.
6. Under **Source location**, do one of the following:

Select...	To generate a consumer web service descriptor from...
CentraSite	A service asset in CentraSite
File/URL	A WSDL document that resides on the file system or on the Internet.
UDDI	A WSDL document in a UDDI registry

7. If you specified **CentraSite** as the source, under **Select web service from CentraSite**, select the service asset in CentraSite that you want to use to create the web service descriptor. Click **Next**.

Designer filters the contents of the Services folder to display only service assets that are web services.

If Designer is not configured to connect to CentraSite, Designer displays the **CentraSite > Connections** preference page and prompts you to configure a connection to CentraSite.

8. If you specified **File/URL** as the source, do one of the following:
 - Enter the URL for the WSDL document. The URL should begin with http:// or https://. Click **Next**
 - Click **Browse** to navigate to and select a WSDL document on your local file system. Click **Next**
9. If you specified **UDDI** as the source, under **Select web service from UDDI Registry**, select the web service from the UDDI registry. Click **Next**.

If Designer is not currently connected to a UDDI registry, the Open UDDI Registry Session dialog box appears. Enter the details to connect to the UDDI registry and click **Finish**.

10. Under **Content model compliance**, select one of the following to indicate how strictly Integration Server enforces content model compliance when creating IS document types from the XML Schema definition in the WSDL document.

Select...	To...
Strict	<p>Generate the IS document type only if Integration Server can represent the content models defined in the XML Schema definition correctly. Document type generation fails if Integration Server cannot accurately represent the content models in the source XML Schema definition.</p> <p>Currently, Integration Server does not support repeating model groups, nested model groups, or the any attribute. If you select strict compliance, Integration Server does not generate an IS document type from any XML schema definition that contains those items.</p> <p>Note: If Integration Server cannot generate an IS document type that complies with the content model in the XML schema definition in the WSDL document, Integration Server will not generate the consumer web service descriptor.</p>
Lax	<p>When possible, generate an IS document type that correctly represents the content models for the complex types defined in the XML schema definition from the WSDL document. If Integration Server cannot correctly represent the content model in the XML Schema definition in the resulting IS document type, Integration Server generates the IS document type using a compliance mode of None.</p> <p>When you select lax compliance, Integration Server will generate the IS document type even if the content models in the XML schema definition cannot be represented correctly.</p>
None	<p>Generate an IS document type that does not necessarily represent or maintain the content models in the source XML Schema definition.</p> <p>When compliance is set to none, Integration Server generates IS document types the same way they were generated in Integration Server releases prior to version 8.2.</p>

11. Under **Document type generation**, select the **Enable MTOM streaming for elements of type base64Binary** check box if you want elements declared to be of type base64Binary in the WSDL or schema to be enabled for streaming of MTOM attachments. For more information about MTOM streaming for web services, see the *Web Services Developer's Guide*.
12. If you want to use the Xerces Java parser to validate any schema elements in the WSDL document or any referenced XML Schema definitions before creating the web service descriptor, select the **Validate schema using Xerces** check box.

Note: Integration Server uses an internal schema parser to validate the schemas in or referenced by a WSDL document. However, the Xerces Java parser provides stricter validation than the Integration Server internal schema parser. As a result, some schemas that the internal schema parser considers to be valid might be considered invalid by the Xerces Java parser.

While validation by the Xerces Java parser can increase the time it takes to create a web service descriptor and its associated elements, using stricter validation can help ensure interoperability with other web service vendors.

13. Under **Enforce WS-I Basic Profile 1.1 compliance** do one of the following:

- Select **Yes** if you want Designer to validate all the WSD objects and properties against the WS-I requirements before creating the WSD.
- Select **No** if you do not want Designer to enforce compliance for WS-I Basic Profile 1.1.

Note:

WS-I Basic Profile 1.0 supports only HTTP or HTTPS bindings. Consequently, WS-I compliance cannot be enforced if the WSDL contains a SOAP over JMS binding.

14. Click **Next** if you want to specify different prefixes than those specified in the XML schema definition. If you want to use the prefixes specified in the XML schema definition itself, click **Finish**.

15. On the Assign Prefixes panel, if you want the web service descriptor to use different prefixes than those specified in the XML schema definition, select the prefix you want to change and enter a new prefix. Repeat this step for each namespace prefix that you want to change.

Note:

The prefix you assign must be unique and must be a valid XML NCName as defined by the specification <http://www.w3.org/TR/REC-xml-names/#NT-NCName>.

16. Click **Finish**.

Designer creates the consumer web service descriptor and saves it to the specified folder. Designer also creates supporting IS elements, such as web service connectors, IS document types, and response services and places them in the same folder. For more information about what elements Integration Server creates, see “[Supporting Elements for a Consumer Web Service Descriptor](#)” on page 37.

If Designer cannot create or cannot completely generate a consumer WSD, Designer displays error messages or warning messages.

If Integration Server determines that an XML Schema definition included in or referenced by the WSDL document is invalid or cannot be generated according to the selected content model compliance option, Designer displays the validation error message at the top of the Select Document Type Generation Options panel. Click **Cancel** to abandon this attempt to create a consumer web service descriptor. Alternatively, click **Back** to navigate to previous panels and change your selections.

Notes:

- If the WSDL document contains a construct supported on the earlier web services implementation introduced in Integration Server 7.1, but not on the current Web Services Stack, Designer gives you the option of creating the web service descriptor using the earlier

web services implementation. If the WSDL document contains any of the following, Designer prompts you to use the web services implementation introduced in 7.1:

- Mixed “use” values across bindings and operations referenced by services in the WSDL document.
- Mixed “style” values across bindings referenced by services in the WSDL.
- More than one operation with the same name in the same port type.
- Bindings that do not contain all of the operations declared in the port type.
- Services with multiple bindings that reference different port types.

If you create the web services descriptor using the earlier web services implementation, the **Pre-8.2 compatibility mode** property will be set to true for the resulting web service descriptor.

Note:

The **Pre-8.2 compatibility mode** property and the ability to run in pre-8.2 compatibility mode are deprecated as of Integration Server 10.4 due to the deprecation of the web services implementation introduced in Integration Server version 7.1.

- Integration Server does not create binders for unsupported bindings in the WSDL document. If the WSDL document does not contain any bindings supported by Integration Server, Integration Server does not create a consumer web service descriptor.
- When creating the document types for the consumer web service descriptor, Integration Server registers each document type with the complex type definition from which it was created in the schema. This enables Integration Server to provide derived type support for document creation and validation.
- Integration Server uses the internal schema parser to validate the XML schema definition associated with the WSDL document. If you selected the **Validate schema using Xerces** check box, Integration Server also uses the Xerces Java parser to validate the XML Schema definition. With either parser, if the XML Schema does not conform syntactically to the schema for XML Schemas defined in *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures* (which is located at <https://www.w3.org/TR/xmlschema11-1/>), Integration Server does not create an IS schema or an IS document type for the web service descriptor. Instead, Designer displays an error message that lists the number, title, location, and description of the validation errors within the XML Schema definition.

Note: Integration Server uses Xerces Java parser version Xerces-J 2.12.1-xml-schema-1.1. Limitations for this version are listed at <http://xerces.apache.org/xerces2-j/xml-schema.html>.

- When validating XML schema definitions, Integration Server uses the Perl5 regular expression compiler instead of the XML regular expression syntax defined by the World Wide Web Consortium for the XML Schema standard. As a result, in XML schema definitions consumed by Integration Server, the pattern constraining facet must use valid Perl regular expression syntax. If the supplied pattern does not use proper Perl regular expression syntax, Integration Server considers the pattern to be invalid.

Note:

If the `watt.core.datatype.usejavaregex` configuration parameter is set to true, Integration Server uses the Java regular expression compiler instead of the Perl5 regular expression compiler. When the parameter is true, the pattern constraining facet in XML schema definitions must use valid syntax as defined by the Java regular expression.

- If you selected strict compliance and Integration Server cannot represent the content model in the complex type accurately, Integration Server does not generate any IS document types or the web service descriptor.
- For an IS document type from a WSDL document, Designer displays the location of the WSDL in the **Source URI** property. Designer also sets the **Linked to source** property to true which prevents any editing of the document type contents. To edit the document type contents, you first need to make the document type editable by breaking the link to the source. However, Software AG does not recommend editing the contents of document types created from WSDL documents.
- The contents of an IS document type with a **Model type** property value other than “Unordered” cannot be modified.
- Operations and binders cannot be added, edited, or removed from a consumer web service descriptor.
- The Message Exchange Pattern (MEP) that Integration Server uses for an operation defined in the WSDL can be In-Out MEP, In-Only MEP, or Robust In-Only MEP. Integration Server always uses In-Out MEP when the web service descriptor’s **Pre-8.2 compatibility mode** property is true. When this property is false, Integration Server uses:
 - In-Out MEP when an operation has defined input and output.
 - In-Only MEP when an operation has no defined output and no defined fault. The web service connector that Integration Server creates will no SOAP message-related output parameters and, when executed, will not return output related to a SOAP response.
 - Robust In-Only MEP when an operation has no defined output, but has a defined fault. The web service connector that Integration Server creates will return no output related to a SOAP response if the operation executes successfully. However, if an exception occurs, the web service connector returns the SOAP fault information as output.





For more information about Integration Server MEP support, see the section *About Consumer Web Service Descriptors* in the *Web Services Developer’s Guide* .






- Integration Server creates response services for all In-Out and Robust In-Only MEP operations in the WSDL document.
- When creating a web service descriptor from a WSDL document, Integration Server treats message parts that are defined by the type attribute instead of the element attribute as an error and does not allow the web service descriptor to be created. You can change this behavior by setting the `watt.server.SOAP.warnOnPartValidation` parameter to true. When this parameter is set to true, Integration Server will return a warning instead of an error and will allow the web service descriptor to be created.
- If the WSDL document is annotated with WS-Policy:

- Integration Server enforces the annotated policy at run time. However, if you attach a policy from the policy repository to the web service descriptor, the attached policy will override the original annotated policy.
- Integration Server will only enforce supported policy assertions in the annotated policy. Currently Integration Server supports only WS-Security policies.
- Integration Server does not save the annotated policy in the policy repository.
- If an XML Schema definition referenced in the WSDL document contains the `<!DOCTYPE` declaration, Integration Server issues a `java.io.FileNotFoundException`. To work around this issue, remove the `<!DOCTYPE` declaration from the XML Schema definition.
- When creating a consumer web service descriptor from an XML Schema definition that imports multiple schemas from the same target namespace, Integration Server throws Xerces validation errors indicating that the element declaration, attribute declaration, or type definition cannot be found. The Xerces Java parser honors the first `<import>` and ignores the others. To work around this issue, you can do one of the following
 - Combine the schemas from the same target namespace into a single XML Schema definition. Then change the XML schema definition to import the merged schema only.
 - When creating the consumer web service descriptor, clear the **Validate schema using Xerces** check box to disable schema validation by the Xerces Java parser. When generating the web service descriptor, Integration Server will not use the Xerces Java parser to validate the schemas associated with the XML Schema definition.

Supporting Elements for a Consumer Web Service Descriptor

When Designer creates a consumer web service descriptor, it also creates supporting IS elements. Each of the IS elements that Designer creates corresponds to an element in the WSDL document.

This IS element...	Corresponds to this WSDL element...
 <i>consumerWsdName_</i>	The WSDL document. All supporting IS elements for the consumer web service descriptor are contained in this new folder. The folder name is the same as the web service descriptor, with a suffix of an “_” (underscore).
 docType folder	All of the IS document types generated from the messages in the WSDL document.
 connectors folder	All of the web service connectors generated from the operations in the WSDL document.
 responseServices folder (Available in Integration Server version 9.0 and later only.)	All of the response services generated from the operations in the WSDL document and the genericFault_Response service.

This IS element...	Corresponds to this WSDL element...
 Web service connector	Each unique <operation> element in a <portType> element; the web service connector name corresponds to the portType name and operation name.
 IS document type	Each <message> element in the WSDL document. The IS document type name corresponds to the message name.
 IS schema	Each target namespace to which the element declarations, attribute declarations, and type definitions that define the message parts (input and output signature) belong. Note: Integration Server assigns any IS schemas to a unique schema domain for that web service descriptor.
 Response service (Available in Integration Server version 9.0 and later only.)	Each unique <operation> element in a <portType> element; the response service name corresponds to the portType name and operation name, with a suffix of “_Response”.
 genericFault_ Response service	The default response service that Integration Server invokes when Integration Server cannot determine the specific response service for an asynchronous SOAP response or if there are errors while processing the response.

Note:

The *consumerWSDName_* folder and its subfolders docTypes, connectors, and responseServices are reserved for elements created by Integration Server for the web service descriptor only. Do not place any custom IS elements in these folders.

About Web Service Connectors

A web service connector is a flow service that Integration Server creates at the time it creates the consumer web service descriptor. A web service connector contains the information and logic needed to invoke an operation defined in the WSDL document used to create the consumer web service descriptor.

When creating a consumer web service descriptor from a WSDL document, Integration Server creates a web service connector for each operation and portType combination contained in the WSDL document. For example, if a WSDL document contains two portType declarations and each portType contains three operations, Integration Server creates six web service connectors.

A web service connector:

- Uses an input and output signature that corresponds to the input message, output message, and headers defined for the operation in the WSDL document. The web service connector signature also contains optional inputs that you can use to control the execution of logic in the web service connector.

- Represents a SOAP fault structure in the output signature differently based on the version of the Integration Server on which the web service descriptor is created. To learn more about the output signature of a web service connector, see *Web Services Developer's Guide*.
- Contains flow steps that create and send a message to the web service endpoint using the transport, protocol, and location information specified in the web service's WSDL document in conjunction with input supplied to the web service connector.
- Contains flow steps that extract data or fault information from the response message returned by the web service.

Important:

Do not edit the flow steps in a web service connector.

Note:

A web service connector that worked correctly with previous versions of Developer, Designer, and Integration Server should continue to work with version 8.2 and later. In addition, any external clients created from WSDL generated from previous versions of Developer and Integration Server should continue to work as they did in the previous version.

For detailed information about a web service connector, such as a description of the web service connector signature, see the *Web Services Developer's Guide*.

Refreshing a Web Service Connector

When you create a consumer web service descriptor, Designer automatically generates the web service connector(s). You must refresh the web service connectors after you have added, deleted, or modified any of the following.

- Header
- Fault
- Endpoint alias within a binder for a consumer web service descriptor

Refreshing the web service connectors overwrites all the existing web service connectors for a consumer web service descriptor.


Keep the following points in mind when refreshing a web service connector:

- When refreshing a web service connector, Integration Server deletes the *consumerWSDName_* folder and all elements contained in that folder and its subfolders. Integration Server will not recreate any elements manually added to the folder or its subfolders. Integration Server will not recreate modifications made to any of the original elements in the *consumerWSDName_* folder.
- Refreshing a web service connector does not change the structure of the fault in the output signature. That is, when you refresh a web service connector for a web service descriptor created using an Integration Server version prior to 8.2, the web service connector output signature retains the SOAP fault document that is specific to the SOAP protocol (i.e., SOAP 1.1 or SOAP 1.2). Similarly, if you refresh a web service connector for a web service descriptor created using Integration Server 8.2, the web service connector's output signature will continue

to have the generic SOAP fault structure. For more information about how the output signature of web service connector depends on the version of the Integration Server on which the web service descriptor is created, see *Web Services Developer's Guide*.

- If the **Validate Schema using Xerces** property is set to true for a web service descriptor, Integration Server validates the schemas associated with a consumer web service descriptor when you refresh the web service connector.
- Refreshing web service connectors is different than refreshing a web service descriptor. When refreshing web service connectors, Integration Server uses the original WSDL document to recreate the web service connectors and the contents of the *consumerWSDName_* folder. When refreshing a web service descriptor, Integration Server uses an updated version of the WSDL document to regenerate the web service descriptor and its associated IS elements. For more information about refreshing web service descriptors, see [“About Refreshing a Web Service Descriptor” on page 41](#).
- If you are using the local service development feature, using versions of Subversion prior to 1.7 as your VCS client might cause issues while refreshing web service connectors. Software AG recommends that you use Subversion 1.7 or higher as your VCS client.

➤ To refresh a web service connector

1. In Package Navigator view, open the consumer WSD for which you want to refresh web service connectors.
2. Click the Operations tab or the Binders tab.
3. Click  or right-click and select **Refresh Web Service Connectors**.

Integration Server regenerates *all* web service connectors in the consumer WSD, overwriting the existing web service connectors.

Invoking a Web Service Using a Web Service Connector

To invoke a web service, or more specifically, an operation in a web service, create a flow service that invokes the web service connector that corresponds to the operation you want to use. Because a web service connector is a flow service, you invoke the web service connector in the same way in which you would a regular flow service.

Note:

If the web service connector uses a JMS binding to send a message using SOAP over JMS, you can specify how Integration Server proceeds when the JMS provider is not available at the time the message is sent. For more information, see [“Configuring Use of the Client Side Queue” on page 67](#).

About Response Services

In versions 9.0 and later, Integration Server creates a `responseServices` folder along with `connectors` and `docTypes` folders when you create a consumer web service descriptor.

The `responseServices` folder contains a response service for each In-Out and Robust-In-Only MEP operation in the WSDL document from which the consumer web service descriptor is created. Response services are flow services to which you can add custom logic to process asynchronous SOAP responses. Integration Server creates the response services only if the consumer web service descriptor:

- Is created on Integration Server version 9.0 or later.
- Has the **Pre-8.2 compatibility mode** property set to false.

Integration Server invokes the response services for processing asynchronous SOAP responses received for the associated consumer web service descriptor. That is, Integration Server invokes a response service when Integration Server receives a SOAP response with the endpoint URL pointing to a consumer web service descriptor and if this SOAP response contains a WS-Addressing action through which the response service can be resolved.

The `responseServices` folder also contains a `genericFault_Response` service, which is the default response service that Integration Server invokes when Integration Server cannot determine the specific response service for a SOAP response or if there are errors while processing the response.

For more information about response services and how Integration Server processes responses asynchronously, see the *Web Services Developer's Guide*

About Refreshing a Web Service Descriptor

If the WSDL document used to create a web service descriptor changes, you may want to refresh the web service descriptor to reflect the recent changes. For example, if you created a WSDL first provider web service descriptor from a WSDL document that has since changed to include a new operation or new input/output messages, you can refresh the web service descriptor. Refreshing the web service descriptor does the following:

- Updates the web service descriptor or its associated IS elements to reflect changes in existing elements in the updated WSDL document.
- Adds elements, such as operations, headers, or binders, to the web service descriptor to reflect new elements in the updated WSDL document.
- Adds new IS elements, such as IS document types, IS schemas, and services, that correspond to new elements in the updated WSDL document.
- Removes web service descriptor elements or IS elements that correspond to elements that have been removed from the updated WSDL document.
- Preserves any changes you made to the web service descriptor since it was created from the original WSDL document.

Refreshing a web service descriptor is different than refreshing web service connectors. When refreshing a web service descriptor, Integration Server uses an updated version of the WSDL document to regenerate the web service descriptor and its associated IS elements. When refreshing web service connectors, Integration Server uses the original WSDL document to recreate the web service connectors and the contents of the *consumerWSDName_* folder. For more information about refreshing web service connectors, see [“Refreshing a Web Service Connector” on page 39](#).

The following table provides an overview of the activities involved in refreshing a web service descriptor.

Step	Description
1	<p>You select the web service descriptor that you want to refresh. You can refresh WSDL first provider web service descriptors or consumer web service descriptors created on Integration Server version 7.1 or later.</p> <div>Note: Service first provider web service descriptors are not created from a WSDL document and therefore cannot be refreshed.</div>
2	<p>You specify the location of the WSDL document that you want Integration Server to use when refreshing the web service descriptor. If the web service descriptor was created on Integration Server version 8.2 or later, Integration Server defaults to the WSDL document at the location specified by the Source URI property. If the web service descriptor was created before Integration Server version 8.2, Designer prompts you to select the location of the WSDL document to use as the source for the refresh.</p>
3	<p>Integration Server creates a backup copy of the web service descriptor and its associated elements, such as IS document types, IS schemas, services, and web service connectors. Integration Server uses the most recently saved version of the web service descriptor and its associated elements as the backup copy. Integration Server makes a backup copy in case it cannot refresh the web service descriptor successfully.</p>
4	<p>Integration Server regenerates the web service descriptor using the options specified in the New Web Service Descriptor wizard at the time the web service descriptor was first created. For example, if you specified Strict for the Content model compliance option, Integration Server uses the Strict option when refreshing.</p> <div>Note: If you want Integration Server to use different options than the original ones when refreshing the web service descriptor, do not refresh the web service descriptor. Instead, delete the web service descriptor and then recreate it using the updated WSDL document and the different New Web Service Descriptor wizard options.</div>
5	<p>During regeneration, Integration Server does the following:</p>

Step	Description
	<ul style="list-style-type: none"> ■ If an existing element in the WSDL document has been modified, Integration Server updates the corresponding elements in the web service descriptor or its associated IS elements. For example, suppose that a complex type definition changed and that type definition was used in the input message for an operation. During refresh, Integration Server would recreate the document type that corresponds to the input signature. ■ If the WSDL document includes new information or elements, Integration Server adds that information to the web service descriptor or one if its associated IS elements. For example, if the WSDL used to create a provider web service descriptor includes a new operation, Integration Server generates a new skeleton service and adds the service as an operation in the web service descriptor.
	<ul style="list-style-type: none"> ■ If information or an element has been removed from the updated WSDL document, Integration Server removes the corresponding web service descriptor information or associated IS element. For example, suppose that the original WSDL document included a header but the updated WSDL document does not include that header. During refresh, Integration Server removes the header from the web service descriptor and deletes the IS document type that corresponds to the original header. <p>Note: Integration Server considers a renamed element to be a new element. For example if the name of an operation changed from “myOperation” to “yourOperation”, Integration Server removes the operation “myOperation” from the web service descriptor. Integration Server creates a new service for “yourOperation” and adds that service to the web service descriptor as an operation.</p>
	<ul style="list-style-type: none"> ■ Integration Server merges in changes that you made since the web service descriptor was first created or since the last refresh. For example, if you added logic to a skeleton service generated for a WSDL first provider web service descriptor, Integration Server adds that logic to the refreshed service. If you added a header or fault to the web service descriptor, Integration Server adds that header or fault to the refreshed web service descriptor. <p>Note: When refreshing a web service descriptor, Integration Server does not preserve any changes made to IS document types or IS schemas that were generated from the WSDL document.</p> <ul style="list-style-type: none"> ■ Integration Server sets the properties of the refreshed web service descriptor to match the property values that were specified before the web service descriptor was refreshed. Integration Server also ensures that any services created from the original WSDL document have the same properties after the refresh.

Step	Description
6	Upon successful refresh of the web service descriptor, Integration Server deletes the backup copy of the web service descriptor and its associated elements. If Integration Server cannot successfully refresh the web service descriptor, Integration Server reverts to the backup copy.

How Refresh Affects a Web Service Descriptor

Integration Server handles the updating of each web service descriptor element or associated IS element differently depending on:

- The web service descriptor element, such as an operation, binder, or header.
- The type IS element.
- Whether the web service descriptor element or IS element changed since the web service descriptor was first created.
- Whether the updated WSDL document contains an element that corresponds to the web service descriptor element or IS element.

The following table provides details about how Integration Server handles specific IS elements during refresh.

For this element...	During refresh Integration Server...
IS document type	Deletes all of the document types that Integration Server generated from the WSDL document. Integration Server then creates new document types using the updated WSDL document. Any changes made to the original document types will be lost.
IS schema	Deletes all of the IS schemas that Integration Server generated from the WSDL document. Integration Server then creates new IS schemas using the updated WSDL document. Any changes made to the original IS schemas will be lost.
Service	Does one of the following for the skeleton services generated for operations in the original WSDL document: <ul style="list-style-type: none">■ If logic has been added to the skeleton service or service properties have been set and the corresponding operation exists in the updated WSDL document, Integration Server merges the logic into the refreshed service and ensures that the property values match the values set prior to refreshing.■ If logic has not been added to the skeleton service, service properties have not been set, and the corresponding operation exists in the updated WSDL document, Integration Server recreates the empty skeleton service.

For this element...	During refresh Integration Server...
	<ul style="list-style-type: none"> ■ If a service corresponds to an operation that does not exist in the updated WSDL document, Integration Server removes the operation that corresponds to the service from the web service descriptor. Integration Server keeps the service in the “services” folder. <p>Note: Integration Server considers a renamed operation to be a new operation.</p>
Web service connector	Deletes and recreates all web service connectors. Any changes made to a web service connector, including changes for pipeline mapping, will be lost.
connectors folder	Deletes the connectors folder and all elements contained in that folder and its subfolders. Integration Server will not recreate any elements manually added to the folder or its subfolders.
<i>consumerWSDName_</i> folder	Deletes the <i>consumerWSDName_</i> folder and all elements contained in that folder and its subfolders. Integration Server will not recreate any elements manually added to the folder or its subfolders. Integration Server will not recreate modifications made to any of the original elements in the <i>consumerWSDName_</i> folder.
docTypes folder	Deletes the docTypes folder and all elements contained in that folder and its subfolders. Integration Server will not recreate any elements manually added to the folder or its subfolders.
responseServices folder	<p>Does the following with the contents of the responseServices folder:</p> <ul style="list-style-type: none"> ■ For new operations in the updated WSDL document, Integration Server adds new response services. ■ For modified operations, Integration Server updates the response services including merging in any logic that was added. ■ For deleted operations, Integration Server removes the operation that corresponds to the service from the web service descriptor. Integration Server keeps the response service in the “responseServices” folder.
services folder	<p>Does the following with the contents of the services folder:</p> <ul style="list-style-type: none"> ■ Adds new skeleton services for new operations in the updated WSDL document. ■ For modified operations, updates the skeleton services including merging in any logic that was added. For more information, see the “Service” row in this table. ■ For deleted operations, Integration Server removes the operation that corresponds to the service from the web service descriptor. Integration Server keeps the service in the “services” folder.

The following table provides details about how refreshing a web service descriptor affects the contents of the web service descriptor itself.

For this web service descriptor element	During refresh Integration Server...
Binders	Updates the binders to reflect any new port information in the WSDL document. Integration Server also updates the operations for each binder to reflect any new or removed operations in the updated WSDL document. Integration Server updates the SOAP action values assigned to operations in binders to reflect any changes in the updated WSDL document.
Header/Fault	<p>For a header/fault defined in the WSDL document, Integration Server does one of the following:</p> <ul style="list-style-type: none"> ■ If the updated WSDL document does not contain the header/fault, Integration Server removes the header/fault from the web service descriptor. Integration Server also deletes the document types used to define the header/fault documents from the docTypes folder. ■ If the updated WSDL document contains the header/fault, Integration Server keeps the header/fault in the web service descriptor. Integration Server recreates the document types used to define the header/fault documents and places them in the docTypes folder. ■ If the updated WSDL contains a new header/fault, Integration Server adds the header/fault to the web service descriptor. Integration Server adds the new document types used to define the header/fault documents to the docTypes folder. <p>For a header/fault that was added manually after the web service descriptor was created, Integration Server adds the document types that define the header/fault to the refreshed web service descriptor.</p> <p>Note: If a header/fault that was added manually after the web service descriptor was created has the same name as a header/fault in the updated WSDL document, Integration Server replaces the manually added header/fault with the header/fault that Integration Server generates programmatically from the WSDL document. This might result in broken mappings or unexpected behavior in the handler service associated with the header/fault.</p>
Handler	Integration Server adds handlers defined in the previous version of the web service descriptor to the refreshed web service descriptor.
Operations	Updates the operations to reflect any new or removed operations in the updated WSDL document.

For this web service descriptor element	During refresh Integration Server...
Policy	Attaches any policies added in the previous version of the web service descriptor to the refreshed version of the web service descriptor.

Considerations for Refreshing a Web Service Descriptor

Because refreshing a web service descriptor involves deleting, recreating, and merging elements, it is possible that refreshing will result in broken mappings, services that do not execute to completion successfully, and other issues that need to be resolved. Before refreshing a web service descriptor, review the following considerations.

- Refresh a web service descriptor only if you are familiar with the original WSDL document, the changes in the updated WSDL document, and the web service descriptor. Designer does not provide a list of changes to the web service descriptor as part of the refresh. You will need to use your knowledge of the WSDL document changes and the web service descriptor to ensure that operations, services, pipeline mapping, and other aspects of the web service descriptor work as expected.
- During refresh, mappings between variables might break or be lost. This is particularly true when the web service descriptor has manually added headers or faults and the updated WSDL document has new headers or faults of the same name.
- During refresh of a consumer web service descriptor, Integration Server deletes and recreates the contents of the *consumerWSDName_*. This includes all of the document types, Integration Server schemas and web service connectors generated from the original WSDL document. Any changes made to these elements will be lost. For web service connectors, this includes maps (links) between variables in the pipeline, variables added to the pipeline, variables dropped from the pipeline, and values assigned to pipeline variables.
- During refresh of a WSDL first provider web service descriptor, Integration Server deletes and recreates the contents of the docTypes folder. Changes made to the IS document types and IS schemas generated from the original WSDL document will be lost.
- Because Integration Server deletes and recreates the contents of the *consumerWSDName_* folder, docTypes folder, connectors folder, and services folder during refresh, do not place any custom elements in these folders. These folders are reserved for elements created by Integration Server for the web service descriptor only. Before refreshing a web service descriptor, remove any custom elements from these folders.
- If you used an IS element created by Integration Server for the web service descriptor with another IS element that is not associated with the web service descriptor, refreshing the web service descriptor might break the other usages of the IS element. For example, suppose that you used an IS document type created for an input message as the input signature of a service not used as an operation in the web service descriptor. If the input messages is removed from the updated WSDL document upon refresh, the other service will have a broken reference. The service will reference a document type that no longer exists.

- If you refresh a WSDL first provider web service descriptor for which web service clients have already been created, the web service clients will need to be recreated. Consumers will need to recreate their web service client using the new WSDL document that Integration Server generates for the provider web service descriptor.
- During refresh, Integration Server regenerates the web service descriptor using the functionality and features available in the Integration Server version on which the web service descriptor was originally created. After refreshing the web service descriptor, the **Created on version** property value is the same version of Integration Server as before the refresh. Refreshing a web service descriptor on the latest version of Integration Server does not update the web service descriptor to include all the web service features and functionality available in the current version of Integration Server. If you want the web service descriptor to use the features available with the current version of Integration Server, delete the web service descriptor and recreate it using Designer and the current version of Integration Server.
- If you are using the local service development feature, using versions of Subversion prior to 1.7 as your VCS client might cause issues while refreshing web service connectors. Software AG recommends that you use Subversion 1.7 or higher as your VCS client.

Refreshing a Web Service Descriptor

To update a web service descriptor to use the latest version of a WSDL document, you can refresh the web service descriptor. Keep the following points in mind when you refresh a web service descriptor.

- You can refresh any consumer web service descriptor or WSDL first provider web service descriptor created on version 7.1 or later.
- To refresh a web service descriptor, you do not need to have Write access to the web service descriptor or any of its associated elements (document types, schemas, services, or web service connectors).
- When refreshing a web service descriptor Integration Server uses the same options you selected in the New Web Service Descriptor wizard when you originally created the web service descriptor. If you want to use different options, you must delete the web service descriptor and recreate it using the updated WSDL document.
- If the **Validate Schema using Xerces** property is set to true for a web service descriptor, Integration Server validates the schemas associated with the web service descriptor during refresh.
- Any pre-requisites that existed for generating the original web service descriptor apply to refreshing the web service descriptor. This includes, but is not limited to the following:
 - To refresh a web service descriptor whose source is in a UDDI registry, Designer must be configured to connect to that UDDI registry.
 - To refresh a web service descriptor whose source is a service asset in CentraSite, Designer must be configured to connect to CentraSite.

- Before you can refresh a web service descriptor from a WSDL document that contains a JMS binding, you must have at least one valid web service endpoint alias that specifies the JMS transport.

For information about pre-requisites for creating a WSDL first provider web service descriptor, see [“Creating a WSDL First Provider Web Service Descriptor” on page 23](#). For information about pre-requisites for creating a consumer web service descriptor, see [“Creating a Consumer Web Service Descriptor” on page 31](#).

- Refreshing a web service descriptor is different than refreshing web service connectors. For more information about refreshing web service connectors, see [“Refreshing a Web Service Connector” on page 39](#).
- Before refreshing a web service descriptor, review the information in [“Considerations for Refreshing a Web Service Descriptor” on page 47](#).

➤ To refresh a web service descriptor

1. In Package Navigator view, lock the web service descriptor that you want to refresh.
2. Right-click the web service descriptor and select **Refresh Web Service Descriptor**.
3. Review the informational message about potential changes to the existing web service descriptor and click **OK** to continue with refresh the web service descriptor.
4. If the **Source URI** property specifies a location for the original WSDL document, Designer asks you if you want to use a different source file for refreshing the web service descriptor. Click **Yes** to select a file at a new location as the source. Click **No** to use the file at the specified location as the source.

If you selected **No** to use the specified location and Designer cannot read the WSDL file at that location, Designer displays a message prompting you to cancel the refresh or to select a new source location. Click **Cancel** to cancel the refresh. Click **OK** to specify a new location.

5. If you indicated that you wanted to select a new source location in the previous step, or if the Source URI property has no value, in the Refresh Web Service Descriptor dialog box, select the location of the WSDL file do one of the following:

Select...	To refresh a web service descriptor using...
CentraSite	A service asset in CentraSite
File/URL	A WSDL document that resides on the file system or on the Internet.
UDDI	A WSDL document in a UDDI registry

6. Click **Next**.

7. If you selected **CentraSite** as the source, under **Select Web Service from CentraSite**, select the service asset in CentraSite that you want to use to create the web service descriptor. Click **Next**.

Designer filters the contents of the Services folder to display only service assets that are web services.

If Designer is not configured to connect to CentraSite, Designer displays the **CentraSite > Connections** preference page and prompts you to configure a connection to CentraSite.

8. If you selected **File/URL** as the source, do one of the following:
 - Enter the URL for the WSDL document. The URL should begin with `http://` or `https://`.
 - Click **Browse** to navigate to and select a WSDL document on your local file system.
9. If you selected **UDDI** as the source, under **Select Web Service from UDDI Registry**, select the web service from the UDDI registry. Click **Next**.

If Designer is not currently connected to a UDDI registry, the Open UDDI Registry Session dialog box appears. Enter the details to connect to the UDDI registry and click **Finish**.

10. Click **Next** if you want to specify different prefixes than those specified in the XML schema definition. If you do not want to change the prefixes specified in the XML schema definition, click **Finish**.
11. On the Assign Prefixes panel, if you want the web service descriptor to use different prefixes than those specified in the XML schema definition or modified at the time of creating the web service descriptor, select the prefix you want to change and enter a new prefix. Repeat this step for each namespace prefix that you want to change.

Note:

The prefix you assign must be unique and must be a valid XML NCName as defined by the specification <http://www.w3.org/TR/REC-xml-names/#NT-NCName>.

12. Click **Finish**.

Designer refreshes the web service descriptor. If Designer cannot refresh a web service descriptor, Designer rolls back to the last saved version of the web service descriptor. If refresh is not successful, use the messages returned by Designer and the messages in the error log to determine why.

Viewing the WSDL Document for a Web Service Descriptor

On the WSDL tab, you can view the WSDL document associated with a consumer or provider web service descriptor.

- For a consumer web service descriptor, the WSDL document is a local copy of the original WSDL document used to create the consumer web service descriptor with the following changes:
 - The addition of any headers or faults added to the consumer web service descriptor.
 - Modifications to the editable properties of the provider web service descriptor or its constituents, such as the use of a web service endpoint alias.
- For a service first provider web service descriptor, the WSDL document contains all the information a consumer needs to create a web service client that invokes the operations described in the WSDL.
- For a WSDL first provider web service descriptor, the WSDL document is the original source WSDL with the following changes:
 - Addition of any headers or faults added to the provider web service descriptor.
 - Modifications to the editable properties of the provider web service descriptor or its constituents, such as the use of a web service endpoint alias.
 - Modification to the name attribute in the `wsdl:service` element to reflect the name of the web service descriptor.
 - Removed all `soapjms` `wsdl` extensions for JMS bindings contained in the original `wsdl:port`, `wsdl:service`, or `wsdl:binding` elements.
 - Addition of `soapjms` `wsdl` extensions to the `wsdl:binding` element for JMS bindings. This includes JMS transport properties defined in the web service endpoint alias assigned to the binder that specifies the JMS transport.
 - Changes to the `location` attribute of the `wsdl:port` element to reflect any JMS connection related settings for the JMS URI.
- The displayed WSDL document contains all the information a consumer needs to create a web service client that invokes the operations described in the WSDL.
- For a web service descriptor created from a WSDL that contains relative URIs that are anonymously addressable, Integration Server replaces any relative URIs with an absolute URI using the base URI of the WSDL file.
- If you attach a WS-Policy to a provider web service descriptor that is *not* running in pre-8.2 compatibility mode (i.e., the **Pre-8.2 compatibility mode** property is set to false), the generated WSDL will be annotated with the policy. If you attach multiple policies to a web service descriptor, the generated WSDL will have policy annotations of all the attached policies. The policy is annotated using `PolicyURI`s attributes. Integration Server identifies the associated policies by specifying the policy IDs in the `PolicyURI`s attributes.

You must save the web service descriptor before the policy annotations will be included in the WSDL. If the web service descriptor is not saved after a policy is attached to it, the WSDL on the WSDL tab will not yet include the policy annotations. When you save the web service descriptor, Integration Server obtains the policy from the policy files so that Designer can display it in the generated WSDL.

- When viewing the WSDL for a WSDL first provider web service descriptor that was created from a policy annotated WSDL, the generated WSDL will be annotated with the *attached* policies. The generated WSDL will not include the annotated policy from which it was generated.
- For a consumer web service descriptor, the generated WSDL will always contain the original annotated policy from the source WSDL document.

➤ **To view the WSDL document for a web service descriptor**

1. In Package Navigator view, open the web service descriptor for which you want to view the WSDL document.
2. Click the WSDL tab.

Designer displays the WSDL document for the web service descriptor.

WS-I Compliance for Web Service Descriptors

The WS-I option specifies whether the web service descriptor enforces compliance with the *WS-I Basic Profile 1.1*, a set of guidelines for using web services specifications to maximize interoperability (including guidance for such core web services specifications such as SOAP, WSDL, and UDDI).

As an example, using the RPC/Encoded style and use is not supported by the WS-I profile. If a web service descriptor makes use of the RPC/Encoded style, and **WS-I compliance** is enabled, Designer displays indicating that the WSD is not compliant and prompts you to save the WSD as non-compliant.

Enforcing WS-I compliance also affects the contents and signature for operations in the WSD. For example, the use of multiple top-level fields is not supported in the WS-I profile; if a service (operation) in a provider WSD includes multiple top-level fields, Designer prompts you to save the WSD as non-compliant

The WS-I compliance option is set to **Yes** or **No** when you create a web service descriptor (No is the default). You can modify this option by changing the **WS-I compliance** property.

Note:

The WS-I profile only address the SOAP 1.1 protocol.

Modifying WS-I Compliance for a Web Service Descriptor

The WS-I compliance option is set to **Yes** or **No** when you create a web service descriptor (**No** is the default). You can modify this option by changing the **WS-I compliance** property. Keep the following points in mind when determining whether to enforce WS-I compliance:

- The WS-I profiles only address the SOAP 1.1 protocol. If the web service descriptor is using the SOAP 1.2 protocol, Designer will display an error message when **True** is selected

- WS-I Basic Profile 1.0 supports only HTTP or HTTPS bindings. Consequently, WS-I compliance cannot be enforced if the WSDL contains a SOAP over JMS binding. The **WS-I compliance** property cannot be set to true if a web service descriptor has a JMS binder.

➤ **To modify WS-I compliance for a web service descriptor**


1. In Package Navigator view, open and lock the web service descriptor for which you want to change WS-I compliance enforcement.
2. In the Properties view, next to **WS-I compliance**, select **True** if you want Integration Server to enforce WS-I Basic Profile 1.1 compliance. Otherwise, select **False**.
3. Click **File > Save**.

Reporting the WS-I Profile Conformance for a Web Service Descriptor

You can analyze a web service descriptor for conformance to the WS-I Basic Profile 1.1.

To analyze whether the web service descriptor is WS-I Compliant, you must be connected to the Internet. To enable connecting to the Internet, ensure that you provide the appropriate proxy server settings in **Window > Preferences > General > Network Connections**. For more information about setting the proxy server details, see *Software AG Designer Online Help*.

➤ **To analyze a web service descriptor for WS-I Profile Conformance**

1. In Package Navigator view, open and lock the web service descriptor that you want to analyze for WS-I Profile Compliance.
2. Click  to check whether the web service descriptor is WS-I Profile Compliant.

If the web service descriptor is WS-I Compliant, Designer displays a confirmation message.

If the web service descriptor is not WS-I Compliant, Designer displays the error details in the Problems view.

Changing the Target Namespace for a Web Service Descriptor

For a service first provider web service descriptor, you can change the target namespace that Integration Server uses in the generated WSDL document. The target namespace specifies the XML namespace to which the elements, attributes, and type definitions in the WSDL belong.

You can only change the target namespace for service first provider WSD. For a WSDL first provider WSD or a consumer WSD, the target namespace is determined by the WSDL document used as the source.

➤ **To change the target namespace for a service first provider web service descriptor**

1. In Package Navigator view, open and lock the service first provider WSD for which you want to change the target namespace.
2. In the Properties view, in the **Target namespace** field, specify the URL that you want to use as the target namespace for elements, attributes, and type definitions in the WSDL generated for this provider WSD.
3. Click **File > Save**.

Viewing the Namespaces Used within a WSDL Document

You can view a list of all the XML namespaces used by the web service descriptor when it is first created. You can also view the prefix associated with each XML namespace.

➤ **To view the namespaces and prefixes used in a web service descriptor**

1. In Package Navigator view, open the service first provider WSD for which you want to view the list of XML namespaces used in the original WSDL document.
2. In the Properties view, click the browse button in the **Namespaces** field.

The Namespaces dialog box appears displaying a list of XML namespaces used within the WSDL document. This is an array of namespace prefixes and their associated XML namespace names. This information is not editable.

3. Click **OK** to close the dialog box.

Enabling MTOM/XOP Support for a Web Service Descriptor

The Message Transmission Optimization Mechanism (MTOM) feature provides optimization of binary message transportation using XOP (XML-binary Optimized Packaging). If attachments are enabled for the web service descriptor, instances of XML-type base64Binary are transported using MIME attachments, which improves the performance of large binary payload transport. Integration Server supports SOAP attachments only for web service descriptors that specify style/use of RPC/Literal or Document/Literal.

Integration Server supports streaming the SOAP attachments based on the MTOM/XOP standards for both inbound and outbound messages. For more information about the configuration required to enable MTOM streaming, see the *Web Services Developer's Guide*.

Enabling SOAP Attachments for a Web Service Descriptor

Before enabling SOAP attachments for a web service descriptor, configure the `watt.server.SOAP.MTOMThreshold` property with the appropriate value. This property specifies

the field size, in kilobytes, that determines whether Integration Server sends base64binary encoded data in an outbound SOAP message as a MIME attachment or whether it sends it inline in the SOAP message. For more information about this property, see *webMethods Integration Server Administrator's Guide*.

If you want to stream MTOM attachments, you have to do additional configuration. For more information about the configuration required to enable MTOM streaming, see the *Web Services Developer's Guide*.

➤ To enable SOAP attachments for a web service descriptor

1. In Package Navigator view, open and lock the web service descriptor for which you want to enable or disable SOAP attachments.
2. In the Properties view, next to **Attachment enabled**, select **True** if you want to enable SOAP attachments for the WSD. Otherwise, select **False**.
3. Click **File > Save**.

Using pub.string:base64Encode with MTOM Implementations

By default, the public service `pub.string:base64Encode` inserts a new line after 76 characters of data. This is not the canonical lexical form expected by MTOM implementations. If you use this public service rather than a custom service for base64 encoding, you can use the optional input parameter *useNewLine* to remove the line break and the optional input parameter *encoding* to change the encoding. The default value for *useNewLine* is true, which keeps the line break at 76 characters. If you do not specify the encoding, ASCII will be used.

If you use the public service `pub.string:base64Decode` for base64 decoding, you can use the optional input parameter *encoding* to change the encoding. If you do not specify the *encoding*, ASCII will be used.

Adding SOAP Headers to the Pipeline

For a web service descriptor, you can instruct Integration Server to add the contents of SOAP headers to the pipeline, making the contents of the SOAP headers available to subsequent services. The value of the **Pipeline headers enabled** property determines whether or not Integration Server places the contents of the SOAP header in the pipeline as a document named *soapHeaders*. The default value is false.

If the **Pipeline headers enabled** property is set to true for a provider WSD, when an IS service that corresponds to an operation in the WSD is invoked, Integration Server places the contents of the SOAP request header in the input pipeline for the IS service.

If the **Pipeline headers enabled** property is set to true for a consumer WSD, when one of the web service connectors is invoked, Integration Server places the contents of the SOAP response header in the output pipeline for the web service connector.

For detailed information about the content and structure of the *soapHeaders* document that Integration Server adds to the pipeline, see *Web Services Developer's Guide*.

Note:

For web service descriptors contained in packages created in versions of Integration Server prior to 8.0, the **Pipeline headers enabled** property is set to **True**.

➤ To add SOAP headers to the pipeline

1. In Package Navigator view, open and lock the web service descriptor for which you want to enable or disable adding SOAP headers to the pipeline.
2. In the Properties view, next to **Pipeline headers enabled**, select **True** if you want to enable SOAP headers for the web service descriptor. Otherwise, select **False**.
3. Click **File > Save**.

Validating SOAP Response

For a consumer web service descriptor, you can indicate whether or not Integration Server validates a SOAP response received by any web service connectors within the consumer WSD.

The value of the `watt.server.SOAP.validateResponse` server configuration parameter determines whether or not the **Validate SOAP response** property is honored or ignored.

- When `watt.server.SOAP.validateResponse` is set to **true**, the value of the **Validate SOAP response** property determines whether or not Integration Server validates the SOAP response.
- When `watt.server.SOAP.validateResponse` is set to **false** (or anything besides **true**), Integration Server ignores the **Validate SOAP response** property and does not validate the SOAP response.

By default, the `watt.server.SOAP.validateResponse` is set to **true**.

➤ To validate SOAP responses received by web service connectors

1. In Package Navigator view, open and lock the consumer WSD for which you want to enable or disable SOAP response validation.
2. In the Properties view, next to **Validate SOAP response**, select **True** if you want Integration Server to validate SOAP response messages received by web service connectors included with this consumer WSD. Otherwise, select **False**.
3. Click **File > Save**.

Validating Schemas Associated with a Web Service Descriptor

To help ensure interoperability between a web service descriptor and other web service vendors or clients, you can use Integration Server to validate the schemas associated with the web service descriptor. Integration Server provides an internal schema parser that it uses to validate XML schema definitions at the following times:

- When you create or refresh a consumer web service descriptor or WSDL first provider web service descriptor from a WSDL document.
- When you change the IS schemas, document types, or signatures of the services associated with a web service descriptor.

While Integration Server uses an internal schema parser to validate the schemas automatically, you can also instruct Integration Server to use the Xerces Java parser. The Xerces Java parser provides stricter validation than that provided by the Integration Server internal schema parser. As a result, some schemas that the internal schema parser considers to be valid might be considered invalid by the Xerces Java parser.

Integration Server uses the Xerces Java parser to validate the schemas associated with a web service descriptor at the following times:

- When you create a consumer web service descriptor or WSDL first provider web service descriptor from a WSDL document. In the New Web Service Descriptor wizard, Designer provides an option named **Validate Schema using Xerces**. When selected, Integration Server validates the schemas defined or referenced in the WSDL document. If the Xerces Java parser determines the schema(s) are invalid, Integration Server does not create the web service descriptor and Designer displays the validation errors.
- When you refresh a consumer web service descriptor or WSDL first provider web service descriptor for which the **Validate Schema using Xerces** was selected at the time the web service descriptor was created.
- When you create a service first provider web service descriptor. In the New Web Service Descriptor wizard, Designer provides an option named **Validate Schema using Xerces**. When selected, as part of creating a service first provider web service descriptor, Integration Server converts the signatures of the services used as operations to XML schema elements. Then Integration Server uses the Xerces Java parser to validate the schema elements. If the Xerces Java parser determines the schema(s) are invalid, Integration Server does not create the web service descriptor and Designer displays the validation errors.
- When the **Validate Schema using Xerces** property is set to true for a web service descriptor and one of the following occurs:
 - You change the IS schemas, document types, or signatures of the services associated with a web service descriptor.
 - You select an element declaration from an XML Schema definition to use as the input or output signature of a 6.5 SOAP-MSG style operation. For more information about using 6.5 SOAP-MSG style services as operations, see [“Using a 6.5 SOAP-MSG Style Service as an Operation” on page 71](#).

- You refresh the web service connectors for a consumer web service descriptor.

While validation by the Xerces Java parser can increase the time it takes to create, update, or refresh a web service descriptor and increase the time to refresh or update a web service connector, using stricter validation can help ensure interoperability with other web service vendors.

Note: Integration Server uses Xerces Java parser version Xerces-J 2.12.1-xml-schema-1.1. Limitations for this version are listed at <http://xerces.apache.org/xerces2-j/xml-schema.html>.

When validating XML schema definitions, Integration Server uses the Perl5 regular expression compiler instead of the XML regular expression syntax defined by the World Wide Web Consortium for the XML Schema standard. As a result, in XML schema definitions consumed by Integration Server, the pattern constraining facet must use valid Perl regular expression syntax. If the supplied pattern does not use proper Perl regular expression syntax, Integration Server considers the pattern to be invalid.

Note:

If the `watt.core.datatype.usejavaregex` configuration parameter is set to true, Integration Server uses the Java regular expression compiler instead of the Perl5 regular expression compiler. When the parameter is true, the pattern constraining facet in XML schema definitions must use valid syntax as defined by the Java regular expression.

Enabling Xerces Schema Validation for a Web Service Descriptor

When you create a web service descriptor from a WSDL document, you can specify that Integration Server use the Xerces Java parser to validate the schemas associated with the WSDL document. Once the web service descriptor exists, you can indicate that Integration Server validate the schemas associated with the web service descriptor by setting the **Validate Schema using Xerces** property to true. When the **Validate Schema using Xerces** property is set to true, Integration Server validates the schemas associated with an existing web service descriptor in the following situations:

- You change the IS schemas, document types, or signatures of the services associated with a web service descriptor

Note: Integration Server uses the internal schema processor to validate the schemas at this point as well.

- You select an element declaration from an XML Schema definition to use as the input or output signature of a 6.5 SOAP-MSG style operation. For more information about using 6.5 SOAP-MSG style services as operations, see [“Using a 6.5 SOAP-MSG Style Service as an Operation” on page 71](#).
- You refresh the web service connectors for a consumer web service descriptor.

Integration Server sets the **Validate Schema using Xerces** property to true for all new web service descriptors. If you migrated a web service descriptor from a previous version of Integration Server, the migration utility set the value based on the version of Integration Server from which the web service descriptor was migrated.

- If the web service descriptor was migrated from Integration Server version 7.1.x, the migration utility set the **Validate Schema using Xerces** property to true.
- If the web service descriptor was migrated from Integration Server version 8.x, the migration utility used the value of the `watt.server.wsdl.validateWSDLSchemaUsingXerces` parameter to determine the value of the **Validate Schema using Xerces** property. If the parameter was set to true, the migration utility set the property to true. If the parameter was set to false, the migration utility set the property to false.

Note:

The `watt.server.wsdl.validateWSDLSchemaUsingXerces` parameter was removed in Integration Server version 9.0.

➤ **To enable or disable schema validation by the Xerces Java parser**

1. In Package Navigator view, open and lock the web service descriptor for which you want to enable or disable schema validation by the Xerces Java parser.
2. In the Properties view, next to **Validate schema using Xerces**, select **True** if you want Integration Server to use the Xerces Java parser to validate the XML Schema definitions associated with the web service descriptor. Otherwise, select **False**. The default is **True**.
3. Click **File > Save**.

Working with Binders

A *binder* is a webMethods term for a collection of related definitions and specifications for a particular port. The binder is a container for the endpoint address, WSDL binding element, transport protocol, and communication protocol information. Designer creates at least one binder when it generates the web service descriptor based on the data in the WSDL or IS service. The Binders tab displays the binders defined for a web service descriptor.

You can add new binder definitions to a service first provider web service descriptor. Binders cannot be added to a WSDL first provider web service descriptor or a consumer web service descriptor.

You can define a separate binder for each combination of endpoint address and protocol information that you want the service first provider web service descriptor to support.

Binders and Mixed Use

Integration Server and Designer do not support mixed “use” across binders and operations in a single web service descriptor. That is, the binders in a web service descriptor must specify the same value for the **SOAP binding use** property.

Integration Server and Designer enforce this restriction in the following way:

- In a service first provider web service descriptor, the first binder determines the “use” for all subsequent binders. If the first binder specifies a **SOAP binding use** of “literal”, any additional binder added to the provider web service descriptor must specify literal as the **SOAP binding use**.
- When creating a WSDL first provider web service descriptor or a consumer web service descriptor from a WSDL document, Integration Server will not create the web service descriptor if the WSDL document contains bindings with different use value or operations with different use values. Integration Server throws the following exception:

```
[ISS.0085.9285] Bindings or operations with mixed "use" are not supported.
```

Existing Web Service Descriptors with Mixed Use Binders

Integration Server continues to support existing web service descriptors that contain binders with mixed use as long as the web service descriptors are not modified. Once the web service descriptor is modified, Designer will not save the web service descriptor if it has mixed use binders.

To edit and save an existing provider web service descriptor with mixed binders, create separate provider web service descriptors for each binder use. For example, if a provider web service descriptor contains binder1 which specifies a “use” of literal and binder2 which specifies a “use” of encoded, copy the provider web service descriptor. In the provider web service descriptor copy, remove binder1. In the original provider web service descriptor, remove binder2. The provider web service descriptors can then be saved.

Binders and Mixed Style

Integration Server and Designer do not support mixed “style” across binders in a single web service descriptor if the web service descriptor does not run in pre-8.2 compatibility mode. That is, the binders in a web service descriptor for which the **Pre-8.2 compatibility mode** property is set to false must specify the same value for the **SOAP binding style** property.

Integration Server and Designer enforce this restriction in the following way:

- In a service first provider web service descriptor, the first binder determines the “style” for all subsequent binders. If the first binder specifies a **SOAP binding style** of “document”, any additional binder added to the provider web service descriptor must specify document as the **SOAP binding style**.
- When creating a WSDL first provider web service descriptor from a WSDL document, Integration Server will not create the web service descriptor if the services reference bindings with different styles.
- When creating a consumer web service descriptor from a WSDL document, Integration Server will not create the web service descriptor if the services reference supported bindings that specify mixed style values.

Note:


The restriction on mixed binding styles across binders does not apply to web service descriptors that run in pre-8.2 compatibility mode.

Adding a Binder to Web Service Descriptor

Keep the following points in mind when creating a new binder:

- You can add a binder definition to a service first provider web service descriptor.
- All existing operations will be duplicated within the new binder.
- For a web service descriptor that runs in pre-8.2 compatibility mode (**Pre-8.2 compatibility mode** property is set to true), the new binder must specify the same “use” as the binder that already exists in the provider web service descriptor. For more information about mixed use in binders, see [“Binders and Mixed Use” on page 59](#).
- For a web service descriptor that does not run in pre-8.2 compatibility mode (**Pre-8.2 compatibility mode** property is set to false), the new binder must specify the same “style” and “use” as the binder that already exists in the provider web service descriptor. For more information about mixed styles in binders, see [“Binders and Mixed Style” on page 60](#).
- You can add a binder that specifies the JMS transport only if a valid provider web service endpoint alias exists for the JMS transport. For example, if the only web service endpoint alias that exists for JMS specifies a SOAP-JMS trigger that no longer exists, Integration Server does not consider the endpoint alias to be valid. Consequently, the endpoint alias cannot be assigned to a JMS binder. For more information about creating a web service endpoint alias, see *webMethods Integration Server Administrator's Guide*.
- You can only add a JMS binder to a web service descriptor that does not run in pre-8.2 compatibility mode (**Pre-8.2 compatibility mode** property is set to false).
- In a JMS binder for a provider web service descriptor, the property values under JMS Settings and JMS Message Details are set by the web service endpoint alias assigned to the binder. The JMS Settings and JMS Message Details properties are read-only.
- If the **WS-I compliance** property is set to **True**, you can only create binders that comply with the WS-I profile.

➤ To add a binder to a service first provider web service descriptor

1. In the Package Navigator view in the Service Development perspective, open and lock the provider web service descriptor to which you want to add a binder.
2. In the Binders tab, click  on the web service descriptor toolbar or right-click and select **Add Binder**.
3. In the New Binder Options dialog box, specify the following information:

In this field...	Specify...
SOAP Version	Whether SOAP messages for this web service should use SOAP 1.1 or SOAP 1.2 message format.

In this field...	Specify...
Transport	<p>The transport protocol used to access the web service. Select one of the following:</p> <ul style="list-style-type: none">■ HTTP■ HTTPS■ JMS
Use and Style for Operations	<p>The style/use for operations in the provider web service descriptor. Select one of the following:</p> <ul style="list-style-type: none">■ Document - Literal■ RPC - Literal■ RPC - Encoded
Endpoint	<p>The address at which the web service can be invoked. Do one of the following:</p> <ul style="list-style-type: none">■ To use a provider web service endpoint alias to specify the address, select the Alias option. Then, in the Alias list, select the provider web service endpoint alias. Select <code>DEFAULT(aliasName)</code> if you want to use the information in the default provider web service endpoint alias for the address. If the Alias list includes a blank row, the Integration Server does not have a default provider web service endpoint alias for the protocol. Note: If a default provider endpoint alias is later set for the selected protocol, then Integration Server uses the information from the alias when constructing the WSDL document and during run-time processing.■ To specify a host and port as the address, select the Host option. Then, in the Host field specify the host name for the Integration Server on which the web service resides. In the Port field, specify an active HTTP or HTTPS listener port defined on the Integration Server specified in the Host field. Note: You can only specify Host and Port for the endpoint if a default provider endpoint alias does not exist for the selected protocol. When a default alias exists, Designer populates the Host and Port fields with the host and port from the default provider end point alias.

In this field...	Specify...
	<p>Note:</p> <p>If you selected JMS as the transport, you must specify an alias. After you select a provider web service endpoint alias, Designer displays the initial portion of the JMS URI that will be used as the address in the Port address (prefix) field.</p>
Directive	<p>The SOAP processor used to process the SOAP messages received by the operation in the provider web service descriptor. The Directive list displays all of the SOAP processors registered on the Integration Server. The default processor is ws - Web Services SOAP Processor.</p>

- Click **OK**. Designer adds the new binder to the Binders tab.
- Click **File > Save**.

Notes:

- If you specify HTTP or HTTPS as the transport, but do not specify a host, port, or provider web service endpoint alias and there is not a default provider endpoint alias for the transport protocol, Integration Server uses the primary port as the port in the endpoint URL. If the selected transport and the protocol of the primary port do not match, web service clients will not execute successfully. For more information see [“Protocol Mismatch Between Transport and Primary Port” on page 22](#).
- You can change the default name that Designer assigns to the binder. You can rename the binder by changing the value of the **Binder name** property or by selecting the new binder, right-clicking it, and selecting **Rename**.

Copying Binders Across Provider Web Service Descriptors

You can cut or copy an existing binder from another provider web service descriptor and paste it into the Binders tab of a service first provider web service descriptor. (Note that drag and drop of binders is not supported.)

Keep the following points in mind when pasting in a binder from another provider web service descriptor:

- The endpoint, directive, WS-I, transport, and use-style values are the same as those in the original (source) binder. You can modify these in the Properties view.
- All operations in the cut or copied binder are carried with it. If a pasted binder contains an operation that is not already in the web service descriptor, the operation is added to the web service descriptor.

Changing the Binder Transport

You can change the specified transport of a binder in a service first provider web service descriptor by changing the value of the binder **Transport** property. Keep the following points in mind when changing the binder transport:

- A binder can specify the JMS transport only if the web service descriptor does not run in compatibility mode (the **Pre-8.2 compatibility mode** property is set to false).
- You can change the transport to JMS only if a provider web service endpoint alias that specifies the JMS transport exists already.
- If you change the transport from HTTP or HTTPS to JMS, Designer automatically assigns the first valid provider web service endpoint alias that specifies the JMS transport to the **Port alias** property of the binder. If there is not valid endpoint alias for JMS, the binder transport cannot be changed. For example, if the only web service endpoint alias that exists for JMS specifies a SOAP-JMS trigger that no longer exists, Integration Server does not consider the endpoint alias to be valid. Consequently, the endpoint alias cannot be assigned to a JMS binder.

Additionally, Designer updates the **Port address** property to display the initial part of the JMS URI, specifically “jms”:<lookup var>:<dest>?targetService.


- If you change the transport from JMS to HTTP or HTTPS, Designer deletes the values of the **Port alias** property and the **Port address** property. If the Integration Server identifies a default provider endpoint alias for the protocol used by the binder, Designer sets **Port alias** property to DEFAULT(*aliasName*). If you want to use a specific web service endpoint alias to specify the hostname and port for the provider web service endpoint URL, make sure to specify that alias in the **Port alias** property.
- When you change the transport, Designer updates the **Binding type** and **SOAP binding transport** properties to match the selected transport.

Deleting a Binder from a Web Service Descriptor

You can delete a binder from a service first provider web service descriptor.

A web service descriptor must contain at least one binder. If you delete the last binder in a web service descriptor, you must add a new binder before the web service descriptor can be saved (the binder can be empty).

> To delete an operation from a binder in a provider web service descriptor

1. In the Package Navigator view in the Service Development perspective, open and lock the service first provider web service descriptor from which you want to delete a binder.
2. In the Binders tab, select the binder to delete.
3. Click  on the web service descriptor toolbar or right-click and select **Delete**.

4. Click **File > Save**.


Deleting an Operation from a Binder

If two binders share an operation, you can delete the operation from one binder but not the other. The operation will still be in the provider web service descriptor, but only be in one binder.

Keep the following in mind when deleting an operation from a binder:

- You can delete operations from a service first provider web service descriptor only.
- If other binders in the provider web service descriptor contain the operation, that operation remains in the web service descriptor.
- If an operation only exists in one binder, deleting it from that binder removes it entirely from the web service descriptor.
- If you delete an operation from the Operations tab, it is deleted entirely from the web service descriptor and from all the binders that exist for that web service descriptor. For more information about deleting operations, see [“Deleting Operations” on page 74](#).

➤ To delete an operation from a binder in a provider web service descriptor

1. In the Package Navigator view in the Service Development perspective, open and lock the service first provider web service descriptor.
2. In the Binders tab, expand the binder containing the operation to delete.
3. In the binder, select the operations you want to delete.
4. Click  on the web service descriptor toolbar or right-click and select **Delete**.

Designer deletes the selected operation from the binder.

5. Click **File > Save**.

Modifying the SOAP Action for an Operation in a Binder

For a service first provider web service descriptor, you can change the SOAP action specified for an operation in a binder. By default, the SOAP action uses the format *binderName_operationName* to ensure that it is unique within a web service. At run time, the values associated with a SOAP action are used to find the actual operation being invoked. For more information about how Integration Server determines which operation to invoke, see *Web Services Developer's Guide*.

➤ To modify the SOAP action property for an operation in a binder

1. In the Package Navigator view in the Service Development perspective, open and lock the provider web service descriptor.
2. In the Binders tab, select the binder containing the operation for which you want to edit the SOAP action.
3. In the Properties view, next to the **SOAP action** property, click the browse button. Designer displays the SOAP Action dialog box which identifies the SOAP action string associated with each operation in the selected binder.
4. For the operation whose SOAP action you want to change, enter the new SOAP action value in the **SOAP Action** column. Make sure that the new SOAP Action value is unique across the web service descriptor.
5. Click **OK**.

Designer applies the SOAP action change to the operation in this binder only.

6. Click **File > Save**.

Assigning a Web Service Endpoint Alias to a Binder

You can associate a web service endpoint alias with a binder in a provider or consumer web service descriptor. A web service endpoint alias represents the network address and, optionally, any security credentials to be used with web services. The network address properties can be used to enable dynamic addressing for web services. The security credentials can be used to control both transport-level and message-level security for web services.

For a consumer web service descriptor and its associated web service connectors (WSC), the alias information (including the addressing information and any security credentials), is used at run time to generate a request and invoke an operation of the web service. For web service connectors behind a firewall, the endpoint alias also specifies the proxy alias for the proxy server through which Integration Server routes the web service request. For more information about proxy server usage, see *webMethods Integration Server Administrator's Guide*.

For a provider web service descriptor, the endpoint alias is used to construct the "location=" attribute of the soap:address element within the wsdl:port element when WSDL is requested for the web service. The security credentials may be used when constructing a response to a web service request.

For information about creating a web service endpoint alias, see *webMethods Integration Server Administrator's Guide*.

To assign a web service endpoint alias to a binder

1. In the Package Navigator view in the Service Development perspective, open and lock the web service descriptor to which you want to associate the web service endpoint alias.

2. In the Binders tab, select the binder to which you want to assign an endpoint alias.
3. In the Properties view, next to the **Port alias** property, select the web service endpoint alias that you want to associate with the web service descriptor. Designer lists only those endpoint aliases of the same type as the web service descriptor and whose protocol matches the binder protocol. If there have been changes to the web service endpoint aliases since you connected Designer to Integration Server, use Designer to refresh the connection to Integration Server.

If this is a provider web service and the binder protocol is HTTP or HTTPS, you can assign the default provider endpoint alias to the binder. Select `DEFAULT(aliasName)` if you want to use the information in the default provider web service endpoint alias for the address. If the **Alias** list includes a blank row, Integration Server does not have a default provider web service endpoint alias for the protocol.

Note:

If you select the blank row and a default provider endpoint alias is later set for the selected protocol, Integration Server then uses the information from the alias when constructing the WSDL document and during run-time processing.

4. Click **File > Save**.

Notes:

- When the **Port alias** property is modified for a consumer web service descriptor and the web service descriptor is viewed on the WSDL tab, the generated WSDL does not reflect the change to the port alias. However, the new value will be used at run-time.
- After assigning an alias to a JMS binder in a provider web service descriptor, if the web service endpoint alias specifies a SOAP-JMS trigger, the web service descriptor has a dependency on the SOAP-JMS trigger. Consequently, at start up or when reloading the package containing the web service descriptor, Integration Server must load the SOAP-JMS trigger before loading the web service descriptor. If the SOAP-JMS trigger and web service descriptor are not in the same package, you need to create a package dependency. The package that contains the web service descriptor must have a dependency on the package that contains the SOAP-JMS trigger.

Configuring Use of the Client Side Queue

You can enable use of the client side queue for a JMS binder in a consumer web service descriptor. The client side queue is a message store that contains JMS messages sent during service execution when the JMS provider was not available. Each JMS connection alias has its own client side queue. When the JMS provider becomes available, Integration Server sends messages from the client side queue to the JMS provider.

When use of the client side queue is enabled for a JMS binder and the JMS provider is not available at the time a web service connector sends a message using the JMS binding, Integration Server writes the message to the client side queue.

When use of the client side queue is disabled for a JMS binder and the JMS provider is not available at the time the web service connector executes, Integration Server throws an `ISRuntimeException`.

Integration Server includes the exception in the *fault* document returned to the web service connector.

Keep the following points in mind when enabling use of the client side queue for a JMS binder:

- The client side queue associated with the JMS binder is determined by the JMS connection alias in the consumer web service endpoint alias for the binder. The maximum size of the client side queue must be greater than zero. If the JMS connection alias sets the size of the client side queue to zero (**Maximum Queue Size** is set to 0), the client side queue is effectively disabled. Integration Server will not write messages to a client side queue that has a maximum size of 0 messages. For more information about configuring a JMS connection alias, see *webMethods Integration Server Administrator's Guide*
- The client side queue can be used with web service connectors for In-Only and In-Out operations. For an In-Out operation, the reply to destination for the web service must be a non-temporary queue.

➤ To configure the use of the client side queue for a JMS binder

1. In the Package Navigator view in the Service Development perspective, open and lock the web service descriptor containing the binder for which you want to configure the use of the client side queue.
2. In the Binders tab, select the JMS binder for which you want to configure the use of the client side queue.
3. In the Properties view, next to the **Use CSQ** property, select **True** to enable use of the client side queue. If you do not want Integration Server to use the client side queue for JMS messages sent using the binding represented by this binder, select **False**.
4. Click **File > Save**.

Working with Operations

An *operation* is the WSDL element that exposes some functions of a web service and defines how data is passed back and forth. In a web service descriptor, an operation corresponds to a service on Integration Server.

Each operation contains a single request and a single response. Each request and response contains a single, read-only body element and one or more header elements. A response can also contain fault elements.

The *body elements* contain the application-defined XML data being exchanged in the SOAP message:

- In a service first provider WSD, the body elements represent the signature of the service. The body element in the request contains the input properties. The body element in the response contains the output properties.

- In a WSDL first provider WSD or a consumer WSD, the input/output properties and the body element are defined by the remote WSDL document. Neither the input/output definitions nor the operations can be changed, added, or deleted.

A *header element* defines the format of the SOAP headers that may be present in a SOAP message (request or response). Headers are optional and can be added to or deleted from any web service descriptor.

A *fault element* provides a definition for a SOAP fault (that is, the response returned to the sender when an error occurs while processing the SOAP message). Fault elements are optional and can be added to or deleted from any web service descriptor.

Adding Operations

When you add operations to a service first provider WSD, the operations are also added to every binder in the WSD. The values defined by a specific binder will apply to the operation.

Note:

You can add operations to a service first provider WSD only.

You can add operations by:

- Adding one or more IS services from the Package Navigator. Each service will be converted to an operation in the provider WSD.
- Copying or moving an operation from another provider WSD.
- Adding a 6.5 SOAP-MSG Style Service as an operation.


Adding an IS Service as an Operation

Keep the following points in mind when adding an IS service as an operation to an provider web service descriptor:

- You can add operations to a service first provider WSD only.
- A 6.5 SOAP-MSG style service can only be added as an operation if it meets the requirements identified in [“Using a 6.5 SOAP-MSG Style Service as an Operation” on page 71](#).
- Because Integration Server and Designer do not support mixed “use” across binders and operations and mixed “style” across binders in a single web service descriptor, the service signature must meet the style/use signature requirements established by the existing binder. For more information, see [“Service Signature Requirements for Service First Provider Web Service Descriptors” on page 16](#).

➤ To add an IS service to a service first provider web service descriptor

1. In Package Navigator view, open and lock the service first provider WSD to which you want to add an IS service as an operation.

2. On the web service descriptor editor toolbar, click  or right-click in the Operations tab and select **Add Operation**.
3. In the Select one or more services to include in the web service descriptor dialog box, select one or more services and click **OK**.

The specified operations are added to the provider WSD. The operations appear in the Operations tab and are also added to each binder contained in the provider WSD.

If a service signature does not meet the style/use signature requirements established by the existing binder, Designer does not add the service as an operation.

Designer adds the new operation to all binders in the web service descriptor.

4. Click **File > Save**.

If the operation already exists in the web service descriptor, Designer adds it as a copy and appends “_n” to its name, where n is an incremental number.

Tip:

You can also add operations by selecting one or more services in Package Navigator view and dragging them into the Operations tab.



Adding an Operation from another Provider Web Service Descriptor


You can copy or move an operation from another web service descriptor to a service first provider WSD.

Keep the following points in mind when copying or moving an operation from one provider WSD to another:

- You can add operations to a service first provider WSD only.
- Integration Server and Designer do not support mixed “use” across binders and operations and mixed “style” across binders in a single web service descriptor. If the service signature associated with the operation does not meet the style/use signature requirements established by the existing binder, Designer will not add the operation.

➤ To copy or move an existing operation from one provider web service descriptor to another

1. In Package Navigator view, open and lock the provider WSD that contains the operation you want to copy or move.
2. In the Operations tab, select one or more operations. Click  or  on the web service descriptor editor toolbar.
3. In Package Navigator view, open and lock the provider WSD into which you want to paste the cut or copied operations (the target provider WSD).

4. In the Operations tab of the target WSD, click  on the web service descriptor editor toolbar.
5. Click **File > Save**.

Designer adds the specified operations to the provider WSD. Designer also adds the operations to all binders in the target web service descriptor exactly as they existed in the source web service descriptor. The binder values for each individual binder apply to the operations within the binders.

If the operation being added already exists in the provider WSD, Designer adds it as a copy and appends “_n” to its name, where “n” is an incremental number.

Using a 6.5 SOAP-MSG Style Service as an Operation

In webMethods Integration Server version 6.5, you could expose an IS service as a SOAP-MSG web service. The 6.5 SOAP-MSG style services used the default SOAP processor, specified SOAP version 1.1, and specified a style/use of Document/Literal. You can migrate 6.5 SOAP-MSG style services to the web service descriptor framework introduced in Integration Server 7.1 by adding the service as an operation to a provider WSD. By migrating the service, you can leverage the inherent functionality of a provider WSD, such as headers, handlers, faults, and WS-Security.

By default, Integration Server derives the input and output signatures for operations from the services used to create the operation. Integration Server 6.5 required that an IS service in the SOAP-MSG style use a signature that took a *soapRequestData* object and a *soapResponseData* object as input and produced a *soapResponseData* object as output. This signature requirement does not result in meaningful signature information for the operation in WSDL documents generated for the provider WSD. To produce a meaningful, descriptive signature for an operation that corresponds to a 6.5 SOAP-MSG style service, you must select an IS document type or an XML schema element declaration to represent the service input and output signature.

Keep the following points in mind when adding a 6.5 SOAP-MSG style service as an operation to a provider WSD:

- You can add the 6.5 IS service to an existing provider WSD or create a new provider WSD for the IS service. For information about adding an IS service as an operation to a provider WSD, see [“Adding an IS Service as an Operation” on page 69](#). For information about creating a service first provider WSD, see [“Creating a Service First Provider Web Service Descriptor” on page 18](#).
- The provider WSD must have a single binder with the following properties:
 - SOAP version = SOAP 1.1 protocol
 - SOAP binding style = document
 - SOAP binding use = literal
- To produce a meaningful signature for the operation in a WSDL document, you must select an IS document type or an XML schema element declaration to represent the input and output signatures. For information about changing the input or output signature for an operation, a provider WSD, see [“Modifying the Signature of a 6.5 SOAP-MSG Style Operation” on page 72](#).

- If you use an IS document type for the input and/or output signature, the IS document type must satisfy the service signature requirements for the SOAP-MSG style as specified in the *Web Services Developer's Guide* version 6.5.
- If you add any headers to the operation, any existing clients for the 6.5 service must be modified to include the header in the SOAP request.
- Any header handler processing that changes the SOAP message and occurs before service invocation affects the SOAP message passed to the service. Note that 6.5 SOAP-MSG style services expect the SOAP message to be in a certain format. Specifically, any changes to the SOAP body might affect the ability of the 6.5 SOAP-MSG style service to process the request.
- When a 6.5 SOAP-MSG style service is added as an operation, you can add fault processing to the operation response. For fault processing to work, you need to modify the 6.5 SOAP-MSG style service to detect a Fault condition, add Fault output data to the pipeline, and drop the SOAP response message (*soapResponseData* object) from the pipeline.

Modifying the Signature of a 6.5 SOAP-MSG Style Operation

A web service *requires* the input parameters from a signature and *produces* the output parameters. By default, an operation derives the input and output signatures from the services used to create the operation. However, in the case of a 6.5 SOAP-MSG style service, the input and output signatures consist of a *soapRequestData* and *soapResponseData* objects. In a WSDL document, this would result in an vague, meaningless signature. To create a meaningful service signature for a 6.5 SOAP-MSG style operation, you can override the original service signature by selecting an element declaration in an XML schema definition or an IS document type as the service signature. Overriding the service signature is necessary after adding a 6.5 SOAP-MSG style service as an operation to a provider WSD.

Keep the following points when modifying the operation signature source:

- You can only modify the operation signature source in a provider WSD that was created from an IS service. You cannot add or modify the signature of a provider WSD created from a WSDL URL or a UDDI Registry.
- The XML schema definition you select must be located on the web and must be network accessible to consumers of the WSDL. Do not use a local file URL to refer to an external schema.
- If you use an IS document type as the signature for an operation that corresponds to an Integration Server 6.5 SOAP message service, the IS document type must satisfy the service signature requirements for the SOAP MSG protocol a specified in the *Web Services Developer's Guide* version 6.5. For more information about adding an IS 6.5 SOAP message service as an operation, see [“Using a 6.5 SOAP-MSG Style Service as an Operation” on page 71](#).
- An IS document type used to represent the input or output signature of an operation cannot contain top-level fields named “*body” or top-level fields starting with “@”.

➤ To modify the signature type of a 6.5 SOAP-MSG style operation

1. In Package Navigator view, open and lock the provider web service descriptor containing the operation whose signature you want to modify.

2. In the Operations tab, select and expand the operation whose signature you want to modify.
3. Do one of the following:

To change the...	Do this...
Input signature	Expand Request and select the Body element.
Output signature	Expand Response and select the Body element.

4. In the Properties view, next to the **Signature** field, click the browse button.
5. In the Modify I/O Signature dialog box, do one of the following:

Select...	To...
Original IS service	Use the input or output signature from the originating IS service as the input or output signature. This is the default.
Existing external XML schema	Use an element declaration from an XML schema definition as the input or output signature.
Document type	Use an IS document type as the input or output signature.

6. If you selected **Existing external XML schema**, do the following:
 - a. In the **URL** field, after `http://`, type the web location and name of the XML schema definition that contains the element declaration you want to use to describe the signature.
 - b. Click **Load**. Designer displays the global element declarations in the XML Schema.
 - c. Select the global element declaration for the input or output signature.
7. If you selected **Document type**, select the IS document type that you want to use to represent the input or output signature.
8. Click **OK**.
9. Click **File > Save**.


If you selected **Existing external XML schema**, Integration Server automatically uses the internal schema parser to validate the schema. If the **Validate schema using Xerces** property is set to True for the web service descriptor, Integration Server also validates the specified XML Schema definition using the Xerces Java parser. If either parser determines that the schema is invalid, Designer does not save the web service descriptor and displays the validation errors.

Deleting Operations

Keep the following points in mind when deleting operations from a web service descriptor:

- You can delete operations from a service first provider WSD only.
- When you delete an operation on the Operations tab, Designer removes the operation from all the binders in the provider WSD.
- If you delete an operation from within a binder (that is, you delete the operation in the Binders tab), any other instances of that operation in other binders remain in the web service descriptor. If an operation exists in only one binder and is deleted from that binder, the operation is removed from the web service descriptor.

➤ To delete an operation from a provider web service descriptor

1. In Package Navigator view, open and lock the provider WSD that contains the operation to delete.
2. In the Operations tab, select the operation to be deleted.
3. Click  on the web service descriptor editor toolbar. Designer deletes the selected operation from the web service descriptor.
4. Click **File > Save**.

Viewing the Operation Input and Output

You can view the input or output signature of an operation side-by-side with the operations in a web service descriptor.

➤ To view the operation input or output

1. In Package Navigator view, open the web service descriptor.
2. In the Operations tab, navigate to and select the Body element in the Request or Response for an operation.

For a request, Designer displays the operation input. For a response, Designer displays the operation output.

Adding Headers to an Operation

You can add headers to an operation to incorporate additional processing or functionality for the SOAP message. A *header element* defines the format of the SOAP headers that may be present

in a SOAP message (request or response). Headers are optional and can be added to or deleted the request or response in an operation.

For a service first provider web service descriptor, you can:

- Add new headers
- Edit any headers
- Delete any headers

For a WSDL first provider web service descriptor, you can

- Add new headers
- Edit any headers that you add
- Delete any headers that you add or any headers derived from the source WSDL
- Edit the **Must Understand** and **Role** properties for headers derived from the source WSDL

For a consumer web service descriptor, you can:

- Add new headers to the request or response
- Edit any headers that you add
- Delete any headers that you add
- Edit the **Must Understand** and **Role** properties for headers derived from the source WSDL

Note: Integration Server considers all of the headers defined in a web service descriptor to be required. If the header does not exist in the SOAP message at run time, Integration Server throws an error.

While failure when a required header is missing is the correct behavior, Integration Server provides a configuration property to control whether missing required headers in a SOAP response results in an error. If you do not want Integration Server to throw an error in case of missing required headers, set the `watt.server.SOAP.ignoreMissingResponseHeader` server configuration parameter to true.


Adding a Header to an Operation

Keep the following points in mind when adding headers to operations:

- You can copy or move header document types between headers or between faults, but not between a header and a fault. You can use the same document type for a request and a response, subject to the handlers available in the web service descriptor.
- When adding a header element to a provider web service descriptor, be sure that the header does *not* have the same name as any of the fault elements for that web service descriptor.

- An IS document type used as a header or fault for an operation with a binding style/use of RPC/Encoded cannot contain fields named `*body` or `@attribute` fields (fields starting with the “@” symbol).
- You must set up a package dependency if you use an IS document type from a different package as a header.
- A header *must* have a registered header handler. However, you can add the header to an operation and register a header handler for it later. A header without a handler will be ignored or will cause the request to fail (depending on whether the **Must Understand** property for the header is set to **False** or **True**).
- After a header handler is registered in Integration Server, the IS document types associated with the handler will be listed in the selection dialog box that is displayed when you add a header. For more information about registering handlers, see the *Web Services Developer's Guide*.
- The WS Security Handler does not expose supported headers.
- If you add a response header to an operation that uses an In-Only Message Exchange Pattern (MEP), the MEP will change to In-Out MEP. For more information about message exchange patterns, see the *Web Services Developer's Guide*.
- You can also add headers to an operation by dragging IS document types from the Package Navigator view to the Operations tab.
- Integration Server considers all of the headers defined in a web service descriptor to be required.

➤ To add a header to an operation

1. In Package Navigator view, open and lock the web service descriptor to which you want to add a header.
2. In the Operations tab, expand the operation and the request or response to which you want to add the header.
3. Select the header icon and click  (**Add Header or Fault**) on the web service descriptor editor toolbar.

Because a header was selected when you clicked this button, the document selection dialog box displays *only* those IS document types supported by the header handlers listed in the Handlers tab.

4. Select the IS document type to use as a header. Click **OK**.
5. Click **File > Save**.

Important:

When you add a header (or a fault) to a consumer web service descriptor, you must refresh the web service connector(s). See [“Refreshing a Web Service Connector” on page 39](#).

About SOAP Fault Processing

If an error occurs while processing a SOAP request, the response returned to the web service client contains a SOAP fault. You can have the endpoint service signal a fault using one of the following methods:

- Specify a fault whose structure is defined by a fault element, using the fault reasons, code, subcodes, node and role that Integration Server generates.

At design time, you can identify the structure of SOAP faults with which an operation can respond by adding fault elements to the operation response in a web service descriptor. Fault elements are optional and can be added to any web service descriptor. For more information, see [“About SOAP Fault Elements” on page 79](#).

To signal a fault that uses one of the fault elements, set up the endpoint service for the operation so that it places an instance document of one of the fault elements into the top level of the service pipeline. The name of the instance document *must* match the name assigned to the fault element. Integration Server recognizes the fault document in the pipeline, and when generating the fault detail, uses the IS document type defined in the fault element for the structure of the instance document. If the document has a name that matches a fault element, but a different structure, unexpected results will occur.

Integration Server generates a SOAP response that contains a SOAP fault. The SOAP fault contains the detail from the instance document and uses fault reasons, code, subcodes, node and role that Integration Server generates.

- Specify a fault whose structure is defined by a fault element, but override the fault reasons, code, subcodes, node and/or role that Integration Server generates.

Set up the endpoint service for an operation so that it places an instance document of one of the defined fault elements into the *\$fault/detail* variable. The name of the instance document *must* match the name assigned to the fault element. Integration Server recognizes the fault document in the *\$fault/detail* variable, and when generating the fault detail, uses the IS document type defined in the fault element for the structure of the instance document. If the document has a name that matches a fault element, but a different structure, unexpected results will occur.

To override the fault reasons, code, subcodes, node and/or role, set up the endpoint service to also provide the corresponding values in fields within the *\$fault* variable. For a description of the *\$fault* variable, see [“The \\$fault Variable” on page 81](#).

Integration Server recognizes the *\$fault* variable in the pipeline. Subsequently, Integration Server generates a SOAP response that contains a SOAP fault using the information from the *\$fault/detail* variable. The SOAP fault contains the detail from the instance document and uses values specified for fault reasons, code, subcodes, node and/or role within the *\$fault* variable to override the corresponding values that Integration Server generates.

Note:

If there is a top-level instance document for the fault, in addition to the one in the *\$fault/detail* variable, Integration Server ignores the top-level document.

- Specify a fault with a structure that was not previously defined using a fault element. Optionally, override the fault reasons, code, subcodes, node and/or role that Integration Server generates.

Although you can identify the structure of SOAP faults in advance, it is not required. To signal a fault at run time, you can add fault information that does not match defined fault elements to the `$fault/detail` variable in the pipeline. Be sure that the name does *not* match any defined fault elements. Integration Server recognizes the `$fault/detail` variable in the service pipeline. Because the document in the `$fault/detail` variable does not match a defined fault element, Integration Server generates the fault detail without using an IS document type for the structure.

To override the fault reasons, code, subcodes, node and/or role, set up the endpoint service to provide the corresponding values in fields within the `$fault` variable. For more information, see [“The \\$fault Variable” on page 81](#).

Integration Server ignores any top-level instance document that might be in the pipeline for a fault. Using the information from the `$fault/detail` variable, Integration Server generates a SOAP response that contains a SOAP fault. If values are specified for the fault reasons, code, subcodes, node and/or role within the `$fault` variable, Integration Server uses those values instead of values it generates.

Additionally, faults can occur for the following reasons:

- The endpoint service throws a service exception.

In this case, Integration Server constructs a fault message out of the service exception. If the pipeline also contains a `$fault` variable, Integration Server uses the information specified in the `$fault` variable to override the fault information.

To make the `$fault` variable available, you can write a Java service that throws a `ServiceException`, but before throwing the exception, places the `$fault` variable in the pipeline.

Alternatively, for a flow service, you can use the EXIT with failure construct. As a result, before exiting the flow service with a failure, you can place the `$fault` variable into pipeline.

- A request handler service ended in failure and signaled that a fault should be generated.

When the request handler returns a status code 1 or 2, Integration Server generates a SOAP fault, along with the fault code, subcodes, reasons, node, and role for the fault. You can use the `pub.soap.handler:updateFaultBlock` service to modify the code, subcodes, reasons, node, and/or role that Integration Server generates.

Note:

When the request handler returns status code 3, you are expected to build the SOAP fault. As a result, the `pub.soap.handler:updateFaultBlock` service is not necessary.

You can invoke the `pub.soap.handler:updateFaultBlock` service in a response or fault handler to update the fault created due to the failure in the request handler chain. For more information about using the service, see [“Modifying a Returned SOAP Fault” on page 82](#). For more information about handlers, see the *Web Services Developer's Guide*.

About SOAP Fault Elements

To identify the information to provide in a SOAP fault at design time, you add fault elements to the operation response in a web service descriptor. The fault element, which is an IS document type, describes the expected structure of the Detail element in the SOAP fault. Fault elements are optional and can be added to any web service descriptor.

When you create a service first provider web service descriptor, add fault elements to represent the SOAP faults that an operation in the web service descriptor might return. If an error occurs at run time, the underlying service that corresponds to the operation can signal a fault by returning an instance document for one of the IS document types used as a fault element. Integration Server recognizes the fault document in the service pipeline and subsequently generates a SOAP response that contains a SOAP fault. Within the SOAP fault, the Detail element contains the fault document.

When you create a WSDL first provider web service descriptor or a consumer web service descriptor, Integration Server creates an IS document type for each message element in the source WSDL document. If an operation in a WSDL defines a `soap:fault` element, Integration Server generates an IS document type for the fault element.

In a consumer web service descriptor, the web service connector that corresponds to the operation includes logic to detect the fault element in the SOAP response. Integration Server then places the contents of the fault document into the detail document in the output parameter. The structure of the detail element matches the structure of the IS document type used as the fault element.

Note:

The structure of the SOAP fault returned by the web service connector depends on the version of Integration Server on which the web service descriptor was created. For more information, see *Web Services Developer's Guide*.


It is possible for a web service to return a fault that does not appear in a WSDL file. To account for these SOAP faults, you can add fault elements to a WSDL first provider web service descriptor or a consumer web service descriptor. For more information, see [“Adding a Fault Element to an Operation” on page 79](#).

Adding a Fault Element to an Operation

Keep the following points in mind when adding fault elements to an operation:

- You add fault elements to an operation response.
- The fault document must be an IS document type.
- You must set up a package dependency if you use an IS document type from a different package as a fault.
- If you add a fault to an operation that uses an In-Only Message Exchange Pattern (MEP), the MEP will change to Robust In-Only MEP. For more information about message exchange patterns, see the *Web Services Developer's Guide*.

➤ **To add a fault element to an operation**

1. In Package Navigator view, open and lock the web service descriptor to which you want to add a fault element.
2. In the Operations tab, expand the operation and the response to which you want to add the fault element.
3. Select the **Fault** icon and click  (**Add Header or Fault** button) on the web service descriptor editor toolbar.

Because a fault was selected when you clicked this button, Designer displays the default document selector dialog.

4. Select the IS document type to use as the fault element. Click **OK**.
5. If you want to change the name of the fault element, with the fault element selected, in the **General** category of the Properties view, update the **Name** property.
6. Click **File > Save**.

Important:

When you add a fault to a consumer web service descriptor, you must refresh the web service connector(s). See [“Refreshing a Web Service Connector” on page 39](#).

Notes:

- If you add a fault element to an operation in a consumer web service descriptor, and then refresh the web service connector, Integration Server updates the logic of the web service connector to look for and handle the fault at run time.
- If you add a fault element to an operation in a WSDL first provider web service descriptor, the WSDL document generated from the provider web service descriptor will include the new faults as soap:fault elements in the operation.
- You can add multiple fault elements to an operation in a web service descriptor. At run time, if the service that corresponds to the operation returns multiple fault documents, the SOAP fault in the resulting SOAP response will contain only one fault document. Specifically, Integration Server returns the fault document that is an instance of the IS document type that appears first in the operations list of fault elements.

For example, suppose that an operation had three fault elements listed in this order: faultA, faultB, and faultC. Note that each fault element corresponds to an IS document type of the same name. At run time, execution of operation (service) results in two fault documents—one for faultB and one for faultC. In the SOAP response generated by Integration Server, the SOAP fault contains the faultB document only.

The \$fault Variable

Use the *\$fault* variable to override values Integration Server generates for a fault. To do so, specify the fault detail in the *\$fault/detail* variable. Then, to override the fault reasons, code, subcodes, node and/or role, provide the corresponding values.

The following shows the structure of the *\$fault* variable.

Variable	Description
<i>\$fault</i>	Document Fault information that overrides other fault information in the service pipeline, if any.
<i>code</i>	<p>Document Optional. The fault code and possible subcodes. Integration Server uses values you specify to modify the fault code and subcodes it generates for the fault.</p> <p>Note: For a SOAP 1.1 fault, Integration Server ignores any values specified for <i>subcodes</i>.</p>
<i>namespaceName</i>	String The namespace name for the SOAP fault code.
<i>localName</i>	String A code that identifies the fault.
<i>subcodes</i>	<p>Document List Optional. Subcodes that provide further detail. Each Document in the <i>subCodes</i> Document List contains:</p> <ul style="list-style-type: none"> ■ <i>namespaceName</i> for the subcode ■ <i>localName</i> that identifies the subcode
<i>reasons</i>	<p>Document List Optional. Reasons for the SOAP fault. Integration Server uses values you specify to modify the reasons it generates for the fault.</p> <p>Note: For a SOAP 1.1 fault, if you specify more than one reason, Integration Server uses the first reason. Multiple reasons are supported for SOAP 1.2 faults.</p>
<i>@lang</i>	String Language for the human readable description.
<i>*body</i>	String Text explaining the cause of the fault.
<i>node</i>	<p>String Optional. The URI to the SOAP node where the fault occurred. Integration Server uses value you specify to modify the node it generates for the fault.</p> <p>Note:</p>

Variable	Description
	For a SOAP 1.1 fault, Integration Server ignores any values specified for <i>node</i> .
<i>role</i>	String Optional. The role in which the node was operating at the point the fault occurred. Integration Server uses value you specify to modify the role it generates for the fault.
<i>detail</i>	Document Fault information you want Integration Server to use. This overrides any top-level instance document that defines the fault detail.

Modifying a Returned SOAP Fault

If a provider or consumer web service results in a fault, you can modify values in the fault, if needed. For example, you might want to alter the fault code if your error handling requires a specific code.

To update a SOAP fault, use the `pub.soap.handler:updateFaultBlock` service. For more information about this service, see the *webMethods Integration Server Built-In Services Reference*. You can invoke the `updateFaultBlock` service from a response handler or fault handler service for a web service provider. Use the service to customize one or more of the following SOAP fault fields:

Fault field you can customize	Notes
fault code and subcodes	For a SOAP 1.1 fault, if you specify subcode values, the service ignores them because subcodes are only applicable for a SOAP 1.2 fault.
fault reasons	For a SOAP 1.1 fault, if you specify more than one reason, the service only uses the first reason. Multiple reasons are supported for SOAP 1.2 faults.
fault node	For a SOAP 1.1 fault, if you specify a value for <i>node</i> , the service ignores it because the fault node is only applicable for a SOAP 1.2 fault.
fault role	The fault role is supported for both SOAP 1.1 and SOAP 1.2 faults.

Viewing Document Types for a Header or Fault Element

You can view an IS document type used for a header or fault element side-by-side with the operations for a web service descriptor. The IS document type is read-only in the Operations tab.

➤ To view the document type for a header or fault element in an operation

1. In Package Navigator view, open the web service descriptor.

2. In the Operations tab, navigate to and select the header or fault element for which you want to view the IS document type contents.

Working with Handlers

When working with web services on Integration Server, the SOAP body portion of the SOAP message contains the data representing the input and output signatures of the underlying SOAP operation. In typical processing, Integration Server converts the SOAP body between its XML representation in the SOAP message and the Document (IData) representation used within Integration Server automatically.

In addition to the data contained in the SOAP body, a SOAP message might contain data in the SOAP headers. The best way to access the SOAP headers is to use handlers. A handler, sometimes called a header handler, provides access to the entire SOAP message.

Handlers can be used to perform various types of processing, including processing SOAP headers, adding SOAP headers, removing SOAP headers, passing data from the header to the endpoint service or vice versa.

In Integration Server, a handler is a set of up to three handler services. The handler can contain one of each of the following handler services:

- Request handler service
- Response handler service
- Fault handler service

For detailed information about request, response, or fault handler services, see *Web Services Developer's Guide*.

Any IS service can be used as a handler service. However, handler services must use a specific service signature. Integration Server defines the service handler signature in the `pub.soap.handler:handlerSpec` specification. Integration Server also provides several services that you can use when creating handler services. These services are located in the `pub.soap.handler` folder in the `WmPublic` package.

When you register a handler, you name the handler, identify the services that function as the request, response or fault handler services, and indicate whether the handler is for use with provider web service descriptors or consumer web service descriptors.

You can assign multiple handlers to a web service descriptor. Designer displays the handlers on the Handlers tab. The collection of handlers assigned to a web service descriptor is called a handler chain. For a consumer web service descriptor, Integration Server executes the handler chain for output SOAP requests and inbound SOAP responses. For a provider web service descriptor, Integration Server executes the handler chain for inbound SOAP requests and outbound SOAP responses.

When executing the handler chain, Integration Server executes request handler services by working through the handler chain from top to bottom. However, Integration Server executes response handler services and fault handler services from bottom to top.

The order of handlers in the handler chain may be important, depending on what processing the handlers are performing.

Setting Up a Header Handler

To create and implement a header handler, you need to:

- Build the services for handling a request, handling a response, and handling a fault. Use the `pub.soap.handler:handlerSpec` specification as the signature for a service that acts as a header handler.
- Register the combination of those services as a header handler.
- Assign the header handler to the web service descriptor.

Registering a Header Handler

Register the handler as either a consumer or provider using `pub.soap.handler:registerWmConsumer` or `pub.soap.handler:registerWmProvider`, respectively. During registration:

- You provide a logical name for the handler.
- You specify the services for handling a request, a response, and a fault as input.
- You optionally specify the list of QNames on which the handler operates.


Specify QNames only if you want to associate with handler with one or more QNames. Registering QNames with a handler provides the following benefits:

- Integration Server can perform `mustUnderstand` checking for the header with the QName at run time. If a service receives a SOAP message in which a header requires `mustUnderstand` processing by the recipient, Integration Server uses the header QName to locate the handler that processes the header. Note that the handler must be part of the handler chain for the WSD that contains the service.
- When adding headers to a WSD, Designer populates the list of IS document types that can be used as headers in the WSD with the IS document types whose QNames were registered with the handlers already added to the WSD. If you add a IS document type as a header to a WSD and the QName of that IS document type is not associated with a handler, Designer adds the header but display a warning stating that there is not an associated handler.
- When consuming WSDL to create a provider or consumer WSD, Integration Server automatically adds a handler to the resulting WSD if the WSDL contains a QName supported by the handler.

Note: Integration Server stores information about registered header handlers in memory. Integration Server does not persist registered header handler information across restarts. Consequently, you must register header handlers each time Integration Server starts. To accomplish this, create a service that registers a header handler and make that service a start up service for the package that contains the services that act as header handlers.

Adding a Handler to a Web Service Descriptor

> To add a handler to a web service descriptor


1. In Package Navigator view, open and lock the web service descriptor to which you want to add handlers.
2. In the Handlers tab, click  on the web service descriptor toolbar. Or right-click and select **Add Handler**.
3. Select the registered handler that you want to add to the web service descriptor.
4. Click **File > Save**.
5. Once a handler is added to a web service descriptor, you may optionally add any headers associated with the handler to the request or response elements of operations within the web service descriptor.

Note:

You must set up a package dependency if the web service descriptor uses a handler from a different package.

Deleting a Handler from a Web Service Descriptor

> To delete a handler from a web service descriptor

1. In Package Navigator view, open and lock the web service descriptor from which you want to remove handlers.
2. Click the Handlers tab.
3. Select the handler that you want to delete.
4. Click  on the web service descriptor toolbar or right-click and select **Delete**.

Designer removes the selected handler is deleted from the Handlers tab and from the web service descriptor. If the web service descriptor still contains a header associated with the deleted handler, Designer displays a warning.

Working with Policies

WS-Policy is a model and syntax you can use to communicate the policies associated with a web service. *Policies* describe the requirements, preferences, or capabilities of a web service. You attach

a policy to a *policy subject*, for example, a service, endpoint, operation, or message. After attaching a policy to a policy subject, the policy subject becomes governed by that policy.

webMethods Integration Server provides support for *Web Services Policy Framework* (WS-Policy) Version 1.2.

In Integration Server, a policy is specified using policy expressions in an XML file called a *WS-Policy file* (more simply called *policy file*). Integration Server comes with some policy files out of the box. Additionally, you can also create policy files. For more information, see *Web Services Developer's Guide*.

To have a web service governed by the policy in a WS-Policy file, you attach the policy to the web service descriptor. You can attach WS-Policies at the binding operation message type level, such as input, output, and fault, in a web service descriptor.

Attaching a Policy to a Web Service Descriptor

You can attach one or more policies to a single web service descriptor. Also, multiple web service descriptors can share the same policy. You can attach any policy that resides in the policy repository.

Keep the following points in mind when attaching policies to a web service descriptor:

- To attach a policy to a web service descriptor, the **Pre-8.2 compatibility mode** property of the web service descriptor must be set to false.


If you change the **Pre-8.2 compatibility mode** property of a web service descriptor from false to true after a policy is attached to it, the policy subject will no longer be governed by that policy.

For more information about **Pre-8.2 compatibility mode** property, see [“About Pre-8.2 Compatibility Mode” on page 88](#).

- When attaching policies, avoid attaching a policy that contains policy assertions that Integration Server does not support. For information about supported assertions, see the *Web Services Developer's Guide*. If you attach a policy that contains unsupported policy assertions, unexpected behavior may occur.
- If you attach a policy to a WSDL first provider web service descriptor or a consumer web service descriptor, the attached policy will override any annotated policy in the source WSDL.
 - For a web service descriptor with a policy attached to it, the attached policy always takes precedence at run time.
 - For a consumer web service descriptor, even though the consumer WSDL will not show the attached policy, Integration Server will enforce the attached policy at run time.
- When you attach a policy to or remove a policy from a provider web service descriptor, the WSDL generated for that web service descriptor is changed as well. Any web service clients generated from the WSDL will need to be regenerated.

- When you attach a policy to or remove a policy from a consumer web service descriptor, you do not need to refresh the web service connectors to pick up the policy change. Integration Server detects and enforces the policy change at run time.
- If the policy you are attaching contains WS-SecurityPolicy assertions and you also want to use MTOM streaming, be aware that if the fields to be streamed are also being signed and/or encrypted, Integration Server cannot use MTOM streaming because Integration Server needs to keep the entire message in memory to sign and/or encrypt the message.

➤ To attach a policy to a web service descriptor

1. In Package Navigator view, open and lock the web service descriptor to which you want to attach a policy.
2. In the Policies tab, click  on the web service descriptor toolbar, or right-click and select **Attach Policy**.
3. Select the policies that you want to attach to the web service descriptor.

Designer displays the policies that you selected in the Policies tab.

4. Against each policy in the Policies tab, select the appropriate check boxes to attach a WS-Policy to **Input**, **Output**, and/or **Fault** message type. You can choose to attach a policy to all message types or to any of the three message types. You must select at least one message type for each policy in the Policies tab if you want the web service descriptor to be governed by that policy.

Note:


By default, all the three message types are selected.

5. Click **File > Save**.

Removing a Policy from a Web Service Descriptor

If you no longer want a web service descriptor to be governed by a particular policy, you can detach or remove the policy from the web service descriptor.

➤ To remove a policy from a web service descriptor

1. In Package Navigator view, open and lock the web service descriptor from which you want to remove a policy.
2. Click the Policies tab.
3. Select the policy that you want to delete from the web service descriptor.
4. Click  on the web service descriptor toolbar or right-click and select **Delete**.

About Pre-8.2 Compatibility Mode

Integration Server version 8.2 introduces support for web service features, such as SOAP over JMS, and WS-Policy based WS-Security configuration, that are available through the current Web Services Stack. Some of the features and behavior included in the current Web Services Stack are not compatible with the features and run-time behavior of web service descriptors created on the earlier implementation of web services. The earlier web services implementation was introduced in Integration Server version 7.1 and was the only web services implementation available in 7.x, 8.0, and 8.0 SP1. The earlier web services implementation still exists in the current version of Integration Server along with the current Web Services Stack.

Note:

The earlier web services implementation, specifically the implementation introduced in Integration Server version 7.1, is deprecated as of Integration Server 10.4.

To ensure that web service descriptors developed on the earlier web services implementation execute as expected, web service descriptors now have a **Pre-8.2 compatibility mode** property. This property determines the web service implementation on which the web service descriptor runs. The web service implementation used by the web service descriptor determines the design-time features and run-time behavior for the web service descriptor. The value of the **Pre-8.2 compatibility mode** property indicates the web service implementation with which the web service descriptor is compatible:

- When the **Pre-8.2 compatibility mode** property is set to true, the web service descriptor runs on the earlier implementation of web services, specifically the implementation introduced in Integration Server version 7.1. Web service descriptors running in pre-8.2 compatibility mode have the same design-time features and run-time behavior as web service descriptors run in versions of Integration Server prior to version 8.2.
- When the **Pre-8.2 compatibility mode** property is set to false, the web service descriptor runs on the current implementation of web services, specifically the Web Services Stack. Web service descriptors that do not run in pre-8.2 compatibility mode have the design-time features and run-time behavior available in the current version of the Web Services Stack.

Note:

You can use Designer 8.2 or later with an Integration Server 8.2 or later to create and edit a web service descriptor regardless of the compatibility mode.

Note:

The **Pre-8.2 compatibility mode** property and the ability to run in pre-8.2 compatibility mode are deprecated as of Integration Server 10.4 due to the deprecation of the earlier web services implementation that was introduced in Integration Server version 7.1.

Setting Compatibility Mode

Keep the following points in mind when setting the **Pre-8.2 compatibility mode** property for a web service descriptor:

- You can set the compatibility mode using Designer 8.2 or later only.

- The compatibility mode alters the design-time features available for the web service descriptor and might change the run-time behavior of the web service descriptor.
- You can use the `pub.utils.ws:setCompatibilityModeFalse` service to change the **Pre-8.2 compatibility mode** property value for multiple web service descriptors at one time. For more information, see the *webMethods Integration Server Built-In Services Reference*.
- If you intend to change the compatibility mode of a web service descriptor for which you published metadata to CentraSite, first retract metadata for the web service descriptor. Next, change the compatibility mode. Finally, republish metadata for the web service descriptor to CentraSite.
- The **Pre-8.2 compatibility mode** property and the ability to run in pre-8.2 compatibility mode are deprecated as of Integration Server 10.4 due to the deprecation of the earlier web services implementation introduced in Integration Server 7.1.

➤ To set the compatibility mode for a web service descriptor

1. Open Designer.
2. In Package Navigator view, open and lock the web service descriptor for which you want to change the compatibility mode.
3. In the Properties view, next to **Pre-8.2 compatibility mode**, do one of the following:
 - Select **True** if you want the web service descriptor to run in pre-8.2 compatibility mode. **True** indicates that Integration Server will deploy the web service descriptor to the earlier web services implementation introduced in Integration Server versions 7.1.
 - Select **False** if you do not want the web service descriptor to run in pre-8.2 compatibility mode. **False** indicates that Integration Server will deploy the web service descriptor to the current Web Services Stack.

Designer verifies that the web service descriptor can be deployed to the web services stack that corresponds to the chosen compatibility mode and displays any errors or warnings.

4. If Designer displays errors or warnings, do one of the following:
 - If errors occur, Designer determined that the web service descriptor cannot be deployed to the corresponding web services stack successfully. Designer displays the errors that identify the functionality that is incompatible with the web services stack. Click **OK** to cancel the change to the **Pre-8.2 compatibility mode** property.
 - If warnings occur, Designer determined that the web service descriptor can be deployed to the corresponding web services stack successfully but some run-time behavior might change. Designer displays any warnings about the functional changes of the web service descriptor in the web services stack. Click **OK** to proceed with the change to the **Pre-8.2 compatibility mode** property. Click **Cancel** to cancel the change.
5. Click **File > Save**.

Features Impacted by Compatibility Mode

The following table identifies the web service features impacted by the compatibility mode of the web service descriptor.

Name	Description	Behavior
Binders	Multiple binders with different operations	<p>In pre-8.2 compatibility mode, Integration Server permits a single web service descriptor to contain multiple binders with different operations.</p> <p>When not in pre-8.2 compatibility mode, all the binders for a single web service descriptor must contain the same operations. Integration Server will not save a service first provider web service descriptor if it contains multiple binders that list different operations.</p>
Binding Styles	Mixed binding styles in web service descriptors	<p>In pre-8.2 compatibility mode, Integration Server does not restrict the binding styles in a web service descriptor. A web service descriptor could contain binders that used different binding styles.</p> <p>When not in pre-8.2 compatibility mode, Integration Server requires all binders within a web service descriptor to use the same binding style. That is, all the binders in a single web service descriptor should specify a binding style of RPC or Document for all of the binders.</p>
Binding Styles	Mixed binding styles in WSDLs	<p>In pre-8.2 compatibility mode, Integration Server creates web service descriptors from WSDLs that contained bindings that used different binding styles.</p> <p>When not in pre-8.2 compatibility mode, Integration Server requires all bindings in the consumed WSDL to have the same style. Integration Server will not create a web service descriptor from a WSDL that contains mixed styles across its bindings.</p>
JMS Bindings	Support for JMS bindings and JMS binders	<p>In pre-8.2 compatibility mode, Integration Server supports HTTP and HTTPS bindings only.</p> <ul style="list-style-type: none"> ■ Integration Server will not create a WSDL first provider web service descriptor from a WSDL document that contains a JMS binding. ■ When creating a consumer web service descriptor, Integration Server will not create binders that correspond to JMS bindings in the WSDL

Name	Description	Behavior
		<p>document. If the WSDL document contains only JMS bindings, Integration Server will not create the consumer web service descriptor.</p> <ul style="list-style-type: none"> ■ A web service descriptor cannot contain any binders that specify the JMS transport. <p>When not in pre-8.2 compatibility mode, Integration Server supports JMS bindings in addition to HTTP and HTTPS bindings.</p> <ul style="list-style-type: none"> ■ Integration Server can create provider web service descriptors from WSDL documents that contain JMS bindings. ■ When creating a consumer web service descriptor, Integration Server will create binders that correspond to JMS bindings in the WSDL document. ■ A web service descriptor can use binders that specify JMS as the transport.
Message Exchange Pattern (MEP) Support	Added In-Only MEP and Robust In-Only MEP support	<p>In pre-8.2 compatibility mode, Integration Server supports only In-Out MEP for web service operations.</p> <p>When not in pre-8.2 compatibility mode, Integration Server supports In-Only MEP and Robust In-Only MEP, in addition to In-Out MEP.</p>
MTOM Attachments	Streaming MTOM Attachments	<p>In Pre-8.2 compatibility mode, Integration Server supports MTOM attachments in both inbound and outbound messages. However, Integration Server cannot stream the MTOM attachments. Integration Server always holds the MTOM attachment in memory.</p> <p>When not in Pre-8.2 compatibility mode, Integration Server can stream the MTOM attachments for both inbound and outbound SOAP messages.</p>
Port types	Port types in web service descriptors	<p>In pre-8.2 compatibility mode, Integration Server supports multiple port types. When Integration Server generated a WSDL for a multi-binder web service descriptor, the resulting WSDL had multiple port types.</p> <p>When not in pre-8.2 compatibility mode, Integration Server does not support multiple port types. As a result, when Integration Server generates a WSDL</p>

Name	Description	Behavior
		for a multi-binder provider web service descriptor, it changes the port type names so that the generated WSDL has only a single port type.
Port types	Port types in WSDLs	<p>In pre-8.2 compatibility mode, Integration Server creates web service descriptors from WSDLs that contained multiple port types.</p> <p>When not in pre-8.2 compatibility mode, Integration Server does not support creating web service descriptors from WSDLs that contain multiple port types.</p>
Provider web service descriptor	Provider web service descriptor with no operations	<p>In pre-8.2 compatibility mode, Integration Server permits a provider web service descriptor that does not contain any operations.</p> <p>When not in pre-8.2 compatibility mode, Designer will not save a web service descriptor if it does not contain at least one operation.</p>
Response services	Response services in consumer web service descriptor	<p>In pre-8.2 compatibility mode, Integration Server does not create the responseServices folder when creating a consumer web service descriptor.</p> <p>When not in pre-8.2 compatibility mode, Integration Server creates the responseServices folder that contains response services for each operation in the WSDL document and a genericFault_Response service when creating a consumer web service descriptor.</p>
Service name	Name attribute in wsdl:service element	<p>In pre-8.2 compatibility mode, the local name of the web service descriptor determines the value of the name attribute in the wsdl:service element in the associated WSDL document. For example, suppose that a web service descriptor has the fully qualified name folder.myFolder:myWebService. In the WSDL document, the value of the name attribute in the wsdl:service element is "myWebService".</p> <p>When not in pre-8.2 compatibility mode, the fully qualified name of the web service descriptor determines the value of the name attribute in the wsdl:service element. For example, suppose that a web service descriptor has the fully qualified name folder.myFolder:myWebService. In the WSDL document, the value of the name attribute in the wsdl:service element is "folder.myFolder:myWebService".</p>

Name	Description	Behavior
Web service handlers	Web service handlers based on JAX-RPC	<p>In pre-8.2 compatibility mode, Integration Server supports web service handlers based on JAX-RPC.</p> <p>When not in pre-8.2 compatibility mode, Integration Server will not execute web service handlers based on JAX-RPC.</p>
Web service handlers	Web service handler chain execution	<p>In pre-8.2 compatibility mode, you can use the WS-Security facility to secure a web service. The WS-Security facility secures web services using the WS-Security handler. Because a handler is used, Integration Server can perform additional processing before and after the security processing. That is, for inbound messages Integration Server can invoke handlers before invoking the WS-Security handler to perform security processing. For outbound messages, Integration Server can invoke custom handlers after it invokes the WS-Security handler, but before it sends the outbound message. The order of the handlers on the Handlers tab determines the order in which Integration Server invokes them.</p> <p>When not in pre-8.2 compatibility mode, for inbound messages, Integration Server always performs the security processing first upon receiving the message. As a result, Integration Server cannot invoke custom handlers before the security processing of an inbound message. For outbound messages, Integration Server always performs the security processing last, right before it sends the message. As a result, Integration Server cannot invoke handlers after the security processing of an outbound message.</p>
WS-Addressing	WS-Addressing	<p>In pre-8.2 compatibility mode, Integration Server does not support WS-Addressing.</p> <p>When not in pre-8.2 compatibility mode, Integration Server implements WS-Addressing by associating WS-Addressing policies that conform to Web Services Policy framework to web service descriptors.</p>
WS-Policy	WS-Policy	<p>In pre-8.2 compatibility mode, Integration Server does not support WS-Policy.</p> <p>When not in pre-8.2 compatibility mode, Integration Server supports the WS-Policy framework.</p>

Name	Description	Behavior
WS-Security	WS-Security facility	<p>In pre-8.2 compatibility mode, Integration Server implemented WS-Security by associating WS-Security handlers to web service descriptors.</p> <p>When not in pre-8.2 compatibility mode, Integration Server implements WS-Security by associating WS-Security policies that conform to Web Services Policy framework to web service descriptors.</p>

2 SOAP Message Exchange Patterns

■	Message Exchange Patterns that Integration Server Supports	96
■	How Integration Server Determines the MEP Type to Use	96
■	How the MEP Type Affects the SOAP Response a Provider Returns	98
■	How <wsdl:output> and <wsdl:fault> Elements Affect a Consumer	99
■	How Adding Response Headers or Faults Affect In-Only MEP Operations	100
■	How the MEP Affects the Execution of Handlers	100
■	Considerations When Changing a Provider's Compatibility Mode	100
■	Considerations When Changing a Consumer's Compatibility Mode	102

Message Exchange Patterns that Integration Server Supports

Each operation of a web service uses a SOAP Message Exchange Pattern (MEP), which is a template that defines how the operation exchanges SOAP messages. Integration Server supports the following MEP types:

- **In-Out MEP**, a request-response pattern where a consumer sends a request and expects a SOAP response from the provider.
- **In-Only MEP**, a one-way message exchange where the consumer sends a request and expects no SOAP response from the provider.
- **Robust In-Only MEP**, a reliable one-way message exchange where the consumer sends a request and expects no SOAP response from the provider if the operation is successful, but expects a SOAP fault if an exception occurs.

Note:

For consumers, Integration Server provides partial support for Robust In-Only for SOAP over JMS. For more information, see [“Consumer Support for Robust In-Only Operations with SOAP/JMS Bindings” on page 115](#).

How Integration Server Determines the MEP Type to Use

Typically, the MEP type that an operation uses depends on whether its WSDL includes `<wsdl:output>` and/or `<wsdl:fault>` elements. A `<wsdl:output>` element defines the output a provider returns or that a consumer expects. A `<wsdl:fault>` element indicates whether a provider should respond with a fault or a consumer should expect a fault if an exception occurs. The table below summarizes the MEP type based on the WSDL for an operation.

<code><wsdl:output></code> element?	<code><wsdl:fault></code> element?	MEP Type
Yes	Yes or No	In-Out MEP
No	No	In-Only MEP
No	Yes	Robust In-Only MEP

For Integration Server MEP support, a web service descriptor’s **Pre-8.2 compatibility mode** property can alter the MEP type defined by the WSDL.

- For provider web service descriptors, when the **Pre-8.2 compatibility mode** property is `false`, Integration Server uses the MEP type based on the WSDL, as described in the table above. However, when the **Pre-8.2 compatibility mode** property is `true`, Integration Server always uses In-Out MEP.
- For consumer web service descriptors, the web service connector always expects responses based on the WSDL. For example, if the WSDL includes a `<wsdl:output>` element, it expects a SOAP response. If the provider appropriately provides a response/fault based on the WSDL for the operation, the **Pre-8.2 compatibility mode** property does not affect the behavior.

However, the **Pre-8.2 compatibility mode** property does dictate how a web service connector reacts to receiving unexpected SOAP responses. When the **Pre-8.2 compatibility mode** property is false, Integration Server ignores unexpected responses. When the **Pre-8.2 compatibility mode** property is true, Integration Server honors unexpected responses. For more information, see [“How <wsdl:output> and <wsdl:fault> Elements Affect a Consumer” on page 99.](#)

Note:

For consumers, Integration Server provides partial support for Robust In-Only for SOAP over JMS. For more information, see [“Consumer Support for Robust In-Only Operations with SOAP/JMS Bindings” on page 115.](#)

WSDL for Service First Providers

How Integration Server generates the WSDL for a service first provider web service descriptor depends on the **Pre-8.2 compatibility mode** property.

- When the **Pre-8.2 compatibility mode** property is false, Integration Server includes or omits the <wsdl:output> element based on the service’s output signature. If the service:
 - Has defined output, Integration Server includes a <wsdl:output> element, and the operation uses In-Out MEP.
 - Has *no* defined output, Integration Server omits the <wsdl:output> element, and the operation uses In-Only MEP.

Note: Integration Server *never* includes a <wsdl:fault> element when generating WSDL for a service first provider. To have Integration Server use Robust In-Only MEP, after creating the web service descriptor, add a fault to the In-Only MEP operation to change the MEP to Robust In-Only MEP.

- When the **Pre-8.2 compatibility mode** property is true, Integration Server always includes a <wsdl:output> element for the operation, even when the service used for the web service descriptor has no output parameters. As a result, the operation uses In-Out MEP.

WSDL for WSDL First Providers and Consumers

For WSDL first provider or consumer web service descriptors, Integration Server uses the original WSDL as is.

For a WSDL first provider web service descriptor, when the **Pre-8.2 compatibility mode** property is false, Integration Server uses a MEP type based on the WSDL. When **Pre-8.2 compatibility mode** property is true, Integration Server uses In-Out MEP regardless of whether the WSDL for an operation includes a <wsdl:output> element.

For a consumer web service descriptor, the **Pre-8.2 compatibility mode** property does dictate how a web service connector reacts to receiving unexpected SOAP responses. When the **Pre-8.2 compatibility mode** property is set to true, Integration Server is more tolerant of unexpected SOAP responses. For more information, see [“How <wsdl:output> and <wsdl:fault> Elements Affect a Consumer” on page 99.](#)

Note:

For consumer web service descriptors, Integration Server provides partial support for Robust In-Only for SOAP over JMS. For more information, see [“Consumer Support for Robust In-Only Operations with SOAP/JMS Bindings” on page 115](#).

How the MEP Type Affects the SOAP Response a Provider Returns

At run time, a web service operation’s MEP type dictates the SOAP response that a provider web service descriptor returns.

- **For In-Out MEP**, the **Pre-8.2 compatibility mode** property is a factor in the type of response a provider returns.

- When the **Pre-8.2 compatibility mode** property is `false`, the operation will only use In-Out MEP if it actually returns output. As a result, when the operation executes successfully, it returns a SOAP response that corresponds to the output defined for the operation. If an exception occurs, the operation returns a SOAP fault.
- When the **Pre-8.2 compatibility mode** property is `true`, although In-Out MEP is in use, an operation might not actually return a SOAP response.
 - If an operation actually returns output, when the operation executes successfully, it returns a SOAP response that corresponds to the output defined for the operation. If an exception occurs, the operation returns a SOAP fault.
 - If an operation does *not* actually return output, Integration Server still returns a SOAP response. If the operation executes successfully, Integration Server returns a SOAP response with an empty element in the SOAP body. If an exception occurs, a SOAP fault is returned.

This is the case for a service first provider when the service used for the web service descriptor does not have output parameters or for a WSDL first provider that does not include a `<wsdl:output>` element for an operation. In both cases, although the operation does not have output, because the **Pre-8.2 compatibility mode** property is `true`, Integration Server still uses In-Out MEP.

- **For In-Only MEP**, the **Pre-8.2 compatibility mode** property must be set to `false`.

When a client requests an In-Only MEP operation, the operation does not provide a SOAP response, even if an exception occurs.

- **For Robust In-Only MEP**, the **Pre-8.2 compatibility mode** property must be set to `false`.

When a client requests a Robust In-Only MEP operation, if the operation completes successfully, the operation does not return a response. However, if an exception occurs, the operation returns a SOAP fault.

How <wsdl:output> and <wsdl:fault> Elements Affect a Consumer

At run time, a consumer web service descriptor expects a SOAP response or SOAP fault based on whether the operation's WSDL has a <wsdl:output> and/or <wsdl:fault> element. The **Pre-8.2 compatibility mode** property dictates whether a web service connector ignores or honors unexpected SOAP responses.

- **When an operation has a <wsdl:output> element (i.e., In-Out MEP operation),** the web service connector expects a SOAP response.

If the operation completes successfully, the web service connector expects a SOAP response that corresponds to the output defined for the operation. The web service connector returns the corresponding output values.

If an exception occurs, the web service connector expects a SOAP fault. The web service connector returns the SOAP fault information as output.

- **When an operation has no <wsdl:output> element and no <wsdl:fault> element (i.e., In-Only MEP operation),** the web service connector expects no SOAP response, even if the operation results in an exception. How the web service connector handles an unexpected SOAP response/fault depends on the **Pre-8.2 compatibility mode** property.

If an operation completes successfully and the provider returns an unexpected SOAP response, when the **Pre-8.2 compatibility mode** property is:

- `false`, the connector ignores the response and does not return output.
- `true`, the connector honors the unexpected response, returning the corresponding output values.

Similarly, if an exception occurs and the provider returns a SOAP fault, when the **Pre-8.2 compatibility mode** property is:

- `false`, the connector ignores the SOAP fault and does not return the SOAP fault as output.

Note:

If an exception is thrown on the consumer side for any reason, the web service connector will return information about the exception in its output.

- `true`, the connector honors the unexpected SOAP fault, returning the SOAP fault information as output.
- **When an operation has no <wsdl:output> element but has a <wsdl:fault> element (i.e., Robust In-Only MEP operation),** the web service connector expects no SOAP response unless an exception occurs. If an exception occurs, the connector receives a SOAP fault, and in turn, returns the SOAP fault information as output.

If the operation completes successfully and returns a SOAP response, how the web service connector handles the unexpected response depends on the **Pre-8.2 compatibility mode** property. When the **Pre-8.2 compatibility mode** property is:

- `false`, the connector ignores the response and does not return output.
- `true`, the connector honors the unexpected response, returning the corresponding output values.

Note: Integration Server does not fully support Robust-In Only for SOAP over JMS bindings. For more information, see [“Consumer Support for Robust In-Only Operations with SOAP/JMS Bindings” on page 115](#).

How Adding Response Headers or Faults Affect In-Only MEP Operations

If you have an In-Only MEP operation, the MEP type changes if you add a response header or a fault to the operation.

- If you add a response header to an In-Only MEP operation, the MEP changes to In-Out MEP.
- If you add a fault to an In-Only MEP operation, the MEP changes to Robust In-Only MEP.

How the MEP Affects the Execution of Handlers

Web service descriptors usually have several handlers assigned to them. A handler is a set of up to three handler services: a request handler service, response handler service, and a fault handler service. Depending on the MEP type, Integration Server might not execute the response and fault handler services. For more information, see [“About Handlers and Handler Services” on page 157](#).

Considerations When Changing a Provider’s Compatibility Mode

Changing a provider web service descriptor’s **Pre-8.2 compatibility mode** property might affect its operations’ MEP types.

- If an operation has defined output, its MEP remains In-Out MEP and does not change.
- If an operation has no defined output, its MEP might change. For more information about how changing the **Pre-8.2 compatibility mode** property affects an operation’s MEP, see [“Impacts of Changing a Provider’s Compatibility Mode to False” on page 100](#) and [“Impacts of Changing a Provider’s Compatibility Mode to True” on page 101](#).

Impacts of Changing a Provider’s Compatibility Mode to False

When you change a provider web service descriptor’s **Pre-8.2 compatibility mode** property from `true` to `false`, In-Out MEP operations with *no* defined output change to In-Only MEP or Robust In-Only MEP.

- **For a service first provider web service descriptor**, when you change the **Pre-8.2 compatibility mode** property to `false`, Integration Server regenerates the WSDL. Because the service has no output, Integration Server does *not* include a `<wsdl:output>` element for the operation. As a

result, the operation changes to In-Only MEP. When the operation executes, it will no longer return a SOAP response.

Integration Server issues a warning message letting you know the MEP has changed. Also, if there are handlers assigned to the web service descriptor, because the operation now uses In-Only MEP, Integration Server will *not* execute the response or fault handler services.

- **For a WSDL first provider web service descriptor**, when you change the **Pre-8.2 compatibility mode** property to `false`, Integration Server begins to use the MEP that matches the operation's WSDL. That is, if the operation:
 - Has no `<wsdl:output>` and `<wsdl:fault>` element, the operation will start using In-Only MEP. When the operation executes, it will no longer return a SOAP response.
 - Has no `<wsdl:output>` element but does have a `<wsdl:fault>` element, the operation will start using Robust In-Only MEP. When the operation executes, it will no longer return a SOAP response if the operation executes successfully, but will return a SOAP fault if an exception occurs.

Integration Server issues a warning message letting you know the MEP has changed. Also, if there are handlers assigned to the web service descriptor:

- If the operation now uses In-Only MEP, Integration Server will *not* execute the response or fault handler services.
- If the operation now uses Robust In-Only MEP, Integration Server will only execute the response handler service if an exception occurs.

Impacts of Changing a Provider's Compatibility Mode to True

For provider web service descriptors, when you change the **Pre-8.2 compatibility mode** property from `false` to `true`, for operations that have *no* defined output, the MEP (either In-Only MEP or Robust In-Only MEP) becomes In-Out MEP.

- **For a service first provider web service descriptor**, when you change the **Pre-8.2 compatibility mode** property to `true`, Integration Server regenerates the WSDL and includes a `<wsdl:output>` element for the operation. As a result, the operation uses In-Out MEP. Integration Server issues a warning message letting you know the MEP has changed.

If handlers are assigned to the web service descriptor, because the MEP is now In-Out MEP, Integration Server starts executing the response and fault handler services.

- **For a WSDL first provider web service descriptor**, when you change the **Pre-8.2 compatibility mode** property to `true`, at run time, Integration Server will use In-Out MEP even when the WSDL for an operation does not include a `<wsdl:output>` element. Integration Server issues a warning message letting you know the MEP has changed.

If handlers are assigned to the web service descriptor, because the MEP is now In-Out MEP, Integration Server starts executing the response and fault handler services.

Note:

The **Pre-8.2 compatibility mode** property and the ability to run in pre-8.2 compatibility mode are deprecated as of Integration Server 10.4 due to the deprecation of the web services implementation introduced in Integration Server version 7.1.

Considerations When Changing a Consumer's Compatibility Mode

Regardless of the **Pre-8.2 compatibility mode** property setting, at run time a consumer web service descriptor always expects a response based on the `<wsdl:output>` and `<wsdl:fault>` elements in the WSDL for an operation. If the provider appropriately provides a response/fault based on the WSDL, changing the **Pre-8.2 compatibility mode** property does not affect the run-time behavior.

The **Pre-8.2 compatibility mode** property setting does dictate whether the web service connector honors or ignores unexpected SOAP responses. For more information, see [“How `<wsdl:output>` and `<wsdl:fault>` Elements Affect a Consumer” on page 99](#).

Note:

The **Pre-8.2 compatibility mode** property and the ability to run in pre-8.2 compatibility mode are deprecated as of Integration Server 10.4 due to the deprecation of the web services implementation introduced in Integration Server version 7.1.

3 Using SOAP over JMS with Web Services

■ Introduction	104
■ Pre-Requisites for Using SOAP/JMS	104
■ Using SOAP/JMS with Provider Web Service Descriptors	104
■ Using SOAP/JMS with Consumer Web Service Descriptors	108
■ Using SOAP/JMS with Web Services with Transactions	112
■ Asynchronously Invoking an In-Out Operation	114
■ Consumer Support for Robust In-Only Operations with SOAP/JMS Bindings	115

Introduction

Integration Server provides support for using SOAP over JMS (SOAP/JMS) bindings with web services based on the SOAP over Java Message Service standard.

When Integration Server acts as a web service client, Integration Server provides support for creating web service clients from WSDL documents that contain SOAP/JMS bindings and sending SOAP messages over JMS to invoke remote web services.

When Integration Server serves as a web service host, Integration Server provides support for creating SOAP/JMS bindings and for retrieving and processing SOAP messages received over JMS.

Each role (consumer or provider) requires configuration in Integration Server and in Designer. As with other web service functionality, Integration Server and Designer use web service descriptors to encapsulate SOAP/JMS binding information. The following sections provide more information about the tasks you need to complete to use SOAP/JMS bindings with web services.

Pre-Requisites for Using SOAP/JMS

To use SOAP/JMS bindings with consumer or provider web service descriptors, the following must be true:

- The web service descriptor must be created using Designer on Integration Server version 8.2 or higher.
- The web service descriptor must have the **Pre-8.2 compatibility mode** property set to false. For more information about compatibility mode, see [“About Pre-8.2 Compatibility Mode” on page 88](#).
- Integration Server must be configured for JMS messaging, which includes creating JMS connection aliases. For more information, see the section *Creating a JMS Connection Alias* in the *webMethods Integration Server Administrator's Guide*.the section *Creating a JMS Connection Alias* in the *webMethods Integration Server Administrator's Guide*.
- One or more destinations must be configured on the JMS provider. If the JMS provider is the webMethods Broker or Software AG Universal Messaging, you can also manage destinations through the JMS connection alias and Designer.

Using SOAP/JMS with Provider Web Service Descriptors

For web services to receive and process SOAP/JMS messages, a provider web service descriptor needs a way to receive JMS messages from a destination on a JMS provider. In addition, the web service descriptor needs to contain SOAP/JMS binding information that can be used in the WSDL document that Integration Server generates for the provider web service descriptor. To provide this information, you create SOAP-JMS triggers and web service endpoint aliases, and then associate these items with a JMS binder in a provider web service descriptor.

The following table identifies the basic tasks that you need to complete to add SOAP/JMS support to a provider web service descriptor.

Step	Description
-------------	--------------------

1	Create a SOAP-JMS trigger.
----------	-----------------------------------

A SOAP-JMS trigger is a JMS trigger that receives messages from a destination (queue or topic) on a JMS provider and then routes the SOAP message to the internal web services stack for processing.

The SOAP-JMS trigger also specifies the JMS connection alias that Integration Server uses to receive messages from the provider and to send response messages to the requesting client. The properties assigned to the SOAP-JMS trigger determine how Integration Server acknowledges the message, provides exactly-once processing, and handles transient or fatal errors.

For more information about creating SOAP-JMS triggers, see *webMethods Service Development Help*.

Note:

Instead of creating a SOAP-JMS trigger, you can create a WS (web service) endpoint trigger at the same time you create a provider web service endpoint alias for JMS. A WS endpoint trigger is a SOAP-JMS trigger with limited configuration options (for example, you cannot configure transient error handling, an acknowledgement mode, or exactly-once processing for WS endpoint triggers). Typically, WS endpoint triggers are used with virtual services deployed to webMethods Mediator. For more information about WS endpoint triggers, see the section *About WS Endpoint Triggers* in the *webMethods Integration Server Administrator's Guide*.the section *About WS Endpoint Triggers* in the *webMethods Integration Server Administrator's Guide*. For more information about webMethods Mediator, see *Administering webMethods Mediator*.

2	Create a provider web service endpoint alias for JMS.
----------	--------------------------------------------------------------

The provider web service endpoint alias for JMS specifies from where and how Integration Server receives messages for a web service descriptor to which the alias is assigned.

The alias specifies a SOAP-JMS trigger which indicates:

- The destination from which the trigger receives messages.
- The JMS connection alias that Integration Server uses to establish a connection to the JMS provider from which it receives messages.

For information about creating an endpoint alias for a provider web service descriptor for use with JMS, see the section *Creating an Endpoint Alias for a Provider Web Service Descriptor for Use with JMS* in the *webMethods Integration Server Administrator's Guide*.the section *Creating an Endpoint Alias for a Provider Web Service Descriptor for Use with JMS* in the *webMethods Integration Server Administrator's Guide*.

3	Add a JMS binder to the provider web service descriptor.
----------	-----------------------------------------------------------------

Step	Description
------	-------------

	For a service first provider web service descriptor, you can add the JMS binder at the time you create the web service descriptor or at a later point. A valid provider web service endpoint alias for JMS must exist before you can add a JMS binder to a web service descriptor. For more information about adding binders, see <i>webMethods Service Development Help</i> .
--	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Note:

You cannot add binders to WSDL first provider web service descriptors. If you want a WSDL first provider web service descriptor to have a JMS binder, you must create the web service descriptor from a WSDL document that contains a SOAP/JMS binding.

4	Assign the provider web service endpoint alias to the JMS binder.
---	--------------------------------------------------------------------------

When the alias is assigned to a JMS binder in a provider web service descriptor, the SOAP-JMS trigger (or WS endpoint trigger) can receive messages that can be processed by the operations in the web service descriptor. The assigned alias creates an association between the SOAP-JMS trigger and the web service descriptor.

The provider web service endpoint alias also includes information that Integration Server needs for generating the WSDL document for a provider web service descriptor. For example, Integration Server uses the SOAP-JMS trigger information to construct most of the JMS URI. Integration Server also uses the alias information to populate the binding elements in the WSDL. This includes JMS message header information for the request message, such as delivery mode and the reply destination.

For more information about assigning web service endpoint aliases to binders, see *webMethods Service Development Help*.

When you save the web service descriptor, Integration Server deploys the web service to the web services stack. At this point, the web service is available for invocation by web service clients. You can use the **WSDL URL** property value for the web service descriptor to obtain the WSDL document for the web service and create web service clients.

Run-Time Behavior for a Provider Web Service that Uses SOAP/JMS

The following table provides an overview of how Integration Server receives and processes SOAP/JMS messages for web services.

- | Step | Description |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | The SOAP-JMS trigger receives JMS messages from the destination it subscribes to on the JMS provider. |
| 2 | <p>Integration Server extracts the SOAP message from the JMS message.</p> <p>Integration Server also retrieves JMS message properties including <code>targetService</code>, <code>soapAction</code>, <code>contentType</code>, and <code>JMSMessageID</code>. These properties specify the web service descriptor and operation that the web service client wants to invoke.</p> <p>Integration Server also retrieves JMS message properties including <code>targetService</code> and <code>soapAction</code>. These properties specify the web service descriptor and operation that the web service client wants to invoke. For more information, see “Determining the Operation for a SOAP/JMS Request” on page 223.</p> |
| 3 | Integration Server passes the SOAP message and JMS message properties to the internal web services stack for processing. |
| 4 | <p>The web services stack processes the SOAP message by executing handler services (if specified) and invoking the web service operation specified in the SOAP request message.</p> <div style="background-color: #f0f0f0; padding: 10px; border: 1px solid #ccc;"> <p>Note: Integration Server also performs authentication and authorization for the web service descriptor, handler services, and the operation. For more information, see “Authentication and Authorization for Provider Web Service Descriptors” on page 190.</p> </div> |
| 5 | For an In-Out operation (or for a Robust-In-Only operation that ends with a SOAP fault), Integration Server constructs the response SOAP message based on the operation output, uses it as the payload in a JMS message, and sends the response to the reply destination specified in the request message. Integration Server uses the JMS connection alias specified in the SOAP-JMS trigger that received the request message to send the response message. |

Additional Guidelines for Using SOAP/JMS with Provider Web Service Descriptors

When using SOAP/JMS with provider web service descriptors, keep the following information in mind:

- After assigning an alias to a JMS binder in a provider web service descriptor, the web service descriptor has a dependency on the SOAP-JMS trigger. Consequently, at start up or when reloading the package containing the web service descriptor, Integration Server must load the SOAP-JMS trigger before loading the web service descriptor. If the SOAP-JMS trigger and web service descriptor are not in the same package, you need to create a package dependency. The package that contains the web service descriptor must have a dependency on the package that contains the SOAP-JMS trigger. For information about creating package dependencies, see *webMethods Service Development Help*.

- A SOAP-JMS trigger receives a JMS message from a destination and passes the SOAP message used as the payload on to the web services stack for processing. The web services stack determines which operation to invoke. It is possible for the web services stack to invoke an operation in a provider web service descriptor that is not affiliated with the SOAP-JMS trigger through a web service endpoint alias.
- If you suspend or disable a SOAP-JMS trigger, the SOAP-JMS trigger will not retrieve any messages from the destination to which the trigger subscribes.
- If you delete a SOAP-JMS trigger specified in a provider web service endpoint alias, any provider web service descriptor with a binder to which the alias is assigned cannot be deployed to the web services stack. As a result, a WSDL document will not be available for the provider web service descriptor.
- If you rename a SOAP-JMS trigger, you need to update any provider web service endpoint alias that uses that trigger with the new name.
- The transient error handling for the SOAP-JMS trigger overrides any transient error handling configured for the service used as the operation. For details about how Integration Server handles transient errors for web services that process SOAP/JMS messages, see [“Transient Error Handling for Provider Web Service Descriptors” on page 205](#).
- Integration Server does not perform any transient error handling for handler services. Integration Server ignores any transient error handling properties configured for the service handlers and does not use the transient error handling defined for the SOAP-JMS trigger. The handler services need to be coded to handle any errors or exceptions that may occur and return the appropriate status code.
- When creating the reply JMS message, Integration Server uses the same message type as the request (BytesMessage or TextMessage). However, if the server parameter `watt.server.soapjms.defaultMessageType` is set to `TextMessage`, Integration Server overrides the request and sends the response as `TextMessage`. It may be useful to send the response as a `TextMessage` for debugging purposes.
- Integration Server has additional guidelines for using a transacted SOAP-JMS trigger with a web service descriptor. For more information about using transactions with web service descriptors and SOAP/JMS, see [“Using SOAP/JMS with Web Services with Transactions” on page 112](#).

Using SOAP/JMS with Consumer Web Service Descriptors

To use Integration Server as a web service client that sends SOAP over JMS messages, you need a service that creates a SOAP message, places it in a JMS message, and then sends the JMS message to the JMS provider. Integration Server creates a web service connector that does all of these tasks when you create a consumer web service descriptor from a WSDL document with SOAP/JMS bindings. In addition to creating the consumer web service descriptor, you can specify how Integration Server will connect to the JMS provider and supplement or overwrite JMS URI information provided in the WSDL document.

The following table identifies the basic steps that you need to complete to use Integration Server as a web service client that sends SOAP/JMS messages.

Step	Description
-------------	--------------------

1	Create a consumer web service descriptor.
----------	--------------------------------------------------

When you create a consumer web service descriptor from a WSDL document that contains a SOAP/JMS binding, Integration Server creates a JMS binder for the web service descriptor. The JMS binder encapsulates the information needed to send a message to the JMS provider, including the destination and JMS message header values.

For more information about creating a consumer web service descriptor, see *webMethods Service Development Help*.

Note:

Binders cannot be added to consumer web service descriptors.

2	Optionally, create a consumer web service endpoint alias for JMS.
----------	--------------------------------------------------------------------------

If you want to supplement or replace the binding information contained in the WSDL document, create a consumer web service endpoint alias.

Note:

The SOAP over Java Message Service standard requires only that the WSDL document contain the lookup variant and the destination. Consequently, it is possible that some of the details needed to connect to the JMS provider are absent from the JMS binder.

For more information about creating a consumer web service endpoint alias for use with JMS, see the section *Creating an Endpoint Alias for a Consumer Web Service Descriptor for Use with JMS* in the *webMethods Integration Server Administrator's Guide*.the section *Creating an Endpoint Alias for a Consumer Web Service Descriptor for Use with JMS* in the *webMethods Integration Server Administrator's Guide*.

3	Optionally, assign the consumer web service endpoint alias to the JMS binder.
----------	--------------------------------------------------------------------------------------

If you created a web service endpoint alias that you want to use to supplement or replace the JMS URI information from the WSDL document, assign the alias to the JMS binder.

For more information about assigning web service endpoint aliases to binders, see *webMethods Service Development Help*.

4	Configure use of the client side queue.
----------	------------------------------------------------

The client side queue is a message store that contains JMS messages sent when the JMS provider was not available. Messages remain in the client side queue until the JMS provider becomes available. For each JMS binder, you can decide whether or not to use the client side queue.

For more information about configuring the client side queue for a JMS binder, see *webMethods Service Development Help*

Step	Description
------	-------------

5	Build a service that invokes the web service connector.
---	----------------------------------------------------------------

When you create a service that invokes a web service connector, you supply values to use as input to the web service connector. These values determine the content of the SOAP message request.

When using a SOAP/JMS binding, you can pass in name/value pairs to the *transportHeaders* parameter. Integration Server creates JMS message headers and properties from the name/value pairs. You can also use name/value pairs to overwrite binding details specified in the source WSDL document or in the consumer web service endpoint alias assigned to the JMS binder.

For more information about the web service connector signature, see [“Signature for a Web Service Connector” on page 118](#).

For more information about supplying transport headers and name/value pairs, see [“Setting Transport Headers for JMS” on page 146](#).

For information about supplying security information into a web service connector, see [“Passing Message-Level Security Information to a Web Service Connector” on page 150](#).

Run-Time Behavior for a Web Service Connector that Uses a SOAP/JMS Binding

The following table provides an overview of how Integration Server receives and processes SOAP/JMS messages for web services.

Step	Description
------	-------------

1	Integration Server invokes the web service connector.
---	-------------------------------------------------------

Note: Integration Server authorizes the user at various points during web service connector execution and handler execution. For more information, see [“Authentication and Authorization for Consumer Web Service Descriptors” on page 186](#).

2	Integration Server builds the SOAP request message using the input passed into the web service connector and information in the JMS binder.
---	---------------------------------------------------------------------------------------------------------------------------------------------

3	Integration Server executes the request handler services.
---	-----------------------------------------------------------

For more information about request handler services, see [“About Request Handler Services” on page 159](#).

4	Integration Server builds the JMS message, using the SOAP request message as the payload. Integration Server uses the name/value pairs passed into <i>transportHeaders</i> input parameter to create JMS message headers and properties.
---	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Step	Description
	Integration Server also uses information in the JMS binder to construct the JMS message.
5	Integration Server sends the JMS message to the JMS provider.
6	<p>If the JMS provider is not available at the time the Integration Server sends the JMS message, one of the following occurs:</p> <ul style="list-style-type: none"> ■ If the client side queue is enabled for the JMS binder, Integration Server writes the JMS message to the client side queue. Integration Server sends the message to the JMS provider when the JMS provider becomes available. ■ If the client side queue is not enabled for the JMS binder, the web service connector throws an <code>ISRuntimeException</code> when the JMS provider is not available.
7	<p>After Integration Server sends the JMS message, one of the following occurs:</p> <ul style="list-style-type: none"> ■ For a synchronous In-Out operation, Integration Server creates a message consumer for the <code>replyTo</code> destination. Integration Server uses the message consumer to retrieve the response JMS message sent to the reply destination, extracts the SOAP response message, and executes response handlers. Integration Server populates the web service connector output. ■ For an asynchronous In-Out operation, Integration Server populates the web service connector output. Integration Server does not populate <code>responseHeaders</code>. ■ For an In-Only and Robust In-Only operations, Integration Server populates the web service connector output. <p>The contents of the <code>transportInfo</code> output parameter vary based on the operation MEP and the success or failure of the web service connector. For more information, see “About Transport and Fault Information Returned by a Web Service Connector” on page 139.</p>

Additional Guidelines for Using SOAP/JMS with Consumer Web Service Descriptors

When using SOAP/JMS with consumer web service descriptors, keep the following information in mind:

- The server parameter `watt.server.soapjms.defaultMessageType` specifies the default message type for web service request messages sent using SOAP over JMS. This parameter can be set to `BytesMessage` or `TextMessage`. The default value is `BytesMessage`. The default message type can be overwritten during web service connector execution by setting the `jms.messageType` property in the `transportHeaders` input parameter.
- The client side queue can be used with web service connectors for In-Only operations and In-Out operations only. Do not use the client side queue with Robust-In-Only operations.

Note:

If you want to use the client side queue with an In-Out operation, you must send the request message asynchronously. For more information about sending an asynchronous request message for an In-Out operation, see [“Asynchronously Invoking an In-Out Operation” on page 114](#).

- The client side queue can be used with a web service descriptor only if the consumer web service endpoint alias uses a JMS connection alias to connect to a JMS provider.
- To receive a response for a synchronous In-Out operation, Integration Server creates a message consumer for the replyTo destination immediately after sending the JMS message request to the JMS provider.
- Integration Server provides partial support for using Robust-In-Only operations with SOAP/JMS bindings. For more information, see [“Consumer Support for Robust In-Only Operations with SOAP/JMS Bindings” on page 115](#).
- A web service connector that sends a JMS message can throw an `ISRuntimeException`. However, Integration Server places the `ISRuntimeException` in the *fault* document returned by the connector. If you want the parent flow service to catch the transient error and rethrow it as an `ISRuntimeException`, you must code the parent flow service to check the *fault* document for an `ISRuntimeException` and then throw the `ISRuntimeException` explicitly.
- Integration Server has additional guidelines for using transactions with SOAP/JMS bindings. For more information about using transactions with web service descriptors and SOAP/JMS, see [“Using SOAP/JMS with Web Services with Transactions” on page 112](#).

Using SOAP/JMS with Web Services with Transactions

Integration Server provides support for sending and receiving SOAP/JMS messages as part of a transaction. A *transaction* is a logical unit of work composed of one or more interactions with one or more resources. The interactions within a transaction are either all committed or all rolled back. A transaction either entirely succeeds or has no effect at all.

Integration Server has different guidelines and requirements for using SOAP/JMS with transactions depending on whether Integration Server acts as the web service provider or the web service client.

- For information about receiving SOAP/JMS messages for web services as part of a transaction, see [“Guidelines for Using Transactions with SOAP/JMS and Provider Web Service Descriptors” on page 112](#).
- For information about using web service connectors to send SOAP/JMS messages as part of a transaction, see [“Guidelines for Using Transactions with SOAP/JMS and Consumer Web Service Descriptors” on page 113](#).

Guidelines for Using Transactions with SOAP/JMS and Provider Web Service Descriptors

If you want to receive and process SOAP/JMS messages for web services as part of a transaction, keep the following requirements and information in mind:

- The WmART package must be installed and enabled.
- The SOAP-JMS trigger must be transacted. A transacted SOAP-JMS trigger is one that executes as part of a transaction. A transacted SOAP-JMS trigger uses a transacted JMS connection alias. A JMS connection alias is considered to be transacted when it has a transaction type of `XA_TRANSACTION` or `LOCAL_TRANSACTION`. For information about creating a JMS connection alias, see the section *Creating a JMS Connection Alias* in the *webMethods Integration Server Administrator's Guide*.the section *Creating a JMS Connection Alias* in the *webMethods Integration Server Administrator's Guide*.
- Transactions can be used with In-Only operations only. Use a transacted SOAP-JMS trigger to receive messages for web services with In-Only operations only. Do not use a transacted SOAP-JMS trigger to receive messages for web services that have Robust-In-Only or In-Out operations.
- Transactions cannot be used with service handlers. Do not use a transacted SOAP-JMS trigger with a provider web service descriptor that has service handlers.
- The execution of a transacted SOAP-JMS trigger is an implicit transaction. In an implicit transaction, Integration Server starts and completes the transaction automatically. You do not need to code your service to include any of the transaction management services in the WmART package. Integration Server starts the implicit transaction when it receives the SOAP/JMS message from the JMS provider. Integration Server implicitly commits or rolls back the transaction based on the success or failure of the trigger service.
 - Integration Server commits the transaction if the web service operation executes successfully.
 - Integration Server rolls back the transaction if the web service operation fails with an `ISRuntimeException` (a transient error). For detailed information about how Integration Server handles a transient error within a transaction, see [“Transient Error Handling for an Operation Invoked by a Transacted SOAP-JMS Trigger”](#) on page 212.
 - Integration Server rolls back the transaction if the web service operation fails with a `ServiceException` (a fatal error).

Guidelines for Using Transactions with SOAP/JMS and Consumer Web Service Descriptors

If you want to send SOAP/JMS messages as part of a transaction, keep the following guidelines in mind:

- The WmART package must be installed and enabled.
- The consumer web service endpoint alias assigned to the JMS binder must use a transacted JMS connection alias. A JMS connection alias is considered to be transacted when it has a transaction type of `XA_TRANSACTION` or `LOCAL_TRANSACTION`. For information about creating a JMS connection alias, see the section *Creating a JMS Connection Alias* in the *webMethods Integration Server Administrator's Guide*.the section *Creating a JMS Connection Alias* in the *webMethods Integration Server Administrator's Guide*.

- SOAP/JMS messages can be sent as part of a transaction for operations with any message exchange pattern (MEP). However, if you want to use a transaction to send a SOAP/JMS message for an In-Out MEP, you must send the request asynchronously. For more information, see [“Asynchronously Invoking an In-Out Operation” on page 114](#). Keep in mind that Integration Server offers limited support for Robust-In-Only operations and asynchronous In-Out operations. For more information, see [“Asynchronously Invoking an In-Out Operation” on page 114](#) and [“Consumer Support for Robust In-Only Operations with SOAP/JMS Bindings” on page 115](#).
- The client side queue cannot be used when SOAP/JMS messages are sent as part of a transaction. Do not enable the client side queue for a JMS binder if messages will be sent as part of a transaction.
- SOAP/JMS messages can be sent as part of an implicit transaction or an explicit transaction.
 - Integration Server starts an explicit transaction when the `pub.art.transaction:startTransaction` service executes. The explicit transaction ends when the `pub.art.transaction:commitTransaction` service or the `pub.art.transaction:rollbackTransaction` service executes. You can also allow Integration Server to commit or roll back the transaction implicitly based on the success or failure of the service.
 - Integration Server starts an implicit transaction when Integration Server sends the SOAP/JMS message using a transacted JMS connection alias. Integration Server only starts an implicit transaction if another service has not yet started a transaction. Integration Server implicitly commits the transaction when the top-level service succeeds. Integration Server implicitly rolls back the transaction if the top-level service fails.

Asynchronously Invoking an In-Out Operation

Integration Server provides limited support for using a web service connector to asynchronously execute an In-Out operation with a SOAP/JMS binding. Keep the following information in mind when using a web service connector to execute an In-Out operation asynchronously:

- For the web service connector, you must pass `jms.async=true` into the *transportHeaders* input parameter.
- To instruct Integration Server to write the request message for an asynchronous request/reply to the client side queue when the JMS provider is not available, the JMS binder must be configured to use the client side queue.
- When a web service connector sends an asynchronous request, it executes to completion without populating any response headers for the *transportInfo/responseHeaders* output parameter
- Even though a web service connector does not wait for a SOAP response when invoked asynchronously, it will execute the response handlers assigned to the consumer web service descriptor. Because the *messageContext* that is available to handler services will not contain a response message, handler services that operate on the response message will not provide much, if any, benefit. However, in an asynchronous request/reply, it might be useful to execute response handler services that perform activities such as cleaning up request handler invocation.

- To retrieve the SOAP response to an asynchronous request from the provider, you need to create a custom solution to receive and process the response. This might include using a standard JMS trigger to receive the request from the reply destination and then invoking a trigger service that uses the `pub.soap*` services to process the SOAP message. You could also create an on-demand message consumer using the `pub.jms*` services to receive the message and then using the `pub.soap*` services to process the SOAP message.
- Using a JMS trigger or message consumer to receive the response bypasses any response handlers or policies applied to the SOAP response, including any `WS-SecurityPolicy`. The SOAP response does not undergo any processing provided by the response handlers or policies attached to the consumer web service descriptor. Any response messages that require decryption or authentication will not be usable. Consequently, do not use an asynchronous request/reply to invoke an In-Out operation to which the `WS-SecurityPolicy` is applied

Consumer Support for Robust In-Only Operations with SOAP/JMS Bindings

For a consumer web service descriptor, Integration Server provides partial support for Robust In-Only operations with a SOAP/JMS binding. When Integration Server creates a consumer web service descriptor from a WSDL that contains a Robust In-Only operation and that operation is defined as part of a portType with a SOAP over JMS binding, Integration Server populates the reply destination in the JMS message header (the `JMSReplyTo` header field), but otherwise treats the operation as In-Only.

Specifically, the generated web service connector will not produce or wait for any output besides the `transportInfo` parameter. If an exception occurs while the provider processes the request, the web service connector does not retrieve or process the SOAP response.

If you want to retrieve a SOAP response (which includes the SOAP fault) that the provider sends when an exception occurs during web service execution, you need to receive and process the response with a custom solution. This might include using a standard JMS trigger or an on-demand message consumer created using the `pub.jms*` services to receive the message and using the `pub.soap*` services to process the SOAP message.

Note:

Using a JMS trigger or message consumer to receive the response bypasses any policies applied to the SOAP response and any response handlers assigned to the consumer web service descriptor.

Note:

The SOAP over Java Message Service standard considers the Robust In-Only message exchange pattern (MEP) to be non-normative and does not define the MEP. The SOAP over Java Message Service standard considers a solution for non-normative patterns to be proprietary. Other vendors might not interoperate with these solutions.

4 Working with Web Service Connectors

■ About Web Service Connectors	118
■ Signature for a Web Service Connector	118
■ How a SOAP Fault is Mapped to the Generic Fault Output Structure	144
■ Setting Transport Headers for HTTP/S	145
■ Setting Transport Headers for JMS	146
■ Passing Message-Level Security Information to a Web Service Connector	150

About Web Service Connectors

A web service connector is a flow service that Integration Server creates at the time it creates the consumer web service descriptor. A web service connector contains the information and logic needed to invoke an operation defined in the WSDL document used to create the consumer web service descriptor.

When creating a consumer web service descriptor from a WSDL document, Integration Server creates a web service connector for each operation contained in the WSDL document. For example, if a WSDL document contains two portType declarations and each portType contains three operations, Integration Server creates six web service connectors. Or, if a WSDL document contains two portTypes where the first portType contains three operations and the second portType contains two operations, Integration Server creates five web service connectors.

A web service connector:

- Uses an input and output signature that corresponds to the input message, output message, and headers defined for the operation in the WSDL document. The web service connector signature also contains optional inputs that you can use to control the execution of logic in the web service connector.
- Represents a generic fault structure in the output signature differently based on the version of the Integration Server on which the web service descriptor is created. To learn more about the output signature of a web service connector, see [“Signature for a Web Service Connector” on page 118](#).
- Contains flow steps that create and send a message to the web service endpoint using the transport, protocol, and location information specified in the web service’s WSDL document in conjunction with input supplied to the web service connector.
- Contains flow steps that extract data or fault information from the response message returned by the web service.

Important:

Do not edit the flow steps in a web service connector.

Note:

A web service connector that worked correctly with previous versions of Integration Server should continue to work with version 8.2 and later. In addition, any external clients created from WSDL generated from previous versions of Integration Server should continue to work as they did in the previous version.

Signature for a Web Service Connector

Important:

The web service connector signature cannot be modified.

All web service connectors have an identical input and output signature with the exception of:

- The variables used to represent the input and output messages.

For information about how a web service connector represents the input and output messages in the signature, see [“How a Web Service Connector Represents the Input and Output Messages” on page 138](#).

- The format of the fault structure in the output signature is based on the version of the Integration Server on which the web service descriptor is created.
 - When the web service descriptor is created on Integration Server 8.2, the output signature of the web service connector contains a generic fault structure.
 - For information about how Integration Server maps the contents of a SOAP 1.1 or SOAP 1.2 fault to the generic fault structure, see [“How a SOAP Fault is Mapped to the Generic Fault Output Structure” on page 144](#).
- When the web service descriptor is created on versions of Integration Server prior to 8.2, the output signature of the web service connector contains a SOAP fault document that is specific to the SOAP protocol (i.e., SOAP 1.1 or SOAP 1.2).

Input Parameters

auth

Document Optional. Specifies the transport-level and message-level credentials to include in the request.

Integration Server uses the information provided in *auth* to create the HTTP request and the SOAP request.

Note:

Information specified in *auth* overwrites any authentication credentials specified in the consumer web service endpoint alias that is assigned to the binder used by the web service connector.

Key	Description
<i>transport</i>	<p>Document Optional. Specifies the transport-level credentials to include in the HTTP request. Integration Server uses the information specified in the <i>transport</i> variable to populate the <i>Authorization</i> header in the HTTP request.</p> <p>You only need to provide credentials in <i>transport</i> if the endpoint URL specifies HTTPS and you want to overwrite the credentials specified in the consumer web service endpoint alias assigned to the binder.</p> <p>Note: If the <i>Authorization</i> header is passed into <i>transportHeaders</i>, the values specified for the <i>transport</i> document and its children will not be used in the <i>Authorization</i> header.</p>
Key	Description

<i>type</i>	<p>String Optional. Specifies the type of authentication required by the host.</p> <ul style="list-style-type: none">■ Specify <code>Basic</code> to use basic authentication (user name and password)■ Specify <code>Digest</code> to use password digest to authenticate the credentials.■ Specify <code>NTLM</code> to use NTLM authentication.■ If <i>type</i> is not specified or if any value other than <code>Basic</code>, <code>Digest</code>, or <code>NTLM</code> is specified, Integration Server uses <code>Basic</code>.
<i>user</i>	<p>String Optional. User name used to authenticate the consumer at the HTTP or HTTPS transport level on the host server.</p> <div><p>Note: If you have specified <code>NTLM</code> as <i>type</i>, you must specify <i>user</i> in the following format:</p><p><i>domain_name\user_name</i></p></div>
<i>pass</i>	<p>String Optional. Password used to authenticate the consumer on the host server.</p>
<i>serverCerts</i>	<p>Document Optional. The message signer's private key and certificate chain.</p> <ul style="list-style-type: none">■ <i>keyStoreAlias</i> String Optional Alias to the keystore that contains the private key used to connect to the web service host securely.■ <i>keyAlias</i> String Optional. Alias to the key in the keystore that contains the private key used to connect to the web service host securely. The key must be in the keystore specified in <i>keyStoreAlias</i>.
<i>message</i>	<p>Document Optional. Specifies the message-level credentials to include in the WS-Security SOAP headers included in the SOAP request.</p>

For information about specifying message-level credentials in a web service connector, see [“Passing Message-Level Security Information to a Web Service Connector”](#) on page 150.

Note:

You cannot use message level authentication for consumer web service descriptors for which reliable messaging is enabled.

Key	Description
<i>user</i>	<p>String Optional. User name used to authenticate the consumer at the message level.</p> <p>Specify a value for <i>user</i> if the SOAP message request requires credentials for a UsernameToken.</p>
<i>pass</i>	<p>String Optional. Password used to authenticate the consumer at the message level.</p> <p>Specify a value for <i>pass</i> if the SOAP message request requires credentials for a UsernameToken.</p>
<i>serverCerts</i>	<p>Document Optional. The message signer’s private key and certificate chain.</p> <ul style="list-style-type: none"> ■ <i>keyStoreAlias</i> String Optional. Alias to the keystore that contains the private key used to: <ul style="list-style-type: none"> ■ Sign outbound SOAP requests ■ Include an X.509 authentication token for outbound SOAP requests ■ Decrypt inbound SOAP responses <div style="background-color: #f0f0f0; padding: 10px; margin: 10px 0;"> <p>Note: To verify messages from this consumer, the web services provider must have a copy of the corresponding public key.</p> </div> <ul style="list-style-type: none"> ■ <i>keyAlias</i> String Optional. Alias to the private key used to sign and/or include X.509 authentication token for

- outbound SOAP messages and/or decrypt inbound SOAP responses. The key must be in the keystore specified in *keyStoreAlias*.
- partnerCert* **Object** Optional. The partner's complete certificate chain, where element 0 in the list contains the message signer's certificate and element 1 contains the CA's certificate.
- kerberos Settings* **Document** Optional. Kerberos-related details that will be used to provide Kerberos authentication for web service requests.
- *jaasContext* **String** Optional. The custom JAAS context used for Kerberos authentication.

Integration Server includes a JAAS context named `IS_KERBEROS_OUTBOUND` that can be used with outbound requests.
 - *clientPrincipal* **String** Optional. The name of the client principal to use for Kerberos authentication.

The `IS_KERBEROS_OUTBOUND` JAAS context does not include a `principal` parameter. You must specify *clientPrincipal* if you specified `IS_KERBEROS_OUTBOUND` as the *jaasContext* value.
 - *clientPassword* **String** Optional. The password for the specified client principal. You must specify *clientPassword* if the specified *jaasContext* does not specify a keytab file.
 - *servicePrincipal* **String** Optional. The name of the principal that is used with the service that the Kerberos client wants to access. This can be obtained from the WSDL document published by the provider of Kerberos service. Specify the Service Principal Name in the following format:


```
principal-name.instance-name@realm-name
```

- **servicePrincipalForm String** Optional. The format in which you want to specify the principal name of the service that is registered with the principal database. Specify *host-based* to represent the principal name using the service name and the hostname, where hostname is the host computer. Specify *username* to represent the principal name as a named user defined in the LDAP or central user directory used for authentication to the KDC.

Note:

When using Kerberos authentication, principal name and principal password can be specified in the JAAS context file and in the web service connector. If the principal name and password are specified in the JAAS context file supplied to *jaasContext* and in the web service connector inputs, the values in the JAAS context file take precedence.

timeout

String Optional. Time (measured in milliseconds) to wait for a response from the server hosting the web service before timing out and terminating the request.

If *timeout* is not specified or specifies a value < 0 , Integration Server uses one of the following values:

- For HTTP, Integration Server uses the value of the `watt.server.SOAP.request.timeout` server property as the *timeout* value.
- For JMS, Integration Server uses the value of the `watt.server.soapjms.request.timeout` server property as the *timeout* value.

For more information about server configuration properties, see *webMethods Integration Server Administrator's Guide*.

A *timeout* value of 0 means Integration Server waits for a response indefinitely. If the connection to the host or JMS provider ends before Integration Server receives a response, the web service connector ends with an exception and a status code of 408.

Integration Server ignores *timeout* if the name/value pair `jms.async=true` is passed in to *transportHeaders*.

- _port* **String** Optional. Specifies the port that Integration Server uses to invoke the operation represented by the web service connector. You only need to specify a value for *_port* when the `portType` in the WSDL is associated with multiple bindings. Reference the WSDL document to determine the names of the ports associated with the bindings for a given `portType`.
- If you do not specify a value for *_port*, Integration Server uses the first port defined in the WSDL for the web service.
- _url* **String** Optional. The URL to use as the endpoint URL for the web service. If supplied, the value of *_url* overwrites the endpoint URL in the original WSDL.
- If a consumer web service endpoint alias is assigned to the binder used by the web service connector, the host and/or port in the consumer web service endpoint alias overwrite the host and/or port specified in *_url*. For more information about how Integration Server constructs the endpoint URL, see [“How Integration Server Builds the Consumer Endpoint URL” on page 218](#).
- endpointAlias* **String** Optional. The endpoint alias to use as the endpoint alias for the web service. If supplied, the value of *endpointAlias* overwrites the endpoint alias in the original WSDL.
- To invoke the operations represented by the web service connector, you must have the credentials and user group membership permission to execute this endpoint alias as specified in its **Execute ACL** property.
- Note:**
The *endpointAlias* parameter is not available in Integration Server versions prior to 9.5 SP1. To include this parameter in the signature of web service connectors created in previous versions of Integration Server, refresh the web service connectors.
- transport Headers* **Document** Optional. Transport-specific header fields that you want to explicitly set in the request issued by the web service connector. Specify a key in *transportHeaders* for each header field that you want to set, where the key's name represents the name of the header field and the key's value represents the value of that header field.
- The names and values supplied to *transportHeaders* must be of type String. For information about using *transportHeaders* with HTTP/S requests including a description of the default Integration Server behavior, see [“Setting Transport Headers for HTTP/S” on page 145](#).
- For information about using *transportHeaders* with JMS requests including a description of the default Integration Server behavior, see [“Setting Transport Headers for JMS” on page 146](#).

*message
Addressing
Properties*

Document Optional. WS-Addressing-specific header fields that you want to explicitly set in the request issued by the web service connector. Integration Server uses this information to specify addressing information for a SOAP message, such as the message's destination or where to reply to the message, without relying on transport-specific headers.

For more information about WS-Addressing, see [“Web Services Addressing \(WS-Addressing\)” on page 321](#).

Note:

The *messageAddressingProperties* parameter is not available in Integration Server versions prior to 9.0.

Key	Description						
<i>messageID</i>	String Optional. Unique identifier of the SOAP message. If no message ID is specified, Integration Server generates a unique ID.						
<i>to</i>	<p>Document Optional. The endpoint reference to which you are sending the SOAP message.</p> <table> <tr> <th>Key</th><th>Description</th></tr> <tr> <td><i>address</i></td><td>Address of the intended receiver of the message and its attributes, which includes namespaceName, localname, and their values.</td></tr> <tr> <td><i>referenceParameters</i></td><td>Parameters that correspond to <wsa:ReferenceParameters> properties of the endpoint reference to which the message is addressed.</td></tr> </table>	Key	Description	<i>address</i>	Address of the intended receiver of the message and its attributes, which includes namespaceName, localname, and their values.	<i>referenceParameters</i>	Parameters that correspond to <wsa:ReferenceParameters> properties of the endpoint reference to which the message is addressed.
Key	Description						
<i>address</i>	Address of the intended receiver of the message and its attributes, which includes namespaceName, localname, and their values.						
<i>referenceParameters</i>	Parameters that correspond to <wsa:ReferenceParameters> properties of the endpoint reference to which the message is addressed.						
<i>from</i>	<p>Document Optional. The endpoint reference that specifies the source of the SOAP message. The <i>from</i> input parameter includes:</p> <ul style="list-style-type: none"> ■ <i>attributes</i>, which includes namespaceName, localname, and their values. ■ <i>address</i> and its attributes and values. ■ <i>referenceParameters</i> ■ <i>metadata</i> and its attributes and elements. ■ <i>extensibleElements</i>, which are any other elements usually provided for future extensions. 						

replyTo **Document** Conditional. The endpoint reference that specifies the destination address of the response (reply) message. The *replyTo* input parameter includes:

- *attributes*, which includes namespaceName, localname, and their values.
- *address* and its attributes and values.
- *referenceParameters*
- *metadata* and its attributes and elements.
- *extensibleElements*, which are any other elements usually provided for future extensions.

faultTo **Document** Conditional. The endpoint reference that specifies the address to which the SOAP fault messages are routed. The *faultTo* input parameter includes:

- *attributes*, which includes namespaceName, localname, and their values.
- *address* and its attributes and values.
- *referenceParameters*
- *metadata* and its attributes and elements.
- *extensibleElements*, which are any other elements usually provided for future extensions.

mustUnderstand **String** Optional. Specifies whether the recipients (the actor or role to which the header is targeted) are required to process the WS-Addressing headers. Recipients that cannot process a mandatory WS-Addressing header reject the message and return a SOAP fault.

Value	Description
true	Indicates that processing the WS-Addressing headers is required by the recipients (the actor or role to which the header is targeted).If you specify True for <i>mustUnderstand</i> and the SOAP node receives a header that it does not understand or cannot process, it returns a fault.
false	Indicates that processing the WS-Addressing headers is optional. This is the default.

soapRole **String** Optional. Target of the WS-Addressing headers in the SOAP message. *soapRole* determines the value of the role attribute of the WS-Addressing headers. The actor or role attribute specifies a URI for the recipient of WS-Addressing header entries. *soapRole* can be any valid URI that you specify or any of the following predefined roles.

Value	Description
Ultimate Receiver	Indicates that the recipient is the ultimate destination of the SOAP message. This is the default.
Next	Specifies the following URI for the role attribute: <ul style="list-style-type: none"> ■ For SOAP 1.2: http://www.w3.org/2003/05/soap-envelope/role/next ■ For SOAP 1.1: http://schemas.xmlsoap.org/soap/actor/next
None	Specifies http://www.w3.org/2003/05/soap-envelope/role/none for the role attribute.

reliable Messaging Properties **Document** Optional. WS-ReliableMessaging-specific header fields that you want to explicitly set in the request issued by the web service connector. Integration Server uses this information to specify reliable messaging information for a SOAP message. For more information about WS-ReliableMessaging, see [“Web Services Reliable Messaging \(WS-ReliableMessaging\)” on page 335](#).

Note:

The *reliableMessagingProperties* parameter is not available in Integration Server versions prior to 9.0.

Key	Description
<i>sequenceKey</i>	<p>String Optional. Unique identifier of the SOAP message sequence.</p> <p>A reliable messaging client associates a sequence key to a message sequence based on the endpoint URL to which the message sequence is directed. In cases where there are several message sequences directed to the same endpoint URL, you can specify a custom sequence key to identify each sequence. Each sequence is then uniquely identified by the endpoint URL and the user-specified sequence key.</p>

Note:

The user-specified sequence key should not exceed 32 characters in length.

You use the `pub.soap.wsm:createSequence` service to create a new message sequence. You can specify the sequence key as the input of this service.

isLast Message **String** Optional. Whether the message issued by the web service connector is the last message in the specified sequence key.

Value	Description
false	Specifies that the message is not the last message in the sequence. This is the default. This is the default.
true	Specifies that the message is the last message in the sequence.

message Number **String** Optional. Specifies the message number in the message sequence. In cases where there are several message numbers in the message sequence, you can specify a custom message number.

acksTo **Document** Optional. Consumer response endpoint address to which the reliable message destination must send the acknowledgement. To specify the consumer response endpoint address, use the **Response endpoint address template** binder property of the consumer web service descriptor for which the reliable message sequence is to be created, as the address template and replace the placeholders `<server>` and `<port>` with appropriate values.

If no address is specified as *acksTo*, the acknowledgement messages are sent back to the *replyTo* address. If no address is specified as *replyTo*, the acknowledgement messages are sent back to the requester.

Key	Description
address	Document Document that contains the address to which the acknowledgement message is to be sent. <ul style="list-style-type: none">■ <i>value</i> String Address to which the acknowledgement message is to be sent.

Output Parameters

transportInfo **Document** Conditional. Headers from response and request messages.

The contents of the *transportInfo* vary depending on the actual transport (HTTP, HTTPS or JMS) used by the connector.

transportInfo contains the following keys:

Key	Value
<i>requestHeaders</i>	<p>Document Conditional. Header fields from the request message. The contents of the <i>requestHeaders</i> document are not identical to <i>transportHeaders</i> used as input to the web service connector. The transport can add, remove, or alter specific headers while processing the request.</p> <p>Whether or not the web service connector returns the <i>requestHeaders</i> parameter depends on the success or failure of the connector. In the case of failure, the point at which the failure occurs determines the presence of the <i>requestHeaders</i> parameter. For more information, see “About Transport and Fault Information Returned by a Web Service Connector” on page 139.</p> <ul style="list-style-type: none"> ■ For the HTTP or HTTPS transports, the <i>requestHeaders</i> parameter will not contain any HTTP headers that the transport mechanism added or modified when sending the request. ■ For the JMS transport, each key in <i>requestHeaders</i> represents a JMS message header. Key names represent the names of header fields. Key values are Strings containing the values of the header fields. <p>The JMS provider populates some JMS message header fields after it successfully receives the JMS message. Additionally, the Integration Server specific run-time properties (properties that begin with the “jms.” prefix) are not returned in <i>requestHeaders</i>. The JMS provider uses the information in these properties to populate the JMS message header fields that correspond to the properties.</p>
<i>responseHeaders</i>	<p>Document Conditional. Header fields from the response. Each key in <i>responseHeaders</i> represents a</p>

field (line) of the response header. Key names represent the names of header fields. The keys' values are Strings containing the values of the fields.

Whether or not the web service connector returns the *responseHeaders* parameter depends on the success or failure of the connector. In the case of failure, the point at which the failure occurs determines the presence of the *responseHeaders* parameter. For more information, see [“About Transport and Fault Information Returned by a Web Service Connector”](#) on page 139.

- **For the HTTP or HTTPS transports**, the *responseHeaders* parameter contains any HTTP/HTTPS headers present in the response.
- **For the JMS transport**, the *responseHeaders* parameter contains the JMS headers present in the response.

status **String** Conditional. Status code from the request, returned by the underlying transport.

For more information about status codes and status messages returned by a web service connector, see [“About Transport and Fault Information Returned by a Web Service Connector”](#) on page 139.

statusMessage **String** Conditional. Description of the status code returned by the transport.

For more information about status codes and status messages returned by a web service connector, see [“About Transport and Fault Information Returned by a Web Service Connector”](#) on page 139.

messageAddressingInfo **Document** Conditional. Message addressing headers from response and request messages.

Note:

The *messageAddressingInfo* parameter is not available in Integration Server versions prior to 9.0.

The *messageAddressingInfo* parameter contains the following keys:

Key	Value
<i>requestMessageAddressingProperties</i>	Document Conditional. Message addressing header fields from the request message. The value of this parameter can be the values you provide while

executing the web service connector, the values specified in the associated endpoint alias, or the values that Integration Server generates (for example, *messageID* and *action*).

Key	Description
<i>messageID</i>	String Conditional. Unique identifier of the SOAP message.
<i>action</i>	String Conditional. WS-Addressing action specified in the message addressing property of the SOAP message.
<i>to</i>	<p>Document Conditional. The endpoint reference that specifies the address of the intended receiver of the SOAP message. The <i>to</i> endpoint reference includes:</p> <ul style="list-style-type: none"> ■ <i>attributes</i>, which includes namespaceName, localname, and their values. ■ <i>address</i> and its attributes and values. ■ <i>referenceParameters</i> ■ <i>metadata</i> and its attributes and elements. ■ <i>extensibleElements</i>, which are any other elements usually provided for future extensions.
<i>from</i>	<p>Document Conditional. The endpoint reference that specifies the source of the SOAP message. The <i>from</i> endpoint reference includes:</p> <ul style="list-style-type: none"> ■ <i>attributes</i>, which includes namespaceName, localname, and their values. ■ <i>address</i> and its attributes and values. ■ <i>referenceParameters</i>

	<ul style="list-style-type: none">■ <i>metadata</i> and its attributes and elements.■ <i>extensibleElements</i>, which are any other elements usually provided for future extensions.
<i>replyTo</i>	<p>Document Conditional. The endpoint reference that specifies the destination address of the response (reply) message. The <i>replyTo</i> endpoint reference includes:</p> <ul style="list-style-type: none">■ <i>attributes</i>, which includes namespaceName, localname, and their values.■ <i>address</i> and its attributes and values.■ <i>referenceParameters</i>■ <i>metadata</i> and its attributes and elements.■ <i>extensibleElements</i>, which are any other elements usually provided for future extensions.
<i>faultTo</i>	<p>Document Conditional. The endpoint reference that specifies the address to which the SOAP fault messages are routed. The <i>faultTo</i> endpoint reference includes:</p> <ul style="list-style-type: none">■ <i>attributes</i>, which includes namespaceName, localname, and their values.■ <i>address</i> and its attributes and values.■ <i>referenceParameters</i>■ <i>metadata</i> and its attributes and elements.■ <i>extensibleElements</i>, which are any other elements usually provided for future extensions.

<i>response Message Addressing Properties</i>	Document Conditional. Header fields from the response.
Key	Description
<i>messageID</i>	String Conditional. Unique identifier of the SOAP message.
<i>relatesTo</i>	Document List Conditional. Contains the relation ship information to another SOAP message.
Key	Description
<i>value</i>	String Message ID of the related message.
<i>relationship Type</i>	String Conditional. The relationship type.
<i>action</i>	String Conditional. WS-Addressing action specified in the message addressing property of the SOAP message.
<i>to</i>	<p>Document Conditional. The endpoint reference that specifies the address of the intended receiver of the SOAP message. The <i>to</i> endpoint reference includes:</p> <ul style="list-style-type: none"> ■ <i>attributes</i>, which includes namespaceName, localname, and their values. ■ <i>address</i> and its attributes and values. ■ <i>referenceParameters</i> ■ <i>metadata</i> and its attributes and elements. ■ <i>extensibleElements</i>, which are any other elements usually provided for future extensions.
<i>from</i>	Document Conditional. The endpoint reference that specifies the

source of the SOAP message. The *from* endpoint reference includes:

- *attributes*, which includes namespaceName, localname, and their values.
- *address* and its attributes and values.
- *referenceParameters*
- *metadata* and its attributes and elements.
- *extensibleElements*, which are any other elements usually provided for future extensions.

replyTo

Document Conditional. The endpoint reference that specifies the destination address of the response (reply) message. The *replyTo* endpoint reference includes:

- *attributes*, which includes namespaceName, localname, and their values.
- *address* and its attributes and values.
- *referenceParameters*
- *metadata* and its attributes and elements.
- *extensibleElements*, which are any other elements usually provided for future extensions.

faultTo

Document Conditional. The endpoint reference that specifies the address to which the SOAP fault messages are routed. The *faultTo* endpoint reference includes:

- *attributes*, which includes namespaceName, localname, and their values.

- *address* and its attributes and values.
- *referenceParameters*
- *metadata* and its attributes and elements.
- *extensibleElements*, which are any other elements usually provided for future extensions.

reliableMessagingInfo

Document Conditional. Reliable messaging headers from response and request messages.

Note:

The *reliableMessagingInfo* parameter is not available in Integration Server versions prior to 9.0.

The *reliableMessagingInfo* parameter contains the following keys:

Key	Value
<i>responseReliableMessagingProperties</i>	Document Header fields from the response.

Key	Description
<i>serverSequenceId</i>	String Unique identifier of the SOAP message sequence.

Note:

The *serverSequenceId* is returned as the output parameter of the `pub.soap.wsm:createSequence` service. The *serverSequenceId* is used as the input parameter of `pub.soap.wsm:closeSequence`, `pub.soap.wsm:sendAcknowledgementRequest`, `pub.soap.wsm:terminateSequence`, and `pub.soap.wsm:waitUntilSequenceCompleted` services. For more information about these services, see *webMethods Integration Server Built-In Services Reference*.

fault

Document Conditional. The retrieved fault block. The web service connector returns *fault* when:

- A SOAP fault occurs, returning the SOAP fault information in *fault*.
- Any exception occurs while executing the web service connector, returning information about the exception in *fault*.

<i>code</i>	Document Contains the fault code and possible subcodes. <table><tr><th><u>Key</u></th><th>Descriptions</th></tr><tr><td><i>namespaceName</i></td><td>String Conditional. For a SOAP fault, namespace name for the SOAP fault code. For an exception in the client side connector, <i>namespaceName</i> will not be returned.</td></tr><tr><td><i>localName</i></td><td>String For a SOAP fault, a code that identifies the fault. For an exception in the client side connector, <i>localName</i> is set to <code>Exception</code>.</td></tr><tr><td><i>subCodes</i></td><td>Document List Conditional. Subcodes that provide further detail. Each Document in the <i>subCodes</i> Document List contains:<ul style="list-style-type: none">■ <i>namespaceName</i> for the subcode■ <i>localName</i> that identifies the subcode</td></tr></table>	<u>Key</u>	Descriptions	<i>namespaceName</i>	String Conditional. For a SOAP fault, namespace name for the SOAP fault code. For an exception in the client side connector, <i>namespaceName</i> will not be returned.	<i>localName</i>	String For a SOAP fault, a code that identifies the fault. For an exception in the client side connector, <i>localName</i> is set to <code>Exception</code> .	<i>subCodes</i>	Document List Conditional. Subcodes that provide further detail. Each Document in the <i>subCodes</i> Document List contains: <ul style="list-style-type: none">■ <i>namespaceName</i> for the subcode■ <i>localName</i> that identifies the subcode
<u>Key</u>	Descriptions								
<i>namespaceName</i>	String Conditional. For a SOAP fault, namespace name for the SOAP fault code. For an exception in the client side connector, <i>namespaceName</i> will not be returned.								
<i>localName</i>	String For a SOAP fault, a code that identifies the fault. For an exception in the client side connector, <i>localName</i> is set to <code>Exception</code> .								
<i>subCodes</i>	Document List Conditional. Subcodes that provide further detail. Each Document in the <i>subCodes</i> Document List contains: <ul style="list-style-type: none">■ <i>namespaceName</i> for the subcode■ <i>localName</i> that identifies the subcode								
<i>reasons</i>	Document List Reasons for the fault. Each Document in the Document List contains a human readable explanation of the cause of the fault. <table><tr><th><u>Key</u></th><th>Descriptions</th></tr><tr><td><i>*body</i></td><td>String Text explaining the cause of the fault.</td></tr><tr><td><i>@lang</i></td><td>String Conditional. Language for the human readable description.</td></tr></table>	<u>Key</u>	Descriptions	<i>*body</i>	String Text explaining the cause of the fault.	<i>@lang</i>	String Conditional. Language for the human readable description.		
<u>Key</u>	Descriptions								
<i>*body</i>	String Text explaining the cause of the fault.								
<i>@lang</i>	String Conditional. Language for the human readable description.								
<i>node</i>	String URI to the SOAP node where the fault occurred.								
<i>role</i>	String Conditional. Role in which the node was operating at the point the fault occurred.								
<i>detail</i>	Document Conditional. Application-specific details about the fault.								

Output Parameters for a web service descriptor created on Integration Server prior to 8.2

SOAP-FAULT **Document** Conditional. Document containing the SOAP fault returned by the web service. A SOAP fault is returned when an error occurs during invocation of the web service operation.

Key	Description
<i>soapProtocol</i>	String Specifies the SOAP protocol used to send messages to and receive messages from the web service host.

A value of...	Indicates that...
---------------	-------------------

SOAP 1.1 Protocol	Messages were sent using SOAP 1.1 protocol.
SOAP 1.2 Protocol	Messages were sent using the SOAP 1.2 protocol.

Fault_1_1 **Document** Conditional. Fault information. *Fault_1_1* and its child variables are populated only when a SOAP fault is returned to the web service connector and *soapProtocol* is SOAP 1.1 Protocol.

Key	Description
<i>faultcode</i>	String Conditional. A code that identifies the fault. This field corresponds to the <i>faultcode</i> element in SOAP 1.1.
<i>faultstring</i>	String Conditional. A human readable explanation of the fault. This field corresponds to the <i>faultstring</i> element in SOAP 1.1.
<i>faultactor</i>	String Conditional. Information about the cause of the fault. This field corresponds to the <i>faultactor</i> element in SOAP 1.1.
<i>detail</i>	Document Conditional. Application-specific details about the SOAP fault. This field corresponds to the <i>detail</i> element in SOAP 1.1.

Fault_1_2 **Document** Conditional. Fault information. *Fault_1_2* and its child variables are populated only when a SOAP fault is returned to the web service connector and *soapProtocol* is SOAP 1.2 Protocol.

Key	Description
-----	-------------

SOAP-ENV:Code **Document** Conditional. Contains the fault code.

- *SOAP-ENV:Value* **String** A code that identifies the fault. This corresponds to the `Code` element in SOAP 1.2.

SOAP-ENV:Reason **Document** Conditional. Document containing the reason for the SOAP fault. This corresponds to the `Reason` element in SOAP 1.2.

- *SOAP-Env:Text* **Document** Conditional. Document containing the human readable explanation of the cause of the fault.
 - *@XML:lang* **String** Conditional. Specifies the language for the human readable description. This field corresponds to the `xml:lang` attribute in SOAP 1.2.
 - **body* **String** Conditional. Text explaining the cause of the fault. This field corresponds to the content of the `Text` element in SOAP 1.2.

SOAP-ENV:Node **String** Conditional. URI to the SOAP node where the fault occurred. This field corresponds to the `Node` element in SOAP 1.2.

SOAP-ENV:Role **String** Conditional. Role in which the node was operating at the point the fault occurred. This field corresponds to the `Role` element in SOAP 1.2.

SOAP-ENV:Detail **Document** Conditional. Application-specific details about the SOAP fault. This field corresponds to the `Detail` element in SOAP 1.2.

How a Web Service Connector Represents the Input and Output Messages

How a web service connector represents the contents of the input and output message in the signature depends on the style/use of the binder for the web service connector.

- For a web service connector that uses a style/use of Document/Literal:

- The input signature contains an optional document reference to the IS document type created to represent the operation input message. At run time, if you do not specify any input for the document reference variable or any of its child variables, Integration Server sends an empty SOAP body in the SOAP message.
- The output signature contains a document reference to the IS document type created to represent the operation output message. This document reference is conditional and is only returned by the web service connector if the web service operation executes successfully. If returned at run time, this document reference contains the response from a successful invocation of a web service operation. If a SOAP fault occurred during execution of the web service operation, this document reference is not populated.
- For a web service connector that uses a style/use of RPC/Encoded or RPC/Literal:
 - The input signature contains variables that represent the top-level elements in the operation input message. All of these variables are optional. At run-time, if you do not specify any input for the variable (or variables) that represent the input message, Integration Server sends an empty SOAP body in the SOAP message.
 - The output signature contains variables that represent the top-level elements in the operation output message. All of these variables are conditional and are only returned by the web service connector if the web service operation executes successfully. If returned, these variables contain the response from a successful invocation of a web service operation.

About Transport and Fault Information Returned by a Web Service Connector

The transport information, such as headers, status codes, and status messages, returned by a web service connector created on Integration Server 8.2 or later varies depending on the following:

- The transport used to send and receive the SOAP message
- The success or failure of the web service connector
- The point at which failure occurs
- The message exchange pattern (MEP) for the operation

The preceding conditions also determine whether the *fault* document returned by the web service connector contains a SOAP fault or an exception. The fault document contains a SOAP fault when a SOAP fault occurs. The *fault* document contains an exception when an exception occurs while executing the web service connector.

The following table identifies the basic success and failure scenarios for web service connector execution and the transport information that would be returned in each scenario. The table also indicates whether the scenario results in a SOAP fault or an exception being returned in the *fault* document.

Note:

JMS status codes as well as the status code 900 are specific to Integration Server.

Use Case

Web service connector execution fails before sending the SOAP request.

Possible causes for this include improper message construction that results from validation failure for transport header information, consumer request handler failure, or WS-Security failure.

Parameter	Description
<i>status</i>	900
<i>statusMessage</i>	Error occurred while preparing SOAP request
<i>requestHeaders</i> returned?	Yes, if the web service connector created the SOAP request successfully but execution failed before sending the request.
<i>responseHeaders</i> returned?	No
<i>fault</i> contains fault or exception?	Exception

Use Case

Web service connector execution fails while sending the SOAP request.

Possible causes include an inaccessible HTTP server or JMS provider, an error from the HTTP server, or an exception thrown by the JMS provider.

Parameter	Description
<i>status</i>	For HTTP, the status code will be the value returned by the HTTP server. For JMS, the status code will be: 400
<i>statusMessage</i>	For HTTP, the status message will be the message returned by the HTTP server. For JMS, the status message will be: Error occurred while sending request to JMS provider
<i>requestHeaders</i> returned?	Yes
<i>responseHeaders</i> returned?	For HTTP, <i>responseHeaders</i> may be returned. For example, when the provider returns a status code in the 300 range or 400 range, it is possible that the provider populated <i>responseHeaders</i> .
<i>fault</i> contains fault or exception?	Exception If web service connector execution fails while using the JMS protocol to send the SOAP request, the <i>fault</i> document

contains the *lastError/\$errorType* field. This field indicates whether the exception was caused by a transient error or a *ServiceException*. *\$errorType* will be one of the following:

- `com.wm.app.b2b.server.ServiceException`
- `com.wm.app.b2b.server.ISRuntimeException`

Use Case

Web service connector execution fails while sending the SOAP request because a timeout occurs.

Possible causes include failure to receive a response from the HTTP server within the timeout period or failure to receive a JMS response message within the timeout period.

Parameter	Description
<i>status</i>	408
<i>statusMessage</i>	Timeout
<i>requestHeaders</i> returned?	Yes
<i>responseHeaders</i> returned?	No
<i>fault</i> contains fault or exception?	Exception

Use Case

Web service connector executes successfully.

Parameter	Description
<i>status</i>	<p>For HTTP, the status code will be the value returned by the HTTP server. The status code will typically be in the 200 range.</p> <p>For JMS and an In-Out or Robust In-Only operation, the status code will be: 200</p> <p>For JMS and an In-Only operation, the status code will be: 202</p>
<i>statusMessage</i>	<p>For HTTP, the status message will be the message returned by the HTTP server.</p> <p>For JMS, the status message will be: OK</p>
<i>requestHeaders</i> returned?	Yes

<i>responseHeaders</i> returned?	Dependent on the MEP of the operation. <ul style="list-style-type: none">■ For In-Only and Robust In-Only operation, <i>responseHeaders</i> is not returned.■ For a synchronous In-Out operation, <i>responseHeaders</i> is returned.
<i>fault</i> contains fault or exception?	Not applicable.

Use Case

Web service connector executes successfully but the JMS provider is not available, causing Integration Server to write the JMS message to the client side queue.

Note:

This use case applies to JMS only. It occurs only when the client side queue is enabled for the JMS binder.

Parameter	Description
<i>status</i>	300
<i>statusMessage</i>	Message written to the client side queue
<i>requestHeaders</i> returned?	Yes
<i>responseHeaders</i> returned?	No
<i>fault</i> contains fault or exception?	Not returned.

Use Case

Web service connector sends a SOAP request successfully but receives a SOAP fault from the web service provider.

SOAP faults can result from:

- Absence of required data from the client.
- Failed processing by the web service provider.
- Failed request handlers on the provider.
- WS-Security processing failures on the provider.

Parameter	Description
-----------	-------------

<i>status</i>	For HTTP, the status code will be the value returned by the HTTP server. The status code will typically be in the 500 range. For JMS, the status code will be: 500
<i>statusMessage</i>	For HTTP, the status message will be the message returned by the HTTP server. For JMS, the status message will be: SOAP Fault
<i>requestHeaders</i> returned?	Yes
<i>responseHeaders</i> returned?	Yes
<i>fault</i> contains fault or exception?	SOAP Fault

Use Case

Web service connector execution fails while processing the SOAP response.

Possible causes include output signature validation failure, response handler failure on the consumer, or incorrect conversion of the SOAP response to IData.

Parameter	Description
<i>status</i>	900
<i>statusMessage</i>	Error occurred while processing SOAP response
<i>requestHeaders</i> returned?	Yes
<i>responseHeaders</i> returned?	Yes
<i>fault</i> contains fault or exception?	Exception

Note:

Transport information is returned in the *transportInfo* output parameter. For more information about this parameter, see [“Signature for a Web Service Connector” on page 118](#).

Note:

If you want to drop the extra response document that the web service connector service adds to the output in the pipeline, you must add a MAP step to do so at the end of the web service connector service.

How a SOAP Fault is Mapped to the Generic Fault Output Structure

When the web service descriptor is created on Integration Server 8.2, the output signature of the web service connector uses a generic fault structure. In case of SOAP faults, to populate the generic fault structure, Integration Server maps data from a source SOAP 1.1 or SOAP 1.2 fault into the generic fault output.

The following table lists the fields in the generic fault structure, that is the fields in the *fault* output parameter, and how Integration Server sets the field values based on whether a SOAP 1.2 or SOAP 1.1 fault message is returned.

Output parameter in the generic fault structure	Value of the parameter for a SOAP 1.2 fault	Value of the parameter for a SOAP 1.1 fault
<code>fault</code> <code>code</code>		
<code>namespaceName</code>	Namespace name of the Code element's value	Namespace name of the faultcode element's value
<code>localName</code>	Local part of the Code element's value	Local part of the faultcode element's value
<code>subcodes</code>		
<code>namespaceName</code>	Namespace name of the Subcode element's value	null*
<code>localName</code>	Local name of the Subcode element's value	null*
<code>reasons</code>	Reason elements	
<code>*body</code>	Text child element of a Reason element	faultstring element's value
<code>@lang</code>	Accept-Language header in the <code>xml:lang</code> attribute of the Text child element	null*
<code>node</code>	Node element's value	null*
<code>role</code>	Role element's value	faultactor element's value
<code>detail</code>	Detail element's value	detail element's value

*Some of the fields in the generic fault structure apply only to a SOAP 1.2 fault message because there is no corresponding field in a SOAP 1.1 fault message. As a result, when Integration Server populates the *fault* output variable from a SOAP 1.1 fault message, the fields that do not have a corresponding value will be null.

If for some reason you want to map data from the generic fault structure (i.e., *fault* output parameter) into the *SOAP_1_2* document variable that is used when working with a web service descriptor created using an Integration Server version prior to 8.2, be aware that you might not be able to map all the data. Keep the following in mind when you are mapping data from the generic fault structure to the *SOAP_1_2* document variable:

- The *subcodes* element, which is not available in the *SOAP_1_2* document and is recursive in SOAP 1.2 fault structure, is represented as an array in the generic fault structure.
- The *reason* element, which is a simple document in the *SOAP_1_2* document, is represented as an array in the generic fault structure.

Setting Transport Headers for HTTP/S

When creating a service that executes a web service connector, you can pass transport header information directly into the web service connector by passing name/value pairs in to the *transportHeaders* input parameter. When creating the SOAP request, Integration Server adds a transport header for each name/value pair.

Keep the following information in mind when setting *transportHeaders* for an HTTP/S request:

- Specify a key in *transportHeaders* for each header field that you want to set, where the key's name represents the name of the header field and the key's value represents the value of that header field.
- The names and values supplied to *transportHeaders* must be of type String. If a transport header has a name or value that is not of type String, the header will not be included in the message.
- For any header name/value pair supplied in *transportHeaders* for an HTTP/S request, Integration Server simply passes through the supplied headers and does not perform any validation for the headers beyond verifying that the name and value are of type String.
- If you do not set *transportHeaders* or do not specify the following header fields in *transportHeaders*, Integration Server adds and specifies values for the following standard header fields:
 - Accept
 - Authorization
 - Connection
 - Content-Type
 - Host
 - SOAPAction (Added when *soapProtocol* is SOAP 1.1 only)
 - User-Agent

Note:

Pass in the preceding headers to *transportHeaders* only if you are an experienced web service developer. Incorrect header values can result in failure of the request.

- If you specify `Authorization` in *transportHeaders*, the values specified for the *auth/transport* document and its children will not be used in the `Authorization` header.
- If you specify `Content-Type` in *transportHeaders* and the SOAP Protocol is SOAP 1.2, Integration Server ignores the value of `soapAction` obtained from the WSDL used to create the web service connector.
- If you specify the `SOAPAction` header in *transportHeaders* and the SOAP Protocol is SOAP 1.1, Integration Server ignores the value of `SOAPAction` obtained from the WSDL used to create the web service connector.
- If you specify the `SOAPAction` header but do not set a value for it and the web service descriptor does not run in pre-8.2 compatibility mode (the **Pre-8.2 compatibility mode** property is set to false), Integration Server ignores the `SOAPAction` header.
- If MTOM processing converts any portion of the SOAP request to an MTOM/XOP attachment, it will overwrite the `Content-Type` value supplied to the *transportHeaders* input.
- Integration Server sets the value of `Content-Length` automatically and overwrites any value passed in to *transportHeaders*.
- Integration Server automatically adds the `Cookie` header to the HTTP header and supplies any cookies established between Integration Server and the HTTP server with which it is interacting. If you supply the `Cookie` header to *transportHeaders*, Integration Server prepends the values you supply to the already established `Cookie` header value.
- The following headers are considered to be standard and require the specified capitalization: `Accept`, `Authorization`, `Connection`, `Content-Type`, `Cookie`, `Host`, `SOAPAction`, `User-Agent`.

Important:

Using capitalization other than that which is specified results in undefined behavior.

Important:

Supplying duplicate entries for any standard header results in undefined behavior.

Setting Transport Headers for JMS

When a web service connector sends a SOAP message over JMS, you can pass in name/value pairs to the *transportHeaders* input parameter of the web service connector. Integration Server uses these name/value pairs to create JMS message headers and properties. For some headers, values that you specify in *transportHeaders* might override a corresponding value in the source WSDL and the web service endpoint alias assigned to the JMS binder.

By passing in name/value pairs into *transportHeaders*, you can specify the following in the JMS message sent by the web service connector:

- JMS message header fields
- Standard message properties as defined by JMS
- Application-specific properties

- Provider-specific properties
- Run-time properties used by Integration Server

Keep the following information in mind when setting name/value pairs for *transportHeaders* for JMS.

- Specify a key in *transportHeaders* for each header field that you want to set, where the key's name represents the name of the header field and the key's value represents the value of that header field.
- The names and values supplied to *transportHeaders* must be of type String. If a transport header has a name or value that is not of type String, the header will not be included in the message.
- You can specify the following JMS message header fields in *transportHeaders*:
 - JMSCorrelationID
 - JMSType

Note:

The JMSCorrelationID and JMSType names are case-sensitive.

- You can specify the following JMS-defined properties in *transportHeaderes*:
 - JMSXGroupID
 - JMSXGroupSeq

If the value of JMSXGroupSeq is not an integer, Integration Server ignores the name/value pair and does not place it in the message header.

Note:

The JMSXGroupID and JMSXGroupSeq names are case-sensitive.

- The "JMSX" prefix is reserved for JMS-defined properties. If a header whose name starts with "JMSX" is passed into *transportHeaders* and it is not named JMSXGroupID or JMSXGroupSeq, Integration Server generates a fault and returns it to the web service connector.
- You can set any provider-specific property whose name starts with "JMS_" in *transportHeaders*. Integration Server maps a supplied name/value pair whose name starts with "JMS_" directly to a JMS message property. Because the JMS standard reserves the prefix "JMS_<vendor_name>" for provider-specific properties, Integration Server does not validate the name or value of this content.

Note:

The JMS provider determines which provider-specific properties to accept and include in the JMS message properties. For more information about provider-specific message properties and how the JMS provider handles them, review the JMS provider documentation.

- You can use *transportHeaders* to specify run-time properties that affect the values of the JMS message and JMS message headers. The following table identifies these properties and indicates the JMS message header fields affected by each property.

Property Name **Description**`.jms.async`

Indicates whether this is a synchronous or asynchronous request/reply.

This run-time property does not affect a JMS message header field

If the web service connector calls an In-Out operation and you want Integration Server to write the request message to the client side queue when the JMS provider is not available, you must pass `.jms.async= true` into *transportHeaders*. Additionally, to use the client side queue, the **Use CSQ** property must be set to true for the JMS binder. For information about how to retrieve the response to an asynchronous response, see [“Asynchronously Invoking an In-Out Operation” on page 114](#).

Value	Description
true	Indicates this is an asynchronous request/reply. Integration Server does not wait for a response message before executing the next step in the flow service. If <code>.jms.async</code> is true, Integration Server ignore the <i>timeout</i> value passed in to the web service connector.
false	Default. Indicates this is a synchronous request/reply. Integration Server waits for a response before executing the next step in the flow service.

`.jms.deliveryMode`

Specifies the message delivery mode for the message. Integration Server uses this value to set the `JMSDeliveryMode` header.

Specify one of the following values:

Value	Description
PERSISTENT	Indicates the request message is persistent.
2	Default. Indicates the request message is persistent.
NON_PERSISTENT	Indicates the request message is not persistent.

Property Name	Description
	<p>1 Indicates the request message is not persistent.</p> <p>Note: If the <code>jms.deliveryMode</code> is not one of the above values, Integration Server ignores the name/value pair and uses the default value of 2.</p>
<code>jms.messageType</code>	<p>Specifies the message type for the request message sent by the web service connector.</p> <p>Specify one of the following:</p> <ul style="list-style-type: none"> ■ <code>BytesMessage</code> ■ <code>TextMessage</code> <p>The message type specified by <code>jms.messageType</code> overwrites the default message type set by the <code>watt.server.soapjms.defaultMessageType</code> configuration parameter.</p> <p>Sending a <code>TextMessage</code> can be useful for debugging purposes because the resulting message will be a human-readable format.</p> <p>The message type of the request message determines the message type of the response message.</p> <p>Note: If the <code>jms.messageType</code> value is not <code>BytesMessage</code> or <code>TextMessage</code>, Integration Server ignores the name/value pair and uses the default value of <code>BytesMessage</code>.</p>
<code>jms.timeToLive</code>	<p>Length of time, in milliseconds, that the JMS provider retains the message. A value of 0 means that the message does not expire.</p> <p>The JMS provider uses this value to set the <code>JMSExpiration</code> header in the sent JMS message.</p> <p>Note: If the <code>jms.timeToLive</code> value is not a valid <code>Long</code>, Integration Server ignores the property and uses the default value of 0.</p>
<code>jms.priority</code>	<p>Specifies the message priority. The JMS standard defines priority levels from 0 to 9, with 0 as the lowest priority and 9 as the highest.</p> <p>Integration Server uses this value to set the <code>JMSPriority</code> header.</p>

Property Name	Description
---------------	-------------

Note:

If the `jms.priority` value is not a value between 0 and 9, Integration Server ignores the property and uses the default value of 4.

- The lowercase “jms.” prefix is reserved for run-time properties used by Integration Server. If a header starts with “jms.” and is not one of the properties defined by Integration Server, Integration Server ignores the property.

Passing Message-Level Security Information to a Web Service Connector

When using WS-Security to secure a web service, you can pass security information directly into a web service connector (WSC). At run-time, Integration Server uses the information to build the WS-Security header and the SOAP message request.

For more information about securing a web service, see [“Securing Web Services \(WS-Security\)” on page 237](#). For more information about how Integration Server obtains the security information it uses, see [“WS-Security Certificate and Key Requirements” on page 243](#).

➤ To pass security information into a web service connector

1. In Package Navigator view, open and lock the service that invokes the web service connector.
2. If the SOAP message request requires credentials for a UsernameToken, do the following in the pipeline for the web service connector:
 - a. Map or set the value of `auth/message/user` to the user name used to authenticate the consumer client on the web services host.
 - b. Map or set the value of `auth/message/pass` to the password used to authenticate the consumer client on the web services host.
3. If the SOAP message request needs to be signed, set the following fields in the web service connector:

In this field...**Specify**

<code>auth/message/serverCerts/keyStoreAlias</code>	Alias to the keystore that contains the private key used to sign outbound SOAP requests.
-----------------------------------------------------	------------------------------------------------------------------------------------------

<code>auth/message/serverCerts/keyAlias</code>	Alias to the private key used to sign and/or include X.509 authentication token for outbound SOAP messages and/or decrypt
------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------

In this field...**Specify**

inbound SOAP responses. The key must be in the keystore specified in *auth/message/serverCerts/keyStoreAlias*.

Note:

The method you use to fetch these credentials depends upon their location at your site. If they are stored in the file system, you can retrieve them using the `pub.file.getFile` service. If they are stored in a special repository or a DBMS, you may need a custom service for their retrieval.

4. If the SOAP message request requires encryption, set the following field:

In this field...**Specify**

auth/message/partnerCert

The path and file name of the provider's certificate, which contains its public key.

5. If the SOAP message response needs to be verified, do the following:

In this field...**Specify**

auth/message/partnerCert

The path and file name of the provider's certificate, which contains its public key.

6. If the SOAP message must be decrypted, do the following:

In this field...**Specify**

auth/message/serverCerts/keyStoreAlias

Alias to the keystore that contains the private key that the consumer will use to decrypt the SOAP response.

auth/message/serverCerts/keyAlias

Alias to the private key used to decrypt inbound SOAP responses. The key must be in the keystore specified in *auth/message/serverCerts/keyStoreAlias*.

5 Working with Response Services

■ About Response Services	154
■ Signature for a Response Service	155
■ Signature for a genericFault_Response Service	155

About Response Services

At the time of creating a consumer web service descriptor, Integration Server creates a *responseServices* folder along with the connectors and docTypes folders.

The *responseServices* folder contains a response service for each In-Out and Robust-In-Only MEP operation in the WSDL document from which the consumer web service descriptor is created and a *genericFault_Response* service. Integration Server creates the response services and *genericFault_Response* service only if the consumer web service descriptor:

- Is created on Integration Server version 9.0 or later.
- Has the **Pre-8.2 compatibility mode** property set to false.

The *responseServices* folder contains:

- **Response services.** Integration Server creates a response service for each In-Out and Robust-In-Only MEP operation contained in the WSDL document. Response services are flow services to which you can add custom logic to process asynchronous SOAP responses. Integration Server invokes these services for processing SOAP responses received for the associated consumer web service descriptor. That is, Integration Server invokes a response service when Integration Server receives a SOAP response with the endpoint URL pointing to a consumer web service descriptor and if this SOAP response contains a WS-Addressing action through which the response service can be resolved.

The name of a response service is the same as the corresponding web service connector suffixed with the term “_Response”. The input signature of each response service is the same as the output signature of the corresponding web service connector.

- **A *genericFault_Response* service.** The default response service that will be invoked when Integration Server cannot determine the specific response service for an asynchronous SOAP response or if there are errors while processing the response (for example, errors related to authorization, handler processing, or missing headers). Each *responseServices* folder contains one *genericFault_Response* service.

Integration Server will also invoke the *genericFault_Response* service if the addressing action on an asynchronous response is one of the following URLs:

- W3C Member Submission WS-Addressing standard version:
<http://schemas.xmlsoap.org/ws/2004/08/addressing/fault>
- W3C Final WS-Addressing standard version:
<http://www.w3.org/2005/08/addressing/fault>

Keep the following information in mind when working with response services:

- Response services are created only for In-Out and Robust In-Only MEP operations. For example, in case of a consumer web service descriptor with one In-Only MEP and one In-Out MEP operation, Integration Server creates one response service that corresponds to the In-Out MEP operation.

- You can use the response services to process asynchronous SOAP responses only if you have attached WS-Addressing policies to the web service descriptor. To know more about how to handle asynchronous SOAP responses, see [“Processing Responses Asynchronously” on page 329](#).
- You must refresh the web service descriptors or web service connectors to update the signature of the response services based on the changes that you make to the consumer web service descriptor or the WSDL from which this consumer web service descriptor is created.

For example, if you add a response or fault header to a consumer web service descriptor, you must refresh the connectors to update the response services with the response or fault headers.

Signature for a Response Service

Important:

The response service signature cannot be modified.

The input signature of a response service for an operation is the same as the output signature of the web service connector that corresponds to the same operation. The response service signature contains optional inputs that you can use to control the execution of logic.

For more information about the signature of a response service, see the output parameters of a web service connector that is specified in [“Signature for a Web Service Connector” on page 118](#).

Signature for a genericFault_Response Service

The genericFault_Response service is a service to handle any faults or errors that are not directed to a specific response service. If a SOAP fault returned by the web service provider contains an addressing action, Integration Server invokes the appropriate response service. Integration Server invokes the genericFault_Response service only if it cannot determine the specific response service for a SOAP response or if there are errors while processing the response. You must add custom logic to the genericFault_Response service for handling the faults.

The signature of genericFault_Response service is the same as the output signature of web service connector. In addition to these parameters, the genericFault_Response service also contains *messageContext* as an input parameter. You can use the *messageContext* parameter to access the response SOAP message using the `pub.soap.handler.getSOAPMessage` service.

For more information about the signature of the genericFault_Response service, see the output parameters of a web service connector that is specified in [“Signature for a Web Service Connector” on page 118](#).

6 About Handlers and Handler Services

■ What Are Handlers and Handler Services?	158
■ Setting Up a Handler	158
■ Registering a Handler	159
■ About Request Handler Services	159
■ About Response Handler Services	162
■ About Fault Handler Services	167

What Are Handlers and Handler Services?

In addition to the data contained in the SOAP body, a SOAP message might contain data in the SOAP headers. The best way to access the SOAP headers is to use handlers. A handler, sometimes called a header handler, provides access to the entire SOAP message.

Handlers can be used to perform various types of processing, including processing SOAP headers, adding SOAP headers, removing SOAP headers, passing data from the header to the endpoint service or vice versa. Provider and consumer web service descriptors can use handlers.

In Integration Server, a handler is a set of up to three handler services. The handler can contain one of each of the following handler services:

- Request handler service
- Response handler service
- Fault handler service

Integration Server executes each type of handler service at a different point in the web service invocation path.

Any IS service can be used as a handler service. However, handler services must use a specific service signature. Integration Server defines the service handler signature in the `pub.handler.soap:handlerSpec` specification. Integration Server also provides several services that you can use when creating handler services. These services are located in the `pub.soap.handler` folder in the `WmPublic` package.

When you register a handler, you name the handler, identify the services that function as the request, response or fault handler services, and indicate whether the handler is for use with provider web service descriptors or consumer web service descriptors.

You can assign multiple handlers to a web service descriptor. The collection of handlers assigned to a web service descriptor is called a handler chain. For a consumer web service descriptor, Integration Server executes the handler chain for output SOAP requests and inbound SOAP responses. For a provider web service descriptor, Integration Server executes the handler chain for inbound SOAP requests and outbound SOAP responses. The order of handlers in the handler chain may be important, depending on what processing the handlers are performing.

When executing the handler chain, Integration Server executes request handler services by working through the handler chain from top to bottom. However, Integration Server executes response handler services and fault handler services from bottom to top.

Setting Up a Handler

To create and implement a handler, you need to:

1. Build the services for handling a request, handling a response, and handling a fault. Use the `pub.soap.handler:handlerSpec` specification as the signature for a service that acts as a header handler.

2. Register the combination of those services as a header handler. For more information, see [“Registering a Handler” on page 159](#).
3. Assign the header handler to the web service descriptor. For more information, see *webMethods Service Development Help*.

Registering a Handler

Register the handler as either a consumer or provider using `pub.soap.handler:registerWmConsumer` or `pub.soap.handler:registerWmProvider`, respectively. During registration you specify:

- A name for the handler.
- The services to use for handling headers for a request, a response, and a fault.
- Optionally, the list of QNames on which the handler operates.

Specify QNames only if you want to associate with handler with one or more QNames. Registering QNames with a handler provides the following benefits:

- Integration Server can perform `mustUnderstand` checking for the header with the QName at run time. If a service receives a SOAP message in which a header requires `mustUnderstand` processing by the recipient, Integration Server uses the header QName to locate the handler that processes the header. Note that the handler must be part of the handler chain for the web service descriptor that contains the service.
- When adding headers to a web service descriptor, Designer populates the list of IS document types that can be used as headers in the web service descriptor with the IS document types whose QNames were registered with the handlers already added to the web service descriptor. If you add a IS document type as a header to a web service descriptor and the QName of that IS document type is not associated with a handler, Designer adds the header but display a warning stating that there is not an associated handler.
- When consuming WSDL to create a provider or consumer web service descriptor, Integration Server automatically adds a handler to the resulting web service descriptor if the WSDL contains a QName supported by the handler.

Note: Integration Server stores information about registered header handlers in memory. Integration Server does not persist registered header handler information across restarts. Consequently, you must register header handlers each time Integration Server starts. To accomplish this, create a service that registers a handler and make that service a start up service for the package that contains the services that act as handlers.

About Request Handler Services

A request handler service is used with a SOAP request. The processing point at which Integration Server invokes a request handler service depends on whether the web service descriptor is a consumer or a provider.

Web Service Descriptor	Behavior
Consumer	Integration Server invokes the request handler service for each handler in the handler chain after creating the initial outbound SOAP request but before sending the SOAP request to the web service provider.
Provider	Integration Server invokes the request handler service for each handler in the handler chain immediately after the provider receives the SOAP request. After successfully executing the request handler service for each handler in the handler chain, Integration Server invokes the endpoint service.

A request handler service is sometimes referred to as a `handleRequest` service.

Note: Integration Server executes request handler services regardless of the message exchange pattern (MEP) for the web service connector or operation.

Request Handler Services and Status Codes

Each request handler service must return a status code. The status code is an integer from 0–3, indicates the success or failure of the request handler service, and indicates how Integration Server will proceed.

- For a consumer web service descriptor, the request handler service status determines whether or not Integration Server sends the SOAP request to the web service provider. Integration Server sends the SOAP request only after all request handler services in the handler chain return a status code of 0. If a request handler service returns a status code of 1, 2, or 3, handler chain processing stops and Integration Server does not send the SOAP request to the provider.
- For a provider web service descriptor, the request handler service status determines whether or not Integration Server invokes the endpoint service. Integration Server invokes the endpoint service only after all request handler services in the handler chain return a status code of 0. If a request handler service returns a status code of 1, 2, or 3, handler chain processing stops and Integration Server does not invoke the endpoint service.

The following table describes the meaning of each status code for a request handler service and the action Integration Server takes based on the status code.

Status Code	Description
0	Indicates that the request handler service executed successfully. Integration Server executes the next request handler service in the handler chain.
Consumer Side Behavior	
	If this is the last handler in the handler chain, Integration Server sends the SOAP request to the web service provider.
Provider Side Behavior	

Status Code	Description
-------------	-------------

- | | |
|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | If this is the last handler in the handler chain Integration Server invokes the endpoint service, passing in data from the SOAP request. |
| 1 | Indicates that the handler service ended in failure. Integration Server suspends execution of the handler chain. Integration Server then invokes the response handler service for each handler that has already executed in the handler chain. Integration Server starts with the current handler and continues in reverse order until it invokes all the response handlers for the executed request handlers in the handler chain. |

Consumer Side Behavior

If the request handler failed while processing a SOAP request for a consumer web service descriptor, Integration Server does not send the SOAP request to the provider.

Integration Server places the following exception in the *fault* document returned by the web service connector:

[ISS.0088.9431] Handler processing failed on the consumer:
requestHandlerService

Provider Side Behavior

If the request handler failed while processing a SOAP request for a provider web service descriptor, Integration Server does not invoke the endpoint service.

Integration Server constructs a SOAP fault and places it in the message context which can be accessed by the response handler services. The SOAP fault contains the following information:

[ISS.0088.9431] Handler processing failed on the provider:
requestHandlerService

- | | |
|---|--------------------------------------------------------------------------------------------------------------------------|
| 2 | Indicates that the request handler service ended in failure. Integration Server suspends execution of the handler chain. |
|---|--------------------------------------------------------------------------------------------------------------------------|

Consumer Side Behavior

If the request handler failed while processing a SOAP request for a consumer web service descriptor, Integration Server places the following exception in the *fault* document returned by the web service connector:

[ISS.0088.9431] Handler processing failed on the consumer: *faultMessage*

Where *faultMessage* is the content of the *faultMessage* parameter returned by the request handler service.

Integration Server does not send the SOAP request to the provider.

Status Code Description**Provider Side Behavior**

If the request handler failed while processing a SOAP request for a provider web service descriptor, Integration Server constructs a SOAP fault that contains the following information:

[ISS.0088.9431] Handler processing failed on the provider: *faultMessage*

Where *faultMessage* is the content of the *faultMessage* parameter returned by the request handler service.

Integration Server does not invoke the endpoint service.

- 3 Indicates that the request handler service ended in failure. Integration Server suspends execution of the handler chain. Integration Server then invokes the fault handler service for each handler that has already executed in the handler chain. Integration Server starts with the current handler and continues in reverse order until it invokes all the fault handlers for the request handlers executed in the handler chain.

Note:

One of the fault handler services in the handler chain should contain logic that constructs a SOAP fault message. If it does not, the SOAP fault message will be empty. This service also needs to replace the SOAP message in the current message context with the SOAP fault message.

Consumer Side Behavior

If the request handler failed while processing a SOAP request for a consumer web service descriptor, Integration Server does not send the SOAP request to the provider.

Provider Side Behavior

If the request handler failed while processing a SOAP request for a provider web service descriptor, Integration Server does not invoke the endpoint service.

About Response Handler Services

A response handler service is used with a SOAP response. The processing point at which Integration Server invokes a response handler service depends on whether the web service descriptor is a consumer or a provider and the message exchange pattern (MEP) for the web service connector or operation.

Web Service Descriptor	Pre-8.2 Compatibility Mode	MEP	Behavior
Consumer	False	In-Out	<p>For a web service connector in a consumer web service descriptor, Integration Server invokes the response handler service for each handler in the handler chain after it receives the initial SOAP response from the web service provider.</p> <p>Integration Server also invokes a response handler service for a consumer web service descriptor if a request handler service failed and returned a status code of 1. In this situation, Integration Server does not invoke the response handler service for every handler in the handler chain. Instead, Integration Server invokes the response handler service for each handler that has already executed in the handler chain. Integration Server starts with the current handler and continues in reverse order until it invokes all the response handlers for the executed request handlers in the handler chain.</p>
Consumer	False	In-Only	Integration Server does not execute the response handler services assigned to the web service descriptor.
Consumer	False	Robust In-Only	<p>Integration Server executes response handlers if an exception occurs while executing the operation or if a request handler service failed and returned a status code of 1.</p> <div data-bbox="786 1323 1364 1596"> <p>Note: For a consumer, Integration Server provides partial support for Robust In-Only for SOAP over JMS. For more information, see “Consumer Support for Robust In-Only Operations with SOAP/JMS Bindings” on page 115.</p> </div>
Consumer	True	All	<p>For a web service connector in a consumer web service descriptor, Integration Server invokes the response handler service for each handler in the handler chain after it receives the initial SOAP response from the web service provider.</p> <p>Integration Server also invokes a response handler service for a consumer web service</p>

Web Service Descriptor	Pre-8.2 Compatibility Mode	MEP	Behavior
			<p>descriptor if a request handler service failed and returned a status code of 1. In this situation, Integration Server does not invoke the response handler service for every handler in the handler chain. Instead, Integration Server invokes the response handler service for each handler that has already executed in the handler chain. Integration Server starts with the current handler and continues in reverse order until it invokes all the response handlers for the executed request handlers in the handler chain.</p> <div>Note: When Pre-8.2 compatibility mode is true, Integration Server treats all web service connectors as if they have an In-Out MEP.</div>
Provider	False	In-Out	<p>Integration Server invokes the response handler service after the IS service exposed as an operation executes successfully and creates a SOAP response. Integration Server invokes each response handler service in the handler chain for the provider web service descriptor and then sends the updated SOAP response to the web service client.</p> <p>Integration Server also invokes a response handler service for a provider web service descriptor if a request handler service failed and returned a status code of 1. In this situation, Integration Server does not invoke the response handler service for every handler in the handler chain. Instead, Integration Server invokes the response handler service for each handler that has already executed in the handler chain. Integration Server starts with the current handler and continues in reverse order until it invokes all the response handlers for the executed request handlers in the handler chain.</p>
Provider	False	In-Only	<p>Integration Server does not execute the response handler services assigned to the web service descriptor.</p>

Web Service Descriptor	Pre-8.2 Compatibility Mode	MEP	Behavior
Provider	False	Robust In-Only	Integration Server executes response handlers if an exception occurs while executing the operation or if a request handler service failed and returned a status code of 1.
Provider	True	All	<p>Integration Server invokes the response handler service after the IS service exposed as an operation executes successfully and creates a SOAP response. Integration Server invokes each response handler service in the handler chain for the provider web service descriptor and then sends the updated SOAP response to the web service client.</p> <p>Integration Server also invokes a response handler service for a provider web service descriptor if a request handler service failed and returned a status code of 1. In this situation, Integration Server does not invoke the response handler service for every handler in the handler chain. Instead, Integration Server invokes the response handler service for each handler that has already executed in the handler chain. Integration Server starts with the current handler and continues in reverse order until it invokes all the response handlers for the executed request handlers in the handler chain.</p>

Note:

When **Pre-8.2 compatibility mode** is true, Integration Server treats all operations as if they have an In-Out MEP.

A response handler service is sometimes referred to as a `handleResponse` service.

Response Handler Services and Status Codes

Each response handler service must return a status code. The status code is an integer from 0–3, indicates the success or failure of the response handler service, and indicates how Integration Server will proceed. The following table describes the meaning of each status code for a response handler service and the action Integration Server takes based on the status code.

Status Code Description

- 0 Indicates that the response handler service executed successfully. Integration Server executes the next response handler service in the handler chain.

Consumer Side Behavior

If this is the last handler in the handler chain, Integration Server returns the SOAP response to the web service connector.

Provider Side Behavior

If this is the last handler in the handler chain, Integration Server sends the SOAP response to the web service consumer.

Note:Integration Server executes response handler services in the reverse order in which it executes request handler services.

- 1 Indicates that the response handler service ended in failure. Integration Server suspends execution of the handler chain.

Consumer Side Behavior

If the response handler executed for a consumer web service descriptor, Integration Server places the following exception in the *fault* document returned by the web service connector:

[ISS.0088.9431] Handler processing failed on the consumer.
responseHandlerService

Provider Side Behavior

If the response handler executed for a provider web service descriptor, Integration Server generates a SOAP fault that contains the following information:

[ISS.0088.9431] Handler processing failed on the provider.
responseHandlerService

Integration Server returns a SOAP response containing the SOAP fault to the web service consumer.

- 2 or 3 Indicates that the response handler service ended in failure. Integration Server suspends execution of the handler chain.

Consumer Side Behavior

If the response handler executed for a consumer web service descriptor, Integration Server places the following exception in the *fault* document returned by the web service connector:

[ISS.0088.9431] Handler processing failed on the consumer. *faultMessage*

Status Code Description

Where *faultMessage* is the content of the *faultMessage* parameter returned by the response handler service.

Provider Side Behavior

If the response handler executed for a provider web service descriptor, Integration Server generates a SOAP fault that contains the following information:

[ISS.0088.9431] Handler processing failed on the provider: *faultMessage*

Where *faultMessage* is the content of the *faultMessage* parameter returned by the response handler service.

Integration Server returns a SOAP response containing the SOAP fault to the web service consumer.

Note:

For a response handler service, status codes 2 and 3 have identical behavior.

About Fault Handler Services

A fault handler service is used when a request handler service failed and returned a status code of 3. Whether or not Integration Server invokes a fault handler service depends on the message exchange pattern (MEP) of the operation or web service connector and the **Pre-8.2 compatibility mode** value for the web service descriptor.

Pre-8.2 Compatibility Mode	MEP	Behavior
False	In-Out	Integration Server invokes a fault handler service if a request handler service failed and returned a status code of 3. In this situation, Integration Server does not invoke the fault handler service for every handler in the handler chain. Instead, Integration Server invokes the fault handler service for each handler that has already executed in the handler chain. Integration Server starts with the current handler and continues in reverse order until it invokes all the fault handlers for the executed request handlers in the handler chain.
	In-Only	Integration Server does not execute the fault handler services assigned to the web service descriptor.

Pre-8.2 Compatibility Mode	MEP	Behavior
	Robust-In-Only	Integration Server only executes the fault handler services if a request handler service fails and returns a status code of 3. Note: For a consumer, Integration Server provides partial support for Robust In-Only for SOAP over JMS. For more information, see “Consumer Support for Robust In-Only Operations with SOAP/JMS Bindings” on page 115 .
True	All	Integration Server invokes a fault handler service if a request handler service failed and returned a status code of 3. In this situation, Integration Server does not invoke the fault handler service for every handler in the handler chain. Instead, Integration Server invokes the fault handler service for each handler that has already executed in the handler chain. Integration Server starts with the current handler and continues in reverse order until it invokes all the fault handlers for the executed request handlers in the handler chain.

A fault handler service is sometimes referred to as a `handleFault` service.

Fault Handler Services and Status Codes

Each fault handler service must return a status code. The status code is an integer from 0–3, indicates the success or failure of the fault handler service, and indicates how Integration Server will proceed. The following table describes the meaning of each status code for a fault handler service and the actions Integration Server takes based on the status code.

Status Code	Description
-------------	-------------

0	Indicates that the fault handler service executed successfully. Integration Server executes the next handler in the handler chain.
---	------------------------------------------------------------------------------------------------------------------------------------

Consumer Side Behavior

If this is the last handler in the handler chain, Integration Server returns the SOAP fault to the web service connector.

Provider Side Behavior

Status Code	Description
	<p>If this is the last handler in the handler chain, Integration Server sends a SOAP response containing the SOAP fault to the web service consumer.</p> <p>Note: Integration Server executes fault handler services in the reverse order in which it executes request handler services.</p>
1	<p>Indicates that the fault handler service ended in failure. Integration Server suspends execution of the handler chain.</p> <p>Consumer Side Behavior</p> <p>If the fault handler executed for a consumer web service descriptor, Integration Server places the following exception in the <i>fault</i> document returned by the web service connector:</p> <p>[ISS.0088.9431] Handler processing failed on the consumer. <i>faultHandlerService</i></p> <p>Provider Side Behavior</p> <p>If the fault handler executed for a provider web service descriptor, Integration Server generates a SOAP fault that contains the following information:</p> <p>[ISS.0088.9431] Handler processing failed on the provider: <i>faultHandlerService</i></p> <p>Integration Server returns a SOAP response containing the SOAP fault to the web service consumer.</p>
2	<p>Indicates that the fault handler service ended in failure. Integration Server suspends execution of the handler chain.</p> <p>Consumer Side Behavior</p> <p>If the fault handler executed for a consumer web service descriptor, Integration Server places the following exception in the <i>fault</i> document returned by the web service connector:</p> <p>[ISS.0088.9431] Handler processing failed on the consumer. <i>faultMessage</i></p> <p>Where <i>faultMessage</i> is the content of the <i>faultMessage</i> parameter returned by the fault handler service.</p> <p>Provider Side Behavior</p> <p>If the fault handler executed for a provider web service descriptor, Integration Server generates a SOAP fault that contains the following information:</p> <p>[ISS.0088.9431] Handler processing failed on the provider: <i>faultMessage</i></p>

Status Code Description

- | | |
|---|------------------------------------------------------------------------------------------------------------------------|
| | Integration Server returns a SOAP response containing the SOAP fault to the web service consumer |
| 3 | Indicates that the fault handler service ended in failure. Integration Server suspends execution of the handler chain. |

Consumer Side Behavior

If the fault handler executed for a consumer web service descriptor, Integration Server places the fault reason returned by the fault handler in the *fault* document returned by the web service connector.

Provider Side Behavior

If the fault handler executed for a provider web service descriptor, Integration Server returns a SOAP response containing the SOAP fault to the web service consumer. The SOAP fault contains the fault reason returned by the fault handler.

7 About Outbound Callback Services

■ What Are Outbound Callback Services?	172
■ Usage of Outbound Callback Services	172
■ Invoking Outbound Callback Services	173

What Are Outbound Callback Services?

Integration Server provides you the flexibility to add custom processing logic to SOAP requests in case of consumer web service descriptors and to SOAP responses in case of provider web service descriptors. To do this, you use *outbound callback services*, which are user-specified IS services, in outbound SOAP messages.

Integration Server defines the outbound callback service signature in the `pub.soap.utils.callbackServiceSpec` specification. When you specify an IS service as an outbound callback service, Integration Server creates the message context and passes it to the outbound callback service. The message context of the outbound callback service contains the properties for the outbound SOAP message and provides access to the SOAP message. For more information about the `pub.soap.utils.callbackServiceSpec` specification, see the section *pub.soap.utils:callbackServiceSpec* in the *webMethods Integration Server Built-In Services Reference*.

You can use the various services that are located in the `pub.soap.handler` folder in the `WmPublic` package to manipulate the message within the IS service that is used as the outbound callback service.

To specify outbound callback services for outbound SOAP messages, you use **Outbound Callback Service** web service descriptor property. For more information about the property, see *webMethods Service Development Help*.

Usage of Outbound Callback Services

Keep the following points in mind when working with outbound callback services:

- You can use outbound callback services only for In-Out MEP or Robust In-Only MEP operations. Integration Server processes outbound callback services for different MEP operations as follows:
 - **In-Out MEP operations.** Integration Server invokes the outbound callback service for both consumer and provider web service descriptors.
 - **Robust In-Only MEP operations.** Integration Server invokes the outbound callback service for consumer web service descriptors.

If SOAP fault is returned, Integration Server invokes the outbound callback service for provider web service descriptors.

If no SOAP fault is returned, Integration Server does not invoke the outbound callback service for provider web service descriptors, but invokes the outbound callback service for consumer web service descriptors.
- **In-Only MEP operations.** Integration Server invokes the outbound callback service on the client side when sending a SOAP request message. Integration Server does not invoke the outbound callback service for provider web service descriptors.
- You can use outbound callback services with web service descriptors regardless of whether the **Pre-8.2 compatibility mode** property of the web service descriptors is set to `true` or `false`.

- Integration Server checks the execute ACL for an outbound callback service only if the service permissions specify that the **Enforce execute ACL** option is set to **Always**. Integration Server does not consider outbound callback services to be top-level services.
- You can use the `pub.soap.handler:getInitialSOAPRequest` service to retrieve the initial SOAP request message in the outbound callback service.

Invoking Outbound Callback Services

The following table describes the different phases at which Integration Server executes an outbound callback service.

Phase	Description
PRESECURITY	<p>Integration Server invokes the outbound callback service before the security processing phase, also known as the PRESECURITY phase.</p> <p>In case of consumer web service descriptors, Integration Server invokes the outbound callback service when the request handler services are executed. In case of provider web service descriptors, Integration Server invokes the outbound callback service when the response and fault handler services are executed.</p> <p>Integration Server executes the outbound callback service in the PRESECURITY phase only if a WS-Security policy or a security handler service is attached to the web service descriptor.</p>
TRANSPORT	<p>Integration Server invokes the outbound callback service at the transport sender phase, also known as the TRANSPORT phase.</p> <p>Even if the outbound callback service is executed in the PRESECURITY phase, Integration Server will execute the outbound callback service in the TRANSPORT phase as well.</p>

Note:

You can use the `pub.soap.handler:getProperty` service to know the processing point at which Integration Server invokes an outbound callback service. Specify `CallbackPhase` as the value for the *key* parameter of the `pub.soap.handler:getProperty` service.

8 MTOM Streaming

■	Configuring MTOM Streaming for a Web Service Descriptor	176
■	Integration Server Parameters for MTOM Streaming	176
■	Using MTOM Streaming for Service First Provider Web Service Descriptors	177
■	Using MTOM Streaming for WSDL First Provider Web Service Descriptors	177
■	Using MTOM Streaming for Consumer Web Service Descriptors	178
■	How MTOM Streaming Affects Saved Pipelines	179

Configuring MTOM Streaming for a Web Service Descriptor

Integration Server supports streaming the SOAP attachments based on the MTOM/XOP standards for both inbound and outbound messages. The configuration and setup that you need to perform to implement MTOM streaming differs based on the type of web service descriptor to use streamed MTOM attachments and whether you want to stream fields in inbound SOAP messages and/or in outbound SOAP messages.

The following sections describe configuration and setup for using MTOM streaming for a web service descriptor:

- [“Using MTOM Streaming for Service First Provider Web Service Descriptors” on page 177](#)
- [“Using MTOM Streaming for WSDL First Provider Web Service Descriptors” on page 177](#)
- [“Using MTOM Streaming for Consumer Web Service Descriptors” on page 178](#)

Note:

If you use WS-SecurityPolicy, and the field that is to be streamed is also being signed and/or encrypted, Integration Server cannot use MTOM streaming because Integration Server needs to keep the entire message in memory to sign and/or encrypt the message.

Note:

You can process streamed MTOM attachments in service handlers. However, be aware that once a handler has read data from a streamed MTOM attachment, it is no longer available to subsequent handlers.

Integration Server Parameters for MTOM Streaming

To use MTOM streaming with web service descriptors, you must configure server configuration parameters on Integration Server, specifically:

- **watt.server.SOAP.MTOMStreaming.enable** must be set to true to enable MTOM streaming for inbound SOAP messages and HTTP chunking for outbound requests.
- **watt.server.SOAP.MTOMStreaming.threshold** indicates the maximum number of bytes for Integration Server to handle an inbound MTOM attachment field as an in-memory `ByteStream`. Integration Server writes an inbound MTOM attachment field exceeding this size to a temporary disk file and processes the inbound MTOM Streams as a `FileStream`. The default value of 4000 bytes. Edit this server configuration parameter if you want to use a different threshold value.
- **watt.server.SOAP.MTOMStreaming.cachedFiles.location** Specifies the absolute path to the hard disk drive space that Integration Server uses to temporarily store inbound SOAP messages when performing MTOM streaming. The default value is *Integration Server_directory* \instances\instance_name\temp\mtom\cached files. Edit this server configuration parameter to use a different location for the temporary disk files.

For more information about these server configuration parameters, see the *webMethods Integration Server Administrator's Guide*.

Using MTOM Streaming for Service First Provider Web Service Descriptors

The following lists the requirements for using MTOM streaming with a service first provider web service descriptor:

- Before generating the web service descriptor, ensure the fields in the service signature for which you will want to use MTOM streaming are Objects that use the `com.wm.util.XOPObject` Java wrapper type.
- In the service, add logic to handle MTOM streaming.
 - To process an inbound request message, for each MTOM attachment that is streamed, use the `pub.soap.utils:getXOPObjectContent` service to retrieve the contents of a `com.wm.util.XOPObject` instance as a stream.

After processing the contents of the stream, close the stream by invoking the `pub.io:close` service.
 - For an outbound response message, for each MTOM attachment that is to be streamed use the `pub.soap.utils:createXOPObject` service to create an instance of the `com.wm.util.XOPObject` class from an input stream for the attachment to stream.
- Configure the server configuration parameters identified in “[Integration Server Parameters for MTOM Streaming](#)” on page 176:
- If you want to stream outbound MTOM attachments, set the web service descriptor’s **Attachment enabled** property to `true`.
- Ensure the **Pre-8.2 compatibility mode** property of the web service descriptor is set to `false`. The web service descriptor must not be running in compatibility mode.

Using MTOM Streaming for WSDL First Provider Web Service Descriptors

The following lists the requirements for using MTOM streaming with a WSDL first provider web service descriptor:

- When creating the WSDL first provider web service descriptor, select the **Enable MTOM streaming for elements of type base64Binary** check box.

This selection indicates that when creating the web service descriptor, Designer should represent the `xsd:base64Binary` types in the WSDL as Object types with the Java wrapper type `com.wm.util.XOPObject`. As a result, you will be able to use MTOM streaming for the `xsd:base64Binary` fields.
- In the service used as an operation, add logic to handle MTOM streaming for the `xsd:base64Binary` fields.

- To process an inbound request message, for each MTOM attachment that is streamed, use the `pub.soap.utils:getXOPObjContent` service to retrieve the contents of a `com.wm.util.XOPObj` instance as a stream.

After processing the contents of the stream, close the stream by invoking the `pub.io:close` service.

- For an outbound response message, for each MTOM attachment that is to be streamed use the `pub.soap.utils:createXOPObj` service to create an instance of the `com.wm.util.XOPObj` class from an input stream for the attachment to stream.
- Configure the server configuration parameters identified in “[Integration Server Parameters for MTOM Streaming](#)” on page 176.
- If you want to stream outbound MTOM attachments, set the web service descriptor’s **Attachment enabled** property to `true`.
- Ensure the **Pre-8.2 compatibility mode** property of the web service descriptor is set to `false`. The web service descriptor must not be running in compatibility mode.

Using MTOM Streaming for Consumer Web Service Descriptors

The following lists the requirements for using MTOM streaming with a consumer web service descriptor:

- When creating the consumer web service descriptor, select the **Enable MTOM streaming for elements of type base64Binary** check box.

This selection indicates that when creating the web service descriptor, Designer should represent the `xsd:base64Binary` types in the WSDL as Object types with the Java wrapper type `com.wm.util.XOPObj`. As a result, you will be able to use MTOM streaming for the `xsd:base64Binary` fields.

- When adding logic to the service that invokes the web service connector, add logic to handle MTOM streaming for the `xsd:base64Binary` fields.
 - For an outbound request message, for each MTOM attachment that is to be streamed use the `pub.soap.utils:createXOPObj` service to create an instance of the `com.wm.util.XOPObj` class from an input stream for the attachment to stream.
 - To process an inbound response message, for each MTOM attachment that is streamed, use the `pub.soap.utils:getXOPObjContent` service to retrieve the contents of a `com.wm.util.XOPObj` instance as a stream.

After processing the contents of the stream, close the stream by invoking the `pub.io:close` service.

- Configure the server configuration parameters identified in “[Integration Server Parameters for MTOM Streaming](#)” on page 176.

For more information about these server configuration parameters, see *webMethods Integration Server Administrator’s Guide*.

- If you want to stream outbound MTOM attachments, set the web service descriptor's **Attachment enabled** property to `true`.
- Ensure the **Pre-8.2 compatibility mode** property of the web service descriptor is set to `false`. The web service descriptor must not be running in compatibility mode.

How MTOM Streaming Affects Saved Pipelines

When using MTOM streaming, the *messageContext* variables and/or *XOPObj* fields will not be available in the pipeline that is saved when using:

- Audit logging when the pipeline is included
- Service caching
- `pub.flow:savePipeline` or `pub.flow:savePipelineToFile` services
- Transient error handling for a service

Note:

The *messageContext* variable is used by many services in the `pub.soap` folder to hold the SOAP message on which the service acts. *XOPObj* fields are Objects that use the `com.wm.util.XOPObj` Java wrapper type.

9 Including SOAP Headers in the Pipeline

■ Anatomy of a SOAP Header in the Pipeline	182
■ Example of a SOAP Header in the Pipeline	183

Anatomy of a SOAP Header in the Pipeline

For a web service descriptor, you can instruct Integration Server to add the contents of SOAP headers to the pipeline, making the contents of the SOAP headers available to subsequent services.

For a web service connector, the *soapHeaders* document contains the element and parameter information from the SOAP response header.

The *soapHeaders* document adheres to the following generic structure:



The following table describes the contents of the *soapHeaders* document in the pipeline:

<i>soapHeaders</i>	Document	An IData containing the SOAP headers.
<i>HDRDOC1:localName</i>	Document List	Structure of a header element (block) from the SOAP message. The <i>HDRDOC1:localName</i> document list contains one document for each header with the same QName. Note: <i>HDRDOC1</i> represents the prefix of the first header block. <i>localName</i> is a placeholder and will be replaced by the local name of the header block. Note: The <i>soapHeaders</i> document contains a document list named <i>HDRDOC#:localName</i> for each header element (block) in the SOAP message header.
Key	Description	
<i>ns1:fieldName</i>	String	Name of the first field in the header element (block) Note: The prefix <i>ns1</i> is a placeholder and will be replaced by the prefix of the child element of the header element (block). Likewise, <i>fieldName</i> is a placeholder and will be replaced by the local name of

the first child element contained in the header element (block).

nsDecls **Document** Namespaces associated with any namespace prefixes that are used in the child element names in *HDRDOC1:localName*. Each entry in *nsDecls* represents a namespace prefix/URI pair, where a key name represents a prefix and the value of the key specifies the namespace URI.

Key	Description
<i>ns1</i>	String Namespace declaration associated with the prefix <i>ns1</i> , where <i>ns1</i> is a placeholder and will be replaced by the prefix used with the first child element of the header element (block).

nsDecls **Document** Namespaces associated with any namespace prefixes used by the header elements (blocks).

Note:
This document will contain a child string named *HDRDOC#* for each namespace prefix used with a header element (block).

Key	Description
<i>HDRDOC 1</i>	String Namespace declaration associated with the prefix <i>HDRDOC1</i> .

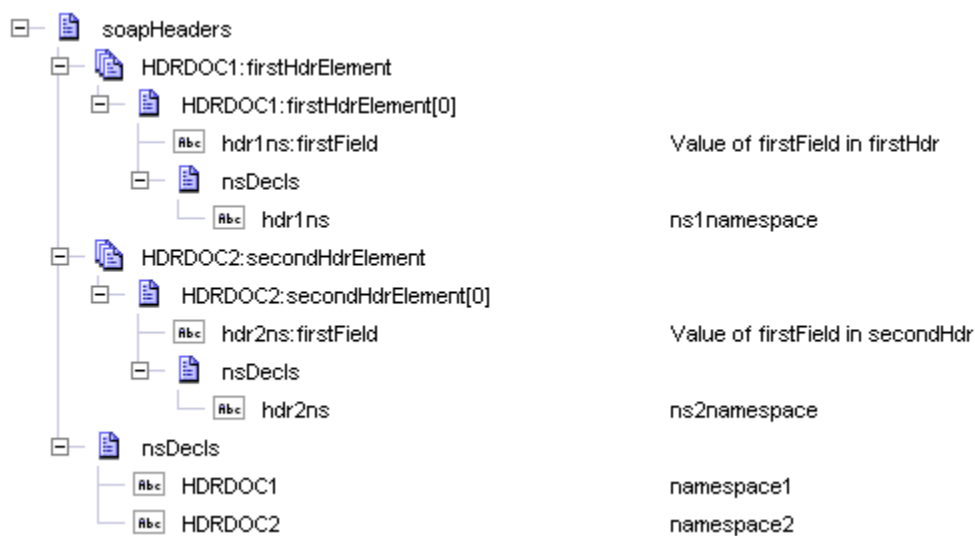
Example of a SOAP Header in the Pipeline

For example, suppose that an IS service invoked as a web service received this SOAP request:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <pfx1:firstHdrElement xmlns:pfx1="namespace1" xmlns:hdr1ns="ns1namespace">
      <hdr1ns:firstField>Value of firstField in firstHdr</hdr1ns:firstField>
    </pfx1:firstHdrElement>
    <pfx2:secondHdrElement xmlns:pfx2="namespace2"
      xmlns:hdr2ns="ns2namespace">
      <hdr2ns:firstField>Value of firstField in secondHdr</hdr2ns:firstField>
    </pfx2:secondHdrElement>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    ...
  </SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
```

Integration Server creates a *soapHeaders* document that looks like the example and adds it to the input pipeline of the IS service:



10 Web Service Authentication and Authorization

■ Introduction	186
■ Authentication and Authorization for Consumer Web Service Descriptors	186
■ ACL Checking Scenarios for Consumer Web Service Descriptors	188
■ Authentication and Authorization for Provider Web Service Descriptors	190
■ ACL Checking Scenarios for Provider Web Service Descriptors	194
■ Authentication and Authorization for Consumer Web Service Descriptors While Processing Asynchronous Responses	195
■ ACL Checking Scenarios for Consumer Web Service Descriptors While Processing Asynchronous Responses	198
■ Authentication and Authorization for Provider Web Service Descriptors on an Earlier Web Services Implementation	199
■ ACL Checking Scenarios for Provider Web Service Descriptors on an Earlier Web Services Implementation	203

Introduction

At specific points in the execution of a web service descriptor, Integration Server performs authentication and authorization to verify that the user has permission to execute the web service descriptor or one of its associated services. For the web service descriptor and its associated services, this can include checking the execute ACL assigned to the descriptor or service.

- On the consumer and provider sides, Integration Server always performs ACL checking for the web service descriptor.
- On the consumer side, Integration Server performs ACL checking for the web service connector if the connector is a top-level service (one that is invoked directly by a user). If another service invokes the web service connector, Integration Server performs ACL checking only if the **Enforce execute ACL** option for the web service connector is set to **Always**.

Integration Server performs ACL checking for handler services, if the service has permissions configured such that the **Enforce execute ACL** option is set to **Always**.

- On the provider side, Integration Server performs ACL checking for handler services, endpoint service, or any services called by the endpoint only if the service has permissions configured such that the **Enforce execute ACL** option is set to **Always**.

Authentication and Authorization for Consumer Web Service Descriptors

When Integration Server acts as the web service consumer, Integration Server authorizes the user by performing ACL checking for the consumer web service descriptor. Integration Server may perform authorization at other points by performing ACL checking for the web service connector and handler services. The following table summarizes the points at which Integration Server performs authorization and indicates when Integration Server executes the services used with a consumer web service descriptor.

Note:

On the consumer side, Integration Server performs ACL checking with the credentials used to connect to the Integration Server. The transport and message credentials passed into the web service connector or specified in the consumer web service endpoint alias are used only when sending the SOAP request to the provider.

Step	Description
------	-------------

1	Authorization check for the web service connector.
---	-----------------------------------------------------------

Integration Server determines whether the user is authorized to invoke the web service connector by checking the user credentials against the execute ACL assigned to the web service connector.

- If the web service connector is the top-level service, Integration Server performs ACL checking for the web service connector.

Step	Description
-------------	--------------------

- If the web service connector is not the top-level service, Integration Server performs ACL checking for the web service connector only if the web service connector permissions specify that the **Enforce execute ACL** option is set to **Always**.

If access is denied, Integration Server does not continue to the next steps and the web service connector fails.

2	Authorization check for the consumer web service descriptor.
----------	---------------------------------------------------------------------

Integration Server determines whether the user is authorized to access the web service descriptor by checking the user credentials against the execute ACL assigned to the web service descriptor.

If access is denied, Integration Server does not continue to the next steps and the web service connector fails.

3	Authorization check for all handler services.
----------	------------------------------------------------------

Integration Server determines whether the user is authorized to access the handler services by performing ACL checking. Integration Server checks the execute ACL for a handler service only if the handler service permissions specify that the **Enforce execute ACL** option is set to **Always**. Integration Server does not consider handler services to be top-level services.

If access is denied to any of the handler services, Integration Server does not continue to the next steps and the web service connector fails.

Note: Integration Server performs ACL checking for all request, response, and fault handler services at this point in the process. When non-Anonymous ReplyTo and/or FaultTo addresses are provided, Integration Server performs ACL checking for request handler services only. If ReplyTo is Anonymous and FaultTo is non-Anonymous, then Integration Server performs ACL checking for request and fault handler services.

4	Request handler services execute.
----------	------------------------------------------

Integration Server executes the request handler services in the handler chain. For more information, see [“About Request Handler Services” on page 159](#).

5	Authorization check for outbound callback service
----------	----------------------------------------------------------

Integration Server determines whether the user is authorized to access the outbound callback service by performing ACL checking. Integration Server checks the execute ACL for an outbound callback service only if the service permissions specify that the **Enforce execute ACL** option is set to **Always**. Integration Server does not consider outbound callback services to be top-level services.

Step Description

If access is denied to the outbound callback service, Integration Server logs an access denied error in the error logs and the processing will continue without interruption. No SOAP fault is added to the SOAP message.

For more information about outbound callback services, see [“About Outbound Callback Services” on page 171](#).

6 Outbound callback service executes

Integration Server executes the outbound callback service.

7 Send request message.

Integration Server sends the request message to the web service provider. For HTTP/S, Integration Server sends a SOAP message. For JMS, Integration Server sends a JMS message that contains a SOAP message.

8 Response handler services execute.

Integration Server receives the SOAP response and executes the response handler services in the handler chain. For more information, see [“About Response Handler Services” on page 162](#).

9 Authorization check for response services.

Note: Integration Server performs this authorization check only if the web service descriptor is processing asynchronous responses.

Integration Server determines whether the user is authorized to access the response services by performing ACL checking. Integration Server checks the execute ACL for a response service only if the response service permissions specify that the **Enforce execute ACL** option is set to **Always**. Integration Server does not consider response services to be top-level services.

If access is denied to any of the response services, Integration Server invokes the generic_FaultResponse service. If access is denied to the generic_FaultResponse service, Integration Server logs an error.

Note: Integration Server performs ACL checking for all response and fault handler services at this point in the process.

ACL Checking Scenarios for Consumer Web Service Descriptors

The following table identifies the various combinations of the ACL checking results for a consumer web service descriptor and its handler services.

Note:

The following table assumes that the user-supplied credentials passed the ACL check for the web service connector. If the user-supplied credentials did not pass the ACL check, Integration Server does not perform ACL checking for the consumer web service descriptor or its handler services.

WSD ACL Result	Handler ACL Result	Outbound Callback Service ACL Result	SOAP Fault Returned?	Exception Thrown?	Behavior
Pass	Pass	Pass	NA	NA	Integration Server executes the request handler services and outbound callback service.
Pass	Fail	NA	No	Yes	Integration Server returns a SOAP fault with the message: [ISS.0088.9433] Access denied to Handler Service <handlerServiceName> in the Handler Chain
Fail	NA	NA	Yes	No	Integration Server returns a SOAP fault with the message: [ISS.0088.9164] Access to WSDescriptor <webServiceDescriptorName> denied.
Pass	Pass	Fail	No	Yes	Integration Server logs an error in the error logs with the message IST [ISC.0088.9998E] Exception [ISS.0084.9004] Access Denied and the processing will continue without interruption.

Authentication and Authorization for Provider Web Service Descriptors

Important:

This information only applies to a provider web service descriptor that runs on the web services stack available in Integration Server 8.2 or later (that is, the **Pre-8.2 compatibility mode** property is set to false). For information about the authentication and authorization process for a provider web service descriptor that runs on the web services implementation introduced in Integration Server versions 7.1, see [“Authentication and Authorization for Provider Web Service Descriptors on an Earlier Web Services Implementation” on page 199](#).

When Integration Server acts as the web service provider, Integration Server authenticates the user when it receives a web service request. Integration Server performs authorization by checking the execute ACL for the provider web service descriptor and, if necessary, for the handler services and endpoint service. Integration Server performs these tasks using the credentials of the effective user. The identity of the effective user begins as Anonymous, but may be supplanted by transport-level credentials or message-level credentials.

Within the context of authentication and authorization for provider web service descriptors, the terms below have the specified definitions:

- **Transport-level credentials.** The credentials present available with the transport layer. For example, the userid and password in the HTTP headers are considered transport-level credentials.
- **Message-level credentials.** The use of message-level credentials for authorization purposes is only possible if WS-Security is used with the web service descriptor.
- **Effective user.** The user identity that is currently being used for purposes of authorization. The effective user identity begins as Anonymous and may be replaced subsequently by a user identity from the transport-level credentials or the message-level credentials.
- **Authentication.** The act of validating a set of credentials to verify the identity of a user. For example, authentication may involve checking the validity of a provided userid/password combination or checking the validity of an X509 certificate or its expiration. After the user is authenticated, the user identity becomes the “effective user”.
- **Authorization.** The act of determining whether a given user identity is allowed to access a particular resource.

The table below summarizes the processing points at which Integration Server performs authentication and authorization for a provider web service descriptor.

Note:

You can use WS-SecurityPolicy to secure a web service only when the web service descriptor is running on the running web services stack available in Integration Server 8.2 or later (i.e., the **Pre-8.2 compatibility mode** property is false). For more information about compatibility mode, see [“About Pre-8.2 Compatibility Mode” on page 88](#).

Step	Description
1	<p>Port access verification</p> <p>When Integration Server receives an inbound web service request through an HTTP/S port, Integration Server verifies that the provider web service descriptor can be invoked through that port.</p> <ul style="list-style-type: none"> ■ If access to the provider web service descriptor is allowed through the port, Integration Server proceeds to step 2, Transport-level authentication. ■ If access to the provider web service descriptor is denied on the port, Integration Server returns an HTTP 403 response and a SOAP fault. Integration Server rejects the web service request and no further processing occurs. <p>For more information about restricting access to ports, see the section <i>Controlling Access to Resources by Port</i> in the <i>webMethods Integration Server Administrator's Guide</i>. the section <i>Controlling Access to Resources by Port</i> in the <i>webMethods Integration Server Administrator's Guide</i>.</p> <p>Note:Integration Server does not perform port access verification for a web service request received via the JMS transport.</p>
2	<p>Transport-level authentication.</p> <p>When Integration Server receives an HTTP/S web service request, the transport mechanism authenticates the transport-level credentials.</p> <ul style="list-style-type: none"> ■ If the transport-level credentials were supplied and successfully authenticated, the user associated with the transport-level credentials becomes the effective user. Processing continues to step 3, Message-level authentication. ■ If the transport-level credentials were supplied and are invalid, the transport mechanism rejects the web service request and no further processing occurs. ■ If no transport-level credentials were provided, the effective user is set to Anonymous and processing continues to step 3, Message-level authentication. <p>When Integration Server receives a SOAP/JMS message via a SOAP-JMS trigger, the execution user assigned to the SOAP-JMS trigger becomes the effective user. Processing continues to step 3, Message-level authentication.</p>
3	<p>Message-level authentication.</p> <p>If a WS-SecurityPolicy policy is attached to the provider web service descriptor, Integration Server authenticates the message-level credentials.</p> <ul style="list-style-type: none"> ■ If authentication succeeds, Integration Server extracts the message-level credentials. The user associated with the message-level credentials becomes

Step	Description
	<p>the effective user. Processing continues with step 4, Authorization check for the provider web service descriptor.</p> <ul style="list-style-type: none">■ If authentication fails for the message-level credentials, Integration Server rejects the request, adds a SOAP fault to the SOAP response, and skips to 8, Response handler services execute. Integration Server will execute the response handler services only if the effective user is authorized to do so.
4	<p>Authorization check for the provider web service descriptor.</p> <p>Integration Server determines whether the user is authorized to access the web service descriptor by checking the credentials of the effective user against the execute ACL assigned to the web service descriptor.</p> <ul style="list-style-type: none">■ If access is granted, processing continues with step 5, Authorization check for all handler services.■ If access is denied, Integration Server adds a SOAP fault to the SOAP response and skips to 8, Response handler services execute. Integration Server will execute the response handler services only if the effective user is authorized to do so.
5	<p>Authorization check for all handler services.</p> <p>Integration Server determines whether the user is authorized to access the handler services by performing ACL checking.</p> <p>Integration Server performs ACL checking for a handler service only if the handler service permissions specify that the Enforce execute ACL option is set to Always. Integration Server does not consider handler services to be top-level services.</p> <p>Integration Server uses the credentials of the effective user when performing ACL checking for handler services. If access is denied to any of the handler services, Integration Server processing does not continue.</p> <p>Note: Integration Server performs ACL checking for all request, response, and fault handler service at this point.</p>
6	<p>Authorization check for the endpoint service.</p> <p>Integration Server determines whether the user is authorized to access the endpoint service by performing ACL checking.</p> <p>Integration Server performs ACL checking for an endpoint service only when the service permissions specify that the Enforce execute ACL option is set to Always. Integration Server does not consider an endpoint service to be a top-level service.</p>

Step	Description
	<p>If Integration Server performs ACL checking for the endpoint service, Integration Server uses the credentials of the effective user.</p> <ul style="list-style-type: none"> ■ If access is granted, processing continues to step 7, Endpoint service executes. ■ If access is denied, Integration Server adds a SOAP fault to the SOAP response and skips to step 8, Response handler services execute. Integration Server will execute the response handler services only if the effective user is authorized to do so.
7	<p>Endpoint service executes.</p> <p>Integration Server executes the endpoint service.</p> <ul style="list-style-type: none"> ■ If service execution succeeds, Integration Server converts the response from the endpoint service to a SOAP response and processing continues to step 8, Response handler services execute. ■ If service execution fails, Integration Server converts the failure to a SOAP fault and places it in the SOAP response. Processing continues to step 8, Response handler services execute.
8	<p>Response handler services execute.</p> <p>Integration Server takes the SOAP response produced by the endpoint service and begins to execute the response handler services in the handler chain. For more information, see “About Response Handler Services” on page 162.</p>
9	<p>Authorization check for outbound callback service</p> <p>Integration Server determines whether the user is authorized to access the outbound callback service by performing ACL checking. Integration Server performs ACL checking for an outbound callback service only if the service permissions specify that the Enforce execute ACL option is set to Always. Integration Server does not consider outbound callback services to be top-level services.</p> <p>Integration Server uses the credentials of the effective user when performing ACL checking for outbound callback services. If access is denied to the outbound callback service, Integration Server logs an access denied error in the error logs and the processing will continue without interruption. No SOAP fault is added to the SOAP message.</p>
10	<p>Outbound callback service executes</p> <p>Integration Server takes the SOAP response message produced by the handler service and begins to execute the outbound callback services in the handler chain.</p>

ACL Checking Scenarios for Provider Web Service Descriptors

The following tables identify the various combinations of the ACL checking results for a provider web service descriptor and its handler services.

Important:

This information only applies to a provider web service descriptor that runs on the web services stack available in Integration Server 8.2 or later (that is, the **Pre-8.2 compatibility mode** property is set to false). For information about the ACL checking scenarios for a provider web service descriptor that runs on the web services implementation introduced in Integration Server versions 7.1, see [“ACL Checking Scenarios for Provider Web Service Descriptors on an Earlier Web Services Implementation” on page 203](#).

Note:

For the JMS transport, Integration Server proceeds as if the transport credentials are provided and valid.

Transport Credentials Provided?	Message Credentials Provided?	WSD ACL Result	Handler ACL Result	Request Handler Executes?	SOAP Fault Returned?	Response Handler Executes?	Behavior
No	No	Fail	NA	No	Yes	No	2
No	No	Pass	Fail	No	Yes	No	4
No	No	Pass	Pass	Yes	NA	Yes	1
No	Yes	Fail	NA	No	Yes	No	2
No	Yes	Pass	Fail	No	Yes	No	4
No	Yes	Pass	Pass	Yes	NA	Yes	1
Yes	No	Fail	NA	No	Yes	No	2
Yes	No	Pass	Fail	No	Yes	No	4
Yes	No	Pass	Pass	Yes	NA	Yes	1
Yes	Yes	Fail	NA	No	Yes	No	2
Yes	Yes	Pass	Fail	No	Yes	No	4
Yes	Yes	Pass	Pass	Yes	NA	Yes	1
Yes, but incorrect	NA	NA	NA	No	No	No	6

Behavior Description

1 Endpoint service executes.

Behavior	Description
2	Integration Server returns the following: <ul style="list-style-type: none"> ■ An HTTP response status code of 401 ■ A SOAP fault with the message [ISS.0088.9164] Access to WSDescriptor <i><webServiceDescriptorName></i> denied.
3	Integration Server returns the following: <ul style="list-style-type: none"> ■ An HTTP response status code of 401 ■ A SOAP fault with the message [ISS.0088.9164] Access to WSDescriptor <i><webServiceDescriptorName></i> denied.
4	Integration Server returns the following: <ul style="list-style-type: none"> ■ An HTTP response status code of 401 ■ A SOAP fault with the message [ISS.0088.9433] Access denied to Handler Service <i><handlerService></i>
5	Integration Server returns the following: <ul style="list-style-type: none"> ■ An HTTP response status code of 500 ■ A SOAP fault with the message [ISS.0088.9433] Access denied to Handler Service <i><handlerService></i>
6	Integration Server returns the following: <ul style="list-style-type: none"> ■ An HTTP response status code of 401 ■ A “<h4>Access Denied</h4>” message

Authentication and Authorization for Consumer Web Service Descriptors While Processing Asynchronous Responses

Important:

This information only applies to a consumer web service descriptor that is created on Integration Server 9.0 or later and has the **Pre-8.2 compatibility mode** property set to false.

The table below summarizes the processing points at which Integration Server performs authentication and authorization for a consumer web service descriptor while processing asynchronous responses.

Step	Description
1	Port access verification

Step Description

When Integration Server receives an inbound web service response through an HTTP/S port, Integration Server verifies that the consumer web service descriptor can be invoked through that port.

- If access to the consumer web service descriptor is allowed through the port, Integration Server proceeds to step 2, Transport-level authentication.
- If access to the consumer web service descriptor is denied on the port, Integration Server returns an HTTP 403 response and a SOAP fault.

Integration Server rejects the web service request and no further processing occurs.

For more information about restricting access to ports, see the section *Controlling Access to Resources by Port* in the *webMethods Integration Server Administrator's Guide*.the section *Controlling Access to Resources by Port* in the *webMethods Integration Server Administrator's Guide*.

Note:Integration Server does not perform port access verification for a web service response received via the JMS transport.

2 Transport-level authentication.

When Integration Server receives an HTTP/S web service response, the transport mechanism authenticates the transport-level credentials.

- If the transport-level credentials were supplied and successfully authenticated, the user associated with the transport-level credentials becomes the effective user. Processing continues to step 3, Message-level authentication.
- If the transport-level credentials were supplied and are invalid, the transport mechanism rejects the web service response and no further processing occurs.
- If no transport-level credentials were provided, the effective user is set to Anonymous and processing continues to step 3, Message-level authentication.

When Integration Server receives a SOAP/JMS message via a SOAP-JMS trigger, the execution user assigned to the SOAP-JMS trigger becomes the effective user. Processing continues to step 3, Message-level authentication.

3 Message-level authentication.

If a WS-SecurityPolicy policy is attached to the consumer web service descriptor, Integration Server authenticates the message-level credentials.

- If authentication succeeds, Integration Server extracts the message-level credentials. The user associated with the message-level credentials becomes the effective user. Processing continues with step 4, Authorization check for the provider web service descriptor.

Step	Description
	<ul style="list-style-type: none"> ■ If authentication fails for the message-level credentials, Integration Server rejects the response.
4	<p>Authorization check for the consumer web service descriptor.</p> <p>Integration Server determines whether the user is authorized to access the web service descriptor by checking the credentials of the effective user against the execute ACL assigned to the web service descriptor.</p> <ul style="list-style-type: none"> ■ If access is granted, processing continues with step 5, Authorization check for response and fault handler services. ■ If access is denied, processing skips to step 7.
5	<p>Authorization check for response and fault handler services.</p> <p>Integration Server determines whether the user is authorized to access the response and fault handler services by performing ACL checking. Integration Server performs ACL checking for a response handler service only if the response handler service permissions specify that the Enforce execute ACL option is set to Always. Integration Server does not consider response handler services to be top-level services.</p> <p>Integration Server uses the credentials of the effective user when performing ACL checking for response handler services.</p> <p>If access is denied to any of the response handler services, Integration Server processing does not continue.</p> <p>Note: Integration Server performs ACL checking for all response and fault handler services at this point.</p>
6	<p>Authorization check and execution of the response service.</p> <p>Integration Server determines whether the user is authorized to access the response service by performing ACL checking.</p> <p>Integration Server performs ACL checking for a response service only when the service permissions specify that the Enforce execute ACL option is set to Always. Integration Server does not consider a response service to be a top-level service.</p> <p>If Integration Server performs ACL checking for the response service, Integration Server uses the credentials of the effective user.</p> <ul style="list-style-type: none"> ■ If access is granted, response service executes. ■ If access is denied, processing skips to step 7, Response handler services execute. Integration Server will execute the response handler services only if the effective user is authorized to do so.

Step	Description
7	Response handler services execute. Integration Server invokes the genericFault_Response service if the user is authorized to access it. If user is denied access to the generic_FaultResponse service, Integration Server logs an error.

ACL Checking Scenarios for Consumer Web Service Descriptors While Processing Asynchronous Responses

Important:

This information only applies to a consumer web service descriptor that is created in Integration Server 9.0 or later and has the **Pre-8.2 compatibility mode** property set to false.

The following tables identify the various combinations of the ACL checking results for a consumer web service descriptor, its handler services, and response services.

Note:

For the JMS transport, Integration Server proceeds as if the transport credentials are provided and valid.

Transport Credentials Provided?	Message Credentials Provided?	WSD ACL Result	Response Handler ACL Result	Response Handler Executes?	Behavior
No	No	Fail	NA	No	2
No	No	Pass	Fail	No	2
No	No	Pass	Pass	Yes	1
No	Yes	Fail	NA	No	2
No	Yes	Pass	Fail	No	2
No	Yes	Pass	Pass	Yes	1
Yes	No	Fail	NA	No	2
Yes	No	Pass	Fail	No	2
Yes	No	Pass	Pass	Yes	1
Yes	Yes	Fail	NA	No	2
Yes	Yes	Pass	Fail	No	2
Yes	Yes	Pass	Pass	Yes	1

Transport Credentials Provided?	Message Credentials Provided?	WSD ACL Result	Response Handler ACL Result	Response Handler Executes?	Behavior
Yes, but incorrect	NA	NA	NA	No	2

Behavior Description

- 1 Response service executes if the user is authorized to access the response service.
- 2 Integration Server executes the genericFault_Response service with the pipeline containing a SOAP fault if the user is authorized to access the genericFault_Response service. Otherwise, Integration Server logs an error.

Authentication and Authorization for Provider Web Service Descriptors on an Earlier Web Services Implementation

Important:

This section summarizes the authentication and authorization process for a provider web service descriptor that runs on the web services implementation introduced in Integration Server versions 7.1 (that is, the **Pre-8.2 compatibility mode** property is set to true). For information about the authentication and authorization process for a web service descriptor that runs on the web services stack available in Integration Server 8.2 or later (that is, the **Pre-8.2 compatibility mode** property is set to false), see [“Authentication and Authorization for Provider Web Service Descriptors” on page 190](#).

Note:

The **Pre-8.2 compatibility mode** property and the ability to run in pre-8.2 compatibility mode are deprecated as of Integration Server 10.4 due to the deprecation of the web services implementation introduced in Integration Server version 7.1.

When Integration Server acts as the web service provider, Integration Server authenticates the user when it receives a web service request. Integration Server performs authorization by checking the execute ACL for the provider web service descriptor and, if necessary, for the handler services and endpoint service. Integration Server performs these tasks using the credentials of the effective user. The identity of the effective user begins as Anonymous, but may be supplanted by transport-level credentials or message-level credentials.

Within the context of authentication and authorization for provider web service descriptors, the terms below have the specified definitions:

- **Transport-level credentials.** The credentials present within the transport layer. For example, the userid and password in the HTTP headers are considered transport-level credentials.
- **Message-level credentials.** The use of message-level credentials for authorization purposes is only possible if WS-Security is used with the web service descriptor.

- **Effective user.** The user identity that is currently being used for purposes of authorization. The effective user identity begins as Anonymous and may be replaced subsequently by a user identity from the transport-level credentials or the message-level credentials.
- **Authentication.** The act of validating a set of credentials to verify the identity of a user. For example, authentication may involve checking the validity of a provided userid/password combination or checking the validity of an X509 certificate or its expiration. After the user is authenticated, the user identity becomes the “effective user”.
- **Authorization.** The act of determining whether a given user identity is allowed to access a particular resource.

Note:

You can use the WS-Security handlers to secure a web service only when the web service descriptor is running in pre-8.2 compatibility mode (i.e., the **Pre-8.2 compatibility mode** property is true). For more information about compatibility mode, see [“About Pre-8.2 Compatibility Mode” on page 88](#).

Step	Description
------	-------------

1	Port access verification
---	---------------------------------

When Integration Server receives an inbound web service request through an HTTP/S port, Integration Server verifies that the provider web service descriptor can be invoked through that port.

- If access to the provider web service descriptor is allowed through 2, Transport-level authentication.
- If access to the provider web service descriptor is denied on the port, Integration Server returns an HTTP 403 response and a SOAP fault. Integration Server rejects the web service request and no further processing occurs.

For more information about restricting access to ports, see the section *Controlling Access to Resources by Port* in the *webMethods Integration Server Administrator's Guide*. the section *Controlling Access to Resources by Port* in the *webMethods Integration Server Administrator's Guide*.

2	Transport-level authentication.
---	----------------------------------------

When Integration Server receives an inbound web service request, the transport mechanism authenticates the transport-level credentials.

- If the transport-level credentials were supplied and successfully authenticated, the user associated with the transport-level credentials becomes the effective user. Processing continues to step 3, Authorization check for all handler services.
- If the transport-level credentials were supplied and are invalid, the transport mechanism rejects the web service request and no further processing occurs.

Step	Description
	<ul style="list-style-type: none"> ■ If no transport-level credentials were provided, the effective user is set to Anonymous and processing continues to step 3, Authorization check for all handler services.
3	<p>Authorization check for all handler services.</p> <p>Integration Server determines whether the user is authorized to access the handler services by performing ACL checking. Integration Server performs ACL checking for a handler service only if the handler service permissions specify that the Enforce execute ACL option is set to Always. Integration Server does not consider handler services to be top-level services.</p> <p>Integration Server uses the credentials of the effective user when performing ACL checking for handler services. If access is denied to any of the handler services, Integration Server processing does not continue.</p> <p>Note: Integration Server performs ACL checking for all request, response, and fault handler service at this point.</p>
4	<p>Request handler services execute.</p> <p>Integration Server takes the SOAP request from the consumer and executes the request handler services in the handler chain. For more information, see “About Request Handler Services” on page 159.</p> <p>If WS-Security is not in use for the provider web service descriptor, processing continues with step 5, Authorization check for the provider Web service descriptor.</p> <p>If WS-Security is in use for the provider web service descriptor, Integration Server authenticates the message-level credentials.</p> <ul style="list-style-type: none"> ■ If authentication succeeds, Integration Server extracts the message-level credentials. The user associated with the message-level credentials becomes the effective user. Processing continues with step 5, Authorization check for the provider web service descriptor. ■ If authentication fails for the message-level credentials, Integration Server rejects the request and processing skips to step 7, Response handler services execute.
5	<p>Authorization check for the provider web service descriptor.</p> <p>Integration Server determines whether the user is authorized to access the web service descriptor by checking the credentials of the effective user against the execute ACL assigned to the web service descriptor.</p> <ul style="list-style-type: none"> ■ If access is granted, processing continues with step 6, Authorization check for the endpoint service.

Step	Description
	<ul style="list-style-type: none">■ If access is denied, Integration Server adds a SOAPFault to the SOAP response and skips to 8, Response handler services execute.
6	<p>Authorization check for the endpoint service.</p> <p>Integration Server determines whether the user is authorized to access the endpoint service by performing ACL checking.</p> <p>Integration Server performs ACL checking for an endpoint service only when the service permissions specify that the Enforce execute ACL option is set to Always. Integration Server does not consider an endpoint service to be a top-level service.</p> <p>If Integration Server performs ACL checking for the endpoint service, Integration Server uses the credentials of the effective user.</p> <ul style="list-style-type: none">■ If access is granted, processing continues to step 7, Endpoint service executes.■ If access is denied, Integration Server adds a SOAP Fault to the SOAP response and skips to step 8, Response handler services execute.
7	<p>Endpoint service executes</p> <p>Integration Server executes the endpoint service.</p> <ul style="list-style-type: none">■ If service execution succeeds, Integration Server converts the response from the endpoint service to a SOAP response and processing continues to step 8, Response handler services execute.■ If service execution fails, Integration Server converts the failure to a SOAP fault and places it in the SOAP response. Processing continues to step 8, Response handler services execute.
8	<p>Response handler services execute.</p> <p>Integration Server takes the SOAP response produced by the endpoint service and begins to execute the response handler services in the handler chain. For more information, see “About Response Handler Services” on page 162.</p>
9	<p>Authorization check for outbound callback service</p> <p>Integration Server determines whether the user is authorized to access the outbound callback service by performing ACL checking. Integration Server performs ACL checking for an outbound callback service only if the service permissions specify that the Enforce execute ACL option is set to Always. Integration Server does not consider outbound callback services to be top-level services.</p> <p>Integration Server uses the credentials of the effective user when performing ACL checking for outbound callback services. If access is denied to the outbound callback service, Integration Server logs an access denied error in the error logs</p>

Step	Description
	and the processing will continue without interruption. No SOAP fault is added to the SOAP message.
10	Outbound callback service executes
	Integration Server takes the SOAP response message and begins to execute the outbound callback services.

ACL Checking Scenarios for Provider Web Service Descriptors on an Earlier Web Services Implementation

Important:

This section summarizes the ACL checking scenarios for a provider web service descriptor that runs on the web services implementation introduced in Integration Server versions 7.1 (that is, the **Pre-8.2 compatibility mode** property is set to true). For information about the ACL checking scenarios for a provider web service descriptor that runs on the web services stack available in Integration Server versions in Integration Server 8.2 or later, see [“ACL Checking Scenarios for Provider Web Service Descriptors” on page 194](#).

Note:

The **Pre-8.2 compatibility mode** property and the ability to run in pre-8.2 compatibility mode are deprecated as of Integration Server 10.4 due to the deprecation of the web services implementation introduced in Integration Server version 7.1.

The following tables identify the various combinations of the ACL checking results for a provider web service descriptor and its handler services when the provider web service descriptor runs on the web services stack implementation introduced in Integration Server version 7.1 (that is, the provider web service descriptors for which the **Pre-8.2 compatibility mode** property is set to true).

Transport Credentials Provided?	Handler ACL Result	Request Handler Executes?	Message Credentials Provided?	WSD ACL Result	SOAP Fault Returned?	Response Handler Executes?	Behavior
No	Pass	Yes	No	Pass	NA	Yes	1
No	Pass	Yes	Yes	Pass	NA	Yes	1
Yes	Pass	Yes	No	Pass	NA	Yes	1
Yes	Pass	Yes	Yes	Pass	NA	Yes	1
No	Pass	Yes	No	Fail	Yes	Yes	2
No	Pass	Yes	Yes	Fail	Yes	Yes	3
Yes	Pass	Yes	No	Fail	Yes	Yes	3

Transport Credentials Provided?	Handler ACL Result	Request Handler Executes?	Message Credentials Provided?	WSD ACL Result	SOAP Fault Returned?	Response Handler Executes?	Behavior
Yes	Pass	Yes	Yes	Fail	Yes	Yes	3
No	Fail	No	NA	NA	Yes	No	4
Yes	Fail	No	NA	NA	Yes	No	5
Yes, but incorrect	NA	No	NA	NA	No	No	6

Behavior Description

- 1 Endpoint service executes.
- 2 Integration Server returns the following:
 - An HTTP response status code of 401
 - A SOAP fault with the message [ISS.0088.9164] Access to WSDescriptor *<webServiceDescriptorName>* denied.
- 3 Integration Server returns the following:
 - An HTTP response status code of 500
 - A SOAP fault with the message [ISS.0088.9164] Access to WSDescriptor *<webServiceDescriptorName>* denied.
- 4 Integration Server returns the following:
 - An HTTP response status code of 401
 - A SOAP fault with the message [ISS.0088.9433] Access denied to Handler Service *<handlerService>*
- 5 Integration Server returns the following:
 - An HTTP response status code of 500
 - A SOAP fault with the message [ISS.0088.9433] Access denied to Handler Service *<handlerService>*
- 6 Integration Server returns the following:
 - An HTTP response status code of 401
 - A “*<h4>Access Denied</h4>*” message

11 Transient Error Handling for Provider Web Service Descriptors

■ Introduction	206
■ Transient Error Handling for an Operation Invoked via SOAP over HTTP	206
■ Transient Error Handling for an Operation Invoked by a Non-Transacted SOAP-JMS Trigger	208
■ Transient Error Handling for an Operation Invoked by a Transacted SOAP-JMS Trigger .	212

Introduction

Transient error handling determines what action Integration Server takes when a service fails because of a transient error that results in 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 failure is temporary, the trigger service might execute successfully if Integration Server waits and then re-executes the service.

You can configure services to specify that retry execution occurs automatically if the service fails because of an `ISRuntimeException`. You can also configure triggers to specify automatic retry if the associated trigger service fails because of an `ISRuntimeException`.

For provider web service descriptors, the **Binding type** property of the binder determines whether Integration Server uses the transient error handling configured for the service or the transient error handling configured for the SOAP-JMS trigger associated with the web service descriptor binder.

How Integration Server handles transient errors for web services depends not only on the value of the Binding type property, but also on the configured transient error handling properties, the message exchange pattern (MEP) of the operation, and whether or not the message is processed as part of a transaction.

Transient Error Handling for an Operation Invoked via SOAP over HTTP

The following table describes how Integration Server handles transient errors for an operation in a provider web service descriptor when the following items are true:

- The service used as the operation contains logic to catch and wrap a transient error and re-throw it as an `ISRuntimeException`.
- The web service is invoked via SOAP over HTTP or HTTPS.

Note: Integration Server ignores any transient error handling properties configured for the service handlers. Integration Server does not perform any transient error handling for handler services. The handler services need to be coded to handle any errors or exceptions that may occur and return the appropriate status code. For more information about request handlers and status codes, see [“About Handlers and Handler Services” on page 157](#)

Note:

Any provider web service descriptor for which the **Pre-8.2 compatibility mode** property is set to true functions like an In-Out operation.

Step	Description
1	Integration Server receives the message from the web service client.

Step	Description
2	Integration Server uses the endpoint URL, SOAP message header, and/or the contents of the message to route the message to the correct provider web service descriptor and operation. For more information about Integration Server determines which operation to invoke, see “Determining the Operation for an HTTP/S Request” on page 222 .
3	<p>Integration Server executes the request handler services assigned to the provider web service descriptor.</p> <p>One of the following occurs:</p> <ul style="list-style-type: none"> ■ If the request handlers execute successfully (all handlers return a status of 0), Integration Server continues to step 4. ■ If a request handler ends in failure and returns a status of 1, 2, or 3, Integration Server suspends execution of the handler chain. The status code returned by the handler service determines what action Integration Server takes next. However, Integration Server considers web service execution to be complete at this point. ■ If Integration Server throws an exception that is not caught or handled by the request handler service, Integration Server proceeds as if the request handler returned a status code of 2. Integration Server considers web service execution to be complete at this point. <p>For more information about request handlers and status codes, see “About Request Handler Services” on page 159.</p>
4	<p>If the service executes successfully, Integration Server does one of the following based on the operation MEP:</p> <ul style="list-style-type: none"> ■ For an In-Only operation, Integration Server considers web service execution to be complete. ■ For a Robust In-Only operation, Integration Server considers web service execution to be complete. ■ For an In-Out operation, Integration Server executes the response handlers and sends a SOAP response to the web service client. Integration Server considers web service execution to be complete.
5	<p>If the service fails because of a <code>ServiceException</code>, Integration Server logs the <code>ServiceException</code> to the error log and then does one of the following:</p> <ul style="list-style-type: none"> ■ For an In-Only operation, Integration Server considers web service execution to be complete. ■ For a Robust In-Only operation, Integration Server creates a SOAP response that contains the SOAP fault which, in turn, contains the <code>ServiceException</code>. Integration Server executes the response handlers and sends a SOAP

Step	Description
	response to the web service client. Integration Server considers web service execution to be complete.
	<ul style="list-style-type: none">■ For an In-Out operation, Integration Server creates a SOAP response that contains the SOAP fault which, in turn, contains the ServiceException. Integration Server executes the response handlers and sends a SOAP response to the web service client. Integration Server considers web service execution to be complete.
6	If the service fails because of an <code>ISRuntimeException</code> and the service used as the operation has a Max retry attempts set to 1 or greater, Integration Server re-executes the service. Integration Server continues to re-execute the service until the service executes to completion or Integration Server makes the maximum retry attempts.
7	If Integration Server makes the final retry attempt and the service fails because of an <code>ISRuntimeException</code> , retry failure occurs. Integration Server treats the last service failure as a ServiceException. Integration Server logs the ServiceException to the error log and then does one of the following: <ul style="list-style-type: none">■ For an In-Only operation, Integration Server considers web service execution to be complete.■ For a Robust In-Only operation, Integration Server creates a SOAP response that contains the SOAP fault which, in turn, contains the ServiceException. Integration Server executes the response handlers and sends a SOAP response to the web service client. Integration Server considers web service execution to be complete.■ For an In-Out operation, Integration Server creates a SOAP response that contains the SOAP fault which, in turn, contains the ServiceException. Integration Server executes the response handlers and sends a SOAP response to the web service client. Integration Server considers web service execution to be complete.

Transient Error Handling for an Operation Invoked by a Non-Transacted SOAP-JMS Trigger

The following table describes how Integration Server handles transient errors for an operation in a provider web service descriptor when the following items are true:

- The service used as the operation contains logic to catch and wrap a transient error and re-throw it as an `ISRuntimeException`.
- The web service is invoked via SOAP over JMS.

- A non-transacted SOAP-JMS trigger listens for messages for the web service descriptor. That is the web service endpoint alias assigned to the JMS binder uses a non-transacted SOAP-JMS trigger.
- The **Pre-8.2 compatibility mode** property is set to false for the web service descriptor.

Keep in mind the following information about transient error handling for web services invoked via SOAP-JMS triggers:

- The transient error handling for the SOAP-JMS trigger overrides any transient error handling configured for the service used as the operation.
- Integration Server ignores any transient error handling properties configured for the service handlers. That is, Integration Server does not perform any transient error handling for handler services. The handler services need to be coded to handle any errors or exceptions that may occur and return the appropriate status code. For more information about handler services and status codes, see [“About Handlers and Handler Services” on page 157](#).

Step	Description
1	The SOAP-JMS trigger receives the message from the JMS provider.
2	Integration Server uses information from the JMS message header and the SOAP message to route the message to the correct provider web service descriptor and operation.
3	Integration Server executes the request handler services assigned to the provider web service descriptor. One of the following occurs: <ul style="list-style-type: none"> ■ If the request handlers execute successfully (all handlers return a status of 0), Integration Server proceeds to step 4. ■ If a request handler ends in failure and returns a status of 1, 2, or 3, Integration Server suspends execution of the handler chain. The status code returned by the handler service determines what action Integration Server takes next. However, Integration Server considers web service execution to be complete at this point. ■ If Integration Server throws an exception that is not caught or handled by the request handler service, Integration Server proceeds as if the request handler returned a status code of 2. Integration Server considers web service execution to be complete at this point. <p>For more information about request handlers and status codes, see “About Request Handler Services” on page 159.</p>
4	Integration Server executes the service that corresponds to the operation. <ul style="list-style-type: none"> ■ If the service executes successfully, Integration Server does one of the following based on the operation MEP:

Step	Description
	<ul style="list-style-type: none">■ For an In-Only operation, Integration Server considers web service execution to be complete.■ For a Robust-In-Only operation, Integration Server considers web service execution to be complete.■ For an In-Out operation, Integration Server executes the response handlers and sends a SOAP response to the web service client. Integration Server considers web service execution to be complete. <p>■ If the service fails because of a <code>ServiceException</code>, Integration Server logs the <code>ServiceException</code> to the error log. Integration Server does one of the following based on the operation MEP:</p> <ul style="list-style-type: none">■ For an In-Only operation, Integration Server considers web service execution to be complete.■ For an Robust-In-Only operation, Integration Server logs the <code>ServiceException</code> to the error log. Integration Server creates a SOAP response that contains the SOAP fault which, in turn, contains the <code>ServiceException</code>. Integration Server executes the response handlers and sends a SOAP response to the web service client. Integration Server considers web service execution to be complete. <p>For more information about response handlers, see “About Response Handler Services” on page 162.</p> <ul style="list-style-type: none">■ For an In-Out operation, Integration Server logs the <code>ServiceException</code> to the error log. Integration Server creates a SOAP response that contains the SOAP fault which, in turn, contains the <code>ServiceException</code>. Integration Server executes the response handlers and sends a SOAP response to the web service client. Integration Server considers web service execution to be complete. <p>For more information about response handlers, see “About Response Handler Services” on page 162.</p> <p>■ If the service fails because of an <code>ISRuntimeException</code> and the SOAP-JMS trigger has a Max retry attempts set to 1 or greater, Integration Server re-executes the service. Integration Server continues to re-execute the service until the service executes to completion or Integration Server makes the maximum retry attempts.</p>
5	<p>If Integration Server makes the final retry attempt and the service fails because of an <code>ISRuntimeException</code>, retry failure occurs. The action Integration Server takes depends on the On retry failure property for the SOAP-JMS trigger used to retrieve the request message.</p> <ul style="list-style-type: none">■ If the On retry failure property is set to Throw exception, proceed to step 6.

Step	Description
	<ul style="list-style-type: none"> ■ If the On retry failure property is set to Suspend and retry later, proceed to step 7.
6	<p>When retry failure occurs and the On retry failure property is set to Throw exception, Integration Server does the following:</p> <ul style="list-style-type: none"> ■ Treats the last service failure as a <code>ServiceException</code> ■ Writes the <code>ServiceException</code> to the error log. ■ Rejects the message. If the message is persistent, Integration Server returns an acknowledgement to the JMS provider. ■ Generates a JMS retrieval failure event if the <code>watt.server.jms.trigger.raiseEventOnRetryFailure</code> property is set to true (the default). ■ If the SOAP-JMS trigger is configured to suspend on error when a fatal error occurs, Integration Server suspends the SOAP-JMS trigger. Otherwise, Integration Server processes the next message for the JMS trigger. ■ Integration Server does one of the following based on the operation MEP. <ul style="list-style-type: none"> ■ For an In-Only operation, Integration Server considers web service execution to be complete. ■ For a Robust-In-Only operation, Integration Server creates a SOAP response that contains the SOAP fault which, in turn, contains the <code>ServiceException</code>. Integration Server executes the response handlers and sends the SOAP response to the web service client. For more information about response handlers, see “About Response Handler Services” on page 162. Integration Server considers web service execution to be complete. ■ For an In-Out operation, Integration Server creates a SOAP response that contains the SOAP fault which, in turn, contains the <code>ServiceException</code>. Integration Server executes the response handlers and sends the SOAP response to the web service client. For more information about response handlers, see “About Response Handler Services” on page 162. Integration Server considers web service execution to be complete.
7	<p>When retry failure occurs and the On retry failure property is set to Suspend and retry later, Integration Server does the following:</p> <ul style="list-style-type: none"> ■ For a Robust-In-Only operation, Integration Server creates a SOAP response that contains the SOAP fault which, in turn, contains the <code>ServiceException</code>. Integration Server executes the response handlers. Integration Server then suppresses the SOAP response so that the retry

Step	Description
	processing can occur. Integration Server does not send the SOAP response to the web service client.
■	For an In-Out operation, Integration Server creates a SOAP response that contains the SOAP fault which, in turn, contains the ServiceException. Integration Server executes the response handlers. Integration Server then suppresses the SOAP response so that the retry processing can occur. Integration Server does not send the SOAP response to the web service client.

For all operations, regardless of MEP, Integration Server does the following:

- Suspends the SOAP-JMS trigger.
- Recovers the message back to the JMS provider.
- Optionally, schedules and executes a resource monitoring service.
- When the resource monitoring service indicates that the resources are available, Integration Server enables the trigger.
- Repeats the entire process beginning with step 1 (retrieving the original message from the JMS provider).

If the maximum delivery count has been met, Integration Server rejects the message. Integration Server does not generate a fault, execute response handlers, or return a response to the web service client. This is true even for a Robust-In-Only operation or an In-Out operation.

Note:

The maximum delivery count, which is controlled by the `watt.server.jms.trigger.maxDeliveryCount` property, determines the maximum number of times the JMS provider can deliver the message to the SOAP-JMS trigger.

If the `watt.server.jms.trigger.raiseEventOnRetryFailure` property is set to true (the default), Integration Server generates a JMS retrieval failure event.

Integration Server considers web service execution to be complete when the maximum delivery count is met.

Transient Error Handling for an Operation Invoked by a Transacted SOAP-JMS Trigger

The following table describes how Integration Server handles transient errors for an operation in a provider web service descriptor when the following items are true:

- The service used as the operation contains logic to catch and wrap a transient error and re-throw it as an `ISRuntimeException`.
- The web service is invoked via SOAP over JMS.
- The message exchange pattern (MEP) of the operation must be In-Only. Integration Server does not support Robust-In-Only or In-Out operations when using a transaction.
- A transacted SOAP-JMS trigger listens for messages for the web service descriptor. That is, the web service endpoint alias assigned to the JMS binder uses a transacted SOAP-JMS trigger.
- The **Pre-8.2 compatibility mode** property is set to false for the web service descriptor.

Keep the following points in mind about transient error handling web services invoked via transacted SOAP-JMS triggers:

- The transient error handling for the SOAP-JMS trigger overrides any transient error handling configured for the service used as the operation.
- Do not use service handlers with web service descriptors for which a transacted SOAP-JMS trigger functions as the listener. A SOAP-JMS trigger is considered to be transacted when the JMS connection alias that the trigger uses to retrieve messages has a transaction type of XA TRANSACTION or LOCAL TRANSACTION.

Step	Description
------	-------------

- | | |
|---|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | The SOAP-JMS trigger receives the message from the JMS provider. |
| 2 | Integration Server uses information from the JMS message header and the SOAP message to route the message to the correct provider web service descriptor and operation. |
| 3 | Integration Server executes the service that corresponds to the operation. |
| 4 | If the service executes successfully, Integration Server considers web service execution to be complete. |
| 5 | If the service fails because of a <code>ServiceException</code> , Integration Server rolls back the entire transaction. Integration Server considers web service execution to be complete. |

Rolling back the transaction causes the message to be recovered back to the JMS provider. The JMS provider marks the message as redelivered and may increment the delivery count (`JMSXDeliveryCount`) in the JMS message. At this point, the JMS provider typically makes the message available for immediate redelivery. The JMS provider continues to deliver the message to the SOAP-JMS trigger until the `watt.server.jms.trigger.maxDeliveryCount` is reached or the message is acknowledged.

If the SOAP-JMS trigger uses a document history database for exactly-once processing, Integration Server assigns the message a status of “completed” in the document history database. The next time the SOAP-JMS trigger receives the message, Integration Server acknowledges the message, discards it, and writes

Step Description

a journal log entry indicating that a Duplicate message was received. For more information about exactly-once processing for JMS triggers, see *Using webMethods Integration Server to Build a Client for JMS*.

Note:

Transacted triggers that encounter an exception must be rolled back to ensure that all the resources enlisted in the transaction are notified of the exception. With JMS triggers, rolling back the transaction results in the message being recovered back to the JMS provider, where it can be redelivered multiple times. To avoid the performance impact caused by redelivery of a message that cannot be processed due to a `ServiceException`, Software AG recommends configuring the SOAP-JMS trigger for exactly-once processing in which a document history database is used to perform duplicate detection. If the document history database is not used, Software AG recommends using another solution to prevent continuous redelivery.

- 6 If the service fails because of an `ISRuntimeException`, Integration Server does the following:
- Rolls back the entire transaction. Rolling back the transaction causes the message to be recovered back to the JMS provider. The JMS provider marks the message as redelivered and may increment the delivery count (`JMSXDeliveryCount`) in the JMS message. At this point, the JMS provider typically makes the message available for immediate redelivery.
 - Takes one of the following actions depending on the value of the **On transaction rollback** property for the SOAP-JMS trigger.
 - If the **On transaction rollback** property is set to **Recover only**, see step 7.
 - If the **On transaction rollback** property is set to **Suspend and recover**, see step 8.
- 7 When a transient error occurs and the **On transaction rollback** property is set to **Recover only**, Integration Server repeats the entire process beginning with step 1.
- Because it is possible that Integration Server receives the message almost immediately after transaction rollback, it is also possible that the temporary condition that caused the `ISRuntimeException` has not resolved and the web service will end with a transient error again.
- If the maximum delivery count has been met, Integration Server rejects the message. Integration Server considers web service execution to be complete when the maximum delivery count is met.

Step	Description
-------------	--------------------

Integration Server does not generate a fault or return a response to the web service client. This is true even for Robust-In-Only operations or In-Out operations.

Note:

The maximum delivery count, which is controlled by the `watt.server.jms.trigger.maxDeliveryCount` property, determines the maximum number of times the JMS provider can deliver the message to the SOAP-JMS trigger.

If the `watt.server.jms.trigger.raiseEventOnRetryFailure` property is set to true (the default), Integration Server generates a JMS retrieval failure event.

- | | |
|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 8 | When a transient error occurs and the On transaction rollback property is set to Suspend and recover , Integration Server does the following: |
|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------|

- Suspends the SOAP-JMS trigger temporarily.

The SOAP-JMS trigger is suspended on this Integration Server only. If the Integration Server is part of a cluster, other servers in the cluster can retrieve and process messages for the trigger.

Keep in mind that when a SOAP-JMS trigger is suspended any web service descriptor that uses the SOAP-JMS trigger as a listener will not receive any messages.

- Optionally schedules and executes a resource monitoring service.
- When the resource monitoring service indicates that the resources are available, Integration Server enables the SOAP-JMS trigger.
- Repeats the entire process beginning with step 1 (retrieving the original message from the JMS provider).

If the maximum delivery count has been met, Integration Server rejects the message. Integration Server considers web service execution to be complete when the maximum delivery count is met.

Integration Server does not generate a fault or return a response to the web service client. This is true even for Robust-In-Only and In-Out operations.

Note:

The maximum delivery count, which is controlled by the `watt.server.jms.trigger.maxDeliveryCount` property, determines the maximum number of times the JMS provider can deliver the message to the SOAP-JMS trigger.

If the `watt.server.jms.trigger.raiseEventOnRetryFailure` property is set to true (the default), Integration Server generates a JMS retrieval failure event.

12 How Integration Server Builds Consumer and Provider Endpoint URLs

- How Integration Server Builds the Consumer Endpoint URL 218
- How Integration Server Builds the Provider Endpoint URL 219

How Integration Server Builds the Consumer Endpoint URL

When invoking a web service connector, Integration Server constructs the endpoint URL using the following pieces of information:

- The port address specified by the location attribute in the WSDL from which the consumer WSD was created.
- The value of the `_url` input parameter passed in to the web service connector.
- The value of the `endpointAlias` input parameter passed in to the web service connector.
- The consumer web service endpoint alias specified in the binding of the consumer WSD.

The value of the `_url` parameter and the assigned consumer web service endpoint alias can determine the URL used to invoke the web service operation. That is, the `_url` parameter and the consumer web service endpoint alias can override all or portions of the endpoint URL specified in the WSDL.

Integration Server builds the endpoint URL at run time as follows:

Step 1 Determine the initial URL

If a value is specified for the `_url` input parameter for a web service connector, that value overrides the entire endpoint URL as specified in the original WSDL. Otherwise, Integration Server uses the endpoint URL from the original WSDL.

Step 2 Check the web service connector signature input

If a value is specified for the `endpointAlias` input parameter for a web service connector, that value overrides the endpoint alias as specified in the original WSDL.

Step 3 Override the host/port portion of the URL.

If a consumer web service endpoint alias is assigned to the binder for the invoked operation, the endpoint alias host and/or port information overrides the host and/or port information in the URL resulting from Step 1. For more information about how Integration Server uses the host and/or port information to construct the endpoint URL, see [“How the Consumer Web Service Endpoint Alias Affects the Endpoint URL” on page 218](#).

How the Consumer Web Service Endpoint Alias Affects the Endpoint URL

The host and/or port information specified in the consumer web service endpoint alias provided as the input to the web service connector or assigned to a binder determine the host:port portion of the endpoint URL. The following table explains how Integration Server builds the endpoint URL at run time when host and/or port are specified in the consumer web service endpoint alias.

Note:

If a value is specified for the `_url` input parameter for a web service connector the url value takes overrides the original URL from the WSDL. Additionally, Integration Server ignores the host and port information provided in the consumer web service endpoint alias.

Original URL in WSDL or Value of <code>_url</code> input parameter	If Host Name is...	If Port is...	Integration Server constructs this endpoint URL...
<code>http://localhost:5555/ws/folder.wsName</code>			<code>http://localhost:5555/ws/folder.wsName</code>
<code>http://localhost:5555/ws/folder.wsName</code>	<code>newHost</code>		<code>http://newHost:5555/ws/folder.wsName</code>
<code>http://localhost:5555/ws/folder.wsName</code>		<code>6666</code>	<code>http://localhost:6666/ws/folder.wsName</code>
<code>http://localhost:5555/ws/folder.wsName</code>	<code>newHost</code>	<code>6666</code>	<code>http://newHost:6666/ws/folder.wsName</code>
<code>http://localhost/ws/folder.wsName</code>			<code>http://localhost/ws/folder.wsName</code>
<code>http://localhost/ws/folder.wsName</code>	<code>newHost</code>	<code>6666</code>	<code>http://newHost:6666/ws/folder.wsName</code>
<code>http://localhost/ws/folder.wsName</code>		<code>6666</code>	<code>http://localhost:6666/ws/folder.wsName</code>
<code>http://localhost/ws/folder.wsName</code>	<code>newHost</code>		<code>http://newHost/ws/folder.wsName</code>

How Integration Server Builds the Provider Endpoint URL

When creating a WSDL for a provider web service descriptor, Integration Server determines the host and port portions of the endpoint URL using information from one of the following:

- Provider web service endpoint alias assigned to the **Port alias** property for the binder, which could be the default provider endpoint alias for Integration Server. For information about setting the default provider endpoint alias, see the section *Setting a Default Endpoint Alias for Provider Web Service Descriptors* in the *webMethods Integration Server Administrator's Guide*.the section *Setting a Default Endpoint Alias for Provider Web Service Descriptors* in the *webMethods Integration Server Administrator's Guide*.
- **Port address** property value specified for the binder.
- Integration Server hostname and primary port.

Integration Server determines the host and port portions of the endpoint URL as follows:

- Step 1** If a provider web service endpoint alias is specified for the **Port alias** property of the binder, Integration Server uses the host name and port from the provider web service endpoint alias in the endpoint URL.

If the **Port alias** property is set to `DEFAULT (aliasName)` where *aliasName* is the name of the provider endpoint alias set as the default for the protocol used by the binder, Integration Server uses the host name and port from *aliasName* in the endpoint URL.

Step 2 If a provider web service endpoint alias is not specified for the **Port alias** property of the binder nor is the property set to DEFAULT (*aliasName*), Integration Server uses the host name and port from the **Port address** property.

Note:
A blank value for **Port alias** indicates that there is not a default provider endpoint alias for the protocol used by the binder.

If the **Port alias** property is blank and later a default web service provider endpoint alias is set for the protocol used by the binder, Integration Server uses the host name and port from the default provider endpoint alias in the endpoint URL.

Step 3 If a provider web service endpoint alias is not specified for the **Port alias** property of the binder nor is the property set to DEFAULT (*aliasName*) and the **Port address** property does not have a value, Integration Server uses the Integration Server host name and primary port.

The following table shows how Integration Server determines the host and port portions in the URL for the soap:address element for a binding:

Provider Web Service Endpoint Alias specified?	Port Address specified?	Endpoint URL uses...
Yes	Yes	Hostname and port from provider web service endpoint alias <i>aliasName</i>
<i>aliasName</i> or DEFAULT(<i>aliasName</i>)		
Yes, <i>aliasName</i> or DEFAULT(<i>aliasName</i>)	No	Hostname and port from provider web service endpoint alias <i>aliasName</i>
No	Yes	Port address hostname and port
		Note: If a default provider endpoint alias is later set for Integration Server, the endpoint URL uses the hostname and port from the default provider endpoint alias.
No	No	Integration Server hostname and primary port
		Note: If a default provider endpoint alias is later set for Integration Server, the endpoint URL uses the hostname and port from the default provider endpoint alias.

13 How Integration Server Determines which Operation to Invoke

- Determining the Operation for an HTTP/S Request 222
- Determining the Operation for a SOAP/JMS Request 223
- Fallback Mechanisms for Determining the Operation 223

Determining the Operation for an HTTP/S Request

When Integration Server receives a web service request sent over HTTP/S, Integration Server uses the following process to identify the SOAP action and determine which operation (IS service) to invoke:

1. Integration Server uses the endpoint URL to determine which web service descriptor and which binder in that web service descriptor contain the operation to invoke. The endpoint URL for an operation in a WSDL document generated by Integration Server has the following format:

transport://host:port/ws/wsdName/portName

Where

- *transport* is http or https
 - *host:port* is the host and port on which the web service resides
 - *ws* is the directive that Integration Server uses for web services
 - *wsdName* is the fully qualified name of the provider web service descriptor that contains the operation
 - *portName* is the name of the port in the WSDL document that contains the operation. In a WSDL document generated by Integration Server, each port name corresponds to a binder in the web service descriptor.
2. Integration Server determines the SOAP action value:
 - For a SOAP 1.1 message, Integration Server obtains the SOAP action value from the SOAPAction HTTP header in the SOAP message
 - For a SOAP 1.2 message, Integration Server obtains the SOAP action value from the action attribute in the Content-Type header.
 3. Integration Server determines which operation is associated with that SOAP action value.
 - If the SOAP action value is unique within the selected binder, Integration Server invokes the service that corresponds to the operation with the assigned SOAP action value.
 - If Integration Server cannot determine the operation to invoke using SOAP Action and WS-Addressing action, Integration Server determines the operation to invoke by examining the fully qualified name (namespace name and local name) of the first element in the SOAP body. For more information about how Integration Server resolves duplicate SOAP action values, see [“Duplicate SOAP Action” on page 224](#).

Determining the Operation for a SOAP/JMS Request

When Integration Server receives a web service request sent over SOAP/JMS, Integration Server uses the following process to identify the SOAP action and determine which operation (IS service) to invoke:

1. Integration Server retrieves the following JMS message properties from the JMS message:

This JMS Message property...	Indicates...
soapJMS:targetService	The fully qualified name of the provider web service descriptor that contains the operation and the name of the port in the WSDL document that contains the operation. In a WSDL document generated by Integration Server, each port name corresponds to a binder in the web service descriptor.
soapJMS:soapAction	The SOAP action value.

2. Integration Server determines which operation is associated with the SOAP action value contained in soapJMS:soapAction.
 - If the SOAP action value is unique within the selected binder, Integration Server invokes the service that corresponds to the operation with the assigned SOAP action value.
 - If the SOAP action value is not unique (more than one operation in a binder share the same SOAP action value), Integration Server cannot use the SOAP action to determine the operation to invoke. Instead, Integration Server determines the operation to invoke by examining the fully qualified name (namespace name and local name) of the first element in the SOAP body. For more information about how Integration Server resolves duplicate SOAP action values, see [“Duplicate SOAP Action” on page 224](#).

Fallback Mechanisms for Determining the Operation

If the endpoint URL and SOAP header do not contain enough information for Integration Server to determine which IS service to invoke, Integration Server employs fallback mechanisms. The approach Integration Server takes to determine the operation depends on what information is missing from the endpoint URL or SOAP message.

Port Name Is Not Specified

If the endpoint URL does not specify a port name (for messages sent over HTTP/S) or the port name is missing from the soapJMS:targetService property (for messages sent via SOAP/JMS), Integration Server attempts to determine the operation by first identifying the SOAP version. Integration Server also obtains the SOAP action value from the SOAP message. Integration Server then looks in the web service descriptor for a binder that uses the specified SOAP version and

contains an operation with the specified SOAP action value. Integration Server invokes the IS service that corresponds to this operation.

If Integration Server cannot find an operation with that SOAP action value or there are multiple operations with that SOAP action value, Integration Server compares the fully qualified name of the first element in the SOAP body to the expected first element for an operation. For operations with a Document style, the expected first element is the first part element declared for the operation's input message in the WSDL document. For operations with an RPC style, the expected first element will have the same name as the operation in the WSDL document. Integration Server then invokes the IS service that corresponds to this operation.

If, after searching for the fully qualified name of the first element in the SOAP body, Integration Server cannot determine which operation to invoke, Integration Server returns a SOAP fault to the web service client.

Duplicate SOAP Action

If multiple operations share the same SOAP action value, Integration Server determines the operation to invoke by examining the fully qualified name (namespace name and local name) of the first element in the SOAP body.

Integration Server compares the element name to the expected first element for an operation. For operations with a Document style, the expected first element is the first part element declared for the operation's input message in the WSDL document. For operations with an RPC style, the expected first element will have the same name as the operation in the WSDL document. Integration Server then invokes the IS service that corresponds to this operation.

If, after searching for the fully qualified name of the first element in the SOAP body, Integration Server cannot determine the operation to invoke, Integration Server returns a SOAP fault to the web service client.

SOAP Action Value Is Empty, Absent or Cannot Be Resolved

If the SOAP message has an empty SOAP action, does not contain a SOAP action, or the SOAP action value cannot be resolved (i.e., no operation in the binder has the specified SOAP action value), Integration Server searches the binder for an operation with no defined SOAP action value or an empty SOAP action. If the binder contains a single operation that does not have a SOAP action value, Integration Server executes the IS service that corresponds to this operation. If the binder contains multiple operations without an assigned SOAP action value, Integration Server treats the absent SOAP action values as duplicate SOAP action values. For information about how Integration Server determines which operation to invoke when there are duplicate SOAP action values in a binder, see [“Duplicate SOAP Action” on page 224](#).

If the binder does not contain an operation with an empty SOAP action, Integration Server compares the fully qualified name of the first element in the SOAP body to the expected first element for an operation. For operations with a Document style, the expected first element is the first part element declared for the operation's input message in the WSDL document. For operations with an RPC style, the expected first element will have the same name as the operation in the WSDL document. Integration Server then invokes the IS service that corresponds to this operation.

If, after searching for the fully qualified name of the first element in the SOAP body, Integration Server cannot determine which operation to invoke, Integration Server returns a SOAP fault to the web service client.

14 Array Handling for Document/Literal and RPC/Literal

- How Integration Server Represents Arrays for Document/Literal and RPC/Literal 228
- Backward Compatibility for Web Service Descriptors Created in Integration Server 7.x . 229
- XML Namespace Decoding for Array Elements 229

How Integration Server Represents Arrays for Document/Literal and RPC/Literal

As of Integration Server version 8.0, the way in which Integration Server represents arrays in the schema for a generated WSDL document with a Document/Literal or RPC/Literal binding style has changed.

In Integration Server 7.x, Integration Server used a wrapping technique to represent an array element. As of Integration Server 8.0, Integration Server represents an array as a series of repeating elements without adding a wrapper element.

In the 7.x wrapping technique, Integration Server used the name of the array field as the name of wrapper element in the schema. Integration Server declared the wrapper element to be of a complex type named *ArrayOflistType*, where *listType* was the actual data type for the repeating element. Integration Server defined the complex type *ArrayOflistType* to be an unbounded sequence of an element named *ArrayOflistTypeItem*, where *listType* was the actual data type of the repeating element. The *ArrayOflistTypeItem* elements contained the actual data at run time.

The following examples illustrate how Integration Server represents an array for a Document/Literal or RPC/Literal service in version 7.x and version 8.0. In these examples, a flow service named *myFlowService* contains an input parameter named *myStringList* which is of type *String* list.

Integration Server 7.x: String List in a WSDL Schema

```
<xsd:complexType name="myFlowService">
  <xsd:sequence>
    <xsd:element name="myStringList" nillable="true" type="tns:ArrayOfstring"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="ArrayOfstring">
  <xsd:sequence>
    <xsd:element name="ArrayOfstringItem" type="xsd:string" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Integration Server 8.0: String List in a WSDL Schema

```
<xsd:complexType name="myFlowService">
  <xsd:sequence>
    <xsd:element name="myStringList" nillable="true" type="xsd:string"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

[“Backward Compatibility for Web Service Descriptors Created in Integration Server 7.x” on page 229.](#)

Backward Compatibility for Web Service Descriptors Created in Integration Server 7.x

The change in how Integration Server represents arrays in a WSDL schema for Document/Literal and RPC/Literal web services affects the structure and content of the SOAP requests and SOAP responses for the operations of the web service. To ensure that web service consumers created using a WSDL generated in Integration Server 7.x still execute successfully, Integration Server only uses the new array handling approach for provider web service descriptors created in version 8.0 or higher. A WSDL generated for any web service descriptor created in Integration Server 7.x continues to use the array wrapping technique even after the web service descriptor is edited in Integration Server 8.0.

To determine which array handling technique is used for a web service descriptor that specifies an RPC/Literal or Document/Literal binding, examine the WSDL file for a web service descriptor with an operation (service) that contains an array in the signature.

If you want a web service descriptor created in Integration Server 7.x to use the new array format in the WSDL schema, you need to create a 8.x provider web service descriptor that uses the same operations (IS services) as the 7.x provider web service descriptor. The WSDL for the new provider web service descriptor contains a schema that makes use of the new array format. Consumers must create new web service clients based on the new WSDL.

XML Namespace Decoding for Array Elements

When decoding the XML namespace for `ArrayOflistNameItem` or `ArrayOfstring` elements in a SOAP message, Integration Server uses document type-directed decoding in which the document type associated with the element determines the decoding.

- If the `ArrayOflistNameItem` or `ArrayOfstring` element corresponds to a document type field that contains a namespace, then Integration Server decodes the XML element with the XML namespace.
- If the `ArrayOflistNameItem` or `ArrayOfstring` element corresponds to a document type field that does not contain a namespace, then Integration Server ignores the namespace provided for the element in the SOAP message.
- If there is not a document type for the element then Integration Server ignores the namespace provided for the element in the SOAP message.

15 Defining Policies for Web Services (WS-Policy)

■ About WS-Policy	232
■ WS-Policy Files	232
■ About Updating WS-Policies	235
■ About Deleting WS-Policies	235

About WS-Policy

WS-Policy is a model and syntax you can use to communicate the policies associated with a web service. *Policies* describe the requirements, preferences, or capabilities of a web service. You attach a policy to a *policy subject*, for example, a service, endpoint, operation, or message. After attaching a policy to a policy subject, the policy subject becomes governed by that policy.

A policy is made up of one or more *policy assertions*. Each policy assertion conveys a requirement, preference, or capability of a given policy subject. You specify policy assertions using XML-based *policy expressions*.

webMethods Integration Server provides support for *Web Services Policy Framework* (WS-Policy) Version 1.2.

In Integration Server, a policy is specified using policy expressions in an XML file called a *WS-Policy file* (more simply called *policy file*). For more information, see [“WS-Policy Files” on page 232](#) and [“Guidelines for Creating WS-Policy Files” on page 233](#).

To have a web service governed by the policy in policy file, you attach the policy to the web service descriptor. You can attach WS-Policies at the binding operation message type level, such as input, output, and fault, in a web service descriptor.

WS-Policy Files

You save a policy in a policy file. Each policy file contains a WS-Policy definition based on the WS-Policy standard.

Integration Server comes with some policy files out of the box. For more information about out-of-the-box policies for:

- **WS-Addressing**, see [“WS-Addressing Policies Provided by Integration Server ” on page 328](#)
- **WS-Security**, see [“Policies Based on WS-SecurityPolicy that Integration Server Provides” on page 281](#)

Additionally, you can create policy files. For more information, see [“Guidelines for Creating WS-Policy Files” on page 233](#). Both out-of-the-box and user-defined policy files are stored in the following directory, which is called the *policy repository*:

`Software AG_directory \IntegrationServer\instances\instance_name\config\wss\policies`

At server startup, Integration Server validates the policies in the policy repository. Integration Server moves any policy that is invalid, for example, if it contains invalid XML syntax or structure, to the following subdirectory:

`Software AG_directory \IntegrationServer\instances\instance_name\config\wss\policies\invalid`

At server startup, Integration Server also deploys all the valid policies in the policy repository. That is, Integration Server makes the valid policies available for attachment to web service descriptors. You can attach policies to any web service descriptors.

Integration Server includes a policy manager that monitors the policies in the policy repository. As a result, when you add new policies to or update existing policies in the policy repository while Integration Server is running, the policy manager recognizes the new or updated policies and immediately validates them without requiring a server restart. If a new or updated policy is invalid, Integration Server moves it to the *Software AG_directory* `\IntegrationServer\instances\instance_name\config\wss\policies\invalid` directory. If a new or updated policy is valid, Integration Server deploys the policy so that it is available for attachment.

Note:

When Integration Servers are in a clustered environment, all servers in the cluster should have the same policy files.

Moving or Copying Web Service Descriptors with Attached Policies

Keep the following points in mind when you are copying or moving a web service descriptor having policy files attached to it to a different Integration Server:

- If you are using webMethods Deployer to deploy a package to another Integration Server, Deployer copies all assets, including the policy files) to the target server.
- If you are using the Package Navigator view, the package replication functionality in Integration Server Administrator, or executing a built-in-service to copy or move the web service descriptor, the attached policies are not copied on to the target server. You should copy all the attached policy files to the target Integration Server.

Guidelines for Creating WS-Policy Files

Keep the following points in mind, when creating policy files:

- Assign the file extension “.policy” to the policy file.
- Ensure that the WS-Policy you specify in the policy file works with the WS-Policy 1.2 standard.
- Include the `wsu:Id` attribute in the policy to provide a policy ID.

You must include a policy ID to uniquely identify the policy. For more information about specifying a policy ID, see [“Policy ID” on page 234](#).

- Include a `Name` attribute to provide a descriptive name for the policy. The `Name` attribute is optional.
- Include only one policy in each policy file.

A policy file can have only one top-level `Policy` element. If a policy file contains multiple `Policy` elements, Integration Server uses only the first one and ignores the subsequent `Policy` elements within the same file. Integration Server will not display any warning messages.

Within the single `Policy` element, you can use the `All` and `ExactlyOne` elements to provide multiple policy alternatives or policy assertions.

- Avoid including policy assertions that Integration Server does not support. If you use a WS-Policy that contains unsupported policy assertions, unexpected behavior might occur.

The following table provides information about the policy assertions that Integration Server supports.

Type	Integration Server supports...
WS-Addressing	The <code>wsaw:UsingAddressing</code> assertion, where <code>wsaw</code> refers to the namespace <code>"http://www.w3.org/2006/05/addressing/wsdl"</code> .
WS-Security	A subset of the SecurityPolicy 1.2 and WS-SecurityPolicy 1.1 assertions. For information about the supported assertions, see “WS-SecurityPolicy Assertions Reference” on page 268 .
WS-ReliableMessaging	A subset of the WS-ReliableMessaging Policy 1.1 assertions. For information about reliable messaging, see “Web Services Reliable Messaging (WS-ReliableMessaging)” on page 335 .

- Place the policy file directly in the *Software AG_directory* \IntegrationServer\instances*instance_name*\config\wss\policies directory. Integration Server will ignore any policy files that are placed in a subfolder of the policies directory.

When you add a policy file to the policies directory, the Integration Server policy manager recognizes the new policy and logs an information message to the server log stating that the policy was added.

The Integration Server policy manager also validates the new policy. If Integration Server determines that the policy is valid, Integration Server makes the policy file immediately available to be attached to web service descriptors. You do not need to restart Integration Server.

If the policy is invalid, for example, if it contains invalid XML syntax or structure, Integration Server does not make the file available. Integration Server moves invalid policy files to the *Software AG_directory* \IntegrationServer\instances*instance_name*\config\wss\policies\invalid subdirectory.

Policy ID

A *policy ID* is a unique identifier for a WS-Policy. A policy ID:

- Must conform to XML NCName (no colon) format. That is, the policy ID should begin with a letter or underscore and should not contain a space or colon.
- Must be specified using the `wsu:Id` attribute, for example:

```
wsu:Id="policyID"
```

- Must be unique among all Integration Server policies that are defined in policy files.

If you create a new policy file with a policy that has the same policy ID as an existing policy, the new policy file will be moved to the *Software AG_directory* \IntegrationServer\instances*instance_name*\config\wss\policies\invalid directory.

Important: Integration Server sorts policies randomly. When duplicate policy IDs exist, it is not possible to determine which policy file Integration Server will move to the invalid directory. Software AG recommends that you ensure that the policy ID does not exist before assigning it to a policy.

About Updating WS-Policies

You can update WS-Policies that reside in the policy repository. Integration Server logs information about the update.

Keep the following in mind when updating policies:

- Ensure that the updated policy still works with the WS-Policy 1.2 standard.
- Ensure that the updated policy still includes only policy assertions that Integration Server supports. For information about supported policy assertions, see [“Guidelines for Creating WS-Policy Files” on page 233](#).
- Avoid changing the policy ID if the policy is already attached to web service descriptors. In this case, Integration Server acts as if the policy is missing because it cannot find a policy with old policy ID. Also, if you change the policy ID to the ID of an existing policy (i.e., a duplicate policy ID), the policy becomes invalid.

The Integration Server policy manager recognizes updated policies in the policy repository and immediately validates them. If an updated policy is valid, web service descriptors associated with the policy begins using the updated policy. If an updated policy is invalid, Integration Server moves it to the following directory:

Software AG_directory \IntegrationServer\instances\instance_name\config\wss\policies\invalid

When the policy is invalid or missing, web service descriptors associated with the policy will not behave as expected. For provider web service descriptors, Integration Server will not be able to handle requests for operations contained in the provider web service descriptor. For consumer web service descriptors, the execution of the web service connectors will fail stating that the policy is not found.

About Deleting WS-Policies

You can delete WS-Policies that reside in the policy repository by removing the file from the policy repository. Integration Server logs information about the deletion.

When you delete a policy, the Integration Server policy manager recognizes the deletion. Designer will no longer include the policy in the list for policies that can be attached to web service descriptors.

If you delete a policy that is currently attached to a web service descriptor, because the policy will be missing, the web service descriptor will no longer behave as expected:

- For a provider web service descriptor, Integration Server will not be able to handle requests for operations contained in the provider web service descriptor and will log a warning message for each provider web service descriptor in the server log.

- For a consumer web service descriptor, the execution of the web service connectors will fail stating that the policy is not found.

16 Securing Web Services (WS-Security)

■ WS-Security in Integration Server	238
■ Transport-Based vs. Message-Based Security	238
■ WS-Security and the SOAP Message Header	239
■ WS-Security and the Message Direction	239
■ How You Can Secure SOAP Messages with WS-Security	240

WS-Security in Integration Server

Integration Server provides message-based security for SOAP messages using WS-Security. In contrast to transport-based authentication frameworks such as HTTPS, which secure the endpoints of a connection against threats, WS-Security secures the message transmission environment *between* endpoints. Using both transport-based and message-based security provides more complete end-to-end protection for data passing across public networks.

Integration Server provides two methods for using WS-Security.

- Using WS-SecurityPolicy.

Starting with Integration Server version 8.2, you can attach standard WS-SecurityPolicy policies to a web service descriptor. To use this method, the web service must *not* be running in pre-8.2 compatibility mode (i.e., the web service descriptor **Pre-8.2 compatibility mode** property must be set to `false`). For more information, see [“Securing Web Services Using WS-SecurityPolicy” on page 261](#).

- Using the Integration Server WS-Security facility.

Integration Server versions prior to 8.2 provided WS-Security support using the WS-Security facility. You can still use the WS-Security facility provided that the web service is configured to run in pre-8.2 compatibility mode (i.e., the web service descriptor **Pre-8.2 compatibility mode** property is set to `true`). For more information, see [“Securing Web Services Using the WS-Security Facility” on page 301](#).

Note:

The WS-Security facility is deprecated as of Integration Server 10.4 because the web services implementation with which the WS-Security facility is used is deprecated. Specifically, the web services implementation introduced in Integration Server version 7.1 is deprecated.

Transport-Based vs. Message-Based Security

WS-Security support in Integration Server is a message-based implementation that is designed to provide end-to-end network coverage.

In a *transport-based* implementation such as HTTPS, credentials and authentication information secure the endpoints of a connection. However, if the information transmitted between the endpoints is not contained within a closed network, or the message traffic is routed through intermediate public nodes, messages can be exposed to threats such as eavesdropping, unauthorized access, message replay, and parameter manipulation.

In a *message-based* implementation, such as one built according to the WS-Security standard, the security information required to pass information between web services is contained within each message header. This design caters to the securing of the message transmission environment between endpoints. You can use authentication safeguards such as signing and encryption at the individual message level to provide greater data protection than just using similar authorization features in a transport-based security architecture.

Note that the two security architectures are not mutually exclusive. You can design a solution for your web services that uses a transport-based security architecture such as SSL to secure the connection endpoints, along with a message-based, WS-Security implementation.

Note: Integration Server support of WS-Security when using the WS-Security facility does not enable or enforce any of the transport-level security measures provided by SSL and HTTP authentication.

WS-Security and the SOAP Message Header

The security information for a SOAP message is contained in an optional header component that follows the SOAP envelope. Its XML semantics and syntax are based on WS-Security design standards and recommendations for authentication components such as signing, encryption, use of authentication tokens, and security timestamps.

The security options of an outbound/inbound message pair often share a dependency. For example, if an outbound message is signed by a web service, the web service receiving the signed message must define the security parameters of an inbound message so that it can address (for example, verify) signed messages.

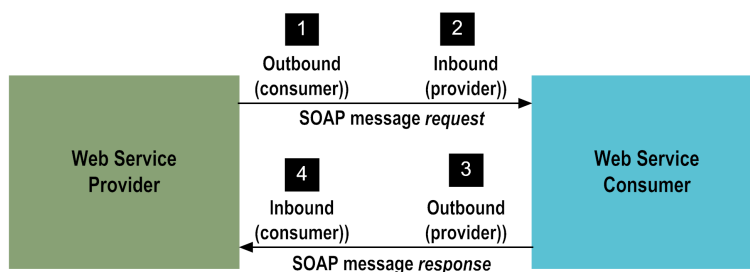
WS-Security and the Message Direction

When considering message-based security for a web service, you must take into account the message direction, that is either *inbound* or *outbound*. This distinction is important, because the security options you can implement depend on the message direction.

The security options of an outbound/inbound message pair often share a dependency. For example, if an outbound message is signed by a web service, the web service receiving the signed message must define the security parameters of an inbound message so that it can address (for example, verify) signed messages.

About Securing Web Service Providers and Consumers

WS-Security is structured around a *request-response* message exchange model between a web service consumer and a web service provider. As shown in the following figure, the message exchange is initiated by a web service consumer requesting a service from a web service provider, who processes the request and sends a response to the consumer.



Step	Description
1	The web service consumer constructs an outbound request and sends it to a web service provider.
2	The provider receives the inbound request.
3	The provider constructs an outbound response and sends it back to the consumer.
4	The consumer receives the inbound response from the provider.

When

Web Service Type	Outbound Message	Inbound Message
Consumer	<i>Sends request</i> <ul style="list-style-type: none">■ Include UsernameToken■ Use digital signature■ Timestamp message	<i>Receives response</i> <ul style="list-style-type: none">■ Decrypt messages
Provider	<i>Sends response</i> <ul style="list-style-type: none">■ Encrypt messages	<i>Receives request</i> <ul style="list-style-type: none">■ Authenticate the UsernameToken■ Verify signature■ Enforce message expiration

How You Can Secure SOAP Messages with WS-Security

The following table lists the principal categories of security options that Integration Server supports via WS-Security.

Security Option	Description
Signature	A signature is a means of authenticating a message so that the recipient is certain of the sender's identity and the integrity of the message content. Signing a message involves encrypting a message digest with the sender's private key. To verify a signed message, the recipient uses the public key corresponding to the sender's private key.
Encryption	Encryption is a means of ensuring only the intended message recipient can read the message. The sender encrypts the message using the recipient's public key. The recipient can then decrypt the message using its private key.

Security Option	Description
Security timestamps	Use a timestamp to specify the message expiration time, as well as the precision of the time measurement. This provides protection against replay attacks because inbound messages arriving after the expiration time can be invalidated.
Authentication tokens	<p>You can use the following standard WS-Security authentication tokens for authenticating a web service client:</p> <ul style="list-style-type: none">■ Username tokens. The web services consumer identifies the requestor by specifying a user name and a password (text) to authenticate the identity to a web services producer.■ X509 Certificate Authentication. A binary token type that represents either a single certificate or certificate path in X.509 certificate format.■ SAML tokens. An XML standard that facilitates secure interchange of authentication and authorization information. SAML security tokens contain assertions about user and are attached to messages using WS-Security by placing assertion elements inside the header.

Note:

You can only use SAML tokens when using WS-SecurityPolicy. The Integration Server WS-Security facility does not support SAML tokens.

17 WS-Security Certificate and Key Requirements

■ Overview	244
■ Certificate and Key Requirements for WS-Security	244
■ About Certificate and Key Resolution Order	246
■ WS-Security Key Resolution Order: Web Services Consumer	246
■ WS-Security Key Resolution Order: Web Services Provider	252

Overview

To sign a message, verify a signature of a signed message, encrypt a message, or decrypt a message, Integration Server requires access to the appropriate certificates and keys. For more information, see [“Certificate and Key Requirements for WS-Security” on page 244](#).

In a SOAP conversation, there is no mechanism to automatically exchange public keys (unlike SSL). The consumer and provider web services have access to or possess copies of the keys needed to authenticate message requests and responses.

Integration Server searches for the certificates and keys in a certain order, called the *resolution order*. As a result, you need to place the certificates/keys in the proper locations, based on the resolution order, so that Integration Server uses the certificate/keys that you want when performing WS-Security functions, for example, signing a message.

Integration Server determines the certificates and keys that it uses to enforce a policy assertion both by considering how WS-Security is applied to the web service provider or consumer and the run-time resolution orders of certificates and keys for policy selections.

When an outbound message is sent or an inbound message received, and a policy assertion invoked, a search for a certificate or key is initiated. The search follows a fixed sequence based on the type of policy assertion, whether the web service is a provider or consumer, and a number of other factors.

After the certificate or key is found, the search ends. The resolved key or certificate loads and is used for the policy assertion. The process is repeated for each policy assertion/attribute/value setting within a message that requires certificate or key resolution. When the next message is sent or received, the process begins again and repeats for each outgoing or incoming message.

Certificate and Key Requirements for WS-Security

The required certificate and keys are different based on whether you are using asymmetric binding or symmetric binding. The most notable difference is that clients do not need a private key when using symmetric binding.

Requirements When Using Asymmetric Binding

The following table describes the certificates and keys to which Integration Server requires access when using asymmetric bindings.

To use asymmetric binding to...	Certificates and Keys Required
Sign outbound messages	The sender of the outbound message requires a private key, which it uses to sign the message. The private key must correspond to the public key that the partner will use to verify the signature.

To use asymmetric binding to...	Certificates and Keys Required
Verify signed inbound messages	<p>The receiver of the inbound message requires a public key to verify the signature. The public key must correspond to the private key that the partner used to sign outbound messages.</p> <p>Additionally, if the signing certificate will be validated to ensure that it is signed by a truststore, a web service needs access to the certificate file containing the trusted root of the signing CA (truststore).</p>
Encrypt outbound messages	<p>The sender of the outbound message requires the partner's certificate with the public key, which it uses to encrypt the message.</p>
Decrypt inbound messages	<p>The receiver of the inbound message requires a private key to decrypt the message. The private key must correspond to the public key that the partner used to encrypt the outbound message.</p>

Requirements When Using Symmetric Binding

The following table describes the certificates and keys to which Integration Server requires access when using symmetric binding.

Note: Integration Server only supports symmetric binding when you implement WS-Security using WS-SecurityPolicy. The WS-Security facility does not support symmetric binding.

To use symmetric binding to...	Certificates and Keys Required
Sign outbound messages	<ul style="list-style-type: none"> ■ For an outbound request message, the consumer requires a symmetric key to sign the message. The consumer generates the symmetric key. <p>The consumer requires the partner's certificate to encrypt the symmetric key, which it places in the security header of the outbound request message.</p> <ul style="list-style-type: none"> ■ For an outbound response message, the provider requires a symmetric key to sign the message. The provider uses the encrypted symmetric key that the consumer passed in the security header of the inbound request message. To decrypt the symmetric key, the provider uses its own private key.
Verify signed inbound messages	<ul style="list-style-type: none"> ■ For an inbound response message, the consumer requires a symmetric key to verify the message signature. It uses the symmetric key it generated for the outbound request message.

To use symmetric binding to...**Certificates and Keys Required**

- For an inbound request message, the provider requires a symmetric key to verify the message signature. The provider uses the encrypted symmetric key in the security header of the inbound request message. To decrypt the symmetric key, the provider uses its own private key.

Encrypt outbound messages

- For an outbound request message, the consumer requires a symmetric key to encrypt the message. The consumer generates the symmetric key.

The consumer requires the partner's certificate to encrypt the symmetric key, which it places in the security header of the outbound request message.

- For an outbound response message, the provider requires a symmetric key to encrypt the message. The provider uses the encrypted symmetric key that the consumer passed in the security header of the inbound request message. To decrypt the symmetric key, the provider uses its own private key.

Decrypt inbound messages

- For an inbound response message, the consumer requires a symmetric key to decrypt the message. It uses the symmetric key it generated for the outbound request message.
- For an inbound request message, the provider requires a symmetric key to decrypt the message. The provider uses the encrypted symmetric key in the security header of the inbound request message. To decrypt the symmetric key, the provider uses its own private key.

About Certificate and Key Resolution Order

At run time, the web service resolves the locations of the certificates and keys it needs for policy assertions by initiating a series of searches. The locations searched depend upon factors such as:

- Whether the web service is a provider or consumer
- Which policy assertion (for example, signing or encryption) needs the key or certificate
- Whether the policy assertion is for an inbound or outbound message
- The policy attributes and their settings

WS-Security Key Resolution Order: Web Services Consumer

The following table lists the key resolution order for a web services consumer's request and response messages when the indicated security settings are enabled.

Security Setting	Request Message	Response Message
Sign	<ol style="list-style-type: none"> 1. Private key passed in to WSC signature 2. Private key specified in WS endpoint alias 3. Private key specified in the Server Settings 	
Encrypt	<ol style="list-style-type: none"> 1. Public key passed in to WSC signature 2. Public key specified in WS endpoint alias 	
Verify		<ol style="list-style-type: none"> 1. Public key in message header 2. Public key passed in to WSC signature 3. Public key specified in WS endpoint alias
Decrypt		<ol style="list-style-type: none"> 1. Private key passed in to WSC signature 2. Private key specified in WS endpoint alias 3. Private key specified in the Server Settings

Web Service Consumer: Response (Inbound Security) Detailed Usage and Resolution Order

Keep the following information in mind when reviewing the table below:

- The table refers to keystore and key aliases for the Signing Key, the Decryption Key, and the SSL Key. You can configure these keystore and key aliases on the **Security > Certificates** page of the Integration Server Administrator.
- The usage order applies to all attributes of a policy assertion except where otherwise specified. If a policy assertion is not specified, then certificate and key resolution order is not applicable.

Security Action	Additional options	Usage/Resolution Order
Signature Verification		1. Passed In (Generated WSC)

Security Action	Additional options	Usage/Resolution Order
		auth/message/partnerCert
		2. Endpoint Alias
		WS Security Properties/Partner's Certificate
		3. WS Security Header
		Public key included in header
	Validate signing certificate	1. Endpoint Alias
		WS Security Properties/Truststore
		2. Server Settings
		Truststore/Truststore Alias
Decryption		1. Passed In (Generated WSC)
		auth/message/serverCerts/keyStoreAlias
		auth/message/serverCerts/keyAlias
		2. Endpoint Alias
		WS Security Properties/Keystore Alias
		WS Security Properties/Key Alias
		3. Server Settings
		Decryption Key/Keystore Alias
		Decryption Key/Key Alias
		4. Server Settings
		SSL Key/Keystore Alias
		SSL Key/Key Alias

Web Service Consumer: Request (Outbound Security) Detailed Usage and Resolution Order

Keep the following information in mind when reviewing the table below:

- The table refers to keystore and key aliases for the Signing Key, the Decryption Key, and the SSL Key. You can configure these keystore and key aliases on the **Security > Certificates** page of the Integration Server Administrator.

- The usage order applies to all attributes of a policy assertion except where otherwise specified. If a policy assertion is not specified, then certificate and key resolution order is not applicable.

Security Action	Options	Usage/Resolution Order
UsernameToken		<ol style="list-style-type: none"> Passed In (Generated WSC) auth/message/user auth/message/pass Endpoint Alias WS Security Properties/User Name WS Security Properties/Password
Signature		<ol style="list-style-type: none"> Passed In (Generated WSC) auth/message/serverCerts/keyStoreAlias auth/message/serverCerts/keyAlias Endpoint Alias WS Security Properties/Keystore Alias WS Security Properties/Key Alias Server Settings Signing Key/Keystore Alias Signing Key/Key Alias Server Settings SSL Key/Keystore Alias SSL Key/Key Alias
	Include the certificate path	<ol style="list-style-type: none"> Passed In (Generated WSC) Entire certificate chain used with the specified value for auth/message/serverCerts/keyAlias Endpoint Alias Entire certificate chain associated with the specified Key Alias is used Server Settings Entire certificate chain associated with the Key Alias specified for Signing is used

Security Action	Options	Usage/Resolution Order
		4. Server Settings Entire certificate chain associated with the Key Alias specified for SSL is used
	Do not include the certificate path	1. Passed In (Generated WSC) Only the server's certificate (first certificate in the chain) with the specified value for auth/message/serverCerts/keyAlias is used 2. Endpoint Alias Only the server's certificate (first certificate in the chain) associated with the specified Key Alias is used 3. Server Settings Only the server's certificate (first certificate in the chain) associated with the Key Alias specified for Signing is used 4. Server Settings Only the server's certificate (first certificate in the chain) associated with the Key Alias specified for SSL is used
Encryption		1. Passed In (Generated WSC) auth/message/partnerCert 2. Endpoint Alias WS Security Properties/Partner's Certificate
X.509 Authentication		1. Passed In (Generated WSC) auth/message/serverCerts/keyStoreAlias auth/message/serverCerts/keyAlias 2. Endpoint Alias WS Security Properties/Keystore Alias WS Security Properties/Key Alias 3. Server Settings Signing Key/Keystore Alias

Security Action	Options	Usage/Resolution Order
		Signing Key/Key Alias
		4. Server Settings
		SSL Key/Keystore Alias
		SSL Key/Key Alias
	Include the certificate path	<ol style="list-style-type: none"> 1. Passed In (Generated WSC) Entire certificate chain used with the specified value for auth/message/serverCerts/keyAlias 2. Endpoint Alias Entire certificate chain associated with the specified Key Alias is used 3. Server Settings Entire certificate chain associated with the Key Alias specified for Signing is use 4. Server Settings Entire certificate chain associated with the Key Alias specified for SSL is used
	Do not include the certificate path	<ol style="list-style-type: none"> 1. Passed In (Generated WSC) Only the server's certificate (first certificate in the chain) with the specified value for auth/message/serverCerts/keyAlias is used 2. Endpoint Alias Only the server's certificate (first certificate in the chain) associated with the specified Key Alias is used 3. Server Settings Only the server's certificate (first certificate in the chain) associated with the Key Alias specified for Signing is used 4. Server Settings Only the server's certificate (first certificate in the chain) associated with the Key Alias specified for SSL is used

WS-Security Key Resolution Order: Web Services Provider

The following table lists the key resolution order for a web services provider's request and response messages when the indicated security settings are enabled.

Security Setting	Request Message	Response Message
Sign		<ol style="list-style-type: none"> 1. Private key specified in WS endpoint alias 2. Private key specified in the HTTPS listener port 3. Private key specified in the Server Settings
Encrypt		<ol style="list-style-type: none"> 1. Public key from certificate mapping 2. Public key in request message header
Verify	<ol style="list-style-type: none"> 1. Public key in message header 2. Public key from certificate mapping 	
Decrypt	<ol style="list-style-type: none"> 1. Private key in WS endpoint alias 2. Private key specified in the HTTPS listener port 3. Private key specified in the Server Settings 	

Web Service Provider: Response (Outbound Security) Detailed Usage and Resolution Order

Keep the following information in mind when reviewing the table below:

- The table refers to keystore and key aliases for the Signing Key, the Decryption Key, and the SSL Key. You can configure these keystore and key aliases on the **Security > Certificates** page of the Integration Server Administrator.
- The usage order applies to all attributes of a policy assertion except where otherwise specified. If a policy assertion is not specified, then certificate and key resolution order is not applicable.

Note:

The message addressing endpoint alias referred to in the table is the endpoint alias that is mapped to the address in the response map of the provider endpoint alias. For more information about message addressing endpoint aliases, see the section *Creating an Endpoint Alias for Message Addressing for Use with HTTP/S* in the *webMethods Integration Server Administrator's Guide*.the section *Creating an Endpoint Alias for Message Addressing for Use with HTTP/S* in the *webMethods Integration Server Administrator's Guide*.

Security Action	Options	Usage/Resolution Order
Signature		<ol style="list-style-type: none"> Message Addressing Endpoint Alias WS Security Properties/Keystore Alias WS Security Properties/Key Alias <div>Note: Applies only in case of non-anonymous asynchronous response messages and if there is a message addressing endpoint alias associated with the response endpoint address.</div> Provider Endpoint Alias WS Security Properties/Keystore Alias WS Security Properties/Key Alias Listener (Port) Settings Listener Specific Credentials/Keystore Alias Listener Specific Credentials/Key Alias Server Settings Signing Key/Keystore Alias Signing Key/Key Alias Server Settings SSL Key/Keystore Alias SSL Key/Key Alias
	Include the certificate path	<ol style="list-style-type: none"> Message Addressing Endpoint Alias Entire certificate chain associated with the specified Key Alias is used <div>Note:</div>

Security Action	Options	Usage/Resolution Order
		<p>Applies only in case of non-anonymous asynchronous response messages and if there is a message addressing endpoint alias associated with the response endpoint address.</p>
		<ol style="list-style-type: none">Provider Endpoint Alias<p>Entire certificate chain associated with the specified Key Alias is used</p>Listener (Port) Settings<p>Entire certificate chain associated with the specified Key Alias is used</p>Server Settings<p>Entire certificate chain associated with the Key Alias specified for Signing is used</p>Server Settings<p>Entire certificate chain associated with the Key Alias specified for SSL is used</p>
	Do not include the certificate path	<ol style="list-style-type: none">Message Addressing Endpoint Alias<p>Only the server's certificate (first certificate in the chain) associated with the specified Key Alias is used</p><p>Note: Applies only in case of non-anonymous asynchronous response messages and if there is a message addressing endpoint alias associated with the response endpoint address.</p>Provider Endpoint Alias<p>Only the server's certificate (first certificate in the chain) associated with the specified Key Alias is used</p>Listener (Port) Settings<p>Only the server's certificate (first certificate in the chain) associated with the specified Key Alias is used</p>

Security Action	Options	Usage/Resolution Order
Encryption		4. Server Settings Only the server's certificate (first certificate in the chain) associated with the Key Alias specified for Signing is used
		5. Server Settings Only server's certificate (1st certificate in chain) associated with the Key Alias specified for SSL is used
		1. Message Addressing Endpoint Alias WS Security Properties/Partner's Certificate <div style="background-color: #f0f0f0; padding: 5px;"> Note: Applies only in case of non-anonymous asynchronous response messages and if there is a message addressing endpoint alias associated with the response endpoint address. </div>
X.509 Authentication		2. WS Security Header Public key included in the request header
		3. Certificate Mapping Public key (certificate) associated with resolved user and Usage (in the order specified below) for one of: <ul style="list-style-type: none"> ■ Encrypt ■ VerifyAndEncrypt ■ SSL
		1. Message Addressing Endpoint Alias WS Security Properties/Keystore Alias WS Security Properties/Key Alias
		2. Endpoint Alias WS Security Properties/Keystore Alias WS Security Properties/Key Alias
		3. Server Settings

Security Action	Options	Usage/Resolution Order
		Signing Key/Keystore Alias
		Signing Key/Key Alias
		4. Server Settings
		SSL Key/Keystore Alias
		SSL Key/Key Alias
	Include the certificate path	1. Message Addressing Endpoint Alias
		Entire certificate chain associated with the specified Key Alias is used
		2. Provider Endpoint Alias
		Entire certificate chain associated with the specified Key Alias is used
		3. Server Settings
		Entire certificate chain associated with the Key Alias specified for Signing is use
		4. Server Settings
		Entire certificate chain associated with the Key Alias specified for SSL is used
	Do not include the certificate path	1. Message Addressing Endpoint Alias
		Only the server's certificate (first certificate in the chain) with the specified Key Alias is used
		2. Endpoint Alias
		Only the server's certificate (first certificate in the chain) associated with the specified Key Alias is used
		3. Server Settings
		Only the server's certificate (first certificate in the chain) associated with the Key Alias specified for Signing is used
		4. Server Settings
		Only the server's certificate (first certificate in the chain) associated with the Key Alias specified for SSL is used

Web Service Provider: Request (Inbound Security) Detailed Usage and Resolution Order

Keep the following information in mind when reviewing the table below:

- The table refers to keystore and key aliases for the Signing Key, the Decryption Key, and the SSL Key. You can configure these keystore and key aliases on the **Security > Certificates** page of the Integration Server Administrator.
- The usage order applies to all attributes of a policy assertion except where otherwise specified. If a policy assertion is not specified, then certificate and key resolution order is not applicable.

Security Action	Options	Usage/Resolution Order
UsernameToken		<ul style="list-style-type: none"> ■ WS Security Header User Name and Password
Signature Verification		<ol style="list-style-type: none"> 1. WS Security Header Public key included in the header 2. Certificate Mapping Public key (certificate) associated with resolved user and Usage (in the order specified below) for one of: <ul style="list-style-type: none"> ■ Verify ■ VerifyAndEncrypt ■ SSL
	Validate signing certificate	<ol style="list-style-type: none"> 1. Endpoint Alias WS Security Properties/Truststore 2. Listener (Port) Settings Listener Specific Credentials/Truststore Alias 3. Server Settings Truststore/Truststore Alias
	Authenticate with signing certificate	<ul style="list-style-type: none"> ■ Certificate Mapping User associated with signed certificate (public key) and Usage of one of the following: <ul style="list-style-type: none"> ■ MessageAuth

Security Action	Options	Usage/Resolution Order
		<ul style="list-style-type: none">■ Verify■ VerifyAndEncrypt■ SSL
Decryption		<ol style="list-style-type: none">1. Endpoint Alias WS Security Properties/Keystore Alias WS Security Properties/Key Alias2. Listener (Port) Settings Listener Specific Credentials/Keystore Alias Listener Specific Credentials/Key Alias3. Server Settings Decryption Key/Keystore Alias Decryption Key/Key Alias4. Server Settings SSL Key/Keystore Alias SSL Key/Key Alias
X.509 Authentication		<ul style="list-style-type: none">■ Certificate Mapping User associated with signed certificate (public key) and Usage of one of the following:<ul style="list-style-type: none">■ MessageAuth■ Verify■ VerifyAndEncrypt■ SSL
	Validate certificate	<ol style="list-style-type: none">1. Endpoint Alias WS Security Properties/Truststore2. Listener (Port) Settings Listener Specific Credentials/Truststore Alias3. Server Settings Truststore/Truststore Alias

Security Action	Options	Usage/Resolution Order
SAML Authentication	<p>Note: You can only use SAML authentication when using WS-SecurityPolicy. The WS-Security facility does not support SAML authentication.</p>	<ul style="list-style-type: none"> ■ Certificate Mapping User associated with the sender certificate (public key) and Usage of one of the following: <ul style="list-style-type: none"> ■ MessageAuth ■ Verify ■ VerifyAndEncrypt ■ SSL
		<ol style="list-style-type: none"> 1. Endpoint Alias WS Security Properties/Truststore 2. Listener (Port) Settings Listener Specific Credentials/Truststore Alias 3. Server Settings Truststore/Truststore Alias

Certificate Mapping User and Usage Resolution Order for WS-Security

Integration Server supports the mapping of a client certificate with a user ID (**User**) and the certificate's **Usage** (for more information, see the section *Importing a Certificate (Client or CA Signing Certificate) and Mapping It to a User* in the *webMethods Integration Server Administrator's Guide*.the section *Importing a Certificate (Client or CA Signing Certificate) and Mapping It to a User* in the *webMethods Integration Server Administrator's Guide*.). At run time, a web service provider can use the information in a certificate mapping.

When determining the user to use for WS-Security, Integration Server uses the following resolution order for the **User** setting when searching through Integration Server certificate mappings:

1. **User** associated with a SAML assertion.

Note:

You can only use SAML tokens when using WS-SecurityPolicy. The Integration Server WS-Security facility does not support SAML tokens.

2. **User** associated with the certificate that is used for authentication (X.509 token or signature token).
3. **User** specified in a WS-Security UsernameToken (*not* in a certificate)
4. **User** authenticated at the transport level (SSL or HTTP)

The following table lists the order for matching a requested **Usage** by a policy assertion against the **Usage** value in a certificate mapping.

If this Usage is requested...	A mapping with the first of these Usage values is returned...
Verify	Verify, VerifyAndEncrypt, SSL
Encrypt	Encrypt, VerifyAndEncrypt, SSL
VerifyAndEncrypt	VerifyAndEncrypt, SSL
MessageAuth	MessageAuth, SSL
SSLAuth	SSL

18 Securing Web Services Using WS-SecurityPolicy

■ About Implementing WS-SecurityPolicy	262
■ Securing Web Services Using Policies Based on WS-SecurityPolicy	263
■ WS-SecurityPolicy Files	267
■ WS-SecurityPolicy Assertions Reference	268
■ Policies Based on WS-SecurityPolicy that Integration Server Provides	281

About Implementing WS-SecurityPolicy

Starting with Integration Server 8.2, you can implement WS-Security using standard WS-SecurityPolicy. Using WS-SecurityPolicy for securing web services is an alternative to using the Integration Server WS-Security facility.

Integration Server supports a subset of the security assertions described in WS-SecurityPolicy 1.2, as well as WS-SecurityPolicy 1.1. For a description of the WS-SecurityPolicy assertions that Integration Server supports, see [“WS-SecurityPolicy Assertions Reference” on page 268](#).

Integration Server supports attaching WS-Policies at the binding operation message type level, such as input, output, and fault, in consumer and provider web service descriptors. To attach a WS-Policy to a web service descriptor, the **Pre-8.2 compatibility mode** property of the web service descriptor must be set to false.

The WS-Policies that you can attach to web service descriptors must reside in WS-Policy files. Integration Server provides pre-defined WS-Policies with settings for a number of standard security configurations. For a description of the out-of-the-box WS-Policies, see [“Policies Based on WS-SecurityPolicy that Integration Server Provides” on page 281](#).

You can use the out-of-the-box policies as is, or use them as templates for creating custom WS-Policies. For more information about defining your own policies, see [“WS-Policy Files” on page 232](#) and [“Guidelines for Creating WS-Policy Files” on page 233](#). When defining your own WS-Policies, be sure to only include supported policy assertions. If you use a WS-Policy that contains unsupported policy assertions, unexpected behavior might occur.

Security Options You Can Achieve with WS-SecurityPolicy

With the WS-SecurityPolicy assertions that Integration Server supports, you can achieve the following types of security for web services.

- **Authentication of the sender of a SOAP message.** Authentication ensures that the recipient of a SOAP message is sure of the sender’s identity. You can use WS-SecurityPolicy assertions to authenticate the sender using:
 - Basic authentication using the sender’s username and password for identification.
 - X.509 certificate authentication using the sender’s X.509 certificate for identification.
 - SAML authentication using the sender’s SAML assertion for identification.
 - Kerberos authentication using the sender’s Kerberos ticket for identification.
- **Integrity of web service SOAP message content.** The sender of a message can sign all or parts of the message. The recipient then verifies the signature to ensure the integrity of the message content. Signing a message involves encrypting a message digest with the sender’s private key. To verify a signed message, the recipient uses the public key that corresponds to the sender’s private key.

- **Confidentiality of the SOAP message.** The sender of a message can encrypt the message so that only the intended recipient can read the message. The sender encrypts the message using the recipient's public key. The recipient can then decrypt the message using its private key.
- **Protection against replay attacks.** The sender can place a creation and expiration timestamp in the SOAP message header. The recipient checks the timestamp and can invalidate messages that arrive after the expiration time.

Securing Web Services Using Policies Based on WS-SecurityPolicy

The following lists the main steps you need to complete to secure a web service using WS-SecurityPolicy.

➤ To secure a web service using WS-SecurityPolicy

1. Determine the WS-SecurityPolicy policies you want to use to secure a web service.
 - **You can use out-of-the-box WS-SecurityPolicy policies** that are provided with Integration Server. For more information, see [“Policies Based on WS-SecurityPolicy that Integration Server Provides” on page 281](#).
 - You can create custom WS-SecurityPolicy policies. If you want, you can use an out-of-the-box WS-SecurityPolicy policy as a template to start your custom WS-Policy. For more information about creating your own WS-Policy, see [“WS-Policy Files” on page 232](#) and [“Guidelines for Creating WS-Policy Files” on page 233](#).

Be sure to only use WS-SecurityPolicy assertions that Integration Server supports. For more information, see [“WS-SecurityPolicy Assertions Reference” on page 268](#).

2. If you want to use policies that include SAML tokens for authentication, ensure you have set up Integration Server to use SAML. For more information, see [“Requirements for Using SAML for Authentication” on page 264](#).

If you want to use policies that include Kerberos tickets for authentication, make sure you have set up Integration Server to use Kerberos. For more information, see [“Using Kerberos for Authentication” on page 266](#).

3. Ensure all the WS-SecurityPolicy policies you want to use to secure a web service are located in the following directory:

```
Software AG_directory
\IntegrationServer\instances\instance_name\
config\wss\policies
```

4. Ensure you have the certificates and keys needed to support the WS-SecurityPolicy policies in place. For more information, see [“WS-Security Certificate and Key Requirements” on page 243](#).

5. Attach the WS-SecurityPolicy policies to the web service descriptor. For instructions, see *webMethods Service Development Help*.

Note:

If you want to use MTOM streaming, be aware that if the fields to be streamed are also being signed and/or encrypted, Integration Server *cannot* use MTOM streaming because Integration Server needs to keep the entire message in memory to sign and/or encrypt the message.

Requirements for Using SAML for Authentication

If you want to use policies based on WS-SecurityPolicy that include SAML tokens for authentication, you must set up Integration Server so that it can process the SAML tokens. Integration Server supports SAML tokens only in policies attached to provider web service descriptors for inbound requests

For a provider inbound request message Integration Server must be able to validate the SAML token using its Java Authorization and Authentication Service (JAAS) login modules.

The following table lists the requirements you must meet so that Integration Server can process SAML tokens in policies based on WS-SecurityPolicy.

Requirement	Description
Security Token Service (STS) provider	<p>You must determine which STSs you want Integration Server to trust. Clients can use any STS provider that generates SAML 1.0 or 2.0 tokens. The generated SAML token must:</p> <ul style="list-style-type: none">■ Contain the client certificate if Integration Server is to process Holder-of-Key (HOK) type SAML assertions. Integration Server uses the client certificate to identify the client and map the client to an Integration Server user.■ Be signed by the STS.
Certificates for each possible issuer of SAML assertions	<p>You must create a truststore that contains the public keys of each STS. For more information about creating a truststore, see the section <i>Creating Truststore Aliases</i> in the <i>webMethods Integration Server Administrator's Guide</i>.the section <i>Creating Truststore Aliases</i> in the <i>webMethods Integration Server Administrator's Guide</i>.</p>
Identification of trusted issuers	<p>You must identify trusted STSs to Integration Server. For instructions, see "Identifying Trusted STSs to Integration Server " on page 265.</p>
Client certificates	<p>If Integration Server is to process Holder-of-Key (HOK) type SAML assertions, which contain the public key of the client, you must map the client's public key to an Integration Server user. For more information about configuring client certificates, see the section <i>Client Certificates</i> in the <i>webMethods</i></p>

Requirement	Description
	<i>Integration Server Administrator's Guide</i> .the section <i>Client Certificates</i> in the <i>webMethods Integration Server Administrator's Guide</i> .

Identifying Trusted STSs to Integration Server

If you want to use policies based on WS-SecurityPolicy that include SAML tokens for authentication, you must set up Integration Server so that it can process the SAML tokens. One of the requirements is to identify STSs you want Integration Server to trust. For other requirements, see [“Requirements for Using SAML for Authentication” on page 264](#).

➤ To identify a trusted STS to Integration Server

1. In Integration Server Administrator, go to **Security > SAML**.
2. Click **Add SAML Token Issuer**.
3. Provide information in the following fields:

For this parameter...	Specify...
Issuer Name	<p>Name of a SAML token issuer from which Integration Server should accept and process SAML assertions. Integration Server will reject SAML assertions from issuers not configured on this screen and will log a message similar to the following to the Server log:</p> <pre>2010-06-09 23:35:38 EDT [ISS.0012.0025E] Rejecting SAML assertion from issuer "SAMPLE_STS" because issuer is not configured on the Security > SAML screen.</pre> <p>This value must match the value of the Issuer field in the SAML assertion.</p>
Truststore Alias	Specifies a text identifier for the truststore, which contains the public keys of the SAML token issuer.
Certificate Alias	Specifies a text identifier for the certificate associated with the truststore alias.
Clock Skew	Clock difference between your Integration Server and the SAML token issuer.

4. Click **Save Changes**.

Using Kerberos for Authentication

If you want to use policies based on WS-SecurityPolicy that include Kerberos tickets for authentication, you must set up Integration Server so that it can process the Kerberos tickets. Integration Server supports Kerberos tickets in policies attached to provider and consumer web service descriptors for inbound and outbound requests, respectively.

For a provider inbound request message, Integration Server must be able to validate the Kerberos ticket using its Java Authorization and Authentication Service (JAAS) login modules. For a consumer outbound request message, Integration Server must be able to create the Kerberos ticket using its JAAS login modules.

Note:

Integration Server currently supports Kerberos authentication for outbound web service requests of transportType *HTTPS* only.

Note:

Kerberos Delegated Authentication is currently not supported for web service requests.

➤ To use Kerberos for inbound authentication or outbound requests

1. Configure the Kerberos settings. For instructions, see the section *Kerberos Delegated Authentication* in the *webMethods Integration Server Administrator's Guide*.the section *Kerberos Delegated Authentication* in the *webMethods Integration Server Administrator's Guide*.
2. Go to *Software AG_directory /IntegrationServer/instances/instance_name/config* and open the *is_jaas.cnf* file.

- a. For inbound, create a JAAS login context similar to this:

```
WS_KERBEROS_INBOUND {  
  
com.sun.security.auth.module.Krb5LoginModule required  
  
refreshKrb5Config=true storeKey=true isInitiator=false debug=true;  
  
};
```

- b. For outbound, create a JAAS login context similar to this:

```
WS_KERBEROS_OUTBOUND {  
  
com.sun.security.auth.module.Krb5LoginModule required debug=true;  
  
};
```

3. Supply Kerberos settings in the web service endpoint aliases. Requests that use Kerberos tickets must be transported securely so the Kerberos settings are available only after you select HTTPS as the Transport Type. For inbound, supply the Kerberos settings in the provider web service endpoint alias. For outbound, supply these settings in the consumer endpoint alias or pass in

the settings when you run the web service connector. For instructions, see the section *Creating an Endpoint Alias for a Consumer Web Service Descriptor for Use with HTTP/S* in the *webMethods Integration Server Administrator's Guide*.the section *Creating an Endpoint Alias for a Consumer Web Service Descriptor for Use with HTTP/S* in the *webMethods Integration Server Administrator's Guide*.

4. Attach the Kerberos authentication policy to the web service descriptor. For instructions, see *webMethods Service Development Help*.
5. For inbound, configure an LDAP directory service for the Kerberos key distribution center (KDC). If you are using Central Users in Integration Server, see *Administering My webMethods Server* for instructions. If you want to directly configure an LDAP in Integration Server, see the section *Configuring the Server to Use LDAP* in the *webMethods Integration Server Administrator's Guide*.the section *Configuring the Server to Use LDAP* in the *webMethods Integration Server Administrator's Guide*. for instructions.
6. For inbound, Integration Server uses the KerberosPrincipalMapper login module, which resolves the Kerberos ticket to a user name and is already present in the `is_jaas.cnf` file. The resolved principal appears as `username@realm-name` (for example, `alice@ARGOS.RNDLAB.LOC`). The KerberosPrincipalMapper login module exposes a parameter named `parse_kerberos_principal`. By default, it is set to `true`, so Integration Server parses out the user name (for example, `alice`) and then searches for that user name in Central Users or in the LDAP directories.

If you want to resolve the identified principal to something else, add your own login module above the KerberosPrincipalMapper login module. You can also remove the default KerberosPrincipalMapper by commenting out that line. You can access the authenticated principal using this code in the `authenticate` method of your JAAS login module:

```
Map headers = userCreds.getHeaderFields();

if(headers == null || headers.isEmpty()) {

    return false;

}

Principal principal = (Principal)headers.get

("sin.jaas.binary.security.token.principal");
```

For more information on how to build and deploy your own JAAS login module, see the customizing authentication section of *webMethods Integration Server Administrator's Guide*.

WS-SecurityPolicy Files

You save the WS-Policy files in the following location:

`Software AG_directory \IntegrationServer\instances\instance_name\config\wss\policies`

For more information about WS-Policy and creating WS-Policy files, see [“Defining Policies for Web Services \(WS-Policy\)” on page 231](#).

Integration Server provides a number of pre-defined WS-SecurityPolicy policies with settings for a number of standard security configurations. You can use these policies out of the box, or as templates for creating custom WS-Policies. For a description of the out-of-the-box WS-Policies, see [“Policies Based on WS-SecurityPolicy that Integration Server Provides” on page 281](#).

Integration Server supports the security assertions described in WS-SecurityPolicy 1.2, as well as WS-SecurityPolicy 1.1 standards. However, when creating your own policies based on WS-SecurityPolicy standards, be aware that Integration Server support of WS-SecurityPolicy 1.2 and WS-SecurityPolicy 1.1 assertions are limited.

If you use a WS-Policy that contains policy assertions that Integration Server does not support, unexpected behavior might occur. For more information about the security properties that Integration Server supports, see [“WS-SecurityPolicy Assertions Reference” on page 268](#).

WS-SecurityPolicy Assertions Reference

The following table identifies where you can find information about the security properties of WS-SecurityPolicy 1.2 and WS-SecurityPolicy 1.1 assertions that Integration Server supports.

Important: Integration Server currently supports only the WS-SecurityPolicy 1.2 and WS-SecurityPolicy 1.1 assertions that are specified in this section.

Note:
Some of the WS-SecurityPolicy assertions mentioned in this help file are specific to WS-SecurityPolicy 1.2 standards and are not available in WS-SecurityPolicy 1.1 standards.

See this topic...	For supported security properties of WS-SecurityPolicy assertions used to...
“Protection Assertions” on page 268	Identify the parts or elements of a message that are protected and the level of protection to provide.
“Token Assertions” on page 270	Protect or bind tokens and claims to a message.
“Security Binding Assertions” on page 273	Secure message exchanges.
“Supporting Tokens” on page 276	Add additional tokens to a message.
“WSS: SOAP Message Security Options” on page 280	Indicate whether the initiator and recipient of a message must be able to process a given reference mechanism or whether the initiator and recipient can send a fault when such references are encountered.

Protection Assertions

Use protection assertions to identify the parts or elements of a SOAP message that are protected and to specify the level of protection to provide. There are two basic types:

- Integrity assertions that define the parts or elements of the message that should be signed.
- Confidentiality assertions that define the parts or elements of message to encrypt.

Important:

If you are implementing WS-SecurityPolicy 1.2 standards, the `sp` prefix in the assertions described below represents this namespace: <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702>. If you are implementing WS-SecurityPolicy 1.1 standards, the `sp` prefix represents this namespace: <http://schemas.xmlsoap.org/ws/2005/07/securitypolicy>.

Integrity Assertions

The following table lists the WS-Security integrity assertions that Integration Server supports.

Integrity Assertions	Add to a WS-Policy to...
<code><sp:SignedParts></code>	<p>Specify whether the body is to be signed and the SOAP header elements are to be signed.</p> <p>Integration Server supports the following elements:</p> <ul style="list-style-type: none"> ■ <code><sp:Body></code> ■ <code><sp:Header></code> ■ <code><sp:Attachments></code> (WS-SecurityPolicy 1.2 only)
<code><sp:SignedElements></code>	Specify the elements of a message that require integrity protection.

Confidentiality Assertions

The following table lists the WS-Security confidentiality assertions that Integration Server supports.

Confidentiality Assertions	Add to a WS-Policy to...
<code><sp:EncryptedParts></code>	<p>Specify whether the body and the SOAP header elements in the message are to be encrypted.</p> <p>Integration Server supports the following elements:</p> <ul style="list-style-type: none"> ■ <code><sp:Body></code> ■ <code><sp:Header></code> ■ <code><sp:Attachments></code> (WS-SecurityPolicy 1.2 only)
<code><sp:EncryptedElements></code>	Specify the elements in the message that require confidentiality protection.
<code><sp:ContentEncrypted Elements></code>	Specify the elements in the message that require confidentiality protection of their content.

Confidentiality Assertions

Add to a WS-Policy to...

Note:
WS-SecurityPolicy 1.2 only.

Required Elements Assertions

Required Elements Assertions

Add to a WS-Policy to...

<sp:RequiredElements> Specify the header elements that *must* be present in a message.

Required Parts Assertion

Required Parts Assertions

Add to a WS-Policy to...

<sp:RequiredParts> Specify the header elements that *must* be present in a message.

Integration Server supports the following element:

■ <sp:Header>

Token Assertions

Use token assertions to specify the types of tokens to use to protect messages. The following table lists the WS-Security token assertions that Integration Server supports.

Important:

If you are implementing WS-SecurityPolicy 1.2 standards, the `sp` prefix in the assertions described below represents this namespace: <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702>. If you are implementing WS-SecurityPolicy 1.1 standards, the `sp` prefix represents this namespace: <http://schemas.xmlsoap.org/ws/2005/07/securitypolicy>.

Token Assertion

Add to a WS-Policy to...

<sp:UsernameToken> Specify that a UsernameToken should be used to protect the message with a username and password. The child elements of <sp:UsernameToken> assertion are optional and are not needed for ordinary authentication.

Integration Server supports the following attribute of <sp:UsernameToken>:

■ <sp:IncludeToken>

Token Assertion**Add to a WS-Policy to...**

For more information on setting the value for the `<sp:IncludeToken>` attribute, see [“Valid Values for `<sp:IncludeToken>` Attribute” on page 272](#).

Integration Server also supports the following nested assertions:

- `<sp:NoPassword>` (WS-SecurityPolicy 1.2 only)
- `<sp:HashPassword>` (WS-SecurityPolicy 1.2 only)
- `<sp:WssUsernameToken10>`
- `<sp:WssUsernameToken11>`

Note: Integration Server supports the nested assertions `<sp:NoPassword>` and `<sp:HashPassword>` for consumer web service descriptors only.

`<sp:X509Token>`

Specify that an X509Token should be used to protect the message with an X.509 certificate.

Integration Server supports the following attribute of `<sp:X509Token>`:

- `<sp:IncludeToken>`

For more information on setting the value for the `<sp:IncludeToken>` attribute, see [“Valid Values for `<sp:IncludeToken>` Attribute” on page 272](#).

Integration Server also supports the following nested assertions:

- `<sp:RequireIssuerSerialReference>`
- `<sp:RequireThumbprintReference>`
- `<sp:WssX509V3Token10>`
- `<sp:WssX509PkiPathV1Token10>`
- `<sp:WssX509V1Token11>`
- `<sp:WssX509V3Token11>`
- `<sp:WssX509PkiPathV1Token11>`

`<sp:HttpsToken>`

Indicate that an HttpsToken should be used to protect messages, which means HTTPS is used.

Integration Server supports the following nested assertions:

Token Assertion

Add to a WS-Policy to...

- `<sp:HttpBasicAuthentication>` (WS-SecurityPolicy 1.2 only)
- `<sp:RequireClientCertificate>`

`<sp:IssuedToken>`

Indicate that an issued token is required. An IssuedToken is issued by a token issuer using the mechanisms defined in WS-Trust and Integration Server uses this when authenticating using SAML. For example, the initiator may need to request a SAML token from a given token issuer in order to secure messages sent to the recipient.

Integration Server supports `<sp:IssuedToken>` only in case of provider web service descriptors.

Integration Server supports the following attribute of `<sp:IssuedToken>`:

- `<sp:IncludeToken>`

For more information on setting the value for the `<sp:IncludeToken>` attribute, see [“Valid Values for `<sp:IncludeToken>` Attribute” on page 272](#)

- Integration Server supports the following nested assertions:
 - `<sp:RequestSecurityTokenTemplate>`
 - `<sp:RequireInternalReference>`

Valid Values for `<sp:IncludeToken>` Attribute

When you use a UsernameToken, you can use the IncludeToken property to specify when the UsernameToken should be included when messages are exchanged.

The following table lists the URI values you can use for an `<sp:IncludeToken>` in a WS-Policy.

Important:

When using the values described in the following table, replace `<URI>` with the appropriate value. If you are implementing WS-SecurityPolicy 1.2 standards, replace `<URI>` with `http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702`. If you are implementing WS-SecurityPolicy 1.1 standards, replace `<URI>` with `http://schemas.xmlsoap.org/ws/2005/07/securitypolicy`.

**Set `<sp:IncludeToken>` to If the Username token....
this URI value...**

`<URI>/IncludeToken/
Never`

Must *not* be included in any messages sent between the initiator and the recipient.

Set `<sp:IncludeToken>` to `If the Username token....` this URI value...

An external reference to the token should be used.

`<URI>/IncludeToken/ Once` Must be included in only one message sent from the initiator to the recipient.

References to the token can use an internal reference mechanism. Subsequent related messages sent between the recipient and the initiator can refer to the token using an external reference mechanism.

`<URI>/IncludeToken/
AlwaysToRecipient` Must be included in all messages sent from initiator to the recipient.

-and-

Must *not* be included in messages sent from the recipient to the initiator.

`<URI>/IncludeToken/
AlwaysToInitiator` Must be included in all messages sent from the recipient to the initiator.

-and-

Must *not* be included in messages sent from the initiator to the recipient.

Note:

This URI value for `<sp:IncludeToken>` is specific to WS-SecurityPolicy 1.2.

`<URI>/IncludeToken/
Always` Must be included in all messages sent between the initiator and the recipient. This is the default behavior when no `IncludeToken` property is specified with the `UserName` token.

Security Binding Assertions

Use security binding assertions to define the properties of the mechanisms used to secure message exchanges, such as the keys being used, algorithms, and layout. Integration Server supports all three security binding assertions: `<sp:TransportBinding>`, `<sp:SymmetricBinding>`, and `<sp:AsymmetricBinding>`.

You can use the following security binding properties to provide additional information to the security binding assertions:

- **Algorithm Suite (`<sp:AlgorithmSuite>`)**, which specifies an algorithm suite that includes cryptographic algorithms for performing operations, such as signing and encryption. Integration Server support of WS-SecurityPolicy includes the following algorithm suites in the WS-SecurityPolicy 1.2 and WS-SecurityPolicy 1.1 standards:

- `<sp:Basic256>`
- `<sp:Basic192>`
- `<sp:Basic128>`
- `<sp:TripleDes>`
- `<sp:Basic256Rsa15>`
- `<sp:Basic192Rsa15>`
- `<sp:Basic128Rsa15>`
- `<sp:TripleDesRsa15>`
- `<sp:Basic256Sha256>`
- `<sp:Basic192Sha256>`
- `<sp:Basic128Sha256>`
- `<sp:TripleDesSha256>`
- `<sp:Basic256Sha256Rsa15>`
- `<sp:Basic192Sha256Rsa15>`
- `<sp:Basic128Sha256Rsa15>`
- `<sp:TripleDesSha256Rsa15>`

- **Timestamp (`<sp:IncludeTimestamp>`)**, which specifies whether a Timestamp element must be included in the security header.

If this property is set to `true`, the timestamp element must be present and be integrity protected either by transport or message level security.

If this property is set to `false`, the timestamp element must not be present. The default value for this property is `false`.

- **Protection order**, which specifies the order of signature and encryption operations in case the content has to be signed as well as encrypted. You can set either of the following values for this property:
 - `<sp:EncryptBeforeSigning>`: Performs the encryption operation before generating a signature for the message. Signature must be computed over cipher text. Encryption key and signing key must be derived from the same source key unless distinct keys are provided.
 - `<sp:SignBeforeEncrypting>`: Generates a signature for the message before encrypting any portion of the message. Signature must be computed over plain text. The resulting signature should be encrypted and the supporting signatures must be over the plain text signature. This is the default value.
- **Signature protection (`<sp:EncryptSignature>`)**, which specifies whether the signature must be encrypted.

If this property is set to `true`, the primary signature as well as any signature confirmation elements, if present, must be encrypted. If there are no signature confirmation elements, it is not required to encrypt the primary signature element.

If this property is set to `false`, the primary signature as well as any signature confirmation elements, if present, must *not* be encrypted. The default value for this property is `false`.

- **Token protection (<sp:ProtectTokens>)**, which specifies whether the signature should cover the token that is used to generate that signature.

If this property is set to `true`, each signature digest over the SOAP body must be over the entire SOAP body element and each signature digest over a SOAP header must be over an actual header element and not a descendant of a header element.

If this property is set to `false`, the signature digests can be over a descendant of the SOAP body or header element. The default value for this property is `false`.

- **Entire Header and Body Signatures (<sp:OnlySignEntireHeadersAndBody>)**, which specifies whether the signature digests over the SOAP body and SOAP headers should only cover the entire body and entire header elements.

If this property is set to `true`, each signature digest over the SOAP body must be over the entire SOAP body element and each signature digest over a SOAP header must be over an actual header element and not a descendant of a header element.

If this property is set to `false`, the signature digests can be over a descendant of the SOAP body or header element. The default value for this property is `false`.

- **Security Header Layout (<sp:Layout>)**, which specifies the security header layout. Integration Server supports the `<sp:Strict>` optional element of the `<sp:Layout>` assertion.

Integration Server always generates the Security header following the Strict layout rules specified in the web Services Security 1.0 standards according to a general principle of “declare before use”. While processing an inbound message, Integration Server always follows the Lax layout rules with the security header layout.

Important:

If you are implementing WS-SecurityPolicy 1.2 standards, the `sp` prefix in the assertions described below represents this namespace: <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702>. If you are implementing WS-SecurityPolicy 1.1 standards, the `sp` prefix represents this namespace: <http://schemas.xmlsoap.org/ws/2005/07/securitypolicy>.

The following table lists the WS-Security security binding assertions that Integration Server supports.

Security Binding Assertion Add to a WS-Policy to...

<code><sp:TransportBinding></code>	Indicate that message protection and security correlation is provided using a secure transport.
------------------------------------------	-------------------------------------------------------------------------------------------------

Integration Server supports all the nested assertions of `<sp:TransportBinding>`.

Security Binding Assertion Add to a WS-Policy to...

`<sp:SymmetricBinding>` Indicate that message protection is provided using asymmetric key (Public Key) technology, such as RSA. For a symmetric binding only one party generates the security tokens used for signing and encrypting. Although the same key pair can be used for both signing and encryption, it is common practice to use distinct keys for each because of their different life cycles.

Integration Server supports all the nested assertions of `<sp:SymmetricBinding>`.

`<sp:AsymmetricBinding>` Indicate that message protection is provided using asymmetric key (Public Key) technology, such as RSA. For a asymmetric binding both parties generate the security tokens used for signing and encrypting. With asymmetric bindings:

- For signing, the initiator uses its private key. As a result, the recipient can use the initiator's public key for verification.
- For encryption, the initiator uses the recipient's public key. As a result, the recipient can decrypt using its private key.

Although the same key pair can be used for both signing and encryption, it is common practice to use distinct keys for each because of their different life cycles.

Integration Server supports all the nested assertions of `<sp:AsymmetricBinding>`.

Supporting Tokens

Use supporting tokens to add additional tokens to a message. You can also use Supporting tokens to sign and encrypt additional elements with the help of protection assertions.®

Important:

If you are implementing WS-SecurityPolicy 1.2 standards, the `sp` prefix in the assertions described below represents this namespace: <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702>. If you are implementing WS-SecurityPolicy 1.1 standards, the `sp` prefix represents this namespace: <http://schemas.xmlsoap.org/ws/2005/07/securitypolicy>.

The following table lists the WS-Security supporting token assertions that Integration Server supports.

Supporting Token Assertion**Add to a WS-Policy to...**

`<sp:SupportingTokens>`

Identify the tokens to use to populate the `<Supporting Tokens>` property. Supporting tokens are included in

Supporting Token Assertion**Add to a WS-Policy to...**

the security header and can optionally include additional message parts to sign or encrypt.

Integration Server supports the following nested assertions:

- [Token Assertion] (See [“Token Assertions” on page 270](#))
- <sp:AlgorithmSuite>
- <sp:SignedParts>
- <sp:SignedElements>
- <sp:EncryptedParts>
- <sp:EncryptedElements>

<sp:SignedSupportingTokens>

Include signed tokens in the message signature and can optionally include additional message parts to sign and/or encrypt.

Integration Server supports the following nested assertions:

- [Token Assertion] (See [“Token Assertions” on page 270](#))
- <sp:AlgorithmSuite>
- <sp:SignedParts>
- <sp:SignedElements>
- <sp:EncryptedParts>
- <sp:EncryptedElements>

<sp:Endorsing SupportingTokens>

Sign the message signature and can optionally include additional message parts to sign and/or encrypt.

Integration Server supports the following nested assertions:

- [Token Assertion] (See [“Token Assertions” on page 270](#))
- <sp:AlgorithmSuite>
- <sp:SignedParts>
- <sp:SignedElements>

Supporting Token Assertion**Add to a WS-Policy to...**

`<sp:SignedEndorsing
SupportingTokens>`

- `<sp:EncryptedParts>`
- `<sp:EncryptedElements>`

Sign the token used for the message signature and are also signed by the signed endorsing token and can optionally include additional message parts to sign and/or encrypt.

Integration Server supports the following nested assertions:

- [Token Assertion] (See [“Token Assertions” on page 270](#))
- `<sp:AlgorithmSuite>`
- `<sp:SignedParts>`
- `<sp:SignedElements>`
- `<sp:EncryptedParts>`
- `<sp:EncryptedElements>`

`<sp:SignedEncrypted
SupportingTokens>`

Include supporting tokens in the security header that are also encrypted when they appear in the security header. Make sure that the tokens are encrypted in order to guarantee token confidentiality.

Integration Server supports the following nested assertions:

- [Token Assertion] (See [“Token Assertions” on page 270](#))
- `<sp:AlgorithmSuite>`
- `<sp:SignedParts>`
- `<sp:SignedElements>`
- `<sp:EncryptedParts>`
- `<sp:EncryptedElements>`

`<sp:Encrypted SupportingTokens>`

Include supporting tokens in the security header that are also encrypted when they appear in the security header.

Note:
WS-SecurityPolicy 1.2 only.

Supporting Token Assertion**Add to a WS-Policy to...**

The `sp:EncryptedSupportingTokens` element should be used only when you cannot provide the “message signature”. Make sure that the tokens are encrypted in order to guarantee token integrity and confidentiality.

Integration Server supports the following nested assertions:

- [Token Assertion](See [“Token Assertions” on page 270](#))
- `<sp:AlgorithmSuite>`
- `<sp:SignedParts>`
- `<sp:SignedElements>`
- `<sp:EncryptedParts>`
- `<sp:EncryptedElements>`

`<sp:EndorsingEncryptedSupportingTokens>`

Include endorsing supporting tokens that are also encrypted when they appear in the security header. Make sure that the tokens are encrypted in order to guarantee token confidentiality.

Note:
WS-SecurityPolicy 1.2 only.

Integration Server supports the following nested assertions:

- [Token Assertion](See [“Token Assertions” on page 270](#))
- `<sp:AlgorithmSuite>`
- `<sp:SignedParts>`
- `<sp:SignedElements>`
- `<sp:EncryptedParts>`
- `<sp:EncryptedElements>`

`<sp:SignedEndorsingEncryptedSupportingTokens>`

Include signed, endorsing supporting tokens that are also encrypted when they appear in the security header. Make sure that the tokens are signed and encrypted in order to guarantee token confidentiality.

Supporting Token Assertion**Add to a WS-Policy to...**

Integration Server supports the following nested assertions:

- [Token Assertion](See [“Token Assertions” on page 270](#))
- <sp:AlgorithmSuite>
- <sp:SignedParts>
- <sp:SignedElements>
- <sp:EncryptedParts>
- <sp:EncryptedElements>

WSS: SOAP Message Security Options

Use WSS: SOAP message security options to indicate whether the initiator and recipient must be able to process a given reference mechanism or whether the initiator and recipient can send a fault when such references are encountered.

Important:

If you are implementing WS-SecurityPolicy 1.2 standards, the `sp` prefix in the assertions described below represents this namespace: <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702>. If you are implementing WS-SecurityPolicy 1.1 standards, the `sp` prefix represents this namespace: <http://schemas.xmlsoap.org/ws/2005/07/securitypolicy>.

The following table lists the WSS: SOAP message security options that Integration Server supports.

WSS: SOAP Message Security Option	Add to a WS-Policy to...
-----------------------------------	--------------------------

<sp:Wss10>	Specify the WSS: SOAP Message Security 1.0 options that the WS-Policy supports.
------------	---------------------------------------------------------------------------------

Integration Server supports the following nested assertions:

- <sp:MustSupportRefKeyIdentifier>
- <sp:MustSupportRefIssuerSerial>

<sp:Wss11>	Specify the WSS: SOAP Message Security 1.1 options that the WS-Policy supports.
------------	---------------------------------------------------------------------------------

Integration Server supports the following nested assertions:

- <sp:MustSupportRefKeyIdentifier>
- <sp:MustSupportRefIssuerSerial>

WSS: SOAP Message Add to a WS-Policy to... Security Option

- <sp:MustSupportRefThumbprint>
- <sp:RequireSignatureConfirmation>

Policies Based on WS-SecurityPolicy that Integration Server Provides

Integration Server provides pre-defined WS-Policies based on WS-SecurityPolicy. These policies contain settings for a number of standard security configurations. You can attach WS-Policies at the binding operation message type level, such as input, output, and fault, in consumer and provider web service descriptors.

Note:

When attaching pre-defined WS-Policies, ensure that you attach the policies at the appropriate binding operation message type levels.

In the policies, WS-SecurityPolicy assertions for authentication apply only to request messages, that is, to:

- Consumer outbound request messages
- Provider inbound request messages

WS-SecurityPolicy assertions for message integrity and confidentiality apply to all request and response messages.

The out-of-the-box policies are in the following directory:

Software AG_directory \IntegrationServer\instances\instance_name\config\wss\policies

You can use these policies as is, or you can use them as templates when creating your own custom policies.

All of the out-of-the-box policies include a Timestamp token to guard against replay attacks. The following table provides a quick glance at the other security options that each policy provides. Each policy is described in detail in the sections that follow the table.

Policy	Authentication	SOAP Body Signature	SOAP Body Encryption
"Username_Over_Transport" on page 282	Username		
"Username_Signature" on page 283	Username	X	
"Username_Encryption" on page 285	Username		X
"Username_Signature_Encryption" on page 287	Username	X	X

Policy	Authentication	SOAP Body Signature	SOAP Body Encryption
"X509Authentication" on page 289	X.509 certificates		
"X509Authentication_Signature" on page 290	X.509 certificates	X	
"X509Authentication_Encryption" on page 292	X.509 certificates		X
"X509Authentication_Signature_Encryption" on page 294	X.509 certificates	X	X
"SAMLAuthentication" on page 296	SAML		
"SAMLAuthentication_Signature" on page 296	SAML	X	
"SAMLAuthentication_Encryption.policy" on page 297	SAML		X
"SAMLAuthentication_Signature_Encryption.policy" on page 298	SAML	X	X
"KerberosAuthentication Policy" on page 300	Kerberos		

Username_Over_Transport

The Username_Over_Transport policy uses a Username token to provide client authentication with Transport binding and includes a Timestamp token to guard against replay attacks. The entire message is secured by the HTTPS transport protocol. This policy does not enforce signatures or encryption.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
Consumer web service descriptor	outbound request	<ul style="list-style-type: none">■ Adds a Username token to the security header. Integration Server uses the user name provided on the endpoint alias or the one passed into the connector. For more information, see "Web Service Consumer: Request (Outbound Security) Detailed Usage and Resolution Order" on page 248.■ Adds a Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using watt.server.ws.security server configuration

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
		parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i> .
	inbound response	<ul style="list-style-type: none"> Requires a signed Timestamp token, which Integration Server validates to ensure against replay attacks.
Provider web service descriptor	inbound request	<ul style="list-style-type: none"> Requires a Username token in the security header. Integration Server authenticates the sender of the inbound request messages using the user name supplied in Username token. Requires a signed Timestamp token in the security header, which Integration Server validates to ensure against replay attacks.
	outbound response	Adds a Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i> .

Username_Signature

The Username_Signature policy uses a Username token to provide client authentication, uses symmetric binding to sign messages to ensure message integrity, and includes a Timestamp token to guard against replay attacks. Because this policy uses symmetric binding, the sender of an outbound message does not need a private key. Instead, the client generates a symmetric key.

This policy does not enforce encryption.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
Consumer web service descriptor	outbound request	<ul style="list-style-type: none"> Adds a signed Username token to the security header. Integration Server uses the user name provided on the endpoint alias or the one passed into the connector. Integration Server uses the symmetric key to sign the Username token. Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or

When the policy is attached to: **Message type** **To enforce the policy, Integration Server...**

by using `watt.server.ws.security` server configuration parameters. For more information, see *webMethods Integration Server Administrator's Guide*. Integration Server uses the symmetric key to sign the Timestamp token.

- Signs the SOAP body of the outbound request message using the symmetric key.
- Encrypts the symmetric key that Integration Server generated and adds it to the security header.

For details about how Integration Server determines the user name to use in the Username token and the server certificate to use for encrypting the symmetric key, see [“Web Service Consumer: Request \(Outbound Security\) Detailed Usage and Resolution Order” on page 248](#).

inbound
response

- Requires a signed Timestamp token, which Integration Server validates to ensure against replay attacks.
- Requires that the SOAP body of the inbound response be signed and verifies the signature using the generated symmetric key.

**Provider web
service descriptor**

inbound request

- Requires the symmetric key that the client generated be in the security header. The client encrypts the symmetric key using the Integration Server public key before adding it to the header.
- Requires a Username token in the security header. Integration Server authenticates the sender of the inbound request messages using the user name supplied in Username token.
- Requires a signed Timestamp token in the security header, which Integration Server validates to ensure against replay attacks.
- Requires that the SOAP body of the inbound request be signed using the symmetric key. Integration Server decrypts the symmetric key in the security header using its private key and uses the symmetric key to verify the signature. For the resolution order that Integration Server uses to

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
		determine the private key it uses to decrypt the symmetric key, see “Web Service Provider: Request (Inbound Security) Detailed Usage and Resolution Order” on page 257.
	outbound response	<ul style="list-style-type: none"> ■ Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i>. Integration Server uses the symmetric key to sign the Timestamp token. ■ Signs the SOAP body of the outbound response message using the symmetric key that the client generated.

Username_Encryption

The Username_Encryption policy uses a Username token to provide client authentication, uses symmetric binding to encrypt messages to ensure message confidentiality, and includes a Timestamp token to guard against replay attacks. Because this policy uses symmetric binding, the sender of an outbound message does not need a private key. Instead, the client generates a symmetric key.

This policy does not enforce signatures.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
Consumer web service descriptor	outbound request	<ul style="list-style-type: none"> ■ Adds an encrypted Username token to the security header. Integration Server uses the user name provided on the endpoint alias or the one passed into the connector. Integration Server uses the symmetric key to encrypt the Username token. ■ Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's</i>

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
		<p><i>Guide.</i> Integration Server signs the Timestamp token using the symmetric key.</p> <ul style="list-style-type: none">■ Encrypts the SOAP body of the outbound request message using the symmetric key.■ Encrypts the symmetric key that Integration Server generated and adds it to the security header. <p>For details about how Integration Server determines the user name to use in the Username token and the server certificate to use for encrypting the symmetric key, see “Web Service Consumer: Request (Outbound Security) Detailed Usage and Resolution Order” on page 248.</p>
	inbound response	<ul style="list-style-type: none">■ Requires a signed Timestamp token, which Integration Server validates to ensure against replay attacks.■ Requires that the SOAP body of the inbound response be encrypted and decrypts the SOAP body using the generated symmetric key.
Provider web service descriptor	inbound request	<ul style="list-style-type: none">■ Requires the symmetric key that the client generated be in the security header. The client encrypts the symmetric key using the Integration Server public key before adding it to the header.■ Requires a Username token in the security header. Integration Server authenticates the sender of the inbound request messages using the user name supplied in Username token.■ Requires a signed Timestamp token in the security header, which Integration Server validates to ensure against replay attacks.■ Requires that the SOAP body of the inbound request be encrypted using the symmetric key. Integration Server decrypts the symmetric key in the security header using its private key and uses the symmetric key to decrypt the SOAP body. For the resolution order that Integration Server uses to determine the private key it uses to decrypt the symmetric key, see “Web Service Provider:

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
		Request (Inbound Security) Detailed Usage and Resolution Order” on page 257.
	outbound response	<ul style="list-style-type: none"> ■ Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i>. Integration Server signs the Timestamp token using the symmetric key. ■ Encrypts the SOAP body of the outbound response message using the symmetric key that the client generated.

Username_Signature_Encryption

The Username_Signature_Encryption policy uses a Username token to provide client authentication, uses symmetric binding to sign messages to ensure message integrity, uses symmetric binding to encrypt messages to ensure message confidentiality, and includes a Timestamp token to guard against replay attacks. Because this policy uses symmetric binding, the sender of an outbound message does not need a private key. Instead, the client generates a symmetric key.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
Consumer web service descriptor	outbound request	<ul style="list-style-type: none"> ■ Adds an encrypted Username token to the security header. Integration Server uses the user name provided on the endpoint alias or the one passed into the connector. Integration Server uses the symmetric key to encrypt the Username token. ■ Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i>. Integration Server signs the Timestamp token using the symmetric key. ■ Signs the SOAP body of the outbound request message using the symmetric key.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
		<ul style="list-style-type: none">■ Encrypts the SOAP body of the outbound request message using the symmetric key.■ Server generated and adds it to the security header. <p>For details about how Integration Server determines the user name to use in the Username token and the server certificate to use for encrypting the symmetric key, see “Web Service Consumer: Request (Outbound Security) Detailed Usage and Resolution Order” on page 248.</p>
	inbound response	<ul style="list-style-type: none">■ Requires a signed Timestamp token, which Integration Server validates to ensure against replay attacks.■ Requires that the SOAP body of the inbound response be signed and verifies the signature using the generated symmetric key.■ Requires that the SOAP body of the inbound response be encrypted and decrypts the SOAP body using the generated symmetric key.
Provider web service descriptor	inbound request	<ul style="list-style-type: none">■ Requires the symmetric key that the client generated be in the security header. The client encrypts the symmetric key using the Integration Server public key before adding it to the header.■ Requires a Username token in the security header. Integration Server authenticates the sender of the inbound request messages using the user name supplied in Username token.■ Requires a signed Timestamp token in the security header, which Integration Server validates to ensure against replay attacks.■ Requires that the SOAP body of the inbound request be signed using the symmetric key. Integration Server verifies the signature using the symmetric key.■ Requires that the SOAP body of the inbound request be encrypted using the symmetric key. Integration Server decrypts the SOAP body using the symmetric key.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
		<p>To obtain the symmetric key that Integration Server uses for verifying the signature and decrypting the SOAP body, it decrypts the symmetric key in the security header using its private key. For the resolution order that Integration Server uses to determine the private key it uses to decrypt the symmetric key, see “Web Service Provider: Request (Inbound Security) Detailed Usage and Resolution Order” on page 257.</p>
	outbound response	<ul style="list-style-type: none"> ■ Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i>. Integration Server signs the Timestamp token using the symmetric key. ■ Signs the SOAP body of the outbound response message using the symmetric key that the client generated. ■ Encrypts the SOAP body of the outbound response message using the symmetric key that the client generated.

X509Authentication

The X509Authentication policy uses X.509 certificates to provide client authentication and includes a Timestamp token to guard against replay attacks. This policy does not enforce signatures or encryption.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
Consumer web service descriptor	outbound request	<ul style="list-style-type: none"> ■ Adds an X509 token to the security header. For the resolution order that Integration Server uses to determine the certificate to use, see “Web Service Consumer: Request (Outbound Security) Detailed Usage and Resolution Order” on page 248. ■ Adds a signed Timestamp token to the security header. Integration Server determines the

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
Provider web service descriptor		timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i> . Integration Server signs the Timestamp token using the its private key.
	inbound response	■ Requires a signed Timestamp token, which Integration Server validates to ensure against replay attacks.
	inbound request	■ Requires an X509 token in the security header. Integration Server authenticates the sender of the inbound request using the X.509 certificate from the security header of the inbound request.
	outbound response	■ Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i> . Integration Server signs the Timestamp token using the its private key.

X509Authentication_Signature

The X509Authentication_Signature policy uses X.509 certificates to provide client authentication, uses asymmetric binding to sign messages to ensure message integrity, and includes a Timestamp token to guard against replay attacks. This policy does not enforce encryption.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
Consumer web service descriptor	outbound request	■ Adds an X509 token to the security header.
		■ Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's</i>

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
		<p><i>Guide.</i> Integration Server signs the Timestamp token using the its private key.</p> <ul style="list-style-type: none"> ■ Signs the SOAP body of the outbound request message using its private key. <p>For details about how Integration Server determines the certificate to use in the X509 token and the private key to use for signing, see “Web Service Consumer: Request (Outbound Security) Detailed Usage and Resolution Order” on page 248.</p>
	inbound response	<ul style="list-style-type: none"> ■ Requires a signed Timestamp token, which Integration Server validates to ensure against replay attacks. ■ Requires that the SOAP body of the inbound response be signed and verifies the signature. For the resolution order that Integration Server uses to determine the certificate it uses for verification, see “Web Service Consumer: Response (Inbound Security) Detailed Usage and Resolution Order” on page 247.
Provider web service descriptor	inbound request	<ul style="list-style-type: none"> ■ Requires an X509 token in the security header. Integration Server authenticates the sender of the inbound request using the X.509 certificate from the security header of the inbound request. ■ Requires a signed Timestamp token in the security header, which Integration Server validates to ensure against replay attacks. ■ Requires that the SOAP body of the inbound request be signed and verifies the signature. For the resolution order that Integration Server uses to determine the certificate it uses for verification, see “Web Service Provider: Request (Inbound Security) Detailed Usage and Resolution Order” on page 257.
	outbound response	<ul style="list-style-type: none"> ■ Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's</i>

When the policy is attached to: **Message type** **To enforce the policy, Integration Server...**

Guide. Integration Server signs the Timestamp token using the its private key.

- Signs the SOAP body of the outbound response message using its private key. For the resolution order that Integration Server uses to determine the private key it uses for signing, see [“Web Service Provider: Response \(Outbound Security\) Detailed Usage and Resolution Order” on page 252.](#)

X509Authentication_Encryption

The X509Authentication_Encryption policy uses X509 certificates to provide client authentication, uses asymmetric binding to encrypt messages to ensure message confidentiality, and includes a Timestamp token to guard against replay attacks. This policy does not enforce signatures.

When the policy is attached to: **Message type** **To enforce the policy, Integration Server...**

Consumer web service descriptor outbound request

- Adds an X509 token to the security header.
- Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the **WS Security Properties** of the endpoint alias or by using `watt.server.ws.security` server configuration parameters. For more information, see *webMethods Integration Server Administrator's Guide*. Integration Server signs the Timestamp token using the its private key.
- Encrypts the SOAP body of the outbound request message using the server's certificate.

For details about how Integration Server determines the certificate to use in the X509 token and the certificate to use for encrypting, see [“Web Service Provider: Response \(Outbound Security\) Detailed Usage and Resolution Order” on page 252.](#)

inbound response

- Requires a signed Timestamp token, which Integration Server validates to ensure against replay attacks.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
Provider web service descriptor	inbound request	<ul style="list-style-type: none"> Requires that the SOAP body of the inbound response be encrypted and decrypts the message. For the resolution order that Integration Server uses to determine the private key it uses for decryption, see “Web Service Consumer: Response (Inbound Security) Detailed Usage and Resolution Order” on page 247.
		<ul style="list-style-type: none"> Requires an X509 token in the security header. Integration Server authenticates the sender of the inbound request using the X.509 certificate from the security header of the inbound request.
		<ul style="list-style-type: none"> Requires a signed Timestamp token in the security header, which Integration Server validates to ensure against replay attacks.
	outbound response	<ul style="list-style-type: none"> Requires that the SOAP body of the inbound request be encrypted and decrypts the SOAP body. For the resolution order that Integration Server uses to determine the private key it uses for decryption, see “Web Service Provider: Request (Inbound Security) Detailed Usage and Resolution Order” on page 257. Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i>. Integration Server signs the Timestamp token using the its private key. Encrypts the SOAP body of the outbound response message using the server's certificate. For the resolution order that Integration Server uses to determine the certificate it uses for encryption, see “Web Service Provider: Response (Outbound Security) Detailed Usage and Resolution Order” on page 252.

X509Authentication_Signature_Encryption

The X509Authentication_Signature_Encryption policy uses X509 certificates to provide client authentication, uses asymmetric binding to sign messages to ensure message integrity, uses asymmetric binding to encrypt messages to ensure message confidentiality, and includes a Timestamp token to guard against replay attacks.

When the policy is attached to: **Message type** **To enforce the policy, Integration Server...**

Consumer web service descriptor outbound request

- Adds an X509 token to the security header.
- Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the **WS Security Properties** of the endpoint alias or by using `watt.server.ws.security` server configuration parameters. For more information, see *webMethods Integration Server Administrator's Guide*. Integration Server signs the Timestamp token using the its private key.
- Signs the SOAP body of the outbound request message its private key.
- Encrypts the SOAP body of the outbound request message using the server's certificate.

For details about how Integration Server determines the certificate to use in the X509 token, the private key to use for signing, and the certificate to use for encrypting, see [“Web Service Consumer: Request \(Outbound Security\) Detailed Usage and Resolution Order” on page 248](#).

inbound response

- Requires a signed Timestamp token, which Integration Server validates to ensure against replay attacks.
- Requires that the SOAP body of the inbound response be signed and verifies the signature.
- Requires that the SOAP body of the inbound response be encrypted and decrypts the message.

For details about how Integration Server determines the certificate to use for verification and the private key to use for decryption, see [“Web Service Consumer: Response \(Inbound Security\) Detailed Usage and Resolution Order” on page 247](#).

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
Provider web service descriptor	inbound request	<ul style="list-style-type: none"> ■ Requires an X509 token in the security header. Integration Server authenticates the sender of the inbound request using the X.509 certificate from the security header of the inbound request. ■ Requires a signed Timestamp token in the security header, which Integration Server validates to ensure against replay attacks. ■ Requires that the SOAP body of the inbound request message be signed and verifies the signature. ■ Requires that the SOAP body of the inbound request message be encrypted and decrypts the SOAP body. <p>For details about how Integration Server determines the certificate to use for verification and the private key to use for decryption, see “Web Service Provider: Request (Inbound Security) Detailed Usage and Resolution Order” on page 257.</p>
	outbound response	<ul style="list-style-type: none"> ■ Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i>. Integration Server signs the Timestamp token using the its private key. ■ Signs the SOAP body of the outbound response message using its private key. ■ Encrypts the SOAP body of the outbound response message using the server's certificate. <p>For details about how Integration Server determines the private key to use for signing and the certificate to use for encryption, see “Web Service Provider: Response (Outbound Security) Detailed Usage and Resolution Order” on page 252.</p>

SAMLAuthentication

The SAMLAuthentication policy uses a SAML token to provide client authentication and includes a Timestamp token to guard against replay attacks. This policy does not enforce signatures or encryption.

Important:

Before you can use this policy, you must edit the policy file in the *Software AG_directory* \IntegrationServer\instances*instance_name*\config\wss\policies directory and fill in the address of Secure Token Service (STS).

Note:

The SAMLAuthentication policy is intended for only provider web service descriptors.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
---------------------------------	--------------	----------------------------------------------

Provider web service descriptor	inbound request	Requires a SAML token in the security header. Integration Server authenticates the sender of the inbound request messages using the client certificate from the SAML token.
	outbound response	Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using watt.server.ws.security server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i> . Integration Server signs the Timestamp token using its private key.

SAMLAuthentication_Signature

The SAMLAuthentication_Signature policy uses a SAML token to provide client authentication, uses asymmetric binding to sign messages to ensure message integrity, and includes a Timestamp token to guard against replay attacks. This policy does not enforce encryption.

Important:

Before you can use this policy, you must edit the policy file in the *Software AG_directory* \IntegrationServer\instances*instance_name*\config\wss\policies directory and fill in the address of Secure Token Service (STS).

Note:

The SAMLAuthentication_Signature policy is intended for only provider web service descriptors.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
Provider web service descriptor	inbound request	<ul style="list-style-type: none"> ■ Requires a SAML token in the security header. Integration Server authenticates the sender of the inbound request messages using the client certificate from the SAML token. ■ Requires a signed Timestamp token in the security header, which Integration Server validates to ensure against replay attacks. ■ Requires that the SOAP body of the inbound request be signed and verifies the signature. For the resolution order that Integration Server uses to determine the certificate it uses for verification, see “Web Service Provider: Request (Inbound Security) Detailed Usage and Resolution Order” on page 257.
	outbound response	<ul style="list-style-type: none"> ■ Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the WS Security Properties of the endpoint alias or by using <code>watt.server.ws.security</code> server configuration parameters. For more information, see <i>webMethods Integration Server Administrator's Guide</i>. Integration Server signs the Timestamp token using its private key. ■ Signs the SOAP body of the outbound response message using its private key. For the resolution order that Integration Server uses to determine the private key it uses for signing, see “Web Service Provider: Response (Outbound Security) Detailed Usage and Resolution Order” on page 252.

SAMLAuthentication_Encryption.policy

The SAMLAuthentication_Encryption policy uses a SAML token to provide client authentication, uses asymmetric binding to encrypt messages to ensure message confidentiality, and includes a Timestamp token to guard against replay attacks. This policy does not enforce signatures.

Important:

Before you can use this policy, you must edit the policy file in the *Software AG_directory \IntegrationServer\instances\instance_name\config\wss\policies* directory and fill in the address of Secure Token Service (STS).

Note:

The SAMLAuthentication_Encryption policy is intended for only provider web service descriptors.

When the policy is attached to: **Message type** **To enforce the policy, Integration Server...**

Provider web service descriptor

inbound request

- Requires a SAML token in the security header. Integration Server authenticates the sender of the inbound request messages using the client certificate from the SAML token.

- Requires a signed Timestamp token in the security header, which Integration Server validates to ensure against replay attacks.

- Requires that the SOAP body of the inbound request be encrypted and decrypts the SOAP body. For the resolution order that Integration Server uses to determine the private key it uses for decryption, see [“Web Service Provider: Request \(Inbound Security\) Detailed Usage and Resolution Order”](#) on page 257.

outbound response

- Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the **WS Security Properties** of the endpoint alias or by using watt.server.ws.security server configuration parameters. For more information, see *webMethods Integration Server Administrator's Guide*. Integration Server signs the Timestamp token using the its private key.

- Encrypts the SOAP body of the outbound response message using the server's certificate. For the resolution order that Integration Server uses to determine the certificate it uses for encryption, see [“Web Service Provider: Response \(Outbound Security\) Detailed Usage and Resolution Order”](#) on page 252.

SAMLAuthentication_Signature_Encryption.policy

The SAMLAuthentication_Signature_Encryption policy uses a SAML token to provide client authentication, uses asymmetric binding to sign messages to ensure message integrity, uses asymmetric binding to encrypt messages to ensure message confidentiality, and includes a Timestamp token to guard against replay attacks.

Important:

Before you can use this policy, you must edit the policy file in the *Software AG_directory* \IntegrationServer\instances*instance_name*\config\wss\policies directory and fill in the address of Secure Token Service (STS).

Note:

The SAMLAuthentication_Signature_Encryption policy is intended for only provider web service descriptors.

When the policy is attached to: **Message type** **To enforce the policy, Integration Server...**

Provider web service descriptor

inbound request

- Requires a SAML token in the security header. Integration Server authenticates the sender of the inbound request messages using the client certificate from the SAML token.
- Requires a signed Timestamp token in the security header, which Integration Server validates to ensure against replay attacks.
- Requires that the SOAP body of the inbound request be signed and verifies the signature.
- Requires that the SOAP body of the inbound request be encrypted and decrypts the SOAP body.

For details about how Integration Server determines the certificate to use for verification and the private key to use for decryption, see [“Web Service Provider: Request \(Inbound Security\) Detailed Usage and Resolution Order”](#) on page 257.

outbound response

- Adds a signed Timestamp token to the security header. Integration Server determines the timestamp expiration date to specify using the **WS Security Properties** of the endpoint alias or by using watt.server.ws.security server configuration parameters. For more information, see *webMethods Integration Server Administrator's Guide*. Integration Server signs the Timestamp token using the its private key.
- Signs the SOAP body of the outbound response message using its private key.
- Encrypts the SOAP body of the outbound response message using the server's certificate.

When the policy is attached to:	Message type	To enforce the policy, Integration Server...
----------------------------------------	---------------------	-----------------------------------------------------

For details about how Integration Server determines the private key to use for signing and the certificate it uses for encryption, see [“Web Service Provider: Response \(Outbound Security\) Detailed Usage and Resolution Order”](#) on page 252.

KerberosAuthentication Policy

The KerberosAuthentication policy uses a Kerberos ticket to provide authentication and includes a Timestamp token to guard against replay attacks. This policy does not enforce signatures or encryption.

19 Securing Web Services Using the WS-Security Facility

■ About the Integration Server WS-Security Facility	302
■ Configuring the WS-Security Facility	305
■ WS-Security Facility Policy Reference	308
■ Sample Policy File	316
■ Policy Files Supplied with the WS-Security Facility	318

About the Integration Server WS-Security Facility

The Integration Server WS-Security facility is a message-based security implementation in which authentication information is contained in a SOAP message header that is delivered along with the message payload. The facility implements a subset of the WS-Security protection mechanisms described in the WS-Security, version 1.0 standards, such as message signing and encryption, security timestamps, and use of Username and X.509 certificate tokens.

By encapsulating security policy definitions in an XML file, the facility allows you to define different security policies and select any policy for use with one or more Integration Server-based web services configured for WS-Security.

The WS-Security facility and the ability to associate WS-Security handlers to a web service descriptors is for web service descriptors that run in pre-8.2 compatibility mode only.

Note:

The WS-Security facility is deprecated as of Integration Server 10.4 because the web services implementation with which the WS-Security facility is used is deprecated. Specifically, the web services implementation introduced in Integration Server version 7.1 is deprecated.

Usage of WS-Security Standard for WS-Security Facility

The Integration Server WS-Security facility follows the guidelines of the WS-Security, version 1.0 standards for:

- SOAP Message Security
- Username Token Profile
- X.509 Certificate Token Profile

The facility implements a subset of the message protection mechanisms for the WS-Security model described in these standards, including usage of the UsernameToken as a means of identifying a requestor and the X.509 Certificate Token authentication framework.

For more information, refer to the WS-Security standards at:

- <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>
- <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf>
- <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf>

Supported Types of Message Authentication

Integration Server's WS-Security facility lets you implement policies for several standard message-based authentication scenarios:

- **Username/password.** You can include a UsernameToken in the header of an outbound message containing the user name and password credentials. The token is authenticated by the message recipient if it is found on inbound messages.

- **X.509 Signature Authentication.** Allows the use of a private key from an X.509 standard certificate to sign a document, thus authenticating the identity of the sender to the receiver. The recipient verifies the signed messages through the matching public key.
- **Proprietary X.509 authentication.** You can include an X.509 certificate or a reference to an X.509 certificate as an authentication token in the message header, without any signing or encryption. This combination of settings supports non-standard X.509 configurations.

Because no signing or encryption is used, you may need to provide additional transport-level security such as SSL to secure the endpoints of the connection.

In addition to these standard categories of authentication, the flexibility afforded by the XML policy elements allows for a high degree of customizing. You can assemble and implement many combinations of authentication options to protect your web service, as long as the web service supports the particular option.

Message Security Options Supported by WS-Security Facility

The following table describes the principal categories of security options available with the Integration Server WS-Security facility:

Category	Description
Signature Options	<p>A signature is a means of authenticating a message so that the recipient is certain of the sender's identity and the integrity of the message content. Signing a message involves encrypting a message digest with the sender's private key. To verify a signed message, the recipient uses the public key corresponding to the sender's private key. The signature attributes that the WS-Security facility supports include the following:</p> <ul style="list-style-type: none"> ■ Allow a signature with an expired certificate ■ Require the SOAP message body to be signed ■ Authenticate the message with the signing certificate <p>The WS-Security facility does not support the following signature options:</p> <ul style="list-style-type: none"> ■ Selecting the algorithm to use in creating the message digest ■ Selective or multiple signing of an outbound message
Encryption Options	<p>The WS-Security implementation encrypts SOAP message bodies using the recipient's public key. The available encryption options that the WS-Security facility supports include the following:</p> <ul style="list-style-type: none"> ■ Select an encryption algorithm ■ Select a key wrapping algorithm

Category	Description
	<ul style="list-style-type: none">■ Require the SOAP body of inbound messages to be encrypted <p>The WS-Security facility does not support the following encryption options:</p> <ul style="list-style-type: none">■ The C14N canonicalization algorithm■ Selective or multiple encryption of an outbound message■ Encrypting outbound messages with a password
Security Timestamps	The WS-Security facility allows you to use a Timestamp element that specifies message expiration time, as well as the precision of the time measurement. This element offers protection against replay attacks, since inbound messages arriving after the expiration time can be invalidated.
Username and X.509 Certificate Tokens	<p>The WS-Security facility allows you to use either of two WS-Security standard authentication token categories for authenticating a web service:</p> <ul style="list-style-type: none">■ Username. The web services consumer supplies a UsernameToken block to identify the requestor by “username” and a password (text) to authenticate the identity to a web services producer. Generally, you should use a Timestamp element specifying message expiration with the UsernameToken.■ X509 Certificate Authentication. A binary token type that represents either a single certificate or certificate path in X.509 certificate format.

Token References

The WS-Security facility allows you to specify handling of certificate information through direct or indirect references:

- In a *direct* reference, the actual certificate, or a path to the URI specifying a remote data source containing the token element, is embedded in the SOAP header.
- In an *indirect* reference, a certificate property, such as the X.509 key identifier, is embedded in the SOAP header. Using this property value, the recipient extracts the property value from the message header and uses it to locate the certificate.

Policy Files Used by the WS-Security Facility

To use the WS-Security facility you create policy files. A policy file is equivalent to a complete XML Header component of a web service descriptor. When configuring the WS-Security facility, you map a policy file to a web service descriptor file. Each time a message is sent or received by

the web service, the authentication, signing, and encryption settings specified in the SOAP header are enabled.

Important:

The policy files that the WS-Security facility uses are *not* standard WS-Policy files. They are a unique format used only by the Integration Server WS-Security facility. Integration Server version 8.2 now provides WS-Security support using standard WS-Policies. For more information, see [“Securing Web Services Using WS-SecurityPolicy” on page 261](#).

Security options that you can specify depend the message direction, that is, either inbound or outbound. You specify the message direction using XML components in the policy file. Rules for inbound messages are specified within an <InboundSecurity> section, and rules for outbound messages are specified within an <OutboundSecurity> section. Security elements specifying username/password, signing, encryption, and all other properties, are contained within these sections. For more information and XML code examples specifying message direction, see [“InboundSecurity and OutboundSecurity Elements” on page 309](#).

For a complete listing and description of the XML components and attributes that you can use in an Integration Server WS-Security facility policy file, see [“WS-Security Facility Policy Reference” on page 308](#). A description of the authentication settings for a typical policy file is shown in [“Sample Policy File” on page 316](#).

A number of pre-defined WS-Security facility policy files supplied with Integration Server are located in the *Software AG_directory \IntegrationServer\instances\instance_name\config\policy* directory. These policy files contain the settings for a number of standard security configurations. You can use these file out of the box, or as templates for creating custom policy files. For more information, see [“Policy Files Supplied with the WS-Security Facility” on page 318](#).

Configuring the WS-Security Facility

Following is a summary of the step sequence to follow when configuring the WS-Security facility on IS web service providers and consumers:

Title	Description
“Before Configuring the WS-Security Facility” on page 306	Complete any prerequisites for configuring the WS-Security facility, including verifying the existence and location of the XML, WS-Security facility policy file.
“Assigning a WS-Security Handler to a Web Service Descriptor” on page 307	Using Designer, specify the XML, WS-Security facility policy file and assign it to a web service descriptor.
Creating a Consumer Web Service Endpoint Alias	Configure an endpoint alias for a consumer web service descriptor.
- OR -	- OR -

Title	Description
Creating a Provider Web Service Endpoint Alias	<p>Configure an endpoint alias for a provider web service descriptor.</p> <p>Note: For details about creating a consumer or provider web service endpoint alias, see the section <i>Creating an Endpoint Alias for a Provider Web Service Descriptor for Use with HTTP/S</i> in the <i>webMethods Integration Server Administrator's Guide</i>.the section <i>Creating an Endpoint Alias for a Provider Web Service Descriptor for Use with HTTP/S</i> in the <i>webMethods Integration Server Administrator's Guide</i>.</p>
Assigning a Web Service Alias to a Binder	<p>The set of properties associated with the alias must be linked to a web services descriptor.</p> <p>Note: For details about assigning a web service alias to a binder, see <i>webMethods Service Development Help</i>.</p>
“Passing Message-Level Security Information to a Web Service Connector” on page 150	<p>Note: This item is an alternate to web service endpoint alias configuration for the web service connector.</p> <p>Integration Server uses the information from the web services connector to build the WS-Security header and the SOAP message request</p>

Before Configuring the WS-Security Facility

Several prerequisites are necessary before beginning the Integration Server WS-Security facility configuration process:

- Make sure that the following applications have been started and are running:
 - The Integration Server hosting the web service for which you are configuring the WS-Security facility.
 - An instance of Designer.
- Certificate files for the web service provider or consumer must exist.

You will need to specify the locations of these certificate files during the configuration process. The certificate files contain the signed certificate (or chain of certificates), and the *trusted authority directory* contains the trusted roots of the certificate signing authority.

- Verify that the security policy you want to enforce is already specified in an XML, WS-Security facility policy file.

Integration Server provides several “out-of-the-box” WS-Security facility policy files for coverage of typical message-based security situations. In most cases, you will be able to use one of these policy files as is.

If your security needs require the creation of a custom policy file, you can do so by copying one of the supplied WS-Security facility policy files and editing the copy. Make sure to give the copied file a unique ID (see “[Sample Policy File](#)” on page 316 for more information).

- Verify that the WS-Security facility policy file contains the settings you want and is located in the proper directory. WS-Security facility policy files must be placed in the following folder on the machine hosting Integration Server:


Software AG_directory \IntegrationServer\instances\instance_name\config\policy

Assigning a WS-Security Handler to a Web Service Descriptor

To specify the WS-Security facility policy that you want to use with a web service descriptor, first assign a WS-Security handler to the web service descriptor, then associate a WS-Security facility policy with the handler.

Use the following steps to assign a WS-Security handler and associated security policy to the web service descriptor.

➤ To assign a WS-Security handler to a web service descriptor

1. In Package Navigator view, open and lock the web service descriptor for which you want to configure WS-Security.
2. In the Properties view, next to **Pre-8.2 compatibility mode**, select **True** to indicate that you want the web service descriptor to run in pre-8.2 compatibility mode.
3. In the web service descriptor editor, on the Handlers tab, use one of the following mechanisms for adding a header handler:
 - Click  on the toolbar. A list of available header handlers appears. Select **WS Security Handler**.
 - Right-click in the Handlers tab and click **Add Handler**. A dialog box appears with a list of available header handlers. Select **WS Security Handler** and click **OK**.
 - Cut or copy an existing WS-Security handler
4. Make sure the WS Security Handler is selected. In the Properties view, in the **Policy name** field, select the name of the WS-Security facility policy that you want to associate with the WS-Security handler.

Designer sets the value of the **Effective policy name** property to match the selected policy name.

Note: Designer sets the value of the effective policy name to match policyName the first time only that a policy name is specified. If you later change the policy name, Designer does not automatically update the effective policy name. If you want the effective policy name to match the new policy name, you must specify the new policy in the **Effective policy** property.

5. Click **File > Save**.

Important:

The order of handlers is important. For provider web service descriptors, it is recommended that the WS-Security handler be the first handler listed. For consumer web service descriptors, it is recommended that the WS-Security handler be the last handler listed.

WS-Security Facility Policy Reference

To use the WS-Security facility you create policy files that specify the security options you want to apply to messages.

Important:

The policy files that the WS-Security facility uses are *not* standard WS-Policy files. They are a unique format used only by the Integration Server WS-Security facility. Integration Server version 8.2 now provides WS-Security support using standard WS-Policies. For more information, see [“Securing Web Services Using WS-SecurityPolicy” on page 261](#).

The following table describes the XML elements of that you can use in a policy document for the WS-Security facility.

Element	Description
<Policy>	Specifies the policy namespace and the unique identifier of the policy.
<SecurityPolicy>	Contains the elements that describe a WS-Security policy.
<InboundSecurity> -OR- <OutboundSecurity>	Indicates whether the security and authentication information is for an inbound message or an outbound message from the web service.
<Timestamp>	Generates a timestamp that enforces message expiration.
<UsernameToken>	Includes a WS-Security UsernameToken for authentication.
<Signature>	Specifies use of a digital signature.
<Encryption>	Specifies encryption of the SOAP message body.
<X509Authentication>	Includes a WS-Security X.509 Certificate token reference for authentication.

Policy Element

The <Policy> element contains two attributes: the namespace for WS-Security facility policy files, and an identifier for the policy specification. The identifier must be specified by the policy file writer or author and must be unique.

Example

```
<Policy xmlns="http://www.webmethods.com/2007/07/policy"
      Id="Confidential File Policy A">
```

SecurityPolicy Element

The <SecurityPolicy> element contains all of the elements that specify the policy's security settings.

Example

```
<SecurityPolicy
xmlns="http://www.webmethods.com/2007/07/policy/security">
```

InboundSecurity and OutboundSecurity Elements

The differences in the authentication requirements for incoming vs. outgoing messages (message direction) or a web service are covered by specifying their properties within separate XML sections, labeled <InboundSecurity> and <OutboundSecurity>, respectively.

```
<InboundSecurity>
. . .
</InboundSecurity>

<OutboundSecurity>
. . .
</OutboundSecurity>
```

Setting a Policy Element's Usage Attribute

The "Usage" attribute applies to any policy element in an <InboundSecurity> section to explicitly indicate how the element should be treated.

Usage Value	Description
Optional	If the element is present, it will be processed. Absence of the policy element will not result in an error being generated.
Required	The element must be present or processing will fail with an error.
Rejected	The incoming message must not contain any instances of this element. If one or more instances are present, processing fails and the message will be rejected with an error.

Usage Value	Description
Ignored	Instances of this token type are not processed. Whether the element is present or absent, an error will not be generated. As an example, this setting could be used to disable the processing of UsernameToken credentials when more secure credentials (such as X.509 certificates) are being used.

Default: All policy elements are treated as “Optional”.

Example

```
<InboundSecurity>
  . . .
  <Signature
    Usage="Optional"
    . . . />
  <Encryption
    Usage="Required"
    . . . />
  . . .
</InboundSecurity>
```

Timestamp Element for Outbound Messages

The <Timestamp> element supplies settings to enforce message expiration.

For outbound messages, the presence of this element generates a timestamp with a specified “time to live” value. You can increase the precision of the value by specifying milliseconds.

The default value is 5 min (300 sec).

Example

```
<Timestamp
  TimeToLiveInSeconds="300"
  IncludeMilliseconds="True"/>
```

Timestamp Element for Inbound Messages

The <Timestamp> element supplies settings to enforce message expiration.

By default, expired messages generate an exception. However, you can use a setting to turn off message expiration.

Default: Expiration is enforced.

Example

```
<Timestamp
```



```
EnforceExpiration="False"/>
```

UsernameToken Element

For outbound messages, the <UsernameToken> element specifies whether or not to include a WS-Security UsernameToken in the message header.

Outbound Messages

Password Type

The “PasswordType” attribute specifies the password form to use. Specify one of the following settings:

Setting	Description
“Text”	Integration Server includes the password in plain or clear text.
“digest”	Integration Server creates a password digest where the password contains a hash of the timestamp.
“digestwithnonce”	Integration Server creates a password digest where the password contains a hash of the timestamp and nonce.

Note: Integration Server supports “digest” and “digestwithnonce” for consumer web service descriptor only. If you use it with a provider web service descriptor, Integration Server will process the incoming SOAP request, however, the authentication of the username will fail.

Note:

If your password contains a nonce, ensure that each message includes a new nonce value. Integration Server will reject a UsernameToken if it includes a nonce that is already used.

Example

```
<UsernameToken
  PasswordType="Text"/>
```

Signature Element for Outbound Messages

Inclusion of this element causes the facility to sign the outbound SOAP message body.

Token Reference Type

The token reference type attribute indicates how the signed certificate will be included in the message header:

Reference Type	Item Included in Header
Direct	The token itself, as a sequence of base-64-encoded bytes

Reference Type	Item Included in Header
IssuerAndSerial	The token's X.509 issuer and serial number
SubjectKeyIdentifier	The token's X.509 subject key identifier
Thumbprint	The token's thumbprint

Example

```
<Signature  
  TokenReferenceType="IssuerAndSerial"/>
```

Include Certificate Path

This parameter controls whether to send the signing certificate as a single certificate or as a certificate path (specified as “True” or “False”).

Default: False (meaning, send the signing certificate as a single certificate). Applies only when the TokenReferenceType is set to “Direct.”

Note:

Partial or multiple signing of a message, or changing the message digest algorithm, is not supported.

Example

```
<Signature  
  TokenReferenceType="Direct"  
  IncludeCertPath="True"/>
```

Signature Element for Inbound Messages

These settings indicate how to process signature information contained in the incoming SOAP header.

Allow Expired Certificates

If this attribute is set to “False,” generates an exception when a signature is encountered that was created with an invalid certificate (either expired or not yet valid). If this attribute is set to “True,” message signatures created with an expired signing certificate are allowed.

Default: False

Example

```
<Signature  
  AllowExpiredCerts="True"/>
```

Validate Signing Certificate

When set to “True,” the signing certificate will be validated to ensure that it is signed by a trusted authority.

Default: False

Example

```
<Signature
  ValidateSigningCert="True"/>
```

Authenticate with Signing Certificate

This setting specifies that the certificate used for authentication has been mapped to a valid user using Integration Server’s certificate mapping facility.

Default: True

Example

```
<Signature
  AuthenticateWithSigningCert="True"/>
```

Require Signed Body

When set to “True,” requires that the body of the SOAP message body be signed or else an exception is thrown. Signatures are still verified when this attribute is set to “False,” however, no exception is thrown if the SOAP body is not digitally signed.

Default: True

Example

```
<Signature
  RequireSignedBody="False"/>
```

Encryption Element for Outbound Messages

Inclusion of this element causes the facility to encrypt the outbound message body.

Token Reference Type

The token reference type attribute indicates how the encrypted certificate will be included in the message header.

Reference Type	Item Included in Header
Direct	The token itself, as a sequence of base-64-encoded bytes
IssuerAndSerial	The token’s X.509 Issuer and Serial Number
SubjectKeyIdentifier	The token’s X.509 Subject Key Identifier

Reference Type	Item Included in Header
Thumbprint	The token's thumbprint

Example

```
<Encryption
  TokenReferenceType="Direct"/>
```

Encryption Algorithm

This setting specifies the algorithm to use for encrypting the message. The following table lists the available algorithms.

Algorithm Name	Algorithm ID
tripledes	http://www.w3.org/2001/04/xmlenc#tripledes-cbc
aes128	http://www.w3.org/2001/04/xmlenc#aes128-cbc
aes192	http://www.w3.org/2001/04/xmlenc#aes192-cbc
aes256	http://www.w3.org/2001/04/xmlenc#aes256-cbc

Example

```
<Encryption
  EncryptionAlgorithm="aes256"/>
```

Key Wrapping Algorithm

This setting specifies the algorithm to use for encrypting keys passed in a message. The following table lists the available algorithms.

Algorithm Name	Algorithm ID
rsa15	http://www.w3.org/2001/04/xmlenc#rsa-1_5
rsaoaep	http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p

Encryption Element for Inbound Messages

These

Require Encrypted Body

When

Default

Example

```
<Encryption
  KeyWrappingAlgorithm="rsa15"/>
```

X509 Authentication Element for Outbound Messages

Inclusion of this element causes the facility to include a WS-Security X.509 token reference in the message header (without using the token to sign any part of the message).

Token Reference Type

The token reference type attribute indicates how the signing certificate will be included in the header:

Reference Type	Item Included in Header
Direct	The token itself, as a sequence of base-64-encoded bytes
IssuerAndSerial	The token's X.509 Issuer and Serial Number
SubjectKeyIdentifier	The token's X.509 Subject Key Identifier
Thumbprint	The token's thumbprint

Example

```
<X509Authentication
  TokenReferenceType="Thumbprint"/>
```

Include Certificate Path

This setting controls whether to send the signing certificate as a single certificate or as a certificate path (specified as "True" or "False").

Default: The default value of "False" (meaning, send the signing certificate as a single certificate) applies only when the TokenReferenceType is set to "Direct."

Example

```
<X509Authentication
  TokenReferenceType="Direct"
  IncludeCertPath="True"/>
```

X509 Authentication Element for Inbound Messages

These settings indicate how to process messages with a WS-Security X.509 token reference in the message header.

Allow Expired Certificates

If this attribute is set to “False,” an exception is thrown whenever a certificate is encountered that is either expired or not yet currently valid. If this attribute is set to “True,” the certificate’s expiration date is ignored.

Default: False

Example

```
<X509Authentication  
  AllowExpiredCerts="True"/>
```

Validate Certificates

Ensures that the X.509 certificate is signed by a trusted authority.

Default: False

Example

```
<X509Authentication  
  ValidateCerts="True"/>
```

Sample Policy File

The following figure shows the contents of a sample WS-Security policy file for a web service consumer. The example outlines the incoming and outgoing message blocks of the policy file, and highlights several sections of code to illustrate policy file set-up and the XML code specifying security components.

```

<Policy xmlns="http://www.webmethods.com/2007/07/policy"
  Id="Consumer policy for Username, Signature">
  <SecurityPolicy xmlns="http://www.webmethods.com/2007/01/policy/security">
    <InboundSecurity>
      <Timestamp
        EnforceExpiration="true"/>
      <Signature
        Usage="Required"
        AllowExpiredCerts="false"
        AuthenticateWithSigningCert="false"
        RequireSignedBody="true"
        ValidateSigningCert="true"/>
    </InboundSecurity>
    <OutboundSecurity>
      <Timestamp
        TimeToLiveInSeconds="300"
        IncludeMilliseconds="True"/>
      <UsernameToken
        PasswordType="Text"/>
      <Signature
        TokenReferenceType="Direct"
        IncludCertPath="true"/>
    </OutboundSecurity>
  </SecurityPolicy>
</Policy>

```

Unique policy identifier

Inbound message block

Uses a security timestamp

Accepts only signed certificates

Outbound message block

Specifies usage of Username token

- The policy ID attribute highlighted in line 1 is *required* for every policy you use.
- The policy specifies use of a WS-Security Username token. This means that a token containing the user name and password for the web service is included in the SOAP header of an outbound message to identify the requesting service.
- The policy specifies the use of a digital signature.
 - The settings for the <Signature> component in the outbound message section indicate that the certificate to use for authentication is specified by a path location contained in the message header (TokenReferenceType = "Direct", and IncludeCertPath="True").
 - The settings for the inbound message section indicate that a digital signature is required on the body of incoming messages, that messages signed by an expired certificate will not be accepted by this web service, and that signatures will be validated to make sure that they were signed by a trusted authority or CA.
- The policy also includes a security timestamp component indicating that message expiration will be enforced on incoming messages, and specifying that the expiration time of outgoing

message expiration is 300 ms. After 300 ms, messages sent from this consumer can be invalidated by the recipient.

Policy Files Supplied with the WS-Security Facility

This section lists and describes the supplied policy files located in the *Software AG_directory* \IntegrationServer\instances*instance_name*\config\policy directory. There are three sets of policy files:

- **Consumer policy files.** Policies that should be used at the consumer web service descriptor.
- **Provider policy files.** Policies that should be used at the provider web service descriptor.
- **Consumer and provider policy files.** Policies that you can use for consumers and providers. The inbound and outbound requirements are the same, therefore, you can use the policies for either direction.

Important:

When using a supplied policy file or customizing a copy of a supplied policy file, make sure to specify a unique identifier in the Policy element's ID attribute.

Consumer Policy Files

Policy Name	File Name (XML)	Description
Consumer policy for username	Username_for_consumer	Use at the consumer web service descriptor for enabling basic authentication.
Consumer policy for username, signature	Username_Sign_for_consumer	Use at the consumer web service descriptor for enabling basic authentication with digital signature.
Consumer policy for username, signature, encryption	Username_Sign_Encrypt_for_consumer	Use at the consumer web service descriptor for enabling basic authentication with digital signature and encryption.
Consumer policy for signature, authentication	Sign_Auth_for_consumer	Use at the consumer web service descriptor for enabling signature-based authentication.
Consumer policy for signature, authentication, encryption	Sign_Auth_Encrypt_for_consumer	Use at the consumer web service descriptor for enabling signature-based authentication and encryption.

Provider Policy Files

Policy Name	File Name (XML)	Description
Provider policy for username	Username_for_provider	Use at the provider web service descriptor for enabling basic authentication.
Provider policy for username, signature	Username_Sign_for_provider	Use at the provider web service descriptor for enabling basic authentication with digital signature.
Provider policy for username, signature, encryption	Username_Sign_Encrypt_for_provider	Use at the provider web service descriptor for enabling basic authentication with digital signature and encryption.
Provider policy for signature, authentication	Sign_Auth_for_provider	Use at the provider web service descriptor for enabling signature-based authentication.
Provider policy for signature, authentication, encryption	Sign_Auth_Encrypt_for_provider	Use at the provider web service descriptor for enabling signature-based authentication and encryption.

Consumer and Provider Policy Files

Policy Name	File Name (XML)	Description
Digital signature	Sign	Use to enable a digital signature.
Digital signature, encryption	Sign_Encrypt	Use to enable a digital signature and encryption.

20 Web Services Addressing (WS-Addressing)

■ About WS-Addressing in Integration Server	322
■ Applying WS-Addressing to Web Service Descriptors	324
■ WS-Addressing Behavior of Web Service Descriptors	325
■ WS-Addressing Policies Provided by Integration Server	328
■ Accessing WS-Addressing Headers of a SOAP Message	328
■ Processing Responses Asynchronously	329
■ WS-Addressing and WSDL	330
■ Generation of the WS-Addressing Headers: Resolution Order and Usage	330

About WS-Addressing in Integration Server

Use Web Services Addressing (WS-Addressing) to allow web services to communicate SOAP message addressing information in a way that is independent of the transport in use, for example, HTTP, HTTPS, or JMS.

WS-Addressing provides a standard way of providing addressing information for a SOAP message, such as the message's destination or where to reply to the message, without relying on transport-specific headers. When using WS-Addressing, web service providers and clients communicate the addressing information by adding a set of message addressing properties as headers to SOAP messages.

How WS-Addressing Works

The addressing information of SOAP messages generally depends on transport-specific headers that ensure that the message reaches the intended destination. The transport-specific headers do not contain any mechanism to allow the sender of the message to indicate that the reply should be sent to a different destination, rather than back to the sender. WS-Addressing allows you to send a reply to a different destination and not back to the sender. This is done by defining WS-Addressing headers, which contain information describing where the reply to the message should be sent.

To do this, WS-Addressing standard defines the following constructs:

- **Endpoint Reference.** Contains information that is needed to route a message to an endpoint. An endpoint reference is an XML structure, which includes the endpoint address of the message and optional reference parameters and metadata.
- **Message Addressing Properties.** Contains a set of WS-Addressing properties that conveys information, such as a unique message ID, action, and endpoint references that specifies the source and destination of the message and where the reply or fault messages are to be sent.

About Endpoint Aliases for WS-Addressing

Integration Server uses message addressing endpoint aliases to send responses to endpoints other than the one which initiated or sent the request. That is, when WS-Addressing is enabled and the request SOAP message contains a non-anonymous ReplyTo or FaultTo endpoints, Integration Server uses the message addressing endpoint alias to determine the authentication details to be used to send response to the ReplyTo and FaultTo endpoints.

When you define an endpoint alias for a provider or consumer web service descriptor, you can specify message addressing properties, which provides addressing information relating to the delivery of a message to a web service. This includes the destination address of a message or fault and the authentication credentials required to send a response to a different address than the one from which request was received.

For more information about creating a message addressing endpoint alias or specifying message addressing properties for a provider or consumer web service descriptor, see the section *Creating an Endpoint Alias for Message Addressing for Use with JMS* in the *webMethods Integration Server*

Administrator's Guide, the section *Creating an Endpoint Alias for Message Addressing for Use with JMS* in the *webMethods Integration Server Administrator's Guide*.

WS-Addressing Versions

The following table provides the WS-Addressing versions that Integration Server supports and the corresponding namespaces.

WS-Addressing version	Associated namespace
W3C final versions of the WS-Addressing core and SOAP standards	http://www.w3.org/2005/08/addressing
W3C WS-Addressing Submission standard	http://schemas.xmlsoap.org/ws/2004/08/addressing

Using WS-Addressing in Integration Server

Keep the following points in mind when using WS-Addressing:

- Integration Server supports W3C Final and W3C Member Submission WS-Addressing standard versions.
- Integration Server supports the following WS-Addressing headers:
 - `wsa:Action`
 - `wsa:To`
 - `wsa:MessageID`
 - `wsa:RelatesTo`
 - `wsa:ReplyTo`
 - `wsa:FaultTo`
 - `wsa:From`
- The namespace that the `wsa` prefix in the WS-Addressing headers represents depends on the version of the WS-Addressing standard being used.
 - **W3C Final WS-Addressing standard version:** `wsa` represents <http://www.w3.org/2005/08/addressing>
 - **W3C Member Submission WS-Addressing standard version:** `wsa` represents <http://schemas.xmlsoap.org/ws/2004/08/addressing>
- Integration Server supports both synchronous and asynchronous MEP for WS-Addressing.
- To use WS-Addressing, you attach a standard WS-Policy that includes addressing assertions to a web service descriptor.

- Integration Server supports the `wsaw:UsingAddressing` assertion, where `wsaw` refers to the namespace `"http://www.w3.org/2006/05/addressing/wsdl"`.
- You can use pre-defined WS-Policies for WS-Addressing that Integration Server provides. For more information, see ["WS-Addressing Policies Provided by Integration Server" on page 328](#).
- You can also use the pre-defined policies for WS-Addressing that Integration Server provides as a template for creating a custom WS-Addressing policy. For more information about defining your own policies, see ["WS-Policy Files" on page 232](#) and ["Guidelines for Creating WS-Policy Files" on page 233](#).
- For information about web service descriptor behavior when WS-Addressing is enforced, see ["WS-Addressing Behavior of Web Service Descriptors" on page 325](#).

Applying WS-Addressing to Web Service Descriptors

The following lists the main steps you need to complete to use WS-Addressing for a web service descriptor.

Note:

You can use WS-Addressing assertions along with other types of WS-Policies, such as WS-Security.

➤ To apply WS-Addressing to a web service descriptor

1. Determine the WS-Addressing policy you want to use.
 - **Out-of-the-box policy** that is provided with Integration Server. For more information, see ["WS-Addressing Policies Provided by Integration Server" on page 328](#).
 - **Custom WS-Addressing policies.** You can create your own WS-Addressing policy. If you want, you can use the out-of-the-box policy as a template to start your custom WS-Addressing policy. For more information about creating your own WS-Policy, see ["WS-Policy Files" on page 232](#) and ["Guidelines for Creating WS-Policy Files" on page 233](#).
2. Ensure that the WS-Addressing policy you want to use is located in the following directory:

`Software AG_directory \IntegrationServer\instances\instance_name\config\wss\policies`
3. Attach the WS-Addressing policy to the web service descriptor. For instructions, see *webMethods Service Development Help*.

Note: Integration Server supports attaching WS-Policies to a binder, specifically the input, output, and fault messages for the operations in a binder. A policy attached to a binder applies to all of the operations in the binder. Integration Server and Designer do not support applying policies to some operations in a binder and not others. Additionally, to attach a WS-Policy to a web service descriptor, the **Pre-8.2 compatibility mode** property of the web service descriptor must be set to false.

WS-Addressing Behavior of Web Service Descriptors

When a WS-Addressing policy that includes the `wsaw:UsingAddressing` assertion is attached to a web service descriptor, Integration Server behavior is based on whether the `wsdl:required` attribute of the `wsaw:UsingAddressing` assertion in the policy is set to `true` or `false`. When the `wsdl:required` attribute is set to:

wsdl:required value	Indicates...
<code>true</code>	WS-Addressing is mandatory. SOAP messages must include WS-Addressing headers.
<code>false</code>	WS-Addressing is optional. SOAP messages might or might not include WS-Addressing headers. If WS-Addressing headers are present, Integration Server honors them.

For specific information about how Integration Server behaves, see [“Behavior for Inbound Messages” on page 325](#) and [“Behavior for Outbound Messages” on page 326](#).

Behavior for Inbound Messages

The following table describes how Integration Server handles provider inbound request messages and consumer inbound response messages based on the setting of the `wsdl:required` attribute of the `wsaw:UsingAddressing` assertion in the attached policy and whether the inbound message includes WS-Addressing headers.

wsdl:required attribute is set to...	The inbound message...	Integration Server...
<code>true</code>	Contains WS-Addressing headers	<p>Processes the WS-Addressing headers in the inbound message, extracting the addressing information from the header to use for later processing.</p> <p>For an inbound request message, the provider expects:</p> <ul style="list-style-type: none"> ■ Optionally, the <code>wsa:To</code> header for the destination address. ■ The <code>wsa:Action</code> header to determine the web service operation to execute. ■ The <code>wsa:MessageID</code> header for a unique identifier for the message. Integration Server only requires this header when In-Out MEP or Robust In-Only MEP is in use.

wsdl:required attribute is set to...	The inbound message...	Integration Server...
	Does <i>not</i> contain WS-Addressing headers	Returns a fault message.
false	Contains WS-Addressing headers	Honors the WS-Addressing headers in the inbound message and behaves as it does when the <code>wsdl:required</code> attribute is set to <code>true</code> . For more information, see the description above for when <code>wsdl:required</code> attribute is set to <code>true</code> and the inbound message contains WS-Addressing headers.
	Does <i>not</i> contain WS-Addressing headers	Processes the inbound request without using WS-Addressing.

Behavior for Outbound Messages

The following table describes whether Integration Server adds WS-Addressing headers to provider outbound response messages or consumer outbound request messages based on the setting of the `wsdl:required` attribute of the `wsaw:UsingAddressing` assertion in the attached policy:

wsdl:required attribute is set to...	Integration Server...
true	<p>Includes WS-Addressing headers in the outbound message.</p> <p>For an outbound response message, the provider includes:</p> <ul style="list-style-type: none">■ A <code>wsa:Action</code> header with a value that is explicitly or implicitly associated with the corresponding WSDL definition, for example, the <code>Action</code> attribute in the WSDL document.■ A <code>wsa:MessageID</code> header with a unique identifier that is specified in the web service connector or generated by Integration Server.■ A <code>wsa:RelatesTo</code> header with the value of the <code>wsa:MessageID</code> of the associated request message.■ A <code>wsa:To</code> header with the value of the destination address.■ A <code>wsa:ReplyTo</code> header with the endpoint address for the web service to execute.■ A <code>wsa:FaultTo</code> header with the endpoint address for the web service to execute.

**wsdl:required
attribute is set
to...**

Integration Server...

- A `wsa:From` header for the source address.

For an outbound request message, the consumer includes:

- A `wsa:To` header with the endpoint address for the web service to execute.
- A `wsa:Action` header with a value that is explicitly or implicitly associated with the corresponding WSDL definition, for example, the SOAP Action or the Action attribute in the WSDL document.
- A `wsa:MessageID` header with a unique identifier that Integration Server generates.
- A `wsa:ReplyTo` header with the endpoint address for the web service to execute.
- A `wsa:FaultTo` header with the endpoint address for the web service to execute.
- A `wsa:From` header for the source address.

false

Might or might not include WS-Addressing in the outbound message.

For an outbound response message, whether the provider includes WS-Addressing headers depends on the use of WS-Addressing headers in the associated inbound request message. If the inbound request message:

- Included WS-Addressing headers, the provider includes WS-Addressing headers in the outbound response message.

The WS-Addressing headers that the provider includes will be the same as those included when the `wsdl:required` attribute is `true`. See above for a description.

- Did *not* include WS-Addressing headers, the provider does *not* include WS-Addressing headers in the outbound message.

For an outbound request message, the consumer always include WS-Addressing headers. The WS-Addressing headers that the consumer includes are the same as those included when the `wsdl:required` attribute is `true`. See above for a description. For more information, see the description above for when `wsdl:required` attribute is set to `true`.

Note:

For the resolution order of these WS-Addressing headers, see [“Generation of the WS-Addressing Headers: Resolution Order and Usage” on page 330](#).

WS-Addressing Policies Provided by Integration Server

Integration Server supports W3C WS-Addressing and Member Submission WS-Addressing standard of WS-Addressing. To support these standards of WS-Addressing, Integration Server provides two pre-defined WS-Policies:

- **Addressing.** Contains settings for adding and processing the W3C WS-Addressing headers. This standard is identified by the `http://www.w3.org/2005/08/addressing` namespace.
- **Addressing Submission.** Contains settings for adding and processing the Member Submission WS-Addressing headers. This standard is identified by the `http://schemas.xmlsoap.org/ws/2004/08/addressing` namespace.

The out-of-the-box Addressing and Addressing Submission policies contain the `wsaw:UsingAddressing` assertion that has the `wsdl:required` attribute set to `true`. As a result, WS-Addressing is enforced when you attach the policy to a web service descriptor.

You can attach the policy to both consumer and provider web service descriptors. All WS-Policies that Integration Server can use are in the following directory:

`Software AG_directory \IntegrationServer\instances\instance_name\config\wss\policies`

You can use this policy as is, or as a template when creating your own custom policies.

For a description of how Integration Server enforces the out-of-the-box Addressing policy when it is attached to a web service descriptor, refer to information in [“WS-Addressing Behavior of Web Service Descriptors” on page 325](#) for when the `wsdl:required` attribute of the `wsaw:UsingAddressing` assertion is set to `true`.

Accessing WS-Addressing Headers of a SOAP Message

If you need to access and/or update the WS-Addressing headers of a SOAP message, you can do so using the following methods:

- **Use `pub.soap.handler:getMessageAddressingProperties`.**

In a handler service, invoke the `pub.soap.handler:getMessageAddressingProperties` service to get the message addressing properties of the SOAP message in the provided message context.

For more information about the `pub.soap.handler:getMessageAddressingProperties` service, see the section `pub.soap.handler:getMessageAddressingProperties` in the *webMethods Integration Server Built-In Services Reference*. the section `pub.soap.handler:getMessageAddressingProperties` in the *webMethods Integration Server Built-In Services Reference*.

- **Use web service handlers to access, remove, or add the WS-Addressing headers.**

In a handler service, invoke the following services:

- The `pub.soap.handler:getHeaderBlock` service to access the WS-Addressing headers
- The `pub.soap.handler:removeHeaderBlock` service to remove the WS-Addressing headers
- The `pub.soap.handler:addHeaderBlock` service to add the WS-Addressing headers

Note:

To update a header, first remove it, then add the updated header.

Integration Server provides IS document types that define the structure of the WS-Addressing headers. These IS document types reside in the `pub.soap.wsa` folder in the `WmPublic` package. You can use these IS document types when specifying a value for the `documentType` input parameter of the `getHeaderBlock` and `addHeaderBlock` services.

For more information about the `pub.soap.handler:getHeaderBlock` service, the `pub.soap.handler:removeHeaderBlock` service `pub.soap.handler:addHeaderBlock` services, and the IS document types for WS-Addressing headers, see the *webMethods Integration Server Built-In Services Reference*.

Note:

These IS document types for WS-Addressing headers cannot be used for an RPC/Encoded web service descriptor.

■ Make the SOAP headers available in the pipeline.

You can instruct Integration Server to place the SOAP headers in the endpoint service or connector pipeline by enabling the provider and/or consumer web service descriptor's **Pipeline headers enabled** property. For more information about adding headers to the pipeline, see [“Including SOAP Headers in the Pipeline” on page 181](#).

Alternatively, in a service handler you can use the `pub.soap.handler:getServicePipeline` service to access the pipeline for an endpoint service or connector. Use `pub.soap.handler:getMessageAddressingProperties` or `pub.soap.handler:getHeaderBlock` services to retrieve the addressing headers, and then, put the retrieved headers into the endpoint service or connector pipeline. For more information about these services, see the *webMethods Integration Server Built-In Services Reference*.

Processing Responses Asynchronously

Consumer web service descriptors created in Integration Server version 9.0 or later can process SOAP responses asynchronously. To process responses asynchronously, you must ensure that a WS-Addressing handler is assigned to the consumer web service descriptor. You must also provide a `ReplyTo` or `FaultTo` URL that points to the consumer web service descriptor when invoking the web service connector.

When Integration Server receives a response with the endpoint URL pointing to a consumer web service descriptor, Integration Server invokes the corresponding response services that are generated when the consumer web service descriptor is created. If the SOAP response received contains a WS-Addressing action through which a response service can be resolved, Integration Server invokes that response service. If Integration Server cannot determine the specific response service for the SOAP response or if there are errors while processing the asynchronous response, the `genericFault_Response` service is invoked.

For more information about response services, see [“Working with Response Services” on page 153](#).

➤ To process responses asynchronously using WS-Addressing

1. Ensure that the WS-Addressing policy you want to use is attached to the web service descriptor.
2. Use the **Response endpoint address template** binder property of the consumer web service descriptor as the address template and replace the placeholders <server> and <port> or <topic/queue/jndi> and <destinationName> with appropriate values depending on the transport mechanism used to invoke the web service.
3. While invoking the corresponding web service connector service, specify this response endpoint address as the value for ReplyTo and/or FaultTo address in the *messageAddressingProperties* parameter of the web service connector to use this consumer web service descriptor to process responses asynchronously by invoking the callback response services.

WS-Addressing and WSDL

Keep the following information in mind when using WS-Addressing and working with WSDL.

- If the WSDL for a consumer web service descriptor or a WSDL first provider web service descriptor contains WS-Addressing policy, Integration Server will honor the policy if Integration Server supports the assertions in the policy.

Note:

If a WS-Policy file is also attached to the web service descriptor, the policy in the attached WS-Policy file overrides the policy in the WSDL.

- If the WSDL for a WSDL first provider web service descriptor contains a port type operation with input, output, or fault containing `wsaw:Action` attribute, Integration Server honors the attribute. However, if you change the **Messaging action** property of the body or fault documents of a WSDL First provider web service descriptor, the value of this property takes precedence over the value of the `wsaw:Action` attribute. For more information about **Messaging action** property, see *webMethods Service Development Help*.
- If the WSDL binding contains the `wsaw:UsingAddressing` element, consumer or WSDL first provider web service descriptors created from the WSDL will honor the WSDL. That is, WS-Addressing will be enabled for the web service descriptor.
- If you generate WSDL for a web service descriptor to which a WS-Addressing policy is attached, Integration Server generates a WS-Addressing Policy annotated WSDL.

Generation of the WS-Addressing Headers: Resolution Order and Usage

This section describes the resolution order while generating the WS-Addressing headers for consumer requests and provider response messages.

Web Service Consumer: Request (Outbound Message) Resolution Order

The following table describes the resolution order while generating the To, MessageID, ReplyTo, FaultTo, and From WS-Addressing headers for consumer request messages.

WS-Addressing Header	Resolution Order
To (address and endpoint reference parameters)	<ol style="list-style-type: none"> Passed In (Generated Web Service Connector) <i>messageAddressingProperties/to</i> Endpoint Alias Message Addressing Properties/To WSDL <div style="background-color: #f0f0f0; padding: 10px; margin-top: 10px;"> <p>Note: If the endpoint reference parameters of To endpoint reference are given as part of the connector input or endpoint alias, and if the original WSDL from which the consumer was created also contains To endpoint reference with a set of endpoint reference parameters, the resulting SOAP request will contain the merged endpoint reference parameters from the connector input or endpoint alias and the WSDL.</p> </div>
MessageID	<ol style="list-style-type: none"> Passed In (Generated Web Service Connector) <i>messageAddressingProperties/messageID</i> Unique identifier that Integration Server generates.
ReplyTo (address, endpoint reference parameters, and metadata)	<ol style="list-style-type: none"> Passed In (Generated Web Service Connector) <i>messageAddressingProperties/replyTo</i> Endpoint Alias Message Addressing Properties/ReplyTo
FaultTo (address, endpoint reference parameters, and metadata)	<ol style="list-style-type: none"> Passed In (Generated Web Service Connector) <i>messageAddressingProperties/faultTo</i> Endpoint Alias Message Addressing Properties/FaultTo

WS-Addressing Header	Resolution Order
From (address, endpoint reference parameters, and metadata)	<ol style="list-style-type: none"> Passed In (Generated Web Service Connector) <i>messageAddressingProperties/from</i> Endpoint Alias Message Addressing Properties/From

Note: Integration Server picks up the values for `MustUnderstand` and `Role` from the web service connector or from the endpoint alias. If no values are specified, the default values as per the WS-Addressing standard will be used.

Web Service Provider: Response (Outbound Message) Resolution Order

The following table describes the resolution order while generating WS-Addressing headers for service first and WSDL first provider response messages.

WS-Addressing Header	Resolution Order
To (address and endpoint reference parameters)	ReplyTo header in the request message
MessageID	Automatically generated at runtime
RelatesTo (@RelationshipType)	Automatically generated at runtime
ReplyTo (address, endpoint reference parameters, and metadata)	Message addressing endpoint alias Message Addressing Properties/ReplyTo
FaultTo (address, endpoint reference parameters, and metadata)	Message addressing endpoint alias Message Addressing Properties/FaultTo
From (address, endpoint reference parameters, and metadata)	Message addressing endpoint alias Message Addressing Properties/From

Note:

The relevant message addressing endpoint alias from which the values for these WS-Addressing headers are picked up is the endpoint alias that is mapped to the address in the response map of the provider endpoint alias. For more information about response map, see the section *Creating an Endpoint Alias for a Provider Web Service Descriptor for Use with HTTP/S* in the *webMethods Integration Server Administrator's Guide*. the section *Creating an Endpoint Alias for a Provider Web Service Descriptor for Use with HTTP/S* in the *webMethods Integration Server Administrator's Guide*.

Web Service Consumer: Request - Addressing Action Property Usage

The following table describes how Integration Server handles consumer outbound request messages based on the setting of the **Addressing action** property in the fault and body elements of the operation in a web service descriptor and the SOAP Action attributes in the consumed WSDL.

Is SOAP Action present in WSDL?	Is Messaging action property set in fault and body elements of Operation?	Behavior
Yes	Yes	If both SOAP Action and Addressing action are specified, both the values should be the same. At runtime, Integration Server uses the value of the SOAP Action from the WSDL for both SOAP Action as well as wsa:Action WS-Addressing header.
No	Yes	Integration Server uses the value of wsa:Action from the WSDL as the values for SOAP Action as well as wsa:Action WS-Addressing header.
Yes	No	Integration Server uses the value of SOAP Action from WSDL as the values for SOAP Action as well as wsa:Action WS-Addressing header.
No	No	The wsa:Action WS-Addressing header is generated at runtime based on the default action pattern for WSDL 1.1. For more information about the structure of the generated action, see http://www.w3.org/TR/ws-addr-metadata/#defactionwsdl11 or http://www.w3.org/Submission/2004/05/ws-addring-20040810/#_Toc7464327 depending on the version of the WS-Addressing standard you are using.

Service First and WSDL First Provider Web Service Descriptor: Response (Outbound Message) - wsa:Action Detailed Usage

The following table describes how Integration Server handles provider response messages based on the setting of wsa:Action in the binder.

Is <code>wsa:Action</code> present in binder?	Behavior
Yes	<p>The <code>wsa:Action</code> in the SOAP response or fault will be the action specified in the binder against Output or Fault respectively.</p> <p>In case of the WSDL First provider web service descriptor, you can change the <code>ReplyToAction</code> and <code>FaultToAction</code> values that were initially populated by the values from the WSDL.</p>
No	<p>The <code>wsa:Action</code> in the SOAP response is generated at runtime based on the default action pattern for WSDL 1.1. For more information about the structure of the generated action, see http://www.w3.org/TR/ws-addr-metadata/#defactionwsdl11 or http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/#_Toc77464327 depending on the version of the WS-Addressing standard you are using.</p>

21 Web Services Reliable Messaging (WS-ReliableMessaging)

■ About Web Services Reliable Messaging in Integration Server	336
■ Using Reliable Messaging in Integration Server	336

About Web Services Reliable Messaging in Integration Server

Use Web Services Reliable Messaging (WS-ReliableMessaging) to ensure the reliable delivery of SOAP messages between web services clients and providers. The WS-ReliableMessaging standard defines a messaging protocol to identify, track, and manage the reliable delivery of messages between the reliable messaging source and destination.

Integration Server uses the WS-ReliableMessaging protocol to reliably send SOAP messages between web service providers and clients even if the destination endpoint is temporarily unavailable or if the network connection fails. The WS-ReliableMessaging protocol defines how messages should be sent again if they are not delivered successfully, thereby ensuring that the messages are delivered to the destination endpoint and that duplicate messages are not delivered.

Using Reliable Messaging in Integration Server

To ensure that the SOAP messages are reliably delivered and to track the messages, the reliable messaging source and destination use reliable messaging sequence keys. A sequence key is a user-defined identifier to indicate the sequence to which a message belongs. A reliable messaging client associates a sequence key to a message sequence based on the endpoint URL to which the message sequence is directed.

Integration Server also uses sequence Ids that are unique identifiers used by the reliable messaging servers and clients to identify a particular reliable messaging sequence.

The following steps describe what happens during exchange of SOAP messages where reliable messaging is used:

1. The reliable messaging source sends a message or a series of messages that are transmitted across a communication link to a reliable messaging destination. Along with the messages, the reliable messaging source also sends a request to the recipient asking the recipient to acknowledge the messages.
2. When the reliable messaging destination receives the messages, it sends an acknowledgement back to the reliable messaging source either individually for each message or as a single acknowledgement for a series of messages.
3. If the messages are not delivered in the first attempt, the reliable messaging source retransmits the messages based on the reliable messaging configuration until the message is delivered and the acknowledgement is received or the sequence has timed out or is terminated.

Keep the following points in mind when you configure Integration Server to use reliable messaging:

- Integration Server supports WS-ReliableMessaging Version 1.1 only.
- The namespace prefix `wsrc` represents the URI `http://docs.oasis-open.org/ws-rx/wsrc/200702`.
- The namespace prefix `wsrcmp` represents the URI `http://docs.oasis-open.org/ws-rx/wsrcmp/200702`.
- To use WS-ReliableMessaging, you attach a standard WS-Policy, which includes reliable messaging assertions, to a web service descriptor. For more information about defining your

own policies, see [“WS-Policy Files” on page 232](#) and [“Guidelines for Creating WS-Policy Files” on page 233](#). For instructions about attaching a WS-Policy to a web service descriptor, see *webMethods Service Development Help*.

- To use WS-ReliableMessaging, you can attach a pre-defined WS-Policy named ReliableMessaging that Integration Server provides. This policy is available in the following directory:

Integration Server_directory \instances\instance_name\config\wss\policies
- Integration Server uses a unique reliable messaging sequence key to track the progress of a set of messages that are exchanged reliably between a web service consumer and provider.
- Integration Server supports only the In Order delivery assurance for reliable messaging.
- You use the Integration Server Administrator to configure reliable messaging for web services. By default, Integration Server applies the reliable messaging configuration defined on the **Settings > Web Services > Reliable Messaging > Edit Reliable Messaging Configuration** page to all web service providers and consumers. If you want to override the server-level reliable messaging configuration for a specific web service provider or consumer, define reliable messaging properties for the associated web service endpoint alias. For more information about configuring reliable messaging properties for web services and for managing reliable messaging sequences, see the section *Configuring Reliable Messaging in Integration Server* in the *webMethods Integration Server Administrator's Guide*.the section *Configuring Reliable Messaging in Integration Server* in the *webMethods Integration Server Administrator's Guide*.
- Integration Server provides pub.soap.wsrn built-in services to create and manage reliable messaging sequences. For more information about these built-in services, see *webMethods Integration Server Built-In Services Reference*.

Persistent Storage Support for Reliable Messaging Data

Integration Server provides persistent storage capability for reliable messaging transactional data. Persistent storage support ensures that the messages that are being exchanged between a reliable messaging source and a reliable messaging destination are not lost in case of system or communication failures. When messages are exchanged in distributed systems, errors can occur during the transmission of messages over communication links or during the processing of messages in system components. Under these conditions, Integration Server ensures that no messages are lost and that messages can be eventually recovered after system failure.

Integration Server provides support for persistent storage of information related to reliable messaging sequences including the essential routing and delivery information.

Keep the following points in mind while configuring Integration Server to provide persistent storage capability for reliable messaging:

- The ISInternal functional alias (specified on the **Settings > JDBC Pools** page) must be configured to point to either the embedded IS Internal database or to the external RDBMS that Integration Server must use for persistent storage.
- If the Integration Servers used for reliable message exchanges are in a clustered environment and are connected to the same database, all the Integration Servers must have the same

persistence configuration defined on the **Settings > Web Services > Reliable Messaging > Edit Reliable Messaging Configuration** page.

- Integration Server supports reliable messaging in a clustered environment only if you configure Integration Server to provide persistent storage capability for reliable messaging.
- For the authentication details to be persisted across Integration Server restarts, you must provide the authentication details for the consumer web service descriptor in the associated consumer endpoint alias and not in the associated connector signature.

Limitations When Using Reliable Messaging in Integration Server

- Integration Server supports WS-ReliableMessaging Version 1.1.
- Integration Server does not support reliable messaging over JMS.
- Because of interoperability issues with WS-Security and WS-ReliableMessaging standards, Integration Server does not support reliable messaging if the attached reliable messaging policy contains both WS-ReliableMessaging and WS-Security policy assertions.
- Integration Server does not support the following WS-ReliableMessaging policy assertions:
 - `wsrmp:SequenceSTR`
 - `wsrmp:SequenceTransportSecurity`

A Provided WS-SecurityPolicies vs. WS-Security Facility Policies

■ Overview	340
■ Policies that Provide Username Authentication	340
■ Policies that Provide Authentication Using X.509 Certificates	341
■ Policies that Provide SAML Authentication	341
■ Policies that Provide Signature and Encryption Without Authentication	342

Overview

Integration Server provides predefined policies based on WS-SecurityPolicy and also a set of predefined policies that you can use with the Integration Server WS-Security facility. The main differences in the policies are listed in the table below.

Policies based on WS-SecurityPolicy	WS-Security Facility Policies
<ul style="list-style-type: none">■ The policies work with the WS-SecurityPolicy 1.2 standard.■ You can attach the same policy to both consumer and provider web service descriptors.■ The policies support using SAML authentication by including a standard SAML token.■ The policies enforce signing the Timestamp tokens that are added to the security header.	<ul style="list-style-type: none">■ The policies are a proprietary format.■ A policy is specific to either a consumer web service descriptor or a provider web service descriptor.■ WS-Security facility does not support SAML authentication.■ The policies did not require that the Timestamp tokens be signed.

The sections that follow compare the out-of-the-box WS-SecurityPolicy policies with the out-of-the-box WS-Security facility policies.

Note:

The WS-Security facility is deprecated as of Integration Server 10.4 because the web services implementation with which the WS-Security facility is used is deprecated. Specifically, the web services implementation introduced in Integration Server version 7.1 is deprecated.

Policies that Provide Username Authentication

The following tables lists the out-of-the-box WS-SecurityPolicy policies and the corresponding out-of-the-box WS-Security facility policies that provide username authentication.

The main difference between the policies is that the WS-SecurityPolicy policies use symmetric binding, while the WS-Security facility policies use asymmetric binding.

Policy based on WS-SecurityPolicy	WS-Security Facility Policy
<i>No corresponding policy</i>	<ul style="list-style-type: none">■ Consumer policy for Username■ Provider policy for Username
<ul style="list-style-type: none">■ Username_Over_Transport■ Username_Signature	<i>No corresponding policy</i> <ul style="list-style-type: none">■ Consumer policy for Username, Signature

Policy based on WS-SecurityPolicy

- Username_Encryption
- Username_Signature_Encryption

WS-Security Facility Policy

- Provider policy for Username, Signature

No corresponding policy

- Consumer policy for Username, Signature, Encryption
- Provider policy for Username, Signature, Encryption

Policies that Provide Authentication Using X.509 Certificates

The following tables lists the out-of-the-box WS-SecurityPolicy policies and the corresponding out-of-the-box WS-Security facility policies that provide authentication using X.509 certificates.

Policy based on WS-Security Policy

- X509Authentication
- X509Authentication_Signature
- X509Authentication_Encryption
- X509Authentication_Signature_Encryption

WS-Security Facility Policy

No corresponding policy

- Consumer policy for Signature, Auth
- Provider policy for Signature, Auth

No corresponding policy

- Consumer policy for Signature, Auth, Encryption
- Provider policy for Signature, Auth, Encryption

Policies that Provide SAML Authentication

The WS-Security facility does not support SAML authentication. As a result, there are no out-of-the-box WS-Security facility policies that provide SAML authentication. The following lists the WS-SecurityPolicy policies that provide SAML authentication.

- SAMLAuthentication
- SAMLAuthentication_Signature
- SAMLAuthentication_Encryption
- SAMLAuthentication_Signature_Encryption

Policies that Provide Signature and Encryption Without Authentication

The WS-Security facility provides the following out-of-the-box policies that provide signatures to enforce message integrity and message encryption to enforce message confidentiality. However, these policies do no authentication.

There are no corresponding out-of-the-box WS-SecurityPolicy policies.

- Digital Signature
- Digital Signature, Encryption

B CDATA Blocks in Inbound and Outbound SOAP Messages

- Support for Preserving CDATA Tag Delimiters in Inbound SOAP Messages 344
- Support for Processing CDATA Blocks in Outbound SOAP Messages 344
- Encoding CDATA Blocks in Outbound SOAP Messages 345

Support for Preserving CDATA Tag Delimiters in Inbound SOAP Messages

If an inbound SOAP request envelope has an element that contains a CDATA text block, by default, the inbound processing removes the CDATA delimiter tags found in the SOAP request. Specifically, Integration Server removes the initial CDATA tag "<![CDATA[" and the terminating tag "]]>" from the string values passed as input to the target web service.

Integration Server includes the `watt.server.SOAP.inbound.CDATA.removeTags` server configuration parameter that you can use to control whether or not Integration Server preserves the CDATA delimiter tags found in a SOAP request.

- When set to true, during inbound processing, Integration Server removes the initial CDATA tag "<![CDATA[" and the terminating tag "]]>" from the string values passed as input to the target web service. This is the default.
- When set to false, Integration Server preserves the CDATA delimiter tags in inbound SOAP requests; the CDATA delimiter tags will remain in the text of the request and reach the target web service.

You do not need to restart Integration Server for changes to this parameter to take effect.

Support for Processing CDATA Blocks in Outbound SOAP Messages

Integration Server provides CDATA block support for processing of outbound SOAP messages only when Integration Server hosts the web service provider. For example, suppose that a service used as an operation in a web service provider returns String values containing CDATA blocks. When encoding this IData object into a SOAP message, Integration Server places the CDATA text in a separate CDATA section and does not url-encode special characters in the delimiters or text blocks.

When using CDATA blocks, consider the following:

- A CDATA block begins with `<![CDATA[` and ends with `]]>`.
- Multiple CDATA blocks can be used in a single String value.
- CDATA blocks cannot overlap or be nested.

Note:

When acting as a web service client, Integration Server does *not* provide CDATA block support for processing of outbound SOAP messages. If a String value containing the request is passed to the web service connector and the string contains CDATA, the contents of CDATA block are treated as regular text. In addition, the special characters in the delimiters and text blocks are url-encoded in the outbound SOAP request.

Encoding CDATA Blocks in Outbound SOAP Messages

Integration Server includes an option to control whether or not Integration Server encodes CDATA blocks in outbound SOAP messages. The `watt.server.SOAP.preserveCDATA` server configuration parameter specifies whether Integration Server encodes CDATA blocks in outbound messages.

- When set to true, when Integration Server encounters a CDATA in an outbound SOAP message, Integration Server will maintain it in the wire request unchanged and unencoded.
- When set to false, Integration Server treats the CDATA section as regular text resulting in the html encoding of the CDATA tag and illegal characters in the CDATA section.

The default value is true, CDATA is preserved and not encoded.

Note that encoding occurs before Integration Server places the outbound SOAP message on the wire.

Outbound SOAP messages that are affected by this setting include SOAP requests sent by web service connectors and SOAP responses sent by web service providers.

Example

When `watt.server.SOAP.preserveCDATA` is set to false, this CDATA section:

```
<![CDATA[  
< " & " >  
]]>
```

Would be encoded in the following way:

```
&lt;![CDATA[  
&lt; " & " >  
]]&gt;
```


C

Omitting Well-Known Schema Locations from Generated WSDL

Integration Server includes an option to allow the URL for a well-known XML schema definitions namespace and schemaLocation attribute to be omitted from the import statement in the WSDL document generated for a provider web service descriptor.

During web service processing, which includes creating a web service descriptor from a WSDL document and generating a WSDL document for a provider web service descriptor, Integration Server de-references XML schema definitions that contain import statements with a schemaLocation attribute that specifies an external URL. To resolve the URLs, Integration Server requires a connection to the Internet.

Integration Server provides a configuration option to indicate that, when generating a WSDL document for a provider web service descriptor, Integration Server will not include the schemaLocation attribute for well-known namespaces. When the schemaLocation attribute is not present, an Integration Server that consumes the WSDL document will not de-reference the schemaLocation attribute value during web service processing. If a schemaLocation attributes in the WSDL document reference only the well-known namespace, then web service development can occur without an Internet connection.

The well-known namespaces that are affected by this setting are:

- <http://www.w3.org/XML/1998/namespace>
- <http://www.w3.org/2003/05/soap-envelope>

To configure Integration Server to omit the addition of schemaLocation for the above namespaces, set the `watt.core.xsd.useKnownSchemaLocation` server configuration parameter to false.

Note:

WS-I compliance requires that in a WSDL document, or files referenced by a WSDL document, all XML Schema imports must contain a schemaLocation attribute. If an XML Schema in the WSDL generated for a web service descriptor contains an import statement for either of the specified namespaces, setting `watt.core.xsd.useKnownSchemaLocation` to false breaks WS-I compliance

D Preserving Namespace Declarations when Decoding xsd:any Elements

When decoding a SOAP request or response that includes an `xsd:any` element, Integration Server preserves all the namespace declarations associated with an `xsd:any` element. This includes the namespace declarations for any elements nested in the `xsd:any` element.

Integration Server preserves namespace declarations for the `xsd:any` element by inserting the following field into the document:

`@xmlns: <prefix>`

Where `<prefix>` is the prefix defined in the SOAP message. If no prefix is defined, meaning that the default namespace is being declared, the variable name will be `@xmlns`.

The value of the `@xmlns: <prefix>` variable is the namespace declaration. Integration Server adds one `@xmlns:<prefix>` field for each namespace declaration in the `xsd:any` field. Additionally, the corresponding field for an element that belongs to the namespace includes the prefix in the field name.

Preserving `xmlns` attributes for namespace qualified elements in an `xsd:any` element can result in unexpected and unwanted fields. Unlike fields that correspond to declared elements, for an `xsd:any` field that corresponds to an `xsd:any` element in the WSDL document there is not a corresponding IS document type that identifies the possible fields and namespaces that appear at run time. Because the possible elements and corresponding namespaces are not defined in a corresponding document type, Integration Server considers the namespaces from an `xsd:any` element to be undeclared.

Integration Server represents an any element from an XML Schema definition referenced by a WSDL document by indicating that the IS document type that contains the corresponding any field may have unspecified fields (**Allow unspecified fields** is set to true). However, this does not alleviate the issue that the namespace declarations in the SOAP message are not present in an IS document type.

To control whether or not Integration Server retains namespaces in an `xsd:any` element when decoding a SOAP request or SOAP response, Integration Server includes the server configuration parameter `watt.server.SOAP.retainUndeclaredNamespace`. When set to the default value of true, Integration Server preserves namespace declarations for the `xsd:any` element. Additionally, the corresponding field for an element that belongs to the namespace includes the prefix in the field name, when set to false, Integration Server does not retain the namespace declarations for an `xsd:any` element and the name of the field corresponding to an element in the declared namespace names do not include the prefix.

