BEA AquaLogic Service Bus™

User Guide

Version: 2.6
Document Revised: January 2007

Contents

Introduction to AquaLogic Service Bus

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

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

AquaLogic Service Bus 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 provide an overview of AquaLogic Service Bus and of this document:

- "Document Scope and Audience" on page 1-2
- "Document Organization" on page 1-2

Document Scope and Audience

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

Information for operations specialists such as Monitoring, Reporting, and Tracing resides in the AquaLogic Service Bus Operations Guide.

Information for security architects and developers resides 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 AquaLogic Service Bus Console. For more information on using the AquaLogic Service Bus Console, see Using the AquaLogic Service Bus Console.

Document Organization

This document includes the following topics:

- Modeling Message Flow in AquaLogic Service Bus: Guidelines for modeling message flows in AquaLogic Service Bus. A message flow defines the implementation of a proxy service, which is the AquaLogic Service Bus definition of an intermediary Web services that is hosted locally on AquaLogic Service Bus. In AquaLogic Service Bus, service clients exchange messages with an intermediary proxy service rather than directly with a business service.
- Using the Test Console: Using the test console to test proxy services, business services, and some of the resources created and used in AquaLogic Service Bus.
- UDDI: Using Universal Description, Discovery and Integration (UDDI) registries with AquaLogic Service Bus. 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.
- Transports: Transport protocols available in AquaLogic Service Bus.
- EJB Transport: EJB transport features and business services.
- Local Transport: Local transport features and use cases.

- Extensibility Using Java Callouts and POJOs: Guidelines for using the Java callout action with POJOs.
- XQuery Implementation: AquaLogic Service Bus uses the BEA AquaLogic Data Services Platform implementation. This section describes valid extensions of the AquaLogic Data Services Platform for BEA AquaLogic Service Bus and AquaLogic Service Bus-specific XQuery functions.
- Tuning AquaLogic Service Bus: Optimizing the AquaLogic Service Bus performance in a production environment.
- Debugging AquaLogic Service Bus: Enabling debugging in AquaLogic Service Bus modules.

CHAPTER 2

Modeling Message Flow in AquaLogic Service Bus

In AquaLogic Service Bus, the Message Flow defines the implementation of a proxy service. You configure AquaLogic Service Bus proxy services in the AquaLogic Service Bus Console, which is described in Using the AquaLogic Service Bus Console. This section presents guidelines for modeling and designing message flows. It contains the following topics:

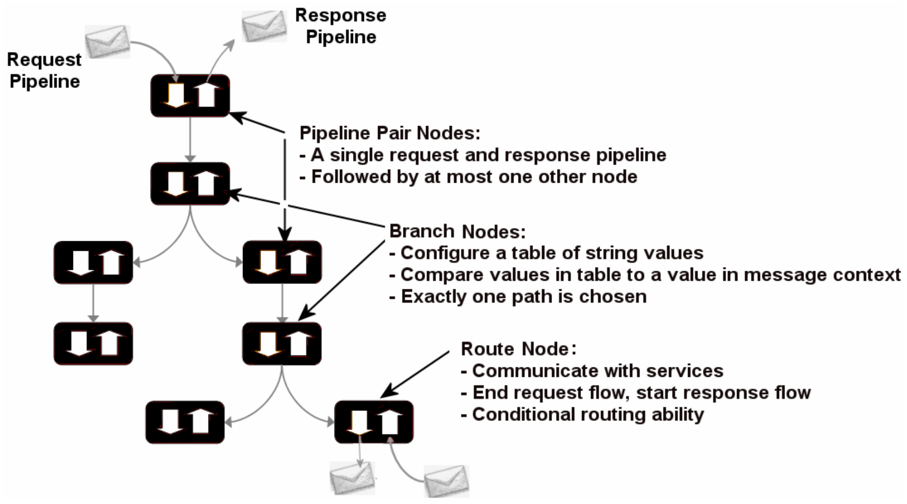
- “Working with Variable Structures” on page 2-54
- “Quality of Service” on page 2-70
- “Content Types, JMS Type, and Encoding” on page 2-76
- “Throttling Pattern” on page 2-77
- “WS-I Compliance” on page 2-77
- “Converting Between SOAP 1.1 and SOAP 1.2” on page 2-82

About AquaLogic Service Bus Message Flow

A message flow consists of the pipelines, branch nodes, and route nodes that together define the implementation of an AquaLogic Service Bus proxy service. A proxy service is an AquaLogic Service Bus definition of an intermediary Web Service that is hosted locally on AquaLogic Service Bus. Using the AquaLogic Service Bus Console, you can configure the logic for the manipulation of messages in proxy service message flow definitions. This logic includes such activities as transformation, publishing, and reporting—the logic is configured in individual actions within the message flow.

The following figure shows a high level view of the components of the message flow definition.

Figure 2-1 Components of Message Flow


Response Pipeline
Request Pipeline
Pipeline Pair Nodes:
- A single request and response pipeline
- Followed by at most one other node
Branch Nodes:
- Configure a table of string values
- Compare values in table to a value in message context
- Exactly one path is chosen
Route Node:
- Communicate with services
- End request flow, start response flow
- Conditional routing ability

This topic includes the following sections:

- "Building a Message Flow" on page 2-3
- "Message Execution" on page 2-5

Building a Message Flow

Any component can be at the root of a message flow. (For a description of the components, see Table 2-1, "Message Flow Components," on page 2-4). One of the simplest of message flow designs is to have only a route node representing the entire flow. No restrictions exist on what two components you can chain together to create a message flow. For example, two pipeline pair nodes can be linked together without a branch node in between. In the case of branch nodes, each branch node can start with a different element. One branch can terminate with a route node, another can be followed by a pipeline pair, and yet another may have no descendant. In the latter case, a branch with no descendants means that at run time, when this branch is executed, response processing begins immediately. However, in general a message flow is likely to be designed in one of the following forms:

- In the case of non-operational services (services that are not based on WSDLs with operations), the flow likely consists of a single pipeline pair at the root followed by a route node.
- In the case of operational services, the flow likely 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.

A message flow is constructed by linking together instances of the top-level components described in the following table. Subsequent sections in this topic describe the node types in more detail.

Table 2-1 Message Flow Components

Node Type	Summary
Pipeline Pair See "Pipelines" on page 2-6.	A pipeline pair combines a single request and a single response pipeline into 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 AquaLogic Service Bus processes a pipeline pair node. The execution path is reversed when AquaLogic Service Bus processes the response pipeline. For an example of a simple pipeline pair node, see Figure 2-3. To learn how to configure a pipeline pair node, see "Adding a Pipeline Pair Node" in Proxy Services: Message Flow in *Using the AquaLogic Service Bus Console*.
Branch See "Branching in Message Flows" on page 2-9.	A branch node allows processing to proceed along exactly one of several possible paths. Branching is driven by an XPath-based switch table. Each branch in the table specifies a condition (for example, `<500`) that is evaluated in order down the message flow against a single XPath expression (for example, `./ns:PurchaseOrder/ns:totalCost` on `$body`). Whichever condition is satisfied first determines which branch is followed. If no branch condition is satisfied, then the default branch is followed. A branch node may have several descendants in the message flow: one for each branch, including the default branch. **Note:** It is highly recommended that you define a default branch whenever your message flow involves conditional branching. To learn how to add a branch node, see "Adding a Conditional Branch Node" in Proxy Services: Message Flow in *Using the AquaLogic Service Bus Console.* For information about working with the message context variables to design conditions, see Chapter 3, "Message Context."

Table 2-1 Message Flow Components

Node Type	Summary
Route	A route node is used to perform 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. To learn how to add a route node, see Adding a Route Node in Proxy Services: Message Flow in the *Using the AquaLogic Service Bus Console*.

To create a message flow, see “Viewing and Changing Message Flow” in Proxy Services: Message Flow in *Using the AquaLogic Service Bus Console*.

Message Execution

The following table gives brief description of the components in a typical message flow

Table 2-2 Path Of a Message during a Message Flow

Message Flow Node	What Happens During Message Processing?
Request Processing	Request processing begins at the root of the message flow.
Pipeline Pair	Executes the request pipeline only.
Branch	Evaluates the branch table and proceeds down the relevant branch.
Route	Performs the route along with any request transformations. **Note:** In the message flow, regardless of whether routing takes place or not, the route node represents the change-over 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.

Table 2-2 Path Of a Message during a Message Flow

Message Flow Node	What Happens During Message Processing?
Route	Executes any response transformations. See "Route" on page 2-5 for Request Processing.
Branch	Skips any branch nodes and continues with the node that preceded the branch.
Pipeline Pair	Executes the response pipeline.
Root of the Message Flow	Sends the response back to the client.

Pipelines

The principal component in a proxy service implementation is the *pipeline*. A pipeline is a named sequence of stages representing a non-branching one-way processing path.

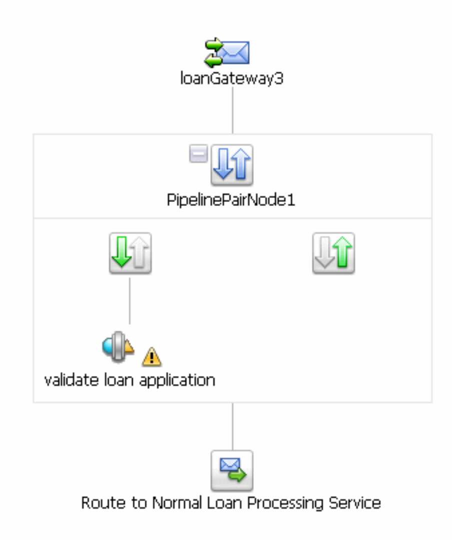
Pipelines belong to one of the following categories:

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

To create the request and response paths, you pair request and response pipelines and organize them into a single node called a *pipeline pair node*.

"Message Flow Definition for a Proxy Service" on page 2-7 shows an example of a simple message flow. It defines a proxy service named `loanGateway3`.

Figure 2-2 Message Flow Definition for a Proxy Service

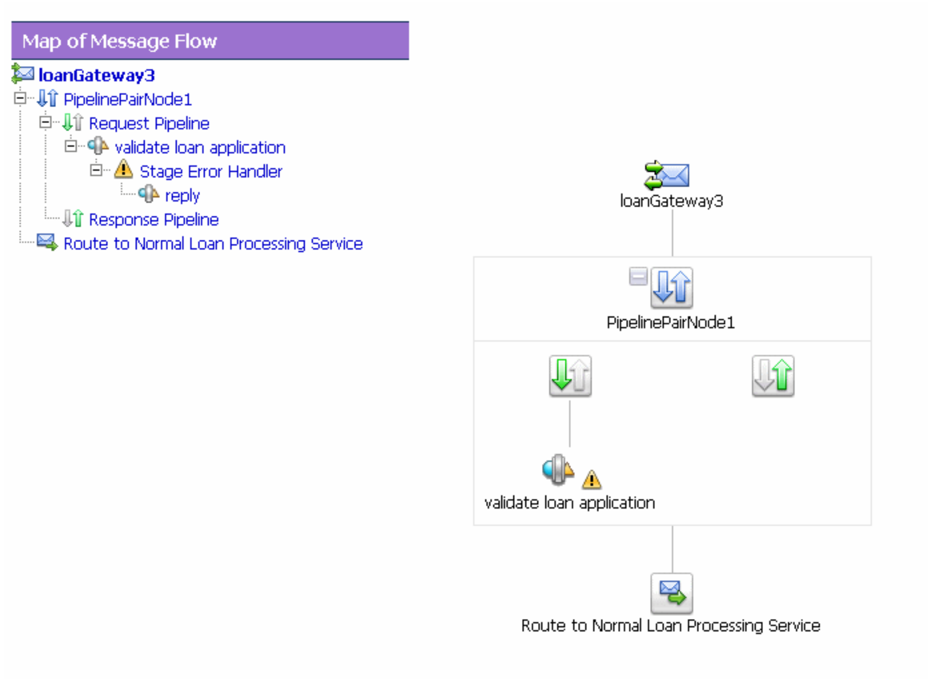

loanGateway3
PipelinePairNode1
validate loan application
Route to Normal Loan Processing Service

The message flow in the preceding figure shows:

- A start node is the root of the tree structure for the `loanGateway3` proxy service.
- A pipeline pair node (`PipelinePairNode1`), which includes request and response pipelines. The request pipeline includes one stage (`validate loan application`). The ⚠ icon associated with the `validate loan application` stage indicates that an error handler is defined for this stage. For more information about error handlers, which are also implemented as message flows, see "Handling Errors" on page 2-29.
- A Route node (`Route to Normal Loan Processing Service`)

In addition to the view of the message flow shown in the preceding figure, the AquaLogic Service Bus Console displays the corresponding tree view map of the message flow to help you navigate components of a message flow at design time.

Figure 2-3 Message Flow Definition for a Proxy Service



To view or edit the components of the message flow, click the component in the **Map of Message Flow** view. To edit or view a component from the tree view map, click the component and select the appropriate action from the list.

This flow structure provides a clear overview of the message flow behavior at design time, making both routes and branch conditions explicit parts of the overall design, rather than locating them out of view inside a pipeline stage or route node. A branch node allows you to conditionally execute these pipeline pairs, and route nodes at the ends of the branches perform the request and response dispatching. For more information about branch nodes, see "Branching in Message Flows" on page 2-9.

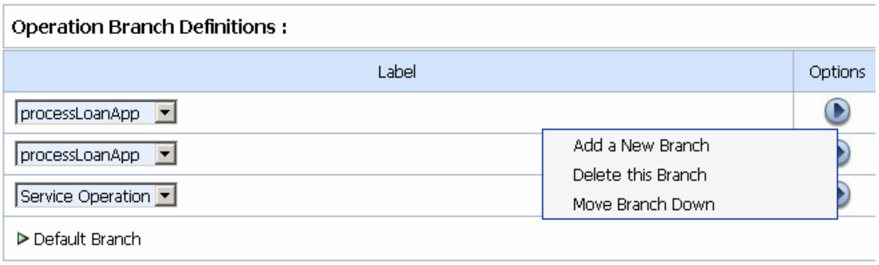
Branching in Message Flows

Two kinds of branching are supported in message flows: *operational* and *conditional* branching. The following sections explain when to use operational branching and when to use conditional branching.

Operational Branching

When message flows define Web Services Description Language (WSDL)-based proxy services, operation-specific processing is required. Instead of configuring a branching node based on operations manually, AquaLogic Service Bus provides a minimal configuration branching node that automatically branches based on operations. In other words, when you create an operational branch node in a message flow, you can quickly build your branching logic based on the operations defined in the WSDL because the AquaLogic Service Bus Console presents those operations in the branch node configuration page (Figure 2-4).

Figure 2-4 Definition for an Operation Branch



You must use operational branching in situations 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. To learn how to configure operational branch nodes, see “Adding an Operational Branch Node” and “Viewing and Changing Operational Branch Details” in Proxy Services: Message Flow in *Using the AquaLogic Service Bus Console*.

Conditional Branching

If the proxy service is not based on a WSDL and receives multiple document types as input, consider using a conditional branch node.

Conditional branching is driven by a lookup table with each branch tagged with a simple, but unique, string value. A variable in the message context is designated as the lookup variable for that node, and at run time, its value is used to determine which branch to follow. If no branch matches the value of the lookup variable, then the default branch is followed. You should design the proxy service in such a way that the value of the lookup variable is set before reaching the branch node.

Note: It is highly recommended that you define a default branch whenever your message flow involves conditional branching.

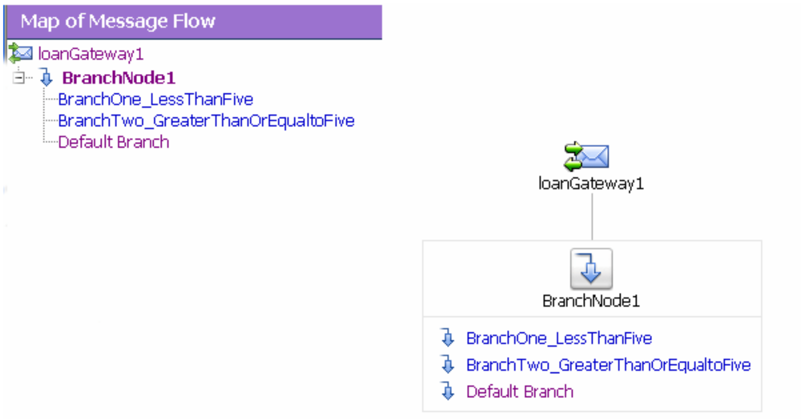
For example, consider a case when a proxy service is of type **Any SOAP** or **Any XML**, and you need to determine the type of the message is so that you can perform conditional branching. In this case you can design a stage action to identify the message type and then design a conditional branching node in the flow to separate processing based on the message type you receive. When you design the conditional branch node in a message flow, you build the branching logic based on evaluation of the value of the variable populated in the preceding stage.

For more information on conditional branch nodes, see “Adding a Conditional Branch Node” in Proxy Services: Message Flow in *Using the AquaLogic Service Bus Console*.

You can also use conditional branching to expose the routing alternatives at the top level flow view. For example, if you invoke service A or service B based on a condition, instead of configuring conditional branching by using a routing table within the route node, you can expose this branching in the message flow itself and use simple route nodes as the subflows for each of the branches.

Figure 2-5 shows a simple message flow with a top-level branch node (`BranchNode1`) and two subordinate route nodes. At run time, one branch is executed, causing messages to be routed to either service A or service B.

Figure 2-5 Branch Node



For more information on configuring a conditional branch in a route node, see "Adding Route Node Actions" in Proxy Services: Message Flow in *Using the AquaLogic Service Bus Console*.

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 awkward in the design interface if a large number of branches extend from the branch node.

For more information, see "Overview of Message Flow" in Proxy Services: Message Flow in *Using the AquaLogic Service Bus Console*.

Performing Transformations

This section presents guidelines to follow when you design transformations. Transformation maps describe the mapping between two data types. AquaLogic Service Bus supports data mapping that uses XQuery and the eXtensible Stylesheet Language Transformation (XSLT) standards. XSLT maps describe XML-to-XML mappings, whereas XQuery maps can describe XML-to-XML, XML to non-XML, and non-XML to XML mappings. For more information, see XQuery Transformations and XSL Transformations in *Using the AquaLogic Service Bus Console*. For information on using the BEA XQuery Mapper to create XQueries, see Transforming Data Using the XQuery Mapper in *Transforming Data Using the XQuery Mapper*.

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. AquaLogic Service Bus provides Publish Table actions also. 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 more information, see Proxy Services: Actions in *Using the AquaLogic Service Bus Console* and Chapter 3, “Message Context.”.

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

In a situation in which you need to route messages to one of two possible destinations, based on a WS-addressing header, content-based routing and the second destination requires 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” in Chapter 3, “Message Context.” 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.

For more information about:

- Quality of Service: See “Quality of Service” on page 2-70.
- Configuring pipelines: See “Pipelines” in “Overview of Message Flow” in Proxy Services: Message Flow in *Using the AquaLogic Service Bus Console*.
- Actions: See “Adding an Action” in Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.
- Route nodes: See “Adding a Route Node” in Proxy Services: Message Flow in *Using the AquaLogic Service Bus Console*.

Configuring Single and Multiple Stages in Pipelines

In AquaLogic Service Bus message flows, stages are the containers for actions that define the logic of the message flow. In most cases it is sufficient to use a single stage in a pipeline. However, some situations require the use of multiple stages. Section “Using Multiple Stages” on page 2-17 explains the usage of multiple stages in a pipeline. For information about configuring a stage, see “Adding a Stage” in Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.

The BEA AquaLogic Service Bus provides a wide range of actions with which you can configure a stage in message flows. The actions are divided into following categories:

- “Communication” on page 2-14
- “Flow Control” on page 2-15
- “Message Processing” on page 2-15
- “Reporting” on page 2-16

Note:

- Only communication and flow control actions are available in a stage under a route node.

- The actions available to a stage under a message flow pipeline are different from those in a stage under a route node.
- In a stage under a pipeline error handler all the categories of actions similar to that of the message flow pipeline are available.

Communication

The actions in this category control the message flow in the pipeline. You use them to specify the target URL for a message flow, a mode of packaging for a message flow, and a mode to configure a synchronous callout to an AquaLogic Service Bus registered proxy service or a business service. Communication actions in a stage in a message flow pipeline include:

- Dynamic Publish
- Publish Overview
- Publish Table
- Routing Options
- Service Callout
- Transport Headers

For more information on communication actions, see Proxy Services: Action in *Using the AquaLogic Service Bus Console*. The communication actions available in a route node are:

- Dynamic Routing
- Routing
- Routing Table

Note: For more information on adding action to a stage in a route node, see Proxy Services: Message Flow-Adding Route Node Actions in *Using the AquaLogic Service Bus Console.*

The communication actions available in an error handler stage are:

- Dynamic Publish
- Publish Table
- Routing Options
- Service Callout
- Transport Headers

Flow Control

The actions in this category control the message flow in the pipeline. You use them to implement conditional routing, conditional looping, and error handling within a stage in a message flow. Also you can use them to notify the invoker of success or to skip the rest of the actions in the stage. Flow actions in a stage in a pipeline include:

– For Each

– If... Then...

– Raise Error

– Reply

– Skip

For more information on actions in this category, see Proxy Services: Actions in *Using the AquaLogic Service Bus Console.*

The flow control action available in a route node is If... Then...

Note: For more information on adding an action to a stage in a route node, see Proxy Services: Message Flow-Adding Route Node Actions in *Using the AquaLogic Service Bus Console.*

The flow control actions available in an error handler stage are

– For Each

– If... Then...

– Raise Error

– Reply

– Resume

– Skip

Note: For more information on adding an action to a stage in a route node, see Proxy Services: Error Handlers in *Using the AquaLogic Service Bus Console.*

Message Processing

The actions in this category process the message flow. You can use the actions under this category to modify the XPath expressions, invoke Java methods for processing, transform the message format, and set transport headers. Message Processing actions in a stage in a message flow pipeline include:

- Assign
- Delete
- Insert
- Java Callout
- MFL Transform
- Rename
- Replace
- Validate

For more information on message processing actions, see Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.

Message processing actions available in a stage in a pipeline error handler includes:

- Assign
- Delete
- Insert
- Java Callout
- MFL Transform
- Rename
- Replace
- Validate

Reporting

You use the actions in this category to log or report errors and generate alerts if required in a message flow within a stage. Reporting actions in a stage in a message flow pipeline include:

- Alert
- Log
- Report

Reporting actions in a stage under a pipeline error handler includes:

- Alert
- Log
- Report

For more information on the reporting actions, see Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.

Using Multiple Stages

Having multiple stages in a message flow enables you to define error handlers at a modular level. Each stage in a message flow can have a separate error handling pipeline. You can use two types of actions to control runtime execution of the actions in a stage:

- **Resume**: This is typically used in the error handlers to resume the next action in the message flow pipeline.

Note: The message flow processing resumes at the next stage in the pipeline.

- **Skip**: On encountering this action the processing of the current stage is skipped and the processing continues with the next stage in the message flow.

For more information, see “Adding a Stage” and “Viewing and Changing Stage Configuration Details” in Proxy Services: Message Flow in *Using the AquaLogic Service Bus Console*.

Constructing Service Callout Messages

When AquaLogic Service Bus 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 2-18
- “SOAP RPC Style Services” on page 2-20
- “XML Services” on page 2-23
- “XML Services” on page 2-23

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 2-1 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 2-2 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 as shown in the following listing (the contents of the `myreq` and reqheader variables are shown in bold).

Listing 2-3 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 as shown in the following listing.

Listing 2-4 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 the following listing.

Listing 2-5 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, take an 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 as shown in Listing 2-6:

Listing 2-6 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 the following listing.

Listing 2-7 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 the following listing.

Listing 2-8 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 2-9 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 the preceding listing.

- The response and the value assigned to the message context variable, myresp, is shown in the following listing.

Listing 2-10 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

You can configure error handling at the Message Flow, pipeline, route node, and stage level. For information about doing so, see "Error Messages and Handling" on page 20-1. 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 A-14 and Appendix A, "Error CodesLocal Transport"

Transport Errors

When a transport error is received from an external service and there is no error response payload returned to AquaLogic Service Bus 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 the following listing.

Listing 2-11 Contents of the AquaLogic Service Bus 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. The following listing shows an example of the `ErrorResponseDetail` element.

Listing 2-12 Contents of the AquaLogic Service Bus 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 2-28.

SOAP Faults

In case an external service returns a SOAP fault, the AquaLogic Service Bus 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 AquaLogic Service Bus fault element.

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

Listing 2-13 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 2-13 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 the following listing.

Listing 2-14 Contents of the AquaLogic Service Bus 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’s 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 for the Service Callout-generated fault is shown in Listing 2-15.

XML Schema for the Service Callout-Generated Fault Details

The XML schema definition of the service callout-generated fault details is shown in the following listing.

Listing 2-15 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

The process described in the next paragraph constitutes an error handling pipeline for the error handling stage. 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 2-3 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 3-2.

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 AquaLogic Service Bus 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.

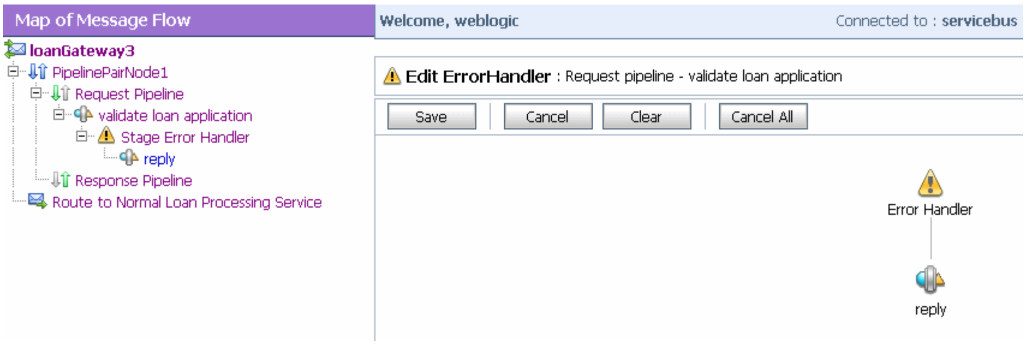
Some use cases require error reporting. You can use the report action in these situations. For example, consider a scenario in which the request pipeline reports a message for tracking purposes, but the service invoked by the route node fails after the reporting action. In this case, the reporting system 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 AquaLogic Service Bus 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 2-2 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 AquaLogic Service Bus Console as shown in the following figure.

Figure 2-6 Error Handler Message Flow



Map of Message Flow

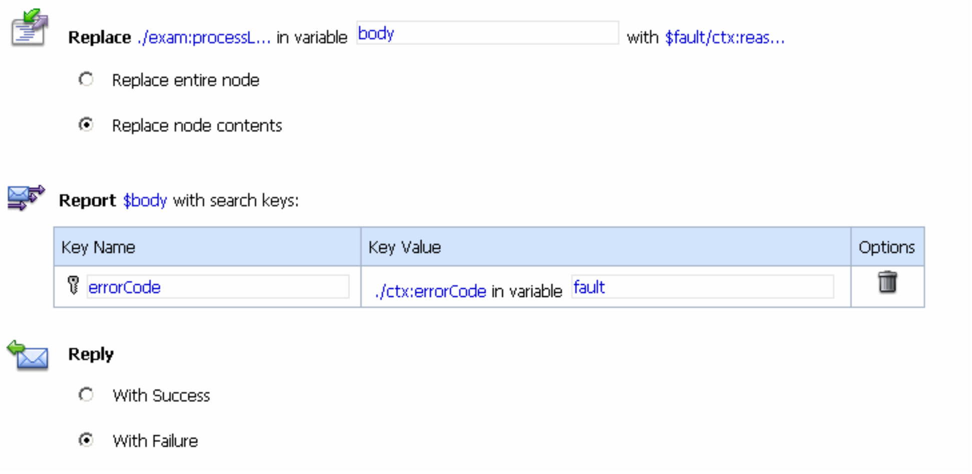
- loanGateway3
  - PipelinePairNode1
    - Request Pipeline
      - validate loan application
        - Stage Error Handler
          - reply
    - Response Pipeline
  - Route to Normal Loan Processing Service

Welcome, weblogic — Connected to : servicebus

Edit ErrorHandler : Request pipeline - validate loan application

The stage is, in turn, configured with actions (Replace, Report, and Reply) as shown in the following figure.

Figure 2-7 Actions in Stage Error Handler



Replace ./exam:processL... in variable body with $fault/ctx:reas...

- Replace entire node
- [✓ Replace node contents]

Report $body with search keys:

Key Name	Key Value	Options
errorCode	./ctx:errorCode in variable fault	🗑

Reply

- With Success
- [✓ With Failure]

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 AquaLogic Service Bus Reporting Data Stream if the error handler configured with this action is invoked. The JMS Reporting Provider reports the messages on the AquaLogic Service Bus Dashboard. AquaLogic Service Bus 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 2-2) 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*.

Selecting a Service Type

AquaLogic Service Bus supports a variety of service types that range from conventional Web services (using XML or SOAP bindings in WSDLs) to non-XML or generic services. This section provides guidelines on selecting a service type.

AquaLogic Service Bus service types for a proxy service include:

- **SOAP Services**—SOAP services receive and respond with SOAP messages. SOAP messages are constructed by wrapping the contents of the header and body variables inside a `<soap:Envelope>` element. SOAP services can be SOAP 1.1 or SOAP 1.2 services.

- **XML** Services (Non SOAP)—The messages to XML-based services are XML, but can be of any type allowed by the proxy service configuration.

- **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, and untyped binary.

- **WSDL Web Service**—In AquaLogic Service Bus you define proxy services based on WSDL. The WSDL indicates if the service is a SOAP 1.1 or a SOAP 1.2 service. Although it is not mandatory, BEA recommends that you use a WSDL to define a proxy service. For more information on WSDL based services, see "Using a WSDL to Define a Service" on page 2-35.

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

For more information on selecting a service type, see Adding a Proxy Services in *Using the AquaLogic Service Bus Console*

The following table shows the service types and the transports, which AquaLogic Service Bus supports.

Table 2-4 Supported Service Types and Transports

Service Type	Transport Protocols
SOAP or XML WSDL	HTTP HTTP(S) JMS Local
SOAP (no WSDL)	HTTP HTTP(S) JMS Local
XML (no WSDL)[1]	e-mail File FTP HTTP HTTP(S) JMS Local Tuxedo[2]

Table 2-4 Supported Service Types and Transports

Service Type	Transport Protocols
Transport Typed	EJB
Messaging Type (Binary, Text, MFL, XML)	e-mail File FTP HTTP HTTP(S) JMS Local Tuxedo

1. HTTP GET is supported for the XML (no WSDL) service type and Messaging Service.
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.

BEA recommends that you use the local transport for communication between two proxy services. For more information on local transport, see Chapter 8, "Local Transport."

Using a WSDL to Define a Service

If a service has a well defined Web Services Description Language (WSDL) interface, it is recommended, although not required, that you use the WSDL to define the service. For more information on WSDL resources in AquaLogic Service Bus, see WSDLs in *Using the AquaLogic Service Bus Console*.

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

- "SOAP Document Wrapped Web Services" on page 2-35
- "SOAP Document Style Web Services" on page 2-36
- "SOAP RPC Web Services" on page 2-38

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.

For more information on SOAP Document Wrapped Web Services see Adding a Business Service in *Using the AquaLogic Service Bus Console.*

SOAP Document Style Web Services

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

The following listing provides an example of a WSDL for a sample document style Web service using SOAP 1.1.

Listing 2-16 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 the following listing.

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

Listing 2-17 Body Variable Value

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

In Listing 2-17, `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 the following listing.

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

Listing 2-18 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.

The following listing provides an example of a WSDL for a sample RPC style Web service using SOAP 1.1.

Listing 2-19 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
```

When the WSDL shown in Listing 2-19 is used for a request, the value of the body variable (`$body`) that the SOAP RPC proxy service obtains is displayed in the following listing.

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

Listing 2-20 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 2-17, the value for the body variable (`$body`), shown in Listing 2-18, is the value of the body variable (`$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 the following listing.

Listing 2-21 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 2-9.
- 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. See WS-Polices in *Using the AquaLogic Service Bus Console*.
- 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 3-1.

  Note: The run-time contents of `$body` for a specific action can be different from the default mapping displayed in the editor. This is because AquaLogic Service Bus 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 2-54.

Binding a Service to a WSDL Port Instead of to a Binding

If you use a WSDL service type, it is useful to bind the service to a WSDL port instead of to a binding because:

- If the service is bound to port X in the template WSDL, then port X is also defined in the generated WSDL. Any other ports defined in the template WSDL are not included in the generated WSDL. Furthermore, if you base the proxy service on a WSDL port, the generated WSDL uses that port name and preserves any WS-Policies associated with that port.

  (The template WSDL is the WSDL for the service upon which you based your proxy service; the generated WSDL is the WSDL created for the new proxy service.)

- If the service is bound to binding Y in the template WSDL, the generated WSDL defines one service and port (`<service-name>QSService` and `<port-name>QSPort`). None of the ports defined in the template WSDL are included in the generated WSDL.

You can get the WSDL for an HTTP or HTTP(S)-based proxy service by entering the following URL in your browser's address field:

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

In the WSDL returned by the `http://host:port/sbresource?PROXY/project/`*`proxyname`* URL or the WSDL, which is obtained from the URL for the proxy service, the port name is preserved if the proxy service is bound to a port on the WSDL and the URL accurately reflects the URL of the proxy service. This can be important to some tools, which generate a client. The URL in the WSDL port that is bound to the service is not used when you define a service, except to populate the URL in the WSDL port as the default URL for a business service. You can overwrite the transport type and transport URL in the transport configuration UI for the service definition.

Any WS-Security policies at the port level apply. See "Overview of Proxy Services" in Proxy Services in *Using the AquaLogic Service Bus Console*.

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.

Note: For Any SOAP, you need to 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.

AquaLogic Service Bus 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 "Validate" in Proxy Services: Actions in *Using the AquaLogic Service Bus Console*. For more information on conditional XQuery expressions, see "Using the XQuery Condition Editor" in Proxy Services: Editors in *Using the AquaLogic Service Bus Console*.

You can use AquaLogic Service Bus 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 "Overview of Proxy Services" in Proxy Services in *Using the AquaLogic Service Bus Console*.

Viewing Resource Details

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

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

- WSDL (a WSDL registered as a resource in AquaLogic Service Bus)
- Schema
- MFL
- WS-Policy
- WSDL (a derived WSDL with resolved policies and port information for a proxy service—this derived WSDL is available if the proxy 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`

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

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 AquaLogic Service Bus 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 2-45.

- 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 2-45.

  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 AquaLogic Service Bus). 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 AquaLogic Service Bus.

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 2-45.
- "Creating an XQuery Resource From the Sample XML" on page 2-46.
- "Creating and Configuring the Proxy Service to Implement Dynamic Routing" on page 2-46

Sample XML File

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

Listing 2-22 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 the **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. In the **Edit Stage Configuration** page enter the name of the variable in the field. By this you assign the XQuery resource to this variable.
    This 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>{$routingtable/row[logical/text()=$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. In the **Edit Stage Configuration** page, enter the name of the variable (for example, *routeresult*) in the field.

By this you extract 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

AquaLogic Service Bus 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 AquaLogic Service Bus 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 10-4 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 AquaLogic Service Bus 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 AquaLogic Service Bus, 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 2-57.

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 AquaLogic Service Bus. Together, the `header`, `body`, and `attachments` variables, (referenced as `$header`, `$body` and `$attachments` in XQuery statements) represent the message as it flows through AquaLogic Service Bus. The canonical form of the message is SOAP. Even if the service type is not SOAP, the message appears as SOAP in the AquaLogic Service Bus 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 AquaLogic Service Bus 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 3-2.

The `$body contains` message payload variable. When a message is dispatched from AquaLogic Service Bus 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 Chapter 3, "Message Context."

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` and the transport-specific schemas, which are available in a JAR file at the following location in your AquaLogic Service Bus installation:

```
<BEA_HOME>\weblogic92\servicebus\lib\sb-schemas.jar
```

where `BEA_HOME` represents the directory in which you installed AquaLogic Service Bus.

To learn about the message context schema and the transport specific schemas, see "Message Context Schema" on page 3-28.

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 2-54.

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, AquaLogic Service Bus 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 in the AquaLogic Service Bus Console, 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” on page 2-55
- “Using Variable Structures” on page 2-55
- “Creating Variable Structure Mappings” on page 2-57

Using the Inline XQuery Expression Editor

AquaLogic Service Bus 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.

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 2-55.
- 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 2-55

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 name and type are explicitly declared. The variable can be accessed anywhere within the static scope.

In AquaLogic Service Bus, 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" on page 2-57
- "Creating the Resources You Need for the Examples" on page 2-59
- "Example 1: Selecting a Predefined Variable Structure" on page 2-62
- "Example 2: Creating a Variable Structure That Maps a Variable to a Type" on page 2-63
- "Example 3: Creating a Variable Structure that Maps a Variable to an Element" on page 2-64
- "Example 4: Creating a Variable Structure That Maps a Variable to a Child Element" on page 2-65
- "Example 5: Creating a Variable Structure that Maps a Variable to a Business Service" on page 2-66
- "Example 6: Creating a Variable Structure That Maps a Child Element to Another Child Element" on page 2-68

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.

Listing 2-23 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 tasks in this procedure include:

- "Save the WSDL as a Resource" on page 2-59
- "Create a Proxy Service That Uses the Sample WSDL" on page 2-60
- "Build a Message Flow for the Sample Proxy Service" on page 2-61
- "Create a Business Service That Uses the Sample WSDL" on page 2-61

Save the WSDL as a Resource

1. In the left navigation pane in the AquaLogic Service Bus 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 on **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 2-60.

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. Use the default values for all other fields on the **General Configuration** page, then click **Next**.
7. Use 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 2-61.

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, 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 on **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. Use the default values for all other fields on the **General Configuration** page. 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 2-62.

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, AquaLogic Service Bus 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**.

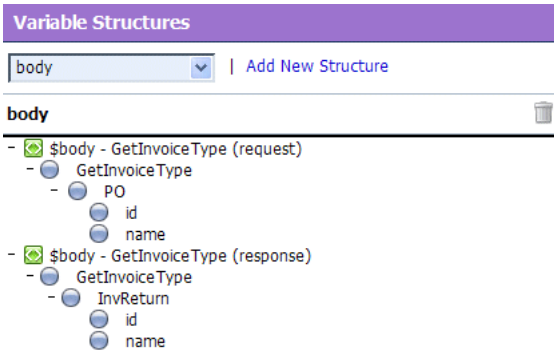
Note: This predefined structure is also supported for messaging services with a typed interface.

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 as follows:

Figure 2-8 Variable Structures—body



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

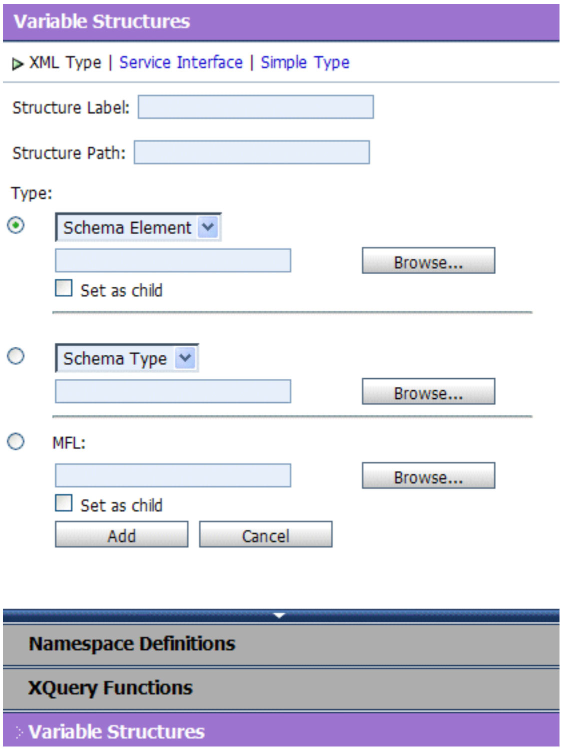
Suppose the proxy service `ProxyWithSampleWSDL` invokes a service callout to the business service `BusinessWithSampleWSDL`, which also has a service type **WSDL Web Service** that uses the binding **`POBinding`** from `SampleWSDL`. The operation `GetInvoiceType` is invoked.

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 as follows:

Figure 2-9 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.

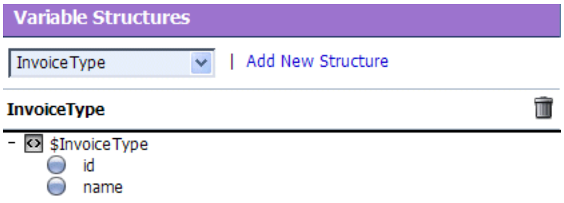
   c. In the **WSDL Browser**, select **SampleWSDL**, then select **InvoiceType** under **Types** in the **Select WSDL Definitions** pane.

   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 as follows:

Figure 2-10 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:

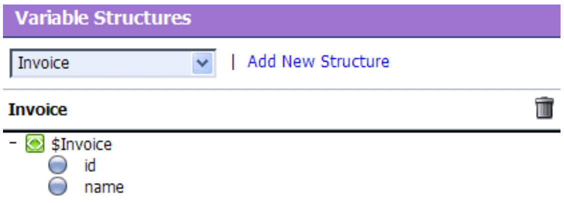
1. In the Variable Structures panel, click **Add New Structure**.

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 as follows:

Figure 2-11 Variable Structures—Invoice



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

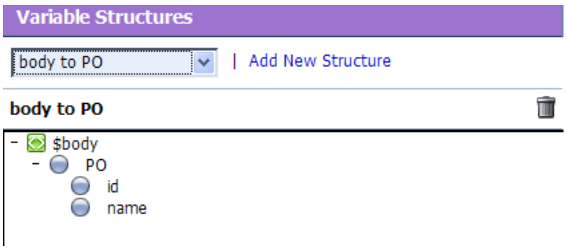
The `ProxyWithSampleWSDL` proxy service routes to the document style `Any SOAP` business service that returns the Purchase Order in the `SOAP` body. In this example, the `ProxyWithSampleWSDL` proxy service message flow must then manipulate the response. To achieve this, you can create a new structure 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 as follows:

Figure 2-12 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.

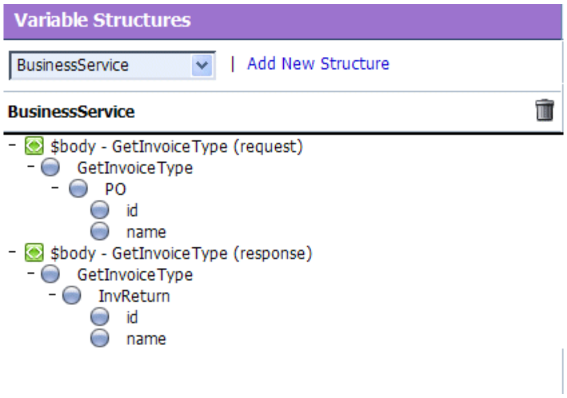
Note: This mapping is also supported for messaging services with a typed interface.

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 as shown below:

Figure 2-13 Variable Structures—BusinessService



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
```

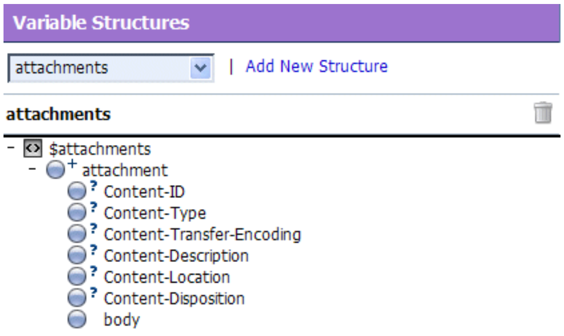
You can select and copy the **body** element from the predefined **attachments** structure, paste this element as the variable path to be mapped in the new mapping definition.

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 as follows:

Figure 2-14 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 AquaLogic Service Bus messaging:

- "Delivery Guarantees" on page 2-70
- "Outbound Message Retries" on page 2-76

Delivery Guarantees

BEA AquaLogic Service Bus supports reliable messaging. The value of the `qualityOfService` element in the outbound context variable provides AquaLogic Service Bus with a hint on the desired delivery behavior. When messages are routed to another service from a route node, the default `Quality of Service` element in `$outbound` is either `exactly-once` or `best-effort`.

The following delivery guarantee types are provided in AquaLogic Service Bus:

Table 2-5 Delivery Guarantee Types

Delivery Reliability	Description
Exactly once	*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. 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 • Transactional Tuxedo **Note:** Do not retry the outbound transport when the `QoS` is exactly once

Table 2-5 Delivery Guarantee Types

Delivery Reliability	Description
At least once	*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 performance or availability 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: • JMS/nonXA • HTTP • HTTP(S) • 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.

Overriding the Default Element Attribute

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 3-28.

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 “Adding a Proxy Service” and “Adding a Business Service” in Proxy 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:

Table 2-6 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 AquaLogic Service Bus, see *AquaLogic Service Bus Deployment Guide*.

The following are some more 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 BEA AquaLogic Service Bus 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 AquaLogic Service Bus 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 and HTTP(S) transports, any HTTP status other than 200 or 202 is considered an error by AquaLogic Service Bus and must be retried. Because of this algorithm, it is possible that AquaLogic Service Bus retries errors like authentication failure that may never be rectified for that URL within the time period of interest. On the other hand, if AquaLogic Service Bus 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, AquaLogic Service Bus allows you to control the content type used, the JMS type used, and the encoding used.

AquaLogic Service Bus does not make assumptions about what the external client or service needs, and uses the information configured for this purpose in the service definition. AquaLogic Service Bus derives the content type for outbound messages from the service type and interface. Content type is a part of the e-mail and HTTP(S) 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 AquaLogic Service Bus Console.

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 3-8.

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

Throttling Pattern

A throttling pattern is typically used with an HTTP Web service to restrict the degree of concurrency, that is to keep the number of outstanding requests without a response below a limit. Instead of accessing the business service directly, you access the business service through another proxy service. This proxy service typically uses the JMS one-way transport or JMS request response transport to communicate with the business service.You should define a work manager for the JMS request queue. For more information on defining a work manager, see Work Manager. Configure the work manager to have the maximum number of threads. This restricts the number of requests that can be placed in the request queue. That is no requests can be placed in the queue if the number of incoming request exceeds the maximum number of threads configured in the work manager.

Note: Set the `qualityOfService` of the business service `$outbound` to `Exactly Once`. You can use the **Routing Options** action to set the required `qualityOfService`. For more information, see Routing Options *Using the AquaLogic Service Bus Console*.

When a throttling pattern is implemented in a cluster, the total number of requests across all the Managed Servers should be equal to
`maximum number of threads on the work manager / number of managed servers`

WS-I Compliance

BEA AquaLogic Service Bus 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 AquaLogic Service Bus Console to specify whether you want AquaLogic Service Bus to enforce WS-I compliance for the service. For more information on how to do this, see “Adding a Proxy Service” in Proxy Services in *Using the AquaLogic Service Bus Console*.

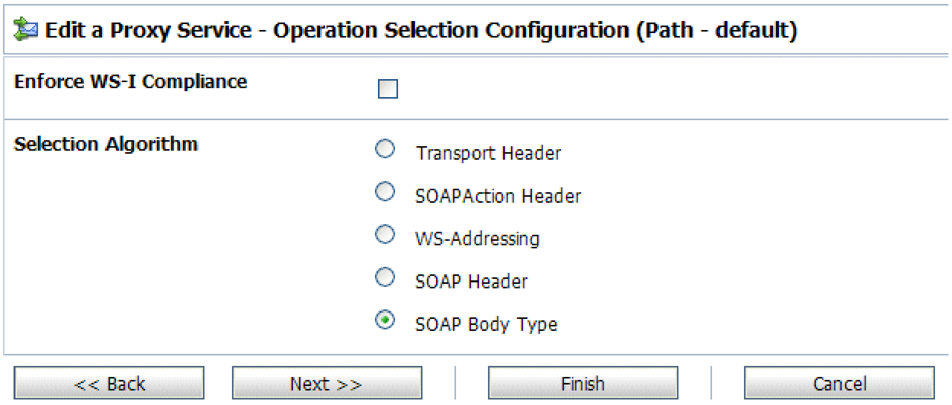
When you configure WS-I compliance for a proxy service, checks are performed on inbound request messages received by that proxy service. When you configure WS-I compliance for an invoked service, checks are performed when any proxy receives a response message from that invoked service. BEA recommends 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 “Error Messages and Handling” in Proxy Services in *Using the AquaLogic Service Bus Console*.

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 AquaLogic Service Bus 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 2-15:

Figure 2-15 Enforce WS-I Compliance Checkbox



Edit a Proxy Service - Operation Selection Configuration (Path - default)

Enforce WS-I Compliance ☐

Selection Algorithm

- ○ Transport Header
- ○ SOAPAction Header
- ○ WS-Addressing
- ○ SOAP Header
- ● SOAP Body Type

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, AquaLogic Service Bus carries out the following checks.

Table 2-7 AquaLogic Service Bus WS-I Compliance Checks

Check	WS-I Basic Profile Details	AquaLogic Service Bus 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 2-7 AquaLogic Service Bus WS-I Compliance Checks

Check	WS-I Basic Profile Details	AquaLogic Service Bus 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 2-7 AquaLogic Service Bus WS-I Compliance Checks

Check	WS-I Basic Profile Details	AquaLogic Service Bus 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. AquaLogic Service Bus 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

AquaLogic Service Bus 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

AquaLogic Service Bus 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.

Message Context

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

The Message Context Model

The BEA AquaLogic Service Bus message context is a set of properties that hold message content as well as information about messages as they are routed through AquaLogic Service Bus. These properties are referred to as context variables—for example, service endpoints are represented by predefined context variables. AquaLogic Service Bus 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

The following table 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 3-28.

Table 3-1 Predefined Context Variables in AquaLogic Service Bus

Context Variable[1]	Description	See Also...
`header`	For SOAP message, 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 3-3
`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.	"Message-Related Variables" on page 3-3
`attachments`	Contains the MIME attachments for a given message.	"Message-Related Variables" on page 3-3

Table 3-1 Predefined Context Variables in AquaLogic Service Bus

Context Variable[1]	Description	See Also...
`inbound`	Contains: • Information about the proxy service that received a message • The inbound transport headers	"Inbound and Outbound Variables" on page 3-8
`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 3-8
`operation`	Identifies the operation that is being invoked on a proxy service.	"Operation Variable" on page 3-18
`fault`	Contains information about errors that have occurred during the processing of a message.	"Fault Variable" on page 3-18

1. The "Message Context Schema" on page 3-28 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 AquaLogic Service Bus. 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 AquaLogic Service Bus. 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 3-6.

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 the following table.

Table 3-2 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 3-28 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 3-6.

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).

AquaLogic Service Bus 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, AquaLogic Service Bus 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 AquaLogic Service Bus 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 AquaLogic Service Bus 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 AquaLogic Service Bus, 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 3-25.

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—AquaLogic Service Bus does not de-reference the URI, and the content is not substituted into an outgoing message.

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 AquaLogic Service Bus 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 Settings in the Test Console" on page 4-24. 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 3-28.
- 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).

AquaLogic Service Bus 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 3-20. 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 3-3 Sub-Elements of the service Element

Sub-Elements[1]	Description...
`providerName`	Specifies the name of the proxy service 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 3-28 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 the following table.

Table 3-4 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 3-4 Sub-Elements of the Transport Element

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 is available in a JAR file at the following location in your AquaLogic Service Bus installation: *`BEA_HOME`*`\weblogic92\servicebus\lib\sb-schemas.jar` where *`BEA_HOME`* represents the directory in which you installed AquaLogic Service Bus. **Initialization** The URI element is initialized as follows: • Initialized on the `inbound` variable using information from the request message received by AquaLogic Service Bus. • 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 3-4 Sub-Elements of the Transport Element

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 is available in a JAR file at the following location in your AquaLogic Service Bus installation: *`BEA_HOME`*`\weblogic92\servicebus\lib\sb-schemas.jar` where *`BEA_HOME`* represents the directory in which you installed AquaLogic Service Bus. **Initialization** The `URI` element is initialized as follows: • Initialized on the `outbound` variable using information from the response message received by AquaLogic Service Bus. • 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 3-4 Sub-Elements of the Transport Element

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 3-4 Sub-Elements of the Transport Element

Sub-Elements[1]	Description...
`retryCount` (`outbound` only)	Specifies the number of retries to attempt when sending a message from AquaLogic Service Bus. If `retryCount` is set, the setting overrides any retry count value configured in the target service configuration.
`retryInterval` (`outbound` only)	Specifies the interval, in seconds, to wait before attempting to re-send a message from AquaLogic Service Bus. If `retryInterval` is set, the setting overrides any retry interval value configured in the target service configuration.

1. The "Message Context Schema" on page 3-28 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 the following table.

Table 3-5 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 AquaLogic Service Bus. The inbound `transportClient` element is read-only.

Table 3-5 Sub-Elements of the Security Element

Sub-Elements[1]	Description...
`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 AquaLogic Service Bus. The inbound `messageLevelClient` element is read-only.
`doOutboundWss` (`outbound` only)	AquaLogic Service Bus 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 AquaLogic Service Bus 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 3-28 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.

AquaLogic Service Bus 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 the following table.

Table 3-6 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

Table 3-6 Sub-Elements of the Fault Variable

Elements of the Fault Variables (Continued)	Description[1]...
`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/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 3-28 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 AquaLogic Service Bus 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()
```

AquaLogic Service Bus 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.

AquaLogic Service Bus 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" on page 2-29.

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. The following table summarizes how context variables are initialized.

Table 3-7 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 3-8. For information about the initialization of sub-elements of `outbound`, see "Sub-Elements of the inbound and outbound Variables" on page 3-9.
`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 3-9.
`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 3-22 • "Initializing the header and body Context Variables" on page 3-22 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 3-6)

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 3-6). 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 AquaLogic Service Bus—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 AquaLogic Service Bus 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 in the AquaLogic Service Bus Console, see the following topics:

"Working with Variable Structures" on page 2-54.

Proxy Services: XQuery Editors in *Using the AquaLogic Service Bus Console.*

Constructing Messages to Dispatch

When AquaLogic Service Bus 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 AquaLogic Service Bus 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 AquaLogic Service Bus 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 AquaLogic Service Bus 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 AquaLogic Service Bus 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 AquaLogic Service Bus 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, AquaLogic Service Bus 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 3-3). 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 3-6.

Related Topics

"Message Context Schema" on page 3-28

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 3-28.

When working with the message context variables, you need to reference `MessageContext.xsd` and the transport-specific schemas, which are available in a JAR file at the following location in your AquaLogic Service Bus installation:

```
BEA_HOME\weblogic92\servicebus\lib\sb-schemas.jar
```

where *`BEA_HOME`* represents the directory in which you installed AquaLogic Service Bus. `sb-schemas.jar` includes the following context-related schemas:

- Alert Reporting Schema (`AlertReporting.xsd`)
- Email Transport Schema (`EmailTransport.xsd`)
- File Transport Schema (`FileTransport.xsd`)
- FTP Transport Schema (`FTPTransport.xsd`)
- HTTP Transport Schema (`HttpTransport.xsd`)
- HTTPS Transport Schema (`HttpsTransport.xsd`)
- Message Context Schema (`MessageContext.xsd`)
- Message Reporting Schema (`MessageReporting.xsd`)
- JMS Transport Schema (`JmsTransport.xsd`)
- Reference Schema(`ServiceBusReference.xsd`)
- Common Transport Schema (`TransportCommon.xsd`)

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 3-8

"Performing Operations on Context Variables" on page 3-23

"Constructing Messages to Dispatch" on page 3-25

Using the Test Console

The BEA AquaLogic Service Bus Test Console is a browser-based test environment used to validate and test the design of your system. It is an extension of the AquaLogic Service Bus Console. You can configure the object of your test (proxy service, business service, XQuery, XSLT, MFL resource), execute the test, and view the results in the console. In some instances 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.

The test console can be invoked to test any proxy service or business service and certain resources used by these services. You can also do in-line XQuery testing.

You can invoke the test console in a number of ways in the AquaLogic Service Bus Console, depending on what part of your process you want to test. You can invoke the test console from:

- The Project Explorer
- The Resource Browser
- The XQuery Editor

You can run and test a proxy service that makes a call to another proxy service or business service and vice versa. You can test the resources used by your services. When testing services you must be aware of the information that is passing from the test console to the service and vice versa.

Features

The test console supports the following features:

- Testing proxy services
- Testing business services
- Testing resources
- Testing in-line XQueries
- Tracing the message through the message flow (for proxy services only)

Prerequisites

To use the test console:

- You must have AquaLogic Service Bus 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 inline 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. Inline 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 AquaLogic Service Bus Console's Project Explorer module. The URI displays in the Endpoint URI row of the Transport Configuration table.

Testing Proxy Services

You must have activated a session to test a proxy service. You can test a proxy service from the Resource Browser or Project Explorer. You can test the following types of proxy services:

- WSDL Web Service
- Messaging Service
- Any Soap Service
- Any XML Service

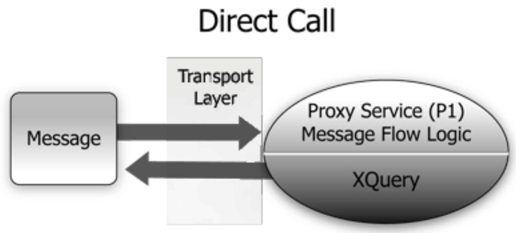
Direct Calls

A Direct Call is used to test a proxy service that is collocated in the AquaLogic Service Bus 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 a direct call testing you bypass the monitoring framework for the message and

The following figure illustrates a direct call. Note that the message bypasses the transport layer; it is delivered directly to the proxy service (P1).

Figure 4-1 Direct Call to Test a Proxy Service



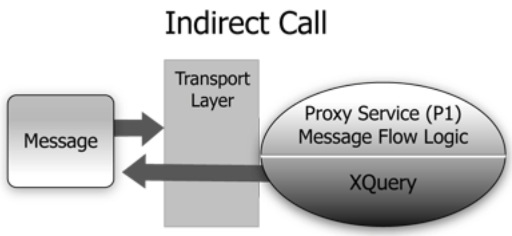
A Direct Call strategy is best suited for testing proxy services' internal message flow logic. Your test data should simulate the expected message state at the time it is dispatched. Use this test approach in conjunction with setting custom (inbound) transport headers in the test console's Transport section 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 checked), 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 invocation run-time path.

The following figure 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 4-2 Indirect Call to Test a Proxy Service



This testing strategy is recommended 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's Transport panel to accurately simulate the service call. For more information on Transport settings in the test console, see "Test Console Transport Settings" on page 4-22.

Using the *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.

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.

(This transport-level access control is achieved through the Web Application layer—in other words, even in the case that an indirect call is made through the AquaLogic Service Bus Console transport layer, an HTTP request is not sent over the network and this 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" on page 4-24.

Testing Business Services

You must have activated a session to test 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 Calls" on page 4-3 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 is 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 AquaLogic Service Bus 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.

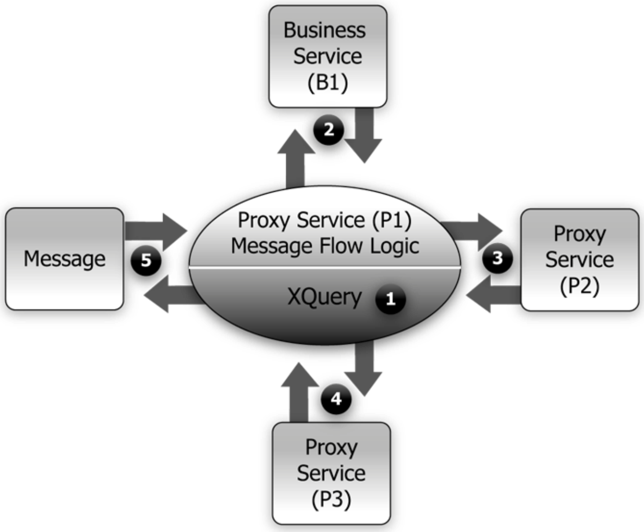
Transport Security

When using the test console to test HTTP(S) business services with BASIC authentication, the test console authenticates with the user name-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 with the service account associated with the business service.

Recommended Approaches to Testing Proxy and Business Services

In the scenario depicted in the following figure, 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 4-3 Test Scenario Example



There are many valid test strategies for this scenario. The following are recommended test strategies:

- It is recommended that you 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 the preceding figure, 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.
- It is recommended that all the XQuery expressions in a message flow be validated and tested prior to a system test. In the preceding figure, interface 1 refers to XQuery expression tests.

- Proxy service to business service (interface 2 in the preceding figure) is tested using a *indirect call*. In other words, the messages are routed through the transport layer.
- Proxy service to proxy service tests (Interfaces 3 and 4 in the preceding figure) are tested using an *indirect call*. In other words, disable the Direct Call option, which means that during the testing, the messages are routed through the transport layer.
- Your final *system* test simulates the client invoking the proxy service P1. This test is represented by interface 5 in the preceding figure.

  Test interface 5 with a Direct Call. In this way, during the testing, the messages bypass the transport layer. Tracing is automatically enabled with a Direct Call.
- It is recommended that the message state be saved 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 and 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 AquaLogic Service Bus. AquaLogic Service Bus 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:

- New variables— ✚ added —the names of all new variables and their value (values can be seen by clicking +)
- Deleted variables— ✖ deletec—the names of all deleted variables
- Changed variables—Δ changed—the names of all variables for which the value changed. The new value is visible 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)

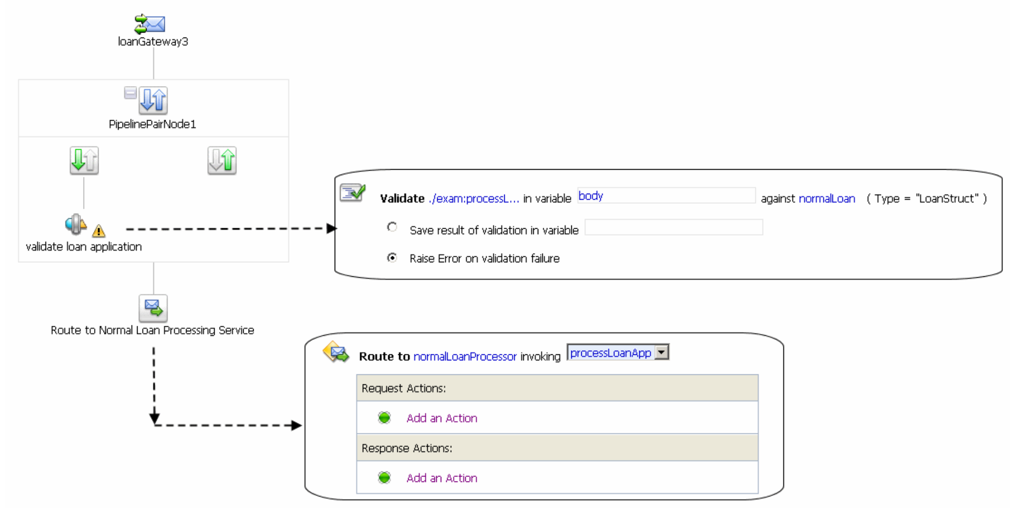
Example: Testing and Tracing a Proxy Service

This example uses one of the proxy services in the example AquaLogic Service Bus domain as a basis of instruction.

For more information on how to start the examples domain and run the examples provided there, see BEA AquaLogic Service Bus Samples. This example scenario uses the proxy service named `loanGateway3`, associated with the *Validating a Loan Application* example.

The message flow for `loanGateway3` is represented in the following figure. The figure is annotated with the configuration for the *validate loan application* stage and the configuration for the route node.

Figure 4-4 Message Flow for Proxy Service (LoanGateway3)


loanGateway3
PipelinePairNode1
validate loan application
Route to Normal Loan Processing Service
Validate ./exam:processL... in variable body against normalLoan (Type = "LoanStruct")
Save result of validation in variable
Raise Error on validation failure
Route to normalLoanProcessor invoking processLoanApp
Request Actions:
Add an Action
Response Actions:
Add an Action

To test this proxy service in the AquaLogic Service Bus examples domain using the test console, complete the following procedure:

1. Start the AquaLogic Service Bus examples domain and load the samples data, as described in *BEA AquaLogic Service Bus Samples.*
2. Log in to the AquaLogic Service Bus Console, then select **Project Explorer** and locate the `LoanGateway3` proxy service.
3. Select 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 message for the test.

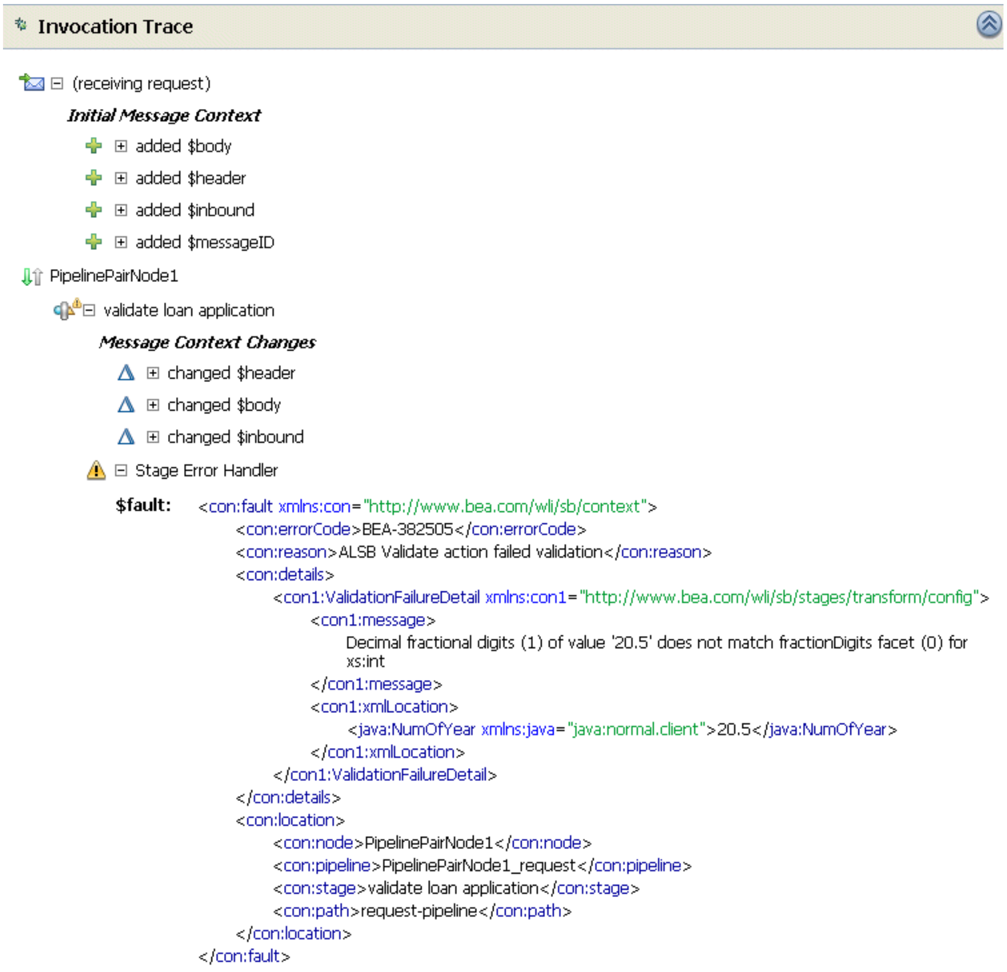
Listing 4-1 Test Message for LoanGateway3

```
<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**.

   The results page is displayed. Scroll to the bottom of the page to see the tracing results in the **Invocation Trace** panel.

Figure 4-5 Invocation Trace for a Proxy Service (LoanGateway3) Test



Invocation Trace

(receiving request)

Initial Message Context

added $body

added $header

added $inbound

added $messageID

PipelinePairNode1

validate loan application

Message Context Changes

changed $header

changed $body

changed $inbound

Stage Error Handler

$fault:

```
<con:fault xmlns:con="http://www.bea.com/wli/sb/context">
    <con:errorCode>BEA-382505</con:errorCode>
    <con:reason>ALSB Validate action failed validation</con:reason>
    <con:details>
        <con1:ValidationFailureDetail xmlns:con1="http://www.bea.com/wli/sb/stages/transform/config">
            <con1:message>
                Decimal fractional digits (1) of value '20.5' does not match fractionDigits facet (0) for
                xs:int
            </con1:message>
            <con1:xmlLocation>
                <java:NumOfYear xmlns:java="java:normal.client">20.5</java:NumOfYear>
            </con1:xmlLocation>
        </con1:ValidationFailureDetail>
    </con:details>
    <con:location>
        <con:node>PipelinePairNode1</con:node>
        <con:pipeline>PipelinePairNode1_request</con:pipeline>
        <con:stage>validate loan application</con:stage>
        <con:path>request-pipeline</con:path>
    </con:location>
</con:fault>
```

Compare the output in the trace with the nodes in the message flow shown in Figure 4-4.

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 the + 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 4-1 caused the validation error in this case.)

  For more information about this loan application scenario, see Tutorial 3: Validating a Loan Application in *AquaLogic Service Bus Tutorials*.

You can test the service using different input parameters or change the behavior of the message flow in the AquaLogic Service Bus Console Project Explorer, and run the test again to view the results.

Testing Resources

You can test resources inside an active session or from outside a session. You can test the following resources:

- "MFL" on page 4-12
- "XSLT" on page 4-14
- "XQuery" on page 4-14

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 only accepts one input and provides a single output.

The following example describes an XML input file to be tested in the test console. When you invoke the test console to test the MFL file, sample XML data is generated. Execute the test using the sample XML—in this case, a successful test results in the transformation of the message content of the input XML document in to binary format. The following "Example" on page 4-13 section describes the MFL, the test XML, and the data resulting from the test.

Example

The following listing is an example MFL file.

Listing 4-2 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 input generated by the test console to test the MFL file in the Listing 4-2 is described in the following listing.

Listing 4-3 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—the result is the Stock symbol and the stockPrice in binary format as shown in the following listing.

Listing 4-4 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 AquaLogic Service Bus. 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 displays the output XML document as a result of the test. 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 section 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.

XQuery

XQuery uses the structure of XML intelligently 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 primitive value (string, integer, date), an XML document, a sequence of the previous types.

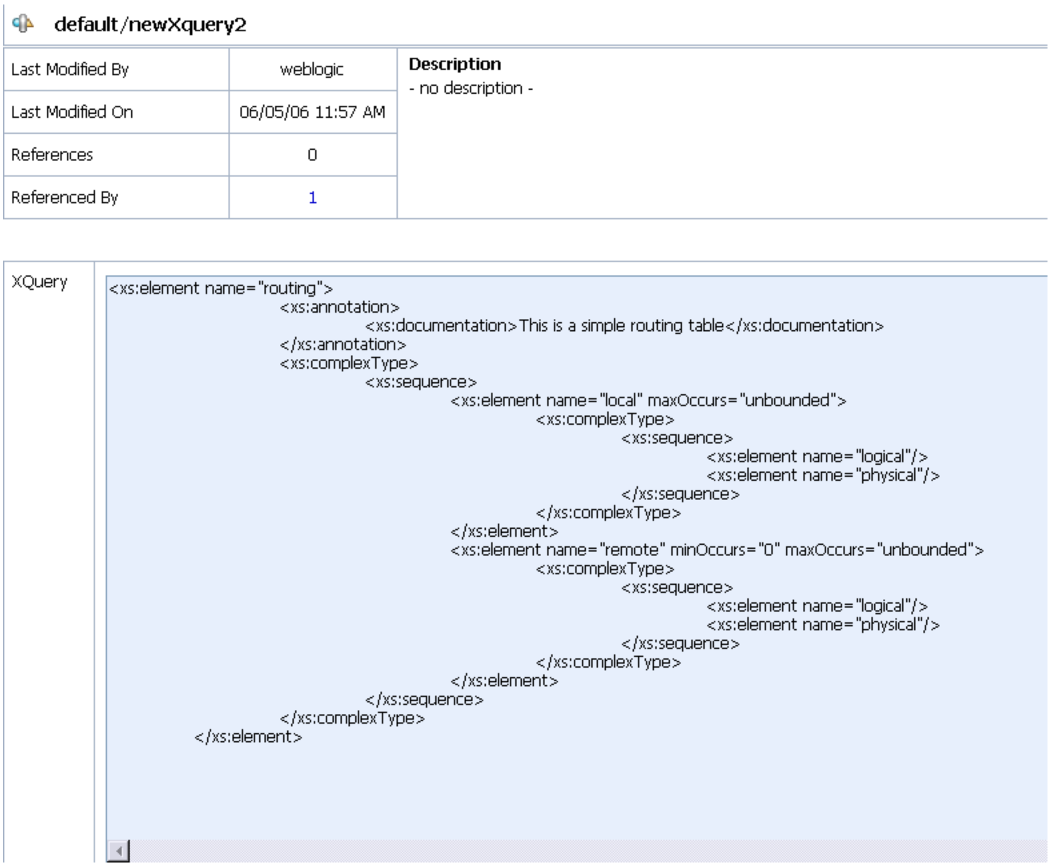
XQuery is a typed language—every external variable is given a type. The types can be categorized into the following groups:

- 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. This makes it easy to understand the type of data you must enter.

For example, the following figure shows an XQuery with three variables: int, XML, and undefined type.

Figure 4-6 Input to the XQuery Test



default/newXquery2

Last Modified By	weblogic
Last Modified On	06/05/06 11:57 AM
References	0
Referenced By	1

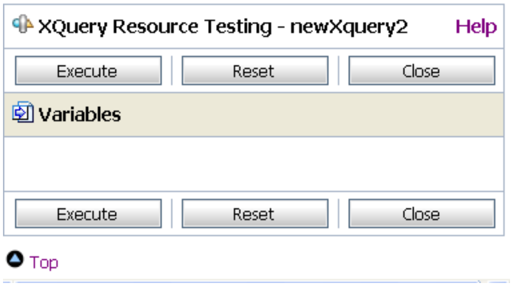
Description
- no description -

XQuery

```
<xs:element name="routing">
                <xs:annotation>
                        <xs:documentation>This is a simple routing table</xs:documentation>
                </xs:annotation>
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="local" maxOccurs="unbounded">
                                        <xs:complexType>
                                                <xs:sequence>
                                                        <xs:element name="logical"/>
                                                        <xs:element name="physical"/>
                                                </xs:sequence>
                                        </xs:complexType>
                                </xs:element>
                                <xs:element name="remote" minOccurs="0" maxOccurs="unbounded">
                                        <xs:complexType>
                                                <xs:sequence>
                                                        <xs:element name="logical"/>
                                                        <xs:element name="physical"/>
                                                </xs:sequence>
                                        </xs:complexType>
                                </xs:element>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
```

In the test console, all three variables are listed in the Variables section. By default, XML is selected for the untyped variable as it is the most typical case. You must configure these variables.

Figure 4-7 Configuring the XQuery Variables in the Test Console



You can also test an XQuery expression from the XQuery Editor.

Performing In-line XQuery Testing

You must disable the pop-up blockers in your browser for the inline 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's Options menu as well as for all toolbars that are configured to block them.

When performing in-line XQuery testing with the test console, you can use the Back button to return to the page from where you can execute a new test. But if you want to execute a new test after making changes to the in-line XQuery, you must close and re-open the test console for the changes to take effect.

Testing Services With Web Service Security

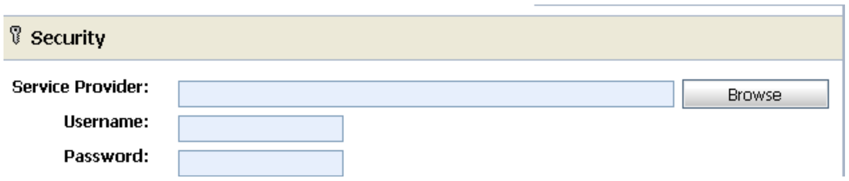
The test console supports testing proxy services and business services protected with Web Service Security (WSS). A SOAP service is protected with WSS if it has WS-Policies with WS-Security assertions assigned to it. Specifically, a service operation is protected with WS-Security if the operation's effective request and/or response WS-Policy includes WS-Security assertions. WS-Policies are assigned to a service by a mechanism called 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 WS-Policy 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 and/or encrypts the message (more precisely, parts of the message) and includes any applicable security tokens. The input to the digital signature and encryption operations is the clear-text SOAP envelope specified by the user 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 and/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 4-8 Security Panel in Test Console



If you specify a proxy service provider in the test console, all client-side PKI key-pair credentials required by WS-Security are retrieved from the proxy service 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, user name, 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.

The following table describes the different scenarios.

Table 4-1 Digital Signature and Encryption Scenarios

Scenario	Is Proxy Service 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 proxy service 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 proxy service 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 4-1 Digital Signature and Encryption Scenarios

The request policy has an Integrity assertion.	**Yes**. The client must sign the request. A proxy service 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 proxy service 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 proxy service 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 Certificate Lookup and Validation, then the certificate that maps to the proxy service provider must be registered 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 4-2 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	The user must specify a user name and password in the security section.
X.509	The service only accepts WSS X.509 tokens	The user must specify a proxy service provider in the security section and the proxy service provider must have an associated WSS X.509 credential.

Table 4-2 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	The user must specify a user name and password in the security section *or* a user name and password in the transport section. If both are specified, the one from the security section is used as the identity in the SAML token.
UNT, X.509	The service accepts UNT or X.509 tokens	The user must specify a user name and password in the security section *or* a proxy service provider in the security section 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	The user must specify a user name and password in the security section *or* a user name and password in the transport section. If both are specified, only a UNT token is sent.
X.509, SAML	The service accepts X.509 or SAML tokens	The user must specify one of the following: • a user name and password in the security section • a user name and password in the transport section • a proxy service provider with an associated WSS X.509 credential
UNT, X.509, SAML	The service accepts UNT, X.509 or SAML tokens	The user must specify one of the following: • a user name and password in the security section • a user name and password in the transport section • a proxy service 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 of 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

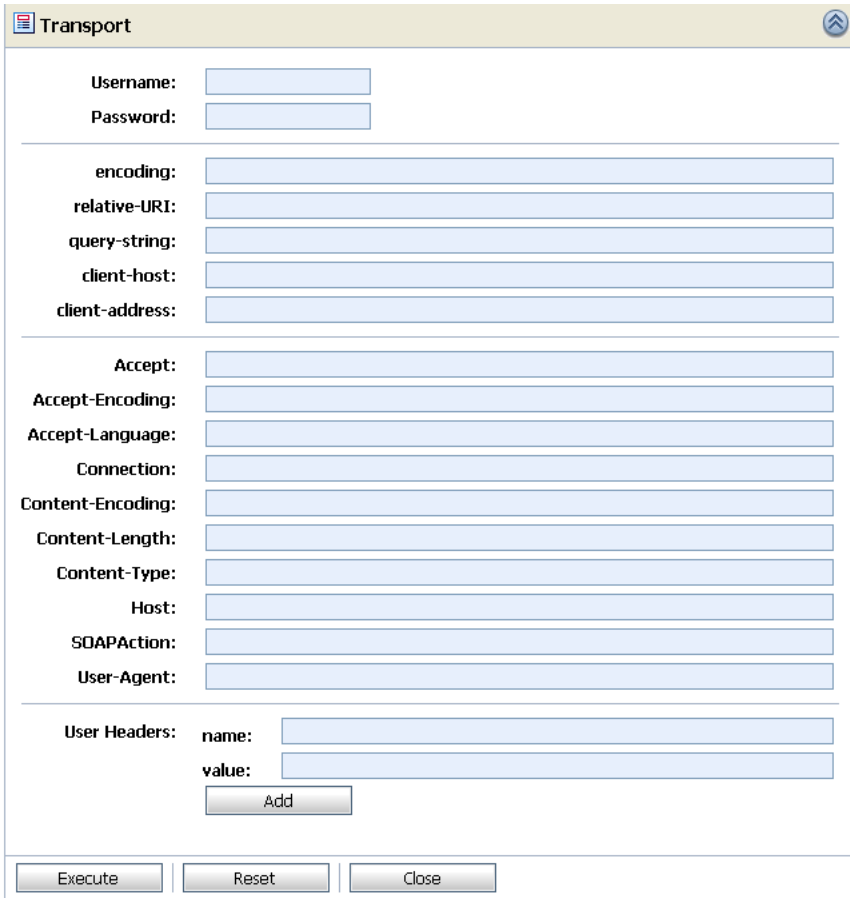
In both cases the user must specify a user name and password—the SAML assertion will be on behalf of this user. If SAML holder-of-key is configured as an integrity policy, the user must also specify a proxy service provider. The proxy service provider must have a digital signature credential assigned 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 printed with extra white spaces. Because white spaces can affect the semantic 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

The transport panel in the test console provides the functionality to specify the metadata and transport headers for messages in your test system. The following figure shows an example of a Transport panel on the test console.

Figure 4-9 Transport Panel in the Test Console



Transport

Username:	
Password:	
encoding:	
relative-URI:	
query-string:	
client-host:	
client-address:	
Accept:	
Accept-Encoding:	
Accept-Language:	
Connection:	
Content-Encoding:	
Content-Length:	
Content-Type:	
Host:	
SOAPAction:	
User-Agent:	

User Headers: name: value:

The preceding figure displays an example of the transport panel for a given service—in this case, a WSDL-based proxy service.

You can set the metadata and the transport headers in the message flow of a proxy service. In doing this, you influence the actions of the outbound transport. You can test the metadata, the message, and the headers so that you can see the output you get in the pipeline. 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 presents depending on the proxy service. The same set of transport parameters are displayed on the page 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 grouped in the **Transport** panel, below the **Username** and **Password** fields and above the group of transport header fields. The fields displayed are based on the transport type of the service. Certain fields are pre populated in the test console depending on the operation selection algorithm you selected for the service when you defined it.

For example, in the case of the transport panel displayed in Figure 4-9, the `SOAPAction` header field is populated with "`http://example.orgprocessLoanApp`". This value was taken from the service definition (the selection algorithm selected for this proxy service was `SOAPAction Header`). For more information about the selection algorithms, see "Adding a Proxy Service" in Proxy Services in *Using the AquaLogic Service Bus Console*.

When you specify values for fields in the transport panel, be aware whether you opted to test the service using a direct or indirect call—see "Direct Calls" on page 4-3 and "Indirect Calls" on page 4-4—and specify the values according to whether the message will be processed through the transport layer or not.

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.

About Security and Transports

- When using the test console to test HTTP(S) business services with BASIC authentication, the test console authenticates with 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 with 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.

Understanding How the Run Time Uses the Transport Settings in the Test Console

The test console allows you to specify header values and metadata. However, when the message is sent out, some headers and metadata may be modified or removed, and the underlying transport may in turn, ignore some of the headers and use its own values when the test is executed.

The following table describes the headers and metadata for which there are limitations when using the test console.

Table 4-3 Limitations to Transport Header and Metadata Values You Specify in the Test Console When Testing a Service

Transport	Testing this Service Type	Description of Limitation	Transport Headers Affected
HTTP(S)[1]	**Proxy Service**	All transport headers and other fields you set are preserved at run time. This is true whether or not the **Direct Call** option is set.	All
	Business Service	The AquaLogic Service Bus run time overrides any values you set for these parameters.	• Content-Length • Content-Type • relative-URI • client-host • client-address

Table 4-3 Limitations to Transport Header and Metadata Values You Specify in the Test Console When Testing a Service

Transport	Testing this Service Type	Description of Limitation	Transport Headers Affected
JMS	**Proxy Service**	**Direct Call** When the Direct Call option is used, all transport headers and other fields you set are preserved at run time.	All
		X **Direct Call** When the Direct Call option is not used, the same limitations apply as for a transport header action configuration.	See the limitations for JMS transport headers described in "Transport Headers" in Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.
	Business Service	The same limitations apply as for a transport header action configuration.	See the limitations for JMS transport headers described in "Transport Headers" in Proxy Services: Actions in *Using the AquaLogic Service Bus Console*.

Table 4-3 Limitations to Transport Header and Metadata Values You Specify in the Test Console When Testing a Service

Transport	Testing this Service Type	Description of Limitation	Transport Headers Affected
E-Mail	**Proxy Service**	No limitations. In other words, any transport headers and other fields you set are honored by the run time. This is true whether or not **Direct Call** is specified.	
	Business Service	The AquaLogic Service Bus run time overrides any values you set for these parameters.	• Content-Type
		`From` and `Date` headers have no meaning for outbound requests. If they are set dynamically (that is, if they are set in the `$outbound` headers section), they are ignored. These headers are received in `$inbound`. `Date` is the time the mail was sent by the sender. `From` is retrieved from incoming mail headers.	• From • Date
File	**Proxy Service**	No limitations. In other words, any transport headers and other fields you set are honored by the run time.[2]	
	Business Service		
FTP	**Proxy Service**	No limitations. In other words, any transport headers and other fields you set are honored by the run time.	
	Business Service		

1. When you test proxy services, the test console never sends a HTTP request over the network, therefore transport-level access control is not applied.
2. In the case of FileName (Transport metadata)—the value you assign is used to append to the output file name. For example, 1698922710078805308-b3fc544.1073968e0ab.-7e8e-{$FileName}

UDDI

This section contains the information on the following topics:

Overview of BEA AquaLogic Service Bus and UDDI

Universal Description, Discovery and Integration (UDDI) registries are used in an enterprise to share Web services. Using UDDI services helps companies organize and catalog these Web services for sharing and reuse in the enterprise or with trusted external partners.

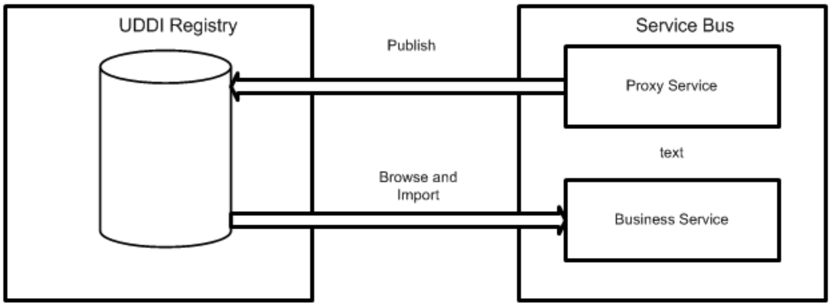
A UDDI registry service for Web services is defined by the UDDI specification available at:

`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 BEA AquaLogic Service Bus can be published to a UDDI registry. AquaLogic Service Bus can interact with any UDDI 3.0 compliant registry. BEA provides the AquaLogic Service Registry.

Figure 5-1 AquaLogic Service Bus integration with UDDI



AquaLogic Service Bus' Web-based interface to AquaLogic Service Registry makes the registry accessible and easy to use. In working with UDDI, AquaLogic Service Bus promotes the reuse of standards based Web services. In this way, AquaLogic Service Bus registry entries can be searched for and discovered and used by a 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 latest version of the UDDI specification is available at:

http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3

An 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 AquaLogic Service Bus

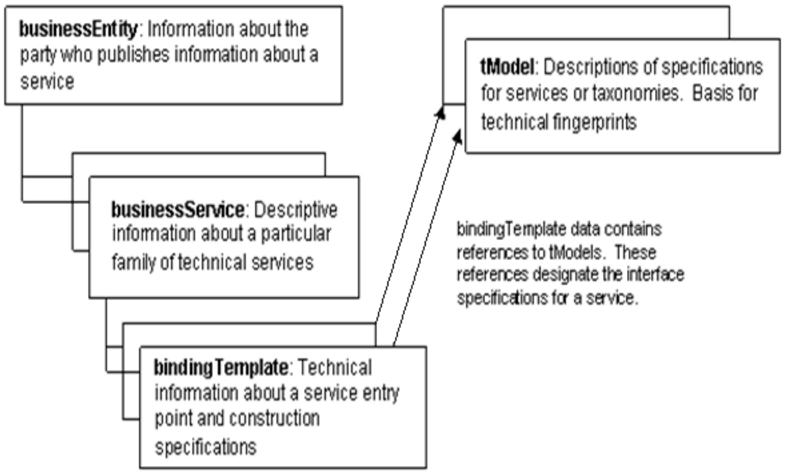
A UDDI registry stores data and metadata about business services. It is a standards-based library of catalogued and managed information about Web services for discovery and reuse by other applications. UDDI offers several benefits to IT managers at both design time and run time, including increasing code reuse. UDDI also provides benefits to developers, including the following:

- UDDI improves infrastructure management by publishing information about proxy services to the registry and categorizes the services for discovery. Thus growing a portfolio of services making it easier to understand and manage relationships among services, component versioning, and dependencies.
- UDDI services can be imported 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 decreasing the development lifecycle 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. You can search on criteria specified by you.

Introduction to UDDI Entities

UDDI uses a specific data model to represent entities that define organizations and services. Figure 5-2 shows the relationship between different UDDI entities.

Figure 5-2 UDDI Entities Representing Organizations and Services


businessEntity: Information about the party who publishes information about a service
businessService: Descriptive information about a particular family of technical services
bindingTemplate: Technical information about a service entry point and construction specifications
tModel: Descriptions of specifications for services or taxonomies. Basis for technical fingerprints
bindingTemplate data contains references to tModels. These references designate the interface specifications for a service.

A high-level overview of the UDDI entities is provided.

Table 5-1 High-Level Description of UDDI Entities

Business Entity	An organization or group of people who own and provide the services. It can be described by a set of names, descriptions, contact details for the service provider, a set of categories that represent the business entity's features, unique identifiers, discovery URLs.
Business Service	A business service represents functionality or resources provided by a business entity. It is described by a name, a description, and a set of categories that represent the function of the service. It is not necessarily a Web service.

Table 5-1 High-Level Description of UDDI Entities

Binding Template	A binding template represents the technical details of how to invoke a business service. A business service can contain one or more binding templates. It is described by an Access Point representing the service endpoint (the endpoint URI and protocol specification), tModel instance information, and categories to reference specific features of the binding template.
`tModel`	This is the technical model 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 2.1 User's Guide*.

Prerequisites

Before using AquaLogic Service Bus with a UDDI registry you must perform the following tasks:

- AquaLogic Service Registry must be installed and running. For information on how to install BEA AquaLogic Service registry 2.1, see in *BEA AquaLogic Service Registry Installation Guide*.
- AquaLogic Service Bus must be installed and running. For information on how to install BEA AquaLogic Service Bus 2.5, see, in *BEA AquaLogic Service Bus Installation Guide*.

Certification

AquaLogic Service Bus works with any UDDI registry that is fully compliant with the version 3 implementation of UDDI (Universal Description, Discovery and Integration).

AquaLogic Service Registry 2.1 is a version 3 UDDI-compliant registry and is certified to work with AquaLogic Service Bus.

Features

The AquaLogic Service Bus Console provides you with access to any version 3 implementation of UDDI registry once it has been set up to work with AquaLogic Service Bus. The following features are available:

- Configure AquaLogic Service Bus to work with one or more version 3 UDDI compliant registries.
- The import feature allows you to search for specific services in a registry or list all services available. You can search on business entity, service name pattern, or both, and then make your selection from the results list.
- Import selected business services from a registry.
- Publish selected AquaLogic Service Bus proxy services to the registry.

For more information on how to configure and search the registry, import business services to AquaLogic Service Bus, and how to publish proxy services to a UDDI registry, see the following topics in System Administration in *Using the AquaLogic Service Bus Console*:

- Configuring a UDDI Registry
- Importing a Business Service from UDDI Registry
- Publishing a Proxy Service to a UDDI Registry

What is the BEA AquaLogic Service Registry?

BEA AquaLogic Service Registry is a compliant fully with version 3 implementation of UDDI and is a key component of a Service Oriented Architecture (SOA).

Note: AquaLogic Service Registry is not provided with AquaLogic Service Bus. In order to use AquaLogic Service Registry you have to buy a separate licence from BEA.

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). Using the Registry Console you can browse and publish registry content. The Registry Console is the primary console for administrators to perform registry management. You can launch the AquaLogic Service Registry console in a Web browser by opening the following URL: `http://`*`hostname:port/uddi/web`*, where hostname and port are defined when AquaLogic Service Registry is installed. The default port is 8080. 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.

Sample Business Scenario for AquaLogic Service Bus and UDDI

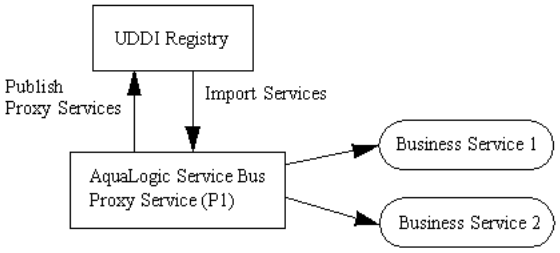
The following are two sample business scenarios that highlight the benefit of using UDDI.

Basic Proxy Service Communication with a UDDI Registry

This scenario describes how you can use AquaLogic Service Bus to import services form a registry and then publish the services back to a registry as part of an AquaLogic Service Bus proxy service.

AquaLogic Service Bus imports business services from a UDDI registry. Proxy services are configured to communicate with the business services in the Message Flow. The proxy services themselves can be published back to the registry and made available for use by other domains.

Figure 5-3 Proxy Service Communication with a UDDI Registry



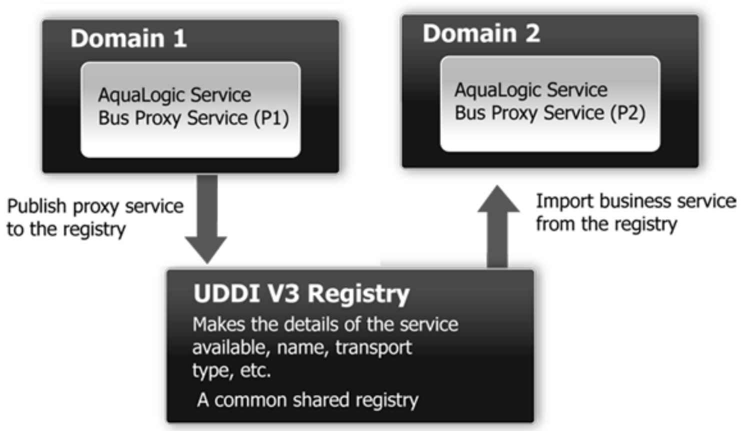
Cross-Domain Deployment in AquaLogic Service Bus

This scenario describes cross-domain deployment using AquaLogic Service Bus. An AquaLogic Service Bus application in one domain requires access to an AquaLogic Service Bus service in another domain at run time.

An instance of AquaLogic Service Bus is deployed in each of two domains. The AquaLogic Service Bus Proxy service (P1) is configured in domain (D1). The AquaLogic Service Bus Proxy service (P2) in domain (D2) requires to access proxy service (P1). As the domains can not communicate directly with each other, P2 in D2 can not discover P1 in D1. The AquaLogic Service Bus import and export feature does not support run-time discovery of services in different domains, but publishing the service to a publicly available UDDI registry allows for the discovery of the 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 services in another AquaLogic Service Bus 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 it needs HTTP access to the domain to import using the URL. In the absence of connectivity an error message will be returned.

Figure 5-4 Sample Business Case of Cross-Domain Deployment


Domain 1
AquaLogic Service Bus Proxy Service (P1)
Domain 2
AquaLogic Service Bus Proxy Service (P2)
Publish proxy service to the registry
Import business service from the registry
UDDI V3 Registry
Makes the details of the service available, name, transport type, etc.
A common shared registry

Using AquaLogic Service Bus and UDDI

You can use the AquaLogic Service Bus Console to:

- Publish information about any proxy service to a registry, including the following service types: WSDL, messaging, any SOAP, and any XML.
- Search a registry for information about a service and discover the service.
- Configure a registry to allow users to publish services and import services.
- Import Web services and integrate them with your application.

UDDI Workflow

The typical workflow for using UDDI with AquaLogic Service Bus is as follows:

- Install AquaLogic Service Bus. For more information on the installation, see *AquaLogic Service Bus Installation Guide*.
- Install AquaLogic Service Registry. For information on installation, see *AquaLogic Service Registry Installation Guide*.
- Configure the registry in the ALSB console. For more information, see “Configuring a UDDI Registry” in System Administration in *Using the AquaLogic Service Bus Console*.

- Set a default registry. For more information 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 AquaLogic Service Bus, and then publish AquaLogic Service Bus proxy services to it or import business services from the registry to be used in a proxy service. You must be in an active AquaLogic Service Bus session in the AquaLogic Service Bus Console to configure the registry.

The following table describes the fields for configuring a UDDI registry. An asterisk denotes a required field.

Table 5-2 UDDI Registry Configuration Settings

Property	Description
Name*	The name of the registry. The name is assigned to it when it is first published. Select the registry name to edit the details for the registry. You can edit the inquiry URL, publish URL, security URL, and the service account, but not the name of the registry.
Inquiry URL*	The URL used to locate and import a service. To read from a registry, you only need to specify a name and inquiry URL.
Publish URL*	The URL used to publish a service. When publishing a service you must also specify a security URL and specify the service account associated with the registry.
Security URL*	The URL used to get an authentication token so that you can publish to the registry. You must specify a publish URL and a security URL if you have a service account defined.
Subscription URL*	The URL used to subscribe to changes from the corresponding service in the registry. You use this URL to synchronize the service in the AquaLogic Service Bus console with the changes in the corresponding service in the registry using Auto-Import.
User Name*	The user name for the registry console. This is required for authentication into the registry console.
Password/(Change Password)	The password for the registry console. This is required for authentication into the registry console.

Table 5-2 UDDI Registry Configuration Settings (Continued)

Property	Description
Load tModels into registry	Loads the tModels into the selected registry. This option only has to be selected once per registry.
Enable Auto-Import	Auto-synchronizes services with the UDDI Registry.

When publishing services to AquaLogic Service Registry, to gain access to the registry, you must be authenticated for which you should have a valid user name and password. The user name and password combination is implemented as a service account resource in AquaLogic Service Bus. Service accounts must be defined before configuring proxy services so that the authentication criterion is set up to work with a service during the configuration of the proxy service.

You can set up registries with multiple user name and passwords allowing different users to have different permissions based on the service account. Permissions in AquaLogic Service Registry are such that administrators can manage users' privileges in BEA AquaLogic Service Registry and create views into the registry, specific to the needs of the different user types. User permissions set in AquaLogic Service Bus govern access to the registries, their content, and the functionality available to you.

Publishing a Proxy Service to a UDDI Registry

You can use the AquaLogic Service Bus Console to publish proxy services to AquaLogic Service Registry. You must have an account set up in AquaLogic Service Registry to do this. You can publish any proxy service to a UDDI registry except proxy services using the local transport. The service types and transports are listed in Table 5-3.

Table 5-3 Service Types and Transports for a Proxy Service

Service Type	Transports
WSDL	HTTP(S), JMS
Any SOAP	HTTP(S), JMS
Any XML	HTTP(S), JMS, E-mail, File, FTP, Tuxedo

Table 5-3 Service Types and Transports for a Proxy Service

Service Type	Transports
Messaging	HTTP(S), JMS, E-mail, File, FTP, 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, 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 (the Business Service Console in the case of AquaLogic Service Registry). The first time you publish to a registry you must load the `tModels` to that registry. This is done at the time you configure the publishing details in the AquaLogic Service Bus Console.

For more information on how to publish to a UDDI registry, see "Publishing a Proxy Service to a UDDI Registry" in System Administration in *Using the AquaLogic Service Bus Console*.

AquaLogic Service Bus works with any UDDI version 3 compliant registry but it has been certified to work with AquaLogic Service Registry only.

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 AquaLogic Service Bus 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.

Using Auto-Publish

When you create a proxy service you can publish it to the default registry automatically. In order to do this you have to first set a default registry to which the proxy services are published when you create or modify them. You can select the check box beside **Publish To Registry** in the **Create a Proxy Service-General Configuration** page to enable or disable the Auto-Publish feature for individual proxy services. For more information on setting up a default registry, see Setting up a Default Registry in *Using the AquaLogic Service Bus Console*.

When you enable the **Publish To Registry** in the **Create a Proxy Service-General Configuration** page the proxy service is published to the default registry. The services are automatically published to the registry when you activate the session, only when the **Publish to Registry** check box is selected for the proxy service. If the Registry is unavailable, the publish is retried in the background. Any further changes to the proxy service resets the retry attempts. When a proxy service is republished to UDDI, 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 are enabled to auto-publish will be published to the new default registry. Synchronization will now take place with the current default registry. When a proxy service is not synchronized, the AquaLogic Service Bus Console console displays this icon beside the proxy service.

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 may now see errors in the **Auto Publish Status** page, if there are any auto-published proxy services. You can correct this by selecting the default registry again.

Importing a Service from a Registry

You can import services from a registry as AquaLogic Service Bus business services. When importing a WSDL-based service, if multiple UDDI binding templates are encountered, AquaLogic Service Bus creates a different business service for each binding template.

To establish access to UDDI registries in AquaLogic Service Bus you must have AquaLogic Service Bus system administration privileges. The registry entries appear on the System Administration > Import from UDDI page of the AquaLogic Service Bus Console. When importing, you make a selection from the list of available registries. To discover a service in a registry you must query a specific registry. Entries in registries are unique. This 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 AquaLogic Service Bus:

- 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, or HTTP(S).
- Services that are categorized as AquaLogic Service Bus services. These are AquaLogic Service Bus proxy services that are published to a UDDI registry. This feature is primarily

used in multi-domain AquaLogic Service Bus 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 AquaLogic Service Bus Console to import services from a UDDI registry, see "Importing a Business Service from a UDDI Registry" in System Administration in Using the AquaLogic Service Bus Console.

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. 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 AquaLogic Service Bus.

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 AquaLogic Service Bus have specific `tmodel` keys identifying the services that are used when searching for the service in the registry.

Import automatically tries to connect to a registry when you attempt to get the list of business entities from 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 systems Administrator.

Related References

- Technical Notes can be found at `http://www.oasis-open.org/committees/uddi-spec/doc/tns.htm`. The note on s *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 The specification defines the following:

  `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm`

- SOAP APIs that applications use to query and to publish information to a UDDI registry
- XML Schema schemata 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. For more information on Using Auto-Import see, Auto-Import in Using the AquaLogic Service Bus Console. You can use Auto-Import to do the following:

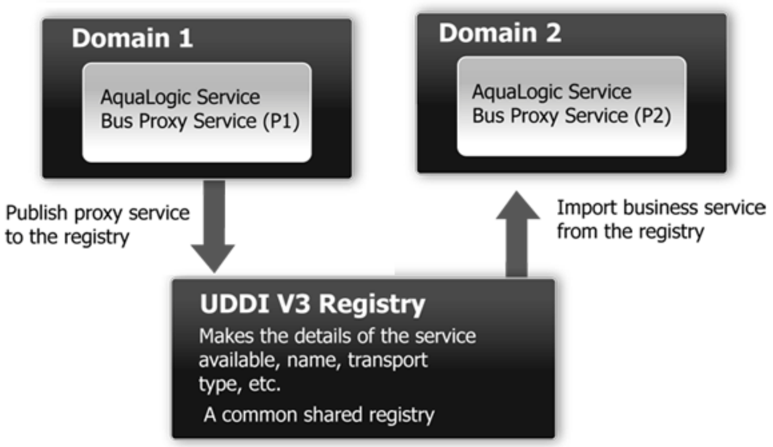
- "Synchronize" on page 5-14
- "Detach" on page 5-15

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 AquaLogic Service Bus 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, AquaLogic Service Bus automatically subscribes to any changes to the service in the registry. If the service changes, the icon in the resource browser and project explorer indicates the service needs to be synchronized. In addition, the **Auto Import Status** page shows this service and provides the options to synchronize the service or detach it from the registry. Under certain circumstances, synchronizing the service might result in semantic validation errors that shows up in the view conflicts page. These will have to be fixed manually fixed before the session is activated.

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.

Figure 5-5 Sample Business Case of Cross-Domain Deployment


Domain 1
AquaLogic Service Bus Proxy Service (P1)
Domain 2
AquaLogic Service Bus Proxy Service (P2)
Publish proxy service to the registry
Import business service from the registry
UDDI V3 Registry
Makes the details of the service available, name, transport type, etc.
A common shared registry

Consider a scenario where you publish services from Domain1(see Figure 5-5) to a registry. You then import these services to another domain, Domain2. When you make changes to the corresponding service in Domain1 and update it in the registry. You can update the services in Domain2 by synchronizing it with the registry using Auto-Import.

Detach

When you do not want the service in the AquaLogic Service Bus Console synchronized with the corresponding service in the registry then, you can avoid synchronization by detaching it from the registry. For more information on using Detach, see “Detaching a Service” in System Administration in *Using the AquaLogic Service Bus Console*.

Auto-Synchronization of Services With UDDI

You can keep the service definitions in AquaLogic Service Bus automatically synchronized (both ways) with those in UDDI.

Services can be automatically published to a UDDI registry after they are created or changed within AquaLogic Service Bus 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 AquaLogic Service Bus Console 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 selected will be kept in synchrony with the UDDI Registry automatically. If there is any failure during auto-synchronization, it will be reported on the Auto-Import Status page where you can update it manually. See "Configuring a UDDI Registry" in System Administration in the *Using the AquaLogic Service Bus Console*.

Mapping AquaLogic Service Bus Proxy Services to UDDI Entities

AquaLogic Service Bus 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 AquaLogic Service Bus proxy service.

Table 5-4 Proxy Service Attributes and Service Types

Service Type	Message Content Type	Transports
WSDL	SOAP or XML (with attachment)	HTTP(S), JMS
Any SOAP	Untyped SOAP (with attachment)	HTTP(S), JMS
Any XML	Untyped XML (with attachment)	HTTP(S), JMS, E-mail, File, FTP, and Tuxedo
Messaging	Binary, Text, MFL, XML (schema)	HTTP(S), JMS, E-mail, File, FTP, and 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, 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.

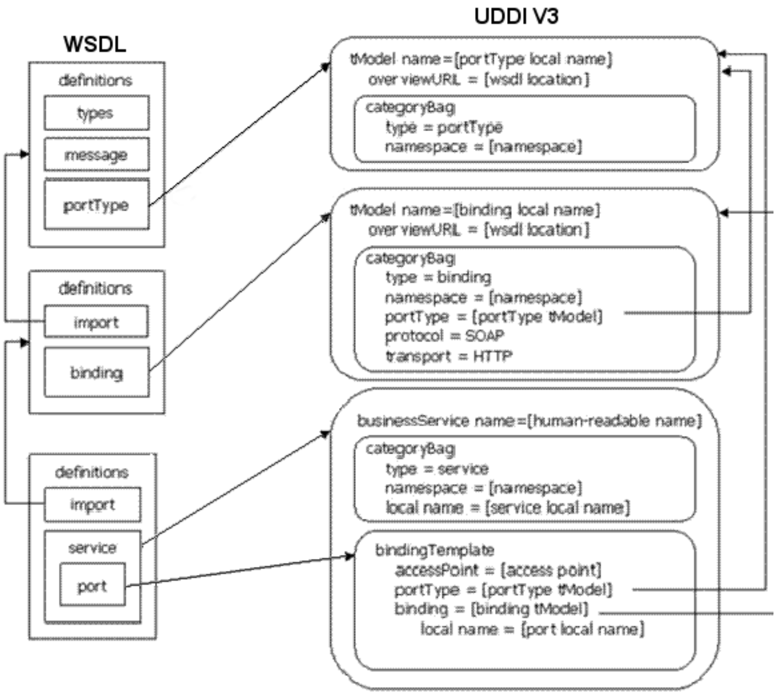
The primary relevant entities in UDDI are:

- businessService: this represents the service as a whole and contains high-level general information about the service.

- bindingTemplate: this contains information for accessing the service.
- `tModels`: tModels are used to supply the individual attributes for categorizing and defining the service.

Figure 5-6 shows how WSDL-based services are mapped to UDDI business entities.

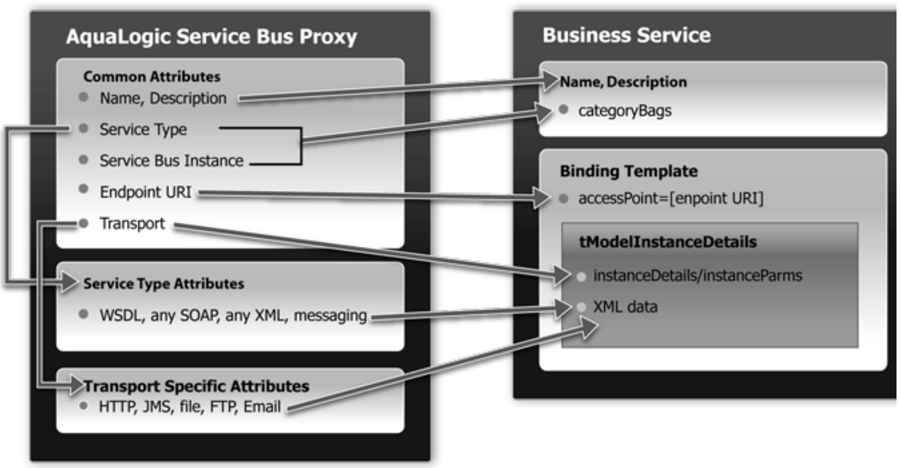
Figure 5-6 WSDL Service to UDDI Mapping


WSDL
definitions
types
message
portType
definitions
import
binding
definitions
import
service
port
UDDI V3
tModel name=[portType local name]
overviewURL = [wsdl location]
categoryBag
type = portType
namespace = [namespace]
tModel name=[binding local name]
overviewURL = [wsdl location]
categoryBag
type = binding
namespace = [namespace]
portType = [portType tModel]
protocol = SOAP
transport = HTTP
businessService name=[human-readable name]
categoryBag
type = service
namespace = [namespace]
local name = [service local name]
bindingTemplate
accessPoint = [access point]
portType = [portType tModel]
binding = [binding tModel]
local name = [port local name]

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 AquaLogic Service Bus proxy services as entities in the UDDI registry, you must add additional canonical tModels to support some of the constructs specific AquaLogic Service Bus. Not all attributes of an AquaLogic Service Bus 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 AquaLogic Service Bus maps to UDDI is shown in Figure 5-7.

Figure 5-7 AquaLogic Service Bus to UDDI Mapping



UDDI Mapping Details for an AquaLogic Service Bus Proxy Service

AquaLogic Service Bus high-level proxy service information maps into the Business Service as follows:

- Name and Description map to `businessService elements`.
- There is a Special keyed Reference Group for AquaLogic Service Bus properties. An example of a key is `uddi:bea.com:attributes:aqualogicservicebus`.
- AquaLogic Service Bus 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 AquaLogic Service Bus Instance maps to a `keyedReference` in the AquaLogic Service Bus `keyedReferenceGroup` (Name = “`AquaLogicServiceBus`”, Values = URL of the AquaLogic Service Bus instance).

This instance serves two purposes:

- To indicate that this service is in fact hosted by an AquaLogic Service Bus server.
- To contain the URL of the AquaLogic Service Bus instance.

Listing 5-1 shows a mapping of high-level proxy service information to a business service.

Listing 5-1 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 AquaLogic Service Bus 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 AquaLogic Service Bus, 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.`

AquaLogic Service Bus 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 AquaLogic Service Bus to support JMS and file transports.
  - Detailed AquaLogic Service Bus 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, Messaging. New `tModels` are packaged with AquaLogic Service Bus to support anySOAP, anyXML, and Messaging.
  - WSDL maps via WSDL to UDDI technology note.
  - Messaging has detailed configuration information that maps to `InstanceParms`.

Listing 5-2 shows a detailed information mapping to the binding template.

Listing 5-2 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 5-4, "Proxy Service Attributes and Service Types," on page 5-16.) This metadata provides 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 detailed configuration details, including the required client authorization and the request and response character encoding, the following Listing 5-3 provides an example of what must appear in the bindingTemplate `tModelInstanceDetails`.

Listing 5-3 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's `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 AquaLogic Service Bus 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 5-5 is organized by transport type and lists the `tModelKey` and `instanceParms` used by each of the transports.

Table 5-5 Transport Attributes

Transport	tModelKey	InstanceParms
HTTP	`uddi:uddi.org:transport:http`	• Client Authentication [None, Basic, Client Cert (HTTP(S) 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 5-5 Transport Attributes

Transport	tModelKey	InstanceParms
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
E-mail[1]	`uddi:uddi.org:transport:smtp`	• Attachment supported [Boolean] • Request Encoding
Tuxedo	`uddi:bea.org:transport:tuxedo`	• Response required • Access point ID • Buffer type • Buffer subtype • Classes jar • Field table classes • View classes

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 AquaLogic Service Bus, 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 5-6 provides a high-level description of each of the service types.

Table 5-6 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's `tModelInstanceDetails`, as well as in the `categoryBag`, defines the Any Soap attributes.
Any XML	A simple marker protocol `tModel` within the bindingTemplate's `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's `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 provide details 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 AquaLogic Service Bus (optional, if input message is XML) • URL of input message MFL in AquaLogic Service Bus (if input message is MFL) • Output message format (none, XML, Text, Binary, MFL) • URL of output message Schema in AquaLogic Service Bus (optional, if output message is XML) • URL of output message MFL in AquaLogic Service Bus (if output message is MFL)

Canonical tModels Supporting AquaLogic Service Bus Services

The AquaLogic Service Bus-UDDI mapping introduces a number of new canonical `tModels` that are used to represent AquaLogic Service Bus 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.

The following table provides a summary of the new `tModels`.

Table 5-7 AquaLogic Service Bus tModels

Name	Value	Description
CategorizationGroup tModel Types		
`bea-com:servicebus:properties`		Describes very specific attributes of an AquaLogic Service Bus 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 AquaLogic Service Bus service.
`bea-com:servicebus:instance`	URL of AquaLogic Service Bus Console	Describes the service instance in AquaLogic Service Bus 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 5-7 AquaLogic Service Bus 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

The following 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 5-4 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>
```

EJB Transport

Using the EJB Transport, AquaLogic Service Bus supports native RMI invocation of Stateless Session Beans deployed on WebLogic Server 8.1, 9.0, 9.1, or 9.2. It allows transactional and secure communications. The EJB transport can also be leveraged to expose an EJB as a Web service through AquaLogic Service Bus.

This section includes the following topics:

Introduction

You can design business services in AquaLogic Service Bus to use the EJB transport. The EJB transport is fully integrated into the AquaLogic Service Bus configuration, management, monitoring, and test consoles. Business services built with the EJB transport can be used for Publish, Service Callout, and service invocations. You cannot create proxy services that use the EJB transport.

An EJB can be exposed as a Web service, without the need for tools or the modification of the legacy code on the application server that hosts the EJB.

The EJB transport provides the following capabilities:

Transactional Integrity

You can call EJB Business service in the context of a global transaction. The EJB Transport can also suspend or start a global transaction before invoking an EJB.

Security Propagation

The security context established at the beginning of a message flow, from an AquaLogic Service Bus client is propagated to the other system. In other words, an incoming SOAP over HTTP request to AquaLogic Service Bus that requires authentication is authenticated by AquaLogic Service Bus and the authenticated subject can then be propagated to the EJB server.

HTTP Tunneling and Encrypted Communication

You can access EJBs that are behind a fire wall with HTTP tunneling. For additional security, you can use SSL to encrypt all of the communications with the EJB Server.

JNDI Provider

EJB transport leverages the JNDI provider—an AquaLogic Service Bus resource. The JNDI provider defines communication protocols and security credentials for accessing remote servers. A JNDI provider can be reused by multiple EJB business services. This provides a centralized way for administrators to manage remote EJB server configurations.

For information about JNDI provider resources, see System Administration in *Using the AquaLogic Service Bus Console*.

High Performance Caching

The EJB transport is built on high performance cache. This allows the reuse of established connections and minimizes EJB stubs lookups.

Failover and Load Balancing

The EJB transport can take advantage of scenarios in which the same EJB is deployed in multiple domains or on a cluster for load balancing or failover or both.

Advanced XML to Java Binding Capabilities

The EJB transport leverages the WebLogic Server JAX-RPC stack to perform Java to XML bindings. The JAX-RPC stack is a high performance engine that supports advanced Java objects such as XML Beans. If the Java type is not recognized by the stack, an extension mechanism is provided to facilitate support of these Java types. For information about this extension mechanism (using the converter classes), see Supported Types and Converter Class.

Intelligent Retries

The EJB transport makes retry decisions based on the nature of the failure that can occur during the invocation of an EJB.

Invoking EJBs from AquaLogic Service Bus

Before you can configure a business service in AquaLogic Service Bus, you must register a JNDI provider resource and a client JAR resource. This section describes how to design and configure an EJB transport business service in AquaLogic Service Bus. It includes the following topics:

- "Register a JNDI Provider Resource" on page 6-3
- "Register an EJB Client JAR Resource" on page 6-4
- "Create an EJB Business Service" on page 6-5
- "Invoking EJB Business Services" on page 6-11

Register a JNDI Provider Resource

A JNDI Provider resource allows you to specify the communication protocols and security credentials used to retrieve EJB stubs bound in the JNDI tree of remote WebLogic 8.1 or 9.x domains. (For more information how to setup a JNDI tree, see Programming WebLogic JNDI in the BEA WebLogic Server documentation.)

Typically, the target EJB is not located in the same domain as AquaLogic Service Bus. In this case, you must register a JNDI Provider resource. When the EJB is located in the same domain, you can define a provider to specify credentials and take advantage of stubs caching, although it is optional in this case.

The JNDI provider has a high performance caching mechanism for remote connections and EJB stubs. The preferred communication protocol from AquaLogic Service Bus to a WebLogic Server domain is `t3` or `t3s`. If messages need to go through a fire wall, you can use HTTP tunneling. For more information about HTTP tunneling, see HTTP Tunneling and Encrypted Communication.

Notes:

- Although it is possible to use a WebLogic Server foreign JNDI Provider, BEA recommends that you do not.
- 2-way SSL is not supported.

- EJB transport provider does not support CLIENT CERT to look-up JNDI tree or access a method on an EJB.

Adding a JNDI Provider

For information about registering and configuring a JNDI provider resource in AquaLogic Service Bus, see "Adding a JNDI Provider" in System Administration in *Using the AquaLogic Service Bus Console*.

Register an EJB Client JAR Resource

A client JAR must be registered as a resource in AquaLogic Service Bus. It is therefore part of the AquaLogic Service Bus configuration and can be exported from and imported into a project.

An EJB client JAR file must contain the interfaces and classes needed by AquaLogic Service Bus to access an EJB. This includes the remote and home interfaces and any dependent types to which the client is exposed, such as method parameter types or application exceptions.

If your business service requires converter classes, you can register a JAR file containing the converter classes as an AquaLogic Service Bus resource and subsequently use these classes to help map parameter and return value types to Java classes that can be mapped to XML. Alternatively, you can package these converter classes in the EJB client JAR. For information about converter classes, see Converter Classes.

Consider the following guidelines when using EJB client JARs:

- Adding Home and remote interfaces in the system classpath is bad practice and is not supported by AquaLogic Service Bus.
- BEA recommends that you keep the client JAR size small, include a single home interface per JAR and not register the entire ejb-jar file.
- You can use WebLogic Workshop to obtain a client JAR for EJBs deployed on WebLogic Server 8.1 or 9.x.
- Client-jars compiled with JDK 1.4 or later are supported.

Adding a Client or Converter JAR

For information about registering and configuring a JAR resource in AquaLogic Service Bus, see "Adding a JAR" in JARs in *Using the AquaLogic Service Bus Console.*

Create a Service Account (Optional)

If the EJB methods are protected, you can specify the credentials you want to use for the invocations. Those credentials are often different than the credentials used by the JNDI provider. For information about adding and using service accounts, see Service Accounts in *Using the AquaLogic Service Bus Console*.

Locate an EJB in the JNDI Tree

If you do not know the JNDI name for an EJB, you can browse the EJB Server JNDI tree. For information about browsing the JNDI tree using the WebLogic Server Administration Console, see:

- JNDI in the *WebLogic Server 8.1 Administration Console Online Help* (for WebLogic Server 8.1)
- View objects in the JNDI tree in the *WebLogic Server 9.2 Administration Console Online Help* (for WebLogic Server 9.x)

Create an EJB Business Service

This section provides information about creating a business service that uses the EJB transport. It includes the following topics:

- "General Configuration" on page 6-5
- "EJB Transport-Specific Configuration" on page 6-7
- "EJB Business Service Interface Configuration" on page 6-9

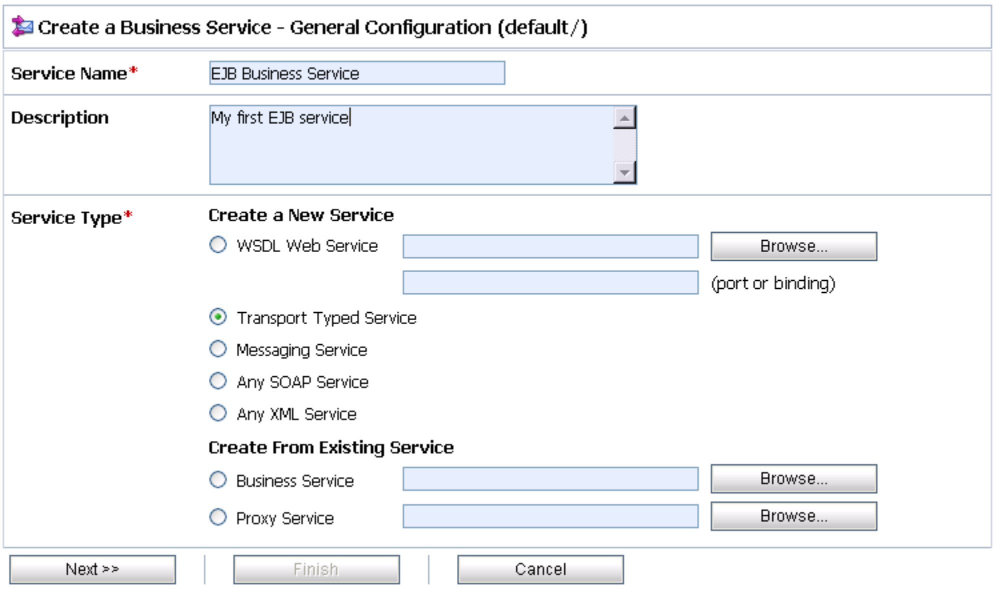
General Configuration

1. Open the AquaLogic Service Bus Console and in an active session, select Project Explorer from the left navigation panel. The **Project View** page is displayed.
2. Select the project in which you want to create the business service. A page in which you can create a business service is displayed—create a new business service.
3. On the General Configuration page, as shown in the following figure, enter a name for the business service and select the **Transport Typed Service** as the **Service Type**.

   An EJB business service is a *Transport Typed Service*, meaning that the type of the transport is determined by the configuration of the service. The EJB transport is currently the only such transport type. You can add other transports by using the AquaLogic Service Bus Transport SDK.

An entry in the **Description** field is optional.

Figure 6-1 Create a Business Service - General Configuration



Create a Business Service - General Configuration (default/)

Service Name*: EJB Business Service

Description: My first EJB service

Service Type*:

Create a New Service

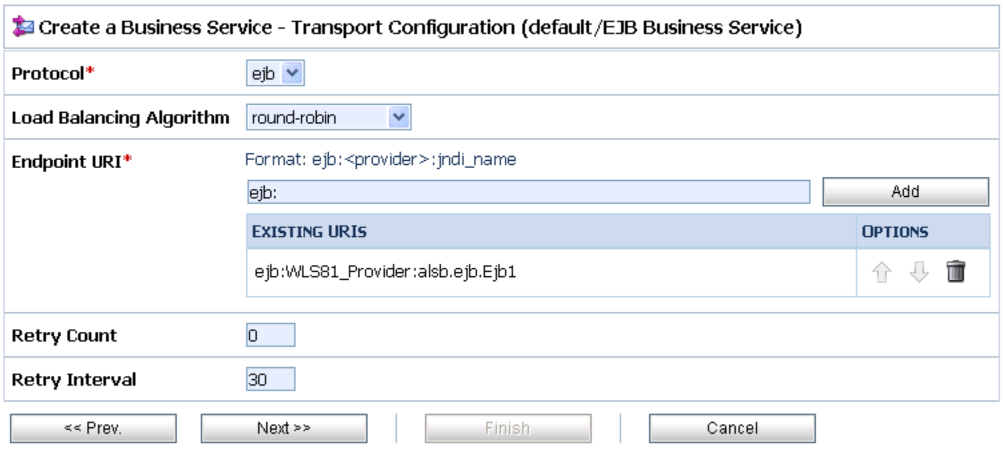
- ( ) WSDL Web Service
- (•) Transport Typed Service
- ( ) Messaging Service
- ( ) Any SOAP Service
- ( ) Any XML Service

Create From Existing Service

- ( ) Business Service
- ( ) Proxy Service

4. Click **Next** to open the transport-specific configuration page.
5. In the Transport Configuration page, select **ejb** as the **Protocol**.

Figure 6-2 Create a Business Service—Transport Configuration



Create a Business Service - Transport Configuration (default/EJB Business Service)

Protocol*	ejb
Load Balancing Algorithm	round-robin
Endpoint URI*	Format: ejb:<provider>:jndi_name ejb:
Retry Count	0
Retry Interval	30

EXISTING URIS	OPTIONS
ejb:WLS81_Provider:alsb.ejb.Ejb1	

6. Enter the Endpoint URI and add it to the list of EXISTING URIs. To build the URI you need the name of the provider you used in "Adding a JNDI Provider" on page 6-4 and the location of the EJB home interface in the JNDI tree you determined in "Locate an EJB in the JNDI Tree" on page 6-5:

   `ejb:`*`provider`*`:`*`jndi_name`*

   If the EJB is deployed locally, you need not provide a JNDI provider name. In this case, the URI format is:

   `ejb::`*`jndi_name`*

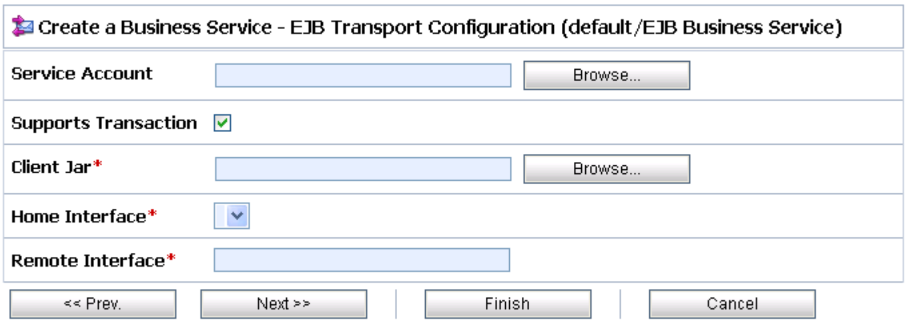
7. As for any AquaLogic Service Bus transport, you can also specify the Load Balancing Algorithm, Retry count, Retry Interval and specify multiple URIs for failover. See "Retries and Failover" on page 6-13.

8. Click **Next** to open the EJB transport-specific configuration page.

EJB Transport-Specific Configuration

After completing the general configuration of the business service you specify EJB transport-specific information such as the Home and Remote interfaces. The EJB Transport Configuration page in the AquaLogic Service Bus Console is shown in the following figure.

Figure 6-3 Create a Business Service— EJB Transport Configuration


Create a Business Service - EJB Transport Configuration (default/EJB Business Service)

Service Account	
Supports Transaction	✓
Client Jar*	
Home Interface*	
Remote Interface*	

To Configure the EJB Transport

1. Optionally select a Service Account.

   If the EJB methods are protected and you defined a service account as described in "Create a Service Account (Optional)" on page 6-5, click **Browse** to locate the appropriate Service Account.

2. By default, the **Supports Transaction** option is selected. This specifies that the EJB supports transaction. If you do not want to propagate transactions, or if the EJB does not support transactions, deselect **Supports Transaction**.

   For information about transaction processing with the EJB Transport, see "Transaction Processing, Retries, and Errors Handling" on page 6-12.

3. Select the **Client JAR**—browse and select the Client JAR you registered previously, as described in "Adding a Client or Converter JAR" on page 6-4.

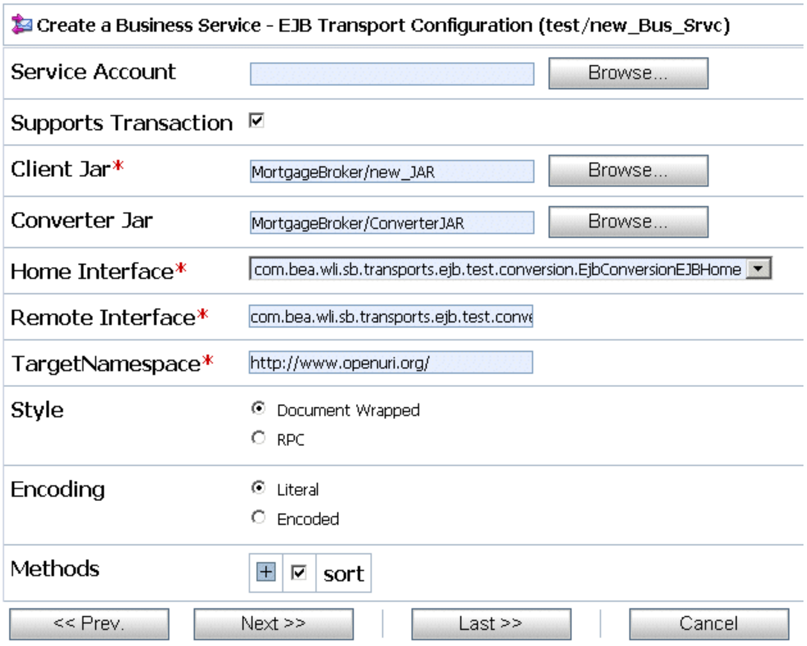
   When you select a Client JAR, a list of its Home Interface is displayed on this page. Additionally, a **Converter JAR** field is displayed.

4. If required, select the **Converter JAR**—browse and select the Converter JAR you registered previously, as described in "Adding a Client or Converter JAR" on page 6-4.

5. Select the Home Interface from the list of interfaces provided in the **Home Interface** field.

   Notice that the Remote Interface is automatically deduced from the Home Interface and the configuration page is refreshed. The **Remote Interface** field is populated and other options are provided that allow you to control the interface of the service and the WSDL generated when you finish configuration of this business service.

Figure 6-4 Create a Business Service— EJB Transport Configuration after Selecting the Home Interface



Create a Business Service - EJB Transport Configuration (test/new_Bus_Srvc)

Service Account	
Supports Transaction	☑
Client Jar*	MortgageBroker/new_JAR
Converter Jar	MortgageBroker/ConverterJAR
Home Interface*	com.bea.wli.sb.transports.ejb.test.conversion.EjbConversionEJBHome
Remote Interface*	com.bea.wli.sb.transports.ejb.test.conve
TargetNamespace*	http://www.openuri.org/
Style	[✓ Document Wrapped] RPC
Encoding	[✓ Literal] Encoded
Methods	☑ sort

EJB Business Service Interface Configuration

An EJB business service is a Transport Typed Service, which means the type of the transport is determined by the configuration of the service.

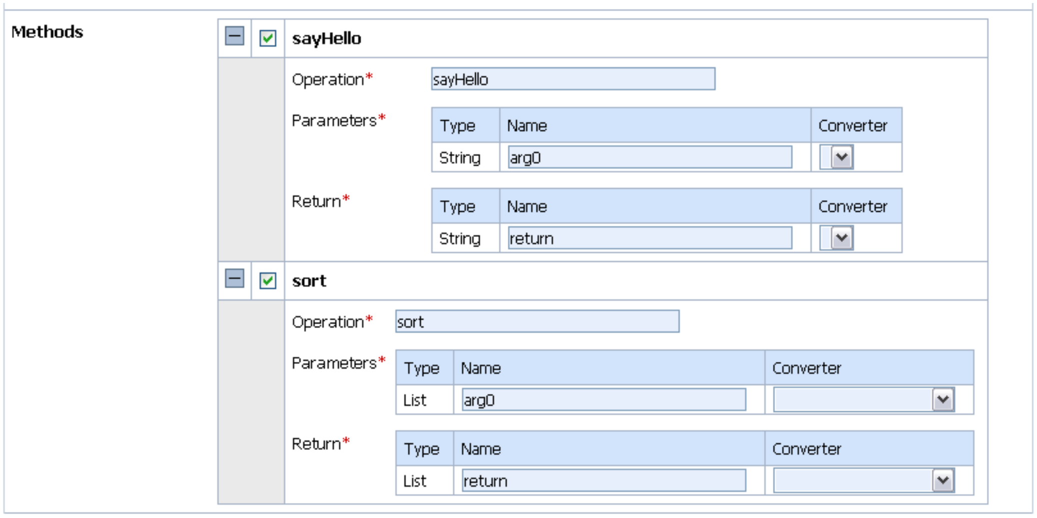
The type of an EJB business service is equivalent to a SOAP XML service—in other words, you can use an EJB business service like any other SOAP XML business service. A WSDL is generated when you save the EJB Transport Configuration.

The WSDL is generated based on the interface of the EJB. The EJB transport configuration page provides configuration options for you to control the interface of the service and the WSDL that is generated. To do so, complete the configuration on the EJB Transport Configuration page as shown in the preceding figure:

1. **TargetNamespace**—Specify the target namespace of the WSDL.
2. **Style**—You can select Document Wrapped or RPC.
3. **Encoding**—Select Literal or Encoded.

4. The methods displayed are those of the EJB Remote Interface you selected. For example, the following figure displays two methods: `sayHello` and `sort`.

Figure 6-5 Create a Business Service— EJB Transport Configuration, Expanded Methods Configuration



Methods

☑ **sayHello**

Operation* sayHello

Parameters*

Type	Name	Converter
String	arg0	

Return*

Type	Name	Converter
String	return	

☑ **sort**

Operation* sort

Parameters*

Type	Name	Converter
List	arg0	

Return*

Type	Name	Converter
List	return	

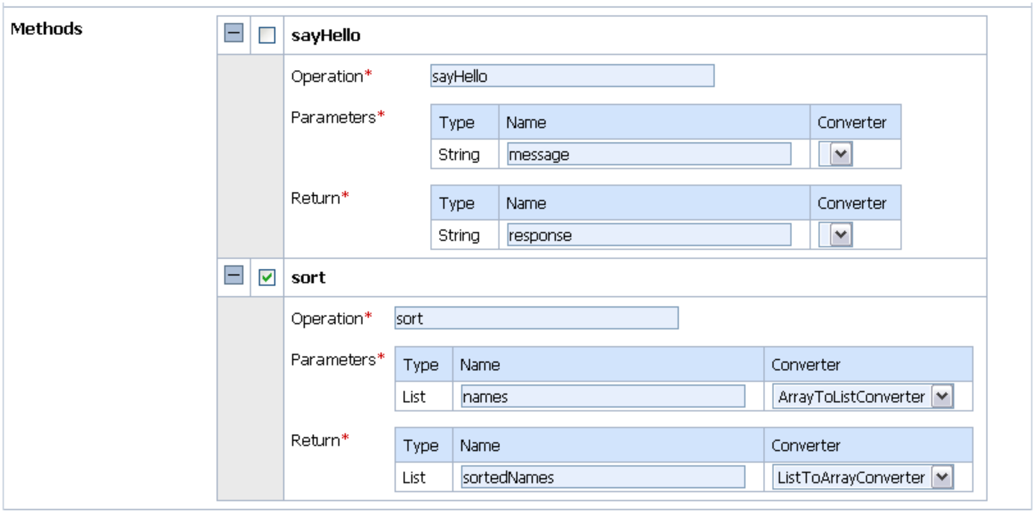
5. You can exclude the methods you do not want to expose by unchecking the check box associated with the method names.

6. You can change the default operation name for a given method. (By default, the operation name is the method name.) If an EJB contains methods with same name, you must change the operation names so that they are unique—WSDLs require unique operation names.

7. You must exclude the methods with parameters or return types that are not supported by the JAX-RPC stack or you must associate such arguments with "Converter Classes" on page 6-15.

   The following figure shows an example of a custom methods configurations.

Figure 6-6 Create a Business Service— EJB Transport Configuration, Customized Methods Configuration



Methods

☐ **sayHello**

Operation* sayHello

Parameters*

Type	Name	Converter
String	message	

Return*

Type	Name	Converter
String	response	

✓ **sort**

Operation* sort

Parameters*

Type	Name	Converter
List	names	ArrayToListConverter

Return*

Type	Name	Converter
List	sortedNames	ListToArrayConverter

8. Click **Next**. Save the service and activate the session.

Note: If the credentials or transaction settings are different between the methods for a given EJB, you can leverage the ability to customize the methods for a given business service, and create a business service per method. This gives you fine-grained control over transactions and credentials.

Invoking EJB Business Services

An EJB business service can be used as a SOAP XML business service. You can publish to, route to, or callout to an EJB business service. If you need transaction support, set the QoS to *Exactly-Once*. See "Transaction Processing, Retries, and Errors Handling" on page 6-12.

You can also use the test console to validate your configuration and to help you to determine the shape of the XML request.

Exposing EJBs as Web Services

You can leverage the EJB transport to easily expose EJBs as Web Services.

Note: You cannot create a proxy service from an existing EJB business service—you must first get the WSDL generated from the EJB business service, and then create the proxy service based on that WSDL. To do so, complete the following steps:

1. Create an EJB business service pointing to the EJB you want to expose, as described in "Create an EJB Business Service" on page 6-5.
2. From the service details page on the AquaLogic Service Bus Console, get the WSDL for the EJB business service.

   The WSDL is contained in a JAR file. You can obtain the WSDL only if there is no pending session.
3. Extract the WSDL from the JAR and register it as a WSDL resource. For information about creating WSDL resources, see WSDLs in *Using the AquaLogic Service Bus Console*.

   If the configuration of the business service changes, a new WSDL is generated. If that happens, you must get the new WSDL and re-register it as a WSDL resource.
4. Create a SOAP XML proxy service based on the WSDL.
5. Edit the proxy service pipeline and route to the EJB business service.

You can now invoke the EJB as a Web Service with no need for purchasing an expensive Web Service toolkit or carrying out intrusive actions on the EJB server.

Advanced Topics

This section includes information about EJB transport that will help you understand how EJB business services behave at run time depending on how they are configured at design time. It includes the following topics:

- "Transaction Processing, Retries, and Errors Handling" on page 6-12
- "Supported Types and Converter Class" on page 6-15

Transaction Processing, Retries, and Errors Handling

Transactions

The EJB transport can create, suspend, and propagate transactions. The transaction between AquaLogic Service Bus and the EJB server are XA transactions. If you use transactions with HTTP tunneling or have a dedicated communication channel and the EJBs are deployed on 8.1 servers, you must set the *security interoperability* mode for the transaction manager to `performance`. For information about setting the security interoperability mode and other transaction configurations, see Configuring Transactions in *Programming WebLogic JTA*.

For the deployment descriptors to be set appropriately for XA capable resources (JMS, TUXEDO, EJB), you must set the XA attribute on the referenced connection factory before creating a proxy service.

To determine the behavior of the EJB business service, considerations include whether the proxy service pipeline has a transactional context, and what qualities of service (QoS) settings are specified in the pipeline when invoking the service:

QoS Best-Effort

If *Best Effort* QoS is specified in the pipeline, no transaction is propagated to the EJB—any ongoing transaction is suspended before invocation, and resumed after invocation.

QoS Exactly-Once

If *Exactly Once* QoS is specified in the pipeline, and

If the EJB does not support transactions (that is, if the **Supports Transaction** option on the EJB transport configuration page is unchecked), no transaction is propagated to the EJB. As in the case of *Best Effort*, any ongoing transaction is suspended before invocation and resumed afterwards.

or

If the EJB supports transactions (that is, if the **Supports Transaction** option on the EJB transport configuration page is checked), the EJB is invoked in the context of a transaction—any ongoing transaction is propagated to the EJB. If no transaction is present, a transaction is created before invocation and committed afterwards.

For more information about QoS in AquaLogic Service Bus services, see "Quality of Service" on page 2-70.

Retries and Failover

Assuming that the EJB business service is configured for retries or failovers, the EJB transport distinguishes the following types of exceptions:

- Runtime Exceptions or Remote Exceptions—typically unexpected fatal errors or communication exceptions
- Exception raised by the JAX-RPC engine—exceptions that occur during the XML to Java conversion
- EJB Checked Exceptions—exceptions declared in the EJB method signature specific to the EJB implementation; also called Business Exceptions

Retries and failover are based on the type of errors and also in the QoS:

QoS Best-Effort

If a run-time or remote exception is thrown, the EJB transport attempts retries or failovers.

If an exception occurs in the JAX-RPC engine, an error is raised to the pipeline and no retries or failover attempts are made.

If an EJB Checked Exception is thrown, an error is raised to the pipeline and no retries or failover attempts are made.

QoS Exactly-Once

If a run-time or remote exception is thrown and the ongoing transaction has been set as *rollback only* (likely by the EJB container), it means the EJB container has been reached and a fatal error either occurred within the EJB container or the EJB. In this case, no retries or failover attempts are made and an error is raised to the pipeline.

If a runtime or remote exception is thrown but the ongoing transaction has not been set as *rollback only,* it means an error occurred before the invocation of the EJB container and the EJB transport will attempt retries or failovers. Note that in this case, the EJB transport still respects the e*xactly-once* semantic.

If an exception occurs in the JAX-RPC engine, the EJB transport sets the ongoing transaction to *rollback only* and an error is raised to the pipeline; no retries or failover attempts are made.

If an EJB Checked Exception is thrown, an error is raised to the pipeline and no retries or failover attempts are made.

See "Transactions" on page 6-12 for other repercussions of QoS specifications for an EJB business service.

Error Handling

When throwing a checked exception, according to the EJB specifications, the ongoing transaction can be specified as *rollback only*.

If the ongoing transaction is set as *rollback only* by the EJB developer, the transaction is eventually rolled back by its creator (most likely the proxy service).

If the ongoing transaction is not set to *rollback only*, and a checked exception is raised, it is important to catch EJB checked exceptions in the pipeline with an error handler. If those exceptions are not caught, the pipeline errors are propagated back to the proxy service. The proxy service, in turn, is likely to rollback the ongoing transaction (depending of the transport implementation)—this may not be the intended result.

For example, assume you have an EJB with the following method:

```
public void withdrawFunds(float amount) throws RemoteException,
InsufficientFundsException {…}
```

Also assume that when an `InsufficientFundsException` exception is thrown, the EJB does not set the current transaction as *rollback only*. In most scenarios, it is wrong to allow the proxy service to roll back the transaction—you may need to configure an error handler in the pipeline to catch the error and avoid this scenario.

Supported Types and Converter Class

The EJB transport is responsible for the XML←→Java conversion. The conversion is performed by the WebLogic Server JAX-RPC engine.

The EJB transport natively supports the following types:

- Primitive types
- XmlObject (both Apache and BEA versions)
- Schema generated XMLBeans (both Apache and BEA versions)
- JavaBean classes

For the full list of natively supported types, see Data Types and Data Binding in *Programming Web Services for WebLogic Server*.

An EJB method can use parameters/return types that are either not supported by the JAX-RPC engine (an error is reported at design time), or that do not map directly to XML (errors occur at run time). The most commonly used unsupported types are:

- “Object”, “Object[]”
- Java Collections as they are not strongly-typed (for example, List, Set)
- Java classes that do not follow the JavaBean pattern (for example, Map)

You can write a custom converter class than converts those types into types more suitable for XML←→Java conversions. The EJB transport supports custom converter classes.

Converter Classes

A Converter class is a Java class that implements and conforms to the contract defined by the `com.bea.wli.sb.transports.ejb.ITypeConverter` Java interface of the AquaLogic Service Bus public API. For information about the `ITypeConverter` Java interface and other AquaLogic Service Bus APIs, see the AquaLogic Service Bus Javadoc.

To use a converter class for an EJB business service, you must:

1. Create a converter class by implementing and compiling the interface.
2. Add the converter class to the client JAR or to a converter class JAR file (See "Adding a Client or Converter JAR" on page 6-4).
3. When customizing the method configuration during the creation of an EJB business service, navigate to one of the parameter/return types and select the desired converter. See step 7 in "EJB Business Service Interface Configuration" on page 6-9—the AquaLogic Service Bus Console displays a list of the converters available that can be applied to a particular parameter/return type.

Troubleshooting

The information in this section is provided to help you troubleshoot problems when designing or running an EJB business service.

Enabling Debug Mode

The EJB transport uses the same logger as other AquaLogic Service Bus transports. To enable the debug mode, before starting the server, edit the `wlidebug.xml` file in the domain directory and set the category **wli-sb-transports-debug** to `true`. For more information about the `wlidebug.xml` file and the debug flags, see Appendix B, "Debugging AquaLogic Service Bus."

Temp Directories

During design time, the EJB transport generates files in the subfolder **alsbejbtransport** and subfolders prefixed with **appcgen_** in the `temp` directory. It is safe to delete those folders and files, and sometimes may be useful to check them to determine what went wrong during activation.

Deployed Application

When an EJB business service is created an application is deployed on the AquaLogic Server. You can use the WebLogic Server Administration Console to monitor and tune this application. The name of EJB business service applications is prepended with `ALSB EJB`, which is followed by the WSDL type and an auto generated suffix.

Errors

The following items may help in the event that you need to troubleshoot a problem with an EJB business service:

- The following error when creating a business service is due to a Windows operating system limitation—paths containing more than 255 characters are not supported:

  ```
  The system cannot find the path specified):Probably the string length of
  the path of the file being extracted was too long
  ```

  You can try to reduce the path length by creating a shorter path to the AquaLogic Service Bus domain, or you can use the following option to override the WebLogic Server `temp` directory when starting the server:

  ```
  -Dweblogic.j2ee.application.tmpDir=$desired_short_dir
  ```

- If you get an XML marshalling error when invoking an EJB business service and you believe the request to be valid against the service WSDL, you probably need to write a converter class. For information, see "Converter Classes" on page 6-15.

- If the EJB interfaces and stubs are changed on the remote server, the first time you try to invoke the new EJB, an error is thrown. Those changes on the remote server are not visible to AquaLogic Service Bus—it tries to invoke the cached EJB stubs, which are no longer valid. However, when the invocation error occurs, the transport assumes that those stubs are now invalid, and remove them from the cache—in this way, the error is prevented on subsequent attempts to invoke the EJB. To avoid this first-time error, you can reset the JNDI Provider in the AquaLogic Service Bus Console.

- For HTTP tunneling between WebLogic Server 9.2 and WebLogic Server 8.1 to work, you must set the `t3-server-abbrev-table-size` element to `255` in the `config.xml` file in the AquaLogic Service Bus domain, as shown in the following code snippet:

  ```
  <server>
      <name>AdminServer</name>
      <ssl>
        <name>AdminServer</name>
        <enabled>true</enabled>
      </ssl>
          <t3-server-abbrev-table-size>255</t3-server-abbrev-table-size>
      <listen-address></listen-address>
    </server>
  ```

Transports

You can configure BEA AquaLogic Service Bus proxy services and business services to use one of the following transport protocols. The transport protocol you select depends on the service type, the type of authentication required, the service type of the invoking service, and so on. This section describes the transport protocols supported by AquaLogic Service Bus. They include:

E-mail

You can select the e-mail transport protocol when you configure a Messaging Type or Any XML Service type of proxy service or business service. The following sections describe

- "Configuring Proxy Services using E-mail Transport Protocol" on page 7-2
- "Configuring Business Services using E-mail Transport Protocol" on page 7-3

Configuring Proxy Services using E-mail Transport Protocol

When you configure a proxy service using the e-mail transport protocol, you must specify an endpoint URI in the following format:

```
mailfrom:<mailserver-host:port>
```

where

- `mailserver-host`: is the name of the host mail server
- `port`: is the port used by the mailserver host

You can configure the following parameters for an e-mail transport proxy service:

- Service Account: This is a mandatory parameter. This is the service account resource. The service account consists of a `user name/password` combination required to access the e-mail account.
- Polling Interval: This is a mandatory parameter. This parameter specifies the interval in milliseconds. The default value is `60` ms.
- E-mail protocol: This is a mandatory parameter. There are two types of protocol from which you can select, `imap` and `pop3`. The default protocol is `pop3`.
- Read Limit: This is a mandatory parameter. This specifies the number of files to be read in each poll. The default value is `10`.
- Pass By Reference: If this parameter is enabled, the file is staged in the archive directory and passed as a reference in the message headers.
- Post Read Action: This is a mandatory parameter. This specifies whether the files should be deleted, moved, or archived after being read by the service. By default the files are deleted after reading.
- Attachments: This is a mandatory parameter. This parameter specifies if the attachments are to be archived or ignored. By default this parameter is set to `ignore`.

- IMAP Move Folder: This is the destination of the messages if the `Post Read Action` is set to `move`.

  Note: You must configure this field only if `Post Read Action` is set to `move`.

- Download Directory: This is a mandatory parameter. It specifies the file system directory path to download the message.
- Archive Directory: This is a mandatory parameter. A file URI that points to the directory where the files are archived. This field is active only when `Post Read Action` parameter is set to `archive`.
- Error Directory: This is a mandatory parameter. This URI points to a directory in which the contents of the file will be stored in case of an error.
- Request Encoding: This is an optional parameter. This parameter specifies the type of encoding to read the request message. The default encoding is `iso-8859-1`.

For more information on how to configure e-mail services, see Adding a Proxy Service: Transport Configuration in *Using the AquaLogic Service Bus Console.*

Configuring Business Services using E-mail Transport Protocol

When you configure a business service using the e-mail transport protocol, you must specify the endpoint URI in the following format:

`mailto:<`*`name@domain_name.com`*`>`

where <`name`*`@domain_name.com`*`>` is the e-mail destination.

You can configure the following parameters for an e-mail transport business service:

- SMTP Server: You must select an SMTP Server from the drop-down list.

  Note: You must first create the SMTP Server resource.

- Mail Session: This parameter is optional. It is the JNDI name of the configured mail session. You can select mail sessions from the drop-down list.

Notes:

- You must first configure mail sessions in the WebLogic Server Console. For more information on configuring a mail session, see Create a Mail Session in *WebLogic Server Administration Console.*

  - Also you should set either the `Mail Session` parameter or the `SMTP Server` parameter.

- From Name: This is an optional parameter. This parameter specifies the name from which the reply should be sent.
- From Address: This is an optional parameter. This parameter specifies the e-mail address from which the e-mail message should be sent.
- Reply To Name: This is an optional parameter. This parameter specifies the name to which the reply should be sent.
- Reply To Address: This is an optional parameter. This parameter specifies the e-mail address, to which the reply should be sent.
- Connection Timeout: This is an optional parameter. You can use this parameter to specify time in milliseconds after which the connection to the SMTP server times out.
- Request Encoding: This is an optional parameter. This parameter specifies the type of encoding to read the request message. The default encoding is `iso-8859-1`.

For more information on how to configure this transport, see Adding a Business Service: Transport Configuration in *Using the AquaLogic Service Bus Console.*

EJB

An EJB business service is a `Transport Typed Service`, that is, the type of the transport is determined by the configuration of the service. The EJB transport supports native Remote Method Invocation (RMI) of Stateless Session Beans deployed on WebLogic Server. The EJB transport can also be leveraged to directly expose an EJB as a Web service through AquaLogic Service Bus. For information about the EJB transport, see “EJB Transport” on page 6-1.

File

You can select the file transport protocol when you configure a Messaging Type or Any XML Service type of proxy service and the endpoint URI is of the form:

```
file:///<root-dir/dir1>
```

where `root-dir/dir1` is the absolute path to the destination directory.
The following sections describe:

- “Configuring Proxy Services using File Transport Protocol” on page 7-5

- "Configuring Business Services using File Transport Protocol" on page 7-6

Configuring Proxy Services using File Transport Protocol

To configure the file transport for a proxy service you must specify the following fields:

- File Mask: This is an optional parameter. This specifies the files that should be polled by the proxy service. If the URI is a directory and `*.*` is specified, then the service will poll for all the files in the directory.
- Polling Interval: This is a mandatory parameter. This specifies the value for the polling interval in milliseconds. The default value is `60` ms.
- Read Limit: This is a mandatory parameter. This specifies the number of files to be read in each poll. The default value is `10`.

  Note: If '`0`' is specified, all the files are read.
- Sort By Arrival: This is an optional parameter. This parameter indicates the sequence of events raised in the order of the arrival of files. The default value for this parameter is `False`.
- Scan Subdirectories: This is optional. If enabled, the sub-directories are also scanned.
- Pass By Reference: If this parameter is enabled, the file is staged in the archive directory and passed as a reference in the headers.
- Post Read Action: This parameter is mandatory. This specifies whether the files should be deleted or archived after being read by the service. By default the files are to be deleted after reading.
- Stage Directory: This is a mandatory parameter. This file URI points to the staging directory.
- Archive Directory: This is a mandatory parameter. This file URI points to the directory where the files are archived. This field is active only when `Post Read Action` parameter is set to `archive`.
- Error Directory: This is a mandatory parameter. This URI points to a directory, in which the contents of the file will be stored in case of a error.
- Request Encoding: This is an optional parameter. This parameter specifies the type of encoding to read the request message. The default encoding is `utf-8`.

For more information on how to configure this transport, see Adding a Proxy Service: Transport Configuration in *Using the AquaLogic Service Bus Console.*

Configuring Business Services using File Transport Protocol

When you configure a business service using the file transport protocol you must specify the endpoint URI in the following format:

`file:///<root-dir/dir1>`

where `root-dir/dir1` is the absolute path to the destination directory.

When you use this type of transport to configure a business service you must configure the following fields:

- Prefix: This is an optional parameter. This parameter specifies the prefix to be attached to the filename.
- Suffix: This is an optional parameter. This parameter specifies the suffix to be attached to the filename.
- Request Encoding: This is an optional parameter. This specifies the type of encoding to read the message. The default encoding which will be used is `utf-8`.

For more information on how to configure this transport, see Adding a Business Service: Transport Configuration in *Using the AquaLogic Service Bus Console.*

FTP

You can select the FTP transport protocol when you configure a Messaging Type or Any XML Service type of proxy service and the endpoint URI is of the form:

`ftp://<hostname:port/directory>`

where

- `hostname`: is the name of the host on which the destination directory is stored.
- `port`: is the port number at which the `FTP` connection is made.
- `directory`: is the destination directory.

The following sections describe

- "Configuring Proxy Services using FTP Transport Protocol" on page 7-7

- “Configuring Business Services using FTP Transport Protocol” on page 7-8

Configuring Proxy Services using FTP Transport Protocol

To configure the FTP transport for a proxy service you must specify the following fields:

- User Authentication: You must select one of the following types of `User Authentication`:
  - anonymous: If you select `anonymous`, you do not require any login credentials to login to the FTP server, but you optionally supply your e-mail ID for identification.
  - external user: If you select `external user`, you have to reference a Service Account resource, which contains your `user name/password` for the FTP server.
- Pass By Reference: This is an optional parameter. If this parameter is enabled, the file is staged in the archive directory and passed as a reference in the headers.
- Remote Streaming: This is an optional parameter. Setting this parameter to `True` will poll FTP files directly from the remote server at processing time.
- File Mask: This is a mandatory parameter. This specifies the files that should be polled by the proxy service. If the URI is a directory and `*.*` is specified, then the service will poll all the files in the directory.
- Polling Interval: This is a mandatory parameter. This specifies the value for the polling interval in milliseconds. The default value is `60` ms.
- Read Limit: This is a mandatory parameter. This specifies the value for the polling interval in milliseconds. The default value is `60` ms.
- Post Read Action: This is a mandatory parameter. This specifies whether the files should be deleted or archived after being read by the service. By default the files are deleted after reading.
- Transfer Mode: This parameter specifies whether the mode of file transfer is `binary` or `ascii`. By default the transfer is `binary`.
- Stage Directory: This is a mandatory parameter. This file URI points to the staging directory.
- Archive Directory: This is a mandatory parameter. This file URI points to the directory where the files are archived. This field is active only when `Post Read Action` parameter is set to `archive`.

- Error Directory: This is a mandatory parameter. This URI points to a directory location, where the contents of the file will be stored in case of a error.
- Request Encoding: This is an optional parameter. This parameter specifies the type of encoding to read the request message. The default encoding is `utf-8`.
- Advanced Settings: Click the icon to expand the Advanced Settings section. Configuring parameters in this section is optional.
  - Scan Subdirectories: This is optional. If enabled the sub-directories are also scanned.
  - Sort By Arrival: This is an optional parameter. This parameter indicates the sequence of the events being raised in the order of the arrival of files. The default value for this parameter is False.
  - Timeout: This is an optional parameter. This parameter specifies the FTP timeout interval, in seconds, before the connection is dropped. The default value for this parameter is `0`.
  - Retry: This is an optional parameter. This parameter specifies the maximum number of retries for the FTP connection failures.

For more information on how to configure this transport, see Adding a Proxy Service: Transport Configuration in *Using the AquaLogic Service Bus Console.*

Configuring Business Services using FTP Transport Protocol

You can select the FTP transport protocol when you configure a Messaging Type or Any XML Service type of business service and the endpoint URI is of the form:

```
ftp://<hostname:port/directory>
```

where

- `hostname:` is the name of the host on which the destination directory is stored.
- `port:` is the port number at which the `FTP` connection is made.
- `directory:` is the destination directory.

To configure the FTP transport for a business service you must specify the following fields:

- User Authentication: You must select one of the following types of `User Authentication`:

  - anonymous: If you select `anonymous`, you do not require any login credentials to login to the FTP server. But you optionally supply your e-mail ID for identification.
  - external user: If you select `external user`, you have to reference a Service Account resource, which contains your `user name/password` for the FTP server.
- Prefix for destination filename: This is a mandatory parameter. This parameter specifies the prefix to be attached to the filename.
- Suffix for destination filename: This is a mandatory parameter. This parameter specifies the suffix to be attached to the filename.
- Request Encoding: This is an optional parameter. This parameter specifies the encoding for the request message.

  For more information on how to configure this transport, see Adding a Business Service: Transport Configuration in *Using the AquaLogic Service Bus Console.*

HTTP

The following sections describe:

- "Configuring Proxy Services using HTTP Transport Protocol" on page 7-9
- "Configuring Business Services using HTTP Transport Protocol" on page 7-10

Configuring Proxy Services using HTTP Transport Protocol

You can select HTTP as the transport protocol when you configure any type of proxy service and the endpoint URI is of the form:

`/<someService>`

where `someService` is the name of proxy service or a business service

To configure the HTTP transport for a proxy service you must specify the following fields:

- Basic Authentication Required: If enabled, basic authentication is required to access this service.
- Dispatch Policy: You must configure Work Managers in the WebLogic Server Administration Console in order to have other dispatch policies in addition to the `default` dispatch policy. For more information on how to configure a Work Manager, see Create a Global Work Manager in WebLogic Server Administration Console.

- Request Encoding: This parameter specifies the character set encoding for the request messages.
- Response Encoding: This parameter specifies the character set encoding for the response messages.

For more information on how to configure this transport, see Adding a Proxy Service: Transport Configuration in *Using the AquaLogic Service Bus Console*.

Configuring Business Services using HTTP Transport Protocol

You can select HTTP as the transport protocol when you configure any type of business service and the endpoint URI is of the form:

`http://<host:port/someService>`

where

- `host:` is the name of the system that hosts the service.
- `port:` is the port number at which the connection is made.
- `someService:` is a target service.

To configure the HTTP transport for a business service you must specify the following fields:

- Timeout: This parameter specifies the HTTP timeout interval, in seconds, before the connection is dropped. The default value for this parameter is `0`.
- HTTP Request Method: This parameter enables you to select the `get` method or the `post` method for the HTTP requests.
- Basic Authentication Required: If enabled, basic authentication is required to access this service.
- Service Account: This resource contains the login credentials required for the `Basic Authentication`.
- Follow HTTP redirects: Enabling this parameter allows the business to follow the `HTTP` redirects.
- Dispatch Policy: You must configure Work Managers in the WebLogic Server Administration Console in order to have other dispatch policies in addition to the `default` dispatch policy. For more information on how to configure a Work Manager, see Create a Global Work Manager in *WebLogic Server Administration Console*.

- Request Encoding: This parameter specifies the character set encoding for request messages.
- Response Encoding: This parameter specifies the character set encoding for response messages.

For more information on how to configure this transport, see Adding a Business Service: Transport Configuration in *Using the AquaLogic Service Bus Console*.

HTTP(S)

You can select the HTTP(S) transport protocol when you configure any type of proxy service and the endpoint URI is of the form:

`/<someService>`

where `someService` is the name of a proxy service or a business service. Following sections describe:

- "Configuring Proxy Services using HTTP(S) Transport Protocol" on page 7-11
- "Configuring Business Services using HTTP(S) Transport Protocol" on page 7-12

Configuring Proxy Services using HTTP(S) Transport Protocol

To configure the HTTP(S) transport for a proxy service you must specify the following fields:

- Client Authentication: The Client Authentication method provides you with three options:
  - None: This option enables one-way SSL. No client authentication is required.
  - Basic: This option enables one-way SSL. But this requires `user/password` client authentication.
  - Client Certificate: This option enables one-way SSL. But this requires user/password client-side and server-side authentication. BEA recommends this method of client authentication.
- Dispatch Policy: You must configure Work Managers in the WebLogic Server Administration Console in order to have other dispatch policies in addition to the `default` dispatch policy. For more information on how to configure a Work Manager, see Create a Global Work Manager in *WebLogic Server Administration Console*.

- Request Encoding: This parameter specifies the character set encoding for the request messages.
- Response Encoding: This parameter specifies the character set encoding for the response messages.

For more information on how to configure this transport, see Adding a Proxy Service: Transport Configuration in *Using the AquaLogic Service Bus Console*.

Configuring Business Services using HTTP(S) Transport Protocol

You can select the HTTP(S) transport protocol when you configure any type of business service and the endpoint URI is of the form:

`http(s)://<host:port/someService>`

where

- `host`: is the name of the system that hosts the service.
- `port`: is the port number at which the connection is made.

To configure the HTTP(S) transport for a business service you must specify values for the following fields:

- Timeout: This parameter specifies the HTTP timeout interval, in seconds, before the connection is dropped. The default value for this parameter is `0`.
- HTTP Request Method: This parameter enables you to select the `get` method or the `post` method for the HTTP requests.
- Basic Authentication Required: Enabling basic authentication is required to access this service.
- Service Account: This resource contains the login credentials required for the Basic Authentication.
- Follow HTTP redirects: Enabling this parameter allows business services to follow `HTTP` redirects.
- Dispatch Policy: You must configure Work Managers in the WebLogic Server Administration Console in order to have other dispatch policies in addition to the `default` dispatch policy. For more information on how to configure a Work Manager, see Create a Global Work Manager in *WebLogic Server Administration Console*.

- Request Encoding: This parameter specifies the character set encoding for the request messages.
- Response Encoding: This parameter specifies the character set encoding for the response messages.

For more information on how to configure this transport, see Adding a Business Service: Transport Configuration in *Using the AquaLogic Service Bus Console*.

JMS

You can select JMS as the transport protocol for all the types of proxy services. AquaLogic Service Bus is certified with the following JMS implementations:

- WebLogic Server 9.x JMS
- IBM WebSphere MQ/JMS release 5.3
- TIBCO Enterprise Message ServiceTM release 4.2

The proxy services and business services must be configured to use the JMS transport as described in Adding a Proxy Service: Transport Configuration and Adding a Business Service: Transport Configuration sections of *Using the AquaLogic Service Bus Console*.

For more information on the JMS transport, see Interoperability Solutions for JMS and WebSphere MQ. The following sections describe:

- "Configuring Proxy Services using JMS Transport Protocol" on page 7-13
- "Configuring Business Services using JMS Transport Protocol" on page 7-16

Configuring Proxy Services using JMS Transport Protocol

You can select the JMS transport protocol when you configure any type of proxy service and the endpoint URI is of the form:

```
jms://<host:port[,host:port]*/factoryJndiName/destJndiName>
```

where

- `host`: is the name of the system that hosts the service.
- `port`: is the port number at which the connection is made.
- *`[,host:port]*:`* indicates that you can configure multiple hosts with corresponding ports.

    - `factoryJndiName`*:* The name of the JNDI Connection Factory. For more information on how to define a connection factory queue, see Configure resources for JMS system modules in *Administration Console Online Help*.
    - `destJndiName`: is the name of the JNDI destination.

To target a JMS destination to multiple servers, use the following format of the URI:
`jms://host1:port,host2:port/QueueConnectionFactory/destJndiName`
where *`QueueConnectionFactory`* is name of the connection factory queue. For more information on how to define a connection factory queue, see Configure resources for JMS system modules in *Administration Console Online Help*.

To configure a proxy service using JMS transport protocol you must specify values for the following fields:

- Destination Type: You must specify the destination to be of one of the following:
  - Queue: This defines a point-to-point destination type. You can use this destination type for peer asynchronous communications. A message, which is delivered to a queue is distributed to only one destination.
  - Topic: This defines a publish or a subscribe destination type. You can use this destination type for peer asynchronous communications. A message, which is delivered to a topic is distributed to all the subscribers of the topic.
- Is Response Required: If you expect a response to the outbound message, you must specify the following parameters for the response message:
  - Response Correlation Pattern: This configures the design pattern for JMS response message to be one of the following:
    - JMS Correlation ID: You must select `JMS Correlation ID` design pattern for your message if you want to correlate by JMS Correlation ID and send the response to the URI configured in the `Response URI` field.

    Note: The `Response URI` field is active only when you select `JMS Correlation ID` design pattern for your response message.

    - JMS Message ID: You can JMS Message ID design pattern for your message if you want to correlate by JMS Message ID and send the response to the `JMSReplyTo Destination` by configuring the `Response Connection Factory` field.

    Note: The `Response Connection Factory` field is active only when you select `JMS Message ID` design pattern for your response message.

  - Response Message Type: You can set type of the response message to `Bytes` or `Text`.

  - Response Encoding: You can set the encoding for the response message. The default encoding is `UTF-8`.
  - Client Response Timeout: This parameter specifies the response timeout interval, in seconds.
- Request Encoding: You can set the encoding for the request message. The default encoding is `UTF-8`.
- Dispatch Policy: This specifies the dispatch policy for the endpoint. You must configure Work Managers in the WebLogic Server Administration Console in order to have other dispatch policies in addition to the default dispatch policy. For more information on how to configure a Work Manager, see Create a Global Work Manager in *WebLogic Server Administration Console*.
- Advanced Settings: Click the icon to expand the Advanced Settings section. Configuring parameters in this section is optional.

  You can set the following parameters in the section:
  - Use SSL: This specifies whether the connections can be made over SSL or not.
  - Message Selector: You can use this field to specify the criteria for selecting messages.
  - Durable Subscription: You can check if the subscription is durable only if the destination type is `Topic`.
  - Retry Count: In this field you can configure the maximum number of retries for the connection.
  - Retry Interval: In this field you can configure the time interval in milliseconds between consecutive retries.
  - Error Destination: In this field you can configure the name of the target destination for the messages, which have reached the maximum number of retry count.
  - Expiration Policy: In this field you can specify the message expiration policy to be used when you encounter an expired message at a WebLogic Server or a JMS destination.
  - JMS Service Account: In this field you can specify the name of the service account resource to be used to make the connection over the secured socket layer.

Configuring Business Services using JMS Transport Protocol

You can select the JMS transport protocol when you configure any type of business service and the endpoint URI is of the form:

`jms://<host:port[,host:port]*/factoryJndiName/destJndiName` >

where

- `host`: is the name of the system that hosts the service.
- `port`: is the port number at which the connection is made.
- *`[,host:port]*:`* indicates that you can configure multiple hosts with corresponding ports.
- `factoryJndiName`*:* The name of the JNDI Connection Factory. For more information on how to define a connection factory queue, see Configure resources for JMS system modules in *Administration Console Online Help*.
- `destJndiName`: is the name of the JNDI destination.

To target a target a JMS destination to multiple servers, use the following format of the URI: `jms://host1:port,host2:port/QueueConnectionFactory/destJndiName`

where *`QueueConnectionFactory`* is name of the connection factory queue. For more information on how to define a connection factory queue, see Configure resources for JMS system modules in *Administration Console Online Help*.

When you register a JMS business service, you must manually edit the URI from the WSDL file when adding it to the service definition. The URI format is as follows:

`jms://<host>:<port>/factoryJndiName/destJndiName`

To configure a business service using the JMS transport protocol you must specify values for the following fields:

- Destination Type: You can specify the destination to be of one of the following:
  - Queue: This defines a point-to-point destination type. You can use this destination type for peer asynchronous communications. A message, which is delivered to a queue is sent to only one destination.
  - Topic: This defines a publish or a subscribe destination type. You can use this destination type for peer asynchronous communications. A message, which is delivered to a topic is distributed to all the subscribers of the topic.

- Is Response Required: You can specify if you expect a response to the outbound message. If you expect a response you must specify the following parameters for the response message:
  - Response Correlation Pattern: You can configure the design pattern for JMS response message to be one of the following:
    - JMS Correlation ID: You must select a `JMS Correlation ID` design pattern for your message if you want to correlate by JMS Correlation ID and send the response to the URI configured in the `Response URI` field.

    Note: The `Response URI` field is active only when you select `JMS Correlation ID` design pattern for your response message.

    - JMS Message ID: You must select JMS Message ID design pattern for your message if you want to correlate by JMS Message ID and send the response to the `JMSReplyTo Destination` by configuring the `Response Connection Factory` field.

    Note: The `Response Connection Factory` field is active only when you select `JMS Message ID` design pattern for your response message.
  - Response Message Type: You can set type of the response message to `Bytes` or `Text`.
  - Response Encoding: You can set the encoding for the response message. The default encoding is `UTF-8`.
  - Client Response Timeout: This parameter specifies the response timeout interval, in seconds.
- Request Encoding: You can set the encoding for the request message. The default encoding is `UTF-8`.
- Dispatch Policy: You can specify the dispatch policy for the endpoint. You must configure Work Managers in the WebLogic Server Administration Console in order to have other dispatch policies in addition to the `default` dispatch policy. For more information on how to configure a Work Manager, see Create a Global Work Manager in *WebLogic Server Administration Console*.
- Advanced Settings: To configure the advance settings click on ⏫ icon on the right hand side to expand the advanced settings section. Configuring parameters in this section is optional.

  You can set the following parameters in the section:

- Use SSL: This specifies if the connections can be made over SSL or not.
- Expiration: This specifies the time interval in milliseconds after which the message will expire. Default value is `0`, which means that the message never expires.
- Unit Of Order: This is a value added feature of WebLogic, which enables a message producer to group messages into a single unit with respect to the order of processing. This single unit is called the `Unit of Order`. All the messages in a unit must be processed sequentially in the same order they were created.

  Note: This is supported by WebLogic Server 9.0.
- JNDI Service Account: You can select the service account resource to be used for JNDI lookups.
- JMS Service Account: You can select the service account resource to be used for the JMS server connection.

Local

Every proxy service is associated with a protocol that determines the level of communication used by the clients to send requests to the proxy service. In AquaLogic Service Bus there two categories of proxy services—the proxy services of first category are invoked directly by the clients; those of the second category are invoked by other proxy services in the message flow. The proxy services of the second category use a new transport called the *local transport*. For more information on Local Transport, see "Local Transport" on page 8-1.

Tuxedo

BEA AquaLogic Service Bus and BEA Tuxedo can inter-operate to use the services each of them offer. The Tuxedo transport is secure, reliable, high performing, and provides bi-directional access to the Tuxedo domain from AquaLogic Service Bus. You can access domains in AquaLogic Service Bus from Tuxedo. Also you can access tuxedo domains from AquaLogic Service Bus. For more information about Tuxedo transport, see *Interoperability Solution for Tuxedo*.

You can configure both proxy services and business services in AquaLogic Service Bus. For more information on configuring a proxy service, see Adding a Proxy Service: Transport Configuration and for more information on configuring a business service, see Adding a Business Service: Transport Configuration sections of *Using the AquaLogic Service Bus Console*.

Data Services Platform (DSP)

AquaLogic Service Bus provides optimized transport for 1-way or 2-way communication for invoking services on AquaLogic Data Services Platform Data using a native Data Services Platform (DSP) transport. You access the AquaLogic Data Services Platform from AquaLogic Service Bus thus allowing an AquaLogic Service Bus client to make full use of data services. For detailed information on accessing AquaLogic Data Services Platform from AquaLogic Service Bus, see `http://edocs.bea.com/alsb/docs26/aldsp_transport_for_alsb2_6.pdf`.

You can select the DSP transport protocol when you configure SOAP or XML business service types. To learn more, see "Adding a Business Service" in *Using the AquaLogic Service Bus Console*.

If you create a proxy service from a DSP transport business service, AquaLogic Service Bus will switch the transport type of the proxy service to HTTP. This is because the DSP transport cannot be used for proxy services. You can then change the transport type of the proxy service to any other available transport.

To learn more, see "Adding a Proxy Service" in *Using the AquaLogic Service Bus Console*.

Local Transport

This chapter provides information about the AquaLogic Service Bus local transport. It includes the following topics:

Introduction

Commonly, service bus architectures include complex message flows, in which messages are routed through multiple proxy services that are organized into larger multiple proxy service flows. Individual proxy services in these multiple proxy service scenarios route, publish, or callout to the next proxy service in the flow. The reason for a multiple proxy services design is to support modularity and compartmentalization of the various components of the end-to-end message flow. The individual proxy services in a multiple proxy service flows need to:

- Communicate efficiently and securely.
- Allow transactions and transactional behavior to be propagated
- Allow security context to be propagated so that the identity can be propagated end-to-end. The security context propagation also allows the client of the first proxy service in a multiple service flow to be authorized by the proxy services that are subsequently invoked

in the flow—thus supporting fine-grained access control generic headers in the local transport.
Using the local transport for proxy services ensures support for these capabilities.

Features and Characteristics of Local Transport Proxy Services

Local transport-based proxy services can only be invoked by other proxy service, not by other clients. The invocation is optimized by AquaLogic Service Bus. Local proxy services do not have an URI. However, there are no constraints on the service and interface types supported by local transport proxy services. The one exception is that SAML is only supported in a pass through scenario.

If the quality of service (QoS) for an invoking proxy service is defined as Exactly Once, the transaction of that service is propagated to the local transport proxy service.

In other words, the invoked local transport proxy service inherits the transactional behavior of the invoking proxy service. A proxy service can authenticate at the transport level or the message level. If it is enabled, the effective client is the message-level authenticated client. If the message-level authenticated client is not enabled, then the transport-level authenticated client is the effective client (if that is enabled). If neither the message-level nor the transport-level authenticated client is enabled, the anonymous client becomes the effective client.

When a proxy service invokes a local transport proxy service, the effective client of the invoking proxy service becomes the transport-level client of the invoked local proxy service. A local transport proxy service can authorize this client for access with an access control policy. In this way, it is possible to propagate the client of the first proxy service to all the subsequent proxy services in the overall end-to-end message flow.

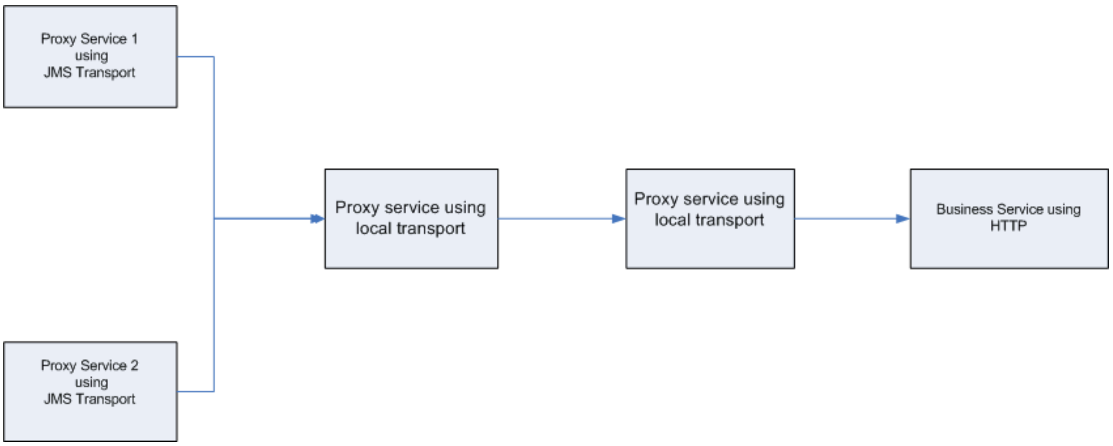
Local transport proxy services support user-defined transport headers. Consider a scenario in which a proxy service uses the HTTP transport; it routes to a local proxy service and the HTTP proxy service passes headers to the local proxy service using the Transport Header action. In this scenario, if the HTTP proxy service received the Content-Type header, this header is available as a user header in the local transport and is therefore accessible through the standard user header, instead of as a typed transport header.

You can invoke a local transport proxy service from the AquaLogic Service Bus test browser. Metrics are collected for a local transport proxy service in the same way as they are any other service. However, local transport proxy services cannot be published to UDDI.

Usage of Local Transport Proxy Services

A common scenario that can be supported using local transport proxy services is one in which a proxy service needs to be invoked using different transports. This can be achieved by putting a set of front-end proxy services (one service per transport) in front of a local transport proxy service in the path of the message flow. These front-end proxy services simply route messages to the local transport proxy service. The following figure illustrates this scenario.

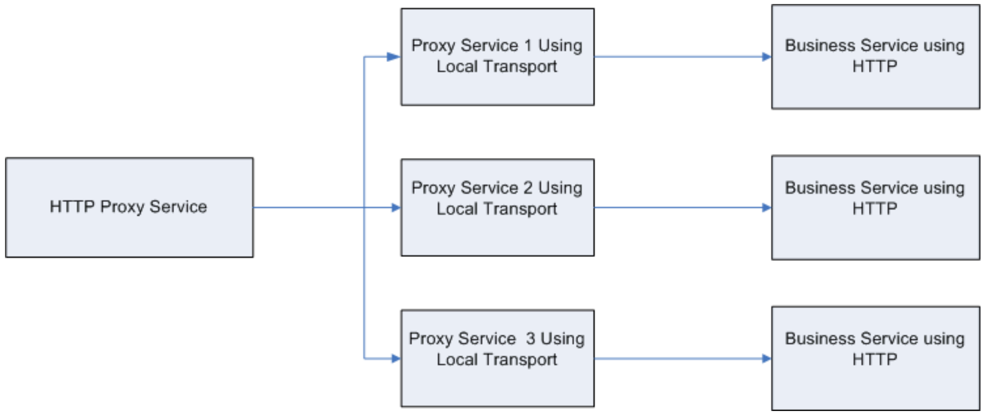
Figure 8-1 Using Local Transport to Implement Convergence


Proxy Service 1 using JMS Transport
Proxy Service 2 using JMS Transport
Proxy service using local transport
Proxy service using local transport
Business Service using HTTP

Another common scenario is one in which an Any SOAP or XML type proxy service acts as a front-end to different enterprise systems. This front-end proxy service can receive messages in a variety of formats and uses a technique common to all these messages (for example, a WS-Addressing SOAP header) to route the messages to an appropriate local transport proxy service. In this scenario, the front-end proxy service is acting as a generic router with little knowledge of the enterprise systems or the message formats and semantics. To further abstract the knowledge of the routing rules at design time, the front-end proxy service can use dynamic routing to route messages to the local transport proxy services. For an example of how dynamic routing is used in the proxy services, see "Using Dynamic Routing" on page 2-44 in the users guide.

Each of the local transport proxy services to which messages are routed from the front-end service in turn acts as a front-end proxy service for a specific business service. The local transport proxy services are aware of the message format required by the business services to which they route. In this scenario, these local transport proxy services act as functional proxy services. The roles of a functional proxy services are to enforce access control for invoking a particular business service, and to perform any transformation of the messages required to invoke the target business service correctly. The following figure illustrates this scenario.

Figure 8-2 Using Local Transport to Access Multiple Business Services


HTTP Proxy Service
Proxy Service 1 Using Local Transport
Proxy Service 2 Using Local Transport
Proxy Service 3 Using Local Transport
Business Service using HTTP
Business Service using HTTP
Business Service using HTTP

Limitations

The limitations of the local transport are:

- You can invoke the proxy service using the local transport only from another proxy service.
- The proxy services using the local transport cannot be published to the UDDI.
- A local transport proxy service cannot process inbound WS-Security SAML tokens.

Extensibility Using Java Callouts and POJOs

To allow you to extend the capabilities of AquaLogic Service Bus in your organization, you can invoke custom Java code from within proxy services. AquaLogic Service Bus 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 AquaLogic Service Bus Console at design time; the parameters can be mapped to message context variables.

For information about configuring a Java Callout to a POJO, see "Java Callout" in Proxy Services: Actions in *Using the AquaLogic Service Bus Console.*

Usage Guidelines

The scenarios in which you can use Java Callouts to POJOs in AquaLogic Service Bus 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 AquaLogic Service Bus 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.

Note: The input and return types for Java Callouts are limited to primitives and XmlObject.

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 AquaLogic Service Bus. 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 AquaLogic Service Bus 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.

XQuery Implementation

AquaLogic Service Bus 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/`

AquaLogic Service Bus 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 10-2.
- AquaLogic Service Bus-specific function extensions. See "Function Extensions from AquaLogic Service Bus" on page 10-2.

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

AquaLogic Service Bus 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 AquaLogic Service Bus—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 AquaLogic Service Bus

AquaLogic Service Bus 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. AquaLogic Service Bus 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 AquaLogic Service Bus 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 AquaLogic Service Bus message flows--see "Accessing Databases Using XQuery" on page 2-48. 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 AquaLogic Service Bus 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 AquaLogic Service Bus:

- "Example 1: Retrieving the URI from a Database for Dynamic Routing" on page 10-5
- "Example 2: Getting XMLType Data from a Database" on page 10-7

Example 1: Retrieving the URI from a Database for Dynamic Routing

AquaLogic Service Bus proxy services support specification of the URI to which messages are to be routed at run time (dynamically)—see "Using Dynamic Routing" on page 2-44. The following listing is an example use of the `fn-bea:execute-sql()` function to retrieve the URI from a database in a dynamic routing scenario.

Listing 10-1 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 the preceding example:

- `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 follows:

Listing 10-2 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 10-1 is executed as a result of a proxy service receiving the request message in Listing 10-3 (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 10-2.)

Listing 10-3 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 10-4 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 10-5.

Listing 10-5 $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
```

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 AquaLogic Service Bus XQueries, see "Accessing Databases Using XQuery" on page 2-48.

For complete information about database and JDBC drivers support in AquaLogic Service Bus, 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	BEA AquaLogic Service Bus 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 2-48

"fn-bea:execute-sql()" on page 10-4

Tuning AquaLogic Service Bus

This section provides AquaLogic Service Bus tuning tips.

- Whenever possible, set the logging level to *warning*. You set the logging level in the WebLogic Server Administration Console. For more information, see Servers: Logging: General in the WebLogic Server Administration Console *Online Help*. The following code displays the output server `config.xml` file when the logging level is set to warning. For more information on logging, see “Log” in Proxy Services: Actions in the *Using the AquaLogic Service Bus Console*.

```
<server>
  <name>AdminServer</name>
  <log>
    <file-min-size>5000</file-min-size>
    <log-file-severity>Warning</log-file-severity>
    <log-file-filter xsi:nil="true"></log-file-filter>
    <stdout-severity>Off</stdout-severity>
    <stdout-filter xsi:nil="true"></stdout-filter>
   <domain-log-broadcast-severity>Error</domain-log-broadcast-severity>
    <domain-log-broadcast-filter
xsi:nil="true"></domain-log-broadcast-filter>
    <memory-buffer-severity>Error</memory-buffer-severity>
    <memory-buffer-filter xsi:nil="true"></memory-buffer-filter>
  </log>
```

```
</server>
```

- Group JMS queues on different JMS servers based on message loads. Different JMS servers use different file stores, which you can distribute to separate disk volumes. For more information, "Adding a Business Service" in Business Services in the *Using the AquaLogic Service Bus Console* and pay particular attention to the *JMS* configuration information.

- If you are using an Oracle database as a JMS persistent store, it is recommended that you use a 10g database and ensure that it has sufficient JDBC connections. Create a JDBC store on a separate schema to use a separate tablespace.

- If you do not require monitoring for a proxy or business service, disable the monitoring capability. For more information, see "Overview of Monitoring" in Monitoring in the *Using the AquaLogic Service Bus Console*.

- If possible, set the routing data in the JMS message properties. AquaLogic Service Bus does not deserialize message content until the content is explicitly accessed in the pipeline. For example, if the content is an XML document, XML parsing does not happen until an XQuery or XSLT operation happens in the pipeline. For more information about working with the message context in the message flow, see "Message Context" on page 3-1.

- If you need to extract some of the inbound header elements for processing, you should specify that AquaLogic Service Bus retrieves specific header elements instead of all the elements.

- If you are using an Oracle database as a JMS persistent store BEA recommends that you should ensure that enough JDBC connections are available. This allows an administrator to tune the block size based upon the message size. It is also possible to create each tablespace on a separate datafile and put these datafiles on separate disks. Such an arrangement allows for greater concurrency at the hardware level when the datafiles are not stored on a RAID.

- Where possible, use the insert action instead of the assign action. The insert action uses "in-place" modification semantics enhancing the performance compared to the assign action. For information on configuring actions, see "Adding an Action" in Proxy Services: Actions in the *Using the AquaLogic Service Bus Console*.

- Use AquaLogic Service Bus clustering and WebSphere MQ clustering to achieve scalability.

- When you are configuring the **Accept Backlog** parameter in the WebLogic Server BEA recommends that you should first increase the default value by twenty five percent.If the 'connection refused, socket exception' is thrown, then continue increasing the value till the exception is not thrown. For more information on tuning the WebLogic Server see Tuning WebLogic Server.

- If a front-end application invokes AquaLogic Service Bus synchronously, AquaLogic Service Bus can use the synchronous-asynchronous feature to communicate with WebSphere MQ synchronously. On the WebSphere MQ side, a request and a response queue is set up. AquaLogic Service Bus sends a request to the request queue and waits for a response from the response queue. To achieve improved performance, you can use a dedicate work manager for the response Message Driven Bean. You configure the dedicate work manager in the WebLogic Server Administration Console. For more information, see Work Manager in the *WebLogic Server Administration Console Online Help*. The following code displays the output server `config.xml` file after the dedicate work manager is configured.

```
<self-tuning>
 <min-threads-constraint>
    <name>minThreadsConstraint</name>
    <target>AdminServer</target>
    <count>20</count>
 </min-threads-constraint>
 <work-manager>
    <name>MQWorkManager</name>
    <target>AdminServer</target>
    <min-threads-constraint> minThreadsConstraint
</min-threads-constraint>
    <ignore-stuck-threads>false</ignore-stuck-threads>
 </work-manager>
</self-tuning>
```

- AquaLogic Service Bus uses a two-level cache for proxy service run-time data with static and dynamic sections. The static section is a size-based Least Recently Used (LRU) cache for proxy services, immune from garbage collection. When a proxy service is bumped from the static section, it is demoted to the dynamic section. Proxy services in the dynamic section can continue to be reused but are susceptible to being deleted during a Java garbage collection cycle.

  The cache introduces a performance trade off between memory consumption and compilation cost. Caching proxy services increases throughput but consume memory otherwise available for data processing.

You can tune the cache by setting its size using the system property `com.bea.wli.sb.pipeline.RouterRuntimeCache.size`

The default value is 100. This is the number of proxy services in the static portion of the cache.

Approximately every minute, a log message will be printed with the following format:

```
[RuntimeRouterCache] XX hits received: YY hits to main cache, ZZ hits to
soft cache, MM misses
```

Where:

– `XX` represents the number of times that the cache was accessed since the last log.

– `YY` represents the nummer of times (out of `XX`) that the cache contained the proxy service.

– `ZZ` represents the number of times (out of `XX`) that the cache contained the proxy service amongst those to be garbaged collected (for example, the cache was able to reclaim the proxy service before the garbage collector did).

– `MM` represents the number of times (out of `XX`) that the cache did not contain the proxy service and had to recreate it.

You can use these logs to tune the size of the cache. Consider the following guidelines:

– If your AquaLogic Service Bus domain contains a low number of proxy services, BEA recommends setting the size of the cache greater than that number.

– If your AquaLogic Service Bus domain contains a large number of proxy services, adjust the size of the cache to decrease the percentage of missed hits and still allow enough memory for data processing.

Debugging AquaLogic Service Bus

This section provides information about enabling debugging for different modules in AquaLogic Service Bus. You can enable and disable debugging by modifying the corresponding entries in the `wlidebug.xml` file, which is located in the root directory of the AquaLogic Service Bus domain. If the `wlidebug.xml` file is not in the root directory or if it has been deleted, it is created again without any contents when the server starts. The following listing provides an example of the contents of the `wlidebug.xml` file with debugging disabled for all modules (all entries set to `false`).

Listing B-1 wlidebug.xml File

```
<?xml version='1.0' encoding='UTF-8'?>

<java:wli-debug-logger xmlns:java="java:com.bea.wli.debug">

  <n1:Name xmlns:n1="java:weblogic.diagnostics.debug">wlidebug</n1:Name>

  <java:wli-management-debug>false</java:wli-management-debug>

  <java:wli-monitoring-debug>false</java:wli-monitoring-debug>

  <java:wli-management-dashboard-debug>false</java:wli-management-dashboard-de
bug>

  <java:wli-config-debug>false</java:wli-config-debug>

  <java:wli-config-transaction-debug>false</java:wli-config-transaction-debug>

  <java:wli-config-deployment-debug>false</java:wli-config-deployment-debug>
  <java:wli-config-component-debug>false</java:wli-config-component-debug>
  <java:wli-sb-transports-debug>false</java:wli-sb-transports-debug>
  <java:wli-sb-pipeline-debug>false</java:wli-sb-pipeline-debug>
  <java:wli-alert-manager-debug>false</java:wli-alert-manager-debug>
  <java:wli-jms-reporting-provider-debug>false</java:wli-jms-reporting-provide
r-debug>
  <java:wli-monitoring-aggregator-debug>false</java:wli-monitoring-aggregator-
debug>
  <java:wli-credential-debug >false</java:wli-credential-debug >
  <java:wli-management-common-debug >false</java:wli-management-common-debug >
</java:wli-debug-logger>
```

Although debugging should be disabled during normal AquaLogic Service Bus 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 debug flags are:

- `wli-config-debug`—Provides information on general aspects of AquaLogic Service Bus configuration.
- `wli-config-deployment-debug`— Provides debug information on session creation, activation, and distribution of configuration in a cluster.
- `wli-config-transaction-debug`—Provides low level debug information about changes made to in-memory data structures and files. This alert flag also generates server startup recovery logs.
- `wli-config-component-debug`—Provides low level debug information about create, update, delete, and import operations.
- `wli-sb-transports-debug`—Provides transport related debug information, including transport headers, which is printed per-message.
- `wli-sb-pipeline-debug`—Prints errors that are generated within the pipeline.
- `wli-alert-manager-debug`—Prints an evaluation of alerts.

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`, except for the `wli-monitoring-aggregator-debug` flag. The `wli-monitoring-aggregator-debug` flag enables debugging for aggregator. This flag logs the aggregated document every minute and stores the log files in the `{domain}\monitoring` folder.

Note: Turning the `wli-monitoring-aggregator-debug` flag on generates large amounts of debug data. Therefore, you should only use this flag for debugging purposes for short periods of time.

AquaLogic Service Bus APIs

AquaLogic Service Bus exposes APIs to allow customizing resources, provide external access to monitoring data, and deployment:

- Resource Update and Customization
- Management and Monitoring
- Deployment

Tip: Javadoc for the AquaLogic Service Bus APIs is provided at the following URL: `http://edocs.bea.com/alsb/docs26/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 in some way.

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 service
- BusinessServiceConfigurationMBean
  - Enable and Disable SLA Alerts
  - Synchronize business services imported from UDDI registries
  - Detach a collection of Business Services from a UDDI registry
- ALSBConfigurationMBean Interface Provides APIs to manage resources in an AquaLogic Service Bus domain, including:
  - Query, export, and import resources
  - Obtain validation errors
  - Get and set environment values
  - Modify references inside resources to new reference
  - 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 AquaLogic Service Bus provides external access to monitoring data. Java Management Extensions (JMX) technology was used for the implementation. AquaLogic Service Bus resources within a domain use Java Management Extensions (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 AquaLogic Service Bus MBeans in Java programs and WLST scripts to automate promotion of AquaLogic Service Bus 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*.