BEA AquaLogic® Service Bus

User's Guide

Version 3.0
Revised: February 2008

Contents

Introduction to ALSB

Configuring Proxy Services and Business Services

Modeling Message Flow in ALSB

Improving Service Performance with Split-Join

Message Context

Using the Test Console

UDDI

Extensibility Using Java Callouts and POJOs

XQuery Implementation

XQuery-SQL Mapping Reference

Debugging ALSB

ALSB APIs

CHAPTER 1

Introduction to ALSB

ALSB is part of the BEA AquaLogic® family of Service Infrastructure Products. ALSB manages the routing and transformation of messages in an enterprise system. Combining these functions with its monitoring and administration capability, ALSB provides a unified software product for implementing and deploying your Service-Oriented Architecture (SOA).

ALSB is a configuration-based, policy-driven Enterprise Service Bus (ESB). From the ALSB Console, you can monitor your services, servers, and operational tasks. Using the Web-based ALSB Console or the Eclipse-based ALSB Plug-in for WorkSpace Studio, you configure proxy and business services, set up security, manage resources, and capture data for tracking or regulatory auditing. ALSB enables you to respond rapidly and effectively to changes in your service-oriented environment.

ALSB relies on WebLogic Server run-time facilities. It leverages WebLogic Server capabilities to deliver functionality that is highly available, scalable, and reliable.

The following sections describe the contents, audience for, and organization of this document—*AquaLogic Service Bus User Guide*.

Document Scope and Audience

This guide provides information on using and configuring ALSB. It is intended for those responsible for messaging and SOA, specifically enterprise architects, application architects and developers.

ALSB concepts, along with an architectural overview, are discussed in *AquaLogic Service Bus Concepts and Architecture.*

Information for operations specialists such as monitoring, reporting, and tracing is presented in the *AquaLogic Service Bus Operations Guide*.

Information for security architects and developers is presented in the *AquaLogic Service Bus Security Guide*.

Information for deployment specialists resides in the *AquaLogic Service Bus Deployment Guide*.

While sometimes providing procedural information, this guide does not provide detailed information on how to configure resources using the Web-based ALSB Console or the Eclipse-based ALSB Plug-in for WorkSpace Studio. For information on using the ALSB Console, see *Using the AquaLogic Service Bus Console*. For information on using the ALSB Plug-in for WorkSpace Studio, see *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*.

For information about ALSB transport providers for configuring proxy and business services based on various transport protocols, see the AquaLogic Service Bus Transports page.

Document Organization

This document includes the following sections:

- Configuring Proxy Services and Business Services: Information about creating and managing ALSB proxy services and business services.
- Modeling Message Flow in ALSB: Guidelines for modeling message flows in ALSB. A message flow defines the implementation of a proxy service. In ALSB, service clients exchange messages with an intermediary proxy service rather than directly with a business service.
- Message Context: Describes the ALSB message context model and the predefined context variables that are used in message flows.

- Using the Test Console: Describes using the test console (available in theALSB Console only) to test proxy services, business services, and some of the resources created and used in ALSB.
- UDDI: Describes using Universal Description, Discovery and Integration (UDDI) registries with ALSB. The UDDI protocol is one of the major building blocks required for successful Web Services. UDDI provides a standard interoperable platform that enables enterprises and applications to find and use Web Services over the Internet.
- Extensibility Using Java Callouts and POJOs: Provides guidelines for using the Java callout action with Plain Old Java Objects (POJOs).
- XQuery Implementation: ALSB uses the BEA AquaLogic Data Services Platform implementation. This section describes valid extensions of the AquaLogic Data Services Platform for BEA ALSB and ALSB-specific XQuery functions.
- XQuery-SQL Mapping Reference: Provides information about the native RDBMS Data Type support and XQuery mappings that the BEA XQuery engine generates or supports.
- Debugging ALSB: Describes enabling debugging in ALSB modules.
- ALSB APIs: Describes available APIs for customizing resources and for external access to monitoring data and deployment.

CHAPTER 2

Configuring Proxy Services and Business Services

ALSB proxy services and business services provide the means for managing services, transforming messages, and routing messages through the enterprise.

You can create and configure proxy services and business services using either the ALSB Console or the ALSB Plug-in for WorkSpace Studio. You can base proxy services or business services on existing WSDL resources, including those imported from a UDDI registry such as the AquaLogic Service Registry, and then further configure them in the console or the plug-in.

The following sections describe services and their configuration:

ALSB Proxy Services

ALSB proxy services are definitions of intermediary Web Services that ALSB implements and hosts locally. ALSB uses proxy services to route messages between business services (such as enterprise Web Services and databases) and service clients (such as presentation applications or other business services).

A proxy service configuration includes its interface, transport settings, security settings, and a message flow definition. The message flow definition defines the logic that determines how messages are handled as they flow through the proxy service. If a proxy service is based on a Web Services Description Language (WSDL) document, the configuration also includes a WSDL port or a WSDL binding. (See Using a WSDL to Define a Service.)

ALSB Business Services

ALSB business services are definitions of enterprise Web Services to which ALSB is a client. Those external Web Services are implemented in and hosted by external systems, so ALSB must know what to call, how to call, and what to expect as a result of a call. ALSB business services model those interfaces so that ALSB can invoke the external services.

A business service configuration includes its interface, transport settings, and security settings. (It does not include a message flow definition.) If the business service is based on a WSDL, the configuration also includes a WSDL port or a WSDL binding. (See Using a WSDL to Define a Service.)

How WSDL is Used in ALSB

ALSB defines some types of business services and proxy services using WSDL, an XML-based specification for describing Web services. A WSDL document describes service operations, input and output parameters, and how a client application connects to the service. For the WSDL 1.1 specification, see the W3C Note, W3C Web Services Description Language (WSDL) 1.1.

About Effective WSDLs and Generated WSDLs

In ALSB, you can base a new proxy service or a new business service on an existing WSDL (called a “WSDL resource”) and then override or add configuration properties in the console or the plug-in. You can also create new services that are not based on pre-existing WSDLs and then configure them completely in the console or the plug-in.

Effective WSDLs

An *effective WSDL* represents a services's WSDL properties as configured in ALSB.

When you create a service based on a WSDL resource, ALSB generates an effective WSDL by combining settings from the original WSDL plus any transport configurations you set in the console or the plug-in, plus any other configuration settings you add or change from the original WSDL. Settings from the original WSDL resource that are not used in the new configuration are omitted from the effective WSDL.

ALSB can generate effective WSDLs for these types of proxy services and business services:

- SOAP services created from a WSDL
- XML services created from a WSDL

Effective WSDLs can be generated for those types of services using any transport that supports WSDL-based services, including HTTP, JMS, SB, etc.

ALSB cannot generate effective WSDLs for these types of proxy services and business services:

- Any SOAP
- Any XML
- Messaging type services

Effective WSDLs have different characteristics for proxy services and business services and for services based on WSDL ports and services based on WSDL bindings. Those characteristics are discussed throughout this documentation. In particular, see Basing Services on WSDL Ports and on WSDL Bindings.

Generated WSDLs

A *generated WSDL* is an effective WSDL that ALSB generates for transport-type services that were not created from a WSDL resource but which can be described using a WSDL. For example, a WSDL generated from an EJB-based service is a generated WSDL.

Accessing Effective WSDLs

There are three ways to access an effective WSDL:

- In a Web browser, enter the URL for an HTTP-based proxy service, appended with `?WSDL`.

  This works only for HTTP-transport-based services for which ALSB can generate effective WSDLs.)

- In a Web browser, enter the fixed HTTP URL, for example:

  `http://host:port/sbresource?PROXY/project/folder/proxyname`

  or

  `http://host:port/sbresource?BIZ/project/folder/businessservicename`

  This works for all services for which ALSB can generate effective WSDLs.

- Export the WSDL from the console or from the plug-in. See:
  - Exporting a WSDL in *Using the AquaLogic Service Bus Console*
  - Generating an Effective WSDL in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*

  Exporting the WSDL generates a Zip file that contains the effective WSDL along with associated dependencies, including schemas and WS-Policies. ALSB evaluates the dependencies, and the appropriate location is added to the `location` attribute of the WSDL `import` element.

  There is no `import` element for WS-policies. For WS-policies, the policy reference is retained, and the policy resource is included in the export.

  You cannot export a generated WSDL.

  See also Viewing Resource Details.

WSDL Overview

A WSDL document describes a service, its location, its operations, and the way in which clients can communicate with it. This section provides a very brief introduction to WSDL, to provide context for discussing ALSB features.

Table 2-1 summarizes the main elements used to define WSDL services.

Table 2-1 High-level WSDL Elements

Element	Description
<types>	Type definitions for message content.
<message>	Abstract definition of the data being exchanged. A message consists of parts, which describe the logical, abstract content of the message.
<portType>	Abstract collection of operations supported by the service.

Table 2-1 High-level WSDL Elements

Element	Description
<operation>	Abstract description of an action supported by the service.
<binding>	Concrete protocol and data format specification for a port type.
<port>	A single endpoint, consisting of a network address and a binding.
<service>	Collection of related ports, or endpoints.

WSDL specifies SOAP, HTTP, MIME, and ALSB-specific binding extensions, which extend the WSDL binding mechanism to support protocol-specific or message format-specific features. ALSB supports SOAP,

Types

The `<types>` element is a container for data type definitions. It uses a type system, such as XML Schema (XSD), to define the vocabulary of messages handled by this service. For example, a service that provides stock quotes might define an XML vocabulary, with the terms `TradePriceRequest` and `TradePrice`, as shown in Listing 2-1.

Listing 2-1 WSDL Types Example

```
<types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/2001/XMLSchema">
        <element name="TradePriceRequest">
            <complexType>
                <all>
                    <element name="tickerSymbol" type="string"/>
                </all>
            </complexType>
        </element>
        <element name="TradePrice">
            <complexType>
                <all>
                    <element name="price" type="float"/>
                </all>
```

```
            </complexType>
        </element>
    </schema>
</types>
```

Message

The `<message>` element provides an abstract, typed definition of the data being communicated. A message consists of parts, each of which describes one logical, abstract unit of the message. A WSDL can define one or more messages, each of which may have one or more parts. For example, the WSDL fragment in Listing 2-2 defines four message types, `sellerInfoMessage`, `buyerInfoMessage`, `response`, and `negotiationMessage`, each of which has one or more parts.

Listing 2-2 WSDL Message Example

```
<message name="sellerInfoMessage">
    <part name="inventoryItem" type="xsd:string"/>
    <part name="askingPrice" type="xsd:integer"/>
</message>

<message name="buyerInfoMessage">
       <part name="item" type="xsd:string"/>
       <part name="offer" type="xsd:integer"/>
</message>

<message name="response">
    <part name="result" type="xsd:string"/>
</message>

<message name="negotiationMessage">
    <part name="item" type="xsd:string"/>
    <part name="price" type="xsd:integer"/>
    <part name="outcome" type="xsd:string"/>
</message>
```

Port Type

The <portType> element defines a set of operations supported by one or more endpoints (which are defined in the <port> element). The port type provides the public interface for the operations provided by the service. Each operation is defined in an <operation> element, each of which is an abstract description of an action supported by the service.

For example, the fragment in Listing 2-3 defines a port type with one operation, GetLastTradePrice, which can handle an input message, GetLastTradePriceInput, and an output message, GetLastTradePriceOuput. The concrete descriptions of these messages are defined in the WSDL binding, as shown in the <soap:operation> subelement in Listing 2-4.

Listing 2-3 WSDL Port Type and Operation Example

```
<portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>
```

Binding

The <binding> element specifies a concrete data format specification and a concrete transport protocol for a port type.

The fragment in Listing 2-4 specifies the binding for the StockQuotePortType port type, which is provided as the value for the type attribute. The <soap:binding> subelement signifies that the binding is bound to the SOAP protocol format. In that subelement, the style attribute specifies that the data format is SOAP document style, and the transport attribute specifies that the transport protocol is HTTP.

Listing 2-4 WSDL Binding Example

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document"
           transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
```

```
        <soap:operation
            soapAction="http://example.com/GetLastTradePrice"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
    </operation>
</binding>
```

Service and Port

The `<service>` element defines a collection of related endpoints, each of which is defined in a child `<port>` element. A port is defined as a binding associated with a network address. For example, the fragment shown in Listing 2-5 defines two ports, `StockQuotePort`, and `StockQuotePortUK`. They both use the same binding, `tns:StockQuoteSoapBinding`, (which is concretely defined in `<binding>`) but have different network addresses: `http://example.com/stockquote` vs. `http://example.uk/stockquote`. These are alternative ports available for this service.

Listing 2-5 WSDL service and port Example

```
<service name="StockQuoteService">
    <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
       <soap:address location="http://example.com:9999/stockquote"/>
    </port>
    <port name="StockQuotePortUK" binding="tns:StockQuoteSoapBinding">
       <soap:address location="http://example.uk:9999/stockquote"/>
    </port>
</service>
```

Using a WSDL to Define a Service

If a service has a well defined WSDL interface, it is recommended, although not required, that you use the WSDL to define the service.

There are three types of WSDLs you can define. They are:

- SOAP Document Wrapped Web Services
- SOAP Document Style Web Services
- SOAP RPC Web Services

SOAP Document Wrapped Web Services

A document wrapped Web Service is described in a WSDL as a Document Style Service. However, it follows some additional conventions. Standard document-oriented Web Service operations take only one parameter or message part, typically an XML document. This means that the methods that implement the operations must also have only one parameter. Document-wrapped Web Service operations, however, can take any number of parameters, although the parameter values will be wrapped into one complex data type in a SOAP message. This wrapped complex data type will be described in the WSDL as the single document for the operation.

SOAP Document Style Web Services

You can configure proxy services as SOAP style proxy services and configure business services as SOAP style business services.

Listing 2-6 provides an example of a WSDL for a sample document style Web Service using SOAP 1.1.

Listing 2-6 WSDL for a Sample Document Style Web Service

```
<definitions name="Lookup"
targetNamespace="http://example.com/lookup/service/defs"
xmlns:tns="http://example.com/lookup/service/defs"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:docs="http://example.com/lookup/docs"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xs:schema targetNamespace="http://example.com/lookup/docs"
elementFormDefault="qualified">
      <xs:element name="PurchaseOrg" type="xs:string"/>
      <xs:element name="LegacyBoolean" type="xs:boolean"/>
    </xs:schema>
```

```
  </types>
  <message name="lookupReq">
    <part name="request" element="docs:purchaseorg"/>
  </message>
  <message name="lookupResp">
    <part name="result" element="docs:legacyboolean"/>
  </message>
  <portType name="LookupPortType">
    <operation name="lookup">
      <input message="tns:lookupReq"/>
      <output message="tns:lookupResp"/>
    </operation>
  </portType>
  <binding name="LookupBinding" type="tns:lookupPortType">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="lookup">
      <soap:operation/>
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
</definitions>
```

The service has an operation (equivalent to a method in a Java class) called `lookup`. The binding indicates that this is a SOAP 1.1 document style Web Service.

When the WSDL shown in the preceding listing is used for a request, the value of the body variable (`$body`) that the document style proxy service obtains is displayed in Listing 2-7.

Note: Namespace declarations have been removed from the XML in the listings that follow for the sake of clarity.

Listing 2-7 Body Variable Value

```
<soap-env:body>
  <req:purchaseorg>BEA Systems</req:purchaseorg>
</soap-env:body>
```

In Listing 2-7, `soap-env` is the predefined SOAP 1.1 namespace and `req` is the namespace of the `PurchaseOrg` element (`http://example.com/lookup/docs`).

If the business service to which the proxy service is routing uses the above WSDL, the value for the body variable (`$body`) given above is the value of the body variable (`$body`) from the proxy service.

The value of the body variable (`$body`) for the response from the invoked business service that the proxy service receives is displayed in Listing 2-8.

Note: Namespace declarations have been removed from the XML in the listings that follow for the sake of clarity.

Listing 2-8 Body Variable Value

```
<soap-env:body>
  <req:legacyboolean>true</req:legacyboolean>
</soap-env:body>
```

This is also the value of the body variable (`$body`) for the response returned by the proxy service using this WSDL.

There are many tools available (including BEA WebLogic Workshop tools) that take the WSDL of a proxy service (obtained by adding the `?WSDL` suffix to the URL of the proxy service in the browser) and generate a Java class with the appropriate request and response parameters to invoke the operations of the service. This Java class can be used to invoke the proxy service that uses this WSDL.

SOAP RPC Web Services

You can configure proxy services as RPC style proxy services and configure business services as RPC style business services.

Listing 2-9 provides an example of a WSDL for a sample RPC style Web Service using SOAP 1.1.

Listing 2-9 WSDL for a Sample RPC Style Web Service

```
<definitions name="Lookup"
targetNamespace="http://example.com/lookup/service/defs"
xmlns:tns="http://example.com/lookup/service/defs"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:docs="http://example.com/lookup/docs"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xs:schema targetNamespace="http://example.com/lookup/docs"
elementFormDefault="qualified">
      <xs:complexType name="RequestDoc">
        <xs:sequence>
          <xs:element name="PurchaseOrg" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="ResponseDoc">
        <xs:sequence>
          <xs:element name="LegacyBoolean" type="xs:boolean"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </types>
  <message name="lookupReq">
    <part name="request" type="docs: RequestDoc"/>
  </message>
  <message name="lookupResp">
    <part name="result" type="docs: ResponseDoc"/>
  </message>
  <portType name="LookupPortType">
    <operation name="lookup">
      <input message="tns:lookupReq"/>
      <output message="tns:lookupResp"/>
    </operation>
  </portType>
  <binding name="LookupBinding" type="tns:lookupPortType">
    <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="lookup">
      <soap:operation/>
      <input>
        <soap:body use="literal"
namespace="http://example.com/lookup/service"/>
      </input>
      <output>
        <soap:body use="literal"
namespace="http://example.com/lookup/service"/>
```

```
      </output>
    </operation>
  </binding>
</definitions>
```

The service described in the preceding listing includes an operation (equivalent to a method in a Java class) called `lookup`. The binding indicates that this is a SOAP RPC Web Service. In other words, the Web Service's operation receives a set of request parameters and returns a set of response parameters. The `lookup` operation has a parameter called `request` and a return parameter called `result`. The namespace of the operation in the binding is:

```
http://example.com/lookup/service/defs
```

When the WSDL shown in Listing 2-9 is used for a request, the value of the body variable (`$body`) that the SOAP RPC proxy service obtains is displayed in Listing 2-10.

Note: Namespace declarations have been removed from the XML in the listings that follow for the sake of clarity.

Listing 2-10 Body Variable Value

```
<soap-env:body>
  <ns:lookup>
    <request>
      <req:purchaseorg>BEA Systems</req:purchaseorg>
    </request>
  </ns:lookup>
<soap-env:body>
```

Where `soap-env` is the predefined SOAP 1.1 name space, `ns` is the operation namespace (`<http://example.com/lookup/service>`) and, `req` is the namespace of the `PurchaseOrg` element (`http://example.com/lookup/docs`).

If the business service to which the proxy service routes the messages uses the WSDL shown in Listing 2-10, the value for the body variable (`$body`), shown in Listing 2-11, is the value of the body variable (`$body`) from the proxy service.

When this WSDL is used for a request, the value of the body variable (`$body`) for the response from the invoked business service that the proxy service receives is displayed in Listing 2-10.

Listing 2-11 Body Variable Value

```
<soap-env:body>
  <ns:lookupResponse>
    <result>
      <req:legacyboolean>true</req:legacyboolean>
    </result>
  </ns:lookupResponse>
<soap-env:body>
```

This is also the value of the body variable (`$body`) for the response returned by the proxy service using this WSDL.

There are many tools available (including BEA WebLogic Workshop tools) that take the WSDL of a proxy service (obtained by adding the `?WSDL` suffix to the URL of the proxy in the browser) and generate a Java class with the appropriate request and response parameters to invoke the operations of that service. You can use such Java classes to invoke the proxy services that use this WSDL.

The benefits of using a WSDL include the following:

- The system can provide metrics for each operation in a WSDL.
- Operational branching is possible in the pipeline. For more information, see "Branching in Message Flows" on page 3-5.
- For SOAP 1.1 services, the `SOAPAction` header is automatically populated for services invoked by a proxy service. For SOAP 1.2 services, the `action` parameter of the `Content-Type` header is automatically populated for services invoked by a proxy service.
- A WSDL is required for services using WS-Security. WS-Policies are attached to WSDLs.
- The system supports the `<url>?WSDL` syntax, which allows you to dynamically obtain the WSDL of a HTTP proxy service. This is useful for a number of SOAP client generation tools, including BEA WebLogic Workshop.

- In the XQuery and XPath editors and condition builders, it is easy to manipulate the body content variable (`$body`) because the editor provides a default mapping of `$body` to the request message in the WSDL of a proxy service. See "Message Context" on page 5-1.
- The run-time contents of `$body` for a specific action can be different from the default mapping displayed in the editor. This is because ALSB is not a programming language in which typed variables are declared and used. Instead, variables are untyped and are created dynamically at run time when a value is assigned. In addition, the type of the variable is the type that is implied by its contents at any point in the message flow. To enable you to easily create XQuery and XPath expressions, the design time editor allows you to map the type for a given variable by mapping the variable to the type in the editor. To learn about using the XQuery and XPath editor to create expressions, see "Working with Variable Structures" on page 3-44.

Basing Services on WSDL Ports and on WSDL Bindings

When you create a service based on a WSDL resource, you must base the service on a WSDL port or on a WSDL binding:

- When you create a new service based on a *binding* in a WSDL resource, you are choosing the protocol and data format defined in the selected `<binding>` element in the WSDL resource.
- When you create a new service based on a *port* in a WSDL resource, you are choosing the binding and the network address defined in the `<port>` element.

When creating or modifying the service, you can change the transport, but you cannot override the data format.

The port and binding definitions from the original WSDL resource are modified in the effective WSDL depending on a number of factors, as described below.

Characteristics of Effective WSDLs Generated for Proxy Services

The following characteristics apply to effective WSDLs generated for proxy services:

- The effective WSDL has one and only one `wsdl:service` section.
- The `wsdl:service` section has one and only one `wsdl:port` section.
- For SOAP services, any existing `<wsdl:service>` definition is removed, and a new service definition containing a single `<wsdl:port>` is created.

- For SOAP binding over any of the supported transports the `wsdl:binding` section contains the standard WSDL SOAP binding elements along with a unique transport URI that identifies the transport.
- For XML binding over HTTP, the `wsdl:binding` section uses the standard binding elements specified in the WSDL 1.1 specification.
- For XML binding over any of the other supported transports the `wsdl:binding` section uses BEA (ALSB) proprietary WSDL XML binding elements.
- If the service is based on a binding:
  - If the service is generated from binding Y in the WSDL resource, the effective WSDL defines a new service and port (`<bindingname>QSService` and `<bindingname>QSPort`). None of the ports defined in the WSDL resource are included in the effective WSDL.
  - There may be multiple ports in that WSDL associated with that binding. Each port can use a different URL. Therefore, the effective WSDL uses the binding but generates an artificial port from the configuration on the service for that binding. All other ports will be removed.
- If the service is based on a port:
  - If the service is generated from port X in the WSDL resource, then port X is also defined in the effective WSDL. Any other ports defined in the WSDL resource are not included. Furthermore, if you base the proxy service on a WSDL port, the effective WSDL uses that port name. The binding is determined from the port, and in turn, the port type is determined from the binding.
  - The effective WSDL preserves any WS-Policies associated with the port defined in the resource WSDL.
  - The transport address specified in the port definition in the resource WSDL is never used as the address for a proxy service in the effective WSDL:

    For HTTP services, you must specify a transport address when configuring the transport in the console or the plug-in. That address is used in the port definition in the effective WSDL.

    The URL specified as the transport address for a proxy service is always relative to a path in an ALSB domain, because ALSB always hosts proxy services.
  - For SOAP-protocol-based WSDL services, the transport URI in the SOAP binding depends on the transport implementation. For standard transports (like HTTP and JMS), this value is as per the SOAP specification or another universally accepted value. For

transports for which SOAP does not define a standard value, ALSB sets one consisting of a predefined namespace with the transport ID appended at the end: `http://www.bea.com/transport/2007/05/`.

– There is one service element in the effective WSDL, and the port address contains a URL whose syntax and semantic is defined by the transport selected in the binding.

Characteristics of Effective WSDLs Generated for Non-Transport-Type Business Services

The following characteristics apply to effective WSDLs generated for non-transport-type business services:

- The effective WSDL has one and only one `wsdl:service` section.
- The `wsdl:service section` may have more than one `wsdl:port` sections. This is generally true when load balancing is used and there are multiple end point addresses that can be used using one of the load-balance algorithms supported.
- For SOAP binding over any of the supported transports, the `wsdl:binding` section contains the standard WSDL SOAP binding elements along with a unique transport URI that identifies the transport.
- For XML binding over any of the supported transports, the `wsdl:binding` section contains the BEA WSDL XML binding elements.
- The URL specified as the transport address is a remote location where the remote service is hosted.
- If the service is based on a port:
  – The effective WSDL defines a port with the same name as the port in the template on which the new service is based. If multiple endpoint addresses are configured for the business service, the effective WSDL generated from it defines ports for all the endpoints, with the names `<portname_in_template>_1`, `<portname_in_template>_2`, `...`
  – The binding for the new service is determined from the port, and the port type is in turn determined from the binding.
  – The transport address URL is a remote location where the remote service is hosted.
- If the service is based on a binding:

- The effective WSDL defines a new service and port, based on the port in the WSDL resource. None of the ports defined in the WSDL resource are included in the effective WSDL.
- A binding in the WSDL resource may be associated with multiple ports. Each port can use a different URL and have a different WS-Policy attached to it. Therefore, the generated WSDL uses the binding but generates an artificial port for that binding with no WS-Policy.
- For XML-based WSDL services, standard HTTP binding is used only if the service uses HTTP transport. For all other services created from an XML-based WSDL, the effective WSDL uses BEA binding.

Characteristics of Effective WSDLs Generated for Transport-Type Business Services

ALSB does not guarantee one and only one `wsdl:service` section in effective WSDLs generated for transport-type business services. Because the WSDL is generated by the transport, ALSB does not generate nor does it clean up extra service-port sections. ALSB does, however, evaluate dependencies and sets their references during export.

Generating Effective WSDLs in Clustered Domains

When generating the effective WSDL in a clustered domain, ALSB rewrites the endpoint URL based on the server or cluster configuration. If a front-end HTTP host/port (or a front-end HTTPS host/port for HTTPS endpoints) has been specified, it will be used; otherwise, the Managed Server host or port will be used. It is strongly advised that a front-end HTTP or HTTPS host/port is assigned in clustered domains.

Examples of Proxy Services Based on a Port and on a Binding

Listing 2-12 shows fragments of port and binding definitions in a WSDL resource.

Listing 2-12 WSDL resource

```
<portType name="StockQuotePortType">
...
</portType>
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document"
```

```
      transport="http://schemas.xmlsoap.org/soap/http"/>
...
</binding>

<service name="StockQuoteService">
   <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
     <soap:address location="http://example.com:9999/stockquote"/>
   </port>
   <port name="StockQuotePortUK" binding="tns:StockQuoteSoapBinding">
      <soap:address location="http://example.uk:9999/stockquote"/>
   </port>
</service>
```

Basing the Service on a Port

If you create a proxy service based on the `StockQuotePort` port in Listing 2-12, the effective WSDL will look something like the fragment in Listing 2-13.

Listing 2-13 Effective WSDL for a Proxy Service Based on a Port

```
<binding name="StockQuoteSoapBinding" type="ns:StockQuotePortType">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
...
</binding>

<service name="StockQuoteService">
  <port name="StockQuotePort" binding="ns:StockQuoteSoapBinding">
    <soap:address location="http://host:port/project/proxyname"/>
  </port>
</service>
```

Notice the following about the above example:

- The service name, **`StockQuoteService`**, is the same in both the template and the effective WSDL.

- The binding is the same in both the template and the effective WSDL. In this example, it specifies that the service will use the HTTP transport protocol for SOAP document style messages. (The binding also specifies the same binding operation in both the template and the effective WSDL, but that is not shown in this example.)
- The second port defined in the WSDL resource, `StockQuotePortUK`, is not defined in the effective WSDL.
- The transport address (URI) defined in the WSDL resource's port, `http://example.com:9999/stockquote`, is different from the address generated in the effective WSDL's port, `http://host:port/project/proxyname`. The effective WSDL uses the address specified for the transport configuration in the ALSB Console or the ALSB Plug-in for WorkSpace Studio.

Basing the Service on a Binding

If you create a proxy service based on the `StockQuoteBinding` binding in Listing 2-12, the effective WSDL will look something like the fragment in Listing 2-14.

Listing 2-14 Effective WSDL for a Proxy Service Based on a Binding

```
<binding name="StockQuoteSoapBinding" type="ns:StockQuotePortType">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
...
</binding>

<wsdl:service name="StockQuoteSoapBindingQSService"
   <wsdl:port name="StockQuoteSoapBindingQSPort"
            binding="ns:StockQuoteSoapBinding">
      <soap:address location="http:/host:port/project/proxyname"/>
   </wsdl:port>
</wsdl:service>
```

Notice the following about the above example:

- The service and the port in the WSDL resource are different from the service and the port generated in the effective WSDL.

- The service name in the WSDL resource and the effective WSDL are different: **`StockQuoteService`** in the template and **`StockQuoteSoapBindingQSService`** in the effective WSDL.
- The binding is the same in both the WSDL resource and effective WSDL. In this example, it specifies that the service will use the HTTP transport protocol for SOAP document style messages.
- The binding also specifies the same binding operation in both the template and the effective WSDL, but that is not shown in this example.
- The transport address (URI) defined in the WSDL resource's port, `http://example.com:9999/stockquote`, is different from the address generated in the effective WSDL's port, `http://`*`host:port/project/`*`stockquote`.

Using Any SOAP or Any XML Service Types

If you want to expose one port to clients for a variety of enterprise applications, use **Any SOAP** or **Any XML** service types. For Any SOAP, you must specify if it is SOAP 1.1 or SOAP 1.2.

Using the Messaging Service Type

If one of the request or response messages is non-XML, you must use the messaging service type.

ALSB does not automatically perform "misunderstand" SOAP header checking. However, you can use XQuery conditional expressions and validate actions to explicitly perform this type of check. For more information on the validate action, see Adding Validate Actions in *Using the AquaLogic Service Bus Console* or Validate Action Properties in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*. For more information on conditional XQuery expressions, see Proxy Services: Editors in *Using the AquaLogic Service Bus Console* or Condition Editor in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*.

You can use ALSB to configure a validate action and use XQuery conditional expressions to perform validation checks explicitly in the message flow.

For more information on service types, see "General Configuration page" in Creating and Configuring Proxy Services in *Using the AquaLogic Service Bus Console*.

Configuring Proxy Services

The following sections provide an overview of proxy service configuration. For specific instructions for configuring proxy services, see:

- Proxy Services: Creating and Managing in *Using the AquaLogic Service Bus Console*
- Working with Proxy Services in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*

Proxy Service Types and Transports

Each proxy service is of a certain type, and each can be used with the transport protocols appropriate for that type. ALSB supports several standard transports plus custom transports. The service types are described in Table 2-2, including the standard transports available for each type.

Table 2-2 Proxy Service Types and Transports Supported by ALSB

Service Type	Description	Standard Transport Protocols Supported for this Service Type
WSDL Web Service	A SOAP or XML proxy service whose interface is described by a WSDL document.	HTTP JMS Local SB WS
Any SOAP service (no WSDL)	A SOAP service that does not have an explicitly defined, concrete interface.	HTTP JMS Local SB

Table 2-2 Proxy Service Types and Transports Supported by ALSB

Service Type	Description	Standard Transport Protocols Supported for this Service Type
Any XML service (non-SOAP, no WSDL)	An XML service that does not have an explicitly defined, concrete interface.	E-mail File FTP HTTP JMS Local MQ SB SFTP Tuxedo
Messaging service (no WSDL)	A messaging service where the request message and the response message can be of different data types, including binary, text, MFL, and XML.	E-mail File FTP HTTP JMS Local MQ Tuxedo

Note: All service types can send and receive attachments using MIME.

Transport and Security Configuration for Proxy Services

You must configure transport settings for all proxy service types in the ALSB Console or the ALSB Plug-in for WorkSpace Studio. Configuration details vary, depending on the transport type. For specific configuration settings, see:

- Proxy service Transport Configuration page and Protocol-Specific Configuration pages in *Using the AquaLogic Service Bus Console*.

- Proxy Service Transport Configuration page and Protocol-Specific Transport Configuration pages in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*.

The transport you select should support the transport mode (request/response, one-way, or both) required by the binding definition, and it should be configured accordingly.

For services exchanging messages in both modes, you must configure the binding layer to choose the transport mode accordingly (for any transport implementing the request/response as two asynchronous calls, for example, JMS). This occurs automatically when the service is a concrete type, as it is described in the binding definition. When it is not a concrete type, to configure the binding layer, you must set the mode in the $outbound variable.

Based on the transport and WSDL, the transport mode is automatically selected, but you can overwrite it in `$inbound` or `$outbound`.

You can specify security for proxy services, using service providers. A service provider is required if the proxy service routes messages to HTTPS services that require client-certificate authentication and may be required in some message-level security scenarios.

A service account can be created to provide authentication when connecting to a business service. It acts as an alias resource for the required username and password pair.

WebLogic Server can be used to directly manage security credentials for a business service requiring credential-level validation.

For more information, see Service Key Providers in *Using the AquaLogic Service Bus Console*. See also Security-Related Validation for Proxy Services.

Configuration Settings For Each Proxy Service Type

Each proxy service type is modeled following the same pattern. Each service type must define these configurations:

- Binding definition
- Run-time configuration
- Run-time variables (`$operation`, `$body`, `$header`, `$attachments`)

Configuration properties specific to individual proxy service types are described in Table 2-3:

Table 2-3 Configuration Properties for Different Proxy Service Types

Proxy Service Type	Configuration Properties
WSDL Web Service	**Binding Definition:** See Using a WSDL to Define a Service. **Run-Time Variables:** • For SOAP-based WSDL services, the variables are similar to Any SOAP service except that `$operation` is initialized based on the Operation Selection algorithm (See Operation Selection Configuration page in *Using the AquaLogic Service Bus Console*. • For XML-based WSDL services, the variables are similar to Any XML service except that the `$operation` will be initialized based on the Operation Selection algorithm.
Any SOAP service	**Binding Definition:** The only information this service type defines is that the service is receiving or sending SOAP messages—regardless of their WSDL binding definition. Therefore the binding configuration for this type is empty. In addition, as there is no binding configuration, the combination of this type and the content-type of the message is sufficient to determine whether or not there are attachments to the message. As per their definition, “any” services (SOAP or XML) do not have any WSDL definition. It is not possible to generate or view a WSDL document for those services. **Run-Time Variables:** The `$body` and `$header` variables respectively hold the `<soap:Body>` and `<soap:Header>` of the incoming SOAP message. (For SOAP 1.1 proxies, `$body` and `$header` use SOAP 1.1 namespace Body and namespace; for SOAP 1.2 proxies, they use SOAP 1.2 namespace Body and namespace.) The `$attachments` variable contains the SOAP message attachments, if any. The `$operation` variable is not applicable to this service type as you do not define a port type. To learn more about the message context variables, see Message-Related Variables and Constructing Messages to Dispatch.

Table 2-3 Configuration Properties for Different Proxy Service Types

Proxy Service Type	Configuration Properties
Any XML service	**Binding Definition:** The only information this service type defines is that the service is receiving/sending XML messages—regardless of their WSDL binding definition. Therefore, the binding configuration for this type is empty. In addition, as there is no binding configuration, the combination of this type and the content-type of the message is sufficient to determine whether or not there are attachments to the message. As per their definition, “any” services (SOAP or XML) do not have any WSDL definition. It is not possible to request a WSDL document for those services. **Run-Time Variables:** The `$body` variable holds the incoming XML message wrapped in a `<soap:Body>` element. (For SOAP 1.1 proxies, `$body` and `$header` use SOAP 1.1 namespace Body and namespace; for SOAP 1.2 proxies, they use SOAP 1.2 namespace Body and namespace.) The `$attachments` variable contains message attachments, if there are any. The `$header` variable is not applicable to this service type and is set to its default value. The `$operation` variable is not applicable to this service type as you do not define a port type. To learn more about the message context variables, see Message-Related Variables and Constructing Messages to Dispatch.

Table 2-3 Configuration Properties for Different Proxy Service Types

Proxy Service Type	Configuration Properties
Messaging service	**Binding Definition:** The binding definition for messaging services consists of configuring the content-type of the messages that are exchanged. The content-type for the response does not need to be the same as for the request; therefore, the response is configured separately (for example, the service could accept an MFL message and return an XML acknowledgment receipt) and could also be set to None. As per their definition, messaging-based services do not have any WSDL definition. It is not possible to request a WSDL document for those services. There are four available content types to choose from for the request (and response): • Binary • Text • XML • MFL **Run-Time Variables:** This service type is message-based. There is no concept of multiple "operations" as for Web Services. Therefore, the `$operation` variable is left empty. The `$body` variable holds the incoming message wrapped in a `<soap:Body>` element. (For SOAP 1.1 proxies, `$body` uses the SOAP 1.1 namespace Body; for SOAP 1.2 proxies, it uses SOAP 1.2 namespace Body.) The `$header` variable is not applicable to this service type, and is set to its default value. The `$attachments` variable contains message attachments if there are any. To learn more about the message context variables, see Message-Related Variables and Constructing Messages to Dispatch.

Configuring Message Flow

A proxy service's message flow definition defines the logic that determines how messages are handled as they flow through the proxy service. A message flow definition transforms messages, as needed, to map the message data to the format required by a business service (for outbound messages) or the originating client (for inbound messages). They then route the messages to the appropriate location. For information about configuring message flows, see Modeling Message Flow in ALSB.

Security-Related Validation for Proxy Services

When you activate a session that contains changes to an active proxy service, ALSB validates the changes to ensure that you have created all of the credentials that the proxy service's static endpoints require. For example, if you configured a proxy service to have a Web Service as a static endpoint and the Web Service requires a digital signature, ALSB verifies that you have associated a service key provider with the proxy service and that the service key provider contains a key-pair binding that can be used as a digital signature.

If a session contains a change to the key-pair bindings of a service key provider, ALSB validates the change against all of the proxy services that use the service key provider. For example, if you remove the encryption key-pair, ALSB reports a validation error for any proxy service that references the service key provider and whose endpoint requires encryption.

The following criteria determine when ALSB performs this security-related validation and the actions that it takes during validation:

- If a proxy service specifies a static route and operation, ALSB determines which credentials the static route and operation require. If the proxy service is missing the required credentials, ALSB will not commit the session until you add the missing credentials.
- If a proxy service specifies a static route but the operation is passed through from the inbound request, ALSB determines which credentials the static route and each of the route's operations require. If the proxy service is missing the required credentials, ALSB issues a validation warning but allows you to commit the session.
- If a proxy service specifies a dynamic route and operation, ALSB cannot validate the security requirements and you risk the possibility of runtime errors. For information about dynamic routing, see "Dynamic Routing" in Modeling Message Flow in ALSB.

Configuring Business Services

The following sections provide an overview of business service configuration. For specific instructions for configuring business services, see:

- Business Services in *Using the AquaLogic Service Bus Console*
- Working with Business Services in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*

Business Service Types and Transport

Each business service is of a certain type, and each can be used with the transport protocols appropriate for that type. ALSB supports several standard transports plus custom transports. The service types are described in Table 2-4, including the standard transports available for each type.

Table 2-4 Business Service Types and Transports Supported by ALSB

Service Type	Description	Transport Protocols
WSDL Web Service	A SOAP or XML business service whose interface is described by a WSDL document.	DSP HTTP JMS JPD Local[1] SB WS
Any SOAP Service (no WSDL)	A SOAP service that does not have an explicitly defined, concrete interface.	DSP HTTP JMS JPD Local SB

Table 2-4 Business Service Types and Transports Supported by ALSB (Continued)

Service Type	Description	Transport Protocols
Any XML Service (no WSDL)	An XML service that does not have an explicitly defined, concrete interface. Only HTTP GET is supported for XML with no WSDL.	DSP E-mail File FTP HTTP GET JMS JPD Local MQ SB SFTP Tuxedo[2]
Transport Typed Service (no WSDL)	A service that uses an EJB transport.	EJB Flow
Messaging Type Service (no WSDL)	A messaging service where the request message and the response message can be of different data types, including binary, text, MFL, and XML.	E-mail File FTP HTTP GET JMS Local MQ SFTP Tuxedo

1. BEA recommends using the local transport for communication between two proxy services. For more information on local transport, see the *Local Transport User's Guide*..
2. For a Tuxedo transport-based service, if the service type is XML, an FML32 buffer with an FLD_MBSTRING field from a Tuxedo client will not be transformed to XML.

For information about configuring proxy and business services based on various transport protocols, see the AquaLogic Service Bus Transports page.

Configuration Settings for All Business Service Types

Each business service type is modeled following the same pattern. Each service type must define these configurations:

- Binding definition
- Run-time configuration
- Run-time variables (`$operation`, `$body`, `$header`, `$attachments`)

The business service configuration properties described in Table 2-5, below, are common to all service types. Properties specific to individual service types are described in Configuration Settings For Each Business Service Type.

Table 2-5 Common Configuration Properties for Business Services

Property	Description
Resource Definition	The resource definition consists of: • The service name (that is, project, path, and local name) • An optional description for the service • The service type
Transport Configuration	You can configure the following parameters for each business service: • List of `<string URI, integer weight>` pairs—for example, `<http://www.bea.com, 100>`. For a random-weighted list, the list should contain at least one element. • Load-balancing algorithm—enumeration, one of round-robin, random, or random-weighted. If you select random-weighted, the weights are applicable for each URI. • Retry Count • Retry Iteration Interval • Retry Application Errors The transport you select must be able to support the transport mode (that is, request/response, one-way or both) required by the binding definition, and be configured accordingly. For services exchanging messages in both modes (i.e., request/response and one-way), you must configure the binding layer so that it can select the transport mode accordingly. This occurs automatically when the service is a concrete type, as it is described in the binding definition. When it is not a concrete type, to configure the binding layer, you must use the routing options action in the message flow to set the mode for a route or publish. Based on the transport and WSDL or interface, the transport mode is automatically selected, but you can overwrite it using the routing options action for a route or publish action.

Configuration Settings For Each Business Service Type

Configuration properties specific to individual business service types are described in Table 2-6:

Table 2-6 Configuration Properties for Different Business Service Types

Property	Description
WSDL Web Service	See Using a WSDL to Define a Service.
Any SOAP Service	**Binding Definition:** The only information this service type defines is that the service is receiving or sending SOAP messages—regardless of their WSDL binding definition. Therefore the binding configuration for this type is empty. In addition, as there is no binding configuration, the combination of this type and the content-type of the message is sufficient to determine whether or not there are attachments to the message. As per their definition, any services (SOAP or XML) do not have any WSDL definition. **Run-Time Variables:** The `$body` and `$header` variables respectively hold the `<soap:Body>` and `<soap:Header>` of the SOAP message to the business service being routed to or published. (For SOAP 1.1 proxies, `$body` and `$header` use SOAP 1.1 namespace Body and namespace; for SOAP 1.2 proxies, they use SOAP 1.2 namespace Body and namespace.) The `$attachments` variable contains the SOAP message attachments if any. To learn more about the message context variables, see Message-Related Variables.
Transport-Typed	Transport-typed services have an empty binding definition and only apply to EJB business services. A WSDL is not specified. Instead the transport automatically defines the WSDL for the service. A zip containing this WSDL can be exported from the ALSB Console or the ALSB Plug-in for WorkSpace Studio. This WSDL will not have a port defined. The EJB Transport-Typed Service is an outbound transport to access EJBs from ALSB. It is a self-described transport that generates a WSDL to describe its service interface. The EJB transport features transaction and security context propagation. Business services built using an EJB transport can be used for publish, service callout, and service invocation.

Table 2-6 Configuration Properties for Different Business Service Types

Property	Description
Any XML Services	**Binding Definition:** The only information this service type defines is that the service is receiving/sending XML messages—regardless of their WSDL binding definition. Therefore, the binding configuration for this type is empty. In addition, as there is no binding configuration, the combination of this type and the content-type of the message is sufficient to determine whether or not there are attachments to the message. As per their definition, any services (SOAP or XML) do not have any WSDL definition. **Run-Time Variables:** The `$body` variable holds the incoming XML message wrapped in a `<soap:Body>` element. (For SOAP 1.1 proxies, `$body` uses SOAP 1.1 namespace Body; for SOAP 1.2 proxies, it uses SOAP 1.2 namespace Body.) The `$attachments` variable contains message attachments if there are any. The `$header` variable is not applicable to this service type and is set to its default value. To learn more about the message context variables, see Message-Related Variables.

Table 2-6 Configuration Properties for Different Business Service Types

Property	Description
Messaging Services	**Binding Definition:** The binding definition for messaging services consists of configuring the content-type of the messages that are exchanged. The content-type for the response does not need to be the same as for the request; therefore, the response is configured separately (for example, the service could accept an MFL message and return an XML acknowledgment receipt). By definition, messaging-based services do not have any WSDL definition. It is not possible to request a WSDL document for those services. The following content types are available for the request (and response): • Binary • Text • XML • MFL • None **Run-Time Variables:** This service type is message based. The `$body` variable holds the incoming message wrapped in a `<soap:Body>` element. (For SOAP 1.1 proxies, `$body` uses SOAP 1.1 namespace Body; for SOAP 1.2 proxies, it uses SOAP 1.2 namespace Body.) The `$header` variable is not applicable to this service type, and is set to its default value. The `$attachments` variable contains message attachments if there are any. To learn more about the message context variables, see Message-Related Variables.

If a business service requires Web Service security, make sure the WSDL you specify has the necessary WS-Policies attached when you create the business service. Furthermore, if the WS-Policy of the business service requires encryption, make sure the public certificate of the business service is embedded in the WSDL. If the business service is a WebLogic Server 9.0 or later Web Service, you can retrieve its WSDL using the `http://<host>:<port>/<service url>?WSDL` URL, the public certificate will be automatically embedded for you if necessary.

Viewing Resource Details

ALSB provides a resource servlet that is used to expose the resources registered in ALSB. The resources registered with ALSB include:

The format of the URLs used to expose the resources is as follows:

- WSDL (a WSDL registered as a resource in ALSB)
- Schema
- MFL
- WS-Policy
- WSDL (an effective WSDL with resolved policies and port information for a service—this derived WSDL is available if the service was created using a WSDL).

You can use the following URL formats to expose the resource details:

- `http://host:port/sbresource?WSDL/project/...wsdlname`
- `http://host:port/sbresource?POLICY/project/...policyname`
- `http://host:port/sbresource?MFL/project/...mflname`
- `http://host:port/sbresource?SCHEMA/project/...schemaname`
- `http://host:port/sbresource?PROXY/project/...proxyname`
- `http://host:port/sbresource?BIZ/project/...business_service_name`

Note: The URLs used to expose the resources in ALSB must be encoded in UTF-8 in order to escape special characters.

CHAPTER 3

Modeling Message Flow in ALSB

In ALSB, a message flow defines the implementation of a proxy service. You can create and configure ALSB proxy services in the ALSB Console or the ALSB Plug-in for WorkSpace Studio. This section describes message flows and presents guidelines for designing them.

The following sections describe message flows in ALSB:

- Throttling Pattern
- WS-I Compliance
- Converting Between SOAP 1.1 and SOAP 1.2

For instructions on creating and configuring message flows in the ALSB Console, see:

- Proxy Services: Message Flows in *Using the AquaLogic Service Bus Console*
- Proxy Services: Actions in *Using the AquaLogic Service Bus Console*
- Proxy Services: Error Handlers in *Using the AquaLogic Service Bus Console.*

For instructions on creating and configuring message flows in the ALSB Plug-in for WorkSpace Studio, see:

- Working with Proxy Services in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*
- Working with Proxy Service Message Flows in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*

Message Flow Components

A message flow is composed of components that define the logic for routing and manipulating messages as they flow through a proxy service. Nodes are configured to route messages through the message flow, and stages and actions contain rules for processing and transforming messages.

Most of the processing logic in a message flow is handled in pipelines. A pipeline is a named sequence of stages representing a non-branching one-way processing path. Pipelines belong to one of the following categories:

- Request pipelines process the request path of the message flow.
- Response pipelines process the response path of the message flow.
- Error pipelines handle errors for stages and nodes in a message flow as well as errors at the level of the message flow (service).

To implement the processing logic of a proxy service, request and response pipelines are paired together in pipeline pair nodes. These pipeline pair nodes can be combined with other nodes into a single-rooted tree structure to control overall flow.

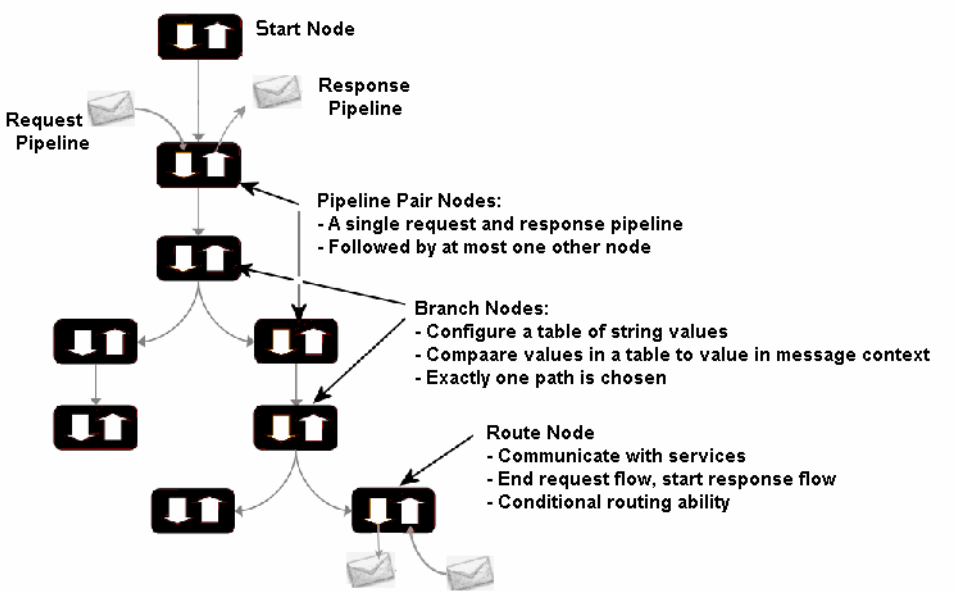
Table 3-1 describes the components available for defining message flows.

Table 3-1 Message Flow Components

Component Type	Summary
Start node	Every message flow begins with a start node. All messages enter the message flow through the start node, and all response messages are returned to the client through the start node. There is nothing to configure in a start node.
Pipeline pair node	A pipeline pair node combines a single request pipeline and a single response pipeline in one top-level element. A pipeline pair node can have only one direct descendant in the message flow. During request processing, only the request pipeline is executed when ALSB processes a pipeline pair node. The execution path is reversed when ALSB processes the response pipeline.
Stage	Request pipelines, response pipelines, and error handlers can contain stages, where you configure actions to manipulate messages passing through the pipeline. See also Configuring Actions in Stages and Route Nodes.
Error handler	An error handler can be attached to any node or stage, to handle potential errors at that location. See also Handling Errors in Message Flows.
Branch node	A branch node allows processing to proceed along exactly one of several possible paths. Operational branching is supported for WSDL-based services, where the branching is based on operations defined in the WSDL. Conditional branching is supported for conditions defined in an XPath-based switch table. See also Branching in Message Flows.
Route node	A route node performs request/response communication with another service. It represents the boundary between request and response processing for the proxy service. When the route node dispatches a request message, the request processing is considered complete. When the route node receives a response message, the response processing begins. The route node supports conditional routing as well as request and response transformations. Because a route node represents the boundary between request and response processing, it cannot have any descendants in the message flow. See also Configuring Actions in Stages and Route Nodes.

Figure 3-1 shows a high level view of components in a message flow definition.

Figure 3-1 Components of Message Flow



Building a Message Flow

The only components required in a message flow are a start node and a route node. No restrictions exist on what other components can be chained together in the message flow. You could create a single route node that contained all the logic for the flow. Or, you could link two pipeline pair nodes without a branch node in between. If you use branch nodes, each branch node could start with a different element. One branch could terminate with a route node, another could be followed by a pipeline pair, and yet another could have no descendant. (When a branch with no descendants is executed at run time, response processing begins immediately.)

However, in general, a message flow is likely to be designed in one of the following ways:

- For non-operational services (services that are not based on WSDLs with operations), the flow consists of a single pipeline pair at the root followed by a route node.
- For operational services, the flow consists of a single pipeline pair at the root, followed by a branch node based on an operation, with each branch consisting of a pipeline pair followed by a route node.

Message Execution

Table 3-2 briefly describes how messages are processed in a typical message flow.

Table 3-2 Path of a Message During a Message Flow

Message Flow Node	What Happens During Message Processing?
Request Processing	
Start node	Request processing begins at the start node.
Pipeline pair node	Executes the request pipeline only.
Branch node	Evaluates the branch table and proceeds down the relevant branch.
Route node	Performs the route along with any request transformations. In the message flow, regardless of whether routing takes place or not, the route node represents the transition from processing a request to processing a response. At the route node, the direction of the message flow is reversed. If a request path does not have a route node, the response processing is initiated in the reverse direction without waiting for any response.
Response Processing	Skips any branch nodes and continues with the node that preceded the branch.
Route node	Executes any response transformations.
Branch node	Skips any branch nodes and continues with the node that preceded the branch.
Pipeline pair node	Executes the response pipeline.
Start node	Sends the response back to the client.

Branching in Message Flows

Two kinds of branching are supported in message flows: *operational* branching, configured in an operational branch node, and *conditional* branching, configured in a conditional branch node.

Operational Branching

When message flows define WSDL-based proxy services, operation-specific processing is required. When you create an operational branch node in a message flow, you can build branching logic based on the operations defined in the WSDL.

You must use operational branching when a proxy service is based on a WSDL with multiple operations. You can consider using an operational branch node to handle messages separately for each operation.

Conditional Branching

Use conditional branching to branch based on a specified condition, for example the document type of a message.

Conditional branching is driven by a lookup table with each branch tagged with simple, unique string values, for example, `QuantityEqualToOrLessThan150` and `QuantityMoreThan150`.

You can configure a conditional branch to branch based on the value of a variable in the message context (declared, for example, in a stage earlier in the message flow), or you can configure the condition to branch based on the results of an XPath expression defined in the branch itself.

At run time, the variable or the expression is evaluated, and the resulting value is used to determine which branch to follow. If no branch matches the value, the default branch is followed. A branch node may have several descendants in the message flow: one for each branch, including the default branch. You should always define a default branch. You should design the proxy service in such a way that the value of a lookup variable is set before reaching the branch node.

For example, consider the following case using a lookup variable. A proxy service is of type any SOAP or any XML, and you need to determine the type of the message so you can perform conditional branching. You can design a stage action to identify the message type and then design a conditional branching node later in the flow to separate processing based on the message type.

Now consider the following case using an XPath expression in the conditional branch node. You want to branch based on the quantity in an order. That quantity is passed via a variable that can be retrieved from a known location in `$body`. You could define the following XPath expression to retrieve the quantity:

```
declare namespace openuri="http://www.openuri.org/";

declare namespace com="bea.com/demo/orders/cmnCust";

./openuri:processCust/com:cmnCust/com:Order_Items/com:Item/com:Quantity
```

The condition (for example, `<500`) is then evaluated in order down the message flow against the expression. Whichever condition is satisfied first determines which branch is followed. If no branch condition is satisfied, then the default branch is followed.

You can use conditional branching to expose the routing alternatives for the message flow at the top level flow view. For example, consider a situation where you want to invoke service A or

service B based on a condition known early in the message flow (for example, the message type). You could configure the conditional branching in a routing table in the route node. However, that makes the branching somewhat more difficult to follow if you are just looking at the top level of the flow. Instead, you could use a conditional branch node to expose this branching in the message flow itself and use simple route nodes as the subflows for each of the branches.

Consider your business scenario before deciding whether you configure branching in the message flow or in a stage or route node. When making your decision, remember that configuring branches in the message flow can be awkward in the design interface if a large number of branches extend from the branch node.

Configuring Actions in Stages and Route Nodes

Actions provide instructions for handling messages in pipeline stages, error handler stages, and route nodes. The context determines which actions are available in the ALSB Console or in the ALSB Plug-in for WorkSpace Studio, as described in the following sections:

- Communication Actions
- Flow Control Actions
- Message Processing Actions
- Reporting Actions

See Also

- Proxy Services: Actions in *Using the AquaLogic Service Bus Console*
- Working with Proxy Service Message Flows in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*

Communication Actions

Communication actions control message flow. Table 3-3 describes the communication actions.

Table 3-3 Communication Actions

Action	Use to...	Available in
Dynamic publish	Publish a message to a service specified by an XQuery expression.	• Pipeline stage • Error handler stage • Route node
Publish	Identify a statically specified target service for a message and to configure how the message is packaged and sent to that service.	• Pipeline stage • Error handler stage
Publish table	Publish a message to zero or more statically specified services. Switch-style condition logic is used to determine at run time which services will be used for the publish.	• Pipeline stage • Error handler stage
Routing options	Modify any or all of the following properties in the outbound request: URI, Quality of Service, Mode, Retry parameters, Message Priority.	• Pipeline stage
Service callout	Configure a synchronous (blocking) callout to an ALSB-registered proxy or business service. See Constructing Service Callout Messages.	• Pipeline stage • Error handler stage
Transport headers	Set the header values in messages. See Configuring Transport Headers in Message Flows.	• Pipeline stage • Error handler stage

Flow Control Actions

Flow controls actions implement conditional routing, conditional looping, and error handling. You can also use them to notify the invoker of success or to skip the rest of the actions in the stage. Table 3-4 describes the flow control actions.

Table 3-4 Flow Control Actions

Action	Use to...	Available in
For each	Iterate over a sequence of values and execute a block of actions	• Pipeline stage • Error handler stage
If... then...	Perform an action or set of actions conditionally, based on the Boolean result of an XQuery expression.	• Pipeline stage • Route node • Error handler stage
Raise error	Raise an exception with a specified error code (a string) and description.	• Pipeline stage • Error handler stage
Reply	Specify that an immediate reply be sent to the invoker. The reply action can be used in the request, response or error pipeline. You can configure it to result in a reply with success or failure. In the case of reply with failure where the inbound transport is HTTP, the reply action specifies that an immediate reply is sent to the invoker.	• Pipeline stage • Error handler stage
Resume	Resume message flow after an error is handled by an error handler. This action has no parameters and can only be used in error handlers.	• Error handler stage
Skip	Specify that at run time, the execution of this stage is skipped and the processing proceeds to the next stage in the message flow. This action has no parameters and can be used in the request, response or error pipelines.	• Pipeline stage • Error handler stage

Message Processing Actions

The actions in this category process the message flow. Table 3-5 describes the message processing actions.

Table 3-5 Message Processing Actions

Action	Use to...	Available in
Assign	Assign the result of an XQuery expression to a context variable.	• Pipeline stage • Error handler stage
Delete	Delete a context variable or a set of nodes specified by an XPath expression.	• Pipeline stage • Error handler stage
Insert	Insert the result of an XQuery expression at an identified place relative to nodes selected by an XPath expression.	• Pipeline stage • Error handler stage
Java callout	Invoke a Java method, or EJB business service, from within the message flow.	• Pipeline stage • Error handler stage
MFL transform	Convert message content from XML to non-XML, or vice versa, in the message pipeline. An MFL is a specialized XML document used to describe the layout of binary data. It is a BEA proprietary language used to define rules to transform formatted binary data into XML data, or vice versa.	• Pipeline stage • Error handler stage
Rename	Rename elements selected by an XPath expression without modifying the contents of the element.	• Pipeline stage • Error handler stage
Replace	Replace a node or the contents of a node specified by an XPath expression. The node or its contents are replaced with the value returned by an XQuery expression. A replace action can be used to replace simple values, elements and even attributes. An XQuery expression that returns nothing is equivalent to deleting the identified nodes or making them empty, depending upon whether the action is replacing entire nodes or just node contents. The replace action is one of a set of Update actions.	• Pipeline stage • Error handler stage
Validate	Validate elements selected by an XPath expression against an XML schema element or a WSDL resource. You can validate global elements only; ALSB does not support validation against local elements.	• Pipeline stage • Error handler stage

Reporting Actions

You use the actions in this category to log or report errors and generate alerts if required in a message flow within a stage. Table 3-4 describes the reporting actions.

Table 3-6 Reporting Actions

Action	Use to...	Available in
Alert	Generate alerts based on message context in a pipeline, to send to an alert destination. Unlike SLA alerts, notifications generated by the alert action are primarily intended for business purposes, or to report errors, and not for monitoring system health. Alert destination should be configured and chosen with this in mind. If pipeline alerting is not enabled for the service or enabled at the domain level, the configured alert action is bypassed during message processing.	• Pipeline stage • Error handler stage
Log	Construct a message to be logged and to define a set of attributes with which the message is logged.	• Pipeline stage • Error handler stage
Report	Enable message reporting for a proxy service.	• Pipeline stage • Error handler stage

Configuring Transport Headers in Message Flows

The transport header action is a communication type action, and it is available in pipeline stages and error handler stages.

Configuring Global Pass Through and Header-Specific Copy Options for Transport Headers

The following options are available when you configure a transport headers action:

- The **Pass all Headers through Pipeline** option specifies that at run time, the transport headers action passes all headers through from the inbound message to the outbound

message or vice versa. Every header in the source set of headers is copied to the target header set, overwriting any existing values in the target header set.

- The **Copy Header from Inbound Request** option and the **Copy Header from Outbound Response** options specifies that at run time, the transport headers action copies the specific header with which this option is associated from the inbound message to the outbound message or vice versa.

Use the options in a way that best suits your scenario. Both options result in the headers in the source header set being copied to the target header set, overwriting any existing value in the target set. Note that the **Pass all Headers through Pipeline** option is executed before the header-specific **Copy Header** options. In other words, for a given transport headers action configuration, if you select **Pass all Headers through Pipeline**, there is no need to select the **Copy Header** option for given headers.

However, you can select **Pass all Headers through Pipeline** to copy all headers, and subsequently configure the action such that individual headers are deleted by selecting **Delete Header** for specific headers.

WARNING: Because transport headers are specific to the transport types, it is recommended that the pass-through (or copy) options only be used to copy headers between services of the same transport type. Passing (or copying) headers between services of different transport types can result in an error if the header being passed is not accepted by the target transport. For the same reasons, be careful when you specify a header name using the Set Header option.

Understanding How the Run Time Uses the Transport Headers Settings

As described above, you can use transport header actions to configure the values of the transport headers for outbound requests (the messages sent out by a proxy service in route, publish, or service callout actions) and inbound responses (the response messages a proxy service sends back to clients). In general, the header values can be:

- Specified using an XQuery expression
- Passed through from the source to the target service
- Deleted while going from the source to the target service

The transport headers action allows you to set, delete, or pass-through the headers in `$inbound` or `$outbound`. If you set or delete these headers and then log `$inbound` or `$outbound`, you can see the effects of your changes. However, when the message is sent out, the ALSB binding layer may modify or remove some headers in `$inbound` or `$outbound` and the underlying transport

may in turn, ignore some of these headers and use its own values. An important distinction is that any modifications done by the binding layer on a header are done directly to `$inbound` and `$outbound`, whereas modifications done by the transport affects only the message's *wire format*. For example, although you can specify a value for the outbound `Content-Length` header, the binding layer deletes it from `$outbound` when sending the message. Consequently, the modification is visible in the response path (for example, you can see the modified value if you log `$outbound`). If you set the `User-Agent` header in `$outbound`, the HTTP transport ignores it and use its own value—however, the value in `$outbound` is not changed.

Table 3-7 describes the transport headers that are ignored or overwritten at run time and other limitations that exist for specific transport headers.

Table 3-7 Limitations to Transport Header Values You Specify in Transport Header Actions

Transport	Description of Limitation...	Transport Headers Affected By Limitation...	
		Outbound Request	**Inbound Response**
HTTP	ALSB run time may overwrite these headers in the binding layer when preparing the message for dispatch. If these headers are modified, `$inbound` and `$outbound` are updated accordingly.	• Content-Type	• Content-Type
	The underlying transport may ignore these headers and use different values when sending the message. Any changes done by the transport will not be reflected in `$inbound` or `$outbound`.	• Accept • Content-Length • Connection • Host • User-Agent	• Content-Length • Date • Transfer-Encoding

Table 3-7 Limitations to Transport Header Values You Specify in Transport Header Actions

Transport	Description of Limitation...	Transport Headers Affected By Limitation...	
		Outbound Request	Inbound Response
JMS	Can only be set when the request is with respect to a one-way service or a request/response service based on `JMSMessageID` correlation. If sending to a request/response service based on `JMSCorrelationID` correlation, these headers are overwritten at run time.	• JMSCorrelationID	• JMSCorrelationID
	Should be set to the message time-to-live in milliseconds. The resulting value in the message received is the sum of the time-to-live value specified by the client and the GMT at the time of the send or publish[1].	• JMSExpiration	• JMSExpiration
	The ALSB run time sets these headers. In other words, any specifications you make for these headers at design time are overwritten at run time.	• JMSMessageID • JMSRedelivered • JMSTimestamp • JMSXDeliveryCount • JMSXUserID • JMS_IBM_PutDate[2] • JMS_IBM_PutTime [2] • JMS_IBM_PutApplType [2] • JMS_IBM_Encoding [2] • JMS_IBM_Character_Set [2]	• JMSMessageID • JMSRedelivered • JMSTimestamp • JMSXDeliveryCount • JMSXUserID • JMS_IBM_PutDate [2] • JMS_IBM_PutTime [2] • JMS_IBM_PutApplType [2] • JMS_IBM_Encoding [2] • JMS_IBM_Character_Set [2]

Table 3-7 Limitations to Transport Header Values You Specify in Transport Header Actions

Transport	Description of Limitation...	Transport Headers Affected By Limitation...	
		Outbound Request	Inbound Response
	Because IBM MQ does not allow certain properties to be set by a client application, if you set these headers with respect to an IBM MQ destination, a run-time exception is raised.	• JMSXDeliveryCount • JMSXUserID • JMSXAppID	• JMSXDeliveryCount • JMSXUserID • JMSXAppID

Table 3-7 Limitations to Transport Header Values You Specify in Transport Header Actions

Transport	Description of Limitation...	Transport Headers Affected By Limitation...	
		Outbound Request	**Inbound Response**
	These headers cannot be deleted when the **Pass all Headers through Pipeline** option is also specified.	• JMSDeliveryMode • JMSExpiration • JMSMessageID • JMSRedelivered • JMSTimestamp • JMSXDeliveryCount	• JMSDeliveryMode • JMSExpiration • JMSMessageID • JMSRedelivered • JMSTimestamp • JMSXDeliveryCount • JMSCorelationID—if the inbound message has the correlation ID set. For example, if the inbound response comes from a registered JMS business service
FTP **File**	No limitations. In other words you can set or delete the header(s)[3] for File and FTP transports and your specifications are honored by the ALSB run time.		
E-mail	The ALSB run time sets these headers. In other words, any specifications you make for these headers at design time are overwritten at run time.	• Content-Type	• Content-Type

Table 3-7 Limitations to Transport Header Values You Specify in Transport Header Actions

Transport	Description of Limitation...	Transport Headers Affected By Limitation...	
		Outbound Request	Inbound Response
	These headers have no meaning for outbound requests. If they are set dynamically (that is, if they are set in the `$outbound` headers section), they are ignored.[4] These headers are received in `$inbound`. `Date` is the time the mail was sent by the sender. `From` is retrieved from incoming mail headers.	• From • Date	

1. For example, if you set the `JMSExpiration` header to 1000, and at the time of the send, GMT is 1,000,000 (as a result of System.currentTimeMillis()), the resulting value of the `JMSExpiration` property in the JMS message is 1,000,1000
2. Header names with the JMS_IBM prefix are to be used with respect to destinations hosted by an IBM MQ server
3. For FTP and file proxies, there is an transport request header 'fileName'. The value of this request header is the name of the file being polled.
4. `From` and `Date` headers are also not applicable for `$outbound` request headers for e-mail business services. So there is no point in setting these headers for e-mail business services.

Note: The same limitations around setting certain transport headers and metadata are true when you set the `inbound` and `outbound` context variables, and when you use the ALSB Test Console to test your proxy or business services.

Performing Transformations in Message Flows

Transformation maps describe the mapping between two data types. ALSB supports data mapping that uses the XQuery and the eXtensible Stylesheet Language Transformation (XSLT) standards. XSLT maps describe XML-to-XML mappings. XQuery maps can describe XML-to-XML, XML to non-XML, and non-XML to XML mappings.

The point in a message flow at which you specify a transformation depends on whether:

- The message format relies on target services—that is, the message format must be in a format acceptable by the route destination. This applies when the transformation is performed in a route node or in one of the publish actions.

  Publish actions identify a target service for a message and configure how the message is packaged and sent to that service. ALSB also provides publish table actions. A publish table action consists of a set of routes wrapped in a switch-style condition table. It is a shorthand construct that allows different routes to be selected, based upon the results of a single XQuery expression.

- You perform the transformation on the response or request message regardless of the route destination. In this case, you can configure the transformations in the request or response pipeline stages.

Transformations and Publish Actions

When transformations are designed in publish actions, the transformations have a local copy of the `$outbound` variable and message-related variables (`$header`, `$body`, and `$attachments`). Any changes you make to an outbound message in a publish action affect only the published message. In other words, the changes you make in the publish action are rolled back before the message flow proceeds to any actions that follow the publish action in your message flow.

For example, consider a message flow that deals with a large purchase order, and you have to send the summary of the purchase order, through e-mail, to the manager. The summary of the of the purchase order is created in the SOAP body of the incoming message when you include a publish action in the request pipeline. In the publish action, the purchase order data is transformed into a summary of the purchase order—for example, all the attachments in `$attachments` can be deleted because they are not required in the summary of the purchase order.

Transformations and Route Nodes

You may need to route messages to one of two destinations, based on a WS-addressing header. In that case, content-based routing and the second destination require the newer version of the document in the SOAP body. In this situation, you can configure the route node to conditionally route to one of the two destinations. You can configure a transformation in the route node to transform the document for the second destination.

You can also set the control elements in the outbound context variable (`$outbound`) to influence the behavior of the system for the outbound message (for example, you can set the Quality of Service).

See Inbound and Outbound Variables and Constructing Messages to Dispatch for information about the sub-elements of the inbound and outbound variables and how the content of messages is constructed using the values of the variables in the message context.

See also:

- Message Context
- XQuery Transformations and XSL Transformations in *Using the AquaLogic Service Bus Console*.
- Creating XQuery Transformations and Creating XSL Transformations in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*.
- Transforming Data Using the XQuery Mapper in *Transforming Data Using the XQuery Mapper*.

Constructing Service Callout Messages

When ALSB makes a call to a service via a service callout action, the content of the message is constructed using the values of variables in the message context. The message content for outbound messages is handled differently depending upon the type of the target service. How the message content is created depends on the type of the target service and whether you choose to configure the SOAP body or the payload (parameters or document), as described in the following topics:

- "SOAP Document Style Services" on page 3-19
- "SOAP RPC Style Services" on page 3-22
- "XML Services" on page 3-25
- "Messaging Services" on page 3-26

SOAP Document Style Services

Messages for SOAP Document Style services (including EJB document and document-wrapped services), can be constructed as follows:

- The variable assigned for the request document contains the SOAP body.
- The variable assigned for the SOAP request header contains the SOAP header.

- The response must be a single XML document—it is the content of the SOAP body plus the SOAP header (if specified)

To illustrate how messages are constructed during callouts to SOAP Document Style services, consider a service callout action configured as follows:

- **Request Document Variable:** `myreq`
- **Response Document Variable:** `myresp`
- **SOAP Request Header:** `reqheader`
- **SOAP Response Header:** `respheader`

Assume also that at run time, the request document variable, `myreq`, is bound to the following XML.

Listing 3-1 Content of Request Variable (myreq)

```
<sayHello xmlns="http://www.openuri.org/">
    <intVal>100</intVal>
    <string>Hello AquaLogic</string>
</sayHello>
```

At run time, the SOAP request header variable, `reqheader`, is bound to the following SOAP header.

Listing 3-2 Content of SOAP Request Header Variable (reqheader)

```
<soap:Header xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing">
    <wsa:Action>...</wsa:Action>
    <wsa:To>...</wsa:To>
    <wsa:From>...</wsa:From>
    <wsa:ReplyTo>...</wsa:ReplyTo>
    <wsa:FaultTo>...</wsa:FaultTo>
  </soap:Header>
```

In this example scenario, the full body of the message sent to the external service is shown in Listing 3-3 (the contents of the `myreq` and `reqheader` variables are shown in bold).

Listing 3-3 Message Sent to the Service as a Result of Service Callout Action

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Header xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing">
        <wsa:Action>...</wsa:Action>
        <wsa:To>...</wsa:To>
        <wsa:From>...</wsa:From>
        <wsa:ReplyTo>...</wsa:ReplyTo>
        <wsa:FaultTo>...</wsa:FaultTo>
    </soap:Header>
    <soapenv:Body>
        <sayHello xmlns="http://www.openuri.org/">
            <intVal>100</intVal>
            <string>Hello AquaLogic</string>
        </sayHello>
    </soapenv:Body>
</soapenv:Envelope>
```

Based on the configuration of the service callout action described above, the response from the service is assigned to the `myresp` variable. The full response from the external service is shown in Listing 3-4.

Listing 3-4 Response Message From the Service as a Result of Service Callout Action

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:soapenc="http://schemas.xmlsoap.
org/soap/encoding/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
    <env:Header/>
    <env:Body
env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <m:sayHelloResponse xmlns:m="http://www.openuri.org/">
            <result xsi:type="xsd:string">This message brought to you by
Hello AquaLogic and the number 100
            </result>
        </m:sayHelloResponse>
    </env:Body>
</env:Envelope>
```

In this scenario, the `myresp` variable is assigned the value shown in Listing 3-5.

Listing 3-5 Content of Response Variable (myresp) as a Result of Service Callout Action

```
<m:sayHelloResponse xmlns:m="http://www.openuri.org/">
    <result ns0:type="xsd:string"
xmlns:ns0="http://www.w3.org/2001/XMLSchema-instance">
This message brought to you by Hello AquaLogic and the number 100
    </result>
</m:sayHelloResponse>
```

SOAP RPC Style Services

Messages for SOAP RPC Style services (including EJB RPC services) can be constructed as follows:

- Request messages are assembled from message context variables.
  - The SOAP body is built based on the SOAP RPC format (operation wrapper, parameter wrappers, and so on).
  - The SOAP header is the content of the variable specified for the SOAP request header, if one is specified.
  - Part as element—the parameter value is the variable content.

  - Part as simple type—the parameter value is the string representation of the variable content.
  - Part as complex type—the parameter corresponds to renaming the root of the variable content after the parameter name.

- Response messages are assembled as follows:
  - The output content is the content of SOAP header, if a SOAP header is specified.
  - Part as element—the output content is the child element of the parameter; there is at most one child element.
  - Part as simple/complex type—the output content is the parameter itself.

To illustrate how messages are constructed during callouts to SOAP RPC Style services, look at this example with the following configuration:

- A message context variable `input1` is bound to a value 100.
- A message context variable `input2` is bound to a string value: `Hello AquaLogic`.
- A service callout action configured as follows:
  - **Request Parameter `intval`:** `input1`
  - **Request Parameter `string`:** `input2`
  - **Response Parameter `result`:** `output1`

In this scenario, the body of the outbound message to the service is shown in Listing 3-6.

Listing 3-6 Content of Outbound Message

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Body>
        <sayHello2 xmlns="http://www.openuri.org/">
            <intVal>100</intVal>
            <string >Hello AquaLogic</string>
        </sayHello2>
    </soapenv:Body>
</soapenv:Envelope>
```

The response returned by the service to which the call was made is shown in Listing 3-7.

Listing 3-7 Content of Response Message From the helloWorld Service

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <env:Header/>
    <env:Body
env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <m:sayHello2Response xmlns:m="http://www.openuri.org/">
            <result xsi:type="n1:HelloWorldResult" xmlns:n1="java:">
                <message xsi:type="xsd:string">
                This message brought to you by Hello AquaLogic and the
number 100
                </message>
            </result>
        </m:sayHello2Response>
    </env:Body>
</env:Envelope>
```

The message context variable `output1` is assigned the value shown in Listing 3-8.

Listing 3-8 Content of Output Variable (output1)

```
<message ns0:type="xsd:string"
xmlns:ns0="http://www.w3.org/2001/XMLSchema-intance">
This message brought to you by Hello AquaLogic and the number 100</message>
```

XML Services

Messages for XML services can be constructed as follows:

- The request message is the content of the variable assigned for the request document.
- The content of the request variable must be a single XML document.
- The output document is the response message.

To illustrate how messages are constructed during callouts to XML services, take for example a service callout action configured as follows:

- **Request Document Variable:** `myreq`
- **Response Document Variable:** `myresp`

Assume also that at run time, the request document variable, `myreq`, is bound to the following XML.

Listing 3-9 Content of myreq Variable

```
<sayHello xmlns="http://www.openuri.org/">
    <intVal>100</intVal>
    <string>Hello AquaLogic</string>
</sayHello>
```

In this scenario:

- The outbound message payload is the value of the `myreq` variable, as shown in Table 3-9.
- The response and the value assigned to the message context variable, `myresp`, is shown in Listing 3-10.

Listing 3-10 Content of myresp Variable

```
<m:sayHelloResponse xmlns:m="http://www.openuri.org/">
     <result xsi:type="xsd:string">This message brought to you by Hello
AquaLogic and the number 100
```

```
        </result>
    </m:sayHelloResponse>
```

Messaging Services

In the case of Messaging services:

- The request message is the content of the request variable. The content can be simple text, XML, or binary data represented by an instance of `<binary-content ref=.../>` reference XML.
- Response messages are treated as binary, so the response variable will contain an instance of `<binary-content ref= ... />` reference XML, regardless of the actual content received.

For example, if the request message context variable myreq is bound to an XML document of the following format: `<hello>there</hello>`, the outbound message contains exactly this payload. The response message context variable (myresp) is bound to a reference element similar to the following:

```
<binary-content ref=" cid:1850733759955566502-2ca29e5c.1079b180f61.-7fd8"/>
```

Handling Errors as the Result of a Service Callout

You can configure error handling at the message flow, pipeline, route node, and stage level. The types of errors that are received from an external service as the result of a service callout include transport errors, SOAP faults, responses that do not conform to an expected response, and so on.

The `fault` context variable is set differently for each type of error returned. You can build your business and error handling logic based on the content of the `fault` variable. To learn more about `$fault`, see "Fault Variable" on page 5-19.

Transport Errors

When a transport error is received from an external service and there is no error response payload returned to ALSB by the transport provider (for example, in the case that an HTTP 403 error code is returned), the service callout action throws an exception, which in turn causes the pipeline to raise an error. The fault variable in a user-configured error handler is bound to a message formatted similarly to that shown in Listing 3-11.

Listing 3-11 Contents of the ALSB fault Variable—Transport Error, no Error Response Payload

```
<con:fault xmlns:con="http://www.bea.com/wli/sb/context">
  <con:errorCode>BEA-380000</con:errorCode>
  <con:reason>Not Found</con:reason>
   <con:details>
   .......
   </con:details>
    <con:location>
       <con:node>PipelinePairNode1</con:node>
       <con:Pipeline>PipelinePairNode1_request</con:Pipeline>
    <con:Stage>Stage1</con:Stage>
    </con:location>
</con:fault>
```

In the case that there is a payload associated with the transport error—for example, when an HTTP 500 error code is received from the business service and there is XML payload in the response—a message context fault is generated with the custom error code: BEA-382502.

The following conditions must be met for a BEA-382502 error response code to be triggered as the result of a response from a service—when that service uses an HTTP or JMS transport:

- (HTTP) The response code must be any code other than 200 or 202.
- (JMS) The response must have a property set to indicate that it is an error response—the transport metadata status code set to1 indicates an error.
- The content type must be text/xml.
- If the service is AnySoap or WSDL-based SOAP, then it must have a SOAP envelope. The body inside the SOAP envelope must be XML format; it cannot be text.
- If the service type is AnyXML, or a messaging service of type text returns XML content with a non-successful response code (any code other than 200 or 202).

If the transport is HTTP, the `ErrorResponseDetail` element will also contain the HTTP error code returned with the response. The `ErrorResponseDetail` element in the fault contains error response payload received from the service. Listing 3-12 shows an example of the `ErrorResponseDetail` element.

Listing 3-12 Contents of the ALSB fault Variable—Transport Error, with Error Response Payload

```
<ctx:Fault xmlns:ctx="http://www.bea.com/wli/sb/context">
    <ctx:errorCode>BEA-382502<ctx:errorCode>
    <ctx:reason> Service callout has received an error response from the
server</ctx:reason>
    <ctx:details>
        <alsb:ErrorResponseDetail xmlns:alsb="http://www.bea.com/...">
            <alsb:detail> <![CDATA[
. . .
       ]]>
            </alsb:detail>
          <alsb:http-response-code>500</alsb:http-response-code>
        </alsb:ErrorResponseDetail>
    </ctx:details>
    <ctx:location>. . .</ctx:location>
</ctx:Fault>
```

Note: The XML schema for the service callout-generated fault is shown in "XML Schema for the Service Callout-Generated Fault Details" on page 3-30.

SOAP Faults

In case an external service returns a SOAP fault, the ALSB run time sets up the context variable `$fault` with a custom error code and description with the details of the fault. To do so, the contents of the 3 elements under the `<SOAP-ENV:Fault>` element in the SOAP fault are extracted and used to construct an ALSB fault element.

Take for example a scenario in which a service returns the following error.

Listing 3-13 SOAP Fault Returned From Service Callout

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Client</faultcode>
```

```
      <faultstring>Application Error</faultstring>
      <detail>
        <message>That's an Error!</message>
        <errorcode>1006</errorcode>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The <faultcode>, <faultstring>, and <detail> elements are extracted and wrapped in an <alsb:ReceivedFault> element. Note that the faultcode element in Listing 3-15 contains a QName—any related namespace declarations are preserved. If the transport is HTTP, the ReceivedFault element will also contain the HTTP error code returned with the fault response.

The generated <alsb:ReceivedFault> element, along with the custom error code and the error string are used to construct the contents of the fault context variable, which in this example takes a format similar to that shown in Listing 3-14.

Listing 3-14 Contents of the ALSB Fault Variable—SOAP Fault

```
<ctx:Fault xmlns:ctx="http://www.bea.com/wli/sb/context">
    <ctx:errorCode>BEA-382500<ctx:errorCode>
    <ctx:reason> service callout received a soap Fault
response</ctx:reason>
    <ctx:details>
        <alsb:ReceivedFault xmlns:alsb="http://www.bea.com/...">
            <alsb:faultcode
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">SOAP-ENV:Clien
            </alsb:faultcode>
            <alsb:faultstring>Application Error</alsb:faultstring>
            <alsb:detail>
                <message>That's an Error!</message>
                <errorcode>1006</errorcode>
            </alsb:detail>
          <alsb:http-response-code>500</alsb:http-response-code>
        </alsb:ReceivedFault>
    </ctx:details>
    <ctx:location> </ctx:location>
</ctx:Fault>
```

Note: The unique error code BEA-382500 is reserved for the case when service callout actions receive SOAP fault responses.

Unexpected Responses

When a service returns a response message that is not what the proxy service run time expects, a message context fault will be generated and initialized with the custom error code BEA-382501. The details of the fault include the contents of the SOAP-Body element of the response. If the transport is HTTP, the `ReceivedFault` element will also contain the HTTP error code returned with the fault response.

The XML schema definition of the service callout-generated fault details is shown in Listing 3-15.

Listing 3-15 XML Schema for the Service Callout-Generated Fault Details

```
<xs:complexType name="ReceivedFaultDetail">
        <xs:sequence>
            <xs:element name="faultcode" type="xs:QName"/>
            <xs:element name="faultstring" type="xs:string"/>
            <xs:element name="detail" minOccurs="0" >
               <xs:complexType>
                 <xs:sequence>
                        <xs:any namespace="##any" minOccurs="0"
maxOccurs="unbounded" processContents="lax" />
                 </xs:sequence>
        <xs:anyAttribute namespace="##any" processContents="lax" />
               </xs:complexType>
            </xs:element>

            <xs:element name="http-response-code" type="xs:int"
minOccurs="0"/>\

type="xs:int" minOccurs="0"/>
        </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="UnrecognizedResponseDetail">
        <xs:sequence>
            <xs:element name="detail" minOccurs="0" type="xs:string" />
        </xs:sequence>
</xs:complexType>

<xs:complexType name="ErrorResponseDetail">
        <xs:sequence>
            <xs:element name="detail" minOccurs="0" type="xs:string" />
        </xs:sequence>
</xs:complexType>
```

Handling Errors in Message Flows

The process described in the next paragraph constitutes an error handling pipeline for the stage of an error handler. In addition, an error pipeline can be defined for a pipeline (request or response) or for an entire proxy service.

The error handler at the stage level is invoked for handling an error; If the stage-level error handler is not able to handle a given type of error, the pipeline error handler is invoked. If the pipeline-level error handler also fails to handle the error, the service-level error handler is invoked. If the service-level error handler also fails, the error is handled by the system.The following table summarizes the scope of the error handlers at various levels in the message flow.

Table 3-8 Scope of Error Handlers

Level	Scope
Stage	Handles all the errors within a stage.
Pipeline	Handles all the errors in a pipeline, along with any unhandled errors from any stage in a pipeline.
Service	Handles all the errors in a proxy service, along with any unhandled errors in any pipeline in a service. **Note:** All WS-Security errors are handled at this level.
System	Handles all the errors that are not handled any where else in a pipeline.

Note: There are exceptions to the scope of error handlers. For example, an exception thrown by a non-XML transformation at the stage level is only caught by the service-level error handler. Suppose a transformation occurs that transforms XML to MFL for an outgoing proxy service response message, it always occurs in the binding layer. Therefore, for example, if a non-XML output is missing a mandatory field at the stage level, only a service-level error handler can catch this error.

For more information on error messages and error handling, see "Error Messages and Handling" in Proxy Services: Error Handlers in *Using the AquaLogic Service Bus Console*.

You can handle errors by configuring a test that checks if an assertion is true and use the reply action configured false. You can repeat this test at various levels. Also you can have an error without an error handler at a lower level and handle it through an error handler at an higher level in message flow. In general, it is easier to handle specific errors at a stage level of the message flow and use error handlers at the higher level for more general default processing of errors that are not handled at the lower levels. It is good practice to explicitly handle anticipated errors in the pipelines and allow the service-level handler to handle unanticipated errors.

Note: You can only handle WS-Security related errors at the service level.

Generating the Error Message, Reporting, and Replying

A predefined context variable (the `fault` variable) is used to hold information about any error that occurs during message processing. When an error occurs, this variable is populated with information before the appropriate error handler is invoked. The `fault` variable is defined only in error handler pipelines and is not set in request and response pipelines, or in route or branch nodes. For additional information about `$fault`, see "Predefined Context Variables" on page 5-2.

In the event of errors for request/response type inbound messages, it is often necessary to send a message back to the originator outlining the reason why an error occurred. You can accomplish this by using a *Reply with Failure* action after configuring the message context variables with the response you want to send. For example, when an HTTP message fails, *Reply with Failure* generates the `HTTP 500` status. When a JMS message fails, *Reply with Failure* sets the `JMS_BEA_Error` property to true. The ALSB error actions are discussed in "Error Messages and Handling" in Proxy Services: Error Handlers in *Using the AquaLogic Service Bus Console*.

An error handling pipeline is invoked if a service invoked by a proxy service returns a SOAP fault or transport error. Any received SOAP fault is stored in `$body`, so if a *Reply with Failure* is executed without modifying `$body`, the original SOAP fault is returned to the client that invoked the service. If a reply action is not configured, the system error handler generates a new SOAP

fault message. The proxy service recognizes that a SOAP fault is returned because a HTTP error status is set, or the JMS property `SERVER_Error` is set to true.

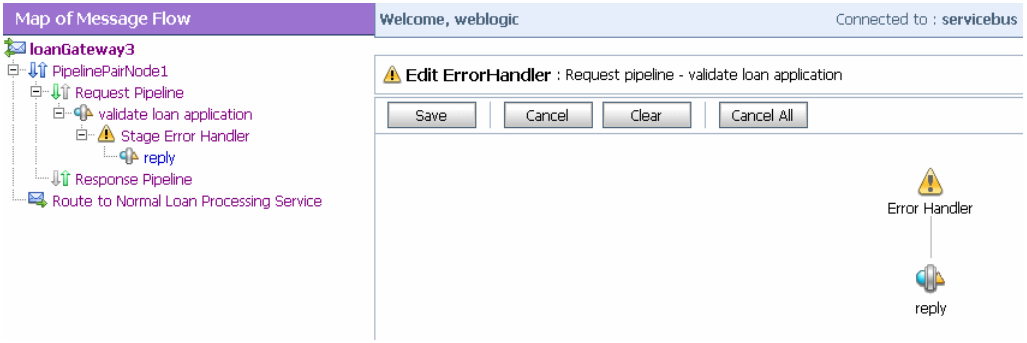
Some use cases require error reporting. You can use the report action in these situations. For example, consider a scenario in which the request pipeline reports a message for tracking purposes, but the service invoked by the route node fails after the reporting action. In this case, the reporting system logged the message, but there is no guarantee that the message was processed successfully, only that the message was successfully received.

You can use the ALSB Console to track the message to obtain an accurate picture of the message flow. This allows you to view the original reported message indicating the message was submitted for processing, and also the subsequent reported error indicating that the message was not processed correctly. To learn how to configure a report action and use the data reported at run time, see Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.

Example of Action Configuration in Error Handlers

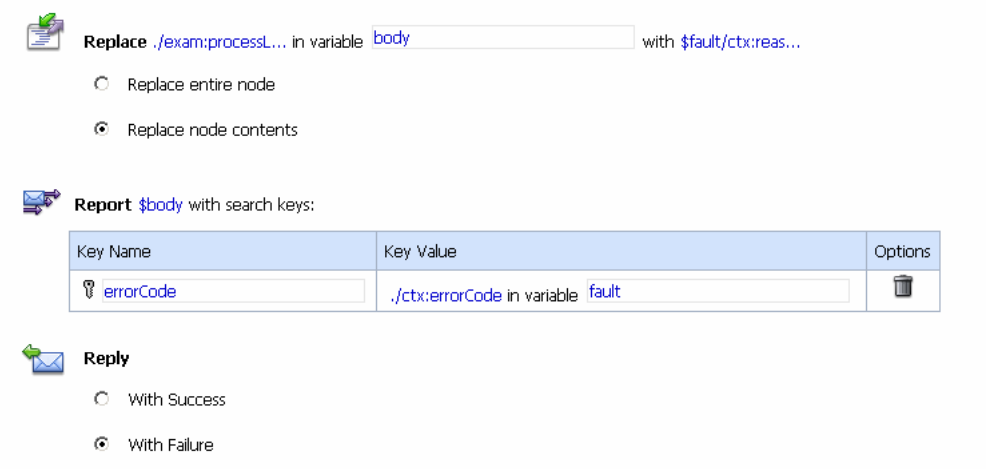
This example shows how you can configure the report and reply actions in error handlers. The message flow shown in Figure 3-2 includes an error handler on the `validate loan application` stage. The error handler in this case is a simple message flow with a single stage configured—it is represented in the ALSB Console as shown in Figure 3-2.

Figure 3-2 Error Handler Message Flow



The stage is, in turn, configured with actions (replace, report, and reply) as shown in Figure 3-3.

Figure 3-3 Actions in Stage Error Handler



The actions control the behavior of the stage in the pipeline error handler as follows:

- **Replace**—The contents of a specified element of the body variable are replaced with the contents of the `fault` context variable. The body variable element is specified by an XPath expression. The contents are replaced with the value returned by an XQuery expression—in this case `$fault/ctx:reason/text()`
- **Report**— Messages from the reporting action are written to the ALSB Reporting Data Stream if the error handler configured with this action is invoked. The JMS Reporting Provider reports the messages on the ALSB Dashboard. ALSB provides the capability to deliver message data to one or more reporting providers. Message data is captured from the body of the message and from any other variables associated with the message, such as header or inbound variables. You can use the message delivered to the reporting provider for functions such as tracking messages or regulatory auditing.

  When an error occurs, the contents of the fault context variable are reported. The key name is `errorCode`, and the key value is extracted from the fault variable using the following XPath expression: `./ctx:errorCode`. Key/value pairs are the key identifiers that identify these messages in the Dashboard at run time.

  To configure a report action and use the data reported at run time, see Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.

- **Reply**— At run time, an immediate reply is sent to the invoker of the loanGateway3 proxy service (see Figure 3-3) indicating that the message had a fault The reply is `With Failure`.

For configuration information, see "Error Messages and Handling" in Proxy Services: Error Handlers in *Using the AquaLogic Service Bus Console*.

Using Dynamic Routing

When you do not know the service you need to invoke from the proxy service you are creating, you can use dynamic routing.

For any given proxy service, you can use one of the following techniques to dynamically route messages:

- In a message flow pipeline, design an XQuery expression to dynamically set the fully qualified service name in ALSB and use the dynamic route or dynamic publish actions.

  Note: Dynamic Routing can be achieved in a route node, whereas dynamic publishing can achieved in a stage in a request pipeline or a response pipeline.

  With this technique, the proxy service dynamically uses the service account of the endpoint business service to send user names and passwords in its outbound requests. For example, if a proxy service is routing a request to Business Service A, then the proxy service uses the service account from Business Service A to send user names and passwords in its outbound request. See "Implementing Dynamic Routing" on page 3-36.

- Configure a proxy service to route or publish messages to a business service. Then, in the request actions section for the route action or publish action, add a Routing Options action that dynamically specifies the URI of a service.

  With this technique, to send user names and passwords in its outbound requests, the proxy service uses the service account of the statically defined business service, regardless of the URI to which the request is actually sent.

  For information on how to use this technique, see "Implementing Dynamic Routing" on page 3-36.

  Note: This technique is used when the overview of the interface is fixed. The overview of the interface includes message types, port types, and binding, and excludes the concrete interface. The concrete interface is the transport URL at which the service is located.

Implementing Dynamic Routing

You can use dynamic routing to determine the destination during the runtime of a proxy service. To achieve this you can use a routing table in an XML file to create an XQuery resource.

Note: Instead of using the XQuery resource, you can also directly use the XML file from which the resource is created.

An XML file or the Xquery resource can be maintained easily. At runtime you provide the entry in the routing table that will determine the routing or publishing destination of the proxy service.The XML file or the XQuery resource contains a routing table, which maps a logical identifier to (such as the name of a company) to the physical identifier (the fully qualified name of the service in ALSB). The logical identifier, which is extracted from the message, maps on to the physical identifier, which is the name of the service you want to invoke.

Note: To use the dynamic route action, you need the fully qualified name of the service in ALSB.

In a pipeline the logical identifier is obtained with an XPath into the message.You assign the XML table in the XQuery resource to a variable. You implement a query against the variable in the routing table to extract the physical identifier based on the corresponding logical identifier. Using this variable you will be able to invoke the required service. The following sections describe how to implement dynamic routing.

- "Sample XML File" on page 3-36.
- "Creating an XQuery Resource From the Sample XML" on page 3-37
- "Creating and Configuring the Proxy Service to Implement Dynamic Routing" on page 3-37

Sample XML File

You can create an XQuery resource from the following XML file. Save this as sampleXquery.xml.

Listing 3-16 Sample XML File

```
<routing>
   <row>
      <logical>BEA Systems</logical>
```

```
      <physical>default/goldservice</physical>
   </row>
   <row>
      <logical>ABC Corp</logical>
      <physical>default/silverservice</physical>
   </row>
</routing>
```

Creating an XQuery Resource From the Sample XML

1. In an active session, select **Project Explorer** from the left navigation panel. The **Project View** page is displayed.
2. Select the project to which you want to add the XQuery resource.
3. In the **Project View** page, select the XQuery resource from the **Select Resource Type** drop-down list. The **Create XQuery** page is displayed.
4. In the **Resource Name** field, enter the name of the resource. This is a mandatory.
5. In the **Resource Description** field, provide the a description for the resource. This is optional.
6. In the XQuery field, provide the path to the XML you are using as an XQuery resource. Click on **Browse** to locate the file. Optionally, you can copy and paste the XML in the XQuery field. This is mandatory.
7. Save the XQuery resource.
8. Activate the session.

Creating and Configuring the Proxy Service to Implement Dynamic Routing

1. In an active session, select **Project Explorer** from the left navigation panel. The **Project View** page is displayed.
2. Select the project to which you want to add the proxy service.
3. In the **Project View** page, select the **Proxy Service** resource from the **Select Resource Type** drop-down list. The **General Configuration** page is displayed.

4. In the **Service Name** field of the **General Configuration** page, enter the name of the proxy service. This is mandatory.
5. Select the type of service by clicking on the button adjacent to various types of services available under **Service Type**. For more information on selecting the service type, see Proxy Services: Actions.
6. Click **Finish**. On the **Summary** page, click **Save** to save the proxy service.
7. On the **Project View** page, click the Edit Message Flow icon against the newly created proxy service in the **Resource** table. The **Edit Message Flow** page is displayed.
8. Click on the message flow to add a pipeline pair to the message flow.
9. Click on **Request Pipeline** icon select **Add Stage** from the menu.
10. Click on the Stage1 icon to and select **Edit Stage** from the menu. The **Edit Stage Configuration** page appears.
11. Click **Add Action** icon. Choose **Add an Action** item from the menu.
12. Choose the **Assign** action from **Message Processing**.
13. Click on **Expression**. The **XQuery Expression Editor** is displayed.
14. Click on **XQuery Resources**. The browser displays the page where you can import the XQuery resource. Click on the **Browse** to locate the XQuery resource.
15. Click on **Validate** to validate the imported XQuery resource.
16. Save the imported XQuery resource on successful validation.
17. On the **Edit Stage Configuration** page, enter the name of the variable in the field.

    This assigns the XQuery resource to this variable. The variable now contains the externalized routing table.
18. Click on the **Assign** action icon to add another assign action.

    Note: To do this repeat step 11 to step 13.
19. Enter the following Xquery:

```
<ctx: route>
<ctx: service isProxy='false'>
{$routingtable/row[logical/text()=$logicalidentifier]/physical/text()}
</ctx: service>
</ctx: route>
```

In the above code, replace `$logicalidentifier` by the actual XPath to extract the logical identifier from the message (example from `$body`).

20. Click on **Validate** to validate the Xquery.
21. Save the Xquery on successful validation.
22. On the **Edit Stage Configuration** page, enter the name of the variable (for example, *routeresult*) in the field.

    This extracts the XML used by the dynamic route action into this variable.
23. Click on the message flow to add a route node to the end of the message flow.
24. Click on the **Route Node** icon and select **Edit** from the menu.
25. Click the **Add Action** icon. Choose **Add an Action** item from the menu.
26. Choose the **Dynamic Route** action.
27. Click on **Expression**. The XQuery Expression Editor is displayed.
28. Enter the variable from step 22 (for example, *$routeresult*)

Accessing Databases Using XQuery

ALSB provides read-access to databases from proxy services without requiring you to write a custom EJB or custom Java code and without the need for a separate database product like AquaLogic Data Services Platform. You can use the `execute-sql()` function to make a simple JDBC call to a database to perform simple database reads. Any SQL query is legal, from a query that gets a single tax rate for the supplied location to a query that does a complex join to obtain an order's current status from several underlying database tables.

A database query can be used to get data for message enrichment, for routing decisions, or for customizing the behavior of a proxy service. Take for example a scenario in which an ALSB proxy service receives “request for quote” messages. The proxy service can route the requests based on the customer's priority to one of a number of quotation business services (say, standard, gold, or platinum level services). The proxy service can then perform a SQL-based augmentation of the results that those services return—for example, based on the selected ship method and the weight of the order, the shipping cost can be looked up and that cost added to the request for quote message.

"fn-bea:execute-sql()" on page 9-4 describes the syntax for the function and provides examples of its use. The `execute-sql()` function returns typed data and automatically translates values between SQL/JDBC and XQuery data models.

You can store the returned element in a user-defined variable in an ALSB message flow.

The following databases and JDBC drivers are supported using the `execute-sql()` function:

- IBM DB2/NT 8
- Microsoft SQL Server 2000, 2005
- Oracle 8.1.x
- Oracle 9.x, 10.x
- Pointbase 4.4, 5.x
- Sybase 12.5.2 and 12.5.3
- WebLogic Type 4 JDBC drivers
- Third-party drivers supported by WebLogic Server

Use non-XA drivers for datasources you use with the fn-bea:execute-sql() function—the function supports read-only access to the datasources.

WARNING: In addition to specifying a non-XA JDBC driver class to use to connect to the database, you must ensure that you disable global transactions and two-phase commit. (Global transactions are enabled by default in the WebLogic Server console for JDBC data sources.) These specifications can be made for your data source via the WebLogic Server Administration Console. See Create JDBC Data Sources in the WebLogic Server Administration Console *Online Help*.

For complete information about database and JDBC drivers support in ALSB, see Supported Database Configurations in *Supported Configurations for AquaLogic Service Bus*.

Databases other than the core set described in the preceding listing are also supported. However, for the core databases listed above, the XQuery engine does a better recognition and mapping of data types to XQuery types than it does for the non-core databases—in some cases, a core database's proprietary JDBC extensions are used when fetching data. For the non-core databases, the XQuery engine relies totally on the standard type codes provided by the JDBC driver and standard JDBC resultset access methods.

When designing your proxy service, you can enter XQueries inline as part of an action definition instead of entering them as resources. You can also use inline XQueries for conditions in

If...Then... actions in message flows. For information about using the inline XQuery editor, see "Creating Variable Structure Mappings" on page 3-48.

Understanding Message Context

The message context is a set of variables that hold message context and information about messages as they are routed through the ALSB. Together, the `header`, `body`, and `attachments` variables, (referenced as `$header`, `$body` and `$attachments` in XQuery statements) represent the message as it flows through ALSB. The canonical form of the message is SOAP. Even if the service type is not SOAP, the message appears as SOAP in the ALSB message context.

Message Context Components

In a Message Context, `$header` contains a SOAP header element and `$body` contains a SOAP Body element. The Header and Body elements are qualified by the SOAP 1.1 or SOAP 1.2 namespace depending on the service type of the proxy service. Also in a Message Context, `$attachments` contains a wrapper element called attachments with one child attachment element per attachment. The attachment element has a body element with the actual attachment.

When a message is received by a proxy service, the message contents are used to initialize the header, body, and attachments variables. For SOAP services, the Header and Body elements are taken directly from the envelope of the received SOAP message and assigned to `$header` and `$body` respectively. For non-SOAP services, the entire content of the message is typically wrapped in a Body element (qualified by the SOAP 1.1 namespace) and assigned to `$body`, and an empty Header element (qualified by the SOAP 1.1 namespace) is assigned to `$header`.

Binary and MFL messages are initialized differently. For MFL messages, the equivalent XML document is inserted into the Body element that is assigned to `$body`. For binary messages, the message data is stored internally and a piece of reference XML is inserted into the Body element that is assigned to `$body`. The reference XML looks like `<binary-content ref="..."/>`, where `"..."` contains a unique identifier assigned by the proxy service.

The message context is defined by an XML schema. You must use XQuery expressions to manipulate the context variables in the message flow that defines a proxy service.

The predefined context variables provided by ALSB can be grouped into the following types:

- Message-related variables
- Inbound and outbound variables
- Operation variable

- Fault variable

For information about the predefined context variables, see "Predefined Context Variables" on page 5-2.

The `$body contains` message payload variable. When a message is dispatched from ALSB you can decide the variables, whose you want to include in the outgoing message. That determination is dependent upon whether the target endpoint is expecting a SOAP or a non-SOAP message:

- For a binary, any text or XML message content inside the Body element in `$body` is sent.
- For MFL messages, the Body element in `$body` contains the XML equivalent of the MFL document.
- For text messages, the Body element in `$body` contains the text. For text attachments, the body element in `$attachments` contains the text. If the contents are XML instead of simple text, the XML is sent as a text message.
- For XML messages, the Body element in `$body` contains the XML. For XML attachments, the body element in `$attachments` contains the XML.
- SOAP messages are constructed by wrapping the contents of the header and body variables inside a `<soap:Envelope>` element. (The SOAP 1.1 namespace is used for SOAP 1.1 services, while the SOAP 1.2 namespace is used for SOAP 1.2 services.) If the body variable contains a piece of reference XML, it is sent.That is the referenced content is not substituted in the message.

For non-SOAP services, if the Body element of `$body` contains a binary-content element, then the referenced content stored internally is sent 'as is', regardless of the target service type.

For more information, see "Message Context" on page 5-1.

The types for the message context variables are defined by the message context schema (`MessageContext.xsd`). When working with the message context variables in the BEA XQuery Mapper, you need to reference `MessageContext.xsd`, which is available in a JAR file, *`BEA_HOME`*`/alsb_3.0/lib/sb-kernel-api.jar`, and the transport-specific schemas, which are available at

`BEA_HOME``/alsb_3.0/lib/transports/`

where *`BEA_HOME`* represents the directory in which you installed ALSB.

To learn about the message context schema and the transport specific schemas, see "Message Context Schema" on page 5-29.

Guidelines for Viewing and Altering Message Context

Consider the following guidelines when you want to inspect or alter the message context:

- In an XQuery expression, the root element in a variable is not present in the path in a reference to an element in that variable. For example, the following XQuery expression obtains the `Content-Description` of the first attachment in a message:

  ```
  $attachments/ctx:attachment[1]/ctx:content-Description
  ```

  To obtain the second attachment

  ```
  $attachments/ctx:attachment[2]/ctx:body/*
  ```

- A context variable can be empty or it can contain a single XML element or a string value. However, an XQuery expression often returns a sequence. When you use an XQuery expression to assign a value to a variable, only the first element in the sequence returned by the expression is stored as the variable value. For example, if you want to assign the value of a WS-Addressing Message ID from a SOAP header (assuming there is one in the header) to a variable named `idvar`, the assign action specification is:

  ```
  assign data($header/wsa:messageID to variable idvar
  ```

  Note: In this case, if two WS-Addressing MessageID headers exist, the `idvar` variable will be assigned the value of the first one.

- The variables `$header`, `$body`, and `$attachments` are never empty. However, `$header` can contain an empty SOAP Header element, `$body` can contain an empty SOAP Body element, and `$attachments` can contain an empty attachment element.

- In cases in which you use a transformation resource (XSLT or XQuery), the transformation resource is defined to transform the document in the SOAP body of a message. To make this transformation case easy and efficient, the input parameter to the transformation can be an XQuery expression. For example, you can use the following XQuery expression to feed the business document in the Body element of a message (`$body`) as input to a transformation:

  ```
  $body/* [1]
  ```

  The result of the transformation can be put back in `$body` with a replace action. That is replace the content of `$body`, which is the content of the Body element. For more information, see XQuery Transformations and XSL Transformations in *Using the AquaLogic Service Bus Console*.

- In addition to inserting or replacing a single element, you can also insert or replace a selected sequence of elements using an insert or replace action. You can configure an

XQuery expression to return a sequence of elements. For example, you can use insert and replace actions to copy a set of transport headers from `$inbound` to `$outbound`. For information on adding an action, see "Adding an Action" in Proxy Services: Actions in *Using the AquaLogic Service Bus Console*. For an example, see "Copying JMS Properties From Inbound to Outbound" on page 3-44.

Copying JMS Properties From Inbound to Outbound

It is assumed that the interfaces of the proxy services and of the invoked business service may be different. Therefore, ALSB does not propagate any information (such as the transport headers and JMS properties) from the inbound variable to the outbound variable.

The transport headers for the proxy service's request and response messages are in `$inbound` and the transport headers for the invoked business service's request and response are in `$outbound`.

For example, the following XQuery expression can be used in a case where the user-defined JMS properties for a one-way message (an invocation with no response) need to be copied from inbound message to outbound message:

Use the transport headers action to set

```
$inbound/ctx:transport/ctx:request/tp:headers/tp:user-header
```

as the first child of:

```
./ctx:transport/ctx:request/tp:headers
```

in the outbound variable.

To learn how to configure the transport header action, see:

- "Transport Headers" in Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.

Working with Variable Structures

The following sections describe

Using the Inline XQuery Expression Editor

ALSB allows you to import XQueries that have been created with an external tool such as the BEA XQuery Mapper. You can use these XQueries anywhere in the proxy service message flow by binding the XQuery resource input to an Inline XQuery, and binding the XQuery resource output to an action that uses the result as the input; for example, the assign, replace, or insert actions.

However, you can enter the XQuery inline as part of the action definition instead of entering the XQuery as a resource. You can also use Inline XQueries for the condition in an **If...Then...** action.

Use the Inline XQuery Expression Editor to enter simple XQueries that consist of the following:

- Fragments of XML with embedded XQueries.
- Simple variable paths along the child axis.

Note: For more complex XQueries, it is recommended that you use the XQuery Mapper, especially if you are not familiar with XQuery.

Inline XQueries can be used effectively to:

- Create variable structures by using the Inline XQuery Expression Editor. See "Using Variable Structures" on page 3-47.
- Extract or access a business document or RPC parameter from the SOAP envelope elements in `$header` or `$body`.
- Extract or access an attachment document in `$attachments`.
- Set up the parameters of a service callout action by extracting it from the SOAP envelope.
- Insert the result parameter of a service callout action into the SOAP envelope.
- Extract a sequence from the SOAP envelope to drive a `for loop`.
- Update an item in the sequence in a `for loop` with an Update action.

Note: You can also use the Inline XQuery Expression Editor to create variable structures. For more information, see "Using Variable Structures" on page 3-47.

Inline XQueries

ALSB allows you to import XQueries that have been created with an external tool such as the BEA XQuery Mapper. You can use these XQueries anywhere in the proxy service message flow by binding the XQuery resource input to an inline XQuery, and binding the XQuery resource

output to an action that uses the result as the action input; for example, the assign, replace, or insert actions. However, you can enter the XQuery inline as part of the action definition instead of entering the XQuery as a resource. You can also use inline XQueries for the condition in an **If...Then...** action.

The inline XQuery and XPath editors allow you to declare a variable's structure by mapping it to a type or element and then creating path expressions with a drag and drop action from the graphical representation of the structure. You can also enter the path expressions manually.

You can use this feature directly for all user-defined variables, as well as `$inbound`, `$outbound`, and `$fault`. However, you cannot use it directly to access XML attachments in `$attachments`, headers in `$header`, or documents and RPC parameters in `$body`, with one exception— you can use it directly to access documents and parameters in `$body` for request messages received by a WSDL proxy service.

To learn more about creating variable structures, see "Creating Variable Structure Mappings" on page 3-48.

To learn more about XQuery engine support and the relationship with the DSP functions and operators, see "XQuery Implementation" on page 9-1.

Uses of the Inline XQuery Expression Editor

You typically use the Inline XQuery Expression Editor to enter simple XQueries that consist of the following:

- Fragments of XML with embedded XQueries.
- Simple variable paths along the child axis.

Note: For more complex XQueries, we recommend that you use the BEA XQuery Mapper, an editor with drag-and-drop functionality. See Transforming Data Using the XQuery Mapper in *Transforming Data Using the XQuery Mapper*.

Examples of good uses of inline XQueries are:

- Extract or access a business document or RPC parameter from the SOAP envelope elements in `$header` or `$body`.
- Extract or access an attachment document in `$attachments`.
- Set up the parameters of a service callout by extracting it from the SOAP envelope.
- Fold the result parameter of a service callout into the SOAP envelope.

- Extract a sequence from the SOAP envelope to drive a `for loop`.
- Update an item in the sequence in a `for loop` with an Update action.

You can also use the Inline XQuery Expression Editor to create variable structures. For more information, see "Using Variable Structures" on page 3-47.

Using Variable Structures

You can use the Inline XQuery Expression Editor to create variable structures, with which you define the structure of a given variable for design purposes. For example, it is easier to browse the XPath variable in the console rather than viewing the XML schema of the XPath variable.

Note: It is not necessary to create variable structures for your runtime to work. Variable structures define the structure of the variable or the variable path but do not create the variable. Variables are created at runtime as the target of the assign action in the stage.

In a typical programming language, the scope of variables is static. Their names and types are explicitly declared. The variable can be accessed anywhere within the static scope.

In ALSB, there are some predefined variables, but you can also dynamically create variables and assign value to them using the assign action or using the loop variable in the for-loop. When a value is assigned to a variable, the variable can be accessed anywhere in the proxy service message flow. The variable type is not declared but the type is essentially the underlying type of the value it contains at any point in time.

Note: The scope of the for-loop variable is limited and cannot be accessed outside the stage.

When you use the Inline XQuery Expression Editor, the XQuery has zero or more inputs and one output. Because you can display the structure of the inputs and the structure of the output visually in the Expression Editor itself, you do not need to open the XML schema or WSDL resources to see their structure when you create the Inline XQuery. The graphical structure display also enables you to drag and drop simple variable paths along the child axis without predicates, into the composed XQuery.

Each variable structure mapping entry has a label and maps a variable or variable path to one or more structures. The scope of these mappings is the stage or route node. Because variables are not statically typed, a variable can have different structures at different points (or at the same point) in the stage or route node. Therefore, you can map a variable or a variable path to multiple structures, each with a different label. To view the structure, select the corresponding label with a drop-down list.

Note: You can also create variable structure mappings in the Inline XPath Expression Editor. However, although the variable or a variable path is mapped to a structure, the XPaths

generated when you select from the structure are XPaths relative to the variable. An example of a relative XPath is `./ctx:attachment/ctx:body`.

Creating Variable Structure Mappings

The following sections describe how to create several types of variable structure mappings:

Sample WSDL

This sample WSDL is used in most of the examples in this section. You need to save this WSDL as a resource in your configuration. For more information, see "Creating the Resources You Need for the Examples" on page 3-50.

Listing 3-17 Sample WSDL

```
<definitions
    name="samplewsdl"
    targetNamespace="http://example.org"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:s0="http://www.bea.com"
    xmlns:s1="http://example.org"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
```

```
<types>
  <xs:schema
    attributeFormDefault="unqualified"
    elementFormDefault="qualified"
    targetNamespace="http://www.bea.com"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="PO" type="s0:POType"/>
    <xs:complexType name="POType">
      <xs:all>
        <xs:element name="id" type="xs:string"/>
        <xs:element name="name" type="xs:string"/>
      </xs:all>
    </xs:complexType>
    <xs:element name="Invoice" type="s0:InvoiceType"/>
    <xs:complexType name="InvoiceType">
      <xs:all>
        <xs:element name="id" type="xs:string"/>
        <xs:element name="name" type="xs:string"/>
      </xs:all>
    </xs:complexType>
</xs:schema>
</types>
<message name="POTypeMsg">
    <part name="PO" type="s0:POType"/>
</message>
<message name="InvoiceTypeMsg">
    <part name="InvReturn" type="s0:InvoiceType"/>
</message>

<portType name="POPortType">
    <operation name="GetInvoiceType">
      <input message="s1:POTypeMsg"/>
      <output message="s1:InvoiceTypeMsg"/>
    </operation>
</portType>
<binding name="POBinding" type="s1:POPortType">
<soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
```

```
    <operation name="GetInvoiceType">
      <soap:operation soapAction="http://example.com/GetInvoiceType"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
    </output>
  </operation>
</binding>
</definitions>
```

Creating the Resources You Need for the Examples

To make use of the examples that follow, you save the sample WSDL as a resource in your configuration and create the sample business service and proxy service that use the sample WSDL.

The instructions that follow tell how to accomplish the tasks in the ALSB Console:

- "Save the WSDL as a Resource" on page 3-50
- "Create a Proxy Service That Uses the Sample WSDL" on page 3-51
- "Build a Message Flow for the Sample Proxy Service" on page 3-52
- "Create a Business Service That Uses the Sample WSDL" on page 3-52

Save the WSDL as a Resource

1. In the left navigation pane in the ALSB Console, under **Change Center**, click **Create** to create a new session for making changes to the current configuration.
2. In the left navigation pane, click **Project Explorer**.
3. In the **Project View** page, click the project to which you want to add the WSDL.
4. In the **Project View** page, in the **Create Resource** field, select **WSDL** under **Interface**.
5. In the **Create a New WSDL Resource** page in the **Resource Name** field, enter `SampleWSDL`. This is a required field.

6. In the **WSDL** field, copy and paste the text from the sample WSDL into this field.

   Note: This is a required field.

7. Click **Save**. The new WSDL `SampleWSDL` is included in the list of resources and saved in the current session. You must now create a proxy service that uses this WSDL, see "Create a Proxy Service That Uses the Sample WSDL" on page 3-51.

Create a Proxy Service That Uses the Sample WSDL

1. In the left navigation pane, click **Project Explorer**.
2. In the **Project View** page, select the project to which you want to add the proxy service.
3. In the **Project View** page, in the **Create Resource** field, select **Proxy Service** under **Service**.
4. In the **Edit a Proxy Service - General Configuration** page, in the **Service Name** field, enter `ProxywithSampleWSDL`. This is a required field.
5. In the **Service Type** field, which defines the types and packaging of the messages exchanged by the service:
   a. Select **WSDL Web Service** from under **Create a New Service**.
   b. Click **Browse**. The **WSDL Browser** is displayed.
   c. Select `SampleWSDL`, then select `POBinding` in the **Select WSDL Definitions** pane.
   d. Click **Submit**.
6. Keep the default values for all other fields on the **General Configuration** page, then click **Next**.
7. Keep the default values for all fields on the **Transport Configuration** pages, then click **Next**.
8. In the **Operation Selection Configuration** page, make sure **SOAP Body Type** is selected in the **Selection Algorithm** field, then click **Next**.
9. Review the configuration data that you have entered for this proxy service, then click **Save**. The new proxy service `ProxywithSampleWSDL` is included in the list of resources and saved in the current session.To build message flow for this proxy service, see "Build a Message Flow for the Sample Proxy Service" on page 3-52.

Build a Message Flow for the Sample Proxy Service

1. In the **Project View** page, in the **Actions** column, click the **Edit Message Flow** icon for the `ProxywithSampleWSDL` proxy service.
2. In the **Edit Message Flow** page, click the `ProxywithSampleWSDL` icon, then click **Add Pipeline Pair**. **PipelinePairNode1** is displayed, which includes request and response pipelines.
3. Click the **Request Pipeline** icon, then click **Add Stage**. The Stage **Stage1** is displayed.
4. Click **Save**. The basic message flow is created for the `ProxywithSampleWSDL` proxy service.

Create a Business Service That Uses the Sample WSDL

1. In the left navigation pane, click **Project Explorer**. The **Project View** page is displayed.
2. Select the project to which you want to add the business service.
3. From the **Project View** page, in the **Create Resource** field, select **Business Service** from under **Service**. The **Edit a Business Service - General Configuration** page is displayed.
4. In the **Service Name** field, enter `BusinesswithSampleWSDL`. This is a required field.
5. In the **Service Type** field, which defines the types and packaging of the messages exchanged by the service, do the following:
   a. Select **WSDL Web Service** from under **Create a New Service**.
   b. Click **Browse**. The **WSDL Browser** is displayed.
   c. Select `SampleWSDL`, then select **`POBinding`** in the Select **WSDL Definitions** pane.
   d. Click **Submit**.
6. Keep the default values for all other fields on the **General Configuration** page, then click **Next**.
7. Enter an endpoint URI in the **Endpoint URI** field on the **Transport Configuration** page. Click **Add**, and then click **Next**.
8. Use the default values for all fields on the **SOAP Binding Configuration** page. Click **Next**.

9. Review the configuration data that you have entered for this business service, and then click **Save**. The new business service `BusinesswithSampleWSDL` is included in the list of resources and is saved in the current session.
10. From the left navigation pane, click **Activate** under **Change Center**. The session ends and the configuration is deployed to run time. You are now ready to use the examples—continue in "Example 1: Selecting a Predefined Variable Structure" on page 3-53.

Example 1: Selecting a Predefined Variable Structure

In this example, you select a predefined variable structure using the proxy service `ProxyWithSampleWSDL`, which has a service type **WSDL Web Service** that uses the binding **`POBinding`** from `SampleWSDL`.

The proxy service message flow needs to know the structure of the message in order to manipulate it. To achieve this, ALSB automatically provides a predefined structure that maps the **body** variable to the SOAP body structure as defined by the WSDL of the proxy service for all the messages in the interface. This predefined structure mapping is labeled **body**.

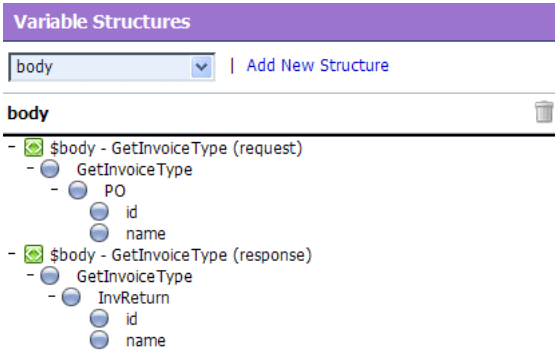
Note: This predefined structure is also supported for messaging services with a typed interface.

To select a predefined variable structure:

In the Variable Structures panel on the XQuery Expression Editor page, select **body** from the drop-down list of built-in structures.

The variable structure **body** is displayed in Figure 3-4.

Figure 3-4 Variable Structures—body



Example 2: Creating a Variable Structure That Maps a Variable to a Type

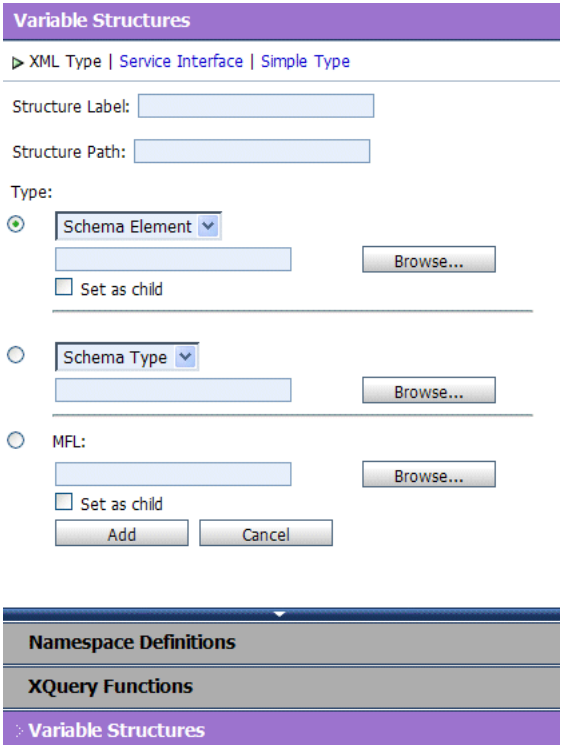
Suppose the proxy service `ProxyWithSampleWSDL` invokes a service callout to the business service `BusinessWithSampleWSDL`, which also has a service type **WSDL Web Service** that uses the binding **`POBinding`** from `SampleWSDL`. The operation `GetInvoiceType` is invoked.

In this example, the message flow needs to know the structure of the response parameter in order to manipulate it. To achieve this, you can create a new variable structure that maps the response parameter variable to the type `InvoiceType`.

To map a variable to a type:

1. In the Variable Structures panel, click **Add New Structure**. Additional fields are displayed in Figure 3-5.

Figure 3-5 Variable Structures—Add a New Structure

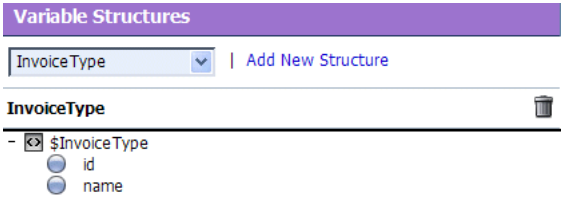


2. Select the **XML Type**.

3. In the **Structure Label** field, enter `InvoiceType` as the display name for the variable structure you want to create. This display name enables you to give a meaningful name to the structure so you can recognize it at design time but it has no impact at run time.
4. In the **Structure Path** field, enter `$InvoiceType` as the path of the variable at run time.
5. To select the type **InvoiceType**, do the following:
   a. Under the **Type** field, select the appropriate radio button, then select **WSDL Type** from the drop-down list.
   b. Click **Browse**. The **WSDL Browser** is displayed.
   c. In the **WSDL Browser**, select **SampleWSDL**, then select **InvoiceType** under **Types** in the **Select WSDL Definitions** pane.
   d. Click **Submit**. **InvoiceType** is displayed under your selection **WSDL Type**.
6. Click **Add**. The new variable structure **InvoiceType** is included under **XML Type** in the drop-down list of variable structures.

   The variable structure **InvoiceType** is displayed in Figure 3-6.

Figure 3-6 Variable Structures—InvoiceType



Example 3: Creating a Variable Structure that Maps a Variable to an Element

Suppose a temporary variable has the element **Invoice** described in the `SampleWSDL` WSDL. In this example, the `ProxyWithSampleWSDL` message flow needs to access this variable. To achieve this, you can create a new variable structure that maps the variable to the element **Invoice**.

To map a variable to an element:

1. In the Variable Structures panel, click **Add New Structure**.
2. Make sure you select the **XML Type**.

3. In the **Structure Label** field, enter `Invoice` as the meaningful display name for the variable structure you want to create.
4. In the **Structure Path** field, enter `$Invoice` as the path of the variable structure at run time.
5. To select the element **Invoice**, do the following:
   a. For the **Type** field, make sure you select the appropriate radio button.Then select **WSDL Element** from the drop-down list.
   b. Click **Browse**.
   c. In the **WSDL Browser**, select `SampleWSDL`, then select **Invoice** under **Elements** in the **Select WSDL Definitions** pane.
   d. Click **Submit**. **Invoice** is displayed under your selection **WSDL Element**.
6. Click **Add**. The new variable structure **Invoice** is included under **XML Type** in the drop-down list of variable structures.

   The variable structure **Invoice** is displayed in Figure 3-7.

Figure 3-7 Variable Structures—Invoice

Example 4: Creating a Variable Structure That Maps a Variable to a Child Element

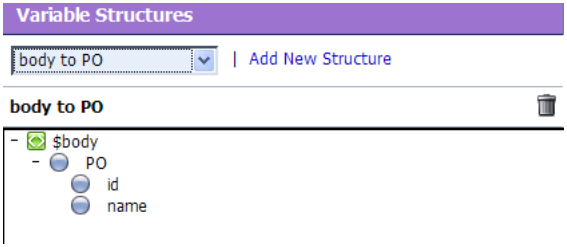
The `ProxyWithSampleWSDL` proxy service routes to the document style `Any SOAP` business service that returns the Purchase Order in the `SOAP` body. In this example, the `ProxyWithSampleWSDL` proxy service message flow must then manipulate the response. To achieve this, you can create a new structure that maps the **body** variable to the **`PO`** element, and specify the **PO** element as a child element of the variable. You need to specify it as a child element because the **body** variable contains the SOAP Body element and the **`PO`** element is a child of the Body element.

To map a variable to a child element:

1. In the Variable Structures panel, click **Add New Structure**.
2. Make sure you select the **XML Type**.
3. In the **Structure Label** field, enter `body to PO` as the meaningful display name for the variable structure you want to create.
4. In the **Structure Path** field, enter `$body` as the path of the variable structure at run time.
5. To select the **PO** element:
   a. Under the **Type** field, make sure you select the appropriate radio button, and then select **WSDL Element** from the drop-down list.
   b. Click **Browse**.
   c. In the **WSDL Browser**, select `SampleWSDL`, then select **PO** under **Elements** in the **Select WSDL Definitions** pane.
   d. Click **Submit**.
6. Select the **Set as child** checkbox to set the **PO** element as a child of the **body to PO** variable structure.
7. Click **Add**. The new variable structure **body to PO** is included under **XML Type** in the drop-down list of variable structures.

   The variable structure **body to PO** is displayed in Figure 3-8.

Figure 3-8 Variable Structures—body to PO



Example 5: Creating a Variable Structure that Maps a Variable to a Business Service

The `ProxyWithSampleWSDL` proxy service routes the message to the `BusinessWithSampleWSDL` business service, which also has a service type **WSDL Web Service** that uses the binding **`POBinding`** from `SampleWSDL`. In this example, the message flow must then

manipulate the response. To achieve this, you can define a new structure that maps the **body** variable to the `BusinessWithSampleWSDL` business service. This results in a map of the **body** variable to the SOAP body for all the messages in the WSDL interface of the service.

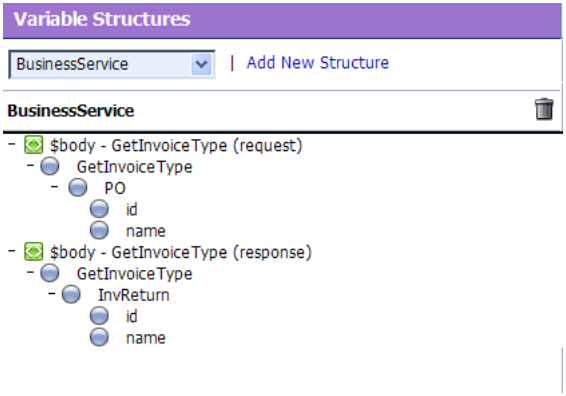
Note: This mapping is also supported for messaging services with a typed interface.

To map a variable to a business service:

1. In the Variable Structures panel, click **Add New Structure**.
2. Select **Service Interface**.
3. In the **Structure Label** field, enter `BusinessService` as the meaningful display name for the variable structure.
4. In the **Structure Path** field, **`$body`** is already set as the default. This is the path of the variable structure at run time.
5. To select the business service, do the following:
   a. Under the **Service** field, click **Browse**. The **Service Browser is displayed**.
   b. In the **Service Browser**, select the `BusinessWithSampleWSDL` business service, then click **Submit**. The business service is displayed under the **Service** field.
   c. In the **Operation** field, select **All**.
6. Click **Add**. The new variable structure `BusinessService` is included under **Service Interface** in the drop-down list of variable structures.

   The variable structure `BusinessService` is displayed in Figure 3-9.

Figure 3-9 Variable Structures—Business Service



Example 6: Creating a Variable Structure That Maps a Child Element to Another Child Element

Modify the `SampleWSDL` so that the `ProxyWithSampleWSDL` proxy service receives a single attachment. The attachment is a Purchase Order. In this example, the proxy service message flow must then manipulate the Purchase Order. To achieve this, you can define a new structure that maps the **body** element in **`$attachments`** to the **`PO`** element, which is specified as a child element. The **body** element is specified as a variable path of the form:

```
$attachments/ctx:attachment/ctx:body
```

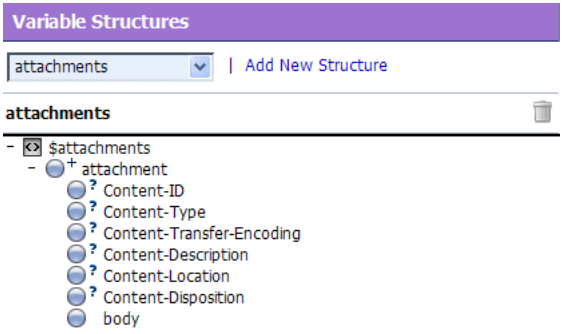
You can select and copy the **body** element from the predefined **attachments** structure, paste this element as the variable path to be mapped in the new mapping definition.

To map a child element to another child element:

1. In the Variable Structures panel, select **attachments** from the drop-down list of built-in structures.

   The variable structure **attachments** is displayed in Figure 3-10.

Figure 3-10 Variable Structures—attachments



2. Select the **body** child element in the attachments structure. The variable path of the body element is displayed in the Property Inspector on the right side of the page:

   `$attachments/ctx:attachment/ctx:body`

3. Copy the variable path of the **body** element.
4. In the Variable Structures panel, click **Add New Structure**.
5. Select the **XML Type**.
6. In the **Structure Label** field, enter **`PO`** `attachment` as the meaningful display name for this variable structure.
7. In the **Structure Path** field, paste the variable path of the body element:

   `$attachments/ctx:attachment/ctx:body`

   This is the path of the variable structure at run time.

8. To select the **`PO`** element:

   a. Under the **Type** field, make sure the appropriate radio button is selected, then select **WSDL Element**.

   b. Click **Browse**.

   c. In the **WSDL Browser**, select `SampleWSDL`, then select **`PO`** under **Elements** in the **Select WSDL Definitions** pane.

   d. Click **Submit**.

9. Select the **Set as child** checkbox to set the PO element as a child of the **body** element.

10. Click **Add**. The new variable structure **`PO` attachment** is included under **XML Type** in the drop-down list of variable structures.
11. If there are multiple attachments, add an index to the reference when you use fields from this structured variable in your XQueries. For example, if you drag the PO field to the XQuery field, but as PO will be the second attachment, change the inserted value from

    ```
    $attachments/ctx:attachment/ctx:body/bea:PO/bea:id
    ```

    to

    ```
    $attachments/ctx:attachment[2]/ctx:body/bea:PO/bea:id
    ```

Quality of Service

The following sections discuss quality of service features in ALSB messaging:

Delivery Guarantees

ALSB supports reliable messaging. When messages are routed to another service from a route node, the default quality of service (QoS) is *exactly once* if the proxy service transport is defined as JMS/XA; otherwise *best effort* QoS is supported.

Quality of service is set in the `qualityOfService` element in the `$outbound` context variable.

The following delivery guarantee types are provided in ALSB, shown in Table 3-9.

Table 3-9 Delivery Guarantee Types

Delivery Reliability	Description
Exactly once	*Exactly once* reliability means that messages are delivered from inbound to outbound exactly once, assuming a terminating error does not occur before the outbound message send is initiated. *Exactly once* means reliability is optimized. *Exactly once* delivery reliability is a hint, not a directive. When *exactly once* is specified, *exactly once* reliability is provided if possible. If *exactly once* is not possible, then *at least once* delivery semantics are attempted; if that is not possible, *best effort* delivery is performed. The default value of the `qualityOfService` element is `exactly-once` for a route node action for the following inbound transports: • e-mail • FTP • File • JMS/XA • SFTP • Transactional Tuxedo **Note:** Do not retry the outbound transport when the `QoS` is exactly once

Table 3-9 Delivery Guarantee Types

Delivery Reliability	Description
At least once	*At least once* semantics means the message is delivered to the outbound from the inbound at least once, assuming a terminating error does not occur before the outbound message send is initiated. Delivery is considered satisfied even if the target service responds with a transport-level error. However it is not satisfied in the case of a time-out, a failure to connect, or a broken communication link. If failover URLs are specified, *at least once* semantics is provided with respect to at least one of the URLs. *At least once* delivery semantics is attempted if *exactly once* is not possible but the `qualityOfService` element is `exactly-once`.
Best effort	*Best effort* means that there is no reliable messaging and there is no elimination of duplicate messages—however, performance is optimized. It is performed if the `qualityOfService` element is `best-effort`. *Best effort* delivery is also performed if *exactly once* and *at least once* delivery semantics are not possible but the `qualityOfService` element is `exactly-once`. The default value of the `qualityOfService` element for a route node is `best-effort` for the following inbound transports: • HTTP • JMS/nonXA • Non-Transactional Tuxedo The default value of the `qualityOfService` element is always `best-effort` for the following: • Service callout action — always `best-effort`, but can be changed if required. • Publish action — defaults to `best-effort`, modifiable **Note:** When the value of the `qualityOfService` element is `best-effort` for a publish action, all errors are ignored. However, when the value of the `qualityOfService` element is `best-effort` for a route node action or a Service callout action, any error will raise an exception.

For more detailed information about quality of service for other transports, see the documentation for the transport, at AquaLogic Service Bus Transports.

Overriding the Default Element Attribute

To override the default *exactly once* quality of service attribute, you must set the `qualityOfService` in the outbound message context variable (`$outbound`). For more information, see "Message Context Schema" on page 5-29.

You can override the default `qualityOfService` element attribute for the following:

- Route node action
- Publish action
- Service callout

To override the `qualityOfService` element attribute, you must use the route options action to route or publish, and also select the checkbox for a service callout action. See "Message Context Schema" on page 5-29.

Delivery Guarantee Rules

The delivery guarantee supported when a proxy service publishes a message or routes a request to a business service depends on the following conditions:

- The value of the `qualityOfService` element.
- The inbound transport (and connection factory, if applicable).
- The outbound transport (and connection factory, if applicable).

However, if the inbound proxy service is a Local Transport and is invoked by another proxy service, the inbound transport of the invoking proxy service is responsible for the delivery guarantee. That is because a proxy service that invokes another proxy service is optimized into a direct invocation if the transport of the invoked proxy service is a Local Transport. For more information on transport protocols, see Proxy Services and Business Services in *Using the AquaLogic Service Bus Console*.

Note: No delivery guarantee is provided for responses from a proxy service.

The following rules govern delivery guarantees, shown in Table 3-10.

Table 3-10 Delivery Guarantee Rules

Delivery Guarantee Provided	Rule
Exactly once	The proxy service inbound transport is transactional and the value of the `qualityOfService` element is `exactly-once` to an outbound JMS/XA transport.
At least once	The proxy service inbound transport is file, FTP, or e-mail and the value of the `qualityOfService` element is `exactly-once`.
At least once	The proxy service inbound transport is transactional and the value of the `qualityOfService` element, where applicable, is `exactly-once` to an outbound transport that is not transactional.
No delivery guarantee	All other cases, including all response processing cases.

Note: To support *at least once* and *exactly-once* delivery guarantees with JMS, you must exploit JMS transactions and configure a retry count and retry interval on the JMS queue to ensure that the message is redelivered in the event of a server crash or a failure that is not handled in an error handler with a *Reply* or *Resume* action. File, FTP, and e-mail transports also internally use a JMS/XA queue. The default retry count for a proxy service with a JMS/XA transport is 1. For a list of the default JMS queues created by ALSB, see *AquaLogic Service Bus Deployment Guide*.

The following are additional delivery guarantee rules:

- If the transport of the inbound proxy service is File, FTP, e-mail, Transactional Tuxedo, or JMS/XA, the request processing is performed in a transaction.
  - When the `qualityOfService` element is set to `exactly-once`, any route node and publish actions executed in the request flow to a transactional destination are performed in the same transaction.
  - When the `qualityOfService` element is set to `best-effort for` any action in a route node, service callout or publish actions are executed outside of the request flow transaction. Specifically, for JMS, Tuxedo, Transactional Tuxedo, or EJB transport, the request flow transaction is suspended and the Transactional Tuxedo work is done without a transaction or in a separate transaction that is immediately committed.
  - If an error occurs during request processing, but is caught by a user error handler that manages the error (by using the resume or reply action), the message is considered successfully processed and the transaction commits. A transaction is aborted if the

system error handler receives the error—that is, if the error is not handled before reaching the system level. The transaction is also aborted if a server failure occurs during request pipeline processing.

- If a response is received by a proxy service that uses a JMS/XA transport to business service (and the proxy inbound is not Transactional Tuxedo), the response processing is performed in a single transaction.
  - When the `qualityOfService` element is set to `exactly-once`, all route, service callout, and publish actions are performed in the same transaction.
  - When the `qualityOfService` element is set to `best-effort`, all publish actions and service callout actions are executed outside of the response flow transaction. Specifically, for JMS, EJB, or transactional Tuxedo types of transports, the response flow transaction is suspended and the service is invoked without a transaction or in a separate transaction that is immediately committed.
  - Proxy service responses executed in the response flow to a JMS/XA destination are always performed in the same transaction, regardless of the `qualityOfService` element setting.
- If the proxy service inbound transport is transactional Tuxedo, both the request processing and response processing are done in this transaction.

  Note: You will encounter a run-time error when the inbound transport is transactional Tuxedo and the outbound is an asynchronous transport, for example, JMS/XA.

Threading Model

The ALSB threading model works as follows:

- The request and response flows in a proxy service execute in different threads.
- Service callouts are always blocking. An HTTP route or publish action is non-blocking (for request/response or one-way invocation), if the value of the `qualityOfService` element is `best-effort`.
- JMS route actions or publish actions are always non-blocking, but the response is lost if the server restarts after the request is sent because ALSB has no persistent message processing state.

Note: In a request or response flow publish action, responses are always discarded because publish actions are inherently a one-way message send.

Splitting Proxy Services

You may want to split a proxy service in the following situations:

- When HTTP is the inbound and outbound transport for a proxy service, you may want to incorporate enhanced reliability into the middle of the message flow. To enable enhanced reliability in this way, split the proxy service into a front-end HTTP proxy service and a back-end JMS (one-way or request/response) proxy service with an HTTP outbound transport. In the event of a failure, the first proxy service must quickly place the message in the queue for the second proxy service, in order to avoid loss of messages.
- To disable the direct invocation optimization for a non-JMS transport when a proxy service, say `loanGateway1` invokes another proxy service, say `loanGateway2.` Route to the proxy service `loanGateway2` from the proxy service `loanGateway1` where the proxy service `loanGateway2` uses JMS transport.
- To have an HTTP proxy service publish to a JMS queue but have the publish action rollback if there is a exception later on in the request processing, split the proxy service into a front-end HTTP proxy service and a back-end JMS proxy service. The publish action specifies a `qualityOfService` element of `exactly-once` and uses an XA connection factory.

Outbound Message Retries

In addition to configuring inbound retries for messages using JMS, you can configure outbound retries and load balancing. Load balancing, failover, and retries work in conjunction to provide performance and high availability. For each message, the list of URLs you provide as failover URLs is automatically ordered based on the load balancing algorithm into a failover sequence. If the retry count is N, the entire sequence is retried N times before stopping. The system waits for the specified retry interval before commencing subsequent loops through the sequence. After completing the retry attempts, if there is still an error, the error handler pipeline for the route node is invoked. For more information on the error handler pipeline, see “Adding Pipeline Error Handling” in Proxy Services in *Using the AquaLogic Service Bus Console*.

Note: For HTTP transports, any HTTP status other than 200 or 202 is considered an error by ALSB and must be retried. Because of this algorithm, it is possible that ALSB retries errors like authentication failure that may never be rectified for that URL within the time period of interest. On the other hand, if ALSB also fails over to a different URL for subsequent attempts to send a given message, the new URL may not give the error.

For `quality of service=exactly once` failover or retries will not be executed.

Content Types, JMS Type, and Encoding

To support interoperability with heterogeneous endpoints, ALSB allows you to control the content type, the JMS type, and the encoding used.

ALSB does not make assumptions about what the external client or service needs, but uses the information configured for this purpose in the service definition. ALSB derives the content type for outbound messages from the service type and interface. Content type is a part of the e-mail and HTTP protocols.

If the service type is:

- XML or SOAP with or without a WSDL, the content type is text/XML.
- Messaging and the interface is MFL or binary, the content type is binary/octet-stream.
- Messaging and the interface is text, the content type is text/plain.
- Messaging and the interface is XML, the content type is text/XML.

Additionally, there is a JMS type, which can be byte or text. You configure the JMS type to use when you define the service in ALSB Console or in the ALSB Plug-in for WorkSpace Studio.

You can override the content type in the outbound context variable (`$outbound`) for proxy services invoking a service, and in the inbound context variable (`$inbound`) for a proxy service response. For more information on `$outbound` and `$inbound` context variables, see "Inbound and Outbound Variables" on page 5-10.

Encoding is also explicitly configured in the service definition for all outbound messages. For more information on service definitions, see Proxy Services in and Business Service in *Using the AquaLogic Service Bus Console*.

Throttling Pattern

In ALSB, you can restrict the message flow to a business service. This technique of restricting a message flow to a business service is known as throttling. For information, see Throttling in ALSB in the *AquaLogic Service Bus Operations Guide*.

WS-I Compliance

ALSB provides Web Service Interoperability (WS-I) compliance for SOAP 1.1 services in the run-time environment. The WS-I basic profile has the following goals:

- Disambiguate the WSDL and SOAP specifications wherever ambiguity exists.
- Define constraints that can be applied when receiving messages or importing WSDLs so that interoperability is enhanced. When messages are sent, construct the message so that the constraints are satisfied.

The WS-I basic profile is available at the following URL:

http://www.ws-i.org/Profiles/BasicProfile-1.1.html.

When you configure a proxy service or business service based on a WSDL, you can use the ALSB Console or the ALSB Plug-in for WorkSpace Studio to specify whether you want ALSB to enforce WS-I compliance for the service. For information on how to do this, see

- "Operation Selection Configuration page" under Proxy Services in *Using the AquaLogic Service Bus Console*
- "Proxy Service Operation Selection Configuration page" in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*

When you configure WS-I compliance for a proxy service, checks are performed on inbound request messages received by that proxy service. When you configure WS-I compliance for an invoked service, checks are performed when any proxy receives a response message from that invoked service. BEA recommends that you create an error handler for these errors, since by default, the proxy service SOAP client receives a system error handler-defined fault. For more information on creating fault handlers, see:

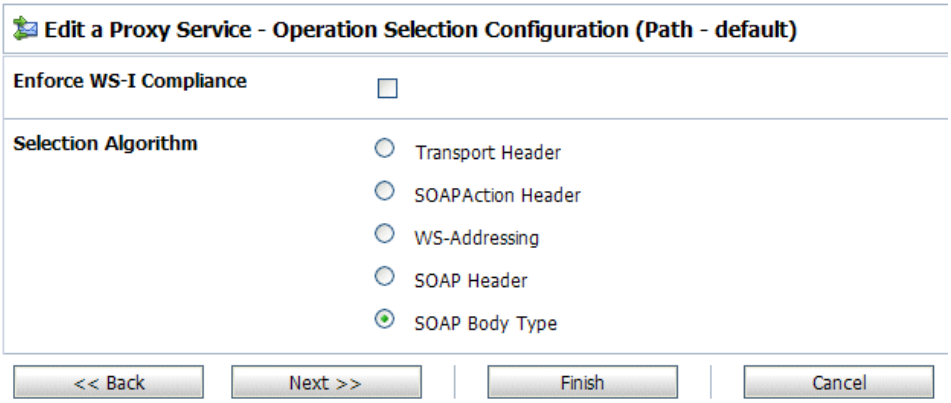
- Proxy Services: Error Handlers in *Using the AquaLogic Service Bus Console*.
- Adding and Configuring Error Handlers in Message Flows in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*.

For messages sent from a proxy service, whether as outbound request or inbound response, WS-I compliance checks are not explicitly performed. That is because the pipeline designer is responsible for generating most of the message content. However, the parts of the message generated by ALSB should satisfy all of the supported WS-I compliance checks. This includes the following content:

- Service invocation request message.
- System-generated error messages returned by a proxy service.
- HTTP status codes generated by a proxy service.

The Enforce WS-I Compliance checkbox is displayed as shown in Figure 3-11.

Figure 3-11 Enforce WS-I Compliance Checkbox



WS-I Compliance Checks

Note: WS-I compliance checks require that the system knows what operation is being invoked on a service. For request messages received by a proxy service, that means that the context variable `$operation` should not be null. That depends upon the operation selection algorithm being configured properly. For response messages received from invoked services, the operation should be specified in the action configurations for route, publish, and service callout.

When you configure WS-I compliance checking for a proxy service or a business service, ALSB carries out the following checks, shown in Table 3-11.

Table 3-11 ALSB WS-I Compliance Checks

Check	WS-I Basic Profile Details	ALSB Description
3.1.1 SOAP Envelope Structure	**R9980** An Envelope must conform to the structure specified in SOAP 1.1, Section 4, "SOAP Envelope" (subject to amendment).	This check applies to request and response messages. If a response message is checked and the message does not possess an outer `Envelope` tag, a `soap:client` error is generated. If the message is an `Envelope` tag but possesses a different namespace, it is handled by the 3.1.2 SOAP Envelope Namespace.
3.1.2 SOAP Envelope Namespace	**R1015** A Receiver must generate an error if they encounter an envelope whose document element is not `soap:Envelope`.	This check applies to request and response messages and is related to the 3.1.1 SOAP Envelope Structure. If a request message has a local name of `Envelope`, but the namespace is not SOAP 1.1, a `soap:VersionMismatch` error is generated.
3.1.3 SOAP Body Namespace Qualification	**R1014** The child elements of the `soap:body` element in an Envelope must be namespace qualified.	This check applies to request and response messages. All request error messages generate a `soap:Client` error.
3.1.4 Disallowed Constructs	**R1008** An Envelope must not contain a Document Type Declaration.	This check applies to request and response messages. All request error messages generate a `soap:Client` error.
3.1.5 SOAP Trailers	**R1011** An Envelope must not have any child elements of `soap:Envelope` following the `soap:body` element.	This check applies to request and response messages. All request error messages generate a `soap:Client` error.

Table 3-11 ALSB WS-I Compliance Checks

Check	WS-I Basic Profile Details	ALSB Description
3.1.9 SOAP attributes on SOAP 1.1 elements	**R1032** The `soap:Envelope`, `soap:header`, and `soap:body` elements in an Envelope must not have attributes in the namespace `http://schemas.xmlsoap.org/soap/envelope/`	This check applies to request and response messages. Any request error messages generate a `soap:client` error.
3.3.2 SOAP Fault Structure	**R1000** When an Envelope is a fault, the `soap:Fault` element must not have element children other than `faultcode`, `faultstring`, `faultactor`, and `detail`.	This check only applies to response messages.
3.3.3 SOAP Fault Namespace Qualification	**R1001** When an Envelope is a Fault, the element children of the `soap:Fault` element must be unqualified.	This check only applies to response messages.
3.4.6 HTTP Client Error Status Codes	**R1113** An instance should use a "`400 Bad Request`" HTTP status code if a HTTP request message is malformed. **R1114** An instance should use a "`405 Method not Allowed`" HTTP status code if a HTTP request message is malformed. **R1125** An instance must use a 4xx HTTP status code for a response that indicates a problem with the format of a request.	Only applies to responses for a proxy service where you cannot influence the status code returned due to errors in the request.
3.4.7 HTTP Server Error Status Codes	**R1126** An instance must return a "`500 Internal Server Error`" HTTP status code if the response envelope is a fault.	This check applies differently to request and response messages. For request messages, any faults generated have a `500 Internal Server Error` HTTP status code. For response messages, an error is generated if fault responses are received that do not have a `500 Internal Server Error` HTTP status code.

Table 3-11 ALSB WS-I Compliance Checks

Check	WS-I Basic Profile Details	ALSB Description
4.7.19 Response Wrappers	**R2729** An envelope described with an `rpc`-literal binding that is a response must have a wrapper element whose name is the corresponding `wsdl:operation` name suffixed with the string `Response`.	This check only applies to response messages. ALSB never generates a non-fault response from a proxy service.
4.7.20 Part Accessors	**R2735** An envelope described with an `rpc`-literal binding must place the part accessor elements for parameters and return value in no namespace. **R2755** The part accessor elements in a message described with an `rpc`-literal binding must have a local name of the same value as the name attribute of the corresponding `wsdl:part` element.	This check applies to request and response messages. Any request error messages generate a `soap:client` error.
4.7.22 Required Headers	**R2738** An envelope must include all `soapbind:headers` specified on a `wsdl:input` or `wsdl:output` of a `wsdl:operation` of a `wsdl:binding` that describes it.	This check applies to request and response messages. Any request error messages generate a `soap:client` error.
4.7.25 Describing SOAPAction	**R2744** A HTTP request message must contain a SOAPAction a HTTP header field with a quoted value equal to the value of the soapAction attribute of `soap:operation`, if present in the corresponding WSDL description. **R2745** A HTTP request message must contain a `SOAP` action a HTTP header field with a quoted empty string value, if in the corresponding WSDL description, the SOAPAction of `soapbind:operation` is either not present, or present with an empty string as its value.	This check applies to request messages and a `soap:client` error is returned.

Converting Between SOAP 1.1 and SOAP 1.2

ALSB supports SOAP 1.1 and SOAP 1.2. A SOAP 1.1 proxy service can invoke a SOAP 1.2 business service or vice versa. The SOAP namespace is automatically changed by ALSB before

invoking the business service. If a fault comes back from the business service it is automatically changed to the SOAP version of the proxy service. It is, however, up to the pipeline actions to map the SOAP header-related XML attributes (like `MustUnderstand`) between the two versions. It is also up to the pipeline actions to change the SOAP encoded name space for encoded envelopes.

CHAPTER 4

Improving Service Performance with Split-Join

ALSB's advanced mediation feature, called Split-Join, helps you improve service performance by concurrently processing individual messages in a request. This topic, which describes Split-Join, includes the following sections:

- Introduction to Split-Join
- Developing Split-Joins

Introduction to Split-Join

ALSB's Split-Join feature lets you split a service payload, such as an order, into individual messages for concurrent processing. Concurrent processing, as opposed to sequential processing, greatly improves service performance. Split-Join achieves this task by splitting an input message payload into sub messages (split), routing them concurrently to their destinations, and aggregating the responses into one overall return message (join). This process of payload splitting and response aggregation is called a Split-Join pattern.

Split-Joins are particularly useful for optimizing overall response times in scenarios where payloads delivered by faster systems are being directed to responding services on slower systems. Without Split-Join, individual messages in a payload are normally resolved in sequential order by the recipient, which can take a long time if the responding system is slow. (The overall response time is the sum of the individual response times for each message.) With Split-Join, multiple messages are processed simultaneously, which reduces burden on the responding system and greatly enhances response times. (The overall response time is roughly that of the longest individual message's response time plus some minor system overhead.)

There are two patterns supported by the Split-Join feature:

- Static Split-Join
- Dynamic Split-Join

Static Split-Join

The static Split-Join branches from the main execution thread of an ALSB message flow by splitting a payload into a fixed number of new branches according to the configuration of the Split-Join. At design time you determine the number and variety of services to be invoked.

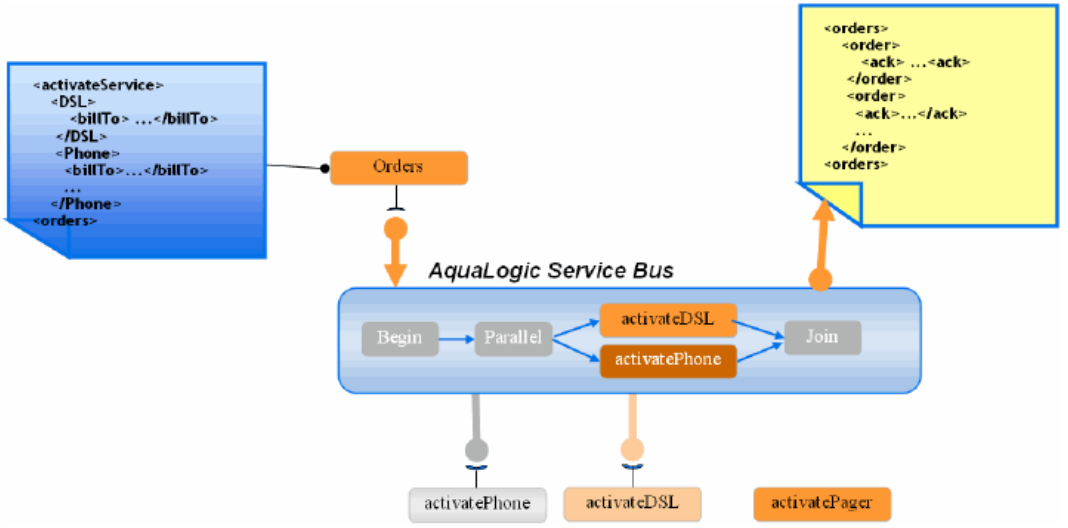
Static Split-Join – Sample Scenario

A telco company might want to employ static Split-Join when processing a customer's order for a communications services package. In this case, the customer might sign up for DSL and voice services all at once. Rather than executing each request in the payload separately in order, the telco can execute the messages in parallel using a callout from the ALSB message flow to a Split-Join employing the static Split-Join pattern.

Static Split-Join is the ideal pattern in this case because the developer knows there will always be exactly two incoming service requests for this particular service package: DSL and voice. Splitting the requests into parallel branches allows them to be processed concurrently, which improves the overall response time for processing the payload. After all messages have been processed, the generated responses are aggregated back into one reply in the execution thread.

Figure 4-1 illustrates a static Split-Join that splits two known service requests, DSL activation and phone activation, processes each request in parallel, and joins the responses into a single reply.

Figure 4-1 Static Split-Join – Known number of service requests



Dynamic Split-Join

The dynamic Split-Join branches from the main execution thread of an ALSB message flow by dynamically creating new branches according to the contents of the incoming payload. The dynamic Split-Join uses conditional logic to determine the number of branches to create. All requests are handled simultaneously, and the responses are aggregated into a single reply.

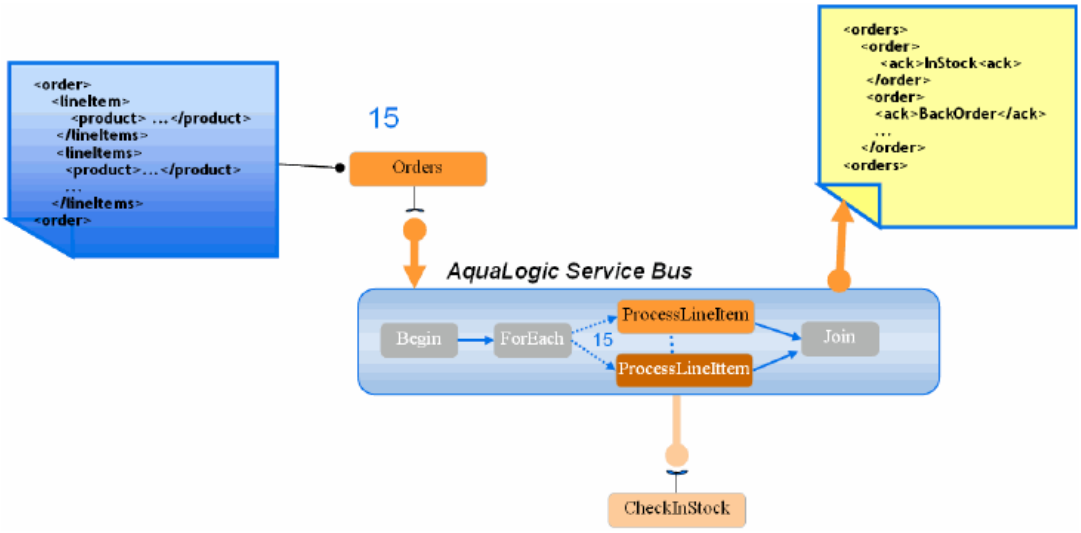
Dynamic Split-Join – Sample Scenario

A company might want to use dynamic Split-Join when placing automated stationery orders for its employees. If the orders are automatically placed every week based on employee submissions, there is no way of knowing how many individual orders are placed in any one weekly order. Rather than placing each order separately, the company could use dynamic Split-Join to place the orders concurrently using a callout from the ALSB message flow to a Split-Join employing the dynamic Split-Join pattern.

Dynamic Split-Join is the ideal pattern in this case, because the developer has no way of knowing how many orders will be submitted each week. The dynamic Split-Join loops through all the orders and places them in parallel. The developer can also limit the number of orders processed. After all of the orders have been processed, the generated order responses are aggregated back into one reply in the execution thread.

Figure 4-2 illustrates a dynamic Split-Join that splits 15 orders, processes them concurrently, and joins the responses into a single reply.

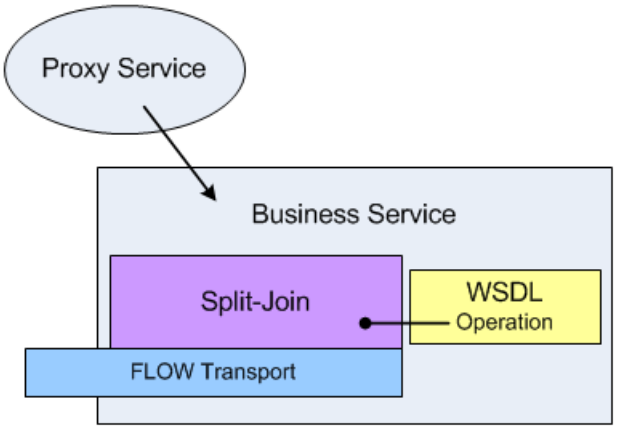
Figure 4-2 Dynamic Split-Join – Unknown number of service requests



Split-Join Framework

A Split-Join, which takes the form of a .flow file in WorkSpace Studio, is based on a WSDL operation. The Split-Join is wrapped in a WSDL-based business service that communicates across a FLOW transport, which is a dedicated transport for Split-Joins. The business service is invoked from a proxy service. Figure 4-3 illustrates the Split-Join framework.

Figure 4-3 Split-Join framework



Developing Split-Joins

You create and configure Split-Joins in WorkSpace Studio, then import them into the ALSB console for use in run-time configuration. For information on developing Split-Joins, see Working with Split-Join in the WorkSpace Studio help system.

Split-Join Resource Type and Environment Variable

If you reference Split-Joins in any scripts or custom code, use the values in Table 4-1:

Table 4-1 Split-Join resource type and environment variable

typeId	FLOW
Work manager environment value type	Split-Join Work Manager

CHAPTER 5

Message Context

This section describes the ALSB message context model and the predefined context variables that are used in message flows. It includes the following topics:

The Message Context Model

ALSB message context is a set of properties that hold message content as well as information about messages as they are routed through ALSB. These properties are referred to as context variables—for example, service endpoints are represented by predefined context variables. ALSB also supports user-defined context variables.

The message context is defined by an XML schema. You typically use XQuery expressions to manipulate the context variables in the message flow that defines a proxy service.

Predefined Context Variables

Table 5-1 describes the predefined context variables. The predefined context variables can be grouped into the following types: message-related variables, inbound and outbound variables, the operation variable, and the fault variable.

For information about the element types in the message context variables, see "Message Context Schema" on page 5-29.

Table 5-1 Predefined Context Variables in ALSB

Context Variable[1]	Description	See Also...
`header`	For SOAP messages, `header` contains the SOAP header. (If the proxy service is SOAP 1.2, `header` contains a SOAP 1.2 Header element.) For message types other than SOAP, `header` contains an empty SOAP header element.	"Message-Related Variables" on page 5-3
`body`	For the following cases: • SOAP messages—contains the `<SOAP:Body>` part extracted from the SOAP envelope. (If the proxy service is SOAP 1.2, the `body` variable contains a SOAP 1.2 Body element.) • Non-SOAP, non-binary messages—contains the entire message content wrapped in a `<SOAP:Body>` element. • Binary messages—contains a `<SOAP:Body>` wrapped reference to an in-memory copy of the binary message. • Java objects—contains a `<SOAP:Body>` wrapped reference to an in-memory copy of the Java object.	"Message-Related Variables" on page 5-3

Table 5-1 Predefined Context Variables in ALSB (Continued)

Context Variable[1]	Description	See Also...
`attachments`	Contains the MIME attachments for a given message.	"Message-Related Variables" on page 5-3
`inbound`	Contains: • Information about the proxy service that received a message • The inbound transport headers	"Inbound and Outbound Variables" on page 5-10
`outbound`	Contains: • Information about the target service to which a message is to be sent • The outbound transport headers	"Inbound and Outbound Variables" on page 5-10
`operation`	Identifies the operation that is being invoked on a proxy service.	"Operation Variable" on page 5-19
`fault`	Contains information about errors that have occurred during the processing of a message.	"Fault Variable" on page 5-19

1. The "Message Context Schema" on page 5-29 specifies the element types for the message context variables.

Message-Related Variables

Together, the message-related variables `header`, `body` and `attachments` represent the canonical format of a message as it flows through ALSB. These variables are initialized using the message content received by a proxy service and are used to construct the outgoing messages that are routed or published to other services.

If you want to modify a message as part of processing it, you must modify these variables.

A message payload (that is, a message content exclusive of headers or attachments) is contained in the `body` variable. The decision about which variable's content to include in an outgoing message is made at the point at which a message is dispatched (published or routed) from ALSB. That determination is dependent upon whether the target endpoint is expecting a SOAP or a non-SOAP message:

- When a SOAP message is expected, the `header` and `body` variables are combined in a SOAP envelope to create the message.

- When a non-SOAP message is expected, the contents of the Body element in the `body` variable constitutes the entire message.
- In either case, if the service expects attachments, a MIME package is created from the resulting message and the `attachments` variable.

Header Variable

The `header` variable contains SOAP headers associated with a message. The `header` variable points to a `<SOAP:Header>` element with headers as sub-elements. (If the proxy service is SOAP 1.2, the `header` variable contains a SOAP 1.2 Header element.) In the case of non-SOAP messages or SOAP messages with no headers, the `<SOAP:Header>` element is empty, with no sub-elements.

Body Variable

The `body` variable represents the core message payload and always points to a `<SOAP:Body>` element. (If the proxy service is SOAP 1.2, `body` contains a SOAP 1.2 Body element.) The core payload for both SOAP and non-SOAP messages is available in the same variable and with the same packaging—that is, wrapped in a `<SOAP:Body>` element:

- In the case of SOAP messages, the SOAP body is extracted from the envelope and assigned to the `body` variable.
- In the case of non-SOAP, non-binary, messages, the full message contents are placed within a newly created `<SOAP:Body>` element.
- In the case of binary messages, rather than inserting the message content into the `body` variable, a `<binary-content/>` reference element is created and inserted into the `<SOAP:Body>` element. To learn how binary content is handled, see "Binary Content in the body and attachments Variables" on page 5-6.
- In the case of Java objects, a `<java-content/>` reference element is created and inserted into the `<SOAP:Body>` element. To learn how Java content is handled, see "Java Content in the body Variable" on page 5-7.

Attachments Variable

The `attachments` variable holds the attachments associated with a message. The `attachments` variable is defined by an XML schema. It consists of a single root node: `<ctx:attachments>`, with a `<ctx:attachment>` sub-element for each attachment. The sub-elements contain information about the attachment (derived from MIME headers) as well as the attachment

content. As with most of the other message-related variables, `attachments` is always set, but if there are no attachments, the `attachments` variable consists of an empty `<ctx:attachments>` element.

Each attachment element includes a set of sub-elements, as described in Table 5-2.

Table 5-2 Sub-Elements of the Attachments Variable

Elements of the Attachments Variable	Description[1]
`Content-ID`	A globally-unique reference that identifies the attachment.The type is `string`.
`Content Type`	Specifies the media type and sub-type of the attachment. The type is `string`.
`Content-Transfer-Encoding`	Specifies how the attachment is encoded. The type is `string`.
`Content-Description`	A textual description of the content. The type is `string`.
`Content-Location`	A locally-unique URI-based reference that identifies the attachment. The type is `string`.
`Content-Disposition`	Specifies how the attachment should be handled by the recipient. The type is `string`.
`body`	Holds the attachment data. The type is `anyType`.

1. The "Message Context Schema" on page 5-29 specifies the element types for the message context variables.

With the exception of the untyped `body` element, all other elements contain string values that are interpreted in the same way as they are interpreted in MIME—for example, valid values for the `Content-Type` element include `text/xml` and `text/xml; charset=utf-8`.

The parsing of attachments is not recursive. If an attachment has a `Content-Type` of `multipart/...`, the `body` element holds the original unpacked MIME content as a stream of bytes and does not contain attachment sub-elements. Because the MIME stream may contain binary data, it is represented by a `<binary-content>` reference element.

To learn how binary content is handled, see "Binary Content in the body and attachments Variables" on page 5-6.

Messages whose `Content-Type` is `multipart/form-data` are constructed at run-time as follows:

- Inbound: All parts of a received inbound `multipart/form-data` type message are assigned to the `$attachments` variable. The `$body` variable is left empty.
- Outbound: The content of an outbound `multipart/form-data` type message is built from the content of the`$attachments` variable. Nothing from `$header` or `$body` is included.

  Note: If the inbound message is of a different `multipart` type than `multipart/form-data` (for example, `multipart/related`) and the outbound message is `multipart/form-data`, you must explicitly preserve the headers and content of the inbound root part, because they will not otherwise be passed through.

Attachments are supported on inbound requests and on outbound responses (that is, in messages received by a proxy service) only when the transport is HTTP, HTTPS or e-mail. Attachments are supported for all transport types for outbound requests and inbound responses (that is for messages sent by a proxy service).

ALSB does not support sending attachments to EJB-based or Tuxedo-based services.

Binary Content in the body and attachments Variables

In the case of both the `body` and `attachments` variables, `text-`, `XML-` and `MFL-`based content is placed directly inside of an XML element. For binary data, which can contain byte values that are illegal in XML, ALSB does not place the binary content in the XML element. Consequently, the binary content cannot be manipulated, but it is handled efficiently.

When binary content is received, the ALSB run time stores it in an in-memory hash table and a reference to that content is inserted into the XML (`body` or `attachments`) element. This reference is represented by the following XML snippet:

```
<binary-content ref="..."/>
```

where the ref attribute contains a URI or URN that uniquely identifies the binary content. This XML can be manipulated in a ALSB pipeline, branch, or route node in the same way any other content can be manipulated, but only the reference and not the underlying binary content is affected.

For example:

- Binary content in the `body` variable can be copied to an attachment by copying the reference XML to the `body` sub-element of an attachment element.

- Binary content in two different attachments can be swapped by swapping the snippets of reference XML or by swapping the values of the ref attributes.

When messages are dispatched from ALSB, the URI in the reference XML is used to restore the relevant binary content in the outgoing message. For information about how outbound messages are constructed, see "Constructing Messages to Dispatch" on page 5-26.

Clients and certain transports, notably e-mail, file and FTP can use this same reference XML to implement pass-by-reference. In this case, the transport or client creates the reference XML rather than the proxy service run time. Also, the value of the URI in the `ref` attribute is specified by the user that creates the reference XML. For these cases in which the reference XML is not created by the proxy service run time—specifically, when the URI is not recognized as one referring to internally managed binary content—ALSB does not de-reference the URI, and the content is not substituted into an outgoing message.

Java Content in the body Variable

The ALSB pipeline supports Java objects as inputs and outputs to Java callout actions. A POJO returned by a Java callout is cached in the pipeline, and its key is returned wrapped in an XML message of the form `<java-content ref="cid:kkkkeeeeyyyy"/>`, where *`cid:kkkkeeeeyyyy`* is a key automatically generated by the producing action and used to index the object in the pipeline's cache. Any subsequent action then passes that XML unmodified as an argument.

The content of a POJO variable is not directly accessible by pipeline actions at configuration time. Rather, the content can be handled in the following ways:

- The content's metadata (that is, its key) can be handled as any other XML, for example in an XQuery such as `$pojo/java-content/@ref`. This may be useful for logging or debugging, but the content of the object cannot be directly accessed.
- The content can be assigned to a new variable that automatically becomes typed (in the pipeline) as a POJO. The object itself is not touched. The `<java-content.../>` XML snippet is copied from the source variable to the target variable.
- The content can be passed to another appropriate action (like Java callout) as a variable (for example, `$pojo`). The object itself is not touched. The argument is automatically de-referenced to the actual object.

The Java object is removed from the pipeline's cache when you delete all variables holding the object's key (in `<java-content…/>`) or when you delete all XPaths pointing to the `<java-content…/>` snippet.

Streaming body Content

For processing message content, you can specify that the ALSB pipeline streams the content rather than loading it into memory. When you enable content streaming for a proxy service, you specify whether to buffer the streamed content to memory or a disk file as an intermediate step during the processing of the message. The creation of these temporary files might affect performance. For information about protecting temporary files, see the *AquaLogic Service Bus Security Guide*.

When you enable the streaming option, content streaming applies *only* to the `body` variable.

In general, use content streaming:

- When processing large content messages. See the guidelines in "Best Practices for Using Content Streaming" on page 5-8.
- In use cases where ALSB accesses the payload a small number of times.
- For content-based routing without transformations; content streaming results in better performance due to the benefits from partial parsing.

Best Practices for Using Content Streaming

Use the following guidelines and recommendations:

- When you enable streaming for large message processing, you cannot use the assign, replace, rename, for each, validate, and delete actions with respect to the `body` message context variable, because these actions require the input variable to be fully materialized in memory and full materialization is incompatible with the content streaming option.
- You can use the results of an XQuery or XSL transformation from a very large `$body` with these pipeline actions:
  - Assign, insert, and replace actions—to update the value of another context variable (not `$body`). However, you must ensure that the result of the expression is small enough to be fully materialized and stored in the message context.
  - Java callouts—to pass input arguments. All input to Java callouts is fully materialized, therefore, the results of expressions used as input must be small enough to be fully materialized.
  - MFL transformations—to transform very large payloads without first materializing the input as an XML Bean. When using a very large `$body` as an input to an MFL transformation, declare a messaging service, binary message type proxy service. If you

declare a messaging service, text message type proxy service, `$body` will get fully materialized to obtain an input stream for the transformation.

  - Alert, log, and report actions—to report the result of an XQuery or XSL transformation on a very large `$body`.
  - Service callouts

- For XSL transformations, all input is fully materialized in order to perform the transformation, therefore, you must ensure that the input is small enough so that it can be fully materialized and processed by the XSLT processor.
- With very large MFL input, you should use a MFL service instead of a MFL stage action to perform a MFL-to-XML transformation.
- Do not use the test console to test proxy services with very large content messages because the content will be fully materialized, potentially causing an OOM exception, and displayed, causing a slowdown in the console window.
- When writing XQueries, use proper indexing to achieve partial parsing.

  For example, instead of writing `$body/*:DateTimeStruct`, which would consume the entire input stream, write:

```
($body/*:DateTimeStruct)[1] or $body[1]/*:DateTimeStruct[1]
```

  By using indexing, only content up to and including the first `DateTimeStruct` element will be parsed.

- Because each variable that is accessed by two or more consumers (expressions) is materialized, when writing XQueries, avoid statements such as:

```
let $labdata1 := $body/*
return <HEADER>{ $labdata1/HEADER/@*, $labdata1/HEADER/node() }</HEADER>
```

  In this case, `$labdata1` is bound to the whole document without the root element so the XQuery engine runs out of memory when trying to materialize it.

  One way of changing this query to avoid excessive materialization would be to move the `/HEADER` path expression inside the `let` clause.

```
let $labdata1 := $body/*/HEADER
...
```

  In this case, the XQuery engine will only materialize the `HEADER` element or elements.

  Another way to avoid materialization would be to use the `fn-bea:rename()` function in which you can rename elements in a streaming mode.

```
fn-bea:rename($oldelements as element()*, $newname as element()) as element()*
```

For example:

```
fn-bea:rename($body/*/HEADER, <HEADER_NEW/>)
```

- At run time, processing large messages is subjected to the limitations and restrictions of the underlying transport; for example, the message size handling limitations of the transport. Be aware of the JVM and RMI settings that limit the capacity of the transport to accept large messages.

Inbound and Outbound Variables

The `inbound` and `outbound` context variables contain information about the inbound and outbound endpoints. The `inbound` variable contains information about the proxy service that received the request message; the `outbound` variable contains information about the target business service to which a message is sent.

The `outbound` variable is set in the route action in route nodes and publish actions. You can modify `$outbound` by configuring request and response actions in route nodes and by configuring request actions in publish actions.

WARNING: Some modifications that you can make for the `inbound` and `outbound` context variables are not honored at run time. That is, the values of certain headers and metadata can be overwritten or ignored by the ALSB run time. The same limitations are true when you set the transport headers and metadata using the transport headers and service callout actions, and when you use the Test Console to test your proxy or business services. For information about the headers and metadata for which there are limitations, see "Understanding How the Run Time Uses the Transport Headers Settings" on page 3-12. Note also that any modifications you make to `$outbound` in the message flow *outside* of the request or response actions in route nodes and publish actions are ignored. In other words, those modifications are overwritten when `$outbound` is initialized in the route nodes and publish actions.

You cannot modify the `outbound` variable in service callout actions.

The `inbound` and `outbound` variables have the following characteristics:

- Have the same XML schema—the `inbound` and `outbound` context variables are instances of the `endpoint` element as described in "Message Context Schema" on page 5-29.

- Contain a single `name` attribute that identifies the name of the endpoint as it is registered in the service directory. The `name` attribute should be considered read-only for both `inbound` and `outbound`.

  WARNING: The read-only rule is not enforced. Changing read-only elements can result in unpredictable behavior.

- Contain the `service`, `transport` and `security` sub-elements described in the following section.

Attachments are supported on inbound requests and outbound responses (that is, in messages received by a proxy service) only when the transport is HTTP, HTTPS or e-mail.

Attachments are supported for all transport types for outbound requests and inbound responses (that is for messages sent by a proxy service).

ALSB does not support sending attachments to EJB-based or Tuxedo-based services.

Sub-Elements of the inbound and outbound Variables

This section describes the sub-elements of the `inbound` and `outbound` context variables, including information about whether a given sub-element is initialized at run time. To learn about how context variables are initialized, see "Initializing Context Variables" on page 5-21. The sub-elements include:

- service
- transport
- security

service

The `service` element is read-only for both `inbound` and `outbound`. Sub-elements include `providerName` and `operation`.

Table 5-3 Sub-Elements of the service Element

Sub-Elements[1]	Description...

Table 5-3 Sub-Elements of the service Element (Continued)

`providerName`	Specifies the name of the service key provider. Initialized based on the configuration of publish and routing actions.
`operation` (`outbound` only)	Used in the `outbound` variable, specifies the name of the operation to be invoked on the target business service. Initialized based on the inbound and outbound. **Note:** This element is used for the `outbound` variable only. In the case of `inbound` messages, the name of the operation to be invoked on the proxy service is specified by the `operation` variable.

1. The "Message Context Schema" on page 5-29 specifies the element types for the message context variables.

transport

The `transport` element is read-only on `inbound`, except for the `response` element, which you can modify to set the response transport headers. The sub-elements of the `transport` element are described in Table 5-4.

Table 5-4 Sub-Elements of the Transport Element

Sub-Elements[1]	Description...
`uri`	Identifies the URI of the endpoint: • When used in the `inbound` variable, this is the URI by which the message arrived. • When used in the `outbound` variable, this is the URI to use when sending the message—it overrides any URI value registered in the service directory. **Initialization** The URI element is initialized as follows: • Always initialized on the `inbound` variable • Never initialized on the `outbound` variable. You can set the URI on `outbound` when you want to override the set of URIs in the service configuration. URI failover is not supported if this element is set.

Table 5-4 Sub-Elements of the Transport Element (Continued)

Sub-Elements[1]	Description...
`request` This element is read-only[2] in the `inbound` variable. You can modify it for the `outbound` variable.	Specifies transport-specific metadata about the request (including transport headers). The value for this element is defined by the transport protocol (specifically, the `RequestMetaData` XML defined by the transport SDK).Therefore, the structure of this element depends on the transport being used. To learn about the transport-specific types for this element, see the appropriate transport schema, which are available in the following directory in your ALSB installation: *BEA_HOME*`/alsb_3.0/lib/transports/` where *BEA_HOME* represents the directory in which you installed ALSB. **Initialization** The URI element is initialized as follows: • Initialized on the `inbound` variable using information from the request message received by ALSB. • On the `outbound` variable, the `request` element is created with the proper typing. The typing is transport-dependent. The `request` element is typically initialized as an empty element, with the exception of certain important transport headers—for example, `content-type` and `SOAPAction`. You can set a filename for an outbound message using the File transport protocol by configuring `$outbound` in a route node request action, as follows: • If the `fileName` only is specified, a file of that name is stored at the location specified by the endpoint URI of the target business service. • If `isFilePath` is set to `true`, the value of `fileName` is used as a relative path appended to the endpoint URI of the target business service. For example, if the endpoint URI is `file:////apollo/ob/data`, and the `fileName` header is set to `./foo/bar.xml`, and `isFilePath` is set to `true`, the message will be stored at `/apollo/ob/data/foo/bar.xml`. If a file already exists with that name, a new name is generated, following the format *path*/*filename_random-number*.xml, where *random-number* is an integer in the range of `0` to `999999`.

Table 5-4 Sub-Elements of the Transport Element (Continued)

Sub-Elements[1]	Description...
`response` This element is read-only in the `outbound` `variable`. You can modify it for the `inbound`. variable.	Specifies transport-specific metadata about the response (including transport headers). The value for this element is defined by the transport protocol (specifically, the `ResponseMetaData` XML defined by the transport SDK).Therefore, the structure of this element depends on the transport being used. To learn about the transport-specific types for this element, see the appropriate transport schema, which are available in the following directory in your ALSB installation: *BEA_HOME*`/alsb_3.0/lib/transports/` where *BEA_HOME* represents the directory in which you installed ALSB. **Initialization** The `URI` element is initialized as follows: • Initialized on the `outbound` variable using information from the response message received by ALSB. • On the `inbound` variable, the `response` element is created with the proper typing. The typing is transport-dependent. The `response` element is typically initialized as an empty element, with the exception of certain important transport headers—for example, `content-type` and `SOAPAction`. For a description of the standard HTTP headers, see `http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html` For a description of the standard JMS headers, see Value-Added Public JMS API Extensions. **Note:** The following MQ headers do not have equivalents in BEA JMS: `ApplOriginData`, `ApplIdentityData`, `Accounting Token`
`mode`	Specifies whether the communication style is `request` (one-way) or `request-response` (two-way). **Initialization** Initialized on the `inbound` and `outbound` variables using information from the service and its operations (if applicable). For example, if a request-only operation is being invoked, the `mode` element is set to `request`, rather than to `request-response`.

Table 5-4 Sub-Elements of the Transport Element (Continued)

Sub-Elements[1]	Description...
`qualityOfService` This element is read only for `inbound`. You can modify it for the `outbound` case—in the outbound request actions of a publish or routing action.	Specifies the quality of service expected when sending or receiving a message. Valid values include `best-effort` and `exactly-once`: • `best-effort` means that each dispatch defines its own transactional context (if the transport is transactional). Best effort means that there is no reliable messaging and no elimination of duplicate messages—however, performance is optimized. For the scenario in which a message is dispatched as a result of a publish action, any dispatch errors are suppressed. For the scenario in which a message is dispatched from a routing node, dispatch errors are not suppressed. • `exactly-once` means that the dispatch is included as part of the inbound transactional context (if one exists and if the outbound transport is transactional) and errors cause processing to abort and trigger the relevant error handler (in the case of both the route and publish scenarios). *Exactly once* reliability means that messages are delivered from inbound to outbound exactly once, assuming a terminating error does not occur before the outbound message send is initiated. **Initialization** The `qualityOfService` element is initialized on the inbound and outbound variables as follows: • In the inbound case, the quality of service (QoS) is dictated by the transport. For example, for the JMS/XA transport, the QoS is *exactly once*; for the HTTP transport, the QoS is *best effort*. • In the outbound case, the QoS is set differently for publishing and for routing, as follows: **Routing**—When messages are routed to another service from a route node, the QoS is always initialized using the value from the inbound context variable. In other words, the outbound QoS is set to *exactly once* if (and only if) the inbound QoS is *exactly once*. Otherwise, the outbound QoS is set to *best effort*. **Publishing—**When a message is published to another service as the result of a publish action, the quality of service (QoS) is always initialized to *best effort* regardless of the inbound setting.

Table 5-4 Sub-Elements of the Transport Element (Continued)

Sub-Elements[1]	Description...
`retryCount` (`outbound` only)	Specifies the number of retries to attempt when sending a message from ALSB. If `retryCount` is set, the setting overrides any retry count value configured in the target service configuration.

1. The "Message Context Schema" on page 5-29 specifies the element types for the message context variables.
2. The read-only rule is not enforced. Changing read-only elements can result in unpredictable behavior.

security

The sub elements of the `security` element are described in Table 5-5.

Table 5-5 Sub-Elements of the Security Element

Sub-Elements[1]	Description...
`transportClient` (`inbound` only, read only[2])	Specifies authenticated transport-level user information. The user information includes a username and any optional principals. The principals can themselves include zero or more groups, one for each group the subject belongs to. **Note:** If the subject is anonymous, then the username is "`anonymous`" and there are no groups. Initialized by ALSB. The inbound `transportClient` element is read-only.
`messageLevelClient` (`inbound` only, read only[2])	Specifies authenticated message-level user information. The user information includes a username and any optional principals. The principals can themselves include zero or more groups, one for each group the subject belongs to. **Note:** If the subject is anonymous, then the username is "`anonymous`" and there are no groups. Initialized by ALSB. The inbound `messageLevelClient` element is read-only.
`doOutboundWss` (`outbound` only)	ALSB sets the value of this element during routing or publishing. Some infrequently used design patterns set the value to `false` to preempt a proxy service from automatically generating the outbound WS-Security SOAP envelope. Future releases of ALSB will provide an easier way to disable outbound WS-Security. For more information, see "Disabling Outbound WS-Security" under Message-Level Security in *AquaLogic Service Bus Security Guide*.

1. The "Message Context Schema" on page 5-29 specifies the element types for the message context variables.
2. The read-only rule is not enforced. Changing read-only elements can result in unpredictable behavior.

Related Topics

Proxy Services: Actions in *Using the AquaLogic Service Bus Console*

"Adding Route Node Actions" in Proxy Services: Message Flow in *Using the AquaLogic Service Bus Console*

For a description of the standard HTTP headers, see
`http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html`

For a description of the standard JMS headers, see
http://e-docs.bea.com/wls/docs92/jms/fund.html#jms_features

Operation Variable

The `operation` variable is a read-only variable. It contains a string that identifies the operation to be invoked on a proxy service. If no operations are defined for a proxy service, the `operation` variable is not set and returns the equivalent of null.

ALSB provides the `operation` variable as a stand-alone variable, rather than as a sub-element of the `inbound` variable to optimize performance—the computation of the operation may be deferred until the `operation` variable is explicitly accessed rather than anytime the `inbound` variable is accessed.

Fault Variable

The `fault` variable is used to hold information about any error that has occurred during message processing. When an error occurs, this variable is populated with information before the appropriate error handler is invoked.

Note: This variable is defined only in error handler pipelines and is not set in request and response pipelines, or in route or branch nodes.

The `fault` variable includes the `errorCode`, `reason`, `details`, and `location` sub-elements described in Table 5-6.

Table 5-6 Sub-Elements of the Fault Variable

Elements of the Fault Variables	Description[1]...
`errorCode`	Specifies the error code as a string value
`reason`	Contains a text description of the error
`details`	Contains user-defined XML content related to the error
`location`	Identifies the node, pipeline and stage in which the error occurred. Also identifies if the error occurred in an error handler. The sub-elements include: • node—the name of the pipeline, branch, or route node where an error occurred; a string. • pipeline—the name of the Pipeline where an error occurred (if applicable); a string. • stage—the name of the stage where an error occurred (if applicable); a string. • error-handler—indicates if an error occurred from inside an error handler; a boolean.

1. The "Message Context Schema" on page 5-29 specifies the element types for the message context variables.

The contents of the `fault` variable are modeled after SOAP faults to facilitate fault generation when replying from a SOAP-based proxy service. The values for error codes generated by ALSB correspond to system error codes and are prefixed with BEA string.

The error codes associated with the errors surface inside the element of the `fault` context variable. You can access the value using the following XQuery statement:

```
$fault/ctx:errorCode/text()
```

ALSB defines three generic error codes for the three classes of possible errors. The format of the generic codes is `BEA-xxx000`, where `xxx` represents a generic category as follows:

- 380 Transport
- 382 Proxy
- 386 Security

- 394 UDDI

This yields the generic codes as follows:

- BEA–380000—BEA–380999

  Indicates a transport error (for example, failure to dispatch a message).

- BEA–382000—BEA–382499

  Indicates a proxy service run-time error (for example, a stage exception).

- BEA–382500—BEA–382999

  Indicates an error in a proxy service action.

- BEA–386000—BEA–386999

  Indicates a WS-Security error (for example, authorization failure).

- BEA–394500—BEA–394999

  Indicates an error in the UDDI sub system.

ALSB defines unique codes for specific errors. For example:

BEA-382030—Indicates a message parse error (for example, a SOAP proxy service received a non-SOAP message).

BEA-382500—Reserved for the case in which a service callout action receives a SOAP Fault response.

For information about these and other specific error codes, see Error Codes in *Using the AquaLogic Service Bus Console*. See also "Handling Errors in Message Flows" on page 3-31.

Initializing Context Variables

The message context and its variables are initialized in the binding layer when a message is received and before message processing begins. Table 5-7summarizes how context variables are initialized.

Table 5-7 Initializing Context Variables

Context Variable	How Initialized
`outbound`	Initialized to null because no routing or errors have yet occurred. The outbound variable is initialized in the route action in route nodes and publish actions. You can modify `$outbound` through the request actions in routing nodes and publish actions (also in the response actions in routing nodes). For more information, see "Inbound and Outbound Variables" on page 5-10. For information about the initialization of sub-elements of `outbound`, see "Sub-Elements of the inbound and outbound Variables" on page 5-11.
`fault`	
`inbound`	Initialized with service, transport and security information that is obtained from Service Bus metadata about the registered proxy service and transport-level metadata (transport headers, authenticated user information, and so on) about the specific incoming request. For information about the initialization of sub-elements of `inbound`, see "Sub-Elements of the inbound and outbound Variables" on page 5-11.
`header`	Initialized using the content of the inbound message. How the initialization is performed depends on the type of proxy service, as described in the subsequent topics in this section: • "Initializing the attachments Context Variable" on page 5-23 • "Initializing the header and body Context Variables" on page 5-23 The `header`, `body`, and `attachments` variables are re initialized after routing using the content of the response that is received. If no routing is performed or if the communication mode is request-only, then these variables are not re initialized. That is, they are not cleared of any content.
`body`	
`attachments`	
`operation`	

Initializing the attachments Context Variable

The `attachments` context variable is initialized with any MIME attachments that accompany the message, but does not include the part representing the main message (whether it is SOAP, XML, MFL, and so on). Each `<attachment>` element is initialized using the MIME headers that accompany each part in the MIME package.

The contents of the `<body>` element in the `<attachment>` can be one of the following depending on the attachment's `Content-Type`:

- XML
- text
- A snippet of reference XML that refers to the attachment content (see "Binary Content in the body and attachments Variables" on page 5-6)

Initializing the header and body Context Variables

This section describes how the initialization of `header` and `body` context variables is performed depending on the type of proxy service: SOAP Services, XML Services (Non SOAP), Messaging Services.

SOAP Services

Messages to SOAP-based services are SOAP messages containing XML that is contained in a `<soap:Envelope>` element. In the case that messages include attachments, the content of the inbound message is a MIME package that includes the SOAP envelope as one of the parts—typically the first part or one identified by the top-level `Content-Type` header. The context variables are initialized as follows:

- `header`—initialized with the `<soap:Header>` element from the SOAP message
- `body`—initialized with the `<soap:Body>` element from the SOAP message

XML Services (Non SOAP)

The messages to XML-based services are XML, but can be of any type allowed by the proxy service configuration. In the case that messages include attachments, the content of the inbound messages is a MIME package that includes the primary XML payload as one of the parts—typically the first part or one identified by the top-level `Content-Type` header.

The context variables are initialized as follows:

- `header`—initialized with an empty `<soap:Header/>` element.
- `body`—initialized with a `<soap:Body>` element that wraps the entire XML payload.

Messaging Services

Messaging services are those that can receive messages of one data type and respond with messages of a different data type. The supported data types include XML, MFL, text, untyped binary. The context variables are initialized as follows:

- `header`—initialized with an empty `<soap:Header/>` element.
- `body`—initialized with a `<soap:Body>` element that wraps the entire payload.
  - In the case of XML, MFL, and text content, it is placed directly within the `<soap:Body>` element.
  - In the case of binary content, a piece of reference XML is created and inserted inside the `<soap:Body>` element (see "Binary Content in the body and attachments Variables" on page 5-6). The binary content cannot be accessed or modified, but the reference XML can be examined, modified, and replaced with inline content.

Performing Operations on Context Variables

You interact with and manipulate the message context through actions in the pipelines, branch, or route nodes that define a proxy service. Most actions expose the XQuery language to do so. Each context variable is represented as an XQuery variable of the same name. For example, the `header` variable is accessible in XQuery as `$header`, the `body` variable is accessible as `$body`, and so on. The examples in this section show the use of XQuery to examine and manipulate context variables.

$body

The `$body` variable includes the `<soap-env:Body>...</soap-env:Body>` element. (If the proxy service is SOAP 1.2, the `body` variable contains a SOAP 1.2 Body element.)

For example, if you assign data to the `body` context variable using the assign action, you must wrap it with the `<soap-env:Body>` element. In other words, you build the SOAP package by including the `<soap-env:Body>` element in the context variable.

There is an exception to this behavior in ALSB—for the case in which you build the Request Document Variable for the service callout action. Service callout actions work with the core payload (RPC parameters, documents, and so on) and ALSB builds the SOAP package around the core payload. In other words, when you configure the Request Document Variable for a service callout action, you do not wrap the input document with `<soap-env:Body>...</soap-env:Body>`.

For information about configuring the service callout action, see Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.

$header

The `$header` variable includes the `<soap-env:Header>...</soap-env:Header>` element. (If the proxy service is SOAP 1.2, the `header` variable contains a SOAP 1.2 Header element.)

For example if you assign data to the header context variable using the assign action, you must wrap it with the `<soap-env:Header>` element. In other words, you build the SOAP package by including the `<soap-env:Header>` element in the context variable. This is true for all manipulations of `$header`, including the case in which you can set one or more SOAP headers for a service callout request. For information about configuring SOAP headers for a service callout action, see Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.

Extract the WS-Addressing Header—From

```
$header/wsa:From
```

Extract the Payload From a Non-SOAP Message

```
$body/*
```

Extract the user-header From an Outbound Response Message

```
$outbound/ctx:transport/ctx:response/tp:user-header[@name='myheader']/@value
```

When creating a `body` input variable that is used for the request parameter in a service callout to a SOAP Service, you would define that variable's contents using `body/*` (to remove the wrapper `soap-env:Body`), not `$body` (which results in keeping the `soap-env:Body` wrapper).

Assign Variable Contents for Request Parameter in a Service Callout

```
$body/*
```

Related Topics

For more information about handling context variables using the XQuery and XPath editors, see:

- “Working with Variable Structures” on page 3-44.
- Proxy Services: XQuery Editors in *Using the AquaLogic Service Bus Console*
- Expression Editors in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*

Constructing Messages to Dispatch

When ALSB publishes or routes a message, the content of the message is constructed using the values of variables in the message context. For example, transport headers and other transport-specific metadata are taken from `$outbound/transport/request`. As is the case with initialization of the context, the message content for outbound messages is handled differently depending upon the type of the target service. How the outbound message content is created depends on the type of the target service, as described in the following topics:

- SOAP Services
- XML Services (Non SOAP)
- Messaging Services

SOAP Services

An outgoing SOAP message is constructed by wrapping the contents of the `header` and `body` variables inside a `<soap:Envelope>` element. If the invoked service is a SOAP 1.2 service, the envelope created is a SOAP 1.2 envelope. If the invoked service is a SOAP 1.1 service, the envelope created is a SOAP 1.1 envelope. If the `body` variable contains a piece of reference XML, it is sent as is—in other words, the referenced content is not substituted into the message.

If attachments are defined in the `attachments` variable, a MIME package is created from the main message and the attachment data. The handling of the content for each attachment part is similar to how content is handled for messaging services.

XML Services (Non SOAP)

The messages to XML-based services from ALSB is constructed from the contents of the `body` variable:

- If the `body` variable is empty, then a zero-size message is sent.

- If the `body` variable contains multiple XML snippets, then only the first snippet is used in the outbound message. For example, if `<soap:Body>` contains `<abc/><xyz/>`, only <abc/> is sent.
- If the content of the `body` variable is text and not XML, an error is thrown.
- If the `body` variable contains a piece of reference XML, it is sent as is—in other words, the referenced content is not substituted into the message.
- If attachments are defined in the `attachments` variable, a MIME package is created from the XML message and the attachment data. In the case of a null XML message, the corresponding MIME body part is empty. The handling of the content for each attachment part is similar to how content is handled for messaging services.

Regardless of any data it contains, the `header` variable does not contribute any content to the outbound message.

For examples of how messages are constructed for service callout actions, see Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.

Messaging Services

The messages to messaging services from ALSB are constructed from the contents of the `body` variable.

- If the `body` variable is empty, then a zero-size message is sent, regardless of the outgoing message type.
- If the outgoing message type is XML, then the message is constructed in the same way as it is for XML Services (Non SOAP).
- If the outgoing message type is MFL, then the behavior is similar to that for XML message types except that the extracted XML is converted to MFL. (An error occurs if the XML→MFL conversion cannot be performed.)
- If the target service requires text messages, the contents of the `body` variable are interpreted as text and sent. In this way, it is possible for ALSB to handle incoming XML messages that must be delivered to a target service as text. In other words, you do not need to configure the message flow to handle such messages.
- For target services that expect binary messages, the `body` variable must contain a piece of reference XML—the reference URI references the binary data stored in the ALSB in-memory hash table. The referenced content is sent to the target service.

For cases in which a client, a transport, or the designer of a proxy service specifies the reference URI, the referenced data is not stored in the ALSB and thus cannot be de referenced to populate the outbound message. Consequently, the reference XML is sent in the message.

If the `body` variable contains a piece of reference XML, and the target service requires a message type other than binary, the reference XML inside the `body` variable is treated as content. In other words, it is sent as XML, converted to text, or converted to MFL. This is true regardless of the URI in the reference XML.

Regardless of any data it contains, the `header` variable does not contribute any content to the outbound message.

For examples of how messages are constructed for service callout actions, see Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.

About Sending Binary Content in Email Messages

For binary messages, ALSB does not insert the message content into the `body` variable. Instead, a `<binary-content/>` reference element is created and inserted into the `<SOAP:Body>` element (see "Message-Related Variables" on page 5-3). However, the email standard does not support sending binary content type as the main part of a message. If you want to send binary messages via email to a messaging service that accepts text or XML documents and optional attachments, you can do so as follows:

1. Transfer the binary-content reference XML from `$body` to `$attachments`.
2. Replace the content of `$body` with text or XML wrapped in a `<SOAP:Body>` element.

For the case in which the outgoing message type is MFL, the contents of `$body` is converted from XML to text or binary based on the MFL transformation:

- If the target service expects to receive text message, you can set the `content-type` (the default is binary for MFL message type) as `text/plain` in `$outbound`
- If the target service expects to receive binary messages, it is not possible to send MFL content via the email transport.

To learn more about how binary content is handled, see "Binary Content in the body and attachments Variables" on page 5-6.

Related Topics

"Message Context Schema" on page 5-29

In *Using the AquaLogic Service Bus Console:*

- "Service Callout" and "Transport Headers" in Proxy Services: Actions
- "Adding a Route Node" in Proxy Services: Message Flow

Message Context Schema

The message context schema (`MessageContext.xsd`) that specifies the types for the message context variables is shown in "Message Context.xsd" on page 5-29.

When working with the message context variables, you need to reference `MessageContext.xsd` which is available in a JAR file, *`BEA_HOME`*`/alsb_3.0/lib/sb-kernel-api.jar`, and the transport-specific schemas, which are available at

`BEA_HOME``/alsb_3.0/lib/transports/`

where *`BEA_HOME`* represents the directory in which you installed ALSB.

Message Context.xsd

//depot/dev/src/wli/public/sb/schemas/MessageContext.xsd last updates @v9 6/11/05

```
<schema targetNamespace="http://www.bea.com/wli/sb/context"
        xmlns:mc="http://www.bea.com/wli/sb/context"
        xmlns="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">
    <!--============================================================== -->

    <!-- The context variable 'fault' is an instance of this element -->
    <element name="fault" type="mc:FaultType"/>

    <!-- The context variables 'inbound' and 'outbound' are instances of this
element -->
    <element name="endpoint" type="mc:EndpointType"/>

   <!-- The three sub-elements within the 'inbound' and 'outbound' variables -->
    <element name="service" type="mc:ServiceType"/>
    <element name="transport" type="mc:TransportType"/>
    <element name="security" type="mc:SecurityType"/>

    <!-- The context variable 'attachments' is an instance of this element -->
    <element name="attachments" type="mc:AttachmentsType"/>

    <!-- Each attachment in the 'attachments' variable is represented by an
instance of this element -->
```

```
    <element name="attachment" type="mc:AttachmentType"/>

   <!-- Element used to represent binary payloads and pass-by reference content
-->
    <element name="binary-content" type="mc:BinaryContentType"/>

   <!-- ================================================================= -->

    <!-- The schema type for  -->
    <complexType name="AttachmentsType">
        <sequence>
         <!-- the 'attachments' variable is just a series of attachment elements
-->
            <element ref="mc:attachment" minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
    </complexType>

    <complexType name="AttachmentType">
        <all>
            <!-- Set of MIME headers associated with attachment -->
            <element name="Content-ID" type="string" minOccurs="0"/>
            <element name="Content-Type" type="string" minOccurs="0"/>
            <element name="Content-Transfer-Encoding" type="string"
minOccurs="0"/>
            <element name="Content-Description" type="string" minOccurs="0"/>
            <element name="Content-Location" type="string" minOccurs="0"/>
            <element name="Content-Disposition" type="string" minOccurs="0"/>

            <!-- Contains the attachment content itself, either in-lined or as
<binary-content/> -->
            <element name="body" type="anyType"/>
        </all>
    </complexType>

    <complexType name="BinaryContentType">
        <!-- URI reference to the binary or pass-by-reference payload -->
        <attribute name="ref" type="anyURI" use="required"/>
    </complexType>

   <!-- ================================================================= -->

    <complexType name="EndpointType">
        <all>
            <!-- Sub-elements holding service, transport, and security details
for the endpoint -->
            <element ref="mc:service" minOccurs="0" />
            <element ref="mc:transport" minOccurs="0" />
            <element ref="mc:security" minOccurs="0" />
        </all>
```

```
       <!-- Fully-qualified name of the service represented by this endpoint -->
        <attribute name="name" type="string" use="required"/>
    </complexType>

   <!-- ================================================================== -->

    <complexType name="ServiceType">
        <all>
            <!-- name of service provider -->
            <element name="providerName" type="string" minOccurs="0"/>

            <!-- the service operation being invoked -->
            <element name="operation" type="string" minOccurs="0"/>
        </all>
    </complexType>

   <!-- ================================================================== -->

    <complexType name="TransportType">
        <all>
            <!-- URI of endpoint -->
            <element name="uri" type="anyURI" minOccurs="0" />

          <!-- Transport-specific metadata for request and response (includes
transport headers) -->
            <element name="request" type="anyType" minOccurs="0"/>
            <element name="response" type="anyType" minOccurs="0" />

            <!-- Indicates one-way (request only) or bi-directional
(request/response) communication -->
            <element name="mode" type="mc:ModeType" minOccurs="0" />

            <!-- Specifies the quality of service -->
            <element name="qualityOfService" type="mc:QoSType" minOccurs="0" />

            <!-- Retry values (outbound only) -->
            <element name="retryInterval" type="integer" minOccurs="0" />
            <element name="retryCount" type="integer" minOccurs="0" />
        </all>
    </complexType>

    <simpleType name="ModeType">
        <restriction base="string">
            <enumeration value="request"/>
            <enumeration value="request-response"/>
        </restriction>
    </simpleType>
```

```
    <simpleType name="QoSType">
        <restriction base="string">
            <enumeration value="best-effort"/>
            <enumeration value="exactly-once"/>
        </restriction>
    </simpleType>

   <!-- ================================================================= -->

    <complexType name="SecurityType">
        <all>
            <!-- Transport-level client information (inbound only) -->
          <element name="transportClient" type="mc:SubjectType" minOccurs="0"/>

            <!-- Message-level client information (inbound only) -->
            <element name="messageLevelClient" type="mc:SubjectType"
minOccurs="0"/>

           <!-- Boolean flag used to disable outbound WSS processing (outbound
only) -->
            <element name="doOutboundWss" type="boolean" minOccurs="0"/>
        </all>
    </complexType>

<complexType name="SubjectType">
  <sequence>
    <!-- User name associated with this tranport- or message-level subject -->
    <element name="username" type="string"/>
    <element name="principals" minOccurs="0">
     <complexType>
      <sequence>
      <!-- There is an element for each group this subject belongs to, as
        determined by the authentication providers -->
      <element name="group" type="string"
                     minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
     </complexType>

<!-- ================================================================= -->

    <complexType name="FaultType">
        <all>
            <!-- A short string identifying the error (e.g. BEA38229) -->
            <element name="errorCode" type="string"/>

            <!-- Descriptive text explaining the reason for the error -->
            <element name="reason" type="string" minOccurs="0" />

            <!-- Any additional details about the error  -->
```

```
            <element name="details" type="anyType" minOccurs="0" />

            <!-- Information about where the error occured in the proxy -->
            <element name="location" type="mc:LocationType" minOccurs="0" />
        </all>
    </complexType>

    <complexType name="LocationType">
        <all>
            <!-- Name of the Pipeline/Branch/Route node where error occured -->
            <element name="node" type="string" minOccurs="0" />

            <!-- Name of the Pipeline where error occured (if applicable) -->
            <element name="pipeline" type="string" minOccurs="0" />

            <!-- Name of the Stage where error occured (if applicable) -->
            <element name="stage" type="string" minOccurs="0" />

            <!-- Indicates if error occured from inside an error handler -->
            <element name="error-handler" type="boolean" minOccurs="0" />
        </all>
    </complexType>
    <!-- Encapsulates any stack-traces that may be added to a fault <details> -->
    <element name="stack-trace" type="string"/>
</schema>
```

Related Topics

“Inbound and Outbound Variables” on page 5-10

“Performing Operations on Context Variables” on page 5-24

“Constructing Messages to Dispatch” on page 5-26

CHAPTER 6

Using the Test Console

The ALSB Test Console is a browser-based test environment you use validate and test the design of your system. It is an extension of the ALSB Console. (The Test Console is not available in the ALSB Plug-in for WorkSpace Studio) You configure the object of your test (proxy service, business service, XQuery, XSLT, or MFL resource), execute the test, and view the results in the test console. In some cases, you can trace through the code and examine the state of the message at specific trace points. Design time testing helps isolate design problems before you deploy a configuration to a production environment.

The test console can test specific parts of your system in isolation and it can test your system as a unit. You can do testing in clustered environments. However, in a clustered domain, you cannot use the test console to test any configured business service or proxy service which routes to a business service.

You can access the test console from:

- The Project Explorer
- The Resource Browser
- The XQuery Editor

For detailed procedural information, see Test Console in *Using the AquaLogic Service Bus Console*.

Features

The test console supports the following features:

- Testing proxy services
- Testing business services
- Testing resources
- Testing XQueries
- Tracing messages through the message flow (for proxy services only)

Prerequisites

To use the test console:

- You must have ALSB running and you must have activated the session that contains the resource you want to test.
- You must disable the pop-up blockers in your browser for the XQuery testing to work. Note that if you have toolbars in the Internet Explorer browser, this may mean disabling pop-up blockers from under the **Options** menu as well as for all toolbars that are configured to block them. XQuery testing is done only in the design time environment (in an active session).
- If you want the test console to generate and send SAML tokens to a proxy service, you must configure the proxy service to require SAML tokens *and* to be a relying party. For more information on creating a SAML relying party, see Create a SAML Relying Party in *WebLogic Server Administration Console Online Help*.

  Note: When creating a SAML relying party:

  - Only WSS/Sender-Vouches and WSS/Holder-of-Key SAML profiles are applicable to a proxy service.
  - When you are configuring the relying party, for the target URL value provide the *URI* of the proxy service. You can view the URI of the proxy service by clicking on the proxy service name in the ALSB Console Project Explorer module. The URI displays in the Endpoint URI row of the Transport Configuration table.

Testing Proxy Services

You can test the following types of proxy services:

- WSDL Web Service
- Messaging Service

- Any Soap Service
- Any XML Service

Direct Calls

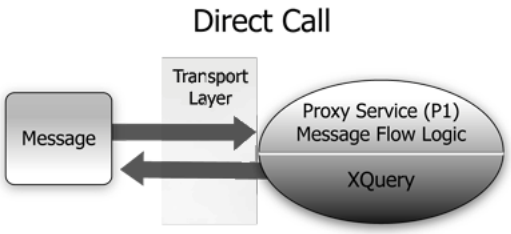
WARNING: Testing proxy services with the direct call option enabled bypasses some important security steps, including access control. BEA Systems recommends that you not use the test service in production systems.

A direct call is used to test a proxy service that is collocated in the ALSB domain. Using the direct call option, messages are sent directly to the proxy service, bypassing the transport layer. When you employ the direct call option, tracing is turned on by default, allowing you to diagnose and troubleshoot a message flow in the test console. By default, testing of proxy services is done using the direct call option.

When you use the direct call option to test a proxy service, the configuration data you input to the test console must be that which is expected by the proxy service from the client that invokes it. In other words, the test console plays the role of the client invoking the proxy service. Also, when you do direct call testing, you bypass the monitoring framework for the message.

Figure 6-1 illustrates a direct call. Note that the message bypasses the transport layer; it is delivered directly to the proxy service (P1).

Figure 6-1 Direct Call to Test a Proxy Service



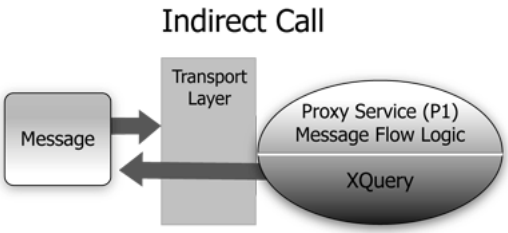
A direct call strategy is best suited for testing the internal message flow logic of proxy services. Your test data should simulate the expected message state at the time it is dispatched. Use this test approach in conjunction with setting custom (inbound) transport headers in the test console **Transport** panel to accurately simulate the service call.

Indirect Calls

When you test a proxy service with an *indirect* call (that is, when the direct call option is not selected), the message is sent to the proxy service through the transport layer. The transport layer performs manipulation of message headers or metadata as part of the test. The effect is to invoke a proxy service to proxy service run-time path.

Figure 6-2 illustrates an indirect call. Note that the message is first processed through the transport layer and is subsequently delivered to the proxy service (P1).

Figure 6-2 Indirect Call to Test a Proxy Service



BEA recommends this testing strategy when testing a proxy service to proxy service interface when both services run in the same JVM. Use this test approach in conjunction with setting custom (outbound) transport headers in the test console **Transport** panel to accurately simulate the service call. For more information on transport settings, see "Test Console Transport Settings" on page 6-20.

Using an indirect call, the configuration data you input to the test is the data being sent from a proxy service, for example, from a route node or a service callout action of another proxy service. In the indirect call scenario, the test console plays the role of the proxy service that routes to, or makes a callout to, another service.

Note: Using an indirect call to a request/response MQ proxy service will not work.

In addition, the test console does not display the response from an indirect call to an MQ or JMS request/response proxy service using a correlation based on a messageID. When you test an MQ or JMS request/response proxy service with an indirect call, the response is retained in the response queue, and not displayed in the test console.

For more information, see the *Native MQ Transport User Guide*.

HTTP Requests

When you test proxy services, the test console never sends a HTTP request over the network, therefore, transport-level access control is not applied. Transport-level access control is achieved through the Web application layer—therefore, even in the case that an indirect call is made through the ALSB Console transport layer, an HTTP request is not sent over the network and transport-level access control is not applied. For information about message processing in the transport layer, see Architecture Overview in *AquaLogic Service Bus Concepts and Architecture*.

For information about transport settings, see Understanding How the Run Time Uses the Transport Settings in the Test Console in *Using the AquaLogic Service Bus Console*.

Testing Business Services

You can test the following types of business services:

- WSDL Web Service
- Transport Typed Service
- Messaging Service
- Any Soap Service
- Any XML Service

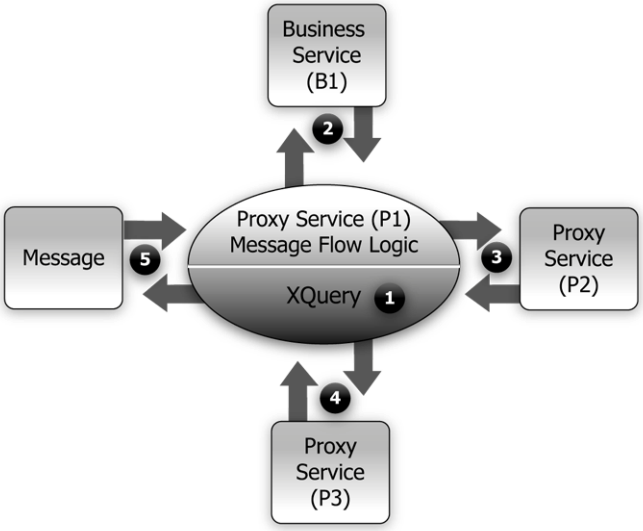
When testing business services, the messages are always routed through the transport layer. The direct call option is not available. The configuration data that you provide to the test console to test the service is that which represents the state of the message that is expected to be sent to that business service—for example, from a route node or a service callout action of a proxy service. The test console functions in the role of the caller proxy service when you use it to test a business service.

Tip: Ensure that the user name and password that you specify in the test console exists in the local ALSB domain even if the business service being tested is in a remote domain. The test service performs a local authentication before invoking any proxy or business service.

Recommended Approaches to Testing Proxy and Business Services

In the scenario depicted in Figure 6-3, a client invokes the proxy service (P1). The message flow invokes business service B1, then proxy service P2, then proxy service P3 before returning a message to the client. Interfaces are identified by number.

Figure 6-3 Test Scenario Example



There are many valid test strategies for this scenario. BEA recommends the following:

- Complete the testing of interfaces other than the client interface to a given proxy service before you test the client call. In the sample scenario illustrated in Figure 6-3, this means that you complete the testing of interfaces 1 through 4 first, then test interface 5. In this way, the message flow logic for the proxy service (P1) can be iteratively changed and tested (via interface 5) knowing that the other interfaces to the proxy service function correctly.
- Validate and test all the XQuery expressions in a message flow prior to a system test. In Figure 6-3, interface 1 refers to XQuery expression tests.
- Test proxy service to business service (interface 2 in Figure 6-3) using a *indirect call*. In other words, route the messages through the transport layer.

- Test proxy service to proxy service (interfaces 3 and 4 in Figure 6-3) using an *indirect call*. In other words, disable the direct call option, which means that during testing, the messages are routed through the transport layer.
- Make your final *system* test simulate the client invoking the proxy service P1. This test is represented by interface 5 in Figure 6-3. Test interface 5 with a direct call. In this way, during the testing, the messages bypass the transport layer. By default, tracing is enabled with a direct call.
- Save the message state after executing successful interface tests to facilitate future troubleshooting efforts on the system. Testing interface 5 is in fact a test of the complete system. Knowing that all other interfaces in the system work correctly helps narrow the troubleshooting effort when system errors arise.

Tracing Proxy Services Using the Test Console

Tracing the message through a proxy service involves examining the message context and outbound communications at various points in the message flow. The points at which the messages are examined are predefined by ALSB. ALSB defines tracing for stages, error handlers, and route nodes.

For each stage, the trace includes the changes that occur to the message context and all the services invoked during the stage execution. The following information is provided by the trace:

- added (New variables)—The names of all new variables and their values. View values by clicking on the + sign.
- deleted (Deleted variables)—The names of all deleted variables.
- changed (Changed variables)—The names of all variables for which the value changed. View the new value by clicking on the + sign.
- Publish—Every publish call is listed. For each publish call, the trace includes the name of the service invoked, and the value of the `outbound`, `header`, `body`, and `attachment` variables.
- Service callout—Every service callout is listed. For each service callout, the trace includes the name of the service that is invoked, the value of the `outbound` variable, the value of the `header`, `body`, and `attachment` variables for both the request and response messages.

The trace contains similar information for route nodes as for stages. In the case of route nodes, the trace contains the following categories of information:

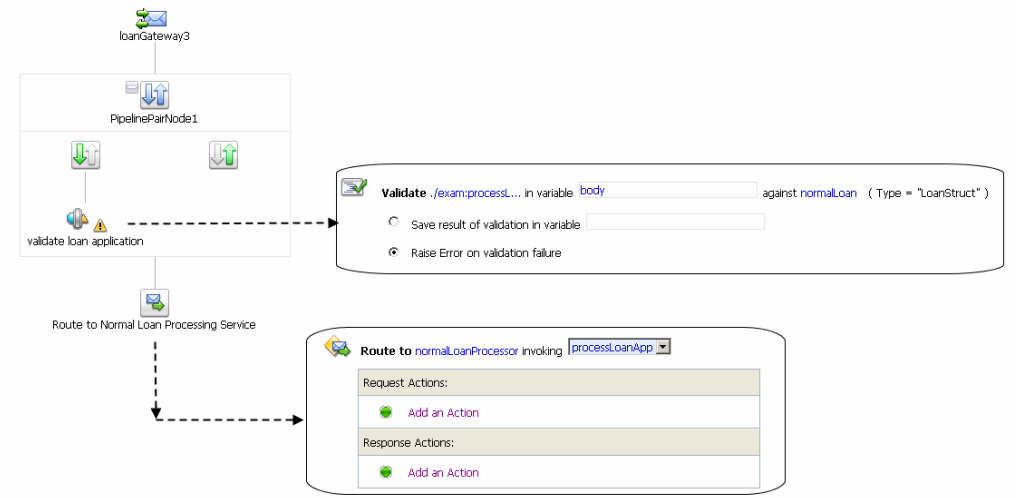
- The trace for service invocations on the request path
- The trace for the routed service
- The trace for the service invocations on the response path
- Changes made to the message context between the entry point of the route node (on the request path) and the exit point (on the response path)

Example: Testing and Tracing a Proxy Service

The following example scenario uses one of the proxy services in the ALSB Examples domain as a basis of instruction, the `loanGateway3` proxy service associated with the *Validating a Loan Application* example. For more information on how to start the Examples domain and run the examples, see BEA AquaLogic Service Bus Samples.

The message flow for `loanGateway3` is represented in Figure 6-4. The figure is annotated with the configuration for the *validate loan application* stage and route node.

Figure 6-4 Message Flow for Proxy Service (LoanGateway3)



To test this proxy service in the ALSB Examples domain using the test console, complete the following procedure:

1. Start the ALSB Examples domain and load the samples data, as described in BEA AquaLogic Service Bus Samples.

2. Log in to the ALSB Console, then select **Resource Browser** and locate the `LoanGateway3` proxy service.

3. Click the *Launch Test Console* icon for the `LoanGateway3` proxy service. The `Proxy Service Testing - LoanGateway3` page is displayed. Note that the **Direct Call** and the **Include Tracing** options are selected.

4. Edit the test XML provided to send the following test message, illustrated in Listing 6-1.

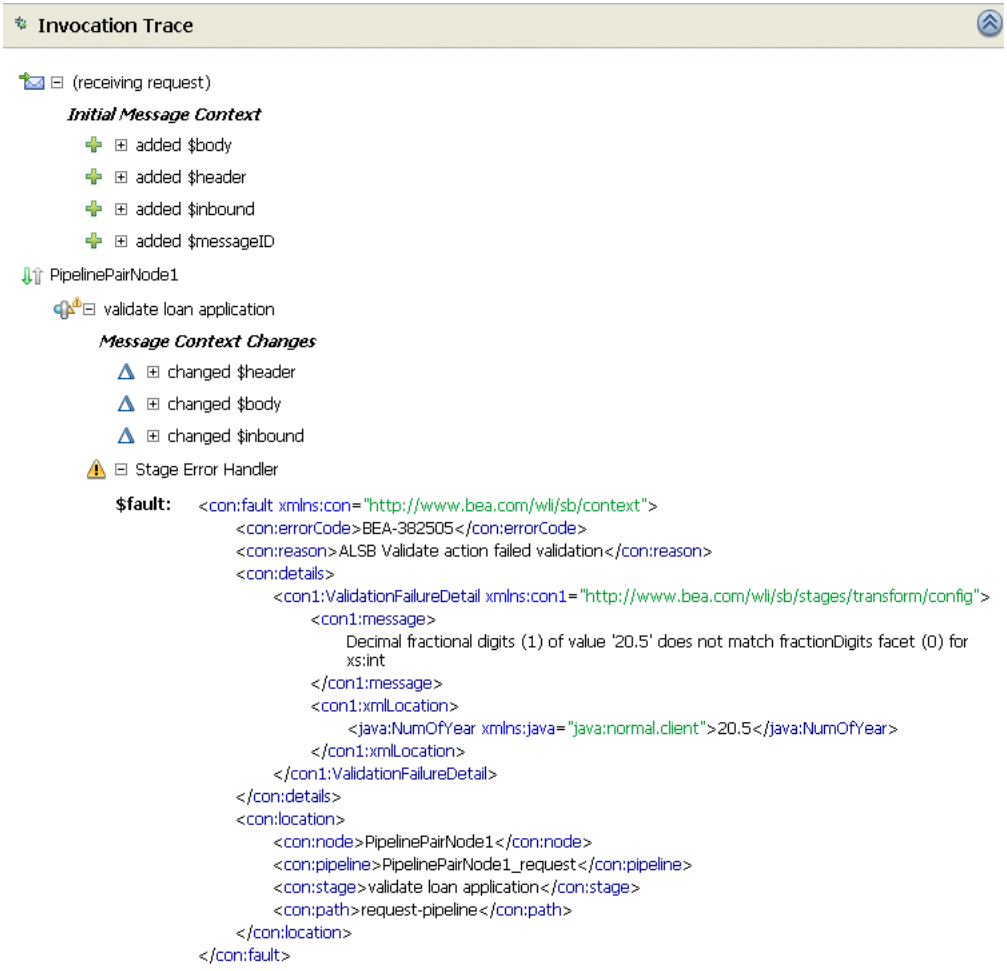
Listing 6-1 Test Message for LoanGateway3

```
<loanRequest xmlns:java=”java:normal.client”>
    <java:Name>Name_4</java:Name>
    <java:SSN>SSN_11</java:SSN>
    <java:Rate>4.9</java:Rate>
    <java:Amount>2500</java:Amount>
    <java:NumOfYear>20.5</java:NumOfYear>
    <java:Notes>Name_4</java:Notes>
</loanRequest>
```

5. Click **Execute**.

   Scroll to the bottom of the results page to view the tracing results in the **Invocation Trace** panel, shown in Figure 6-5.

Figure 6-5 Invocation Trace for LoanGateway3 Proxy Service



Compare the output in the trace with the nodes in the message flow shown in Figure 6-4.

The trace indicates the following:

- Initial Message Context—Shows the variables initialized by the proxy service when it is invoked. To see the value of any variable, click on the + sign associated with the variable name.

- Changed Variables—`$header` `$body` and `$inbound` changed as a result of the processing of the message through the `validate loan application` stage. These changes are seen at the end of the message flow.
- The contents of the `fault` context variable (`$fault`) is shown as a result of the stage error handler handling the validation error. The non-integer value (**20.5**) you entered for the `<java:NumOfYear>` element in Listing 6-1 caused the validation error in this case.

You can test the proxy service using different input parameters or by changing the message flow in the ALSB Console. Then run the test again and view the results.

For more information about this loan application scenario, see Tutorial 3: Validating a Loan Application in *AquaLogic Service Bus Tutorials*.

Testing Resources

You can test the following resources:

- "MFL" on page 6-11
- "XSLT" on page 6-13
- "XQuery" on page 6-13

MFL

A Message Format Language (MFL) document is a specialized XML document used to describe the layout of binary data. MFL resources support the following transformations:

- XML to binary—There is one required input (XML) and one output (binary).
- binary to XML—There is one required input (binary) and one output (XML).

Each transformation accepts only one input and provides a single output.

Example

The following example illustrates testing an MFL transformation. The test console generates a sample XML document from the MFL file. Execute the test using the XML input. A successful test results in the transformation of the message content of the input XML document to binary format.

Listing 6-2 shows an example MFL file.

Listing 6-2 Contents of an MFL File

```
<?xml version='1.0' encoding='windows-1252'?>
<!DOCTYPE MessageFormat SYSTEM 'mfl.dtd'>
  <MessageFormat name='StockPrices' version='2.01'>
   <StructFormat name='PriceQuote' repeat='*'>
    <FieldFormat name='StockSymbol' type='String' delim=':'
codepage='windows-1252'/>
    <FieldFormat name='StockPrice' type='String'
delim='|'codepage='windows-1252'/>
   </StructFormat>
  </MessageFormat>
```

The XML document generated by the test console to test the MFL file in the Listing 6-2 is shown in Listing 6-3.

Listing 6-3 Test Console XML Input

```
<StockPrices>
    <PriceQuote>
        <StockSymbol>StockSymbol_31</StockSymbol>
        <StockPrice>StockPrice_17</StockPrice>
    </PriceQuote>
</StockPrices>
```

In the test console, click **Execute** to run the test. Listing 6-4 shows the resulting test data, the stock symbol and stock price in binary format.

Listing 6-4 MFL Test Console Results

```
00000000:53 74 6F 63 6B 53 79 6D 62 6F 6C 5F 33 31 3A 53 StockSymbol_31:S
00000010:74 6F 63 6B 50 72 69 63 65 5F 31 37 7C .. .. .. StockPrice_17|...
```

XSLT

Extensible Stylesheet Language Transformation (XSLT) describes XML-to-XML mappings in ALSB. You can use XSL transformations when you edit XQuery expressions in the message flow of proxy services.

To test an XSLT resource, you must supply an input XML document. The test console returns the output XML document. You can create parameters in your document to assist with a transformation. XSLT parameters accept either primitive values or XML document values. You cannot identify the types of parameters from the XSL transformation. In the **Input and Parameters** panel of the XSLT Resource Testing page in the test console, you must provide the values to bind to the XSLT parameters defined in your document.

For more information, see Testing XSLT Transformations in *Using the AquaLogic Service Bus Console*.

XQuery

XQuery uses the structure of XML to express queries across different kinds of data, whether physically stored in XML or viewed as XML.

An XQuery transformation can take multiple inputs and returns one output. The inputs expected by an XQuery transformation are variable values to bind to each of the XQuery external variables defined. The value of an XQuery input variable can be a primitive value (String, integer, date), an XML document, or a sequence of the previous types. The output value can be a primitive value (String, integer, date), an XML document, or a sequence of the previous types.

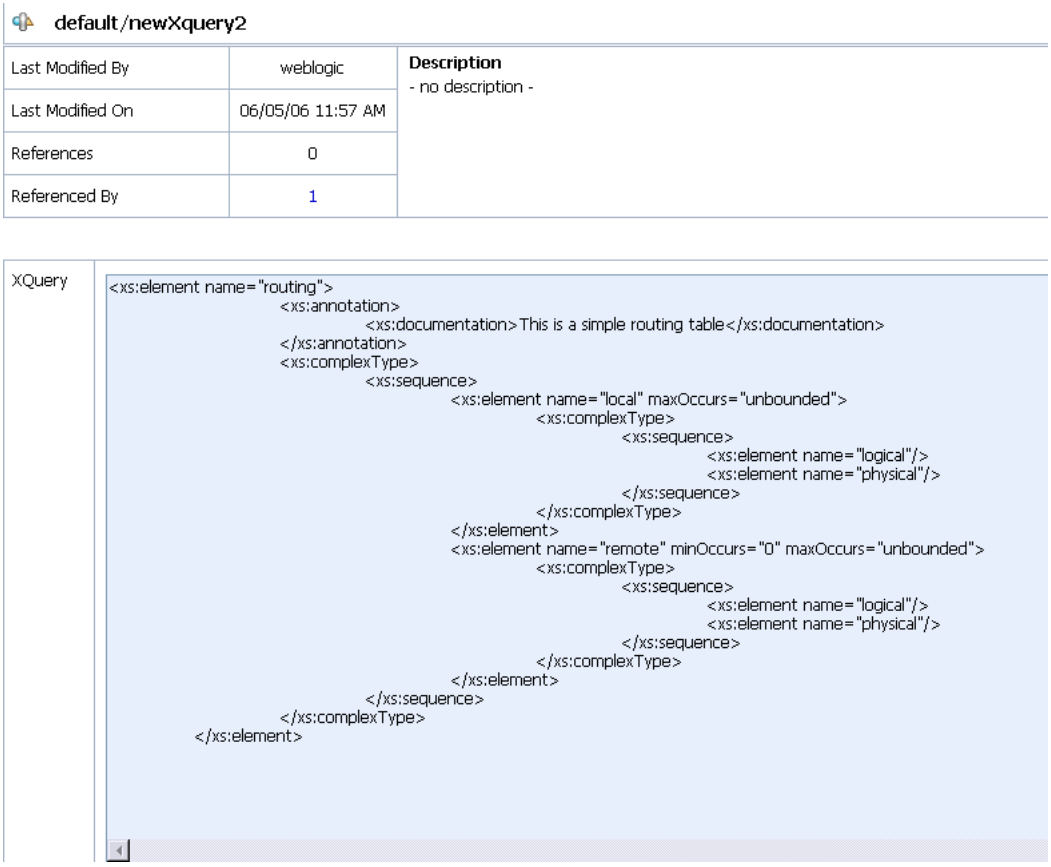
XQuery is a typed language—every external variable is given a type. The types can be categorized into the following groups:

- Simple/primitive type—String, int, float, and so on.
- XML nodes
- Untyped

In the test console, a single-line edit box is displayed if the expected type is a simple type. A multiple-line edit box is displayed if the expected data is XML. A combination input is used when the variable is not typed. The test console provides the following field in which you can declare the variable type: `[] as XML`. Input in the test console is rendered based on the type to make it easier to understand the type of data you must enter.

Figure 6-6 shows an XQuery with three variables: int, XML, and undefined type.

Figure 6-6 Input to the XQuery Test



In the test console, all three variables are listed in the **Variables** panel. By default, XML is selected for the untyped variable as it is the most typical case. You must configure these variables in the **Variables** panel. See Testing XQuery Transformations in *Using the AquaLogic Service Bus Console*.

You can also test an XQuery expression from the XQuery Editor.

Performing XQuery Testing

You must disable the pop-up blockers in your browser for the XQuery testing to work. Note that if you have toolbars in the Internet Explorer browser, you may need to disable pop-up blockers from under the browser **Options** menu as well as for all toolbars that are configured to block them.

When performing XQuery testing in the test console, use the **Back** button to execute a new test. However, if you want to execute a new test after making changes to the XQuery, you must close and re-open the test console for the changes to take effect. For detailed information, see Performing XQuery Testing in *Using the AquaLogic Service Bus Console*.

Testing Services With Web Service Security

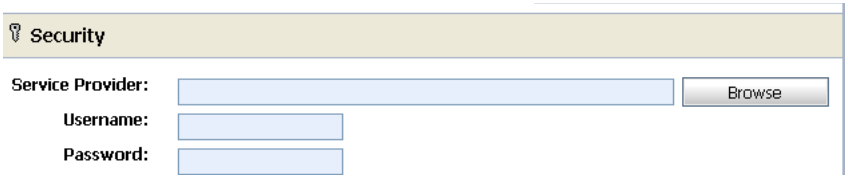
The test console supports testing proxy services and business services protected with Web Service Security (WSS). A SOAP service is protected with WSS if it has WS-Policies with WS-Security assertions assigned to it. Specifically, a service operation is protected with WS-Security if its effective request or response WS-Policy includes WS-Security assertions. WS-Policies are assigned to a service by WS-PolicyAttachment. See "Attaching WS-Policy Statements to WSDL Documents" in Using Web Services Policy to Specify Inbound Message-Level Security in the *AquaLogic Service Bus Security Guide.* Note that an operation may have both a request policy and a response policy.

When an operation has a request or response WS-Policy, the message exchange between the test console and the service is protected by the mechanisms of WS-Security. According to the operation's policy, the test service digitally signs or encrypts the message (more precisely, parts of the message) and includes any applicable security tokens. You specify the input to the digital signature and encryption operations is the clear-text SOAP envelope specified as described in "Configuring Proxy Service Test Data" and "Configuring Business Service Test Data" in Test Console in the *Using the AquaLogic Service Bus Console*.

Similarly, the service processes the response according to the operation's response policy. The response may be encrypted or digitally signed. The test service then processes this response and decrypts the message or verifies the digital signature.

The test console (**Security** panel) displays fields used for testing services with WS-Security: **Service Provider**, **Username**, and **Password**.

Figure 6-7 Security Panel in Test Console



If you specify a service key provider in the test console, all client-side PKI key-pair credentials required by WS-Security are retrieved from the service key provider. You use the user name and password fields when an operation's request policy specifies an Identity assertion and user name Token is one of the supported token types. For more information, see Web Service Policy.

The **Service Provider**, **Username**, and **Password** fields are displayed whenever the operation has a request or response policy. Whether the values are required depends on the actual request and response policies.

Table 6-1 and Table 6-2 describe security scenarios.

Table 6-1 Digital Signature and Encryption Scenarios

Scenario	Is Service Key Provider Required?
The request policy has a Confidentiality assertion.	**No**. The test service encrypts the request with the service's public key. When testing a proxy service, the test service automatically retrieves the public key from the encryption certificate assigned to the service key provider of the proxy service. When testing a business service, the encryption certificate is embedded in the WSDL of the business service. The test service automatically retrieves this WSDL from the WSDL repository and extracts the encryption certificate from the WSDL.
The response policy has a Confidentiality assertion.	**Yes**. In this scenario, the operation policy requires the client to send its certificate to the service. The service will use the public key from this certificate to encrypt the response to the client. A service key provider *must* be specified and *must* have an associated encryption credential. If both request and response encryption are supported, different credentials must be used.

Table 6-1 Digital Signature and Encryption Scenarios

The request policy has an Integrity assertion.	**Yes**. The client must sign the request. A service key provider *must* be specified and *must* have an associated digital signature credential. Furthermore, if this is a SAML holder-of-key integrity assertion, a user name and password is needed in addition to the service key provider.
The response policy has an Integrity assertion.	**No**. In this case, the policy specifies that the service must sign the response. The service signs the response with its private key. The test console simply verifies this signature. When testing a proxy service, this is the private key associated to the service key provider's digital signature credential for the proxy service. When testing a business service, the service signing key-pair is configured in a product-specific way on the system hosting the service. In the case that the current security realm is configured to do a Certificate Lookup and Validation, then the certificate that maps to the service key provider must be registered and valid in the certificate lookup and validation framework. For more information on Certificate Lookup and Validation, see "Configuring the Credential Lookup and Validation Framework" in Configuring WebLogic Security Providers in *Securing WebLogic Server*.

Table 6-2 Identity Policy Scenarios (Assuming that the Policy has an Identity Assertion)

Supported Token Types[1]	Description	Comments
UNT	The service only accepts WSS user name tokens	You must specify a user name and password in the **Security** panel.
X.509	The service only accepts WSS X.509 tokens	You must specify a service key provider in the **Security** panel and the service key provider must have an associated WSS X.509 credential.

Table 6-2 Identity Policy Scenarios (Assuming that the Policy has an Identity Assertion)

Supported Token Types[1]	Description	Comments
SAML	The service only accepts WSS SAML tokens	You must specify a user name and password in the **Security** panel *or* a user name and password in the **Transport** panel. If both are specified, the one from the **Security** panel is used as the identity in the SAML token.
UNT, X.509	The service accepts UNT or X.509 tokens	You must specify a user name and password in the **Security** panel *or* a service key provider in the **Security** panel with an associated WSS X.509 credential. If both are specified, only a UNT token is generated.
UNT, SAML	The service accepts UNT or SAML tokens	You must specify a user name and password in the **Security** panel *or* a user name and password in the **Transport** panel. If both are specified, only a UNT token is sent.
X.509, SAML	The service accepts X.509 or SAML tokens	You must specify one of the following: • user name and password in the **Security** panel • user name and password in the **Transport** panel • service key provider with an associated WSS X.509 credential
UNT, X.509, SAML	The service accepts UNT, X.509 or SAML tokens	You must specify one of the following: • user name and password in the **Security** panel • user name and password in the **Transport** panel • service key provider with an associated WSS X.509 credential

1. From the Identity Assertion inside the request policy.

Limitations for Services and Policies

The following limitations exist for testing proxy services with SAML policies and business services with SAML holder-of-key policies:

- Testing proxy services with inbound SAML policies is not supported.
- Testing business services with a SAML holder-of-key policy is a special case. The SAML holder-of-key scenario can be configured in two ways:
  - as an integrity policy (this is the recommended approach)
  - as an identity policy

In both cases, you must specify a user name and password—the SAML assertion will be on behalf of this user. If SAML holder-of-key is configured as an integrity policy, you must also specify a service key provider. The service key provider must have a digital signature credential assigned to it. This case is special because this is the only case where a user name and password must be specified even if there is not an identity policy.

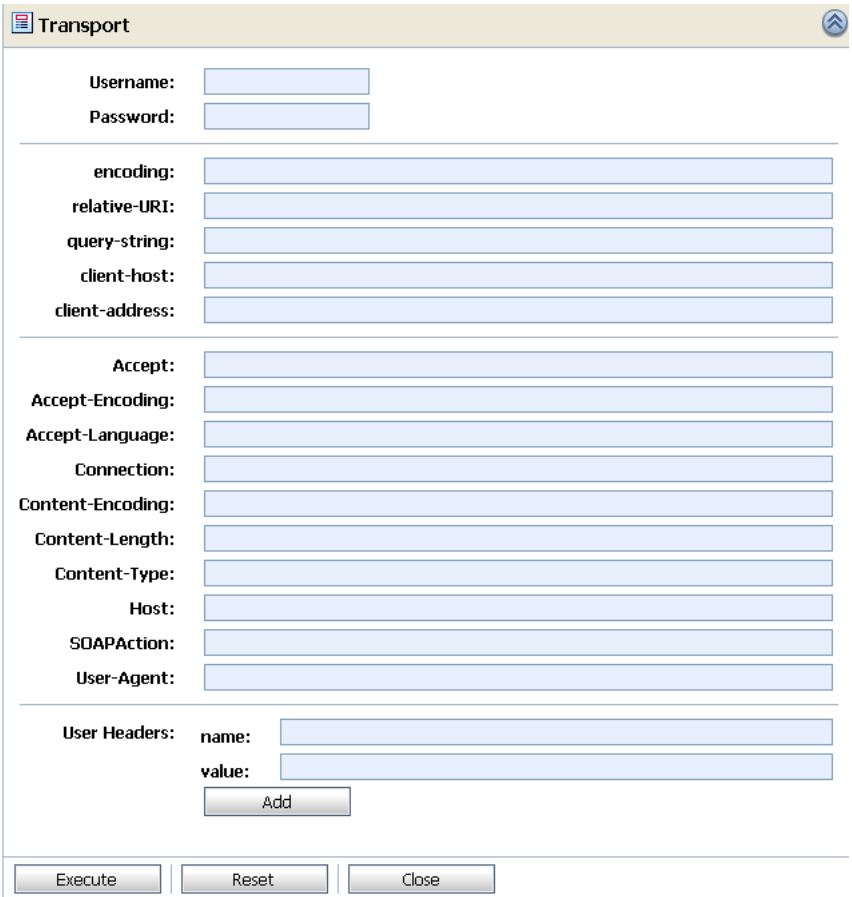
Note: After executing a test in the test console, the envelope generated with WSS is not always a valid envelope—the results page in the test console includes white spaces for improved readability. That is, the secured SOAP message is displayed with extra white spaces. Because white spaces can affect the semantics of the document, this SOAP message cannot always be used as the literal data. For example, digital signatures are white-space sensitive and can become invalid.

Test Console Transport Settings

You use the **Transport** panel in the test console to specify the metadata and transport headers for messages in your test system.

Figure 6-8 shows the **Transport** panel for a WSDL-based proxy service.

Figure 6-8 Transport Panel in the Test Console



By setting the metadata and the transport headers in the message flow of a proxy service, you influence the actions of the outbound transport. You can test the metadata, the message, and the headers so that you can view the pipeline output. The fields that are displayed in the **Transport** panel when testing a proxy service represent those headers and metadata that are available in the

pipeline. The test console cannot filter the fields it displays depending on the proxy service. The same set of transport parameters are displayed for every HTTP-based request.

The **Username** and **Password** fields are used to implement basic authentication for the user that is running the proxy service. The **Username** and **Password** fields are not specifically transport related.

Metadata fields are located below the **Username** and **Password** fields and above the transport header fields. The fields displayed are based on the transport type of the service. Certain fields are pre-populated depending on the operation selection algorithm you selected for the service when you defined it.

For example, in the **Transport** panel displayed in Figure 6-8, the `SOAPAction` header field is populated with "`http://example.orgprocessLoanApp`". This value comes from the service definition (the selection algorithm selected for this proxy service was `SOAPAction Header`). For more information about the selection algorithms, see "Modeling Message Flow in ALSB" on page 3-1.

Specify the values in the **Transport** panel fields according to whether the message will be processed through the transport layer (for example, if you are testing the service using a direct call), or not (an indirect call).

When testing a proxy service with a direct call, the test data must represent the message as if it had been processed through the transport layer. That is, the test data should represent the message in the state expected at the point it leaves the transport layer and enters the service. When testing a proxy or business service using an indirect call, the test data represents the data that is sent from a route node or a service callout. The test message is processed through the transport layer.

For information about specific headers and metadata and how they are handled by the test framework, see Understanding How the Run Time Uses the Transport Settings in the Test Console in *Using the AquaLogic Service Bus Console*.

About Security and Transports

- When using the test console to test HTTP business services with BASIC authentication, the test console authenticates the user name and password from the service account of the business service. Similarly, when testing JMS, e-mail, or FTP business services that require authentication, the test console authenticates the service account associated with the business service.
- When you test proxy services, the test console never sends a HTTP request over the network. Therefore, transport-level access control is not applied.

CHAPTER 7

UDDI

This section contains the following topics:

UDDI, UDDI Registries, and Web Services

UDDI stands for Universal Description, Discovery, and Integration. The UDDI Project is an industry initiative which aims to enable businesses to quickly, easily, and dynamically find and carry out transactions with one another.

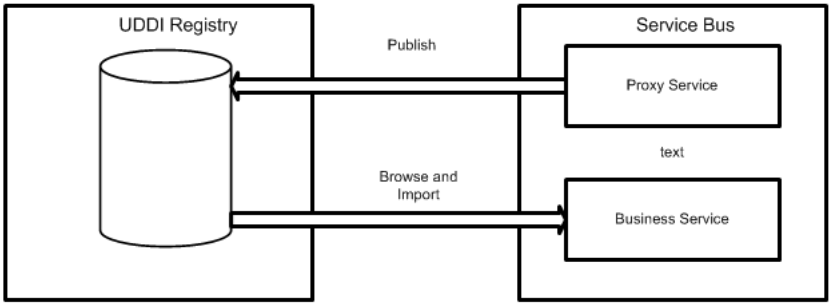
A populated UDDI registry contains cataloged information about businesses; the services that they offer; and communication standards and interfaces they use to conduct transactions. A UDDI registry provides a standards-based foundation infrastructure for locating services, invoking services, and managing metadata about services (security, transport, or quality of service). The UDDI registry can store and provide these metadata using arbitrary categorizations. These categorizations are called taxonomies.

UDDI registries are used in an enterprise to share Web Services. Using UDDI registries helps companies organize and catalog Web Services for sharing and reuse in an enterprise or with trusted external partners. The UDDI version 3.0 specification is available at: `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3`. UDDI registries are based on this specification, which provides details on how to publish and locate information about Web Services using UDDI. The specification does not define run-time aspects of the services (it is only a directory of the services). UDDI provides a framework in which to classify your business, its services, and the technical details about the services you want to expose.

Publishing a service to a registry requires knowledge of the service type and the data structure representing that service in the registry. A registry entry has certain properties associated with it and these property types are defined when the registry is created. You can publish your service to a registry and make it available for other organizations to *discover* and use. Proxy services developed in ALSB can be published to a UDDI registry. ALSB can interact with any version 3.0-compliant UDDI registry. BEA provides the AquaLogic Service Registry.

Figure 7-1 illustrates the integration of ALSB with a UDDI registry.

Figure 7-1 ALSB integration with UDDI



The ALSB Web-based interface to AquaLogic Service Registry makes the registry accessible and easy to use. In working with UDDI, ALSB promotes the reuse of standards-based Web Services. In this way, ALSB registry entries can be searched for, discovered, and used by multiple domains. Web Services and UDDI are built on a set of standards, so reuse promotes the use of acceptable, tested Web Services and application development standards across the enterprise. The Web Services and interfaces can be catalogued by type, function, or classification so that they can be discovered and managed more easily.

Basic Concepts of the UDDI Specification

UDDI is based upon several established industry standards, including HTTP, XML, XML Schema Definition (XSD), SOAP, and WSDL. The UDDI specification describes a registry of Web Services and its programmatic interfaces. UDDI itself is a set of Web Services. The UDDI specification defines services that support the description and discovery of:

- Businesses, organizations, and other Web Services providers
- The Web Services they make available
- The technical interfaces that can be used to access and manage those services

Benefits of Using a UDDI Registry with ALSB

A UDDI registry stores data and metadata about business services. A UDDI registry offers a standards-based mechanism to classify, catalog, and manage Web Services so that they can be discovered and consumed by other applications. UDDI offers several benefits to IT managers and developers at both design time and run time, including the following:

- UDDI improves infrastructure management by publishing information about services to the registry and categorizing the services for discovery. This ability of UDDI to categorize a growing portfolio of services makes it easier to manage them and helps you to understand relationships among components, supports versioning, and manages dependencies.
- You can import UDDI services from a registry to configure the parameters required to invoke the Web Service and the necessary transport and security protocols.
- UDDI promotes the use of standards-based Web Services and business services development in business applications and provides a link to a library of resources for Web Services developers. This decreases development time and improves productivity. It also increases the prospect of interoperability between business applications by sharing standards-based resources.
- UDDI provides a user-friendly interface for searching and discovering Web Services.

Introduction to UDDI Entities

UDDI uses a specific data model to represent entities that define organizations and services. Figure 7-2 shows the relationships between different UDDI entities.

Figure 7-2 UDDI Entities Representing Organizations and Services

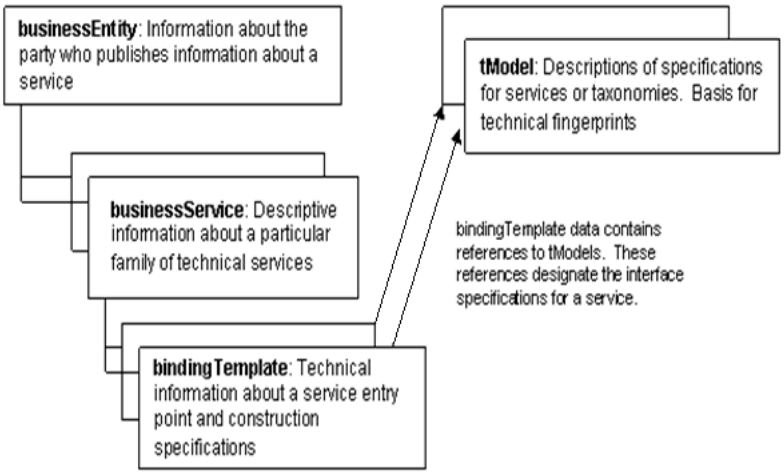


Table 7-1 provides a high-level overview of UDDI entities.

Table 7-1 High-Level Description of UDDI Entities

Business Entity	An organization or group of people who own and provide the services. A business entity can be described by a set of names, descriptions, contact details for the service provider, a set of categories that represent the business entity features, unique identifiers, and discovery URLs.
Business Service	Represents functionality or resources provided by a business entity. A business service is described by a name, a description, and a set of categories that represent the function of the service. A business service in a UDDI registry does not necessarily represent a Web Service. The UDDI registry can register arbitrary services, for example EJB, CORBA, and such.
Binding Template	Represents the technical details of how to invoke a business service. A business service can contain one or more binding templates. Binding templates are described by access points representing service endpoints (the endpoint URI and protocol specification), tModel instance information, and categories to reference specific features of the binding template.
`tModel`	Represents a technical specification; typically a specifications pointer, or metadata about a specification document, describing how services must be represented in the UDDI registry. The description of a service includes a name, a description, an overview document (a reference to a document specifying the purpose of the `tModel`), a category, and an identifier (to uniquely identify the `tModel`).

For more information on the UDDI data model and entities used in UDDI, see Introduction to BEA AquaLogic Service Registry in *BEA AquaLogic Service Registry 3.0 User's Guide*. See also Publishing and Finding Web Services Using UDDI in *WebLogic Web Services: Advanced Programming.*

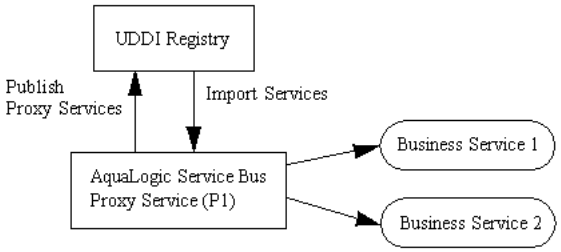
Sample Business Scenarios for ALSB and UDDI

The following are two sample business scenarios that highlight the benefit of using UDDI.

Basic Proxy Service Communication with a UDDI Registry

This scenario shows how you can use ALSB to import services from a registry and then publish ALSB proxy services back to the registry. See Figure 7-3.

Figure 7-3 Proxy Service Communication with a UDDI Registry

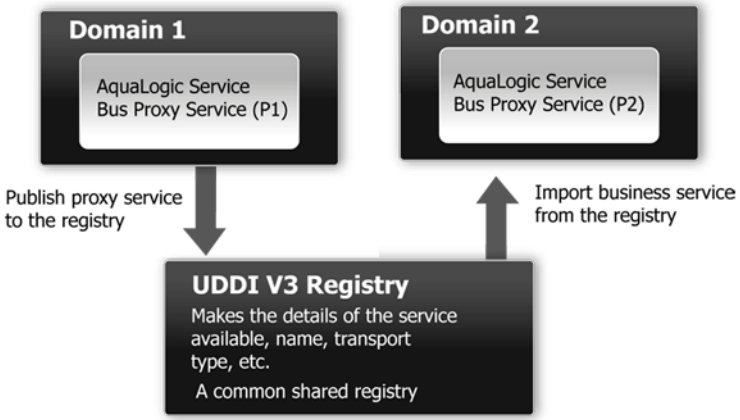


ALSB imports business services from a UDDI registry. Proxy services are configured to communicate with the business services in the proxy service message flow. The proxy services can then be published back to the registry and made available for use by other domains.

Cross-Domain Deployment in ALSB

This scenario shows cross-domain deployment using ALSB. In this scenario, an ALSB application in one domain requires access to an ALSB service in another domain at run time. See Figure 7-4.

Figure 7-4 Sample Business Case of Cross-Domain Deployment



An instance of ALSB is deployed in each of two domains. The ALSB proxy service (P1) is configured in domain (D1). The ALSB proxy service (P2) in domain (D2) requires access to proxy service (P1). As the domains cannot communicate directly with each other, P2 in D2 cannot use P1 in D1. The ALSB import and export feature does not support run-time discovery

of services in different domains, but publishing the service to a UDDI registry allows the discovery and use of a service in any domain. Once P1 is made available in the UDDI registry it can be invoked at run time (for example, get a stock quote) and imported as a business service in another ALSB proxy service.

When importing and exporting from different domains you should have network connectivity. A proxy service might reference schemas located in the repository of a different domain, in which case you need HTTP access to the domain to import it using the URL. In the absence of connectivity an error message is returned.

Using ALSB and UDDI

ALSB works with any UDDI registry that is compliant with the version 3.0 implementation of UDDI. AquaLogic Service Registry 2.1 is a V3.0-compliant UDDI registry and is certified to work with ALSB.

Using the ALSB Console or ALSB Plug-in for WorkSpace Studio, you can:

- Configure ALSB to work with one or more V3.0-compliant UDDI registries.
- Configure a registry to allow users to publish services and import services.
- Publish information about any proxy service to a registry, including the following service types: WSDL, messaging, any SOAP, and any XML.
- Search for specific services in a registry or list all services available. You can search on business entity, service name pattern, or both.
- Import business services from a registry.

For detailed procedural information, see the following topics in *Using the AquaLogic Service Bus Console*:

- Configuring UDDI Registries
- Importing a Business Service from UDDI Registry
- Publishing a Proxy Service to a UDDI Registry

A UDDI Workflow

The typical workflow for using a UDDI registry with ALSB is as follows:

- Install ALSB. See *AquaLogic Service Bus Installation Guide*.

- Install AquaLogic Service Registry or any V3.0-compliant UDDI registry. For information on installing BEA AquaLogic Service registry 3.0, see *AquaLogic Service Registry Installation Guide*.

  Note: AquaLogic Service Registry is not provided with ALSB. In order to use AquaLogic Service Registry you have to buy a separate licence from BEA. For more information on the management of AquaLogic Service Registry, particularly configuring the registry and managing permissions, approval, and replication, see BEA AquaLogic Service Registry Administrator's Guide.

- Configure the registry in the ALSB Console or the ALSB Plug-in for WorkSpace Studio. See
  - Configuring UDDI Registries in *Using the AquaLogic Service Bus Console.*
  - Configuring UDDI Registries in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio.*
- Set a default registry in the ALSB Console. See Setting Up a Default UDDI Registry in *Using the AquaLogic Service Bus Console.*

Configuring a Registry

You can configure a UDDI registry, make it available in ALSB, and then publish ALSB proxy services to it or import business services from the registry to be used in a proxy service. You must be in an active session in the ALSB Console to configure the registry. For detailed information, see:

- Configuring UDDI Registries in *Using the AquaLogic Service Bus Console*
- Configuring UDDI Registries in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio.*

When publishing services to AquaLogic Service Registry, you need a valid user name and password for authentication to gain access to the registry. The user name and password combination is implemented as a service account resource in ALSB. You must define service accounts before configuring proxy services. See Specifying Service Accounts in *Using the AquaLogic Service Bus Console.*

You can set up registries with multiple user names and passwords allowing different users to have different permissions based on the associated service accounts. In BEA AquaLogic Service Registry, administrators manage user privileges and create views into the registry, specific to the

needs of different users. In ALSB, user permissions govern access to the registries, their content, and available functionality.

Publishing a Proxy Service to a UDDI Registry

You can use the ALSB Console or the ALSB Plug-in for WorkSpace Studio to publish proxy services to the AquaLogic Service Registry. To do this you must have a user account set up in AquaLogic Service Registry. You can publish any proxy service to a UDDI registry. The permitted service types and transports are listed in Table 7-2.

Table 7-2 Service Types and Transports for Proxy Services

Service Type	Transports
WSDL	HTTP, JMS, Local, SB, WS
Any SOAP	HTTP, JMS, Local, SB
Any XML	E-mail, File, FTP, HTTP, JMS, Local, MQ, SB, SFTP, Tuxedo
Messaging	E-mail, File, FTP, HTTP, JMS, Local, MQ, SFTP, Tuxedo

Note: Messaging services can have different content for requests and responses, or can have no response at all (one-way messages). E-mail, File, SFTP, and FTP should be one-way.

You can select the Business Entity under which a service is to be published. Business Entity Administration (including creation, removal, update, and deletion of entities) is done using the management console provided by the registry vendor (for example, the Business Service Console of AquaLogic Service Registry). The first time you publish to a registry you must load the `tModels` to that registry. You do this when you configure the publishing details in the ALSB Console or ALSB Plug-in for WorkSpace Studio. For more information on how to publish to a UDDI registry, see Publishing Proxy Services to a UDDI Registry in *Using the AquaLogic Service Bus Console*.

Note: An error can occur when you attempt to import a service from a UDDI registry if that service was originally published to the registry from an ALSB cluster in which *any* of the clustered servers uses the localhost address. Specifically, when the service being imported references a resource (WSDL or XSD) which references other resources (WSDL or XSD).

Ensure that before you publish services to a UDDI registry from a clustered domain, none of the servers in the cluster use localhost in the server addresses. Instead, use either the machine name or the IP address.

Publishing Local Proxy Services to UDDI

You can now publish local proxy services to UDDI so you can associate them with ALSB generic proxy services. For example, you might have an any SOAP or any XML generic proxy service that dynamically routes to multiple local transport proxy services with concrete WSDLs. Alternatively, you might have a generic proxy service in Enterprise Service Bus (ESB) 1 that dynamically routes to a generic proxy service in ESB 2 where the business service is attached. From the UDDI registry, you can get the WSDL of the local proxy service and the URL of the any SOAP or any XML generic proxy service. Combining the WSDL and URL creates an effective WSDL for sending messages to the local proxy service through the generic proxy service.

Using Auto-Publish

When you create a proxy service you can configure it to be published automatically to a default UDDI registry. You must first set up a default registry. See Setting Up a Default UDDI Registry in *Using the AquaLogic Service Bus Console*.

To enable the auto-publish feature for individual proxy services, you select the **Publish To Registry** check box on the **Create a Proxy Service-General Configuration** page. When you enable the **Publish To Registry** option, the proxy service is published to the default registry upon session activation. If the UDDI registry is unavailable, the publish action is retried. Any further changes to the proxy service resets the retry attempts. When a proxy service is republished to a UDDI registry, all taxonomies and categorizations, which are defined in UDDI for the proxy service, are preserved.

When you change the default registry, all the proxy services that have auto-publish enabled will be published to the new default registry. Synchronization then takes place with the current default registry. When a proxy service is not synchronized, the ALSB Console displays a  **unsynchronized** icon.

Note: When you have a default registry and you import a `sbconfig.jar`, which has a default registry set with the same logical name during the import, it is possible that the default registry will have an incorrect value for the business entity. You might now see errors on

the **Auto Publish Status** page, if there are any auto-published proxy services. You can correct this situation by selecting the default registry again.

Importing a Service from a Registry

You can import services from a registry as ALSB business services. When importing a WSDL-based service, if multiple UDDI binding templates are encountered, ALSB creates a different business service for each binding template.

To establish access to UDDI registries in ALSB you must have ALSB IntegrationAdmin or IntegrationDeployer privileges. See Role-Based Access in AquaLogic Service Bus Console in the *AquaLogic Service Bus Security Guide*. The registry entries are located on the **System Administration > Import from UDDI** page in the ALSB Console. When importing, you select from the list of available registries. To discover a service in a registry you must query a specific registry. Entries in registries are unique. The query is performed when you specify what registry you want to use for importing a service.

You can import the following business services types from a UDDI registry into ALSB:

- WSDL over HTTP binding. When multiple UDDI binding templates are present, a business service is created for each binding template.
- SOAP or XML binding over HTTP.
- Services that are categorized as ALSB services. These are ALSB proxy services that are published to a UDDI registry. This feature is primarily used in multi-domain ALSB deployments where proxy services from one domain need to discover and route to proxy services in another domain.

For information on how to use the ALSB Console to import services from a UDDI registry, see:

- Importing Business Services from a UDDI Registry in *Using the AquaLogic Service Bus Console*
- Importing Business Services from a UDDI Registry in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*

When a service is updated, you must re-import the service from the registry to get the most recent version, unless you have selected the **Enable Auto Import** option to auto-synchronize imported services with the UDDI registry. Any service that is imported with this option selected will be kept in synchrony with the UDDI registry. See "Auto-Synchronization of Services With UDDI" on page 7-14. If there is any failure during auto-synchronization, it will be reported on the **Auto-Import Status** page where you can update it manually.

Services have documents associated with them and these documents can include a number of other documents (schemas, policies, and so on). On import, the UDDI registry points to the document location based on the inquiry URL of the service. When a document that includes or references other resources is located, all of the referenced information and each included item is added as a separate resource in ALSB.

Business Entity and pattern are the criteria used to search for a service in a registry. For example, you can enter `foo%`, when searching for a service. Services published by ALSB have specific `tmodel` keys identifying the services that you use when searching for the service in the registry.

The Business Entity is the highest level of organization in the registry, though you can use other search criteria, such as business, application type, and so on. If you require authentication, then you need a user name and password which you must get from your system administrator.

Related References

- Technical Notes can be found at `http://www.oasis-open.org/committees/uddi-spec/doc/tns.htm`. The note on *Using WSDL in a UDDI Registry* is important.
- UDDI product and development tool information is available at the OASIS UDDI Solutions page at `http://uddi.org/solutions.html`.
- The UDDI specifications `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm`

  The specification defines the following:

  – SOAP APIs that applications use to query and to publish information to a UDDI registry

  – XML schema of the registry data model and the SOAP message formats

  – WSDL definitions of the SOAP APIs

  – UDDI registry definitions (`tModels`) of various identifier and category systems that may be used to identify and categorize UDDI registrations

Using Auto-Import

You can use the auto-import feature to synchronize the business services, which are imported from the AquaLogic Service Registry, with the corresponding services in the registry. See Using Auto-Import Status in *Using the AquaLogic Service Bus Console*.

Note: Auto-import is available only in the ALSB Console, not in the ALSB Plug-in for WorkSpace Studio.

You can use the **Auto Import Status** page to do the following:

- "Synchronize" on page 7-13
- "Detach" on page 7-14

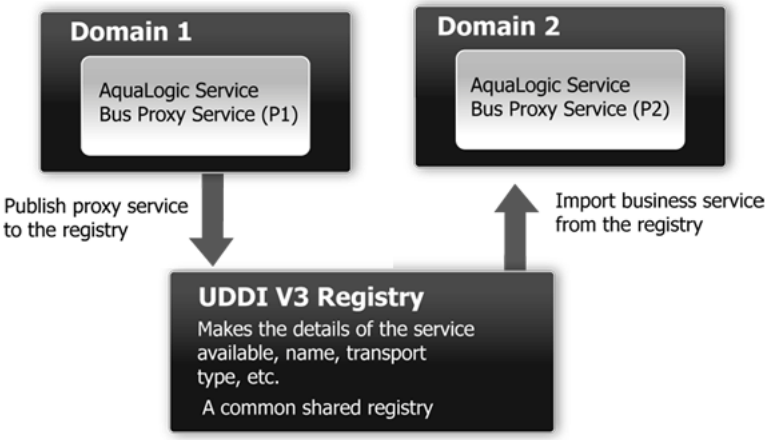
Synchronize

You can synchronize the services you have imported from the registry. If the services in the registry change, you can synchronize services in the ALSB Console with those in the registry. The following use case illustrates the process of synchronization. If the business service is not detached from the registry, ALSB automatically subscribes to any changes to the service in the registry. If the service changes, a  **unsynchronized** icon appears in the **Resource Browser** and **Project Explorer** indicating that the service needs to be synchronized. In addition, the **Auto Import Status** page shows this service and provides options for synchronizing the service or detaching it from the registry. Under certain circumstances, synchronizing the service might result in semantic validation errors that show up on the **View Conflicts** page. You will have to fix these errors before activating the session.

When a service is synchronized, the service is updated only with fields that are obtained from UDDI. Other fields in the service definition will preserve their values if modified since last import.

Consider a scenario where you publish services from Domain1 to a registry (see Figure 7-5). You then import these services from the registry into a domain, Domain2. Then you make changes to the services in Domain1 and update them in the registry. You can update the services in Domain2 by synchronizing them with the registry using the auto-import feature.

Figure 7-5 Sample Business Case of Cross-Domain Deployment



Detach

Sometimes you do not want the service in the ALSB Console to be synchronized with the corresponding service in the registry. You can avoid synchronization by detaching the service from the registry. See Detaching Services in *Using the AquaLogic Service Bus Console.*

Auto-Synchronization of Services With UDDI

You can keep the service definitions in ALSB automatically synchronized (both ways) with those in UDDI.

Services can be automatically published to a UDDI registry after they are created or changed within ALSB and business service definitions can be imported from UDDI and automatically updated when the original service is changed in UDDI. Alternatively, you can configure the ALSB Console or the ALSB Plug-in for WorkSpace Studio to prompt you for approval for synchronization when a service changes in the UDDI registry.

When configuring a registry, select the **Enable Auto Import** option to auto-synchronize imported services with the UDDI registry. Any service that is imported with this option enabled will be kept in synchrony with the UDDI registry automatically. If there is any failure during auto-synchronization, it is reported on the **Auto-Import Status** page where you can update it manually. See Configuring UDDI Registries in *Using the AquaLogic Service Bus Console.*

Mapping ALSB Proxy Services to UDDI Entities

ALSB proxy service attributes must be mapped to the data model supported by the UDDI registry to allow a proxy service to be published as a UDDI business entity. The following table shows the service types, message types, and transports relevant to the UDDI registry mapping for an ALSB proxy service.

Table 7-3 Proxy Service Attributes and Service Types

Service Type	Message Content Type	Transports
WSDL	SOAP or XML (with attachment)	HTTP, JMS, Local, SB, WS
Any SOAP	Untyped SOAP (with attachment)	HTTP, JMS, Local, SB
Any XML	Untyped XML (with attachment)	E-mail, File, FTP, HTTP, JMS, Local, MQ, SB, SFTP, Tuxedo
Messaging	Binary, Text, MFL, XML (schema)	E-mail, File, FTP, HTTP, JMS, Local, MQ, SFTP, Tuxedo

Note: Optional parts are listed in parentheses. Messaging services can have different content for requests and responses, or can have no response at all (one-way messages). E-mail, File, SFTP, and FTP should be one-way.

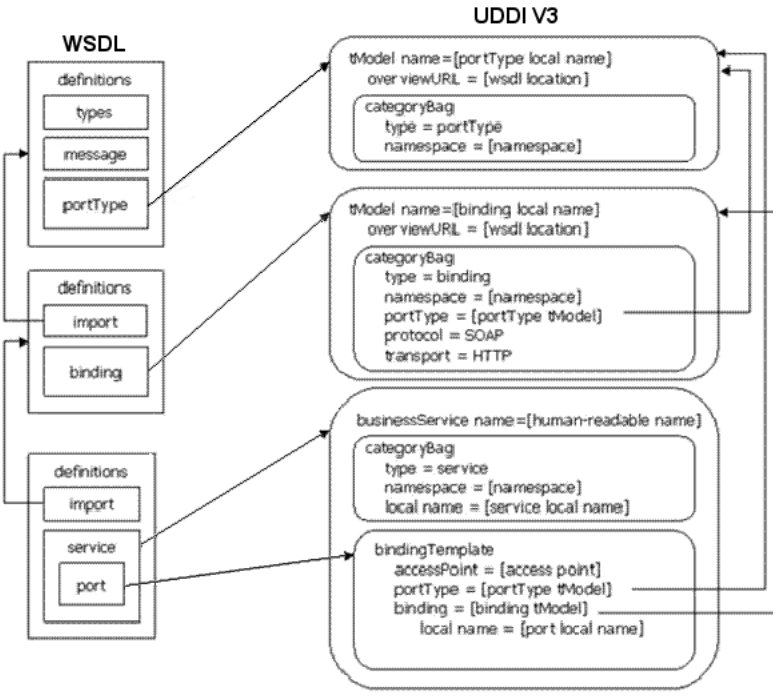
Proxy services have attributes in common and also attributes that are specifically defined by the transport protocols used by the service and the type of service. Each proxy service can deliver messages of a certain type.

The primary relevant entities in UDDI are:

- businessService: represents the service as a whole and contains high-level general information about the service.
- bindingTemplate: contains information for accessing the service.
- `tModels`: supplies the individual attributes for categorizing and defining the service.

Figure 7-6 shows how WSDL-based services are mapped to UDDI business entities.

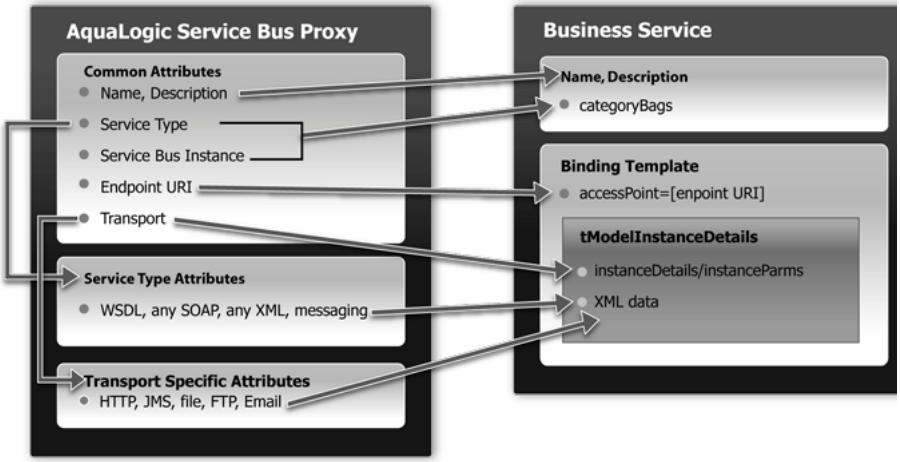
Figure 7-6 WSDL Service to UDDI Mapping



The technical note on *Using WSDL in a UDDI registry, version 2.0.2*, at `http://www.oasis-open.org/committees/uddi-spec/doc/tns.htm`, is used as the basis for publishing WSDL-based proxy services to the UDDI registry. This document is also used as a reference point for publishing non-WSDL based services. The document and the base UDDI specification describe the canonical technical models (`tModels`) used to describe UDDI entities. To publish ALSB proxy services as entities in the UDDI registry, you must provide additional canonical tModels to support some of the constructs specific to ALSB. Not all attributes of an ALSB proxy service are useful when searching for a service, for example, service type and transport details. These attributes do not categorize the service. `tmodels` are configuration details of the service once it has been discovered. These configuration details are mapped to the business service binding template `tmodelinstanceDetails` section. Other attributes specifically identify a service and can be used as the search criteria for the service. These attributes are mapped using keyed references to `tModels` with values in the `categoryBag` of the binding template.

An example of how ALSB maps to UDDI is shown in Figure 7-7.

Figure 7-7 ALSB to UDDI Mapping



UDDI Mapping Details for an ALSB Proxy Service

ALSB high-level proxy service information maps to the business service as follows:

- Name and Description map to `businessService elements`.
- There is a special `keyedReferenceGroup` for ALSB properties. An example of a key is `uddi:bea.com:attributes:aqualogicservicebus`.
- ALSB type (WSDL, SOAP, XML, and Mixed) and instance are mapped to `keyedReferences` in the service category. An example of a key is `uddi:bea.com:servicetype`.
- An ALSB instance maps to a `keyedReference` in the ALSB `keyedReferenceGroup` (Name = "`AquaLogicServiceBus`", Values = URL of the ALSB instance).

  This instance serves two purposes:

  – To indicate that this service is in fact hosted by an ALSB server.

  – To contain the URL of the ALSB instance.

Listing 7-1 shows a mapping of high-level proxy service information to a business service.

Listing 7-1 Sample Proxy Service to Business Service Mapping

```
<keyedReferenceGroup tModelKey="uddi:bea.com:servicebus:properties">
  <keyedReference  tModelKey="uddi:bea.com:servicebus:servicetype"
    keyName="Service Type"
    keyValue="SOAP"/>
  <keyedReference  tModelKey="uddi:bea.com:servicebus:instance"
    keyName="Service Bus Instance"
    keyValue="http://FOO02.amer.bea.com:7001"/>
</keyedReferenceGroup>
```

Note: The key for the businessService created when a proxy service is published is a publisher assigned key name. It is derived from the ALSB domain name, the path of the proxy service, and the proxy service name. It takes the following form:

`uddi:bea.com:servicebus:<domainname>:<path>:<servicename>`.

For example, AnonESBan, which is a domain in ALSB, contains a project named Proxy, which contains a folder named Accounting, which in turn contains a proxy service called PayoutProxy. When PayoutProxy is published to UDDI, its businessService is created with the following key:

`uddi:bea.com:servicebus:AnonESB:Proxies:Accounting:PayoutProxy`.

ALSB detailed proxy service information maps into the binding template as follows:

- The Endpoint URI maps to the access point.
- The Marker tModel for each transport maps to `tModelInstanceDetails`.
  - Transport `tModels` for HTTP, JMS, File, FTP, E-mail. New `tModels` are packaged with ALSB to support JMS and file transports.
  - Detailed ALSB configuration information maps to instanceParms.
- The Market tModel for each service type maps to the tModelInstanceDetails. This includes the following:
  - Protocol `tModels` for WSDL, any SOAP, any XML, and Messaging. New `tModels` are packaged with ALSB to support anySOAP, anyXML, and Messaging.

- WSDL maps via WSDL to UDDI technology note.
- Messaging has detailed configuration information that maps to `InstanceParms`.

Listing 7-2 shows a detailed information mapping to the binding template.

Listing 7-2 Sample Detailed Mapping to the Binding Template

```
<bindingTemplate bindingKey="uddi:" serviceKey="uddi:">
  <accessPoint useType="endPoint">file:///c:/temp/in3</accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo tModelKey="uddi:uddi.org:transport:file">
      <InstanceDetails>
      <InstanceParms><ALSBInstanceParms xmlns="http://www.bea.com/wli/sb/uddi">
        <property name="fileMask" value="*.*"/>
        <property name="sortByArrival" value="false"/> </ALSBInstanceParms>
      </InstanceParms>
      </InstanceDetails>
    </tModelInstanceInfo>
    <tModelInstanceInfo tModelKey="uddi:bea.com:servicebus:protocol:
        messagingservice">
      <InstanceDetails>
      <InstanceParms><ALSBInstanceParms xmlns="http://www.bea.com/wli/sb/uddi">
        <property name="requestType" value="XML"/>
        <property name="RequestSchema" value="http://domain.com:7001
          /sbresource?SCHEMA%2FDJS%2FOAGProcessPO"/>
        <property name="RequestSchemaElement"
              value="PROCESS_PO"/>
        <property name="responseType" value="None"/></ALSBInstanceParms>
    </InstanceParms>
```

```
    </InstanceDetails>
  </tModelInstanceInfo>
</tModelInstanceDetails>
</bindingTemplate>
```

Transport Attributes

Each of the transport types in the `uddi:uddi.org:transport: *` group has a different set of detailed metadata. See Table 7-3. This metadata provides the configuration details of the transport for the proxy service. It is neither useful for characterizing the service nor useful in querying the service. However, after the service has been discovered, this data is needed to access the service. The metadata is represented by an XML string and is located in the `instanceParms` field in `tModelInstanceInfo`.

If you are mapping a proxy service that uses the HTTP transport, and as part of the HTTP configuration you need to describe some configuration details, including the required client authorization and the request and response character encoding. Listing 7-3 provides an example of what must appear in the bindingTemplate `tModelInstanceDetails`.

Listing 7-3 Example of tModelInstanceDetails

```
<tModelInstanceDetails>
  <tModelInstanceInfo tModelKey="uddi:uddi.org:transport:http">
    <instanceDetails>
      <instanceParms>
        <ALSBInstanceParms xmlns="http://www.bea.com/wli/sb/uddi">
          <property name="client-auth" value="basic"/>
          <property name="request-encoding" value="iso-8859-1"/>
          <property name="response-encoding" value="utf-8"/>
          <property name="Scheme" value="http"/>
        </ALSBInstanceParms>
      </instanceParms>
    </instanceDetails>
  </tModelInstanceInfo>
</tModelInstanceDetails>
```

Note: For each transport, the service endpoint is always stored in the bindingTemplate `accessPoint` field.

The `client-auth` property is present in the `instanceParms` of the HTTP or HTTPS transport attributes whenever authentication is configured. The possible values for `client-auth` are `basic`, `client-cert`, and `custom-token`. Whenever the value is `custom-token`, two additional properties are present: `token-header` and `token-type`.

Because ALSB business service definitions do not support custom token authentication in this release, if you import a service from UDDI that has a value of custom-token for `client-auth`, the service is imported as if it does not have any authentication configuration.

Table 7-4 is organized by transport type and lists the `tModelKey` and `instanceParms` used by each of the transports.

Table 7-4 Transport Attributes

Transport	tModelKey	InstanceParms
E-mail[1]	`uddi:uddi.org:transport:smtp`	• Attachment Supported [Boolean] • Request Encoding
File	`uddi:uddi.org:transport:file`	• File Mask • Sort by Arrival [Boolean] • Request Encoding
FTP	`uddi:uddi.org:transport:ftp`	• File Mask • Sort by Arrival [Boolean] • Transfer Mode [Text, Binary] • Request Encoding
HTTP	`uddi:uddi.org:transport:http`	• Client Authentication [None, Basic, Client Cert (HTTP only), and Custom Token] • Request Encoding • Response Encoding
JMS	`uddi:uddi.org:transport:jms`	• Destination Type [Queue, Topic] • Response Required, Response URI • Response Message Type [Bytes, Text] • Request Encoding • Response Encoding

Table 7-4 Transport Attributes (Continued)

Transport	tModelKey	InstanceParms
Local	`uddi:uddi.org:transport:local`	• None
MQ	`uddi:bea.org:transport:mq`	• Response Required • Response URI • Response Correlation Pattern
SB	`uddi:bea.org:transport:sb` The URI scheme is `sb` when `use ssl` is false; `sbs` when `use ssl` is true.	• None
SFTP	`uddi:bea.org:transport:sftp`	• File Mask • Sort by Arrival [Boolean] • Request Encoding • Authentication Mode
Tuxedo	`uddi:bea.org:transport:tuxedo`	• Response Required • Access Point ID • Buffer Type • Buffer Subtype • Classes Jar • Field Table Classes • View Classes
WS	`uddi:uddi.org:transport:http` WS uses the HTTP tModelKey	• None

1. The accessPoint in the Binding Template for an E-mail transport uses the standard mailto URL format: `mailto:name@some_server.com`
This is different from the one configured for the proxy service in ALSB, which is a URL oriented toward reading e-mail. It is not be possible to derive this mailto URL from the proxy service definition as the server name is not known. For example, if the proxy service is defined to read from a POP3 server, it might be defined with a URL such as `mailfrom:pop3.bea.com`. When publishing such a proxy service, a dummy server is added. In the above example, the published URL will take the form `mailto:some_name@some_server.com`.

Service Type Attributes

Table 7-5 provides a high-level description of each of the service types.

Table 7-5 Service Type Attributes

Service	Description
WSDL	WSDL based proxies map to UDDI based on the *Using WSDL in a UDDI Registry, version 2.0.2* technical note at URL: `http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v202-20040631.htm`.
Any SOAP	A simple marker protocol in the tModel in the bindingTemplate `tModelInstanceDetails`, as well as in the `categoryBag`, defines the Any Soap attributes.
Any XML	A simple marker protocol `tModel` within the bindingTemplate `tModelInstanceDetails`, as well as in the `categoryBag` defines the Any XML attributes. This is a new detailed `tModel`.
Messaging Services	A simple marker protocol tModel in the bindingTemplate `tModelInstanceDetails`, defines the messaging services attributes. This is a new detailed tModel. Unlike the other service types, messaging services have additional configuration information associated with them, which provides detail about the request and response messages. The configuration details are represented as XML data in the `InstanceParms` data for the following tModel reference in the `tModelInstanceInfo`: • Input message format (XML, Text, Binary, MFL) • URL of input message schema in ALSB (optional, if input message is XML) • URL of input message MFL in ALSB (if input message is MFL) • Output message format (none, XML, Text, Binary, MFL) • URL of output message schema in ALSB (optional, if output message is XML) • URL of output message MFL in ALSB (if output message is MFL)

Canonical tModels Supporting ALSB Services

The ALSB-UDDI mapping introduces a number of new canonical `tModels` that are used to represent ALSB metadata and relationships. These `tModels` must be registered in the UDDI

registry to support this mapping. You can create these `tModels` in AquaLogic Service Registry under the administrator ID.

Table 7-6 provides a summary of the new `tModels`.

Table 7-6 ALSB tModels

Name	Value	Description
CategorizationGroup tModel Types		
`bea-com:servicebus:properties`		Describes very specific attributes of an ALSB service. In the data model it is used in the business service `categoryBag`.
Categorization tModel Types		
`bea-com:servicebus:serviceType`	WSDL, SOAP, XML, Messaging Service	Describes the service type of the ALSB service.
`bea-com:servicebus:instance`	URL of ALSB Console	Describes the service instance in ALSB responsible for publishing the service to UDDI.
Transport tModel Types		
`uddi-org:jms`		Describes the type of transport used by the service. A reference to it is found in the accessPoint attribute of the business service binding template.
`uddi-org:file`		Describes the type of transport used to invoke the service. A reference to it is found in the accessPoint attribute of the business service binding template.
Protocol tModel Types		
`bea-com:servicebus:anySoap`		Describes the type of protocol used to access the service. It designates services that have a SOAP message but not defined by a WSDL or schema. The message body content is determined dynamically by the application.

Table 7-6 ALSB tModels

Name	Value	Description
`bea-com:servicebus:anyXML`		Describes the type of protocol used to access the service. It designates services having an XML message but not defined by a WSDL or schema. The message body content is determined dynamically by the application.
`bea-com:servicebus:messaging Service`		Describes the type of protocol used to access the service. It designates services where the request message can be any XML (with or without schema), text, binary, or MFL and whose response messge can be any of the above or none. The message body content is determined dynamically by the application.

Example

Listing 7-4 is an example of the mapping for a Messaging Service, configured with JMS transport, the request being XML with a schema and the response being a text message.

Listing 7-4 Sample Messaging Service Mapping

```
<businessService
  serviceKey="uddi:bea.com:servicebus:Domain:Project:JMSMessaging"
  businessKey="uddi:9cb77770-57fe-11da-9fac-6cc880409fac"
  xmlns="urn:uddi-org:api_v3">
    <name>JMSMessagingProxy</name>
    <bindingTemplates>
      <bindingTemplate
        bindingKey="uddi:4c401620-5ac0-11da-9faf-6cc880409fac"
        serviceKey="uddi:bea.com:servicebus:
          Domain:Project:JMSMessaging">
      <accessPoint useType="endPoint">
        jms://server.com:7001/weblogic.jms.XAConnectionFactory/
              ReqQueue
      </accessPoint>
      <tModelInstanceDetails>
        <tModelInstanceInfo tModelKey="uddi:uddi.org:transport:jms">
          <instanceDetails>
            <instanceParms>
```

```
            <ALSBInstanceParms
              xmlns="http://www.bea.com/wli/sb/uddi">
              <property name="is-queue" value="true"/>
              <property name="request-encoding"
                value="iso-8859-1"/>
              <property name="response-encoding"
                value="utf-8"/>
              <property name="response-required"
                value="true"/>
              <property name="response-URI"
                value="jms://server.com:7001/
                .jms.XAConnectionFactory/
                  RespQueue"/>
              <property name="response-message-type"
                value="Text"/>
              <property name="Scheme" value="jms"/>
            </ALSBInstanceParms>
          </instanceParms>
        </instanceDetails>
      </tModelInstanceInfo>
      <tModelInstanceInfo
        tModelKey="uddi:bea.com:servicebus:
            protocol:messagingservice">
        <instanceDetails>
          <instanceParms>
            <ALSBInstanceParms xmlns=
              "http://www.bea.com/wli/sb/uddi">
                <property name="requestType" value="XML"/>
                <property name="RequestSchema"
                  value="http://server.com:7001/
                  sbresource?SCHEMA%2FDJS%2FOAGProcessPO"/>
              <property name="RequestSchemaElement"
                value="PROCESS_PO_007"/>
              <property name="responseType" value="Text"/>
              </ALSBInstanceParms>
            </instanceParms>
          </instanceDetails>
        </tModelInstanceInfo>
      </tModelInstanceDetails>
    </bindingTemplate>
  </bindingTemplates>
<categoryBag>
<keyedReferenceGroup tModelKey="uddi:bea.com:servicebus:properties">
  <keyedReference tModelKey="uddi:bea.com:servicebus:servicetype"
        keyName="Service Type"
        keyValue="Mixed" />
  <keyedReference tModelKey="uddi:bea.com:servicebus:instance"
        keyName="Service Bus Instance"
        keyValue="http://cyberfish.bea.com:7001" />
```

```
    </keyedReferenceGroup>
  </categoryBag>
</businessService>
```

CHAPTER 8

Extensibility Using Java Callouts and POJOs

To allow you to extend the capabilities of ALSB, you can invoke custom Java code from within proxy services. ALSB supports a Java exit mechanism via a Java callout action that allows you to call out to a Plain Old Java Object (POJO). Static methods can be accessed from any POJO. The POJO and its parameters are visible in the ALSB Console or ALSB Plug-in for WorkSpace Studio at design time; the parameters can be mapped to message context variables.

You can also use Java callouts to create Java objects to store in the pipeline and to pass Java objects as parameters to other Java callouts.

For information about configuring a Java callout to a POJO, see:

- "Java Callout" in Proxy Services: Actions in *Using the AquaLogic Service Bus Console*
- Adding and Configuring Java Callout Actions in *Using the AquaLogic Service Bus Plug-in for WorkSpace Studio*

Usage Guidelines

The scenarios in which you can use Java callouts in ALSB include the following:

- Custom validation—Examples of custom validation include validation against a DTD, or doing cross-field semantic validation in Java.
- Custom transformation—Examples of custom transformations can include converting a binary document to base64Binary, or vice versa, or using a custom Java transformation class.

- Custom authentication and authorization—Examples of custom authentication and authorizations include scenarios in which a custom token in a message needs to be authenticated and authorized. However, the authenticated user's identity cannot be propagated by ALSB to the services or POJOs subsequently invoked by the proxy service.
- Lookups for message enrichment—For example, a file or Java table can be used to look up any piece of data that can enrich a message.
- Binary data access—You can use a Java callout to a POJO to sniff the first few bytes of a binary document to deduce the MFL type. The MFL type returned is used for a subsequent NonXML-to-XML transformation using the MFL Transform action.
- Implementing custom routing rules or rules engines.
- Create a Java object and store it in the pipeline.
- Pass a Java object as a parameter to another Java callout.

The input and return types for Java callouts are not restricted. However, any return types other than primitives, `Strings`, or `XmlObjects` can only be passed (unmodified) to other Java callouts. See "Java Content in the body Variable" on page 5-7 for more information about storing and passing Java objects in the pipeline.

Enterprise JavaBeans (EJBs) also provide a Java exit mechanism. The use of EJBs is recommended over the use of POJOs in the following cases:

- When you already have an EJB implementation.
- When you require read access to a JDBC database—Although POJOs can be used for this purpose, EJBs were specifically designed for this and provide better support for management of, and connectivity to, JDBC resources.
- When you require write access to a JDBC database or other J2EE transactional resource—EJBs were specifically designed for transactional business logic and they provide better support for proper handling of failures. However, transaction and security context propagation is supported with POJOs and they can be used for this purpose.

For outbound messaging, BEA recommends that you write a custom transport instead of using POJOs or EJBs.

Best Practices

POJOs are registered as JAR resources in ALSB. For information about JAR resources, see JARs in *Using the AquaLogic Service Bus Console.*

In general, BEA recommends that the JARs are small and simple—any large bodies of code that a JAR invokes or large frameworks that are made use of are best included in the system classpath. Note that if you make a change to the system classpath, you must reboot the server.

BEA recommends that you put dependent and overlapping classes in the same JAR resource; put them in different JARS if they are naturally distinct. Any change to a JAR causes all the services that reference it to be redeployed—this can be time consuming for your ALSB system. The same class can be located in multiple JAR resources without causing conflicts. The JARs are dynamically class loaded when they are first referenced.

A single POJO can be invoked by one or more proxy services. All the threads in the proxy services invoke the same POJO. Therefore, the POJO must be thread safe. A class or method on a POJO can be synchronized, in which case it serializes access across all threads in all of the invoking proxy services. Any finer-grained concurrency (for example, to control access to a DB read results cache and implement stale cache entry handling) must be implemented by the POJO code.

It is generally a bad practice for POJOs to create threads.

CHAPTER 9

XQuery Implementation

ALSB uses the BEA AquaLogic Data Services Platform implementation of the XQuery engine which fully supports all of the language features that are described in the World Wide Web (W3C) specification for XQuery with one exception: **modules**. For more information about the XQuery 1.0 and XPath 2.0 functions and operators (W3C Working Draft 23 July 2004), see the following URL:

`http://www.w3.org/TR/2004/WD-xpath-functions-20040723/`

ALSB supports the following XQuery functions:

- A robust subset of the XQuery functions that are described in W3C specification. For a list of the supported functions and a description of each function, see BEA XQuery Implementation in the *XQuery Developer's Guide*.
- The function extensions and language keywords that BEA AquaLogic Data Services Platform provides—with a small number of exceptions. For information about those exceptions, see "Supported Function Extensions from AquaLogic Data Services Platform" on page 9-2.
- ALSB-specific function extensions. See "Function Extensions from ALSB" on page 9-2.

Note: All of the BEA function extensions use the following function prefix `fn-bea:` In other words, the full XQuery notation for an extended function is of this format: `fn-bea: function_name`.

Supported Function Extensions from AquaLogic Data Services Platform

ALSB supports all function extensions that BEA AquaLogic Data Services Platform provides except for the following:

- `fn-bea:is-access-allowed`
- `fn-bea:is-user-in-group`
- `fn-bea:is-user-in-role`
- `fn-bea:userid`
- `fn-bea:async`
- `fn-bea:timeout`
- `fn-bea:get-property`
- `fn-bea:execute-sql()`

BEA recommends that you do not use the following functions in ALSB—they are better covered by other language features:

- `fn-bea:if-then-else`
- `fn-bea:QName-from-string`
- `fn-bea:sql-like`

For a list of all AquaLogic Data Services Platform function extensions and a description of each function, see BEA XQuery Implementation in the *XQuery Developer's Guide*.

Function Extensions from ALSB

ALSB provides the following XQuery functions:

- fn-bea:lookupBasicCredentials
- fn-bea: uuid()
- fn-bea:execute-sql()
- fn-bea:serialize()

fn-bea:lookupBasicCredentials

The `fn-bea:lookupBasicCredentials` function returns the user name and unencrypted password from a specified service account. You can specify any type of service account (static, pass-through, or user-mapping). See Service Account in *Using the AquaLogic Service Bus Console*.

Use the `fn-bea:lookupBasicCredentials` function as part of a larger set of XQuery functions that you use to encode a user name and password in a custom transport header or in an application-specific location within the SOAP envelope. You do not need to use this function if you only need user names and passwords to be located in HTTP Authentication headers or as WS-Security user name tokens. ALSB already retrieves user names and passwords from service accounts and encodes them in HTTP Authentication headers or as WS-Security user name tokens when required.

The function has the following signature:

```
fn-bea:lookupBasicCredentials( $service-account as xs:string ) as
UsernamePasswordCredential
```

where `$service-account` is the path and name of a service account in the following form:

`project-name``[/`*`folder`*`[...]]/`*`service-account-name`*

The return value is an XML element of this form:

```
<UsernamePasswordCredential
   xmlns="http://www.bea.com/wli/sb/services/security/config">
   <username>name</username>
   <password>unencrypted-password</password>
</UsernamePasswordCredential>
```

You can store the returned element in a user-defined variable and retrieve the user name and password values from this variable when you need them.

For example, your ALSB project is named `myProject`. You create a static service account named `myServiceAccount` in a folder named `myFolder1/myFolder2`. In the service account, you save the user name of `pat` with a password of `patspassword`.

To get the user name and password from your service account, invoke the following function:

```
fn-bea:lookupBasicCredentials(
myProject/myFolder1/myFolder2/myServiceAccount )
```

The function returns the following element:

```
<UsernamePasswordCredential
   xmlns="http://www.bea.com/wli/sb/services/security/config">
   <username>pat</username>
   <password>patspassword</password>
</UsernamePasswordCredential>
```

fn-bea: uuid()

The function `fn-bea:uuid()` returns a universally unique identifier. The function has the following signature:

```
fn-bea:uuid() as xs:string
```

You can use this function in the proxy pipeline to generate a unique identifier. You can insert the generated unique identifier into an XML document as an element. You cannot generate a unique identifier to the system variable. You can use this to modify a message payload.

For example, suppose you want to generate a unique identifier to add to a message for tracking purposes. You could use this function to generate a unique identifier. The function returns a string that you can add it to the SOAP header.

fn-bea:execute-sql()

The `fn-bea:execute-sql()` function provides low-level database access from XQuery within ALSB message flows--see "Accessing Databases Using XQuery" on page 3-39. The query returns a sequence of flat row elements with typed data.

The function has the following signature:

```
fn-bea:execute-sql( $datasource as xs:string, $rowElemName as xs:QName,
$sql as xs:string, $param1, ..., $paramk) as element()*
```

where

- `$datasource` is the JNDI name of the datasource
- `$rowElemName` is the name of the row element—specify `$rowElemName` as whatever QName you want each element of the resulting element sequence to have
- `$sql` is the SQL statement
- `$param1, ..., $paramk` are 1 to k parameters
- `element()*` represents the sequence of elements returned

The return value is a sequence of flat row elements with typed data and automatically translates values between SQL/JDBC and XQuery data models. Data Type mappings that the XQuery engine generates or supports for the supported databases can be found in the "XQuery-SQL Mapping Reference" on page A-1.

When you execute the `fn-bea:execute-sql()` function from an ALSB message flow, you can store the returned element in a user-defined variable.

Use the following examples to understand the use of the `fn-bea:execute sql()` function in ALSB:

Example 1: Retrieving the URI from a Database for Dynamic Routing

ALSB proxy services support specification of the URI to which messages are to be routed at run time (dynamically)—see "Using Dynamic Routing" on page 3-35. Listing 9-1 is an example use of the `fn-bea:execute-sql()` function to retrieve the URI from a database in a dynamic routing scenario.

Listing 9-1 Get the URI for a Business Service from a Database

```
<ctx:route><ctx:service>
{
    fn-bea:execute-sql(
    'ds.myJDBCDataSource',
    xs:QName('customer'),
    'SELECT targetService FROM DISPATCH_MAPPING WHERE customer_priority=?',
      xs:string($body/m:Request/m:customer_pri/text())
    )/TARGETSERVICE/text()
}
</ctx:service></ctx:route>
```

In Listing 9-1:

- `ds.myJDBCDataSource` is the JNDI name to the data source
- `xs:string($body/m:Request/m:customer_pri/text())` interrogates the request message and populates `customer_priority=?` with the value of `customer_pri` in the message
- `/TARGETSERVICE/text()`is the path applied to the result of the SQL statement, which results in the string (CDATA) contents of that element being returned
- `<ctx:route><ctx:service> ... </ctx:service></ctx:route>` are required elements of the XQuery statement for a dynamic routing scenario
- The following is the table definition for `DISPATCH_MAPPING`:

```
create table DISPATCH_MAPPING
(
  customer_priority varchar2(256),
  targetService varchar2(256),
  soapPayload varchar2(1024)
);
```

The `DISPATCH_MAPPING` table is populated as shown in Listing 9-2:

Listing 9-2 DISPATCH_MAPPING Table

```
 INSERT INTO DISPATCH_MAPPING (customer_priority, targetService,
soapPayload)
 VALUES ('0001', 'system/UCGetURI4DynamicRouting_proxy1', '<something/>');
 INSERT INTO DISPATCH_MAPPING (customer_priority, targetService,
soapPayload)
 VALUES ('0002', 'system/UCGetURI4DynamicRouting_proxy2', '<something/>');
```

Note: The third column in the table (`soapPayload`) is not used in this scenario.

Executing the fn-bea:execute-sql for Example 3

If the XQuery in Listing 9-1 is executed as a result of a proxy service receiving the request message in Listing 9-3 (note that the value of `<customer_pri>` in the request message is `0001`), the URI returned for the dynamic route scenario is

`system/UCGetURI4DynamicRouting_proxy1`

(See also Listing 9-2.)

Listing 9-3 Example Request Message $body

```
<m:Request xmlns:m="http://www.bea.com/alsb/example">
<m:customer_pri>0001</m:customer_pri>
</m:Request>
```

Example 2: Getting XMLType Data from a Database

Data Type mappings that the XQuery engine generates or supports for the supported databases can be found in the "XQuery-SQL Mapping Reference" on page A-1. Note that the `XMLType` column type in SQL is not supported. However, you can access the data in an `XMLType` column by using the `getStringVal()` method of the `XMLType` object to convert it to a String value.

The following scenario outlines a procedure you can use to select data from an `XMLType` column in an Oracle database.

1. Use an assign action in a proxy service message flow to assign the results of the following XQuery to a variable (`$result`).

Listing 9-4 Get XMLType Data from a Database

```
fn-bea:execute-sql(
    'ds.myJDBCDataSource',
    'Rec',
    'SELECT a.purchase_order.getStringVal() purchase_order from datatypes
a'
```

```
)
```

where:

- `ds.myJDBCDataSource` is the JNDI name to the data source
- `Rec` is the `$rowElemName`—therefore, Rec is the QName given to each element of the resulting element sequence
- `select a.purchase_order.getStringVal() ...` is the SQL statement that uses the `getStringVal()` method of the `XMLType` object to convert it to a String value
- `datatypes` is the table from which the value of the XML is read (the `datatypes` table in this case contains one row)

  Note: The following is the table definition for the `dataty.pes` table:

```
create table datatypes
(
  purchase_order xmltype
);
```

2. Use a replace action to replace the node contents of `$body` with the results of the `fn-bea:execute-sql()` query (assigned to `$result` in the preceding step):

```
Replace [ node contents ] of [ undefined XPath ] in [ body ] with
[ $result/purchase_order/text() ]
```

   The following listing shows `$body` after the replacement.

Note: The `datatypes` table contains one row (with the purchase order data); the row contains the XML represented in Listing 9-5.

Listing 9-5 $body After XML Content is Replaced with Result of fn-bea:execute-sql()

```
<soap-env:Body>
  <openuri:orders xmlns:openuri="http://openuri.com/">
    <openuri:order>
      <openuri:customerID>123</openuri:customerID>
      <openuri:orderID>123A</openuri:orderID>
```

```
    </openuri:order>
    <openuri:order>
      <openuri:customerID>345</openuri:customerID>
      <openuri:orderID>345B</openuri:orderID>
    </openuri:order>
    <openuri:order>
      <openuri:customerID>789</openuri:customerID>
      <openuri:orderID>789C</openuri:orderID>
    </openuri:order>
  </openuri:orders>
</soap-env:Body>
```

---

fn-bea:serialize()

You can use the `fn-bea:serialize()` function if you need to represent an XML document as a string instead of as an XML element. For example, you may want to exchange an XML document through an EJB interface and the EJB method takes String as argument. The function has the following signature:

```
fn-bea:serialize($input as item()) as xs:string
```

APPENDIX A

XQuery-SQL Mapping Reference

This appendix provides information about the native RDBMS Data Type support and XQuery mappings that the BEA XQuery engine generates or supports. It includes the following topics:

- Core RDBMS Data Type Mapping:
  - IBM DB2/NT 8
  - Microsoft SQL Server
  - Oracle 8.1.x
  - Oracle 9.x, 10.x
  - Pointbase 4.4 (and higher)
  - Sybase 12.5.2 (and higher)
- Base (Generic) RDBMS Data Type Mapping

For information about using these mappings in ALSB XQueries, see "Accessing Databases Using XQuery" on page 3-39.

For complete information about database and JDBC drivers support in ALSB, see Supported Database Configurations in *Supported Configurations for AquaLogic Service Bus*.

IBM DB2/NT 8

This section identifies the data type mappings that the XQuery engine generates or supports for IBM DB2/NT 8.

Table A-1 IBM DB2 Data Type Mappings

DB2 Data Type	XQuery Type
BIGINT	xs:long
BLOB	xs:hexBinary
CHAR	xs:string
CHAR() FOR BIT DATA	xs:hexBinary
CLOB[1]	xs:string
DATE	xs:date
DOUBLE	xs:double
DECIMAL(p,s)[2] (NUMERIC)	xs:decimal (if s > 0), xs:integer (if s = 0)
INTEGER	xs:int
LONG VARCHAR[1]	xs:string
LONG VARCHAR FOR BIT DATA	xs:hexBinary
REAL	xs:float
SMALLINT	xs:short
TIME[3]	xs:time[4]
TIMESTAMP[5]	xs:dateTime[4]
VARCHAR	xs:string[4]
VARCHAR() FOR BIT DATA	xs:hexBinary

1. Pushed down in project list only.
2. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).

3. Accurate to 1 second.
4. Values converted to local time zone (timezone information removed) due to TIME and TIMESTAMP limitations. See XQuery-SQL Data Type Mappings in XQuery Engine and SQL in the *XQuery Developer's Guide* for more information.
5. Precision limited to milliseconds.

Microsoft SQL Server

This section identifies the data type mappings that the XQuery engine generates or supports for Microsoft SQL Server.

Table A-2 SQL Server 2000 Data Type Mapping

SQL Data Type	XQuery Type
BIGINT	xs:long
BINARY	xs:hexBinary
BIT	xs:boolean
CHAR	xs:string
DATETIME[1]	xs:dateTime[2]
DECIMAL(p,s)[3] (NUMERIC)	xs:decimal (if s > 0), xs:integer (if s = 0)
FLOAT	xs:double
IMAGE	xs:hexBinary
INTEGER	xs:int
MONEY	xs:decimal
NCHAR	xs:string
NTEXT[4]	xs:string
NVARCHAR	xs:string
REAL	xs:float

Table A-2 SQL Server 2000 Data Type Mapping

SMALLDATETIME[5]	xs:dateTime
SMALLINT	xs:short
SMALLMONEY	xs:decimal
SQL_VARIANT	xs:string
TEXT[4]	xs:string
TIMESTAMP	xs:hexBinary
TINYINT	xs:short
VARBINARY	xs:hexBinary
VARCHAR	xs:string
UNIQUIDENTIFIER	xs:string

1. Fractional-second-precision up to 3 digits (milliseconds). No timezone.
2. Values converted to local time zone (timezone information removed) and fractional seconds truncated to milliseconds due to DATETIME limitations. See XQuery-SQL Data Type Mappings in XQuery Engine and SQL in the *XQuery Developer's Guide* for more information.
3. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).
4. Pushed down in project list only.
5. Accuracy of 1 minute.

Oracle 8.1.x

This section identifies the data types that the XQuery engine generates or supports for Oracle 8.1.x (Oracle 8i).

Table A-3 Oracle 8.1.x Data Type Mapping

Oracle 8 Data Type	XQuery Type
BFILE	not supported
BLOB	xs:hexBinary

Table A-3 Oracle 8.1.x Data Type Mapping

CHAR	xs:string
CLOB[1]	xs:string
DATE[2]	xs:dateTime
FLOAT	xs:double
LONG[1]	xs:string
LONG RAW	xs:hexBinary
NCHAR	xs:string
NCLOB[1]	xs:string
NUMBER	xs:double
NUMBER(p,s)[3]	xs:decimal (if s > 0), xs:integer (if s <=0)
NVARCHAR2	xs:string
RAW	xs:hexBinary
ROWID	xs:string
UROWID	xs:string

1. Pushed down in project list only.
2. Does not support fractional seconds.
3. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).

Oracle 9.x, 10.x

This section identifies the data type and other mappings that the XQuery engine generates or supports for Oracle 9.x (Oracle 9*i*) and Oracle 10.x (Oracle 10*g*). Note that Oracle treats empty strings as NULLs, which deviates from XQuery semantics and may lead to unexpected results for expressions that are pushed down.

Table A-4 Oracle 9.x, 10.x Data Type Mapping

Oracle 9 Data Type	XQuery Type
BFILE	not supported
BLOB	xs:hexBinary
CHAR	xs:string
CLOB[1]	xs:string
DATE	xs:dateTime[2]
FLOAT	xs:double
INTERVAL DAY TO SECOND	xdt:dayTimeDuration
INTERVAL YEAR TO MONTH	xdt:yearMonthDuration
LONG[1]	xs:string
LONG RAW	xs:hexBinary
NCHAR	xs:string
NCLOB[1]	xs:string
NUMBER	xs:double
NUMBER(p,s)	xs:decimal (if s > 0), xs:integer (if s <=0)
NVARCHAR2	xs:string
RAW	xs:hexBinary
ROWID	xs:string

Table A-4 Oracle 9.x, 10.x Data Type Mapping

TIMESTAMP	xs:dateTime[3]
TIMESTAMP WITH LOCAL TIMEZONE	xs:dateTime
TIMESTAMP WITH TIMEZONE	xs:dateTime
VARCHAR2	xs:string
UROWID	xs:string

1. Pushed down in project list only.
2. When SDO stores xs:dateTime value in Oracle DATE type, it is converted to local time zone and fractional seconds are truncated due to DATE limitations. See XQuery-SQL Data Type Mappings in XQuery Engine and SQL in the *XQuery Developer's Guide* for more information.
3. XQuery engine maps XQuery xs:dateTime to either TIMESTAMP or TIMESTAMP WITH TIMEZONE data type, depending on presence of timezone information. Storing xs:dateTime using SDO may result in loss of precision for fractional seconds, depending on the SQL type definition.

Sybase 12.5.2 (and higher)

This section identifies the data types that the XQuery engine generates or supports for Sybase 12.5.2 (and higher).

Note: Sybase deviates from XQuery semantics (which ignores empty strings) and treats empty strings as a single-space string.

Table A-5 Sybase 12.5.2 Data Type Mapping

Sybase Data Type	XQuery Type
BINARY	xs:hexBinary
BIT	xs:boolean
CHAR	xs:string
DATE	xs:date

Table A-5 Sybase 12.5.2 Data Type Mapping

DATETIME[1]	xs:dateTime[2]
DECIMAL(p,s)[3] (NUMERIC)	xs:decimal (if s > 0), xs:integer (if s == 0)
DOUBLE PRECISION	xs:double
FLOAT	xs:double
IMAGE	xs:hexBinary
INT (INTEGER)	xs:int
MONEY	xs:decimal
NCHAR	xs:string
NVARCHAR	xs:string
REAL	xs:float
SMALLDATETIME[4]	xs:dateTime
SMALLINT	xs:short
SMALLMONEY	xs:decimal
SYSNAME	xs:string
TEXT[5]	xs:string
TIME	xs:time
TINYINT	xs:short
VARBINARY	xs:hexBinary
VARCHAR	xs:string

1. Supports fractional seconds up to 3 digits (milliseconds) precision; no timezone information.
2. When SDO stores xs:dateTime value in Oracle DATE type, it is converted to local time zone and fractional seconds are truncated due to DATE limitations. See XQuery-SQL Data Type Mappings in XQuery Engine and SQL in the *XQuery Developer's Guide* for more information.

3. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).
4. Accurate to 1 minute.
5. Expressions returning text are pushed down in the project list only.

Pointbase 4.4 (and higher)

This section identifies the data types that the XQuery engine generates or supports for Pointbase.

Table A-6 Pointbase 4.4 Data Type Mapping

Pointbase Data Type	XQuery Type
BIGINT	xs:long
BLOB	xs:hexBinary
BOOLEAN	xs:boolean
CHAR (CHARACTER)	xs:string
CLOB	xs:string
DATE	xs:date
DECIMAL(p,s)[1] (NUMERIC)	xs:decimal (if s > 0), xs:integer (if s == 0)
DOUBLE PRECISION	xs:double
FLOAT	xs:double
INTEGER (INT)	xs:int
SMALLINT	xs:short
REAL	xs:float
TIME	xs:time
TIMESTAMP	xs:dateTime
VARCHAR	xs:string

1. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).

Base (Generic) RDBMS Data Type Mapping

When mapping SQL to XQuery data types, the XQuery engine first checks the JDBC typecode. If the typecode has a corresponding XQuery type, the XQuery engine uses the matching native type name. If no matching typecode or type name is available, the column is ignored. Table A-7 shows this mapping.

Table A-7 Base Platform Data Type Mapping (JDBC<–>XQuery Equivalents)

JDBC Data Type	Typecode	XQuery Data Type
BIGINT	-5	xs:long
BINARY	-2	xs:string
BIT	-7	xs:boolean
BLOB	2004	xs:hexBinary
BOOLEAN	16	xs:boolean
CHAR	1	xs:string
CLOB[1]	2005	xs:string
DATE	91	xs:date[2]
DECIMAL (p,s)[3]	3	xs:decimal (if s > 0), xs:integer (if s =0)
DOUBLE	8	xs:double
FLOAT	6	xs:double
INTEGER	4	xs:int
LONGVARBINARY	-4	xs:hexBinary
LONGVARCHAR[1]	-1	xs:string
NUMERIC (p,s)[3]	2	xs:decimal (if s > 0), xs:integer (if s =0)
REAL	7	xs:float
SMALLINT	5	xs:short

Table A-7 Base Platform Data Type Mapping (JDBC<–>XQuery Equivalents)

JDBC Data Type	Typecode	XQuery Data Type
TIME[4]	92	xs:time[4]
TIMESTAMP[4]	93	xs:dateTime[2]
TINYINT	-6	xs:short
VARBINARY	-3	xs:hexBinary
VARCHAR	12	xs:string
OTHER	1111	ALSB uses native data type name to map to an appropriate XQuery data type.
Other vendor-specific JDBC type codes		

1. Pushed down in project list only.
2. Values converted to local time zone (timezone information removed) due to DATE limitations. See XQuery-SQL Data Type Mappings in XQuery Engine and SQL in the *XQuery Developer's Guide* for more information.
3. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).
4. Precision of underlying RDBMS determines the precision of TIME data type and how much truncation, if any, will occur in translating xs:time to TIME.

Related Topics

"Accessing Databases Using XQuery" on page 3-39

"fn-bea:execute-sql()" on page 9-4

APPENDIX B

Debugging ALSB

This section provides information about enabling debugging for different modules in ALSB. You can enable and disable debugging by modifying the corresponding entries in the following debug XML files, which are located in the root directory of your ALSB domain:

- `alsbdebug.xml`—Contains ALSB related debug flags
- `configfwkdebug.xml`—Contains configuration related debug flags

If the XML files are not in the root directory or if they have been deleted, they are created again without any contents when the server starts. Listing B-1 provides an example of the contents of the `alsbdebug.xml` file with debugging disabled for all modules (all entries set to `false`).

Listing B-1 alsbdebug.xml File

```
<java:sb-debug-logger xmlns:java='java:com.bea.wli.debug'>

<java:alsb-stages-transform-runtime-debug>false</java:alsb-stages-transform-ru
ntime-debug>
  <java:alsb-alert-manager-debug>false</java:alsb-alert-manager-debug>
  <java:alsb-credential-debug>false</java:alsb-credential-debug>

<java:alsb-jms-reporting-provider-debug>false</java:alsb-jms-reporting-provide
r-debug>

<java:alsb-management-credential-debug>false</java:alsb-management-credential-
debug>
```

```
<java:alsb-management-dashboard-debug>false</java:alsb-management-dashboard-de
bug>
  <java:alsb-management-debug>false</java:alsb-management-debug>

<java:alsb-management-user-mgt-debug>false</java:alsb-management-user-mgt-debu
g>
  <java:alsb-module-debug>true</java:alsb-module-debug>

<java:alsb-monitoring-aggregator-debug>false</java:alsb-monitoring-aggregator-
debug>
  <java:alsb-monitoring-debug>false</java:alsb-monitoring-debug>
  <java:alsb-pipeline-debug>true</java:alsb-pipeline-debug>
  <java:alsb-security-wss-debug>true</java:alsb-security-wss-debug>

<java:alsb-service-account-manager-debug>false</java:alsb-service-account-mana
ger-debug>

<java:alsb-service-provider-manager-debug>false</java:alsb-service-provider-ma
nager-debug>

<java:alsb-service-repository-debug>false</java:alsb-service-repository-debug>

<java:alsb-service-security-manager-debug>false</java:alsb-service-security-ma
nager-debug>

<java:alsb-service-validation-debug>false</java:alsb-service-validation-debug
  <java:alsb-test-console-debug>false</java:alsb-test-console-debug>
  <java:alsb-transports-debug>true</java:alsb-transports-debug>
  <java:alsb-uddi-debug>true</java:alsb-uddi-debug>
  <java:alsb-wsdl-repository-debug>false</java:alsb-wsdl-repository-debug>

<java:alsb-wspolicy-repository-debug>true</java:alsb-wspolicy-repository-debug
>

<java:alsb-security-encryption-debug>false</java:alsb-security-encryption-debu
g>
  <java:alsb-security-module-debug>false</java:alsb-security-module-debug>
  <java:alsb-sources-debug>false</java:alsb-sources-debug>
  <java:alsb-custom-resource-debug>false</java:alsb-custom-resource-debug>
  <java:alsb-mqconnection-debug>false</java:alsb-mqconnection-debug>
  <java:alsb-throttling-debug>false</java:alsb-throttling-debug>
</java:sb-debug-logger>
```

Listing B-2 provides an example of the contents of the `configfwkdebug.xml` file.

Listing B-2 configfwkdebug.xml File

```
<java:config-fwk-debug-logger xmlns:java='java:com.bea.wli.config.debug'>
    <n1:Name
xmlns:n1='java:weblogic.diagnostics.debug'>configfwkdebug</n1:Name>
    <java:config-fwk-debug>true</java:config-fwk-debug>
<java:config-fwk-transaction-debug>false</java:config-fwk-transaction-debu
g>
    <java:config-fwk-deployment-debug>true</java:config-fwk-deployment-deb
ug>
    <java:config-fwk-component-debug>false</java:config-fwk-component-debu
g>
    <java:config-fwk-security-debug>false</java:config-fwk-security-debug>
</java:config-fwk-debug-logger>
```

Although debugging should be disabled during normal ALSB operation, you may find it helpful to turn on certain debug flags while you are developing your solution and experimenting with it for the first time. For example, you may want to turn on the alert debugging flag when you are developing alerts and would like to investigate how the alert engine works.

Some of the available ALSB debug flags are shown in Table B-1.

Table B-1 ALSB Debug Flags

Debug Flag	Action
`alsb-stages-transform-runtime-debug`	Provides information on transformation related actions.
`alsb-alert-manager-debug`	Prints an evaluation of alerts.
`alsb-jms-reporting-provider-debug`	Provides information on the out of the box, JMS-based reporting provider.
`alsb-management-debug`	Provides information on user and group management in the console.
`alsb-monitoring-debug`	Provides information on the statistics system.

Table B-1 ALSB Debug Flags (Continued)

Debug Flag	Action
`alsb-pipeline-debug`	Provides information on errors that are generated within the pipeline.
`alsb-service-repository-debug`	Provides information on various service related configuration operations.
`alsb-service-security-manager-debug`	Provides information on access control.
`alsb-transports-debug`	Provides transport related debug information, including transport headers, which is printed per-message.
`alsb-wsdl-repository-debug`	Provides information on WSDL related configuration operation.
`alsb-wspolicy-repository-debug`	Provides information on WS policy.
`alsb-custom-resource-debug`	Provides information on custom resources.
`alsb-mqconnection-debug`	Provides information on the MQ connection resource.
`alsb-throttling-debug`	Provides information on the throttling feature.

Table B-2 lists the available configuration framework debug flags.

Table B-2 Configuration Framework Debug Flags

Debug Flag	Action
`config-fwk-debug`	Provides information on general aspects of ALSB configuration.
`config-fwk-transaction-debug`	Provides low level debug information about changes made to in-memory data structures and files. This debug flag also generates server startup recovery logs.
`config-fwk-deployment-debug`	Provides debug information on session creation, activation, and distribution of configuration in a cluster.

Table B-2 Configuration Framework Debug Flags (Continued)

Debug Flag	Action
`config-fwk-component-debug`	Provides low level debug information about create, update, delete, and import operations.
`config-fwk-security-debug`	Provides debug information on encryption and decryption during importing and exporting.

All other debug flags are self explanatory.

For all flags, debug information is logged to the server log at `{domaindir}/servers/{servername}/logs/{servername}.log`.

ALSB APIs

ALSB exposes APIs to allow customizing resources and to provide external access to monitoring data and deployment.

- Resource Update and Customization
- Management and Monitoring
- Deployment

Javadoc for the ALSB APIs is provided at the following URL:
`http://edocs.bea.com/alsb/docs30/javadoc`

Resource Update and Customization

A number of APIs are exposed to allow customization of service definitions, WSDLs, schemas, XQueries, and other design-time resources through programmatic interfaces. The supporting APIs allow loading ZIP files containing resources, in addition to moving, renaming, cloning, or deleting resources, folders, and projects. A typical use case is one in which you have a prototypical proxy service from which you make a number of copies—each copy can be modified programmatically.

Numerous customization options can be applied during deployment. For example, environment variables allow you to preserve or tailor settings when moving from one environment to another.

The available APIs include:

- ProxyServiceConfigurationMBean—Enable and disable SLA alerts and pipeline alerts for proxy services

- BusinessServiceConfigurationMBean
  - Enable and disable SLA alerts
  - Synchronize business services imported from UDDI registries
  - Detach business services from a UDDI registry
- ALSBConfigurationMBean Interface—APIs for managing resources in an ALSB domain, including:
  - Query, export, and import resources
  - Obtain validation errors
  - Get and set environment values
  - Modify references inside resources to new references
  - Move, rename, clone, and delete resources
- Customization Class
  - Find and replace environment values
  - Assign environment values
  - Map references found in resources to other references

Management and Monitoring

The JMX Monitoring API in ALSB provides external access to monitoring data. Java Management Extensions (JMX) technology was used for the implementation. ALSB resources within a domain use JMX Managed Beans (MBeans) to expose their management functions. An MBean is a concrete Java class that is developed according to JMX specifications.

For information, see the JMX Monitoring API Programming Guide.

Deployment

You can use the ALSB MBeans in Java programs and WLST scripts to automate promotion of ALSB configurations from development environments through testing, staging, and finally to production environments.

Numerous customization options can be applied during deployment. For example, an extended list of environment variables allows you to preserve or tailor settings when moving from one environment to another.

For information, see Using the Deployment APIs in the *AquaLogic Service Bus Deployment Guide*.