BEA AquaLogic Service Bus®

Transport SDK User Guide

Version 2.5
Document Revised: October 2006

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

1. Introduction

2. Design Considerations

3. Developing a Transport Provider

6. Sample Socket Transport Provider

A. UML Sequence Diagrams

CHAPTER 1

Introduction

This chapter describes the purpose of this guide, its intended audience, and general organization. The chapter includes these topics:

- Purpose of this Guide
- Audience for this Guide
- Overview of this Guide
- Related Information

Purpose of this Guide

This guide provides developers with the information needed to design, create, and deploy a new custom transport provider.

Audience for this Guide

This guide is written for experienced Java developers who want to add a new custom transport provider to AquaLogic Service Bus. It is assumed that you have solid knowledge of Web services technologies, AquaLogic Service Bus, the transport protocol that you want to use with AquaLogic Service Bus, and WebLogic Server.

Overview of this Guide

This guide provides developers with the information needed to design, create, and deploy a new custom transport provider. This guide is organized as follows:

- Chapter 2, "Design Considerations"

  Describes transport provider concepts and functionality to help you get started. It is important to review this chapter before developing a transport provider.

- Chapter 3, "Developing a Transport Provider"

  Explains the basic steps required to create a new custom transport provider as well as advanced topics.

- Chapter 4, "Deploying a Transport Provider"

  Explains how to package and deploy a new transport provider.

- Chapter 5, "Transport SDK Interfaces and Classes"

  Summarizes each of the interfaces and classes provided by the Transport SDK.

- Chapter 6, "Sample Socket Transport Provider"

  Discusses the sample socket transport provider that is provided with AquaLogic Service Bus. This sample includes public source code that you can examine and reuse.

- Appendix A, "UML Sequence Diagrams"

  Presents UML diagrams that help explain the flow of method calls through AquaLogic Service Bus runtime.

Related Information

The complete set of AquaLogic Service Bus and WebLogic Server documentation is available on edocs (`http://edocs.bea.com`). Specific documents that may be of interest to custom transport provider developers include:

- *AquaLogic Service Bus Concepts and Architecture*
- *AquaLogic Service Bus User Guide*
- *Using the AquaLogic Service Bus Console*
- *AquaLogic Service Bus Deployment Guide*
- Javadoc for AquaLogic Service Bus Classes

CHAPTER 2

Design Considerations

Careful planning of development activities can greatly reduce the time and effort you spend developing a custom transport provider. The following sections describe transport provider concepts and functionality to help you get started:

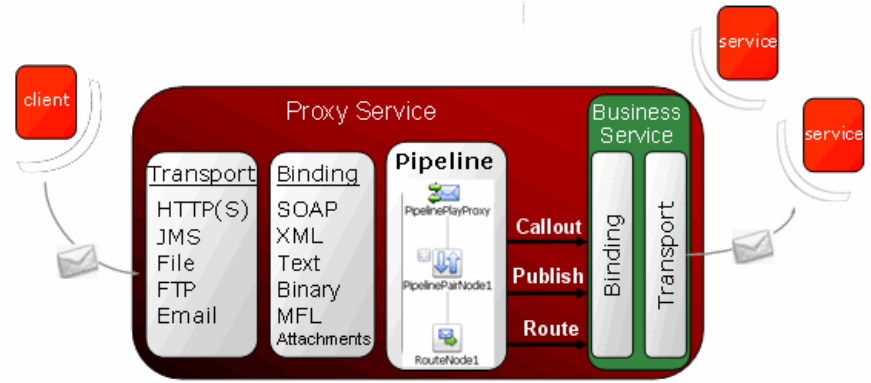
- What is a Transport Provider?
- What is the Transport SDK?
- Do You Need to Develop a Custom Transport Provider?
- Transport Provider Components
- The Transaction Model
- The Security Model
- The Threading Model
- Designing for Message Content

What is a Transport Provider?

A transport provider implements the interfaces of the Transport SDK and provides a bridge between AquaLogic Service Bus and mechanisms by which messages are sent or received. Such mechanisms can include specific transport protocols, such as HTTP, as well as other entities, such as a file or an e-mail message. A transport provider manages the life cycle and runtime behavior of transport endpoints. An endpoint is a resource where messages originate or are targeted.

Figure 2-1 illustrates the basic flow of messages through AquaLogic Service Bus. A client sends a message to AquaLogic Service Bus using a specific transport protocol. A transport provider processes the inbound message, handling communication with the service client endpoint and acting as the entry point for messages into AquaLogic Service Bus.

Figure 2-1 Message Flow Through AquaLogic Service Bus



The binding layer, also shown in Figure 2-1, packs and unpacks messages, handles message security, and hands messages off to the AquaLogic Service Bus Pipeline.

Tip: For more information on AquaLogic Service Bus message brokering and the role of the transport layer, see *AquaLogic Service Bus Concepts and Architecture*. For more detailed sequence diagrams that describe the message flow through AquaLogic Service Bus, see Appendix A, "UML Sequence Diagrams."

By default, AquaLogic Service Bus includes transport providers that support several commonly used transport protocols, such as HTTP, JMS, File, FTP, and others. These native providers let you configure proxy and business services that require these common transport protocols. These built-in providers are listed in Table 2-1.

Table 2-1 Transport Providers Installed with AquaLogic Service Bus

Transport Provider	Description
E-mail	Use the E-mail transport for sending and receiving e-mail messages. Inbound messages are received via IMAP or POP3 and outbound messages are sent via SMTP.
EJB	Use the EJB transport provider in business services to access EJBs potentially in other domains from AquaLogic Service Bus. The EJB transport provider is a self-described transport as defined by the Transport SDK and generates a WSDL to describe the service interface. This transport provider cannot be used in a proxy to expose AquaLogic Service Bus as an EJB.
File	Use the File transport to receive file based messages or to write files from the local file system.
FTP	Use the FTP transport provider to communicate with an FTP server to get or put an FTP file.
HTTP/HTTPS	Use the HTTP or HTTPS transport provider to send and receive HTTP/S messages.
JMS	Use the JMS transport provider to send and receive JMS messages.
Local	Use the Local transport provider with proxy services that are invoked by other proxy services in the message flow. In AquaLogic Service Bus there two categories of proxy services. One category which are invoked directly by the clients, while the proxy services 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.
Tuxedo	Use the Tuxedo transport provider for secure, reliability, high performance, bi-directional access to a Tuxedo domain from AquaLogic Service Bus. With this transport provider, BEA AquaLogic Service Bus and BEA Tuxedo can inter-operate to use the services each of them offer.

Tip: For more information using and configuring these native transport providers, see the *AquaLogic Service Bus User Guide*.

What is the Transport SDK?

This section briefly describes the purpose and features of the Transport SDK. This section includes these topics:

Purpose of the SDK

AquaLogic Service Bus processes messages independently of how they flow into or out of the system. The Transport SDK provides a layer of abstraction between AquaLogic Service Bus and components that deal with the flow of data in and out of AquaLogic Service Bus. This layer of abstraction allows you to develop new transport providers to handle unique transport protocols. For a list of the transport providers that are installed with AquaLogic Service Bus, see Table 2-1, "Transport Providers Installed with AquaLogic Service Bus," on page 2-3.

The SDK abstracts from the rest of AquaLogic Service Bus:

- Handling specific transport bindings
- Deploying service endpoints on the transport bindings. An endpoint is either capable of transmitting or receiving a message.
- Collecting monitoring information
- Managing endpoints (such as performing suspend/resume operations and setting connection properties)
- Enforcing Service Level Agreement (SLA) behavior (such as timing out connections)

Transport SDK Features

This section describes the primary features of the Transport SDK.

Handling Inbound and Outbound Messages

A transport provider developed with the Transport SDK handles inbound and outbound messages as follows:

- Inbound messages typically come into AquaLogic Service Bus from an outside source, such as an HTTP client. The Transport SDK packages the payload and transport level headers, if any, into a generic data structure. The Transport SDK then passes the message, in its generic format, to the AquaLogic Service Bus pipeline.
- Outbound messages originate from AquaLogic Service Bus business services and go to an externally managed endpoint, such as a Web service or JMS queue. The Transport SDK receives a generic data structure from the AquaLogic Service Bus pipeline, converts it to the corresponding transport-specific headers and payload, and sends it out to an external system.

The Transport SDK handles outbound and inbound messages independently. An inbound message can be bound to one transport protocol and bound to a different transport protocol on the outbound endpoint.

For more information on how messages flow through AquaLogic Service Bus, see the *AquaLogic Service Bus User Guide.*

Deploying Transport-Related Artifacts

Certain transports include artifacts that need to be deployed to WLS server. For instance, a JMS proxy is implemented as a message-driven bean. This artifact, an EAR file, must be deployed when the new JMS proxy is registered. Similarly, the EJB transport provider employs an EAR file that must be deployed when a new EJB business service is registered. Other kinds of artifacts might require deployment, such as a JMS transport, which may create queues and topics as part of the service registration. The SDK allows you to support these artifacts and lets you participate in the WLS deployment cycle. If the deployment of one of these artifacts fails, the AquaLogic Service Bus session is notified and the deployment is canceled. This feature of the SDK allows for the atomic creation of services. If something fails, the session reverts to its previous state.

Note: To participate in WLS deployment cycle, the transport provider must implement the TransportWLSArtifactDeployer interface. The primary benefit of this technique is atomic WebLogic Server deployment, which can be rolled back if needed. For more information on this interface, see "Summary of General Interfaces" on page 5-3 and see "When to Implement TransportWLSArtifactDeployer" on page 3-26.

Processing Messages Asynchronously

Because the server has a limited number of threads to work with when processing messages, asynchrony is important. This feature allows AquaLogic Service Bus to scale to handle large numbers of messages. After a request is processed, the thread is released. When the business

service receives a response (or is finished with the request if it is a one-way message), it notifies AquaLogic Service Bus asynchronously through a callback.

See also "Support for Synchronous Transactions" on page 2-15 and "The Threading Model" on page 2-21.

Transport Provider Modes

With the Transport SDK, you can implement inbound property modes and outbound property modes. These connection and endpoint modes are specified in the transport provider's XML Schema definition file. For more information on this file, see "3. Create an XML Schema File for Transport-Specific Artifacts" on page 3-5. This schema is available to the AquaLogic Service Bus Pipeline for filtering and routing purposes.

Related Features

This section lists related features that are provided by the transport manager. The transport manager provides the main point of centralization for managing different transport providers, endpoint registration, control, processing of inbound and outbound messages, and other functions. These features do not require specific support by a transport provider.

Load Balancing

The Transport SDK supports load balancing and failover for outbound messages. Supported load balancing options are:

- **None** – For each outbound request, the transport provider cycles through the URIs in the list in which they were entered and attempts to send a message to each URI until a successful send is completed.
- **Round Robin** – Similar to None, but in this case, the transport provider keeps track of the last URI that was tried. Each time a message is sent, the provider starts from the last position in the list.
- **Random** – The transport provider tries random URIs from the list in which they were entered.
- **Weighted Random** – Each URI is associated with a weight. An algorithm is used to pick a URI based on this weight.

Monitoring and Metrics

The transport manager handles these monitoring metrics:

- response-time (applies to inbound and outbound messages)
- message-count (applies to inbound and outbound messages)
- error-count (applies to inbound and outbound messages)
- failover-count (applies to outbound messages only)

Do You Need to Develop a Custom Transport Provider?

This section explains the basic use cases for writing a custom transport provider. In some cases, it is appropriate to chose an alternative approach. This section includes the following topics:

When to Use the Transport SDK

One of the prime use cases for the Transport SDK is to support a specialized transport that you already employ for communication between your internal applications. Such a transport may have its own concept of setup handshake, header fields, metadata, or transport-level security. Using the Transport SDK, you can create a transport implementation for AquaLogic Service Bus that allows configuring individual endpoints, either inbound, outbound or both. With a custom transport implementation, the metadata and header fields of the specialized transport can be mapped to context variables available in a proxy service pipeline.

Use the Transport SDK when the transport provider needs to be seamlessly integrated into all aspects of AquaLogic Service Bus for reliability, security, performance, management, user interface, and the use of the UDDI registry.

Some cases where it is appropriate to use the Transport SDK to develop a custom transport include:

- Using a proprietary transport that requires custom interfaces and supports an organization's existing applications.
- Using a CORBA or IIOP protocol for communicating with CORBA applications.
- Using other legacy systems, such as IMS and Mainframe.

- Using variations on existing transports, such as SFTP (Secure FTP) and the native IBM WebSphere MQ API (instead of WebSphere MQ JMS).
- Using industry-specific transports, such as LLP, AS3, and ACCORD.
- Using raw sockets, perhaps with TEXT or XML messages. A sample implementation of this type of transport is described in Chapter 6, "Sample Socket Transport Provider."

Alternatively, you can use the Transport SDK to support a specialized protocol over one of the existing transports provided with AquaLogic Service Bus. Examples of this could include supporting:

- Messages consisting of parsed or binary XML over HTTP.
- WS-RM or other new Web service standards over HTTP.
- Request-response messaging over JMS, but with a different response pattern than either of the two patterns supported by the AquaLogic Service Bus JMS transport (for example, a response queue defined in the message context).

When Alternative Approaches are Recommended

Creating a new AquaLogic Service Bus transport provider using the Transport SDK can be a significant effort. The Transport SDK provides a rich, full featured environment so that a custom transport has all of the usefulness and capabilities of the transports that come natively with AquaLogic Service Bus. But such richness brings with it some complexity. For certain cases, you might want to consider easier alternatives.

If you need an extension merely to support a different format message sent or received over an existing protocol, it may be possible to use the existing transport and use a Java Callout to convert the message. For example, suppose you have a specialized binary format (such as ASN.1 or a serialized Java object) being sent over the standard JMS protocol. In this case, you might consider defining the service using the standard JMS transport with the service type being a messaging service with binary input/output messages. Then, if the contents of the message are needed in the pipeline, a Java Callout action can be used to convert the message to or from XML. For information on using Java Callouts, see "Extensibility Using Java Callouts and POJOs" in the *AquaLogic Service Bus User Guide*.

Other cases where it is best not to use the Transport SDK to develop a custom transport provider include:

- When combining existing BEA solutions with AquaLogic Service Bus satisfies the transport requirement: WLS, WLI, ALDSP, ALBPM, Tuxedo, Portal.

- When service enablement tools, like BEA Workshop, provide a simpler and more standards-based mechanism to implement SOA practices.
- When alternative connectivity solutions (certified with AquaLogic Service Bus) also address the requirement. For example: iWay adapters and Cyclone B2B.
- When EJBs can be used instead as a means to abstract some type of simple Java functionality.

Transport Provider Components

This section presents UML diagrams that depict the runtime and design-time components of a transport provider. This section includes these topics:

Overview

In general, a custom transport provider consists of a design-time part and a runtime part. The design-time part is concerned with registering endpoints with the transport provider. This configuration behavior is provided by the implementation of the UI interfaces. The runtime part implements the mechanism of sending and receiving messages.

When you develop a new custom transport provider, you need to implement a number of interfaces provided by the SDK. This section includes UML diagrams that model the organization of the design-time and runtime parts of the SDK.

Tip: In AquaLogic Service Bus, implementations of the TransportProvider interface represent the central point for management of transport protocol-specific configuration and runtime properties. A single instance of a TransportProvider object exists for every supported protocol. For example, there are single instances of HTTP transport provider, JMS transport provider, and others.

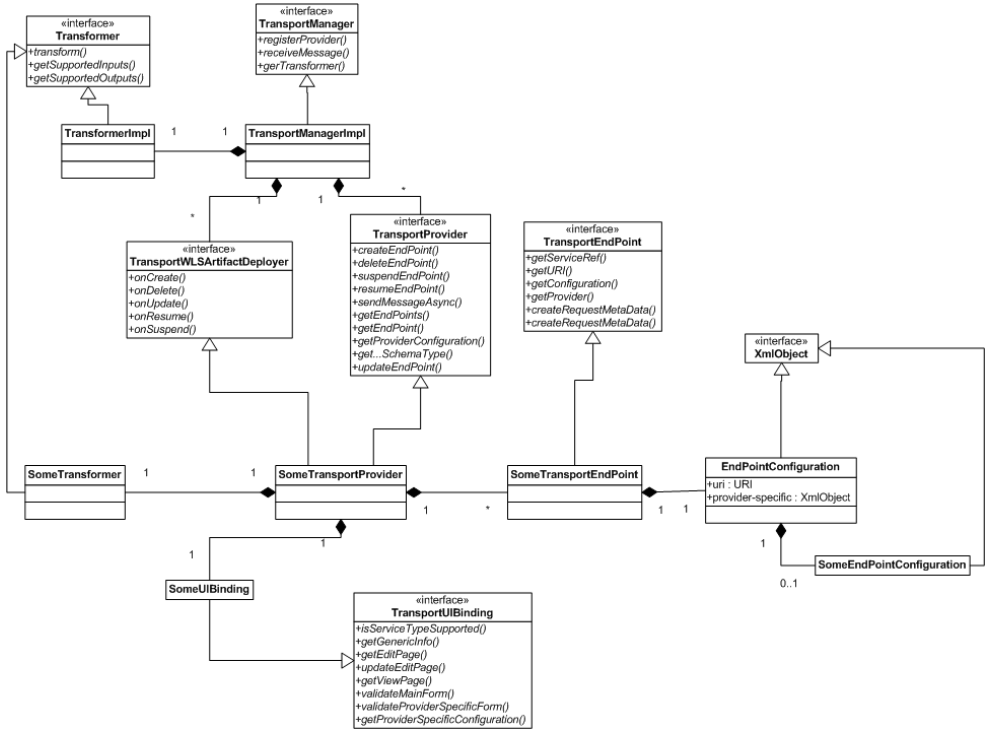
For more information, see Chapter 3, "Developing a Transport Provider" for a list of the required interfaces. A summary of the interfaces and classes provided by the Transport SDK are discussed in Chapter 5, "Transport SDK Interfaces and Classes." Detailed descriptions are provided in Javadoc for AquaLogic Service Bus Classes.

Design-Time Component

The design-time part of a custom transport provider consists of the user interface configuration. This configuration is called by the AquaLogic Service Bus Console when a new business or proxy service is being registered. Figure 2-2 shows a UML diagram that depicts the structure of the design time part of a transport provider. Some of the interfaces described in the diagram include:

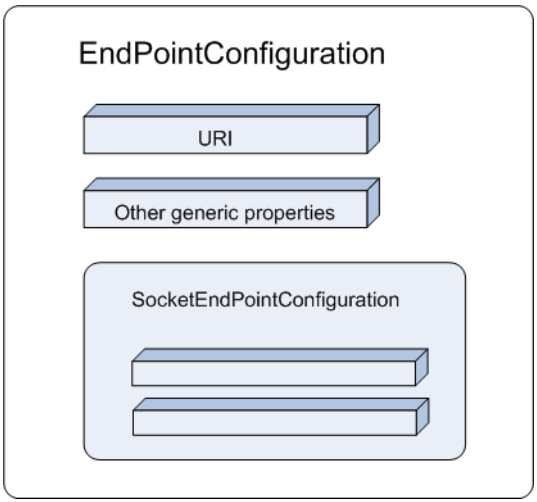
- **TransportManager** – A transport provider communicates with the transport manager through this interface. The implementation is not public.
- **TransportProvider** – Third parties must implement this interface. The TransportProvider keeps track of TransportEndpoint objects. TransportProvider also manages the life cycle of the endpoints. For example, you can suspend a transport endpoint, which is managed through the TransportProvider interface.
- **TransportUIBinding** – Helps the AquaLogic Service Bus Console render the transport specific pages.

Figure 2-2 Design Time UML Diagram



Note: Each transport endpoint has a configuration that consists of some properties that are generic to all endpoints of any transport provider, such as a URI, and some properties that are specific to endpoints of that provider only. Figure 2-3 shows the relationship between the shared endpoint configuration properties and transport provider specific configuration properties. See "Overview of Transport Endpoint Properties" on page 2-13 for more information.

Figure 2-3 EndPointConfiguration Properties



Runtime Component

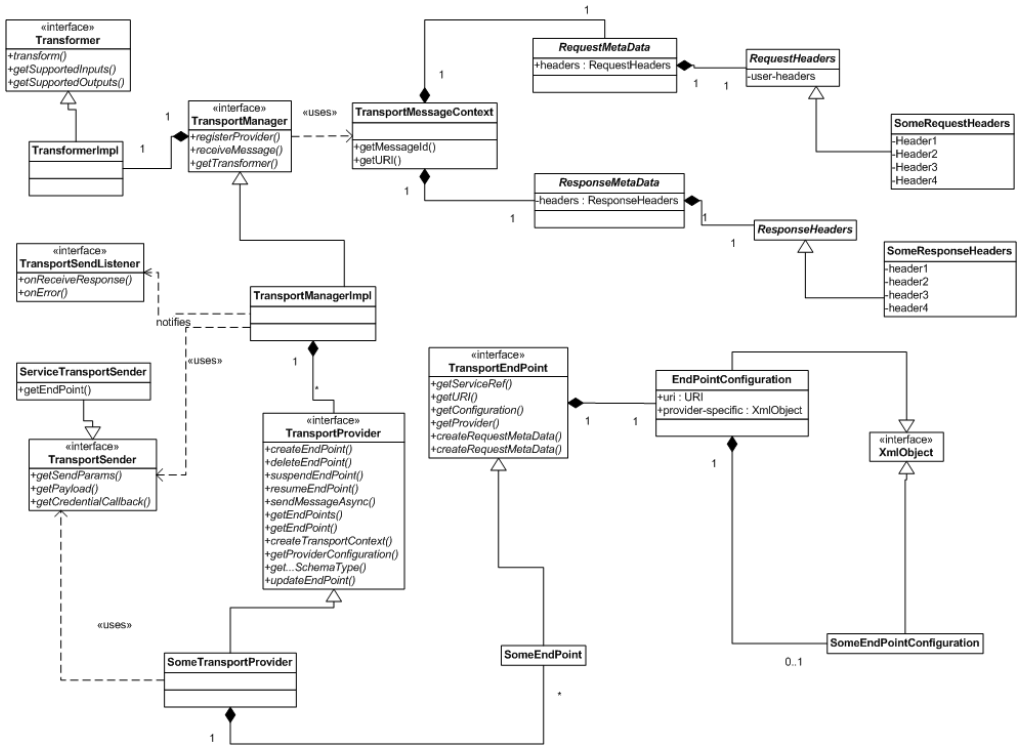
The runtime part of a custom transport provider:

- Receives messages and delivers them to the AquaLogic Service Bus runtime.
- Delivers outbound messages from AquaLogic Service Bus runtime to external services.

In the runtime framework, the transport provider calls the transport manager to acknowledge that an inbound message has been received. The transport message context contains the header and body of the inbound message. For the outbound message, there is a TransportSendListener and TransportSender. The transport provider retrieves the header and body from the message.

Figure 2-2 shows a UML diagram that depicts the structure of the runtime part of a transport provider.

Figure 2-4 Runtime UML Diagram



The Transaction Model

Before you develop a new transport provider using the Transport SDK, it is important to consider the transaction model for your message endpoints. This section discusses the transaction model used by AquaLogic Service Bus and how that model relates to the Transport SDK.

This section includes these topics:

- Overview of Transport Endpoint Properties
- Support for Synchronous Transactions

Overview of Transport Endpoint Properties

A transport endpoint is an AquaLogic Service Bus resource, such as a JMS proxy service, where messages are originated or targeted. In AquaLogic Service Bus, transport endpoints are managed

by protocol-specific transport providers, plug-in objects that manage the life cycle and runtime behavior of transport endpoints.

To understand the transactional model of AquaLogic Service Bus, it is useful to review some of the properties of service transport endpoints.

Transactional vs. Non-Transactional Endpoints

A given endpoint may or may not be transactional. A transactional endpoint has potential to start or enlist in a global transaction context when processing a message. The following examples illustrate how transactional properties vary depending on the endpoint:

- A JMS proxy service that uses the XA connection factory is a transactional endpoint. When the message is received, the container ensures that a transaction is started so that the message is processed in the context of a transaction.
- A Tuxedo proxy service may or may not be a transactional endpoint. A Tuxedo proxy service is only transactional if a transaction was started by the Tuxedo client application before the message is received.
- An HTTP proxy service endpoint is never transactional. In other words, inbound HTTP requests are never processed in the context of a transaction.

For detailed information on specific proxy services, see the *AquaLogic Service Bus User Guide*.

Supported Message Patterns

A given endpoint can use one of the following message patterns:

- **One Way** – No responses are expected. An example of a one-way endpoint is a JMS proxy service that does not expect a response.
- **Synchronous** – A request or response is implied. In this case, the response message is paired with the request message implicitly because no other traffic can occur on the transport channel from the time the request is issued until the time the response is received. In most cases, a synchronous message implies blocking calls for outbound requests. An EJB endpoint is synchronous. An HTTP endpoint is also synchronous: a new request cannot be sent until a response is received.
- **Asynchronous** – A request and response is implied. The response is correlated to a request through a transport-specific mechanism, such as a JMS transport and correlation through a JMSCorrelationID message property. For example, a JMS business service endpoint with request and response is asynchronous.

Support for Synchronous Transactions

The EJB and Tuxedo transports support synchronous transactions. Previously, the only transactional support in AquaLogic Service Bus was for the JMS transport, where transactions originated in and were bounded by the AquaLogic Service Bus domain. With the EJB and Tuxedo transports, transactions can originate outside of AquaLogic Service Bus and can pass through to external domains. Synchronous transactional transports support the following use cases:

Use Case 1 (Response Pipeline Processing)

Response pipeline processing is included in an incoming transaction when the inbound transport supports synchronous transactions. This case is supported when the inbound transport is paired with any other outbound transport, with the exception described in the note below.

Note: A deadlock situation occurs when the inbound transport is synchronous transactional and the outbound transport is asynchronous transactional. The deadlock occurs because the outbound request is not available until after the transaction commits, but the transaction was started externally and does not commit until AquaLogic Service Bus gets the response and returns. The transport manager recognizes this situation and avoids the deadlock by throwing a runtime error.

For example, if a synchronous transactional inbound endpoint is used, such as a Tuxedo proxy service, and the outbound endpoint is asynchronous transactional, such as a JMS proxy service, the outbound request does not commit the transaction until the response is received. It cannot be received until the external entity receives the request and processes it.

Also in this case, the AquaLogic Service Bus Publish action performed in the response pipeline is part of the transaction just like publish actions in the request pipeline are part of the transaction.

Note: There are several actions that can potentially participate in a transaction (in either the request or response pipeline). These include Publish, Service Callout, and Report actions.

For example, if an inbound Tuxedo transport is synchronous transactional, it can be committed only after the request and response pipeline have been completed. In this case, the transport manager transfers the transaction context from the inbound to the outbound thread. When the response thread is finished, the transaction control and outcome are returned to the invoking client.

Use Case 2 (Service Callout Processing)

AquaLogic Service Bus Service Callouts allow you to make a callout from a proxy service to another service. If a Service Callout action is made to a synchronous transactional transport, the

case of *Exactly Once* quality of service is supported in addition to *Best Effort* quality of service. *Exactly Once* 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. *Best Effort* means that each dispatch defines its own transactional context (if the transport is transactional). When *Best Effort* is specified, there is no reliable messaging and no elimination of duplicate messages; however, performance is optimized. See also "Working with TransportOptions" on page 3-19.

Callouts to synchronous transactional transports are optionally part of an existing transaction. For example, while the request pipeline is executing during a global transaction, Service Callouts are permitted to participate in the transaction. For example, if there is a callout to an EJB service, the service can participate in that transaction if it wants to.

For more information on Service Callouts, see "Service Callouts" in *Using the AquaLogic Service Bus Console*. For more information on message reliability, see the *AquaLogic Service Bus User Guide*.

Use Case 3 (Suspending Transactions)

Before calling the transport provider to send an outbound request the transport framework will suspend a transaction if the following conditions apply:

- The outbound service endpoint is transactional.
- There is a global XA transaction in progress.
- The quality of service is set to *Best Effort*.

The suspended transaction will resume, after the "send" operation is complete.

Use Case 4 (Multiple URIs)

If a given outbound service endpoint has multiple URIs associated with it, and is transactional, failover only occurs while the transaction, if any, is not marked for rollback. For example, if a URI is called, and the service returns an error, a failover is normally triggered. In this event, the transport framework detects that the transaction has been marked for rollback; therefore, the framework does not perform a failover to a different URI.

The Security Model

The Transport SDK allows customers and third-parties to plug in new transports into AquaLogic Service Bus. Within the AquaLogic Service Bus security model, transport providers are

considered trusted code. It is critical that transport provider implementations are carefully designed to avoid potential security threats by creating security holes. Although this document does not contain specific guidelines on how to develop secure transport providers, this section discusses the following security goals of the Transport SDK:

- Inbound Request Authentication
- Outbound Request Authentication
- Link-Level or Connection-Level Credentials
- Uniform Access Control to Proxy Services
- Identity Propagation and Credential Mapping

Inbound Request Authentication

Transport providers are free to implement whatever inbound authentication mechanisms are appropriate to that transport. For example: the HTTP transport provider supports these authentication methods:

- HTTP BASIC
- Custom authentication tokens carried in HTTP headers

The HTTPS transport provider supports SSL client authentication, in addition to the ones listed above. Both HTTP and HTTPS transport providers also support anonymous client requests.

The transport provider is responsible for implementing any applicable transport level authentication schemes, if any. If the transport provider authenticates the client it must make the client Subject object available to AquaLogic Service Bus by calling `TransportManager.receiveMessage()` within the scope of `weblogic.security.Security.runAs(subject)`. For information on this method, see `http://edocs.bea.com/wls/docs92/javadocs/weblogic/security/Security.html`.

Tip: For information on the Java class Subject, see `http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/Subject.html`.

The proxy will use this Subject in the following ways:

- During access control to the proxy service

- To populate the message context variable
  `$inbound/ctx:security/ctx:transportClient/*`
- As the input for identity propagation and credential mapping (unless there is also message-level client authentication)

If the transport provider does not support authentication, or if it supports anonymous requests, it must make sure the anonymous subject is on the thread before dispatching the request. Typically the transport provider will already be running as anonymous, but if this is not the case, then the provider must call:

```
Subject anonymous = SubjectUtils.getAnonymousUser()
Security.runAs(anonymous, action)
```

For information on SubjectUtils, see
http://edocs.bea.com/wls/docs92/javadocs/weblogic/security/SubjectUtils.html.

The transport provider is also responsible for providing any AquaLogic Service Bus Console configuration pages required to configure inbound client authentication.

The transport provider must clearly document its inbound authentication model.

Outbound Request Authentication

Transport providers are free to implement whatever outbound authentication schemes are appropriate to that transport. The transport SDK includes APIs to facilitate outbound username/password authentication, (two-way) SSL client authentication, and JAAS Subject authentication.

Outbound Username/Password Authentication

Outbound username/password authentication can be implemented by leveraging AquaLogic Service Bus service accounts. Service accounts are first-class, top-level AquaLogic Service Bus resources. Service accounts are created and managed in the AquaLogic Service Bus Console. Transport providers are free to design their transport-specific configuration to include references to service accounts. That way the transport provider can make use of the credential management mechanisms provided by AquaLogic Service Bus service accounts.

Transport providers don't have to worry about the details of service account configuration. There are three types of service accounts:

- **Static** – A static service account is configured with a fixed username/password.

- **Mapped** – A mapped service account contains a list of remote-users/remote-passwords and a map from local-users to remote-users. Mapped service accounts can optionally map the anonymous subject to a given remote user.
- **Pass-through** – A pass-through service account indicates that the username/password of the AquaLogic Service Bus client must be sent to the back-end.

An outbound endpoint can have a reference to a service account. The reference to the service account must be stored in the transport-specific endpoint configuration. When a proxy service routes a message to this outbound endpoint, the transport provider passes the service account reference to `CredentialCallback.getUsernamePasswordCredential(ref)`. AquaLogic Service Bus returns the username/password according to the service account configuration. This has the advantage of separating identity propagation and credential mapping configuration from the transport-specific details, simplifying the transport SDK. It also allows sharing this configuration. Any number of endpoints can reference the same service account.

Note: The CredentialCallback object is made available to the transport provider by calling `TransportSender.getCredentialCallback()`.

`CredentialCallback.getUsernamePasswordCredential()` returns a `weblogic.security.UsernameAndPassword` instance. This is a simple class which has methods to get the username and password. The username/password returned depends on the type of service account. If the service account is of type static, the fixed username/password is returned. If it is mapped, the client subject is used to look up the remote username/password. If it is pass-through, the client's username/password is returned.

Note: A mapped service account throws CredentialNotFoundException if:

- if there is no map for the inbound client, or
- the inbound security context is anonymous and there is no anonymous map

Note: In AquaLogic Service Bus 2.5, pass-through service accounts only work in two scenarios:

- When the proxy is of type HTTP or HTTPS and the inbound request contains a username/password in the HTTP Authorization header (for example, HTTP BASIC authentication)
- When the proxy is a WS-Security active intermediary and the inbound request includes a WS-Security Username token with a clear-text password

Otherwise the pass-through service account throws CredentialNotFoundException.

Outbound SSL Client Authentication (Two-Way SSL)

AquaLogic Service Bus also supports outbound SSL client authentication. In this case, the proxy making the outbound SSL request must be configured with a PKI key-pair for SSL. (This is done with a reference to a proxy service provider, the details are out of the scope of this document. For more information, see the *AquaLogic Service Bus User Guide*). To obtain the key-pair for SSL client authentication, the transport provider must call `CredentialCallback.getKeyPair()`. The HTTPS transport provider is an example of this.

Outbound JAAS Subject Authentication

Some transport providers send a serialized JAAS Subject on the wire as an authentication token. To obtain the inbound subject the transport provider must call `CredentialCallback.getSubject()`.

Note: The return value may be the anonymous subject.

Link-Level or Connection-Level Credentials

Some transports require credentials to connect to services. For example, FTP endpoints may be required to authenticate to the FTP server. Transport providers can make use of static service accounts to retrieve a username/password for establishing the connection. Note that mapped or pass-through service accounts cannot be used in this case because these connections are not made on behalf of a particular client request. If a transport provider decides to follow this approach, the endpoint must be configured with a reference to a service account. At runtime, the provider must call `TransportManagerHelper.getUsernamePasswordCredential()`, passing the reference to the static service account.

Uniform Access Control to Proxy Services

AquaLogic Service Bus enforces access control to proxy services for every inbound request. Transport providers are not required to enforce access control or to provide interfaces to manage the access control policy.

Note: The access control policy covers the majority of the use cases; however, a transport provider can implement its own access control mechanisms (in addition to the access control check done by AquaLogic Service Bus) if required for transport provider specific reasons. If that is the case, please contact your BEA representative. In general BEA recommends transport providers let AquaLogic Service Bus handle access control.

When access is denied, `TransportManager.receiveMessage()` throws an AccessNotAllowedException wrapped inside a TransportException. Transport providers are

responsible for checking the root-cause of the TransportException. A transport provider may do special error handling when the root cause is an AccessNotAllowedException. For example, the HTTP/S transport provider returns an HTTP 403 (forbidden) error code in this case.

Note: AquaLogic Service Bus makes the request headers available to the authorization providers for making access control decisions.

Identity Propagation and Credential Mapping

As explained in "Outbound Request Authentication" on page 2-18, AquaLogic Service Bus provides three types of service accounts. A transport provider can make use of service accounts to get access to the username/password for outbound authentication. A service account hides all of the details of identity propagation and credential mapping from AquaLogic Service Bus transport providers.

The Threading Model

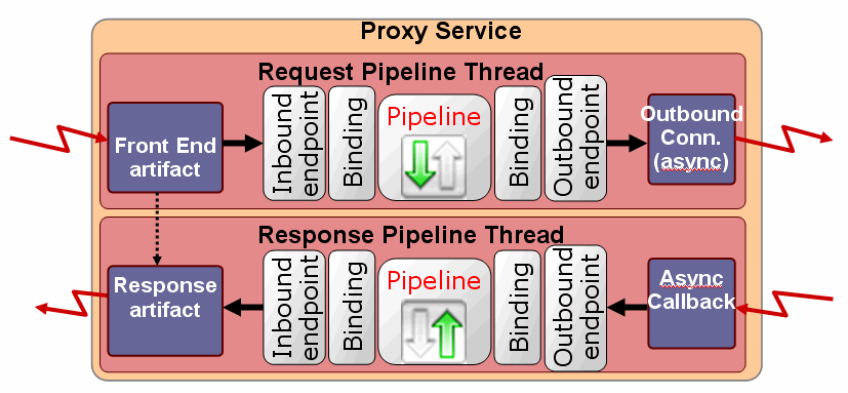
This section discusses the threading model used by AquaLogic Service Bus and how the model relates to the Transport SDK. This section includes these topics:

Overview

Figure 2-5 illustrates the AquaLogic Service Bus threading model for a hypothetical transport endpoint processing a single inbound message.

A front end artifact, such as a Servlet, is responsible for getting the inbound message. A request can be routed to an outbound endpoint and sent asynchronously. At this point, the thread is released. At some later point, a response is sent back to AquaLogic Service Bus (using a callback). The response is received, packaged, and handed to the AquaLogic Service Bus pipeline. Later, the pipeline notifies the inbound endpoint that the response is ready to be sent to the client. This processing is scalable because a thread is only tied up as long as it is needed.

Figure 2-5 Sample AquaLogic Service Bus Threading Model



Inbound Request Message Thread

The following actions occur in the same thread:

1. An inbound message is received by the front end artifact of the transport endpoint. This front end artifact could be, for example, an HTTP servlet or JMS message-driven bean instance.
2. The message is packaged into a TransportMessageContext object by the transport endpoint implementation and passed to the AquaLogic Service Bus runtime. For more information on the TransportMessageContext interface, see "Metadata and Header Representation for Request and Response Messages" on page 5-7.
3. The pipeline performs request pipeline actions configured for the proxy.
4. While processing the inbound message in AquaLogic Service Bus pipeline, in the same (request) thread, AquaLogic Service Bus runtime calls on the registered outbound transport endpoint, which may or may not be managed by the same provider, to deliver an outbound message to an external service.
5. At some later point, the external service asynchronously calls on the outbound endpoint to deliver the response message. The outbound endpoint must have been registered previously with a transport specific callback object.

Note: At this point, the initial request thread is released and placed back into the WebLogic Server thread pool for use by another request.

Outbound Response Message Thread

The following actions occur in the same thread:

1. The response message is packaged into a TransportMessageContext object and delivered back to AquaLogic Service Bus runtime for response processing. This processing occurs in a different thread than the request thread. This new thread is called the response thread.
2. After the response message is processed, AquaLogic Service Bus runtime calls on the InboundTransportMessageContext object to notify it that it is now time to send the response back to the original caller. For more information on the InboundTransportMessageContext interface, see "Metadata and Header Representation for Request and Response Messages" on page 5-7.

   If the transport provider does not have a native implementation of an asynchronous (non-blocking) outbound call, it still needs to deliver the response back to AquaLogic Service Bus runtime on a separate thread than that on which the inbound request message was received. To do this, it can execute the call in a blocking fashion in the request thread and then use a Transport SDK helper method to deliver the response back to AquaLogic Service Bus runtime.

   For example, the EJB transport provider does not have an asynchronous (non-blocking) outbound call. The underlying API is a blocking API. To work around this, the provider makes its blocking call, then schedules the response for processing with `TransportManagerHelper.schedule()`. For more information on the EJB transport provider, see "EJB Transport" in the *AquaLogic Service Bus User Guide*.

Support for Asynchrony

By design, the transport subsystem interacts asynchronously with AquaLogic Service Bus. The reason for this is that asynchronous behavior is more scalable, and therefore, more desirable than synchronous behavior. Rather than create two separate APIs, one for asynchronous and one for synchronous interaction, AquaLogic Service Bus runtime expects asynchronous interaction. It is up to the transport developer to work around this by a method such as posting a blocking call and posting the response in a callback. In any case, the response must be executed in a different thread from the request. See Table 2-2 for a list of AquaLogic Service Bus transport providers that support asynchronous outbound calls.

Table 2-2 Support for Asynchrony by AquaLogic Service Bus Transport Providers

Transport Provider	Supports Asynchronous Non-Blocking Outbound Calls
HTTP/HTTPS	Yes
JMS	Yes
File	N/A (One-way only. No response is sent.)
Email	N/A (One-way only. No response is sent.)
FTP	N/A (One-way only. No response is sent.)
Tuxedo	Yes
EJB	No
Socket	Yes

Publish and Service Callout Threading

The threading diagram shown in Figure 2-5 focuses on routing. The transport subsystem behaves the same way for AquaLogic Service Bus Publish and Service Callout actions which can occur in the middle of the request or response pipeline processing. These actions occur outside the scope of the transport subsystem and in the scope of an AquaLogic Service Bus pipeline. Therefore, some differences exist between the threading behavior of Publish and Service Callout actions and transport providers.

Note, however, the following cases:

- **Service Callout** – The pipeline processor will block the thread until the response arrives asynchronously. The blocked thread would then resume execution of the pipeline. The purpose is to bind variables that can later be used in pipeline actions to perform business logic. Therefore, these actions must block so that the business logic can be performed before the response comes back.
- **Publish** – The pipeline processor may or may not block the thread until the response arrives asynchronously. This thread then continues execution of the rest of the request or response pipeline processing.

Tip: A Service Callout action allows you to configure a synchronous (blocking) call to a proxy or business service that is already registered with AquaLogic Service Bus. Use a Publish

action to identify a target service for a message and configure how the message is packaged and sent to that service. For more information on Service Callout and Publish actions, see the AquaLogic Service Bus Console online help and the *AquaLogic Service Bus User Guide*.

Designing for Message Content

This section includes these topics:

Overview

Transport providers have their own native representation of message content. For example, HTTP transport uses `java.io.InputStream`, JMS has Message objects of various types, Tuxedo has buffers, and the WLS WebServices stack uses SAAJ. However, within the runtime of a proxy service, the native representation of content is the Message Context. While AquaLogic Service Bus supports some common conversion scenarios, such as InputStream to/from Message Context, this conversion between transport representation and the Message Context is ultimately the transport provider's responsibility.

In general, the Transport SDK is not concerned with converting directly between two different transport representations of content. However, if two transports use compatible representations and the content does not require re-encoding, the SDK may allow the source content to be passed-through directly (for example, passing a FileInputStream from an inbound File transport to an outbound HTTP transport). However, if the source content requires any sort of processing, it makes more sense to unmarshall the source content into the Message Context first and then use the standard mechanisms to generate content for the outgoing transport.

Sources and Transformers

Content is represented as an instance of the Source interface. Transport SDK interfaces that deal with message content, such as TransportSender and TransportMessageContext, all use the Source interface when passing message payloads. The requirements on a Source are minimal. A Source

must support push- and pull-based conversions to byte-based streams using the two methods defined in the base Source interface. A Source may or may not take into account various transformation options, such as character-set encoding, during serialization, as specified by the TransformOptions parameter.

While all Source objects must implement the base serialization interface, the underlying representation of the Source object's content is implementation specific. This allows for Source objects based on InputStreams, JMS Message objects, Strings, or whatever representation is most natural to a particular transport. Typically, Source implementations allow direct access to the underlying content, in addition to the base serialization methods. For example, StringSource, which internally uses a String object to store its content offers a `getString()` method to get at the internal data. The ultimate consumer of a Source can then extract the underlying content by calling these source-specific APIs and potentially avoid any serialization overheads.

Sources may also be transformed into other types of Sources using a Transformer object. If a Source consumer, such as a transport provider, is given a Source instance that it does not recognize, it can often transform it into a Source instance that it does recognize. The underlying content can then be extracted from that known Source using the source-specific APIs. However, often a transport provider simply serializes the content and send it using the base serialization methods. See also "Source and Transformer Classes and Interfaces" on page 5-4.

Sources and the MessageContext Object

Sources are the common content representation between the transport layer and the binding layer. The binding layer is the entity responsible for converting content between the Source representation used by the transport layer and the Message Context used by the pipeline runtime. How that conversion happens depends upon the type of service (its binding type) and the presence of attachments. While not strictly part of the Transport SDK, any transport provider that defines its own Source objects should be familiar with this conversion process.

When attachments are not present, the incoming Source represents just the core message content. The MessageContext is initialized by converting the received Source to a specific type of Source and then extracting the underlying content. For example, for XML-based services, the incoming Source is converted to an XmlObjectSource. The XmlObject is then extracted from the XmlObjectSource and used as the payload inside the `$body` context variable. SOAP services are similarly converted to XmlObjectSource except that the extracted XmlObject must be a SOAP Envelope so that the `<SOAP:Header>` and `<SOAP:Body>` elements can be extracted to initialize the `$header` and `$body` context variables.

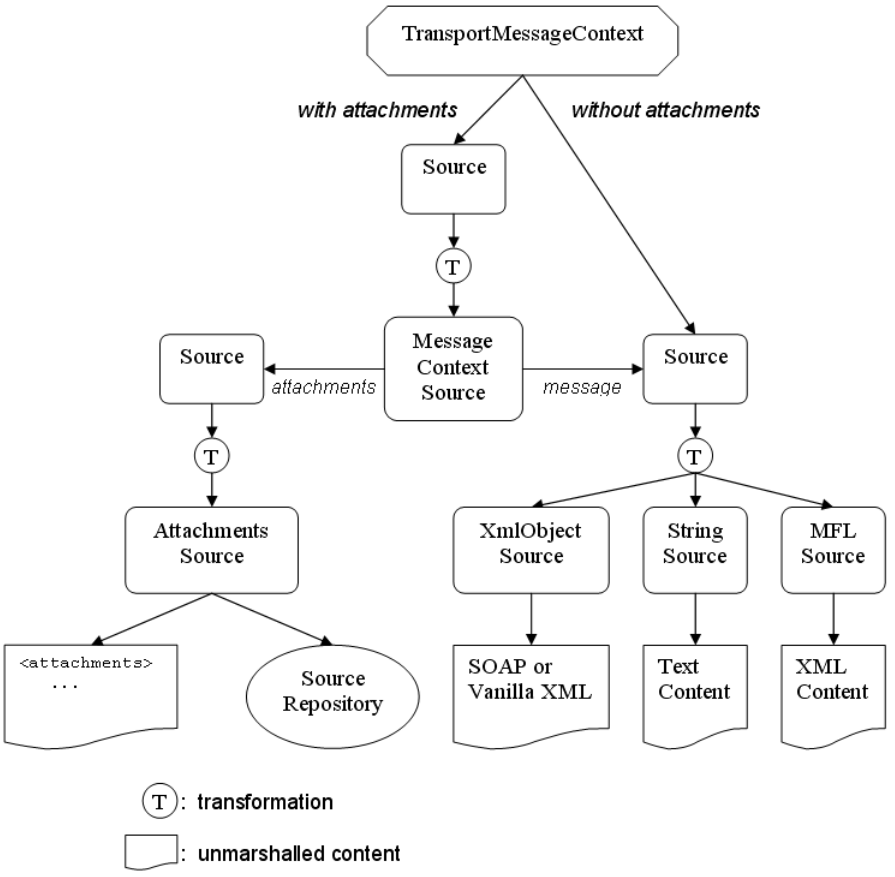
Below are the canonical Source types used for the set of defined service-types:

- **SOAP** – XmlObjectSource
- **XML** – XmlObjectSource
- **TEXT** – StringSource
- **MFL** – MFLSource

For binary services, no Source conversion is done. Instead, the Source is registered with a SourceRepository and the resulting `<binary-content/>` XML is used as the payload inside `$body`.

When attachments are present, the incoming Source is first converted to a MessageContextSource. From the MessageContextSource, two untyped Source objects are obtained, one representing the attachments and one representing the core message. The Source for the core message is handled as described previously. The Source representing attachments is converted to an AttachmentsSource. From the AttachmentsSource, XML is obtained and is used to initialize the `$attachments` context variable and a SourceRepository containing the registered Sources that represent any binary attachment content. This entire process is illustrated in Figure 2-6.

Figure 2-6 Flow of Attachments



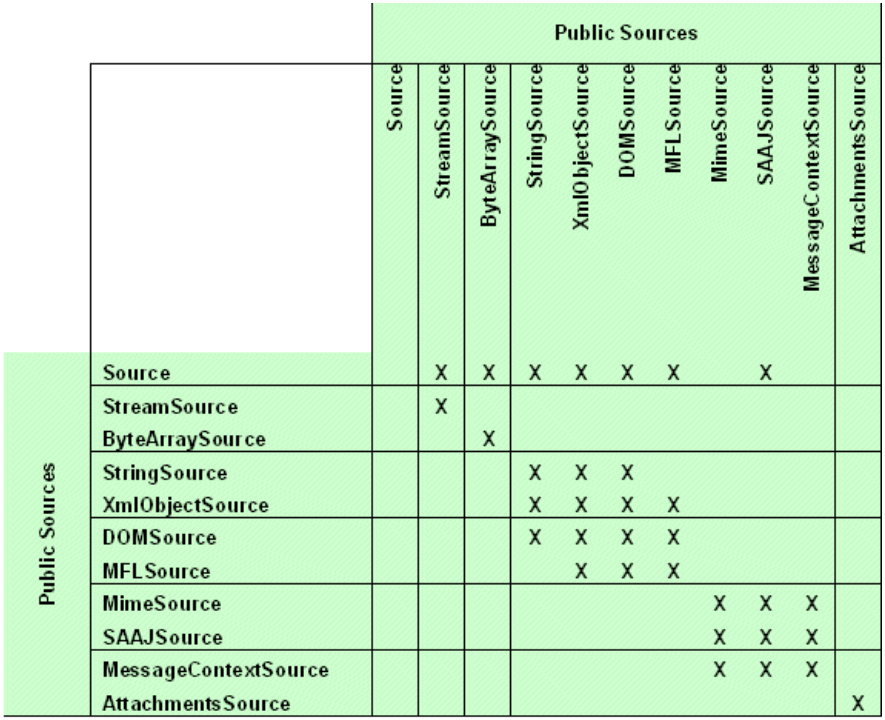
A similar conversion occurs when creating a Source from data in the MessageContext to be passed to the transport layer. The core message is represented by a Source instance that can be converted to the canonical Source for the service type. In most cases, the Source will already be an instance of the canonical Source, but not always. When attachments are present, the Source delivered to the transport layer will be a source that can be converted to an instance of MessageContextSource. If the transport provider supports Content-Type as a pre-defined transport header, then the delivered Source will likely be an instance of MessageContextSource. Otherwise, the delivered Source will likely be an instance of MimeSource, but this can also be converted to a MessageContextSource.

The reason for this difference is that transports that natively support Content-Type as a transport header require that the top-level MIME headers appear in the transport headers rather than in the payload. Examples of this are HTTP and Email. Transports that do not natively support Content-Type must have these top-level MIME headers as part of the payload, as the Content-Type header is critical for decoding a multipart MIME package.

Built-In Transformations

Below is a matrix showing the set of supported transformations offered by the built-in transformers. The column of Source names on the left indicates the initial Source type and the row of Source names on the top indicates the target Source type. An “X” in a given row R and column C means that it is possible to directly convert from initial Source R to target Source C. For example, there is some built-in transformer that handles converting a StringSource into an XmlObjectSource; however, there is no transformer that can convert a StringSource into an AttachmentsSource. Typically, these transformers take advantage of their knowledge of the internal data representation used by both Source types.

Figure 2-7 Transformation Matrix



		Public Sources										
		Source	StreamSource	ByteArraySource	StringSource	XmlObjectSource	DOMSource	MFLSource	MimeSource	SAAJSource	MessageContextSource	AttachmentsSource
Public Sources	Source		X	X	X	X	X	X		X		
	StreamSource		X									
	ByteArraySource			X								
	StringSource				X	X	X					
	XmlObjectSource				X	X	X	X				
	DOMSource				X	X	X	X				
	MFLSource					X	X	X				
	MimeSource								X	X	X	
	SAAJSource								X	X	X	
	MessageContextSource								X	X	X	
	AttachmentsSource											X

Of special interest is the very first row of "X" values in the matrix, as it represents supported transformations from arbitrary Sources into specific Sources. For example, while there is no transformer that specifically handles converting an XmlObjectSource to a ByteArraySource, there is a transformer that will handle converting any instance of Source to a ByteArraySource. These generic transformations are done without any knowledge of the initial Source type but instead rely on the base serialization methods that are implemented by all Sources: `getInputStream()` and `writeTo()`. So, although it is ultimately possible to convert an XmlObjectSource to a ByteArraySource, it is not done using any special knowledge of the internal details of XmlObjectSource.

Note: Many custom Sources implemented by Transports can be handled by these generic transformations, especially if the underlying data is an unstructured collection of bytes. For example, the File Transport uses a custom Source that pulls its content directly from a file on disk. However, as that data is just a set of bytes without structure, there is no

need to provide custom transformations to, for example, XmlObjectSource. The generic transformation Source XmlObjectSource can handle this custom FileSource using just the base serialization methods that all Sources must implement.

For more information, see "Source and Transformer Classes and Interfaces" on page 5-4.

CHAPTER 3

Developing a Transport Provider

The Transport SDK provides a layer of abstraction between transport protocols and the AquaLogic Service Bus runtime system. This layer of abstraction makes it possible to develop and plug in new transport providers to AquaLogic Service Bus. The Transport SDK interfaces provide this bridge between transport protocols, such as HTTP, and the AquaLogic Service Bus runtime.

Tip: Before beginning this chapter, be sure to review Chapter 2, "Design Considerations" first.

This chapter explains the basic steps involved in developing a custom transport provider. This chapter includes these topics:

- Development Roadmap
- Before You Begin
- Basic Development Steps
- Important Development Topics

Development Roadmap

The process of designing and building a custom transport provider is complex. This section offers a recommended path to follow as you develop your transport provider. Development of a custom transport provider breaks down into these basic stages:

- Planning
- Developing
- Packaging and Deploying

Planning

1. Decide if you really need to develop a custom transport provider. See "Do You Need to Develop a Custom Transport Provider?" on page 2-7.
2. Run and study the example socket transport provider. The source code for this provider is installed with AquaLogic Service Bus and is publicly available for you to examine and reuse. See Chapter 6, "Sample Socket Transport Provider."
3. Review Chapter 2, "Design Considerations." This chapter discusses the architecture of a transport provider and many aspects of transporter provider design, such as the security model and the threading model employed by transport providers.
4. Review the section "Before You Begin" on page 3-3.

Developing

The section "Basic Development Steps" on page 3-3 outlines the steps you need to take to develop a transport provider, including schema configurations and interface implementations.

The section "Important Development Topics" on page 3-14 discusses in detail several topics that you might need to refer to during the development cycle. This section includes detailed information on topics such as Handling Messages, Transforming Messages, Handling Errors, and others.

Packaging and Deploying

For detailed information on packaging and deploying a transport provider, see Chapter 4, "Deploying a Transport Provider."

Before You Begin

Before you begin to develop a custom transport provider, you need to consider and review a number of design issues, which include:

- Deciding if you really need to develop a custom transport provider. See "Do You Need to Develop a Custom Transport Provider?" on page 2-7.
- Deciding if your message endpoints are transactional or non-transactional. See "Transactional vs. Non-Transactional Endpoints" on page 2-14.
- Deciding if your message endpoints are one way, synchronous, or asynchronous. See "Supported Message Patterns" on page 2-14 and "Support for Synchronous Transactions" on page 2-15.
- Deciding on the security requirements for outgoing and incoming messages. See "The Security Model" on page 2-16.
- Understanding the threading model used by AquaLogic Service Bus. See "The Threading Model" on page 2-21.
- Understanding whether your transport provider will support synchronous or asynchronous outbound calls. See "Support for Asynchrony" on page 2-23.
- Reviewing the interfaces and classes provided with the Transport SDK, and becoming familiar with how they fit into the design time and runtime parts of a transport provider. See Chapter 5, "Transport SDK Interfaces and Classes."
- Understanding how to package and deploy a custom transport provider. See Chapter 4, "Deploying a Transport Provider."
- Reviewing the flow of method calls through the transport framework. See Appendix A, "UML Sequence Diagrams."

Basic Development Steps

The basic steps to follow when developing a custom transport provider include:

1. Review the Transport Framework Components.
2. Create a Directory Structure for Your Transport Project.
3. Create an XML Schema File for Transport-Specific Artifacts.
4. Define Transport-Specific Artifacts.

5. Define the XMLBean TransportProviderConfiguration.

6. Implement the Transport Provider User Interface.

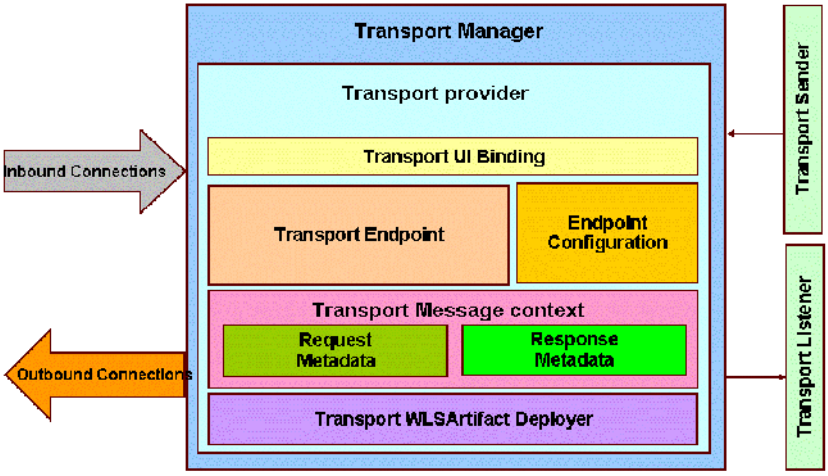
7. Implement the Runtime Interfaces.

8. Deploy the Transport Provider.

1. Review the Transport Framework Components

Figure 3-1 illustrates the components that you must implement and configure to create a custom transport provider. The transport manager controls and manages the registration of transport providers and handles communication with AquaLogic Service Bus. A transport provider manages the life cycle and runtime behavior of transport endpoints (resources where messages originate or are targeted). You use the Transport SDK to develop custom transport providers. Using the Transport SDK to develop a custom transport provider is the subject of this chapter.

Figure 3-1 Transport Subsystem Overview



The parts of the transport subsystem that you must implement and configure include:

- **Transport UI Bindings** – The user interface component for the transport provider. Related interfaces are summarized in "User Interface Configuration" on page 5-8.
- **Transport endpoint** – Responsible for sending and accepting messages. Related interfaces are summarized in "General Classes and Interfaces" on page 5-2.

- **Endpoint configuration** – Stores endpoint configurations. Related interfaces are listed in "Schema-Generated Interfaces" on page 5-1.
- **Transport message context** – Contains metadata for request and response headers and other parts of the message (inbound and outbound). See also "Source and Transformer Classes and Interfaces" on page 5-4 and "Metadata and Header Representation for Request and Response Messages" on page 5-7.
- **WLS Artifact deployer** – (optional) Deploys artifacts, such as servlets that receive and send messages.
- **Transport sender** – Retrieves metadata for the outbound message and the payload. Related interfaces are summarized in "Summary of General Interfaces" on page 5-3
- **Transport listener** – Allows the outbound endpoint to post the result of an outbound request to the rest of AquaLogic Service Bus. See also "Metadata and Header Representation for Request and Response Messages" on page 5-7.
- **Request/Response Metadata** – Related interfaces are summarized in "Metadata and Header Representation for Request and Response Messages" on page 5-7.

2. Create a Directory Structure for Your Transport Project

Before developing a new transport provider, take time to set up an appropriate directory structure for your project. The recommended approach is to copy the directory structure used for the sample socket transport provider. For a detailed description of this structure, see "Sample Location and Directory Structure" on page 6-5.

3. Create an XML Schema File for Transport-Specific Artifacts

Create an XML schema (`xsd`) file for transport-specific definitions. You can base this file on the schema file developed for the sample socket transport provider:

```
BEA_HOME/weblogic92/samples/servicebus/sample-transport/schemas/
SocketTransport.xsd
```

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

Note: The `SocketTransport.xsd` file imports the file `TransportCommon.xsd`. This file is the base schema definition file for service endpoint configurations. This file is located in `BEA_HOME/weblogic92/servicebus/lib/sb-public.jar`. You might want to review the contents of this file before continuing.

4. Define Transport-Specific Artifacts

Define XML schema for the following transport-specific artifacts in the XML schema file described in the previous section, "3. Create an XML Schema File for Transport-Specific Artifacts" on page 3-5.

- EndpointConfiguration
- RequestMetaDataXML
- ResponseMetaDataXML

Note: Only simple XML types are supported when defining transport provider-specific metadata and headers. For example, complex types with nested elements are not supported. Furthermore, an additional restriction is that there can be at most one header with a given name

Tip: Each of these schema definitions is converted into a corresponding Java file and compiled. You will find these converted Java source files for the sample socket transport provider in the directory:
`sample-transport/build/classes/com/bea/alsb/transports/sock/impl`

EndPointConfiguration

EndPointConfiguration is the base type for endpoint configuration, and describes the complete set of parameters necessary for the deployment and operation of an inbound or outbound endpoint. This configuration consists of generic and provider-specific parts. For more information on the EndPointConfiguration schema definition, refer to the documentation elements in the `TransportCommon.xsd` file.

You need to specify a provider-specific endpoint configuration in the schema file. Listing 3-1 shows an excerpt from the `SocketTransport.xsd`.

Listing 3-1 Sample SocketEndPointConfiguration Definition

```
<xs:complexType name="SocketEndpointConfiguration">
    <xs:annotation>
      <xs:documentation>
        SocketTransport - specific configuration
      </xs:documentation>
```

```
</xs:annotation>
<xs:sequence>
  <xs:choice>
    <xs:element name="outbound-properties"
                type="SocketOutboundPropertiesType"/>
    <xs:element name="inbound-properties"
                type="SocketInboundPropertiesType"/>
  </xs:choice>
  <xs:element name="request-response" type="xs:boolean">
    <xs:annotation>
      <xs:documentation>
        Whether the message pattern is synchronous
        request-response or one-way.
      </xs:documentation>
    </xs:annotation>
  </xs:element>
  ...
```

RequestMetaDataXML

It is required that each transport provider store metadata (message headers) in a Plain Old Java Object (POJO) and pass that to the pipeline. Examples of information that might be transmitted in the metadata are the Content-Type header, security information, or locale information. A RequestMetaData POJO is a generic object that extends the RequestMetaData abstract class and describes the message metadata of the incoming or outgoing request. The transport provider has to deliver the message metadata to AquaLogic Service Bus runtime in a RequestMetaData POJO. See also "Request and Response Metadata Handling" on page 3-16.

RequestMetaDataXML is an XML representation of the same RequestMetaData POJO. This XML representation uses Apache XML Bean technology. It is only needed by AquaLogic Service Bus runtime if or when processing of the message involves any actions in the pipeline that need an XML representation of the metadata, such as setting the entire metadata to a specified XML fragment on the outbound request.

You need to specify request metadata configuration in the schema file. Listing 3-2 shows an excerpt from the `SocketTransport.xsd`.

Listing 3-2 Sample SocketRequestMetaDataXML Definition

```
<xs:complexType name="SocketRequestMetaDataXML">
    <xs:annotation>
      <xs:documentation/>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="ts:RequestMetaDataXML">
        <xs:sequence>
          <xs:element name="client-host"
                      type="xs:string" minOccurs="0">
            <xs:annotation>
              <xs:documentation>
                Client host name
              </xs:documentation>
            </xs:annotation>
          </xs:element>
          <xs:element name="client-port" type="xs:int" minOccurs="0">
            <xs:annotation>
              <xs:documentation>Client port</xs:documentation>
            </xs:annotation>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

RequestHeadersXML

RequestHeadersXML is the base type for a set of inbound or outbound request headers. You need to specify the RequestHeadersXML configuration in the schema file. Listing 3-2 shows an excerpt from the `SocketTransport.xsd`.

Listing 3-3 Sample SocketRequestHeadersXML Definition

```
<xs:complexType name="SocketRequestHeadersXML">
    <xs:annotation>
      <xs:documentation/>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="ts:RequestHeadersXML">
        <xs:sequence>
          <xs:element name="message-count" type="xs:long" minOccurs="0">
            <xs:annotation>
              <xs:documentation>
                Number of messages passed till now.
              </xs:documentation>
            </xs:annotation>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

ResponseMetaDataXML

ResponseMetaDataXML is the base type for metadata for a response to an inbound or outbound message. You need to specify the ResponseMetaDataXML configuration in the schema file. Listing 3-2 shows an excerpt from the `SocketTransport.xsd`.

Listing 3-4 Sample SocketResponseMetaDataXML Definition

```
<xs:complexType name="SocketResponseMetaDataXML">
    <xs:complexContent>
      <xs:extension base="ts:ResponseMetaDataXML">
        <xs:sequence>
          <xs:element name="service-endpoint-host"
                      type="xs:string" minOccurs="0">
            <xs:annotation>
```

```
            <xs:documentation>
              Host name of the service end point connection.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="service-endpoint-ip"
                    type="xs:string" minOccurs="0">
          <xs:annotation>
            <xs:documentation>
              IP address of the service end point connection.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

ResponseHeadersXML

ResponseHeadersXML is the base type for a set of response headers. You need to specify the ResponseHeadersXML configuration in the schema file. Listing 3-2 shows an excerpt from the `SocketTransport.xsd`.

Listing 3-5 Sample SocketResponseHeadersXML Definition

```
<xs:complexType name="SocketResponseHeadersXML">
    <xs:annotation>
      <xs:documentation/>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="ts:ResponseHeadersXML"/>
    </xs:complexContent>
  </xs:complexType>
```

5. Define the XMLBean TransportProviderConfiguration

To configure the TransportProviderConfiguration XML bean, edit the transport provider configuration file. This XML file is located in the config directory in the transport provider root directory. See "Sample Location and Directory Structure" on page 6-5 for the location of this file (`SocketConfig.xml`) in the sample socket transport provider implementation.

- If proxy services can use your transport, set the `inbound-direction-supported` element to `true`.
- If business services use your transport, set the `outbound-direction-supported` element to `true`.
- If your transport is self-described, include an element `self-described` with the value set to `true`. A self-described transport is one whose services are responsible for describing their shape (schema or WSDL) based on their endpoint configuration.
- If you want to publish a tModel for your transport to UDDI, include an element `UDDI`. See the section "Publishing Proxy Services to a UDDI Registry" on page 3-24 for more info.

Tip: The schema for TransportProviderConfiguration is defined in `TransportCommon.xsd`, which is located in `BEA_HOME/weblogic92/servicebus/lib/sb-public.jar`. Refer to the schema file for more information.

---

6. Implement the Transport Provider User Interface

When you add a business or proxy service using the AquaLogic Service Bus Console, you select a transport provider from a menu in the Service Creation wizard. This menu includes the transport providers that are provided with AquaLogic Service Bus and any custom transport providers that were developed with the Transport SDK.

This section discusses the Transport SDK API components that bind your custom transport provider to the AquaLogic Service Bus Console user interface. You must implement these APIs to connect your provider to the user interface.

Tip: This section assumes that you are familiar with the Service Creation Wizard. See "Configuring the Socket Transport Sample" on page 6-8 for a detailed, illustrated example.

---

1. After a user creates a new service and chooses the Service Type in the Service Creation wizard, she must then select an appropriate transport provider for the Service Type. To validate the selection, the wizard calls the following method of the TransportUIBinding interface:

   ```
   public boolean isServiceTypeSupported(BindingTypeInfo binding)
   ```

   This method determines if the transport provider is suitable for the selected Service Type.

2. After a valid transport provider is selected, the user enters an endpoint URI. To validate this URI, the wizard calls the following method of the TransportUIBinding interface:

   ```
   public TransportUIError[] validateMainForm(TransportEditField[] fields)
   ```

3. Next, the wizard displays the transport-specific configuration page. To render this page, the wizard calls the following method of the TransportUIBinding interface:

   ```
   public TransportEditField[] getEditPage(EndPointConfiguration config,
   BindingTypeInfo binding) throws TransportException
   ```

   The Transport SDK offers a set of TransportUIObjects that represent fields in the configuration page. For example, you can add text boxes, checkboxes, and other types of UI elements. Use the TransportUIFactory to create them. After creation use the same factory to specify additional properties and obtain TransportEditField objects that can be displayed by the Service Creation wizard.

   For a complete list of the available TransportUIObjects, refer to the Javadoc.

   Tip: You can associate events with most of the UI fields. An event acts like a callback mechanism for the TransportUIBinding class and lets you refresh, validate, and update the configuration page. When an event is triggered, the wizard calls the method:

   ```
   updateEditPage(TransportEditField[] fields, String name) throws
   TransportException
   ```

   ---

4. When the user finishes the transport configuration, the wizard calls the validation method:

   ```
   TransportUIError[] validateProviderSpecificForm(TransportEditField[]
   fields)
   ```

5. Finally, the user saves the new service, and the wizard displays a summary of the configuration. To implement the summary display, you need to implement the method:

   ```
   public TransportViewField[] getViewPage(EndPointConfiguration config)
   throws TransportException
   ```

6. After the service is saved, the transport manager calls the following method of the TransportProvider class:

   ```
   void validateEndPointConfiguration(TransportValidationContext context)
   ```

   If no error is reported, a new endpoint is created. The Transport Manager then calls the method:

   ```
   TransportEndPoint createEndPoint(EndPointOperations.Create context)
   throws TransportException
   ```

   If this method returns successfully, the new service is listed in the AquaLogic Service Bus Console and the underlying transport configuration is associated with an endpoint on the TransportProvider.

   Note: The endpoint configuration is saved in the AquaLogic Service Bus session and does not need to be persisted or recovered in case of a server restart by the transport provider.

7. Once the session is activated, you must deploy the endpoint to start processing requests. See "When to Implement TransportWLSArtifactDeployer" on page 3-26 and "Deploying to a Cluster" on page 4-3 to learn more about deploying an endpoint and processing requests.

Tip: For the sample socket transport provider, you can find the implementations of these interfaces in the `sample-transport/src` directory.

---

7. Implement the Runtime Interfaces

A new custom transport provider must implement the following runtime interfaces. For a summary of the Transport SDK interfaces and related classes, see Chapter 5, "Transport SDK Interfaces and Classes." For detailed information on interfaces and classes, see the AquaLogic Service Bus Javadoc description.

Tip: For the sample socket transport provider, you can find the implementations of these interfaces in the `sample-transport/src` directory.

- TransportProvider
- TransportWLSArtifactDeployer

  Note: Only implement the TransportWLSArtifactDeployer interface if the transport provider needs to deploy WebLogic Server-related artifacts, such as EAR/WAR/JAR

files, that go into a WebLogic Server change list at the time of endpoint creation. For more information, see "When to Implement TransportWLSArtifactDeployer" on page 3-26.

- TransportEndPoint
- InboundTransportMessageContext
- OutboundTransportMessageContext
- Transformer

  Note: Only implement the Transformer interface if the transport provider needs to work with non-standard payload bindings, for example, anything other than Stream, DOM, SAX, or XMLBean. For more information, see "Transforming Messages" on page 3-18.

8. Deploy the Transport Provider

For detailed information on deployment, see Chapter 4, "Deploying a Transport Provider."

Important Development Topics

This section discusses several topics that you will encounter while developing a custom transport provider. These topics include:

- Handling Messages
- Transforming Messages
- Working with TransportOptions
- Handling Errors
- Publishing Proxy Services to a UDDI Registry
- When to Implement TransportWLSArtifactDeployer

Handling Messages

This section discusses message handling in transport providers and includes these topics:

- Overview
- Sending and Receiving Message Data

- Request and Response Metadata Handling
- Character Set Encoding
- Co-Located Calls
- Returning Outbound Responses to AquaLogic Service Bus Runtime

Overview

The Transport SDK features a flexible representation of message payloads. All Transport SDK APIs dealing with payload use the Source interface to represent message content.

The Source-derived message types provided with the Transport SDK include:

- StreamSource
- ByteArraySource
- StringSource
- XmlObjectSource
- DOMSource
- MFLSource
- SAAJSource
- MimeSource

Note: StreamSource is a single use source; that is, it implements the marker interface SingleUseSource. With the other Sources, you can get the input stream from the source multiple times. Each time the Source object gets the input stream from the beginning. With a SingleUseSource, you can only get the input stream once. Once the input is consumed, it is gone (for example, a stream from a network socket); however, AquaLogic Service Bus buffers the input from a SingleUseSource, essentially keeping a copy of all of its data.

If you implement a Source class for your transport provider, you need to determine whether you can re-get the input stream from the beginning. If the nature of the input stream is that it can only be consumed once, it is recommended that your Source class implement the marker interface SingleUseStream.

The Transport SDK provides a set of Transformers to convert between Source objects. You can implement new transformations, as needed, as long as they support transformations to/from a set

of canonical representations. See "Transforming Messages" on page 3-18 for more information. See also "Designing for Message Content" on page 2-25.

Sending and Receiving Message Data

When implementing inbound endpoints to deliver the inbound message to AquaLogic Service Bus runtime, you need to call `TransportManager.receiveMessage()`. The transport provider is free to expose the incoming message payload in either one of the standard Source-derived objects, such as stream, DOM or SAX, or a custom one.

If AquaLogic Service Bus needs to send a response message back to the client that sent the request, it will call methods `setResponseMetaData()` and `setResponsePayload()` followed by `close()` on InboundTransportMessageContext to indicate that the response is ready to be sent back. When AquaLogic Service Bus runtime calls the inbound transport message context `close()` method, this will be done from a different thread than that on which the inbound request message was received. The transport provider should be aware of this as it may affect the semantics of transactions. Also, the transport provider cannot attempt to access the response payload and/or metadata until `close()` method has been called.

Request and Response Metadata Handling

It is required that each transport provider store metadata and headers in a Plain Old Java Object (POJO) and pass that to the pipeline. There are some cases where AquaLogic Service Bus requires an XMLBean. In these cases, you need to implement a conversion from POJO to XMLBean using the API.

The following are the methods you must provide to convert from a POJO to XML:

- `RequestHeaders.toXML()`
- `RequestMetaData<T>.toXML()`
- `ResponseHeaders.toXML()`
- `ResponseMetaData<T>.toXML()`

For the reverse direction (XML to POJO) you need to implement:

- `TransportEndPoint.createRequestMetaData(RequestMetaDataXML)`
- `InboundTransportMessageContext.createResponseMetaData(ResponseMetaDataXML)`

Character Set Encoding

Each transport provider is responsible for specifying the character set encoding of the incoming message payload to AquaLogic Service Bus. For outgoing messages (outbound request and

inbound response), the transport provider is responsible for telling AquaLogic Service Bus what character set encoding to use for the outgoing payload. The character-set encoding is specified in request and response metadata.

In virtually every case, the character-set encoding that the transport is responsible for inserting into the metadata is exactly the encoding that is statically specified in the service configuration. One of the few exceptions to this is HTTP transport, which inspects Content-Type for any "charset" parameters and overrides any encoding configured in the service. This is necessary in order to conform to HTTP specifications. Other transport protocols may need to handle similar issues.

Tip: In general, the encoding for a service is fixed. If someone sends an UTF-16 encoded message to a proxy that is specified to be SHIFT_JIS, then that is generally considered to be an error. Transport providers should not need to inspect the message simply to determine encoding.

For outgoing messages, the transport provider tells AquaLogic Service Bus what encoding it requires for the outbound request, and AquaLogic Service Bus performs the conversion if necessary.

Transports should always rely on this encoding for outgoing messages and should not assume that it is the same as the encoding specified in the service configuration. If there is a discrepancy, the transport can choose to allow it, but others could consider it an error and throw an exception. Also the transport has the additional option of leaving the encoding element blank. That leaves the pipeline free to specify the encoding (for example, via pass-through).

Co-Located Calls

If a given transport provider supports proxy service endpoints, it is possible to configure the request pipeline such that there is a routing step that routes to that provider's proxy service. Furthermore there could be a Publish or a Service Callout action that sends a message to a proxy service instead of a business service. This use case is referred to as co-located calls.

The transport provider needs to be aware of co-located calls, and handle them accordingly. Depending on the nature of the proxy service endpoint implementation, the transport provider may choose to optimize the invocation such that this call bypasses the entire transport communication stack and any inbound authentication/authorization, and instead is a direct call that effectively calls `TransportManager.receiveMessage()` immediately.

Tip: AquaLogic Service Bus has implemented this optimization with the HTTP, File, Email and FTP transport providers. The JMS provider does not use this optimization due to the desire to separate the transactional semantics of send operation versus receive operations.

If you want to use this optimization in a custom transport provider, you need to extend the CoLocatedTransportMessageContext class and call its `send()` method when `TransportProvider.sendMessageAsync()` is invoked.

Returning Outbound Responses to AquaLogic Service Bus Runtime

When AquaLogic Service Bus runtime sends a message to an outbound endpoint and there is a response message to be returned, the transport provider must return this response asynchronously. That means `TransportSendListener.onReceiveResponse()` or `TransportSendListener.onError()` methods need to be called from a different thread than the one on which `TransportProvider.sendMessageAsync()` was called.

If the transport provider has a built-in mechanism by which the response arrives asynchronously, such as responses to JMS requests or HTTP requests when the async response option is used, it happens naturally. However, if the transport provider has no built-in mechanism for retrieving responses asynchronously, it can execute the outbound request in a blocking fashion and then schedule a new worker thread using the `TransportManagerHelper.schedule()` method, in which the response is posted to the TransportSendListener.

Transforming Messages

When AquaLogic Service Bus needs to set either the request payload to an outbound message or the response payload to an inbound message, it asks the transport provider to do so through an object derived from the Source interface. The transport provider then needs to decide what representation the underlying transport layer requires and use the `Transformer.transform()` method to translate the Source object into the desired source.

Tip: For more information on message transformation, see "Designing for Message Content" on page 2-25. For a list of built-in transformations, see "Built-In Transformations" on page 2-29 and "Source and Transformer Classes and Interfaces" on page 5-4.

A custom transport provider can support new kinds of transformations. Suppose a transport provider needs to work with a DOM object in order to send the outbound message. When called with `setRequestPayload(Source src)`, the transport provider needs to call the method:

```
TransportManagerHelper.getTransportManager().getTransformer().
transform(src, DOMSource.class, transformOptions).
```

The return value of the method gives a DOMSource, which can then be used to retrieve the DOM node.

Note: If the transport provider requires a stream, there is a shortcut: each Source object supports transformation to stream natively.

You can add new transformations to a custom transport provider. For example, suppose you want to add a new kind of Source-derived class, called XYZSource. For performance reasons, transport providers are encouraged to provide conversions from XYZSource to one of the two canonical Source objects, XmlObjectSource and StreamSource when applicable. Without such transformation, generic transformers will be used, which rely on the StreamSource representation of XYZSource. Of course, if XYZSource is a simple byte-based Source with no internal structure, then relying on the generic transformers is usually sufficient. Note that any custom transformer that is registered with TransportManager is assumed to be thread-safe and stateless.

To support attachments, the transport provider has three options:

- The Source returned by TransportMessageContext must be an instance of MessageContextSource. A limitation of this option is that MessageContextSource requires that the content has already been partitioned into a core-message Source and an attachments Source.
- The Source is an instance of MimeSource and the Headers objects contain a multipart Content-Type header.
- The Content-Type is a pre-defined header for the transport provider with the specific value `multipart/related`. Both HTTP(S) and Email transports rely on this third option for supporting attachments.

Working with TransportOptions

A TransportOptions object is used to supply options for sending or receiving a message. A TransportOptions object is passed from the transport provider to the transport manager on inbound messages. On outbound messages, a TransportOptions object is passed from the AquaLogic Service Bus runtime to the transport manager, and finally to the transport provider.

This section includes these topics:

- Inbound Processing
- Outbound Processing

- Request Mode

Inbound Processing

The transport provider supplies these parameters to `TransportManager.receiveMessage()`:

- **QOS** – Specifies exactly-once or best-effort quality of service. Exactly-once quality of service is specified when the incoming message is transactional.
- **Throw On Error** – If this flag is set, an exception is thrown to the callee of method `TransportManager.receiveMessage()` when an error occurs during the AquaLogic Service Bus pipeline processing. The options for throwing the exception include: throw the exception back to the inbound message or create a response message from the error and notify the inbound message with the response message. Typically, you set **Throw On Error** to true when QOS is exactly-once (for transactional messages).

  For example, JMS/XA sets this flag to true to throw the exception in the same request thread, so it can mark the exception for rollback. HTTP sets the flag to false, because there is no retry mechanism. The error is converted to a status code and a response message is returned.
- **Any transport-specific opaque data** – Opaque data can be any data that is set by the transport provider and passed through the pipeline to the outbound call. This technique provides optimizes performance when the same transport is used on inbound and outbound. The opaque data is passed directly through the pipeline from the inbound transport to the outbound transport. For example, the HTTP/S transport provider can pass the username and password directly from the inbound to the outbound to efficiently support identity pass-through propagation.

Outbound Processing

For outbound processing, the AquaLogic Service Bus runtime supplies these parameters to the transport manager, which uses some of the parameters internally and propagates some parameters to `TransportProvider.sendMessageAsync()`. These parameters include:

- **QOS** – Specifies whether or not "exactly-once" quality of service can be achieved. For example, for HTTP, if quality of service is set to exactly once, the HTTP call is blocking. If it is set to best effort, it is a non-blocking HTTP call.
- **Mode** – Specifies one-way or request response. See also "Transport Provider Modes" on page 2-6.
- **URI, Retry Interval, and Count** – The transport provider uses the URI to initialize the outbound transport connection. For example, the HTTP transport provider uses the URI

when instantiating a new HttpURLConnection. The transport provider is not required to use Retry Interval and Count.

- **OperationName** – The transport provider can use OperationName if it needs to know what outbound Web Service is being used. The transport manager uses this parameter to keep track of monitoring statistics.
- **Any transport-specific opaque data** – An example of transport-specific opaque data is the value of the "Authorization" header for HTTP/S.

Request Mode

The request mode is defined as an enumeration with two values: `REQUEST_ONLY` (also called "one-way") and `REQUEST_RESPONSE`. These modes are interpreted as follows for requests and responses:

- On outbound requests, the pipeline indicates the mode through TransportOptions and the transport provider must honor the mode.
- On inbound requests, the pipeline knows the mode and closes the inbound request and does not send a response if it computes the mode `REQUEST_ONLY`.
- If a response is sent by the pipeline, then there is a response even if the response is empty.
- For transports that are inherently one-way, the transport must not specify response metadata.

Handling Errors

There are three different use cases to consider with respect to the effect runtime exceptions have on the transactional model. These cases include:

- Case 1: The exception occurs somewhere in the request pipeline but before the outbound call to the business service.
- Case 2: The exception occurs during the business service call.
- Case 3: The exception occurs sometime after the business service call in the response pipeline.

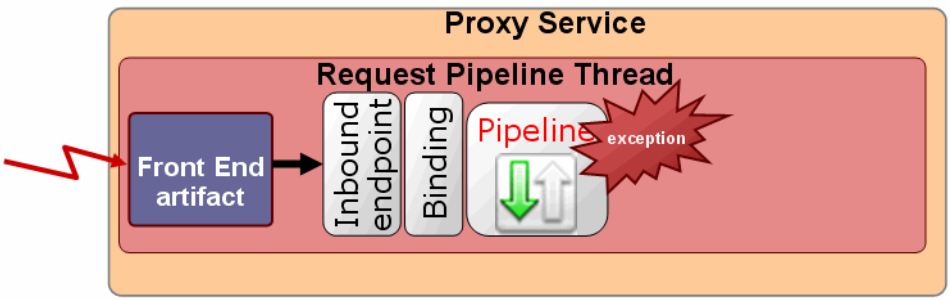
These cases are discussed in this section.

Case 1

The exception occurs somewhere in the request pipeline but before the outbound call to the business service, as shown in Figure 3-2. For example, executing a specific XQuery against the contents of the request message raises an exception.

If there is a user-configured error handler configured for the request pipeline, the error will be handled according to the user configuration. Otherwise, the proxy service will either catch an exception when calling `TransportManager.receiveMessage()` or will be notified in the `InboundTransportMessageContext.close()` method of the error through response metadata, based on the transport options passed as an argument to the `receiveMessage()` call. If the proxy service indicates that the exception should be thrown, surround `receiveMessage()` with a try/catch clause and mark the transaction for rollback.

Figure 3-2 Error Case 1



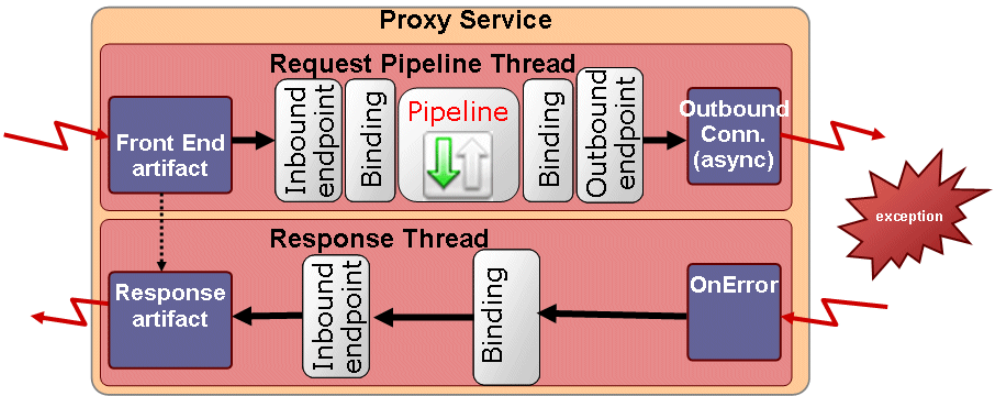
Case 2

The exception occurs during the business service call, as shown in Figure 3-3. The outbound transport provider either:

- Throws an exception from `TransportProvider.sendMessageAsync()`. For example, the outbound provider throws an exception if there was an error while establishing a socket connection to external service. This situation could occur if the business service cannot be called because of an incorrect URL, a faulty connection, or other reasons. In these cases, the transport provider must raise an exception.
- Notifies the listener through `TransportSendListener.onError()`. For example, if the business service was called, but the call resulted in an error (such as a SOAP fault), the transport provider needs to call `TransportSendListener.onError()` instead of raising an exception.

In the first instance, the exception handling is the same as that described in Case 1. In the second instance, if there is an error handler configured for the response pipeline, the error is handled according to the user configuration. Otherwise, the exception is propagated back to the proxy service endpoint in `InboundTransportMessageContext.close()` through the response metadata.

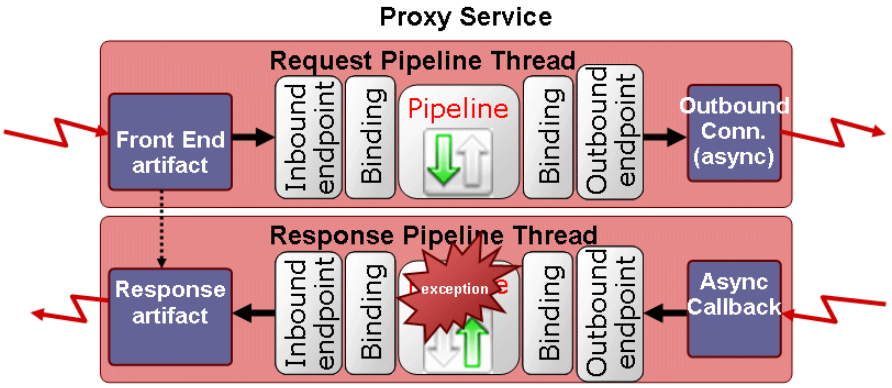
Figure 3-3 Error Case 2



Case 3

The exception occurs sometime after the business service call in the response pipeline, as shown in Figure 3-4. Again, in the absence of a user-defined error handler for the response pipeline, the proxy service endpoint is notified of the error with the `InboundTransportMessageContext.close()` method with appropriate response metadata. If the inbound transport endpoint is a synchronous transactional endpoint, it is guaranteed that the transaction that was active at the time request was received is still active and it may be rolled back. If the inbound endpoint is not transactional or not synchronous, there is not an inbound transactional context to roll back, so some other action might need to be performed.

Figure 3-4 Error Case 3



Publishing Proxy Services to a UDDI Registry

Universal Description, Discovery, and Integration (UDDI) is a standard mechanism for describing and locating Web services across the internet. You might want to publish proxy services based on a custom transport provider to a UDDI registry. This allows proxy services to be imported into another AquaLogic Service Bus server in a different domain as the one hosting the business service.

To publish a proxy service, the transport provider needs to define a tModel that represents the transport type in the "UDDI" section of TransportProviderConfiguration XML schema definition. (For more information on the schema-generated interfaces, see "Schema-Generated Interfaces" on page 5-1.)

This tModel must contain a CategoryBag with a keyedReference whose tModelKey is set to the UDDI Types Category System and keyValue is "transport." You are required to provide only the UDDI V3 tModel key for this tModel.

If UDDI already defines a tModel for this transport type, it is recommended that the definition be copied and pasted into the UDDI section.

An example of such a tModel is provided in Listing 3-6.

Listing 3-6 Example tModel

```
<?xml version="1.0" encoding="UTF-8"?>
<ProviderConfiguration xmlns="http://www.bea.com/wli/sb/transports">
```

```
  …
  …
  <UDDI>
    <TModelDefinition>
      <tModel tModelKey="uddi:bea.uddi.org:transport:socket">
        <name>uddi-org:socket</name>
        <description>Socket transport based webservice</description>
        <overviewDoc>
          <overviewURL useType="text">
            http://www.bea.com/wli/sb/UDDIMapping#socket
          </overviewURL>
        </overviewDoc>
        <categoryBag>
          <keyedReference keyName="uddi-org:types:transport"
                          keyValue="transport"
                          tModelKey="uddi:uddi.org:categorization:types"/>
        </categoryBag>
      </tModel>
    </TModelDefinition>
  </UDDI>
</ProviderConfiguration>
```

If UDDI does not already define a tModel for this transport type, AquaLogic Service Bus can publish the tModel you define here to configured registries. When a UDDI registry is configured to AquaLogic Service Bus, the "Load tModels into Registry" option can be specified. That option causes all of the tModels of AquaLogic Service Bus, including the tModels for custom transport providers, to be published to the UDDI registry. After deploying your transport provider, you can update your UDDI registry configuration to publish your tModel.

During UDDI export, `TransportProvider.getBusinessServicePropertiesForProxy(Ref)` is called and the resulting map is published to the UDDI registry. The provider is responsible for making sure to preserve all important properties of the business service in the map so that during the UDDI import process the business service definition can be correctly constructed without loss of information.

During UDDI import, `TransportProvider.getProviderSpecificConfiguration(Map)` is called and the result is an XmlObject that conforms to the provider-specific endpoint configuration schema, which goes into the service definition.

Tip: OASIS, the Organization for the Advancement of Structured Information Standards, is responsible for creating the UDDI standard. To read more about UDDI, including the full technical specification, go to:

`http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec`

When to Implement TransportWLSArtifactDeployer

Two sets of transport provider interfaces are provided that deal with individual service registration. TransportProvider has methods like create/update/delete/suspend/resume and TransportWLSArtifactDeployer has the same methods. This section discusses these two implementations and explains when you need to implement TransportWLSArtifactDeployer.

Only implement TransportWLSArtifactDeployer if your provider needs to make changes to WebLogic Server artifacts in the AquaLogic Service Bus domain. The methods on TransportWLSArtifactDeployer are only called on an Administration Server. In this case, changes are made through the DomainMBean argument that is passed in, and then the changes are propagated to the entire cluster.

The TransportProvider methods are called on all servers (Administration and Managed Servers) in the domain. Because you cannot make changes to AquaLogic Service Bus domain artifacts on a managed server, the purpose of the method calls on TransportProvider is to update its internal data structures only.

When a given Transport provider implements the TransportWLSArtifactDeployer interface, the methods on TransportWLSArtifactDeployer are called before the corresponding methods on TransportProvider. For example, `TransportWLSArtifactDeployer.onCreate()` is called before `TransportProvider.createEndPoint()`.

For more information on TransportWLSArtifactDeployer, see "Summary of General Interfaces" on page 5-3.

CHAPTER 4

Deploying a Transport Provider

This chapter explains how to package and deploy a custom transport provider and includes these topics:

- Packaging the Transport Provider
- Deploying the Transport Provider
- Undeploying a Transport Provider
- Deploying to a Cluster

Packaging the Transport Provider

It is recommended that you develop your custom transport provider as a self-contained EAR file. You can then deploy the EAR with the AquaLogic Service Bus Kernel EAR and other AquaLogic Service Bus related applications.

Tip: The sample socket transport provider example illustrates how a transport provider is organized and deployed. See Chapter 6, "Sample Socket Transport Provider" for more information.

Each transport provider consists of two distinct parts:

- **Configuration** – The configuration part of a transport provider is used by AquaLogic Service Bus Console to register endpoints with the transport provider. This configuration

behavior is provided by the implementation of the UI interfaces. See "User Interface Configuration" on page 5-8.

- **Runtime** – The runtime part of a transport provider implements the business logic of sending and receiving messages.

Tip: A best practice is to package the transport provider so that the configuration and runtime parts are placed in separate deployment units. This practice makes cluster deployment simpler. See "Deploying to a Cluster" on page 4-3 for more information. See also "Transport Provider Components" on page 2-9.

Deploying the Transport Provider

This section discusses how to deploy a transport provider.

Tip: For more information on deploying applications to AquaLogic Service Bus, see the *AquaLogic Service Bus Deployment Guide*.

After you create a deployable EAR file for your transport provider, you need to deploy it to the AquaLogic Service Bus domain. You can deploy the EAR by whatever method you prefer:

- Programmatically (using WebLogic Deployment Manager JSR-88 API)
- Using the WebLogic Server Administration Console
- Adding an entry similar to Listing 4-1 to the AquaLogic Service Bus domain `config.xml` file

Listing 4-1 Application Deployment Entry

```
<app-deployment>
    <name>My Transport Provider</name>
    <target>AdminServer, myCluster</target>
    <module-type>ear</module-type>
    <source-path>$USER_INSTALL_DIR$/servicebus/lib/mytransport.ear</source-path>
    <deployment-order>1234</deployment-order>
</app-deployment>
```

Note: The deployment order of your transport provider EAR file should be high enough so that the entire ALSB Kernel EAR is deployed before the transport provider.

If you deploy the transport provider as an EAR file, typically `TransportManager.registerProvider()` is called from within the EAR file's `ApplicationLifeCycleListener.postStart()` method.

Undeploying a Transport Provider

Once a transport provider has been registered with ALSB, the undeployment or unregistration of the transport provider is not supported.

Deploying to a Cluster

In a cluster environment, only the configuration part of the transport provider needs to be deployed on the AquaLogic Service Bus domain Administration Server. The runtime parts need only be deployed on the managed servers for load-balancing and failover.

If you deploy the runtime and configuration parts of the transport provider in a single deployment unit, the resulting EAR file needs to be aware of where it is being deployed (Administration Server or Managed Server) and exhibit only configuration behavior on the Administration Server and only runtime behavior on the Managed Server.

For example, in the initialization pseudo code in `some_transport.ear` you can use this logic to decide whether or not to activate the configuration or runtime portion of the provider:

```
protected SomeTransportProvider() throws TransportException {
    … some other initialization code …
    if (!isAdminServer || !clusterExists)
        _engine = new RuntimeEngine(…);
}
```

In this case, creating an instance of the RuntimeEngine class is runtime behavior and only needs to happen on a managed node or administration node in a single server domain.

Furthermore, as mentioned previously, in a cluster environment, `TransportProvider.createEndPoint()` and `deleteEndPoint()` are called on an Administration Server as well as Managed Servers in the cluster (with the exception of WLS HTTP router/front-end host). Some transport providers can choose not to do anything other than registering the fact that there is an endpoint with the given configuration, such as HTTP. In general the transport provider needs to examine whether `createEndPoint()` or

`deleteEndPoint()` is called on the Administration or Managed Server to decide the appropriate behavior.

CHAPTER 5

Transport SDK Interfaces and Classes

This chapter lists and summarizes the classes and interfaces provided by the Transport SDK. For information on which interfaces are required to develop a custom transport provider, see Chapter 3, "Developing a Transport Provider."

This chapter includes these sections:

Introduction

All of the classes and interfaces discussed in this chapter are defined in the package `com.bea.wli.sb.transports`, and are part of `sb-public.jar`, which is listed on the system `CLASSPATH` after AquaLogic Service Bus is installed.

Schema-Generated Interfaces

A number of interfaces are generated from XML Schema by an XML Schema compiler tool. The source (XML Schema) for the following interfaces is provided in the file `TransportCommon.xsd`. This file is the base schema definition file for service endpoint

configurations. This file is located in
`BEA_HOME/weblogic92/servicebus/lib/sb-public.jar`

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

- **EndPointConfiguration** – The base type for endpoint configuration. An endpoint is an AquaLogic Service Bus resource where messages are originated or targeted. EndPointConfiguration describes the complete set of parameters necessary for the deployment and operation of an inbound or outbound endpoint.
- **RequestMetaDataXML** – The base type for the metadata of an inbound or outbound request. Metadata is not carried in the payload of the message, but separately and is used as the "context" for processing the message. Examples of such information that might be transmitted in the metadata are the Content-Type header, security information, or locale information.
- **RequestHeadersXML** – The base type for a set of inbound or outbound request headers.
- **ResponseMetaDataXML** – The base type for response metadata for an inbound or outbound message.
- **ResponseHeadersXML** – The base type for a set of response headers.
- **TransportProviderConfiguration** – Allows you to configure (a) whether this provider generates a service description (for example, WSDL) for its endpoints; (b) whether or not this provider supports inbound (proxy) endpoints; or (c) whether or not this provider supports outbound (business service) endpoints.

General Classes and Interfaces

This section summarizes general classes and interfaces of the Transport SDK.

This section includes these topics:

- Summary of General Classes
- Summary of General Interfaces

Note: For detailed information on each class and interface listed in this section, refer to the AquaLogic Service Bus Javadoc description.

Summary of General Classes

- **class TransportManagerHelper** – Helper class that allows the client to execute some common tasks with respect to the transport subsystem.
- **class ServiceInfo** – Wrapper class that describes information about a service, such as its transport configuration and its binding type.
- **class TransportOptions** – Supplies options for sending or receiving a message. There are two styles for using TransportOptions: multiline setup, and single-line use.
- **class EndPointOperations** – Describes different types of transport endpoint lifecycle-related events by which the transport provider is notified. Nested classes include: CommonOperation, Create, Delete, EndPointOperationTypeEnum, Resume, Suspend, and Update.
- **class Ref** – Uniquely represents a resource, project or folder that is managed by the Configuration system.
- **class TransportValidationContext** – Container that supplies information to transport providers that can be used when implementing validation checks of endpoint configuration.
- **class Diagnostics** – Contains a collection of Diagnostic entries relevant to a particular resource.
- **class Diagnostic** – Represents a particular validation message related to a resource. Diagnostic objects are generated as a result of validation that is performed when a resource changes. Such changes in the system trigger validation for the changed resource, as well as all other resources that (transitively) depend on the changed resource.
- **class EnvValue** – Represents an instance of an environment-dependent value in configuration data. Environment-dependent values normally change when moving the configuration from one domain to another. For example the URI of a service could be different on test domain and production domains.

Summary of General Interfaces

- **interface TransportManager** – A singleton object that provides the main point of centralization for managing different transport providers, endpoint registration, control, processing of inbound and outbound messages, and other points.
- **interface TransportProvider** – Represents the central point for management of transport protocol-specific configuration and runtime properties. There is a single instance of

TransportProvider for every supported protocol. For example, there is a single instance of HTTP transport provider, JMS transport provider.

- **interface BindingTypeInfo** – Describes the binding details of the service. The implementation is a convenience wrapper class around several internal AquaLogic Service Bus structures. Additional methods can be added as needed by transport providers.
- **interface TransportWLSArtifactDeployer** – The plugin interface for modules that need to deploy/undeploy/modify WLS related artifacts along with an AquaLogic Service Bus deployment. For example, in certain cases, WLS queues need to be deployed in response to the creation of a service.

Tip: For more information, see "When to Implement TransportWLSArtifactDeployer" on page 3-26.

- **interface SelfDescribedTransportProvider** – Extends TransportProvider. Those transport providers that generate a service binding type description from a given transport endpoint need to implement this interface. An example is the EJB transport provider.
- **interface SelfDescribedBindingTypeInfo** – Extends the BindingTypeInfo interface for those services that are self-described (for example, EJB services).
- **interface WsdlDescription** – Describes the WSDL associated with a registered AquaLogic Service Bus service.
- **interface ServiceTransportSender** – Sends outbound messages to a registered service associated with a transport endpoint. `TransportProvider.sendMessageAsync()` gets an instance of ServiceTransportSender (which extends TransportSender) from which the provider can retrieve the payload and metadata for outbound requests.
- **interface CredentialCallback** – Transport providers get an instance of this callback interface from AquaLogic Service Bus. The transport provider can call its methods to fetch a credential used for outbound authentication.
- **interface TransportEndPoint** – A transport endpoint is an AquaLogic Service Bus entity/resource where service messages are originated or targeted.

Source and Transformer Classes and Interfaces

Below is a description of the base Source and Transformer interfaces, along with several concrete Sources provided with AquaLogic Service Bus and some supporting classes. For more information, see "Designing for Message Content" on page 2-25.

Summary of Source and Transformer Interfaces

- **interface Source** – Represents source content in some form. Sources may be transformed into other Sources through a Transformer instance. At minimum, a Source must natively support conversion to a byte-based stream via the two methods defined in this interface. Source may or may not take into account various TransformOptions (for example, character-set encoding) during serialization.
- **interface SingleUseSource** – A marker interface indicating that a type of Source can only be consumed once. It also provides one helper method that can be used to determine if the Source is still "consumable" (valid).

  If you create a Source class that implements the Source interface, AquaLogic Service Bus is free to call the `getInputStream()` method multiple times, each time retrieving the input stream from the beginning. If the Source class implements SingleUseSource, AquaLogic Service Bus calls `getInputStream()` only once; however, AquaLogic Service Bus buffers the entire message in memory in this case.
- **interface Transformer** – Transforms one type of Source to another. The instance is responsible for indicating what types of sources it can convert between. Note that a transformer is required to support the full cross-product of transformations implied by the supported input and output sources. In other words, a transformer must support transforming any supported input source to any supported output source.

Summary of Source and Transformer Classes

- **class StreamSource** – A byte-stream Source whose content comes from an InputStream. As a byte-stream source, the serialization methods do not heed any transformation options.

  Note: Because this stream is backed by an InputStream, that means that this source is a single-use source. Both serialization methods pull from the same underlying InputStream, and once that content is consumed, it is gone. The push-based `writeTo()` method results in all data being consumed immediately, assuming no error occurs. The pull-based `getInputStream()` actually gives the underlying InputStream directly to the caller.
- **class ByteArraySource** – A byte-stream Source whose content comes from a byte array. As a byte-stream source, the serialization methods do not heed any transformation options.
- **class StringSource** – A Source that is backed by a single String. Serialization is simply a character-set encoded version of the character data.

- **class XmlObjectSource** – Apache XBean Source content is represented as an Apache XBean. The XBean may be typed and so may be accompanied by a SchemaType object and an associated ClassLoader. However, both of these are entirely optional and the XBean can be untyped XML.
- **class DOMSource** – A Source whose content comes from a DOM node. The referenced node may be a full-fledged `org.w3c.dom.Document`, but it may also be an internal node in a larger document.
- **class MFLSource** – Represents MFL content. MFL data is essentially binary data that has some logical structure imposed on it by an MFL definition. CSV is a simple example of MFL data, but the structure can be arbitrarily complex. The logical/in-memory representation of the data is an XML document, but its serialized representation is the raw unstructured binary data.
- **class SAAJSource** – A Source that is backed by a SAAJ SOAPMessage object. A SAAJSource is typically converted to and from MessageContextSource and MimeSource.
- **class MimeSource** – A Source representing arbitrary content with headers. Essentially this is a Source that represents a MIME part. Headers must conform to RFC822 whereas the Source can be any type of source. The serialization format for this Source is a fully-compliant MIME package. This source is also aware of Content-Transfer-Encoding, and it will perform the proper encoding of the underlying content stream if the header is present. Note that this means that the Source provided to the constructor should be in raw form and not be already encoded.
- **class MessageContextSource** – A Source that represents all message content. The Source for the message and attachments are left untyped to allow for deferred processing. Eventually, however, the attachments source will likely be converted into an AttachmentsSource object and the message source will likely be converted to a specific typed source such as an XmlObjectSource or a StringSource.

  Note: The serialization format of a MessageContextSource is always a MIME multipart/related package, irrespective of the native serializations of the message and attachment sources. However, if this serialized object is needed more than once, it is best to transform the Source into a MimeSource.
- **class AttachmentsSource** – A Source representing a set of attachments. Its stream representation is equivalent to the stream representation of the `$attachments` variable. In other words, the stream representation is XML (not MIME).
- **class TransformOptions** – Represents a set of transformation options. Instances of this class are used in conjunction with the Transformer class to influence how an input source is converted to an output source (for example, a change in character-set encoding from

SHIFT_JIS to EUC-JP). This class is also used by the InputStream/OutputStream methods of the Source interface, since that is effectively also a transformation between the Source and the byte-level representation in the InputStream/OutputStream.

Metadata and Header Representation for Request and Response Messages

This section lists classes and interfaces that deal with request and response message metadata representation. See also "Handling Messages" on page 3-14 and "Designing for Message Content" on page 2-25.

This section includes these topics:

- Runtime Representation of Message Contents
- Interfaces

Runtime Representation of Message Contents

- **abstract class CoLocatedMessageContext** – Needs to be extended by a transport provider that implements optimization for co-located outbound calls to go through a Java method invocation instead of the transport layer. For an example implementation, see the class `com.bea.alsb.transports.sock.SocketCoLocatedMessageContext.java`, which is part of the Sample Socket Transport described in Chapter 5, "Transport SDK Interfaces and Classes." See also "Co-Located Calls" on page 3-17.
- **abstract class RequestHeaders** – Represents a union of standard and user-defined headers in a given inbound or outbound request message. The set of standard headers is specific to each transport provider. This is an abstract class to be extended by each transport provider to implement its version of request headers.
- **abstract class RequestMetaData<T extends RequestHeaders>** – Represents inbound or outbound request message metadata information (for example, headers, request character set encoding, and so on.) Transport providers provide an extension of this class that adds metadata information applicable to the transport provider. For example, HTTP transport provider adds `get/setQueryString()`, `get/setClientHost()` and other methods.
- **abstract class ResponseHeaders** – Represents a union of standard and user-defined headers in a given inbound or outbound response message. The set of standard headers is specific to each transport provider. This is an abstract class to be extended by each transport provider to implement their version of response headers.

- **abstract class ResponseMetaData<T extends ResponseHeaders>** – Represents inbound or outbound response message metadata information (such as headers, request character set encoding, and so on.) Transport providers provide an extension of this class that adds metadata information applicable to the transport provider. For example, HTTP transport provider adds `get/setHttpResponseCode()` and other methods.

Interfaces

- **interface TransportMessageContext** – Most message-oriented middleware (MOM) products treat messages as lightweight entities that consist of a header and a payload. The header contains fields used for message routing and identification; the payload contains the application data being sent. In general, the transport-level message context consists of a message ID, RequestMetadata, request payload, ResponseMetaData, response payload and related properties.
- **interface InboundTransportMessageContext** – Inbound Transport Message Context implements the message context abstraction for incoming messages.
- **interface OutboundTransportMessageContext** – Outbound Transport Message Context implements the message context abstraction for outgoing messages.
- **interface ServiceTransportSender** – Sends outbound messages to a registered service. The service is associated with a transport endpoint.
- **interface TransportSendListener** – This is the callback object supplied to the outbound transport allowing it to signal to the system that response processing can proceed. This callback object should be invoked on a separate thread from the request message.

User Interface Configuration

This section includes these topics:

Overview

Because each transport provider can decide on a list of service endpoint specific configuration properties to persist, a flexible user interface is required that allows the user to enter provider-specific configuration properties for each new service endpoint. What follows is a set of

classes and interfaces that allow each transport provider to expose its own properties for the user to enter as part of AquaLogic Service Bus service definition wizard.

This section lists interfaces and classes used to develop the user interface for a new transport.

Summary of UI Interfaces

- **interface TransportUIBinding** – Represents an object responsible for rendering provider-specific UI pages used during the service definition, summary, as well as validation of transport provider specific endpoint configurations.

Summary of UI Classes

- **class TransportUIContext** – Supplies options for the transport provider specific user interface. It is passed by AquaLogic Service Bus Console to each transport provider.
- **class TransportUIGenericInfo** – Holds transport specific UI information for the common transport page in the AquaLogic Service Bus Service Definition wizard.
- **class TransportUIFactory** – Provides factory methods for creating a Transport Edit Field and different kinds of Transport UI objects associated with the field. Also provides some helper methods for accessing values in these objects.
- **class TransportEditField** – Represents a single editable UI element in the provider-specific portion of AquaLogic Service Bus Console service registration wizard.
- **class TransportViewField** – Represents a single read-only UI element in the provider-specific portion of the service summary page AquaLogic Service Bus Console service registration wizard.
- **class TransportUIError** – Returns validation errors to the AquaLogic Service Bus Console.

CHAPTER 6

Sample Socket Transport Provider

This chapter explains how to build and run the sample socket transport provider. This sample is installed along with AquaLogic Service Bus. The sample serves as an example implementation of a custom transport provider and the sample source code is available to you.

This chapter includes these topics:

Sample Socket Transport Provider Design

The primary purpose of the sample socket transport provider is to serve as an example transport provider implementation. This publicly available sample demonstrates the implementation and configuration details of the Transport SDK.

This section includes these topics:

Concepts Illustrated by the Sample

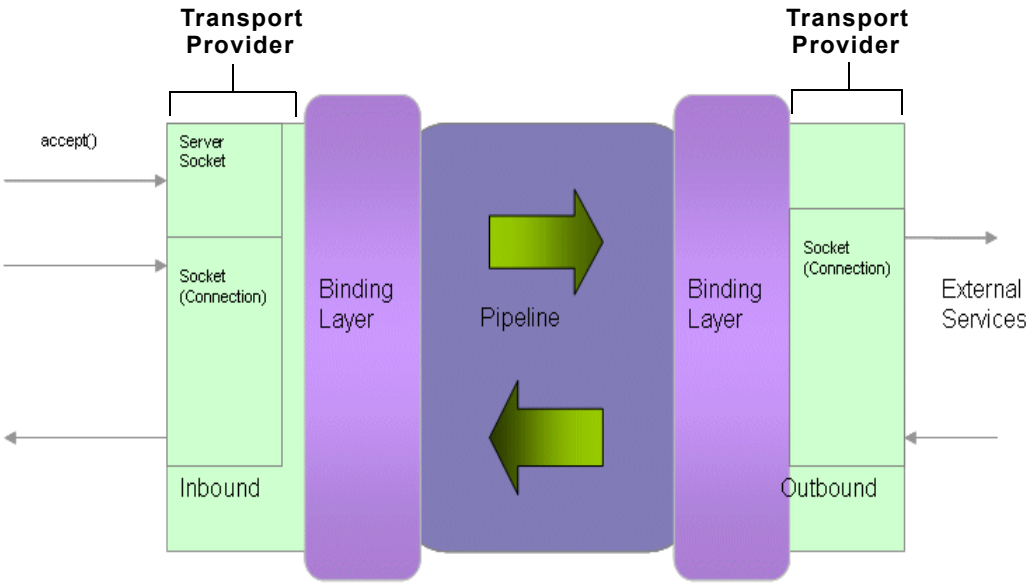
The sample transport is designed to send and receive streamed data to and from a configured TCP socket in AquaLogic Service Bus. The sample transport is intended to illustrate the following Transport SDK concepts:

- Implementing the set of Transport SDK APIs that are required to build a custom transport.
- Performing transport endpoint validations, such as checking that no socket endpoint is listening on the configured address.
- Implementing several UI configuration options, including socket properties and message patterns.
- Implementing a one-way or synchronous request-response message pattern.
- Using POJOs (Plain Old Java Objects) for metadata and headers of endpoint requests and responses.
- Showing how streaming is used in the AquaLogic Service Bus pipeline.

Basic Architecture of the Sample

Figure 6-1 shows the basic architecture of the sample socket transport provider. Any client can connect to the server socket. Data is received at the server socket and passes through the pipeline. The response comes back through the outbound transport. The response is finally sent back to the inbound transport and back to the client.

Figure 6-1 Sample Socket Transport Architecture

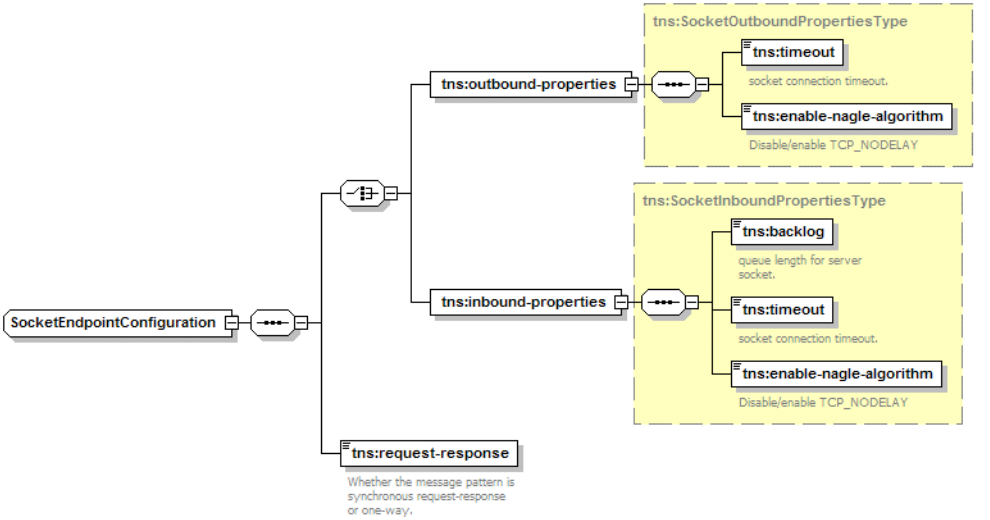


Configuration Properties

Figure 6-2 illustrates the configuration properties for the transport endpoint. These properties are configured in the schema file: `SocketTransport.xsd`. See "Sample Location and Directory Structure" on page 6-5 for information on the location of this file. This file allows you to extend the basic set of properties defined in the common schema provided with the SDK. Refer to the `SocketTransport.xsd` file for information on each of the properties.

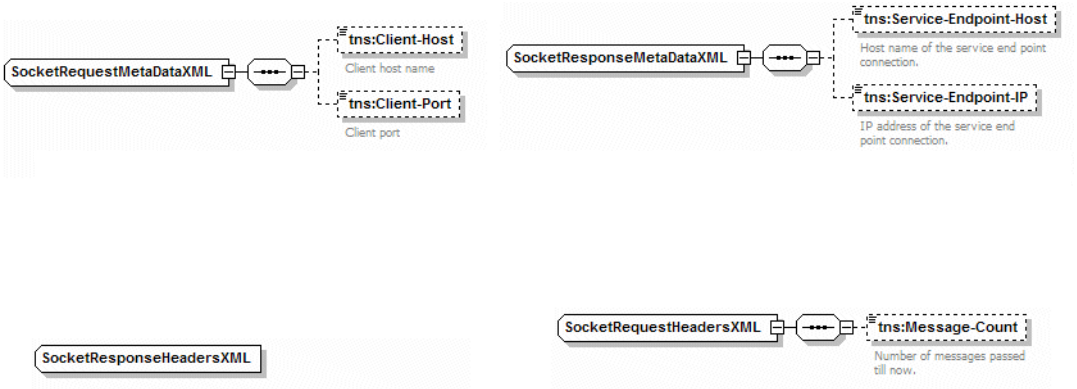
Tip: See also "4. Define Transport-Specific Artifacts" on page 3-6 for more information on these configuration properties.

Figure 6-2 SocketEndpointConfiguration Properties



Also in the `SocketTransport.xsd` file are the request/response header and metadata properties, as illustrated in Figure 6-3. Refer to the `SocketTransport.xsd` file for more information on these properties.

Figure 6-3 Request/Response Header and Metadata Configurations



Sample Location and Directory Structure

The sample socket transport provider is installed with AquaLogic Service Bus and is located in the following directory:

`BEA_HOME/weblogic92/samples/servicebus/sample-transport`

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

Figure 6-4 shows the directory structure for the sample socket transport provider. This section briefly describes the folders in the sample project. You can use this directory structure as a model for developing your custom transport provider.

Figure 6-4 Sample Transport Project Structure

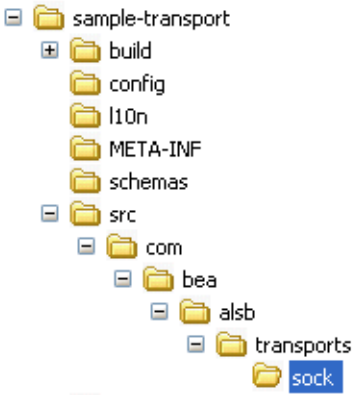


Table 6-1 lists and briefly describes the sample-transport directories.

Table 6-1 Sample Transport Provider Directories

build	Directory which contains all the class files
config	Configuration files directory. `SocketConfig.xml` – Socket transport provider configuration that is used by the Transport SDK.
l10n	Contains Internationalization files: `SocketTransportMessages.xml` – Configuration file for text messages which are displayed on the Aqualogic Service Bus Console. `SocketTransportTextMessages.xml`

Table 6-1 Sample Transport Provider Directories

lib	Directory which contains the sample deployment unit `sock_transport.ear`
META-INF	Contains application deployment descriptor files: `application.xml` – J2EE application descriptor file `weblogic-application.xml` – WebLogic application descriptor file
schemas	Contains the relevant schemas defined for this transport: `SocketTransport.xsd` – Describes Socket Endpoint Request/Response Metadata/headers
src	Source tree of the sample transport
test	(not shown) Test files directory: src – Source tree for test server and client

The following Ant build files are also located in the `sample-transport` directory:

- `build.properties` – Properties file for Ant.
- `build.xml` – An Ant build file with different targets for compile, build, and deploy.

Building and Deploying the Sample

This section explains how to build and deploy the sample transport provider.

Setting Up the Environment

1. Create a new domain or use one of the preconfigured domains that are installed with AquaLogic Service Bus.
2. Set the domain environment by running the following script:

`DOMAIN_HOME/bin/setDomainEnv.cmd` (`setDomainEnv.sh` on a UNIX system)

Building the Transport

To build the socket transport, do the following:

1. In a command window, go to the sample home directory:

   `BEA_HOME/weblogic92/samples/servicebus/sample-transport`

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

2. Execute the following command: `ant build-jar`

This command compiles all the source files, creates `sock_transport.ear`, and puts it in the directory: `BEA_HOME/weblogic92/servicebus/lib`

Deploying the Sample Transport Provider

To deploy the sample transport provider on a server, do the following:

1. Set the following variables in `sample-transport/build.properties`:

   `wls.hostname`

   `wls.port`

   `wls.username`

   `wls.password`

   `wls.server.name`

2. Deploy the transport provider on the server by running the following command:

   ```
   ant deploy
   ```

Start and Test the Socket Server

The sample project includes a simple socket server and a client to test the server. You can use this socket server to test the socket transport provider.

This section includes the following topics:

- Start the Socket Server
- Test the Socket Transport

Start the Socket Server

Run the following command to start the external service, which is a server socket that listens on a specified port and receives/sends the messages.

```
java -classpath .\test\build\test-client.jar -Dfile-encoding=utf-8
-Drequest-encoding=utf-8 com.bea.alsb.transports.sample.test.TestServer
<port> <message-file-location>
```

where:

- `port` – The port number at which ServerSocket is listening, which is the port number in the business service.
- `message-file-location` – (optional) The location of the message-file which will be sent as a response to the business service.
- `file-encoding` – A system property that is the encoding of the file. (default = utf-8)
- `request-encoding` – The encoding of the request that is sent by the socket business service. (default = utf-8)

Test the Socket Transport

Run the following command to start the service, which is a client to a configured socket proxy-service. It sends a message and receives the response from AquaLogic Service Bus.

```
java -classpath .\test\build\test-client.jar -Dfile-encoding=utf-8
-Dresponse-encoding=utf-8 com.bea.alsb.transports.sample.test.TestClient
<host-name> <port> <thread-ct> <message-file-location>
```

where:

- `host-name` – The host name of the AquaLogic Service Bus server.
- `port` – The port number at which the proxy service is listening.
- `thread-ct` – The number of clients that can send a message to AquaLogic Service Bus.
- `message-file-location` – (optional) The location of the message file that will be sent as a response to the business service.
- `file-encoding` – An optional argument specifying the encoding of the file. (default = `utf-8`)
- `response-encoding` – The encoding of the response received from the socket proxy service. (default = `utf-8`)

Configuring the Socket Transport Sample

The sample consists of a test server and a test client. The client sends a message to the server. You configure AquaLogic Service Bus to receive and process the message.

This section describes these tasks:

- Create a New Project

- Create a Business Service
- Create a Proxy Service
- Edit the Pipeline

Create a New Project

1. Start the AquaLogic Service Bus Console.
2. Open the Project Explorer.
3. In the Change Center, click **Edit**.
4. In the Projects panel, enter `SocketTest` in the **Enter New Project Name Field**, as shown in Figure 6-5.

Figure 6-5 Adding a New Project

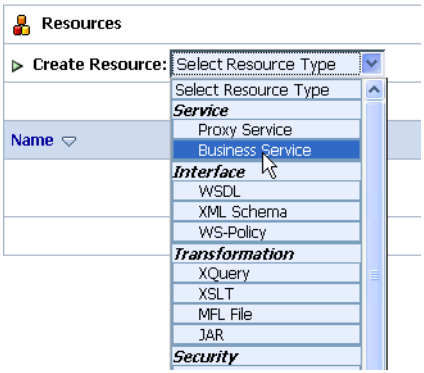
5. Click **Add Project**. The new project appears in the project table.

Create a Business Service

Create a business service to talk to the server.

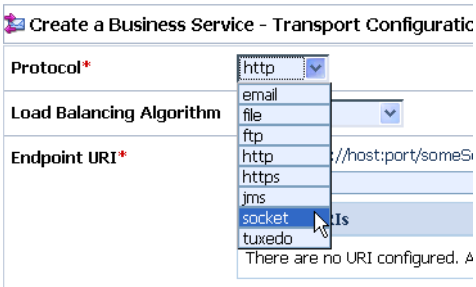
1. Click the SocketTest project name in the project table. The SocketTest panel appears.
2. From the Create Service dropdown menu, select **Business Service**, as shown in Figure 6-6. The General Configuration panel appears.

Figure 6-6 Creating a Business Service



3. In the General Configuration panel, enter `SocketBS` in the **Service Name** field.
4. Be sure **Any XML Service** is selected in the Service Type list, and click **Next**.
5. From the Protocol menu, select **socket**, as shown in Figure 6-7.

Figure 6-7 Choosing a Protocol



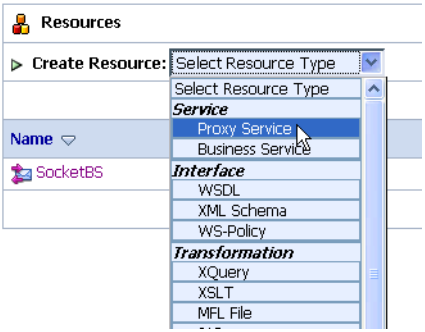
6. In the Endpoint URI field, enter: `tcp://localhost:7031`, and click **Add**.
7. Click **Next**.
8. In the next panel, accept the defaults by clicking **Next**.
9. After viewing the Summary panel, click **Save**.
10. In the Change Center, click **Activate**.

Create a Proxy Service

In this section, you create a proxy service.

1. From the Create Resource menu, select **Proxy Service**, as shown in Figure 6-8.

Figure 6-8 Creating a Proxy Service



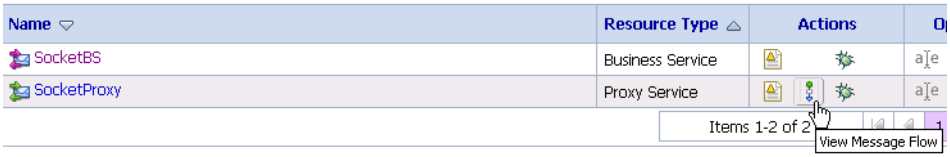
2. In the General Configuration panel, enter `SocketProxy` in the **Service Name** field.
3. Be sure that **Any XML Service** is selected in the Service Type list, and click **Next**.
4. From the Protocol menu, select **socket**.
5. In the **Endpoint URI** field, enter `tcp://7032`, and click **Next**.
6. In the next panel, accept the defaults and click **Next**.
7. After viewing the Summary panel, click **Save**.
8. In the Change Center, click **Activate**.
9. Click **Submit**.

Edit the Pipeline

Now that the business and proxy services are defined, you can edit the pipeline to route incoming messages to the business service.

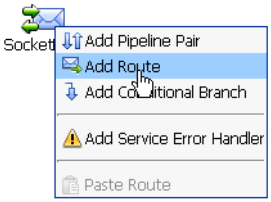
1. In the Change Center, click **Create**.
2. In the Resources section, click the **View Message Flow** icon in the SocketProxy row, as shown in Figure 6-9.

Figure 6-9 Selecting the Message Flow Icon



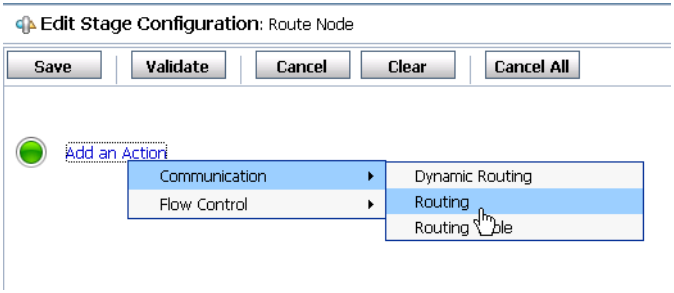
3. In the Edit Message Flow window, click the **SocketProxy** icon and select **Add Route** from the menu, as shown in Figure 6-10.

Figure 6-10 Editing the Message Flow



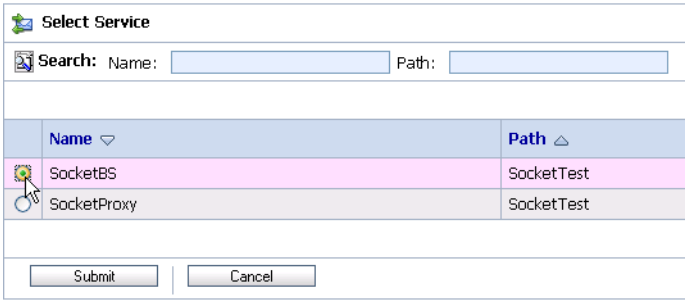
4. Click the **RouteNode1** icon and select **Edit Route** from the menu.
5. In the Edit Stage Configuration window, click **Add an Action**.
6. In the Route Node window, click **Add an Action** and select **Communication > Routing** from the menu, as shown in Figure 6-11.

Figure 6-11 Adding an Action



7. In the next panel, select **<Service>**.
8. In the Select Service window, select **SocketBS** from the list, as shown in Figure 6-12, and click **Submit**.

Figure 6-12 Selecting the Service to Route To



9. In the Edit Stage Configuration window, click **Save**.
10. Optionally, click the **RouteNode1** icon and change the name to **SocketBS**.
11. Click **Save**.
12. In the Change Center, click **Activate**, and then click **Submit**.

Testing the Socket Transport Provider

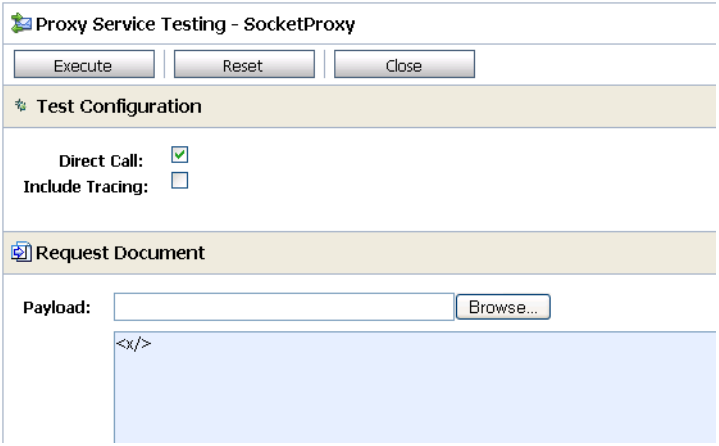
In this section you test the transport provider using AquaLogic Service Bus Console.

1. Start the test server, as explained previously in "Start the Socket Server" on page 6-7.
2. In the Project Explorer, click **SocketTest**.
3. In the SocketProxy row of the Resources table, click the **Launch Test Console** icon, as shown in Figure 6-13.

Figure 6-13 Starting the Test Console

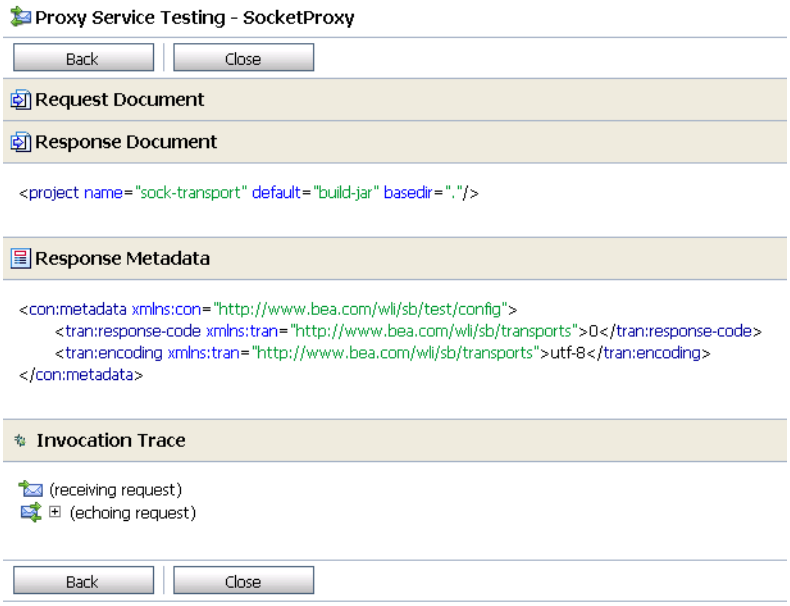
4. In the Test Console, enter any valid XML stanza in the text area, or use the **Browse** button to select a valid XML file on the local system. For example, in Figure 6-14, a simple XML expression `<x/>` is entered in the text area.

Figure 6-14 Test Console



5. Click **Execute**. If the test is successful, information similar that shown in Figure 6-15 appears in the Test Console. In addition, the XML text input into the Test Console is echoed in the server console.

Figure 6-15 Successful Test



6. Close the Test Console.

APPENDIX A

UML Sequence Diagrams

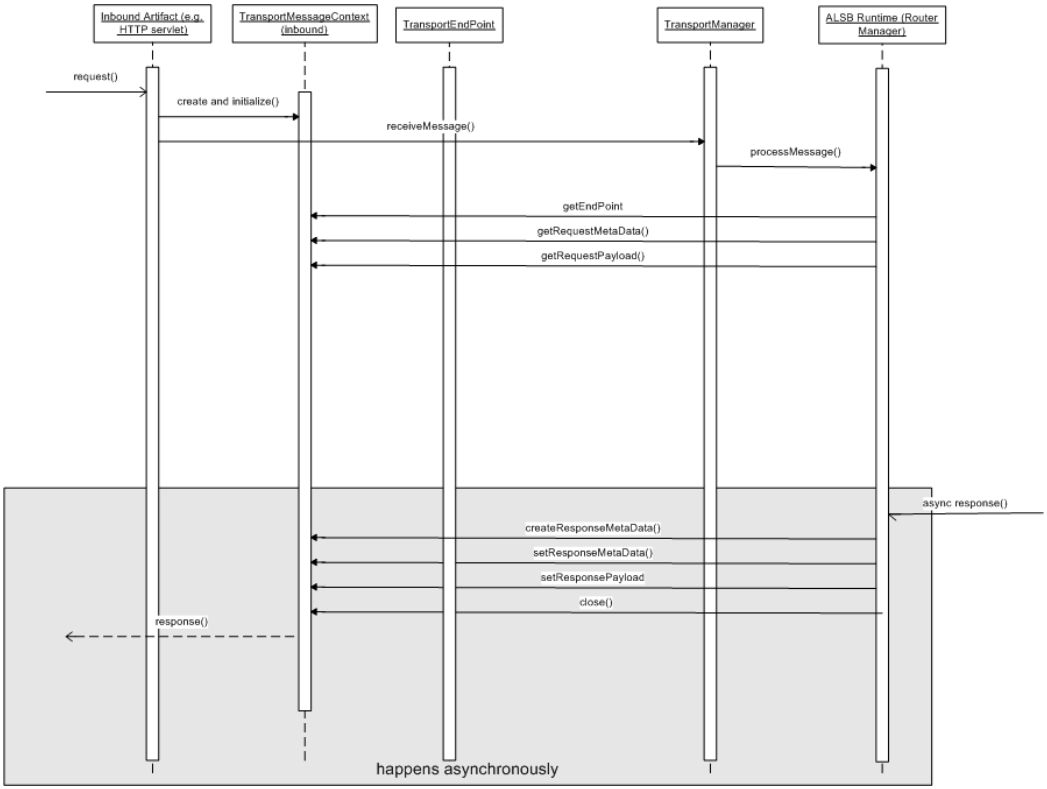
This chapter contains UML sequence diagrams that describe the flow of method calls through AquaLogic Service Bus runtime.

AquaLogic Service Bus Runtime Inbound Messages

The sequence diagram in Figure A-1 describes the flow of inbound messages through AquaLogic Service Bus runtime.

First, an inbound artifact, such as an HTTP Servlet, intercepts a client request. The transport provider creates a data structure called InboundTransportMessageContext. The message context packages headers from the request into a metadata object, converting the payload from an HTTP stream into a specific AquaLogic Service Bus source object. The transport provider calls the transport manager to receive the message. The transport manager preprocesses the message and passes the message to the AquaLogic Service Bus runtime for processing. The AquaLogic Service Bus runtime asks for the message context's service, service version, and other information. It also asks about the metadata and payload, which are required for processing. The runtime asks the MessageContext to create the response metadata and the response payload, and then calls `close()`. The response is sent back to the client.

Figure A-1 Inbound Messages at Runtime



AquaLogic Service Bus Runtime Outbound Messages

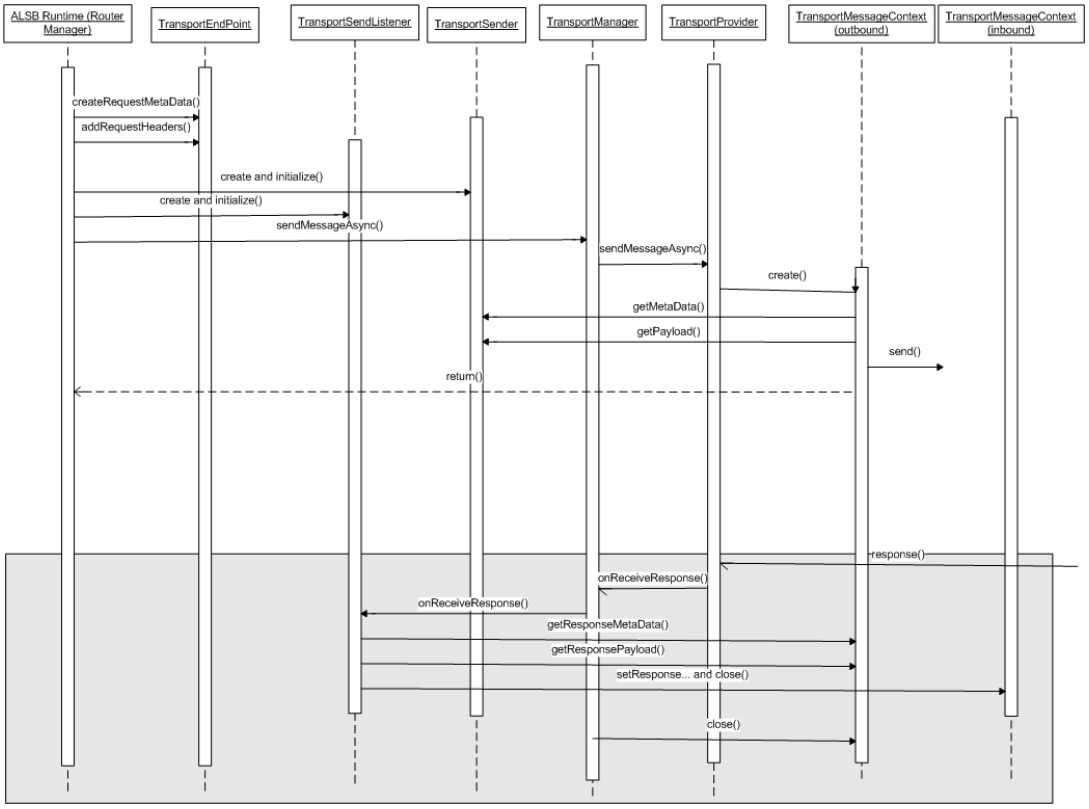
The sequence diagram shown in Figure A-2 describes the flow of outbound messages through AquaLogic Service Bus runtime.

The AquaLogic Service Bus runtime routes the message to an external service. The transport provider creates metadata for the request and creates a TransportSender object, which includes information about the payload and quality of service and retry information. Next, the provider calls TransportManager (the central hub for the transport subsystem) to send the message asynchronously. TransportManager calls the transport provider to send the message. The transport provider creates an OutboundTransportMessageContext. The transport provider then asks about the metadata and payload and other information and takes appropriate action. For

example, for a JMS message, the transport provider uses the JMS API to populate the headers and the payload and calls the protocol-specific send operation.

When a response comes in, the transport provider calls the TransportSendListener object. Eventually the transport manager invokes the response pipeline. After pipeline actions are executed, the outbound endpoint is closed.

Figure A-2 Outbound Messages at Runtime



Design Time Service Registration

During service registration, a wizard guides you through a number of AquaLogic Service Bus Console pages. Figure A-3 describes the service registration process. The basic steps include:

- Specifying the name of the service, the service binding type, and other information.
- Selecting from a dropdown list of transport providers (protocols). The AquaLogic Service Bus Console calls the transport manager to retrieve an object for each one of these entries in the list and gets a UI binding from each transport provider. This binding answers questions that the console requests, such as what is or is not supported. This step allows the console page to be populated with appropriate information.

- Entering transport-specific information. A transport provider specific form is generated automatically. The transport provider controls the contents of the page.
- Reviewing a summary page.

Finally, the transport provider is contacted and asked to validate the endpoint configuration and register the new endpoint. The endpoint is only created after activation occurs.

Figure A-3 Service Registration