Oracle® Service Bus

Transport SDK User Guide

10*g* Release 3 (10.3)

October 2008

Oracle Service Bus Security Guide, 10*g* Release 3 (10.3)

Contents

1. Introduction

2. Design Considerations

3. Developing a Transport Provider

4. Developing Oracle Service Bus Transports for Workshop for WebLogic

5. Transport SDK Interfaces and Classes

6. Sample Socket Transport Provider

7. Deploying a Transport Provider

A. UML Sequence Diagrams

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 Oracle Service Bus. It is assumed that you have solid knowledge of Web services technologies, Oracle Service Bus, the transport protocol that you want to use with Oracle 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 7, "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 Software Development Kit (SDK).

- Chapter 6, "Sample Socket Transport Provider"

  Discusses the sample socket transport provider that is provided with Oracle 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 Oracle Service Bus runtime.

Related Information

Specific documents that may be of interest to custom transport provider developers include:

- *Oracle Service Bus Concepts and Architecture*
- *Oracle Service Bus User Guide*
- *Using the Oracle Service Bus Console*
- *Oracle Service Bus Deployment Guide*
- Javadoc for Oracle 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:

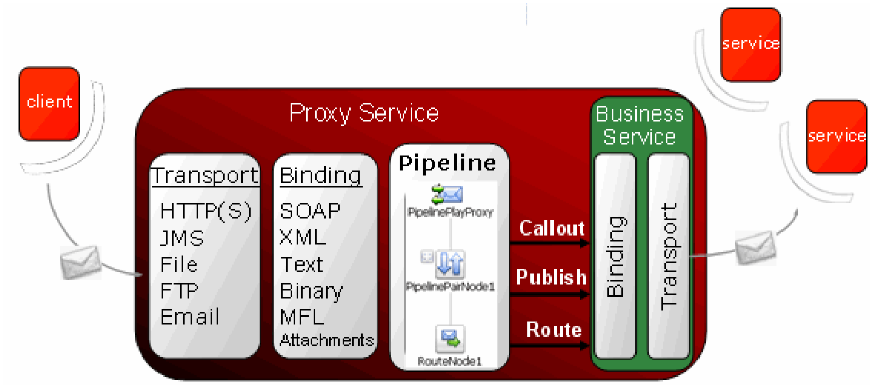
- What is a Transport Provider?
- What is the Transport SDK?
- Do You Need to Develop a Custom Transport Provider?
- Transport Provider Components
- The Transaction Model
- The Security Model
- The Threading Model
- Designing for Message Content

What is a Transport Provider?

A transport provider implements the interfaces of the Transport SDK and provides a bridge between Oracle 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 Oracle Service Bus. A client sends a message to Oracle 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 Oracle Service Bus.

Figure 2-1 Message Flow Through Oracle Service Bus



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

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

By default, Oracle 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 Oracle 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 Oracle 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 Oracle 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 Oracle 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 Oracle Service Bus. With this transport provider, Oracle Service Bus and 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 *Oracle 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
- Transport SDK Features
- Transport Provider Modes
- Related Features

Purpose of the SDK

Oracle Service Bus processes messages independently of how they flow into or out of the system. The Transport SDK provides a layer of abstraction between Oracle Service Bus and components that deal with the flow of data in and out of Oracle 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 Oracle Service Bus, see Table 2-1, "Transport Providers Installed with Oracle Service Bus," on page 2-3.

The SDK abstracts from the rest of Oracle 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 Oracle 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 Oracle Service Bus pipeline.
- Outbound messages originate from Oracle 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 Oracle 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 Oracle Service Bus, see the *Oracle 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 Oracle 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 3-30.

Processing Messages Asynchronously

Because the server has a limited number of threads to work with when processing messages, asynchrony is important. This feature allows Oracle 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 Oracle 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 Oracle 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
- When Alternative Approaches are Recommended

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 Oracle 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 Oracle 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 Oracle 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 Oracle Service Bus JMS transport (for example, a response queue defined in the message context).

When Alternative Approaches are Recommended

Creating a new Oracle 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 Oracle 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 *Oracle 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 Oracle solutions with Oracle Service Bus satisfies the transport requirement: Oracle WebLogic Server, Oracle WebLogic Integration, Oracle Data Service Integrator, Oracle Business Process Management, Oracle Tuxedo, Oracle WebLogic Portal.

- When service enablement tools, like Workshop for WebLogic, provide a simpler and more standards-based mechanism to implement SOA practices.
- When alternative connectivity solutions (certified with Oracle 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
- Design-Time Component
- Runtime Component

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 Oracle 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.

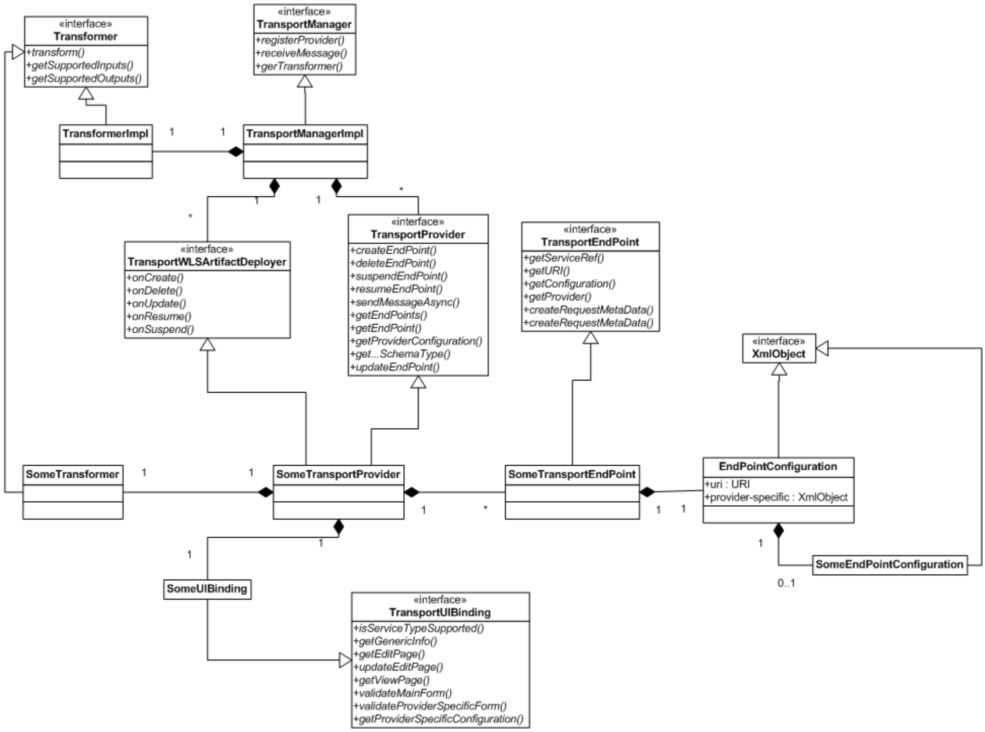
For more information, see Chapter 3, “Developing a Transport Provider” for a list of the required interfaces. A summary of the interfaces and classes provided by the Transport SDK are discussed in Chapter 5, “Transport SDK Interfaces and Classes.” Detailed descriptions are provided in Javadoc.

Design-Time Component

The design-time part of a custom transport provider consists of the user interface configuration. This configuration is called by the Oracle 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:

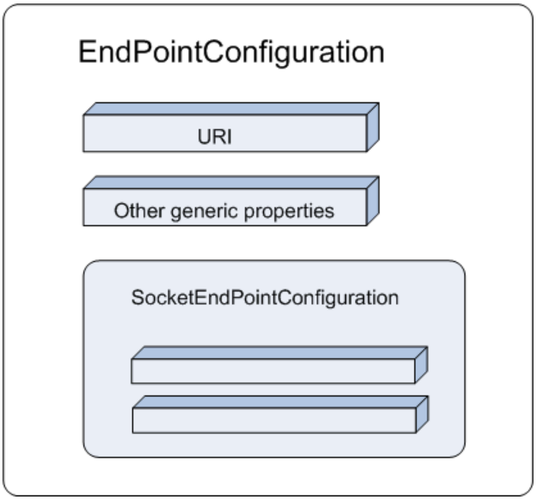
- **TransportManager** – A transport provider communicates with the transport manager through this interface. The implementation is not public.
- **TransportProvider** – Third parties must implement this interface. The TransportProvider keeps track of TransportEndpoint objects. TransportProvider also 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 Oracle 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

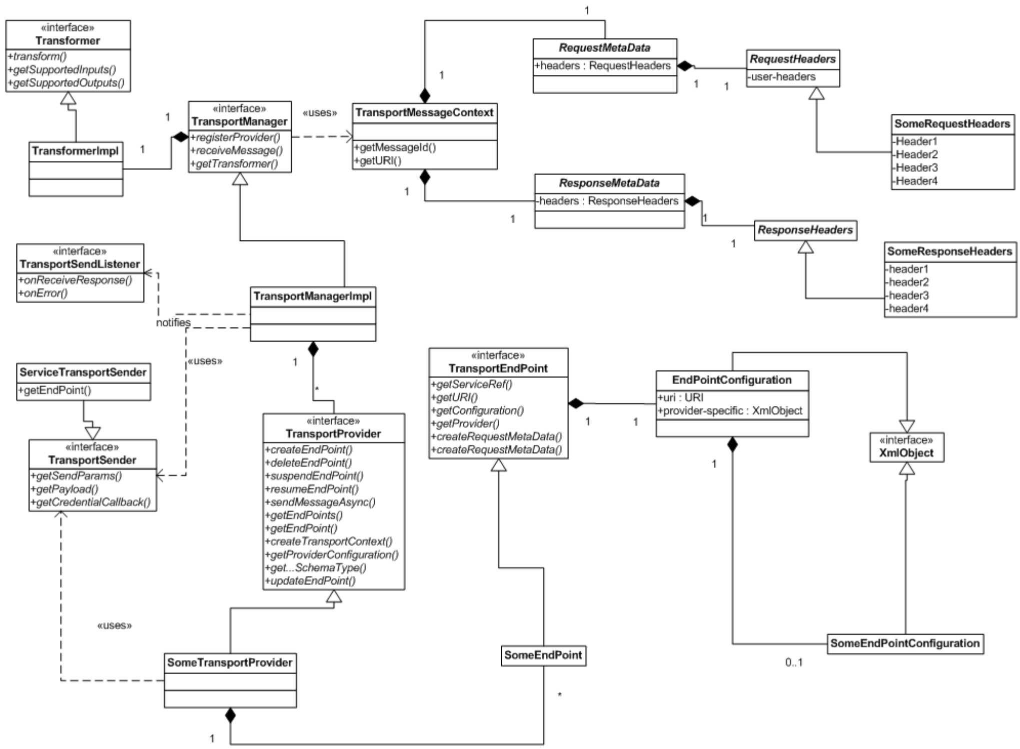
The runtime part of a custom transport provider:

- Receives messages and delivers them to the Oracle Service Bus runtime.
- Delivers outbound messages from Oracle 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 Oracle 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 Oracle Service Bus resource, such as a JMS proxy service, where messages are originated or targeted. In Oracle 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 Oracle 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 *Oracle 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 Oracle Service Bus was for the JMS transport, where transactions originated in and were bounded by the Oracle Service Bus domain. With the EJB and Tuxedo transports, transactions can originate outside of Oracle 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 Oracle 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 Oracle 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)

Oracle 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 3-20.

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 Oracle Service Bus Console*. For more information on message reliability, see the *Oracle 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 Oracle Service Bus. Within the Oracle 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 Oracle Service Bus by calling `TransportManager.receiveMessage()` within the scope of `weblogic.security.Security.runAs(subject)`. For information on this method, see WLS Javadoc.

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 WLS Javadoc.

The transport provider is also responsible for providing any Oracle 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 Oracle Service Bus service accounts. Service accounts are first-class, top-level Oracle Service Bus resources. Service accounts are created and managed in the Oracle 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 Oracle 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 Oracle 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)`. Oracle 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 ALSB 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)

Oracle 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 *Oracle 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.getUsernamePassword()`, passing the reference to the static service account.

Uniform Access Control to Proxy Services

Oracle 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 Oracle Service Bus) if required for transport provider specific reasons. If that is the case, please contact your Oracle representative. In general Oracle recommends transport providers let Oracle 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: Oracle 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, Oracle 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 Oracle Service Bus transport providers.

The Threading Model

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

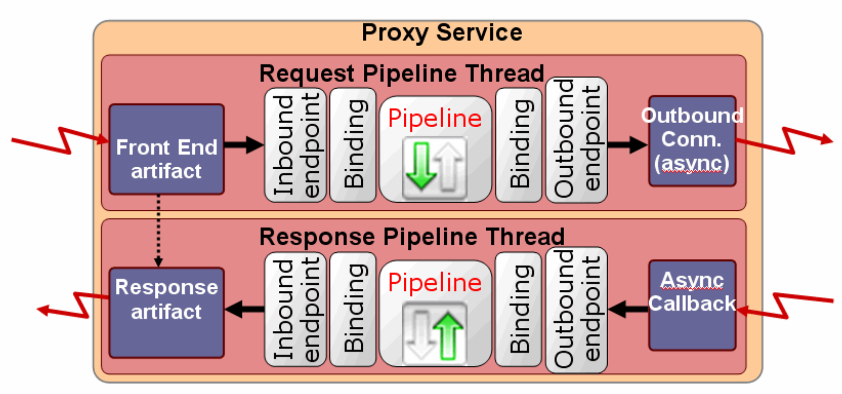
- Overview
- Inbound Request Message Thread
- Outbound Response Message Thread
- Support for Asynchrony
- Publish and Service Callout Threading

Overview

Figure 2-5 illustrates the Oracle 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 Oracle Service Bus (using a callback). The response is received, packaged, and handed to the Oracle 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 Oracle 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 Oracle 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 Oracle Service Bus pipeline, in the same (request) thread, Oracle 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 Oracle 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, Oracle 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 Oracle 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 Oracle 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 Interoperability with the EJB Transprot.

Support for Asynchrony

By design, the transport subsystem interacts asynchronously with Oracle 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, Oracle 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 Oracle Service Bus transport providers that support asynchronous outbound calls.

Table 2-2 Support for Asynchrony by Oracle 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 Oracle 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 Oracle 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 Oracle 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 Oracle Service Bus Console online help and the *Oracle Service Bus User Guide*.

Designing for Message Content

This section includes these topics:

- Overview
- Sources and Transformers
- Sources and the MessageContext Object
- Built-In Transformations

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 Oracle 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.

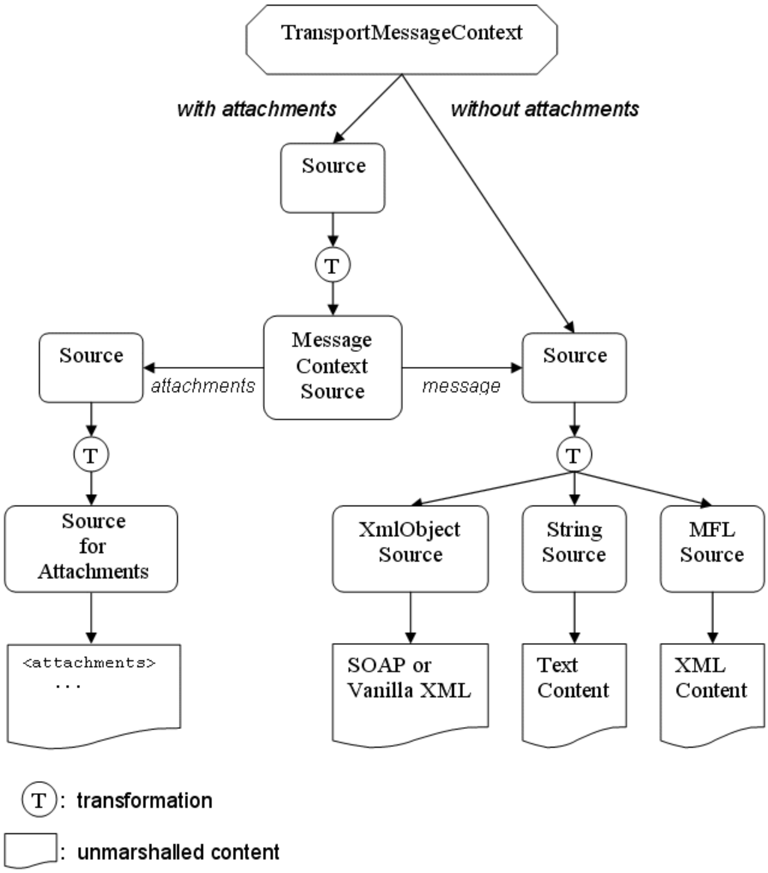
Below are the canonical Source types used for the set of defined service-types:

- **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 MessageContextSource. 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
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

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.

Developing a Transport Provider

The Transport SDK provides a layer of abstraction between transport protocols and the Oracle Service Bus runtime system. This layer of abstraction makes it possible to develop and plug in new transport providers to Oracle Service Bus. The Transport SDK interfaces provide this bridge between transport protocols, such as HTTP, and the Oracle 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 Oracle 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 7, "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 Oracle 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 7, "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.

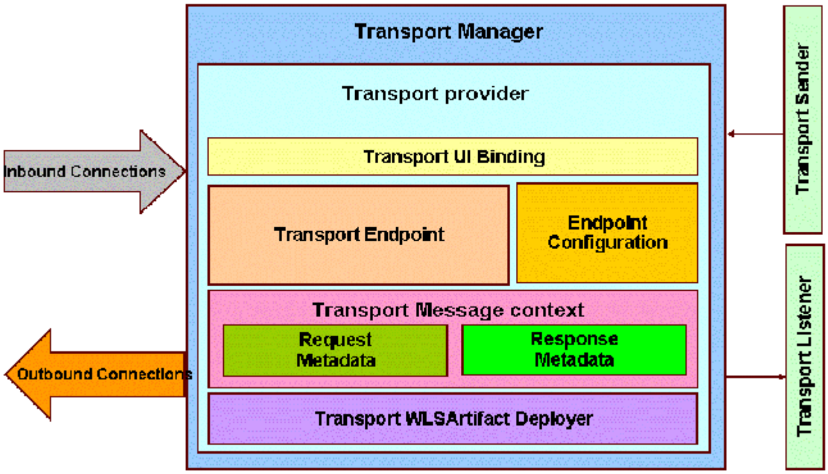
7. Implement the Runtime Interfaces.

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 Oracle 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 Oracle 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:

```
ALSB_HOME/samples/servicebus/sample-transport/schemas/
SocketTransport.xsd
```

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/ALSB_HOME/lib/sb-schemas.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 Oracle 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 Oracle 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 3-28 for more info.

Tip: The schema for TransportProviderConfiguration is defined in `TransportCommon.xsd`, which is located in `BEA_HOME/ALSB_HOME/lib/sb-schemas.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 Oracle 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 Oracle 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 Oracle 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 6-9 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 Oracle Service Bus Console and the underlying transport configuration is associated with an endpoint on the TransportProvider.

Note: The endpoint configuration is saved in the Oracle 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 3-30 and "Deploying to a Cluster" on page 7-3 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 Oracle 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 3-30.

- 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 7, "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
- Defining Custom Environment Value Types
- Publishing Proxy Services to a UDDI Registry
- When to Implement TransportWLSArtifactDeployer
- Creating Help for Custom Transports

Handling Messages

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

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, Oracle 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 Oracle 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 Oracle 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 Oracle 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 Oracle 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 Oracle Service Bus. For outgoing messages (outbound request and inbound response), the transport provider is responsible for telling Oracle 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 Oracle Service Bus what encoding it requires for the outbound request, and Oracle 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: Oracle 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 Oracle Service Bus Runtime

When Oracle 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 Oracle 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."

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 Oracle 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 Oracle 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 Oracle 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 run-time exceptions have on the transactional model. These cases include:

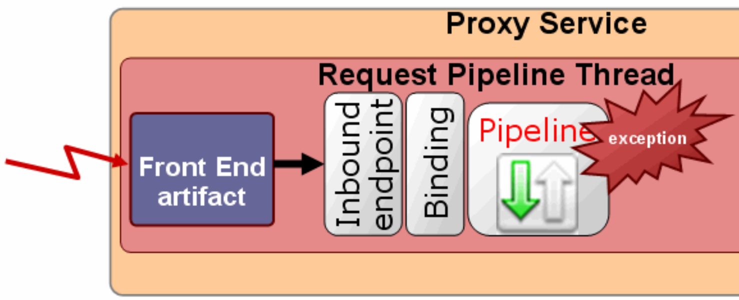
- Case 1: The exception occurs somewhere in the request pipeline but before the outbound call to the business service.
- Case 2: The exception occurs during the business service call.
- Case 3: The exception occurs sometime after the business service call in the response pipeline.
- Catching Application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

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



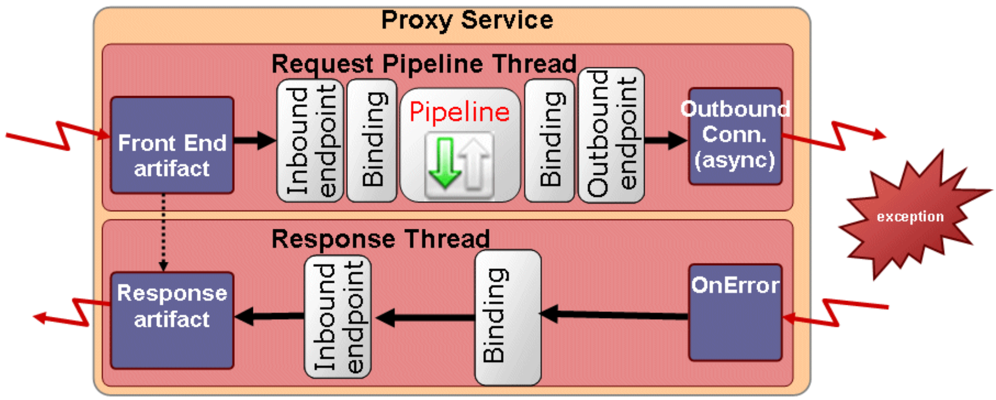
Case 2

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.

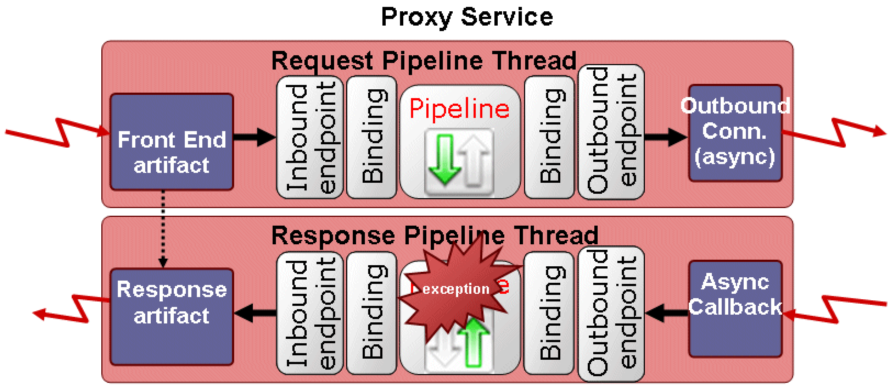
Figure 3-3 Error Case 2



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



Catching Application Errors

When business services try to access an external service and an error occurs in the external service application, such as a SOAP fault, subsequent retries by the services are likely to produce errors until the application is fixed.

Oracle Service Bus lets you identify application errors, giving you the option of preventing retries on application errors when your transport is used.

This section describes how to catch application errors in your transport and configure your transport to prevent application error retries.

Identifying Application Errors

In your transport provider code you must identify the conditions under which an application error occurs.

For example, in the Oracle Service Bus HTTP transport, an application error is one in which the HTTP response has a 500 status code, has a non-empty payload, and has a content type that is consistent with the service type, such as XML for SOAP.

When an error meets the application error conditions you define, return a TRANSPORT_ERROR_APPLICATION type using one of the following methods:

- Errors in the request – Throw a TransportException with the error code TRANSPORT_ERROR_APPLICATION in TransportProvider.sendMessageAsync().
- Errors in the response – Schedule TransportSendListener.onError() with the error code TRANSPORT_ERROR_APPLICATION.

The transport SDK can then identify application errors when they occur and handle them accordingly.

The transport SDK also sends application errors to the pipeline $fault variable.

Configuring Application Error Retries

In your <Transport>Config.xml file, enter the following element as a child of the <ProviderConfiguration> element, according to the TransportCommon.xsd schema in /osb_10.3/lib/sb-schemas.jar:

<declare-application-errors>true</declare-application-errors>

This entry provides a Retry Application Errors option on the business service's main transport configuration page when a user selects your transport. If you do not provide this element, the default value is false, application errors are not caught, and no option is provided to retry application errors.

Catching Connection Errors

Oracle Service Bus lets you identify connection errors in your transport, which triggers the transport SDK to take inaccessible endpoint URIs offline automatically. For example, in a cluster with Oracle Service Bus running on managed servers, a managed server that experiences a connection error on a service request can automatically mark the endpoint URI as offline.

You can re-enable endpoint URIs in the following ways:

- On configuring the business service, an you can set a Retry Count and Retry Iteration Interval to determine the frequency and number of retries after connection errors. A successful connection to the service after a retry automatically reactivates the endpoint URI.

  A Retry Iteration Interval of zero (0) takes the endpoint URI offline indefinitely and requires you to manually re-enable the endpoint URI.

- You can manually re-enable offline endpoint URIs in the Oracle Service Bus console, on the Operational Settings page for the business service.

The automated connection error functionality does not apply to the following situations:

- If a service pipeline dynamically sets an endpoint URI in $outbound/ctx:transport/ctx:uri, the transport SDK cannot take the URI offline, because the endpoint URI is not defined in the service configuration.
- The transport SDK does not persist connection status. After a server restart, all endpoint URIs are considered online.

For more information, see Managing Endpoint URIs for Business Services in the Oracle Service Bus *Operations Guide*.

Identifying Connection Errors

This section describes how to identify connection errors in your transport. Once caught, a connection error triggers the transport SDK to take the affected endpoint URI offline automatically.

In your transport provider code, you must identify the conditions under which a connection error occurs.

For example, in the Oracle Service Bus HTTP transport, a connection error is one in which the HTTP response has a 404 status code or there is an IOException when a connection is attempted on the endpoint URI.

When an exception meets the connection error conditions you define, return a TRANSPORT_ERROR_CONNECTION type using one of the following methods:

- Errors in the request – Throw a TransportException with the error code TRANSPORT_ERROR_CONNECTION in TransportProvider.sendMessageAsync().
- Errors in the response – Schedule TransportSendListener.onError() with the error code TRANSPORT_ERROR_CONNECTION.

The transport SDK can then identify connection errors when they occur and handle them accordingly.

The transport SDK also sends connection errors to the pipeline $fault variable and adds them to the response.

Defining Custom Environment Value Types

This section describes how to define custom environment value types that you want to use in your transport implementation. When you use the TransportProvider.getEnvValues() method to return environment values for an endpoint, you will be able to declare environment values of these custom types.

When your transport is used, custom environment value types can be used in the same ways that standard environment value types are used in Oracle Service Bus, such as for customization, find and replace, and preservation of values on configuration import. For example, you may want to be able to define and preserve references to a service account or a JMS queue in your transport configuration.

Environment value types can be any of the following categories: environment, operational, and security.

Add custom variables to your <Transport>Config.xml file. The schema that determines the XML structure is TransportCommon.xsd, located in /osb_10.3/lib/sb-schemas.jar.

Following is an example of a custom security variable in the JMS transport's JmsConfig.xml:

```
<env-value-type>
    <name>JMS Service Accounts</name>
    <localized-display-name>
        <localizer-class>com.bea.wli.sb.transports.messages.
         TransportsTextLocalizer</localizer-class>
        <message-id>JMS_SERVICE_ACCOUNTS</message-id>
    </localized-display-name>
    <localized-description>
        <localizer-class>com.bea.wli.sb.transports.messages.
          TransportsTextLocalizer</localizer-class>
        <message-id>JMS_SERVICE_ACCOUNTS</message-id>
    </localized-description>
    <simple-value>true</simple-value>
    <category>security</category>
</env-value-type>
```

Following are descriptions of key elements for custom environment value types:

- name – The variable name used by the transport SDK.

- display-name – The name for the variable that appears in the Oracle Service Bus user interface. Following is the localization alternative:
  - localized-display-name – Alternative, localized version of display-name.
  - localizer-class – The localization properties text file containing the localized display-name. The .properties extension is not required.
  - message-id – The property in the localization properties file that provides the value of the display name.
- description – Description of the environment value type. For localization, use the localized-description element instead with the localizer-class and message-id child elements as described in display-name.
- simple-value – If the environment value type is of the category "environment," simple-value determines whether or not this type is searchable with find and replace functionality in Oracle Service Bus (value of true or false).
- category – The category of the environment value type: "environment," "security," or "operational." When the category is "security" or "operational," you can decide whether or not to preserve the environment value type during configuration import. When the category is "environment," the environment value type is available for find and replace.

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 Oracle 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.")

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, Oracle Service Bus can publish the tModel you define here to configured registries. When a UDDI registry is configured to Oracle Service Bus, the “Load tModels into Registry” option can be specified. That option causes all of the tModels of Oracle 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 Oracle 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 Oracle 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()`.

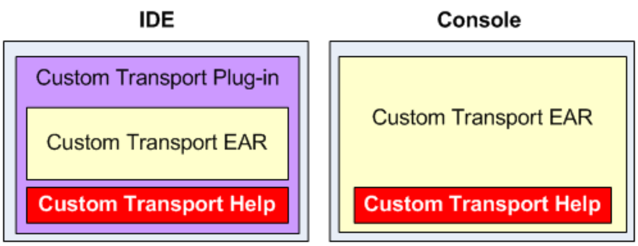
For more information on TransportWLSArtifactDeployer, see "Summary of General Interfaces."

Creating Help for Custom Transports

You can provide online help for your custom transports in the design environment (Workshop for WebLogicWorkshop for WebLogic) and the run-time environment (Oracle Service Bus console). Providing help is optional, but it can be extremely useful in guiding service creators through the transport configuration process.

Figure 3-5 shows help included with a custom transport in the development and run-time environments.

Figure 3-5 Custom transport help in the development and run-time environments



This section includes the following topics:

- Custom Transport Help Overview
- Providing Custom Transport Help in Workshop for WebLogic
- Providing Custom Transport Help in the Oracle Service Bus Console

Custom Transport Help Overview

This section describes the different options available for providing custom transport help in Workshop for WebLogic and the Oracle Service Bus console.

Note: Because of potential user interface and functionality differences between transport configuration in Workshop for WebLogic and the Oracle Service Bus console, consider creating separate help topics for both environments.

Workshop for WebLogic Help

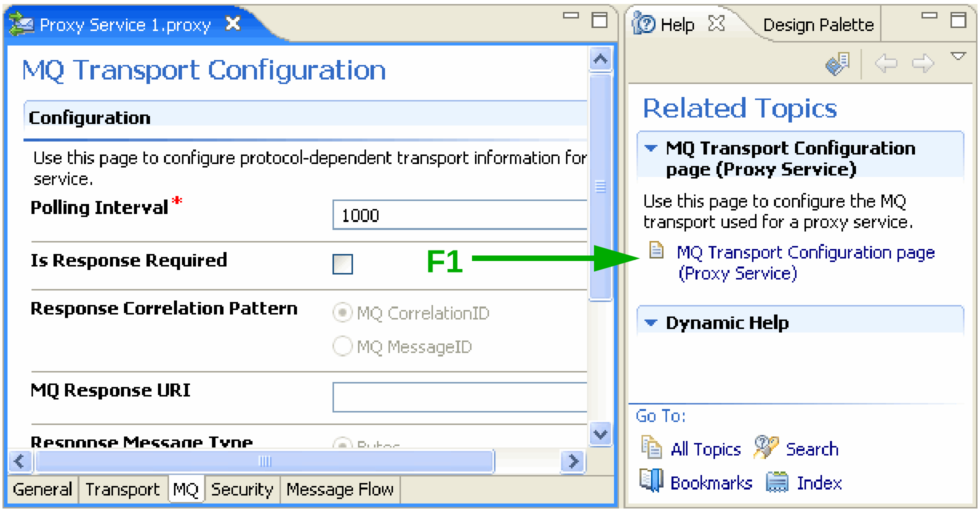
Workshop for WebLogic help is based on the Eclipse help framework. You have choices for your custom transport help implementation in Workshop for WebLogic.

- Context-Sensitive Help (F1)
- Workshop for WebLogic Help Table of Contents

Context-Sensitive Help (F1)

Context-sensitive help, launched by pressing F1 in Workshop for WebLogic, shows help topics within Workshop for WebLogic instead of launching a separate help window that displays the entire help system. Figure 3-6 shows how context-sensitive help appears in Workshop for WebLogic.

Figure 3-6 Pressing F1 on a transport configuration page to display help for the transport



All of the native Oracle Service Bus transports provide context-sensitive help from their respective transport configuration wizards and editors.

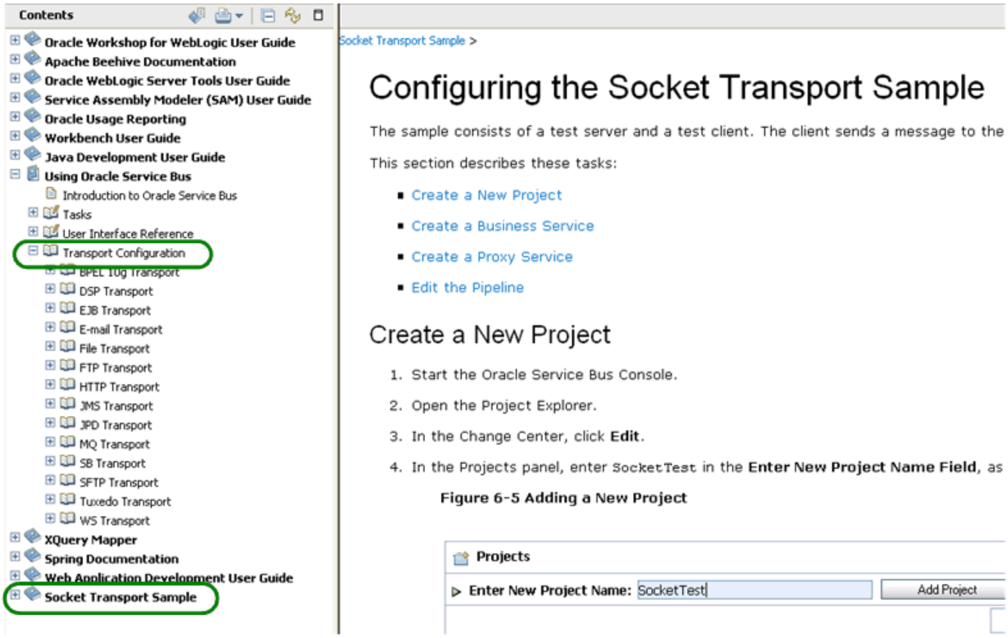
The benefit of context-sensitive help is quick user access to targeted help topics without leaving Workshop for WebLogic, which is particularly useful for help with custom transports.

Workshop for WebLogic Help Table of Contents

You can provide help content for your custom transport in the Workshop for WebLogic help system, accessible from the Workshop for WebLogic Help menu. When you launch Workshop for WebLogic help, a separate window displays the contents of the entire help system.

Figure 3-7 shows online help for transports in the Workshop for WebLogic help system.

Figure 3-7 Custom transport help in Workshop for WebLogic

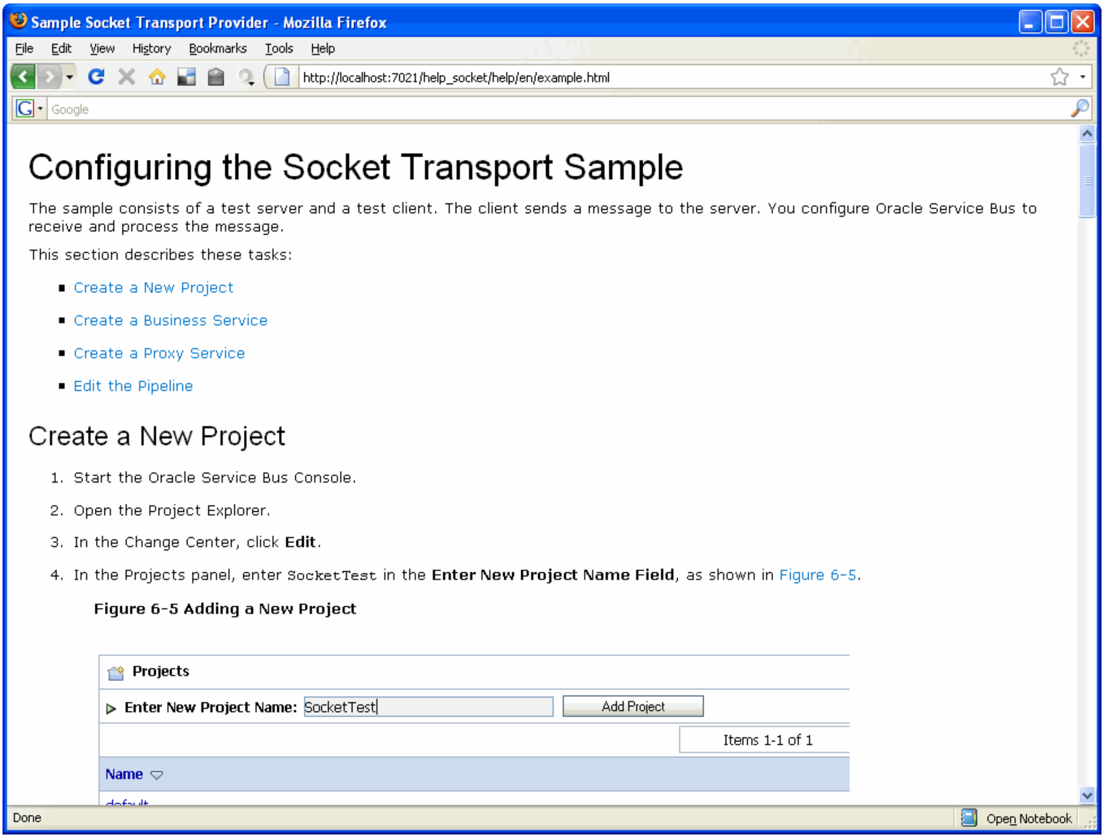


Your help topic(s) can appear in different locations of the help system table of contents, as shown in Figure 3-7, depending on how you package and configure your custom transport. For example, you can merge your transport help with the transport section of the Oracle Service Bus help topics, or you can provide your transport help at the top level of the help system.

Oracle Service Bus Console Help

You can also provide transport configuration help at run time in the Oracle Service Bus console. The Oracle Service Bus console provides its own integrated help system, but Oracle Service Bus displays custom transport help stand-alone in its own browser window, as shown in Figure 3-8. Custom transport help is displayed when you click Help on the transport configuration page.

Figure 3-8 Custom transport help from the Oracle Service Bus console



The following sections show you how to provide online help for your custom transport in both Workshop for WebLogic and in the Oracle Service Bus console.

Providing Custom Transport Help in Workshop for WebLogic

If you make your custom transport available for service configuration in the Workshop for WebLogic, you can provide help content that appears as context-sensitive help in Workshop for WebLogic, in the Workshop for WebLogic help system, or both. This section shows you how.

The sample socket transport and the Oracle Service Bus native transports provide a best-practice reference implementation for Workshop for WebLogic help and are used as examples in this section.

Transport help is part of the Workshop for WebLogic plug-in you create for your transport. For details on plug-in creation, see Chapter 4, "Developing Oracle Service Bus Transports for Workshop for WebLogic."

Providing Context-Sensitive Help in Workshop for WebLogic

Providing context-sensitive help gives users information about transport configuration directly in Workshop for WebLogic, where they are configuring the transport.

You can provide context-sensitive help on the transport configuration pages in both the service configuration wizard and the service editor in Workshop for WebLogic.

The following steps are required for providing context-sensitive help:

1. In plugin.xml, add an extension for org.eclipse.help.contexts that points to a context.xml file. See the `org.eclipse.help.contexts` example in Listing 3-1.

   This entry tells the plug-in where to find the context.xml file. The path to context.xml is relative to the plug-in root.

2. Create a context.xml file that maps the transport configuration user interface context IDs to help files. Oracle Service Bus provides context IDs for custom transports automatically. See "context.xml" on page 3-40.

3. Create your help topics. See "Help Content and Resources" on page 3-42.

4. Package all the help files. See "Packaging Help for the Transport Plug-in" on page 3-43.

Providing Help in the Workshop for WebLogic Help System

You can add your transport help to the main Workshop for WebLogic help system. Your help topics appear in the table of contents, as shown in Figure 3-7.

1. In plugin.xml, add an extension for org.eclipse.help.toc that points to a toc.xml file. See the org.eclipse.help.contexts example in Listing 3-1. Use the following guidance for setting the `primary` attribute.

   – If you are packaging your plug-in as a JAR file, or if you want your transport help to appear in the top level of the help table of contents, as shown in the "Sample Transport" entry in Figure 3-7, set `primary="true"`.

– If you want your transport help to be merged with the Oracle Service Bus help topics, as shown in Figure 3-7, set `primary="false"`.

  To merge your transport help with the main Oracle Service Bus help, your transport plug-in must be packaged as an exploded directory.

For details on plug-in packaging, see Chapter 4, "Developing Oracle Service Bus Transports for Workshop for WebLogic".

2. Create a toc.xml file that provides the table of contents structure for your transport help. See the examples in "toc.xml" on page 3-38.
3. Create your help topics. See "Help Content and Resources" on page 3-42.
4. Package all the help files. See "Packaging Help for the Transport Plug-in" on page 3-43.

Help Implementation Reference

Workshop for WebLogic help, which is based on the Eclipse help framework, requires the resources described in this section.

Use this reference section in conjunction with the previous procedures for implementing transport help in Workshop for WebLogic.

Note: Oracle Service Bus provides a sample help implementation in its sample socket transport, located at ALSB_HOME/samples/servicebus/sample-transport. The sample transport is a good reference implementation for developing your own custom transports and help. The sample plugin.xml is in the /eclipse subdirectory, and the help resources are in the /help subdirectory.

The Oracle Service Bus transport plug-ins located at `ALSB_HOME/eclipse/plugins/com.bea.alsb.transports.<transport>_<version>` also implement help as a custom transport would.

This section describes the following Workshop for WebLogic help resources:

- plugin.xml – The key file that identifies the components you want to add to Workshop for WebLogic; in this case, an addition to the help system.
- toc.xml – The hierarchy of your help topics that appears in the help system table of contents, as shown in Figure 3-7.
- context.xml – Enables context-sensitive help for your transport configuration user interface.

- Help Content and Resources – The HTML files, CSS file(s), images, and any other help resources you want to provide.

plugin.xml

The plugin.xml file is the key to adding your transport and transport help files to the Workshop for WebLogic environment. You must add entries in plugin.xml for your help table of contents (toc.xml) and for context-sensitive help (context.xml).

Listing 3-1 shows the toc.xml and context.xml (contexts_socketTransport.xml) entries in the sample socket transport's plugin.xml file, located in the `ALSB_HOME/samples/servicebus/sample-transport/eclipse` directory.

Listing 3-1 Sample transport plugin.xml

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

<?eclipse version="3.2"?>

<plugin>

...

   <extension

         point="org.eclipse.help.toc">

      <toc file="/help/en/toc.xml" primary="true"/>

   </extension>

   <extension

         point="org.eclipse.help.contexts">

      <contexts

            file="/help/en/contexts_socketTransport.xml"

            plugin="Socket_Transport"/>

   </extension>

</plugin>
```

- All paths are relative to the plug-in root directory.

- The `org.eclipse.help.toc` extension point makes the connection to the Workshop for WebLogic help system left navigation area.

  The `<toc file...>` entry references the toc.xml file containing the help topic hierarchy you create for your transport help.

- The `primary="true"` attribute is important. If set to true, your transport table of contents appears at the top level of the Workshop for WebLogic help system. Set it to true if you are packaging your custom transport plug-in as a JAR file.

  If set to false, Workshop for WebLogic expects your toc.xml to be merged into an existing toc.xml hierarchy, such as the main Oracle Service Bus help system. See the following toc.xml section for more information.

- The entry for the `org.eclipse.help.contexts` lets you implement Eclipse-based context-sensitive (F1) help for your transport. Context-sensitive help topic links appear in the Related Topics area of the Help view in Workshop for WebLogic.

For details on plug-in packaging, see Chapter 4, "Developing Oracle Service Bus Transports for Workshop for WebLogic".

toc.xml

The toc.xml file determines how your custom transport help appears in the left navigation area of the Workshop for WebLogic help system. You can provide your transport help at the top level of the Workshop for WebLogic help system table of contents (Listing 3-2) or merged into the Oracle Service Bus transport help topics (Listing 3-3).

Listing 3-2 Sample socket transport toc.xml for a top-level entry in the Workshop for WebLogic help system

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<?NLS TYPE="org.eclipse.help.toc"?>

<toc label="Socket Transport Sample">
       <topic label="Socket Transport Configuration Page (Business
       Services)" href="help/en/tpSOCKETTransportBizService.html"/>

       <topic label="Socket Transport Configuration Page (Proxy Services)"
       href="help/en/tpSOCKETTransportProxyService.html"/>

       <topic label="Configuring the Socket Transport Sample (Service
       Bus Console)" href="help/en/example.html"/>
</toc>
```

If you add your transport help as a top-level entry in the Workshop for WebLogic help system, be sure to set `primary="true"` in the org.eclipse.help.toc extension point in plugin.xml.

Listing 3-3 Sample toc.xml file for merging into the transports section of the Oracle Service Bus help

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<?NLS TYPE="org.eclipse.help.toc"?>
<toc link_to="../com.bea.alsb.docs/toc.xml#alsbTransports"
    label="Socket Transport Sample">
       <topic label="Socket Transport Configuration Page (Business
       Services)" href="help/en/tpSOCKETTransportBizService.html"/>
       <topic label="Socket Transport Configuration Page (Proxy Services)"
       href="help/en/tpSOCKETTransportProxyService.html"/>
       <topic label="Configuring the Socket Transport Sample (Service Bus
       Console)" href="help/en/example.html"/>
</toc>
```

The toc.xml merge illustrated in Listing 3-3 tells the Workshop for WebLogic help framework to merge your table of contents at that location in the main Oracle Service Bus help toc.xml that contains an anchor called `alsbTransports`.

- The link_to path is relative to the plug-in root directory. In this example, the relative link assumes the transport plug-in is a sibling to the com.bea.alsb.docs directory.
- If you merge your transport help with the Oracle Service Bus help topics, be sure to set `primary="false"` in the org.eclipse.help.toc extension point in plugin.xml.

  This toc.xml merging, which uses a relative reference to another plug-in, is possible only when you package your transport plug-in as an exploded directory.
- You can give your toc.xml a unique name (with an .xml extension) and put it anywhere within the plug-in directory as long as you provide the correct path to it in plugin.xml.
- All paths in toc.xml are relative to the plug-in root directory.

context.xml

The context.xml file, shown in Listing 3-4, maps the context IDs of your transport configuration wizard and editor pages to help topics. When users press F1 on your transport configuration pages, Workshop for WebLogic displays your help links in the Help view, as shown in Figure 3-6.

Listing 3-4 Sample context.xml (contexts_socketTransport.xml) for the sample socket transport

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.contexts"?>

<contexts>
    <!-- Default Socket Transport help -->
<context id="tpSOCKETTransportBizService"
       title="Socket Transport Configuration page (Business Service)">
    <description>The Sample socket transport illustrates Transport SDK
    concepts.</description>
       <topic href="help/en/tpSOCKETTransportBizService.html"
       label="Socket Transport Configuration Page (Business Services)"/>
</context>

<context id="tpSOCKETTransportBizWizard"
       title="Socket Transport Configuration page (Business Service)">
    <description>The Sample socket transport illustrates Transport SDK
    concepts.</description>
       <topic href="help/en/tpSOCKETTransportBizService.html"
       label="Socket Transport Configuration Page (Business Services)"/>
</context>

<context id="tpSOCKETTransportProxyService"
       title="Socket Transport Configuration page (Proxy Service)">
    <description>The Sample socket transport illustrates Transport SDK
    concepts.</description>
       <topic href="help/en/tpSOCKETTransportProxyService.html"
       label="Socket Transport Configuration Page (Proxy Services)"/>
</context>

<context id="tpSOCKETTransportProxyWizard"
       title="Socket Transport Configuration page (Proxy Service)">
    <description>The Sample socket transport illustrates Transport SDK
    concepts.</description>
       <topic href="help/en/tpSOCKETTransportProxyService.html"
       label="Socket Transport Configuration Page (Proxy Services)"/>
</context>
</contexts>
```

- Notice that there are four `<context>` entries:
  - The transport page in the business service editor
  - The transport page in the wizard for defining a new business service
  - The transport page in the proxy service editor
  - The transport page in the wizard for defining a new proxy service

  Also notice that the four entries reference only two HTML files: one for business service configuration in the wizard and the editor and one for proxy service configuration in the wizard and the editor.
- The `context id` attribute value is the context ID of the wizard or editor user interface.
- The `topic href` attribute tells Workshop for WebLogic which help topic to link to when the user presses F1.
- The `topic label` attribute determines the link text that appears in the Related Topics area of the Workshop for WebLogic Help view.
- The `description` element provides the text above the displayed link when the user presses F1.
- Context IDs for your transport user interfaces are available automatically. Use the patterns in Table 3-1 for the `context id` attribute, following the exact text and case sensitivity. If your `id` values are incorrect, context-sensitive help will not work for your transport.

Table 3-1 Context IDs for transport configuration pages

User Interface Component	Value for id attribute
Transport configuration page in the business service editor	tp<TRANSPORT_ID>TransportBizService – For example: tpSOCKETTransportBizService
Transport configuration page in the business service wizard	tp<TRANSPORT_ID>TransportBizWizard – For example: tpSOCKETTransportBizWizard

Table 3-1 Context IDs for transport configuration pages

Transport configuration page in the proxy service editor	tp<TRANSPORT_ID>TransportProxyService – For example: tpSOCKETTransportProxyService
Transport configuration page in the proxy service wizard	tp<TRANSPORT_ID>TransportProxyWizard – For example: tpSOCKETTransportProxyWizard

The <TRANSPORT_ID> value comes from your implementation of the `TransportProvider` class, where you set the String ID of the transport. Notice that the <TRANSPORT_ID> must be all uppercase letters even if you named the transport ID with lowercase letters.

- You can give your context.xml a unique name (with an .xml extension) and put it anywhere within the plug-in directory as long as you provide the correct path to it in plugin.xml.
- All paths in context.xml are relative to the plug-in root directory.

Help Content and Resources

You have a lot of flexibility in deciding what type of help content to provide, from a simple page of text with no graphics to multiple pages with many graphics, PDF files, embedded video and so on.

For example, you could create a single HTML file and reference it from the toc.xml and context.xml files; or you could create separate help files that describe the transport configuration fields for business services and proxy services and also provide a high-level overview, pointing at the three help files in different combinations from toc.xml and context.xml.

You can store your help topics and resources anywhere in your transport plug-in, as long as you reference them correctly in toc.xml and/or context.xml.

If you want your custom transport help to look like the Oracle Service Bus transport help, use the existing Oracle Service Bus transport plug-in HTML files and CSS. Transport help files are located in `ALSB_HOME/eclipse/plugins/com.bea.alsb.transports.<transport>_<version>/help/en`.

Because of potential user interface and functionality differences between transport configuration in Workshop for WebLogic and the Oracle Service Bus console, consider creating separate help topics for Workshop for WebLogic and for the Oracle Service Bus console.

Packaging Help for the Transport Plug-in

Your transport plug-in should contain the following:

- The plugin.xml file
- A transport JAR containing your transport classes and supporting files
- A help directory containing the toc.xml, context.xml, and help files

Whether you package your transport plug-in as a JAR or as an exploded directory, following is a recommended packaging structure for your transport help with relation to other resources:

/plugin_root

  plugin.xml

  transport.jar

  /help

    /en (locale)

      toc.xml

      context.xml

      /html

        <help files and resources>

Notice that with the `/en` directory the help is packaged to support localization. To provide localization, you must create a plug-in for each locale, as described in the Eclipse documentation.

> **Note:** You can also package your help files in a doc.zip file. For more information, see Help server and file locations in the Eclipse Platform Plug-In Development Guide.

Related Topics

For complete information on the Eclipse help framework, see the Eclipse help system at http://help.eclipse.org/help32/topic/org.eclipse.platform.doc.isv/guide/ua_help.htm.

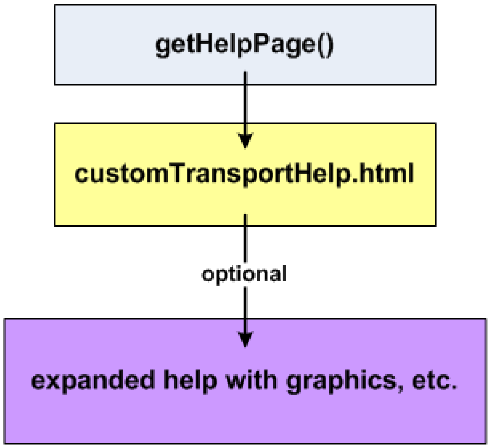
For information on plug-in packaging, see Chapter 4, "Developing Oracle Service Bus Transports for Workshop for WebLogic."

Providing Custom Transport Help in the Oracle Service Bus Console

This section shows you how to provide help for your custom transport at run time in the Oracle Service Bus console. Oracle Service Bus displays custom transport help as a stand-alone help page in a browser, as shown in Figure 3-8.

Figure 3-9 provides a high-level view of the Oracle Service Bus console help framework for custom transports.

Figure 3-9 Oracle Service Bus console help framework



By implementing a specific Oracle Service Bus interface, you use the `getHelpPage()` method to launch a single HTML page when the user clicks Help in the Oracle Service Bus console when your user interface has focus. The HTML file can contain the following:

- Text, inline CSS definitions, inline JavaScript functions
- References to graphics and other resources, as long as those resources are hosted in a Web application or an external Web site

In most situations, you should be able to provide all the help for your custom transport with text and inline formatting.

However, if you want to provide full-featured Web-based help that includes graphics and other external resources, those resources must be hosted in a Web application or an external Web site. You must either reference those external resources in the HTML file or provide a link from the

HTML file to an external location. For example, the sample socket transport help provides a link from the starting HTML file to a help topic with graphics that is running in a custom Web application. Using an embedded JavaScript call, you could also set up your HTML file to automatically redirect to the expanded help URL you want.

Following are the tasks involved in creating custom transport help in the Oracle Service Bus console:

- Implementing the CustomHelpProvider Interface
- Creating an HTML File to Launch
- Creating a Simple Web Application to Display Expanded Help (Optional)
- Packaging Transport Help for the Oracle Service Bus Console

Implementing the CustomHelpProvider Interface

To develop the configuration user interface for your custom transport, you implement the TransportUIBinding interface in a custom class. To provide help for your transport configuration user interface in the Oracle Service Bus console, you must also implement the CustomHelpProvider interface. CustomHelpProvider contains the `getHelpPage()` method you need to launch help for your transport configuration page in the Oracle Service Bus console.

The sample socket transport implements CustomHelpProvider in its SocketTransportUIBinding.java class, located at ALSB_HOME/samples/servicebus/sample-transport/src/com/bea/alsb/transports/sock.

Listing 3-5 contains snippets that illustrate the implementation of CustomHelpProvider.

Listing 3-5 Implementing CustomHelpProvider to provide help for your transport in the Oracle Service Bus console

```
public class SocketTransportUIBinding
    implements TransportUIBinding, CustomHelpProvider {

.
.
.

  public Reader getHelpPage() {
    String helpFile = "help/en/contexts_socketTransport.html";
    ClassLoader clLoader = Thread.currentThread().getContextClassLoader();
    InputStream is = clLoader.getResourceAsStream(helpFile);
```

```
    InputStreamReader helpReader = null;
    if(is!=null)
      helpReader = new InputStreamReader(is);
    else
      SocketTransportUtil.logger
        .warning(SocketTransportMessagesLogger.noHelpPageAvailableLoggable().
                  getMessage(uiContext.getLocale()));
    return helpReader;
  }

}
```

In Listing 3-5, `Reader getHelpPage()` returns a Reader stream that the Oracle Service Bus console uses to send the HTML page to the browser. The `helpFile` path is relative to the root within the transport JAR.

If you are providing help in multiple languages, use `TransportUIContext.getLocale()` to help provide the appropriate path to the localized content; in this case, providing the locale value for /help/<locale>/your.html.

Creating an HTML File to Launch

Create an HTML file for the `getHelpPage()` method to launch, as illustrated by help/en/contexts_socketTransport.html in Listing 3-5.

If you want to keep your help implementation simple, create the HTML file to use text, inline CSS definitions, and inline JavaScript functions. If you do this, you do not need to create a separate Web application to host graphics or other external resources.

However, if you want to provide more expanded help with graphics and other resources, reference those external resources in your HTML file, such as
`img src="/help_socket/help/en/wwimages/addProject.gif"` or
`a href="http://www.yoursite.com"`.

You can also set the HTML file up to automatically redirect to the expanded help with an embedded JavaScript call, as shown in Listing 3-7, which redirects from the sample socket transport HTML page to the expanded help_socket Web application help content.

Listing 3-7 JavaScript function that provides a redirect

```
<script language="JavaScript" type="text/javascript">
```

```
<!-- Begin
window.location="/help_socket/help/en/example.html";
// End -->
</script>
```

The sample socket transport HTML file provides a link to its expanded help. The HTML file, contexts_socketTransport.html, is located at ALSB_HOME/samples/servicebus/sample-transport/help/en/.

Creating a Simple Web Application to Display Expanded Help (Optional)

If you want to go beyond a basic text HTML file for your transport help, you can provide expanded help with graphics and other resources in various ways:

- Link from the self-contained HTML file to an existing URL; for example, if you have an existing Web site that contains your transport documentation. All that is required is that you provide a link to the URL from the self-contained HTML file. You can also insert references to graphics and other resources hosted on an external site.
- Create a Web application for the expanded help, bundle it with your transport, and link to it or reference graphics and other resources from the HTML file. This topic provides instructions on creating a Web application that is bundled in your transport EAR to display your expanded transport help.

Create the following files for your Web application:

META-INF/application.xml

In application.xml, give your Web application a context root that is used for the Web application's root URL. For example, Listing 3-6 shows the context root for the sample socket transport Web application.

Listing 3-6 application.xml for the sample socket transport Web application

```
<application>
    <display-name>Socket Transport</display-name>
    <description>Socket Transport</description>
    <module>
```

```
        <web>
            <web-uri>webapp</web-uri>
            <context-root>help_socket</context-root>
        </web>
    </module>
</application>
```

The sample socket transport application.xml file is located at ALSB_HOME/samples/servicebus/sample-transport/META-INF/.

This entry maps the file system directory /webapp to an alias Web application root URL:

`http://server:port`**`/help_socket/`**

With your help files inside the Web application in a directory such as /help/en/, the full URL to your expanded help would be:

`http://server:port/help_socket/help/en/index.html`

But your internal links to it only need to be

`/help_socket/help/en/index.html`

where index.html is the landing HTML page.

WEB-INF/web.xml

In web.xml, enter a display name and description for the Web application. This is standard deployment descriptor information. For example, Listing 3-7 shows the name and description of the sample socket transport Web application.

Listing 3-7 web.xml for the sample socket transport Web application

```
<web-app>
    <display-name>Sample Socket Transport Help WebApp</display-name>
    <description>
        This webapp implements the help webapp for the socket transport.
    </description>
</web-app>
```

The sample socket transport web.xml file is located at ALSB_HOME/samples/servicebus/sample-transport/webapp/WEB-INF/.

Help Content and Resources

Create and package your expanded help files inside the Web application directory. In the sample socket transport, the help files are stored in ALSB_HOME/samples/servicebus/sample-transport/help/en.

Note: The reason the socket transport help files are not stored in the /webapp directory is because the help directory contains help files and resources for both the Workshop for WebLogic plug-in and the Oracle Service Bus console. When the sample socket ANT build creates the transport JAR, transport EAR, and WorkShop Studio plug-in, it packages the help in different ways. For the transport EAR build, it moves the help files under the /webapp directory.

Because of potential user interface and functionality differences between transport configuration in Workshop for WebLogic and the Oracle Service Bus console, consider creating separate help topics for Workshop for WebLogic and for the Oracle Service Bus console.

Packaging Transport Help for the Oracle Service Bus Console

Your transport EAR should contain the following:

- A transport JAR stored in APP-INF/lib containing:
  - Your transport classes and supporting files
  - The HTML file for your transport help, ideally in a directory such as help/en/ for localization support
- Optionally, a Web application containing expanded help for your transport

Developing Oracle Service Bus Transports for Workshop for WebLogic

This chapter documents the best practices, design considerations, and packaging to develop transports for Oracle Service Bus Workshop for WebLogic design time. The Transport SDK interface provides a bridge between transport protocols and the Oracle Service Bus run time.

Tip: Before you begin this chapter, review Chapter 2, "Design Considerations".

This chapter includes the following sections:

Introduction

Oracle Service Bus transports were originally designed to be deployed on Oracle Service Bus servers and configured through the Oracle Service Bus console. With design environments like Workshop for WebLogic, some modifications to the SDK and the existing transports are necessary.

This document describes the additional steps to ensure Oracle Service Bus transports design time can be used on platforms outside the Oracle Service Bus console.

The sample socket transport was ported to Workshop for WebLogic and can be considered a best practice for Workshop for WebLogic integration. The sample socket resources are located at ALSB_HOME/samples/servicebus/sample-transport/. The Java source files are in the /src subdirectory. The sample also contains a build script that automatically packages the sample socket transport for both Workshop for WebLogic integration and Oracle Service Bus console deployment. For information on building and deploying the sample socket transport, see Chapter 6, “Sample Socket Transport Provider.”

Services Runtime and Services Configuration

When you develop a transport, you should distinguish the runtime aspects from the configuration aspects. The runtime aspects include proxy or business service deployment and service run-time invocation. The configuration aspects include proxy or service configuration and validation.

The runtime aspects do not need to change since they are always exercised in the context of a running Oracle Service Bus server. However, the configuration aspects are dependent on the design environment.

Developers should consider three different modes:

1. Online mode – The services for the custom transport are configured with the Oracle Service Bus console on a running Oracle Service Bus server.
2. Offline mode – The transport is configured with a design environment running outside the Oracle Service Bus server. No remote server is available.
3. Offline mode with remote server – The transport is configured with a design environment running outside the Oracle Service Bus server. However, a remote server is available and can be used for both validation and configuration purposes.

Transports running in Workshop for WebLogic must support offline mode and, optionally, offline mode with a remote server.

This section contains the following topics:

- Offline Methods
- Restrictions when Working Offline
- Working Offline with a Remote Server
- Bootstrapping Transports in Offline Mode
- Packaging Transports in Offline Mode

- Packaging Transports as Workshop for WebLogic Plug-Ins
- Reference

Offline Methods

When you deploy a transport in offline mode, the configuration framework creates a single session for all the resource configurations. This session is never activated. Since proxy or business services can only be deployed on a running Oracle Service Bus server, there is no need to activate the session. However, it is still important to detect conflicts and configuration errors and the validation methods are still exercised.

Following is a list of the minimum set of classes and methods defined by the Transport SDK that must be implemented in offline mode. The exceptions were removed from the methods signature for better readability.

Note: You do not need to completely re-implement your transport for offline mode. In most cases your transport will only need a few changes to existing methods to support both online and offline modes.

- The `public interface TransportProvider` class

```
String getId();
void validateEndPointConfiguration(TransportValidationContext context);
SchemaType getEndPointConfigurationSchemaType();
SchemaType getRequestMetaDataSchemaType();
SchemaType getRequestHeadersSchemaType();
SchemaType getResponseMetaDataSchemaType();
SchemaType getResponseHeadersSchemaType();
TransportProviderConfiguration getProviderConfiguration();
TransportUIBinding getUIBinding(TransportUIContext context);
void shutdown();
Collection<NonQualifiedEnvValue> getEnvValues(Ref ref,
     EndPointConfiguration epConfig);
void setEnvValues(Ref ref, EndPointConfiguration epConfig,
     Collection<NonQualifiedEnvValue> envValues);
Collection<Ref> getExternalReferences(EndPointConfiguration epConfig);
void setExternalReferences(Map<Ref, Ref> mapRefs, EndPointConfiguration
     epConfig);
Map<String, String> getBusinessServicePropertiesForProxy(Ref ref);
XmlObject getProviderSpecificConfiguration(Ref ref, Map<String, String>
     props);
```

- The `public interface TransportProviderFactory` class

  This interface registers transports in offline mode. See “Deploying a Transport Provider” on page 7-1 for more details.

- The `public interface TransportUIBinding` class

  You should implement all the methods in this interface and define the user interface used to configure a proxy or business service.

- The `public class TransportManagerHelper` class

  This class is typically used by `TransportProvider` developers. However, some of the methods that are not valid in offline mode will throw exceptions, which are described below. Other methods are meant only for runtime or deployment, such as public `isAdmin()`.

  Two methods are also available when working in offline mode with remote server:

  – The `public Set<String> getDispatchPolicies(JMXConnector connector);` method

  – The `public DomainRuntimeServiceMBean getDomainRuntimeServiceMBean(JMXConnector connector);` method

  See "Working Offline with a Remote Server" on page 4-5 for more information.

  Do not invoke the following two methods in offline mode:

  – The `isAdmin` method throws exceptions (`public static boolean isAdmin();`). This method throws a `java.lang.IllegalStateException` message.

  – The `clusterExists()` method always returns false (`public static boolean clusterExists();`). This method always returns false.

Restrictions when Working Offline

When you work offline, none of WebLogic Server services running on the server are available. Do not use these services inside the methods described in "Offline Methods" on page 4-3.

Following are examples of restrictions for working offline:

- The WLS MBeans are not available.
- The server Java properties are not available.
- You cannot access the JNDI tree directly. However, if JNDI properties are defined in the service configuration, you can attempt to use them.
- You can not determine if the service is going to run in a cluster or a standalone server.
- You do not have access to WebLogic Server security infrastructure.

- You do not have access to any static singleton service located on the server.

Because some of the services are not available, it is necessary to evaluate how the transport user interface is affected. In general, the user interface should be more flexible to let users manually configure values instead of trying to retrieve values from the server environment.

Workshop for WebLogic design time does not currently support deployment to an Oracle Service Bus cluster. Therefore, the user interface must be populated as if there is no cluster. If necessary, the user can use a customization file to update the configuration and force a deployment to an Oracle Service Bus cluster.

For example, some transports retrieve the list of available WorkManager items by using the TransportManagerHelper and letting the user pick one through a drop-down list. However, in offline mode, the MBeans are not available so the drop-down list cannot be populated. The transport provider has two choices:

1. Let the user type the correct WorkManager name. In that case, the user interface must be changed to be a text box and not a drop-down list when working offline.
2. Another less flexible option is to populate the drop-down list with just the default WorkManager. When the service is pushed to a running Oracle Service Bus server, the WorkManager name can be switched using an environment value substitution.

Working Offline with a Remote Server

When you work offline, a remote server might be available. For instance, when you configure a service on Workshop for WebLogic, the user can associate a remote Oracle Service Bus server to the current project. The transport provider can take advantage of the remote server by accessing the WebLogic Server MBeans and retrieving information. This mode is similar to working online; however, some restrictions still apply since the code is not running on the server and only the MBeans are available.

When you work offline, the following statements apply:

- The server Java properties are not available.
- You cannot use most of TransportManagerHelper methods as described in "Offline Methods" on page 4-3.
- You cannot access the JNDI tree directly. However if JNDI properties are defined in the service configuration, you can attempt to use them.
- You do not have access to any static singleton service located on the server.

To access the MBeans, the framework provides an instance of JMXConnector when it requests the TransportUI object, or when it asks the provider to validate a configuration. The JMXConnector is available in the TransportUIContext or the TransportValidationContext:

```
JMXConnector connector =
(JMXConnector)uiContext.get(TransportValidationContext.JMXCONNECTOR);
```

For more information, see the sample transport in "Reference" on page 4-10.

If the connector is not present, a remote server is not available. This connector object can then be used to access the MBeans. Helper methods have been added to the TransportManagerHelper to retrieve the list of WorkManager and WebLogic Server domain MBean.

Note: This behavior is generalized for both online and offline modes. The `public static Set<String> getDispatchPolicies()` method defined in the TransactionManagerHelper will be deprecated and must be replaced by the same method with JMXConnector as a parameter. If you do not replace it, the following error appears: `com.bea.wli.sb.transports.TransportException`.

Bootstrapping Transports in Offline Mode

In online mode, transports must be packaged as EAR files and deployed on an Oracle Service Bus server. When the EAR is loaded at startup, the transport registers a callback on a startup event and registers an instance of the TransportProvider to the TransportManager.

In offline mode, a new mechanism registers transports. The SDK introduces a new interface called `com.bea.wli.sb.transports.TransportProviderFactory`. A transport developer must implement this interface and must make the default constructor public. The interface is provided in "Reference" on page 4-10, as well as a sample implementation.

If the TransportProvideFactory is instantiated, you can assume the transport needs to work in offline mode (with or without a remote server).

Note: You can set a boolean operator in the TransportProviderFactory when the constructor is invoked to determine if the transport is running in offline mode. This information can also be passed in the TransportUIContext and the TransportValidationContext. Your Engineering Department can assist you in making this decision.

Packaging Transports in Offline Mode

In offline mode, you can use transports in different design environments. The Workshop for WebLogic environment defines specific packaging, which is described in the next section. In general, transports simply need to be available to external design time environments as a self-contained JAR file. A self-contained JAR file has the JAR file include the transport config.xml file, the header, metadata schemas, XBeans classes, TransportProviderFactory implementation, and the compiled transport classes.

Packaging Transports as Workshop for WebLogic Plug-Ins

Packaging your custom transport as a Workshop for WebLogic plug-in, in conjunction with your transport user interface implementation, lets service developers select and configure your transport in the development environment.

Figure 4-1 shows the service creation wizard with a drop-down field populated with available transports.

Figure 4-1 A custom transport available when creating a service

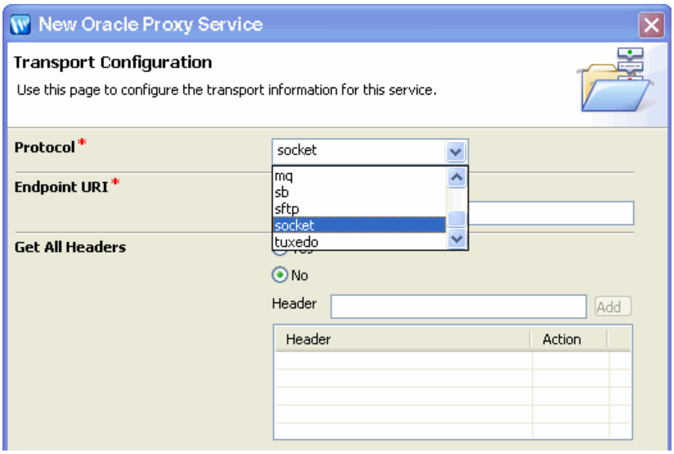
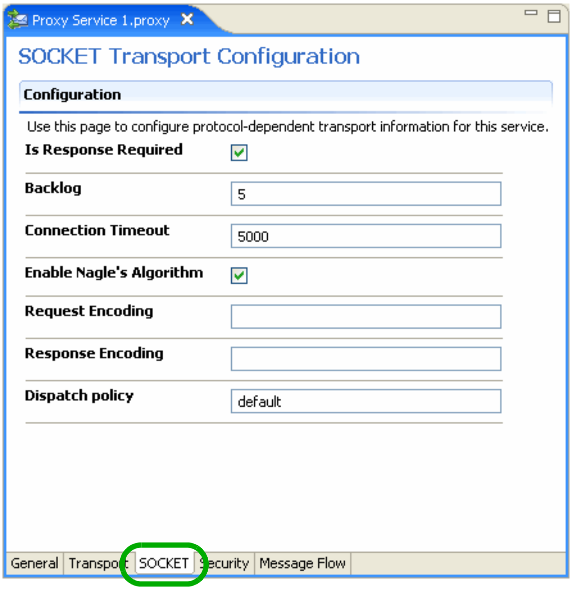


Figure 4-2 shows the service editor—after a service has been created—with a configuration page for the selected transport.

Figure 4-2 Transport configuration page in Workshop for WebLogic



Note: Your implementation of the TransportUIBinding interface determines the user interface for selecting and configuring your transport, both in Workshop for WebLogic and in the Oracle Service Bus console.

Oracle Service Bus provides Eclipse-based plug-ins to Workshop for WebLogic. The core Oracle Service Bus plug-in, com.bea.alsb.core, defines an extension point (com.bea.alsb.core.transports) that is used to register Oracle Service Bus transports in the Workshop for WebLogic environment as plug-ins.

Transport Plug-in Resources

Your transport plug-in must provide the following resources:

- MANIFEST.MF file – Contains key information about your transport plug-in. Use the sample socket transport MANIFEST.MF for reference. See "MANIFEST.MF" on page 4-15.
- In the plugin.xml file, provide an extension point that registers your transport with Workshop for WebLogic as a plug-in. Use the sample socket transport plug-in as a reference. See "plugin.xml" on page 4-15.

- A transport JAR file containing your transport implementation.
- (Optional) Resources for providing online help.

Transport Plug-in Packaging

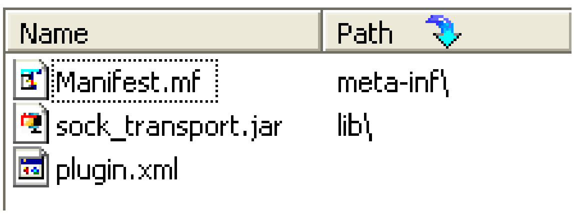
You can package your transport plug-in either as a JAR file or in an exploded directory. Packaging as a JAR makes the transport more portable. Packaging as an exploded directory lets you reference resources in other Oracle Service Bus plug-ins. For example, if you want to merge your transport help with the Oracle Service Bus transport topics in the Workshop for WebLogic help system, you must package your transport plug-in in an exploded directory, as described in "Providing Custom Transport Help in Workshop for WebLogic" on page 3-34.

Whether you package your plug-in in a JAR or an exploded directory, you must package your transport implementation in a JAR file.

Use the following guidance for packaging your transport as a plug-in:

- To construct the plug-in JAR or exploded directory name, append the `Bundle-Version` to the `Bundle-SymbolicName` from the MANIFEST.MF file. For example, the sample socket transport JAR is named Socket_Transport_3.0.0.0.jar.
- Package with the following directory structure, as shown in Figure 4-3:
  - plugin.xml (root of the plug-in)
  - /lib/your_transport.jar (transport classes and resources)
  - /META-INF/Manifest.mf
  - /help – If you are providing help for your transport, include a /help directory for your help resources, as described in "Providing Custom Transport Help in Workshop for WebLogic" on page 3-34.

Figure 4-3 Plug-in packaging



For reference on transport plug-in packaging:

- Build the sample socket transport, as described in Chapter 6, "Sample Socket Transport Provider." View the generated Socket_Transport_3.0.0.0.jar.
- View the packaging of the Oracle Service Bus transports, located at ALSB_HOME/eclipse/plugins/com.bea.alsb.transports.<transport_version>

Reference

This section contains the following topics:

- Working in Different Modes
- TransportProviderFactory
- Extension Point Schema
- plugin.xml
- MANIFEST.MF
- Build.xml
- TransportManagerHelper Methods

Working in Different Modes

Dispatch Policies are used by most transports and allow services throttling. This code distinguishes the three modes described in "Services Runtime and Services Configuration" on page 4-2:

- Online mode
- Offline mode
- Offline mode with remote server

The connection to the remote server is retrieved from the context, as shown in Listing 4-1.

Listing 4-1 Connection to the Remote Server

```
/**
  * Builds the dispatch policies in the ui object.
  *
```

```
* @param curDispatchPolicy
* @return TransportEditField containing existing dispatch policies.
*/
  public TransportEditField getDispatchPolicyEditField(StringcurDispatch
        Policy) {
        TransportUIFactory.TransportUIObject uiObject = null;
        Set<String> wmSet = null;

        if (SocketTransportProviderFactory.isOffline()) { // if on
             Eclipse try to get the MBeans from the UIContext
                  JMXConnector connector = (JMXConnector)uiContext.get
                       (TransportValidationContext.JMXCONNECTOR);
             if (connector != null) {
                  try {
                       wmSet = TransportManagerHelper.getDispatchPolicies
                             (connector);
                  } catch (Exception ex) {
                        wmSet = null; //continue
                  }
             }
        } else { // if running on the server use the helper to get the
             policies
             try {
                  wmSet = TransportManagerHelper.getDispatchPolicies();
             } catch (TransportException e) {
                  SocketTransportUtil.logger
                       .error(SocketTransportMessagesLogger.
                            noDispatchPolicies(), e);
             }
        }

        if (wmSet == null) // if JMXConnector not available or impossible
             to connect provide a simple edit field
        {
             uiObject = TransportUIFactory.createTextBox
                  (curDispatchPolicy);
        } else // create a drop down list
        {
```

```
                // adding default work manager to the list.
                wmSet.add(DEFAULT_WORK_MANAGER);
                String[] values = wmSet.toArray(new String[wmSet.size()]);
                uiObject = TransportUIFactory.createSelectObject(
                     values,
                     values,
                     curDispatchPolicy,
                     TransportUIFactory.SelectObject.DISPLAY_LIST,
                          false);
           }
           return TransportUIFactory.createEditField(DISPATCH_POLICY,
                TextMessages.getMessage(TextMessages.DISPATCH_POLICY,
                     locale),
                TextMessages.getMessage(TextMessages.DISPATCH_POLICY_INFO,
                     locale), uiObject);
     }
```

TransportProviderFactory

TransportProviderFactory, Listing 4-2, lets you provide development-time functionality in Workshop for WebLogic.

Listing 4-2 The TransportProviderFactory Class

```
package com.bea.wli.sb.transports;

import com.bea.wli.sb.transports.TransportException;
import com.bea.wli.sb.transports.TransportManager;

/**
 * This interface is the extension point to plug custom ALSB transports in
   Eclipse.
 * The implementation must declare the default class constructor.
 */
public interface TransportProviderFactory {
```

```
    /**
      * Registers a new provider with the transport manager. Typically
      * called by the ALSB core eclipse plugin.
      * @param tm the transport manager to which to register
      */
      public void registerProvider(TransportManager tm) throws
           TransportException;
    /**
      * @return a unique string that identifies this provider, like "http"
      * This method must return the same ID than provider.getId()
      */
    String getId();
}
```

The code sample in Listing 4-3 shows how the Socket Transport implements this interface

Listing 4-3 Example of the Socket Transport Implementing the Interface

```
package com.bea.alsb.transports.sock;

import com.bea.wli.sb.transports.TransportManager;
import com.bea.wli.sb.transports.TransportException;
import com.bea.wli.sb.transports.TransportProviderFactory;
public class SocketTransportProviderFactory implements
     TransportProviderFactory {

     public static boolean isOffline() {
          return isOffline;
     }

     private static boolean isOffline = false;

     public void registerProvider(TransportManager tm) throws
          TransportException {
          isOffline = true;
```

```
            SocketTransportProvider instance =
                 SocketTransportProvider.getInstance();
            tm.registerProvider(instance, null);
      }

      public String getId() {
            return SocketTransportProvider.ID;
      }
}
```

Extension Point Schema

Listing 4-4 is extracted from the extension point schema that defines the `transport` element and `transport-provider` attribute for adding a transport as a Workshop for WebLogic plug-in, shown in "plugin.xml" on page 4-15.

Listing 4-4 Part of the Extension Point Schema

```
<element name="transport">
     <complexType>
         <attribute name="transport-provider" type="string" use="required">
               <annotation>
                    <documentation>
                    </documentation>
                    <appInfo>
                         <meta.attribute kind="java" basedOn="
                              com.bea.wli.sb.transports.Transport
                              ProviderFactory"/>
                    </appInfo>
               </annotation>
          </attribute>
     </complexType>
</element>
```

plugin.xml

Listing 4-5 shows the transport extension for the sample socket transport plugin.xml file. The file is located at ALSB_HOME/samples/servicebus/sample-transport/eclipse/.

Listing 4-5 Plugin.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
     <extension
          id="socket"
          name="Socket Transport"
          point="com.bea.alsb.core.transports">
          <transport transport-provider="com.bea.alsb.transports
               .sock.SocketTransportProviderFactory"/>
     </extension>
</plugin>
```

- The `extension point` attribute value is always `com.bea.alsb.core.transports` for transports.
- The transport-provider attribute is the fully qualified path to your TransportProviderFactory implementation class.

Note: Transport providers typically are not required to manage the life cycle inside Workshop for WebLogic, so there is no need to extend the org.eclipse.core.runtime.Plugin class like a regular plug-in.

If you are providing help for your custom transport in Workshop for WebLogic, you will also have help entries in plugin.xml. For more information, see "Creating Help for Custom Transports" on page 3-31.

MANIFEST.MF

Listing 4-6 shows the sample socket transport MANIFEST.MF file. The file is located at ALSB_HOME/samples/servicebus/sample-transport/eclipse/META-INF/.

Listing 4-6 Sample MANIFEST.MF File

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Socket Transport Plug-in
Bundle-SymbolicName: Socket_Transport;singleton:=true
Bundle-Version: 3.0.0.0
Bundle-Localization: plugin
Bundle-ClassPath: .,
     lib/sock_transport.jar
Require-Bundle: com.bea.alsb.core
```

Build.xml

Listing 4-7 shows the target to package the sample socket transport for Workshop for WebLogic. The file is located at ALSB_HOME/samples/servicebus/sample-transport/.

Listing 4-7 Sample Transport Packaged for Workshop for WebLogic

```
<target name="create_plugin">
     <mkdir dir="${build.dir.transports.sock.eclipse}/plugin"/>
     <copy todir="${build.dir.transports.sock.eclipse}/plugin"
          overwrite="true">
          <fileset dir="${src.dir.transports.sock.eclipse}"/>
     </copy>

     <mkdir dir="${build.dir.transports.sock.eclipse}/plugin/lib"/>
     <copy file="${sock.transport.jar}"
          todir="${build.dir.transports.sock.eclipse}/plugin/lib"
          overwrite="true"/>

     <zip destfile="${sock.transport.eclipse.jar}" basedir="${build.dir.
          transports.sock.eclipse}/plugin"/>
</target>
```

The `build-jar` target also builds the sample socket transport plug-in, along with the deployable transport EAR.

For more information on building the sample socket transport, see Chapter 6, “Sample Socket Transport Provider.”

TransportManagerHelper Methods

Listing 4-8 shows the TransportManagerHelper methods.

Listing 4-8 TransportManagerHelper Methods

```
public static Set<String> getDispatchPolicies(JMXConnector connector)
     throws TransportException;

     public static DomainRuntimeServiceMBean
     getDomainRuntimeService(JMXConnector connector)
          throws TransportException;
```

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
- Schema-Generated Interfaces
- General Classes and Interfaces
- Metadata and Header Representation for Request and Response Messages
- User Interface Configuration

Introduction

The classes and interfaces discussed in this chapter are located in `BEA_HOME/lib/sb-kernel-api.jar` unless otherwise noted.

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/ALSB_HOME/lib/sb-schemas.jar` where `BEA_HOME` is the directory in which you installed Oracle Service Bus.

- **EndPointConfiguration** – The base type for endpoint configuration. An endpoint is an Oracle 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 Oracle 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. This class is located in `BEA_HOME/modules/com.bea.common.configfwk_<version>.jar`.
- **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. This class is located in `BEA_HOME/modules/com.bea.common.configfwk_<version>.jar`.
- **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. This class is located in `BEA_HOME/modules/com.bea.common.configfwk_<version>.jar`.
- **class NonQualifiedEnvValue** – 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. This class is located in `BEA_HOME/modules/com.bea.common.configfwk_<version>.jar`.

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 Oracle 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 Oracle 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 3-30.

- **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 Oracle 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 Oracle Service Bus. The transport provider can call its methods to fetch a credential used for outbound authentication.
- **interface TransportEndPoint** – A transport endpoint is an Oracle 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 Oracle 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, Oracle 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, Oracle Service Bus calls `getInputStream()` only once; however, Oracle 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 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 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
- Summary of UI Interfaces
- Summary of UI Classes

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 Oracle 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.
- **interface CustomHelpProvider** – Lets you provide context-sensitive help for functionality you add to the Oracle Service Bus console, such as custom transports. For implementation details, see Providing Custom Transport Help in the Oracle Service Bus Console.

Summary of UI Classes

- **class TransportUIContext** – Supplies options for the transport provider specific user interface. It is passed by Oracle Service Bus Console to each transport provider.
- **class TransportUIGenericInfo** – Holds transport specific UI information for the common transport page in the Oracle 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 Oracle 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 Oracle Service Bus Console service registration wizard.
- **class TransportUIError** – Returns validation errors to the Oracle Service Bus Console.

Sample Socket Transport Provider

This chapter explains how to build and run the sample socket transport provider. This sample is installed along with Oracle 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
- Sample Location and Directory Structure
- Building and Deploying the Sample
- Start and Test the Socket Server
- Configuring the Socket Transport Sample
- Testing the Socket Transport Provider

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
- Basic Architecture of the Sample
- Configuration Properties

Concepts Illustrated by the Sample

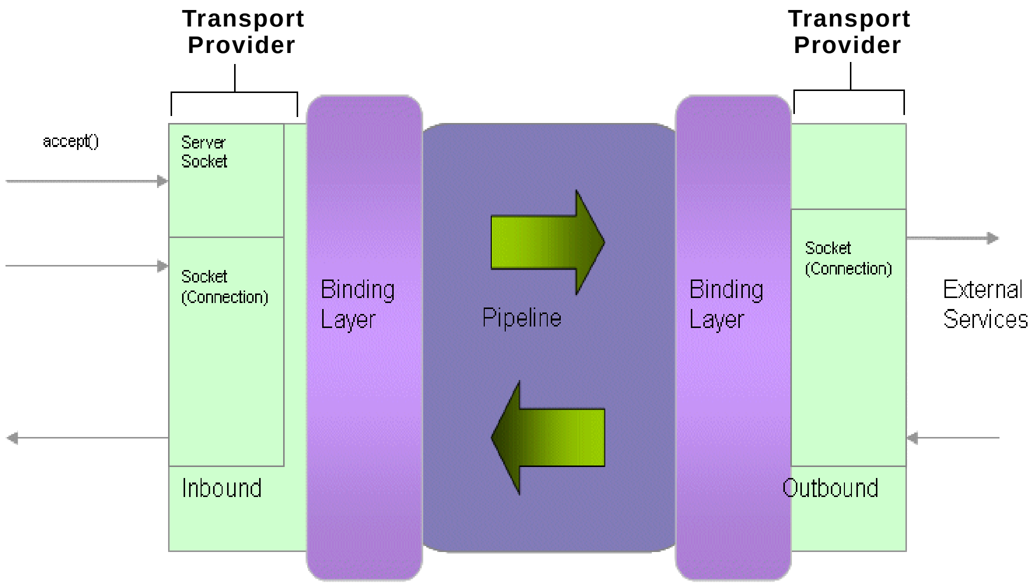
The sample transport is designed to send and receive streamed data to and from a configured TCP socket in Oracle Service Bus. The sample transport is intended to illustrate the following Transport SDK concepts:

- 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 Oracle 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

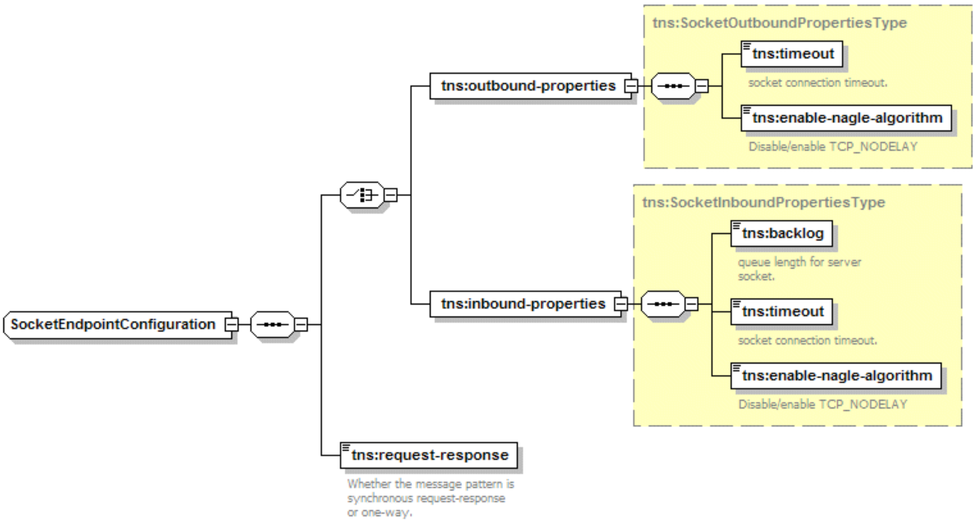


Configuration Properties

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 Oracle Service Bus and is located in the following directory:

`ALSB_HOME/samples/servicebus/sample-transport`

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

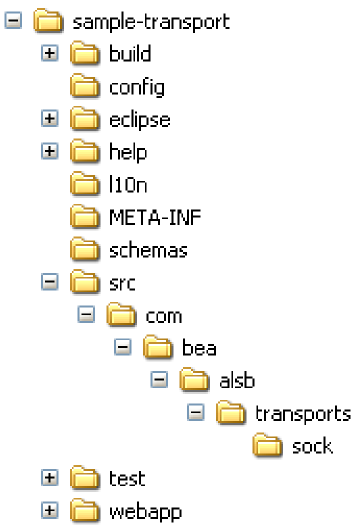


Table 6-1 lists and briefly describes the sample-transport directories.

Table 6-1 Sample Transport Provider Directories

build	Created when you build the sample socket transport. Contains the built and packaged transport for use in Oracle Service Bus.
config	Configuration files directory. `SocketConfig.xml` – Socket transport provider configuration that is used by the Transport SDK.
eclipse	Contains the plugin.xml file needed to add the sample transport to the Workshop for WebLogic environment.

Table 6-1 Sample Transport Provider Directories

help	
l10n	Contains Internationalization files: `SocketTransportMessages.xml` – Configuration file for text messages which are displayed on the Oracle Service Bus Console. `SocketTransportTextMessages.xml`
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
webapp	Contains the deployment descriptors required for the sample transport help Web application.

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 Oracle 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:

   `ALSB_HOME/samples/servicebus/sample-transport`

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

This command compiles the source files and does the following:

- Creates `sock_transport.ear` and copies it to `BEA_HOME/ALSB_HOME/lib`.
- Creates `/sample-transport/build/Socket_Transport_3.0.0.0.jar` as a plug-in to Workshop for WebLogic and copies it to `ALSB_HOME/eclipse/plugins`.

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 Oracle 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 Oracle 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 Oracle 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 Oracle Service Bus to receive and process the message.

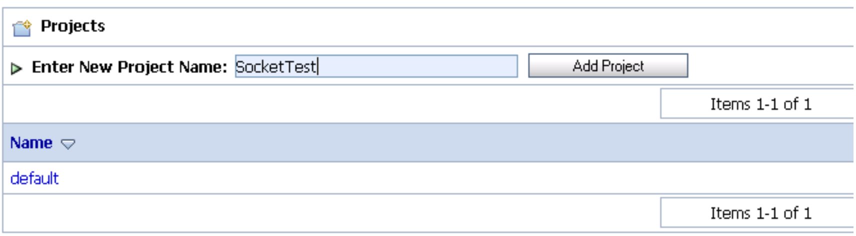
This section describes these tasks:

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

Create a New Project

1. Start the Oracle 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



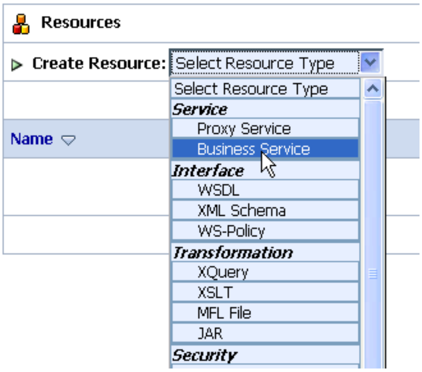
5. Click **Add Project**. The new project appears in the project table.

Create a Business Service

Create a business service to talk to the server.

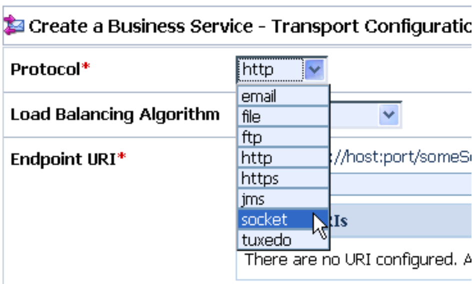
1. Click the SocketTest project name in the project table. The SocketTest panel appears.
2. From the Create Service dropdown menu, select **Business Service**, as shown in Figure 6-6. The General Configuration panel appears.

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**.

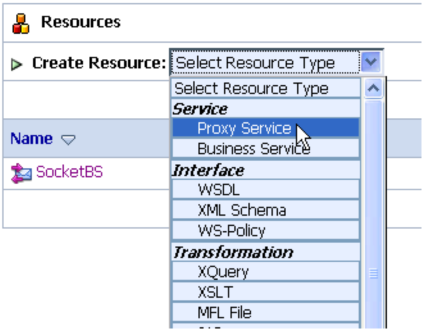
10. In the Change Center, click **Activate**.

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**.

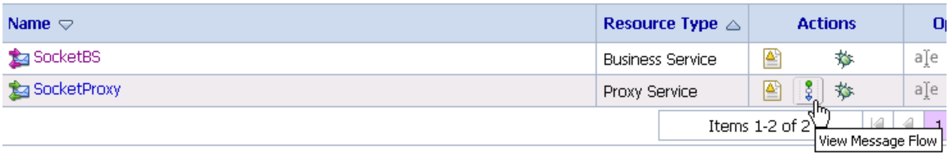
Edit the Pipeline

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**.

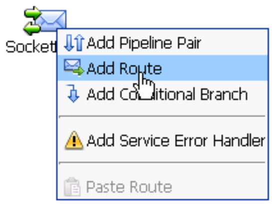
2. In the Resources section, click the **View Message Flow** icon in the SocketProxy row, as shown in Figure 6-9.

Figure 6-9 Selecting the Message Flow Icon



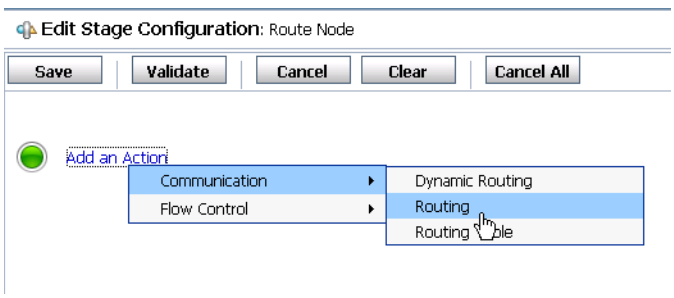
3. In the Edit Message Flow window, click the **SocketProxy** icon and select **Add Route** from the menu, as shown in Figure 6-10.

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.

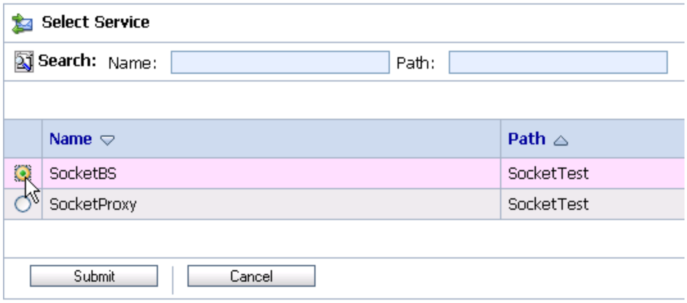
Figure 6-11 Adding an Action



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

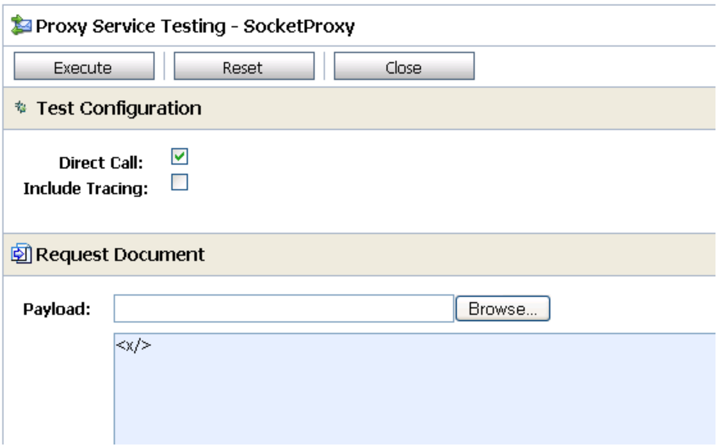
In this section you test the transport provider using Oracle Service Bus Console.

1. Start the test server, as explained previously in "Start the Socket Server" on page 6-8.
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

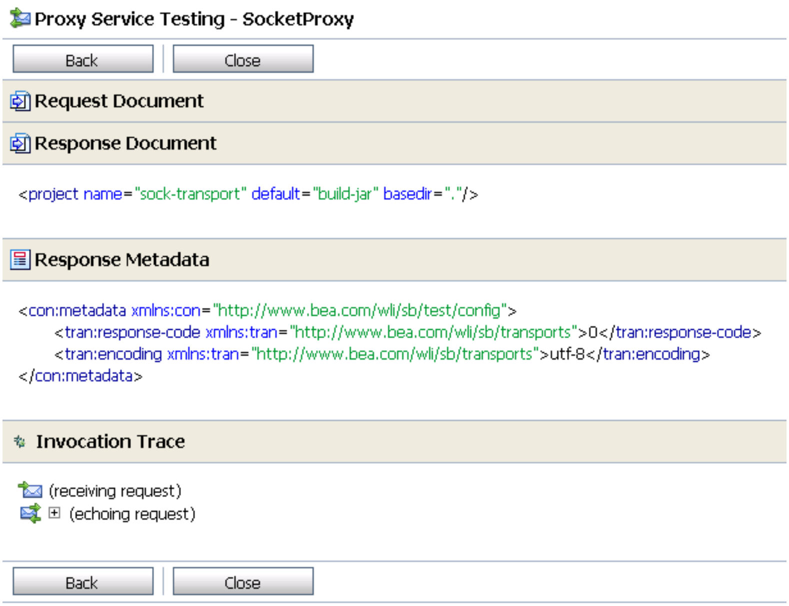
4. In the Test Console, enter any valid XML stanza in the text area, or use the **Browse** button to select a valid XML file on the local system. For example, in Figure 6-14, a simple XML expression <x/> is entered in the text area.

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.

CHAPTER 7

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

You must package your custom transport provider as a self-contained EAR file. You can then deploy the EAR with the Oracle Service Bus Kernel EAR and other Oracle 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 Oracle 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.

- **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 7-3 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 Oracle Service Bus, see the Oracle Service Bus Deployment Guide.

After you create a deployable EAR file for your transport provider, you need to deploy it to the Oracle 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 7-1 to the Oracle Service Bus domain `config.xml` file

Listing 7-1 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 Oracle Service Bus Kernel EAR is deployed before the transport provider.

Transport Registration

On server restart, you want to ensure that your deployed transport can immediately begin to handle service requests. To ensure immediate transport availability, extend the `weblogic.application.ApplicationLifecycleListener` class and use the `preStart()` method to register your transport using `TransportManager.registerProvider()`.

The sample socket transport has an `ApplicationListener` class that you can use for reference, located at ALSB_HOME/samples/servicebus/socket-transport/src/com/bea/alsb/transports/sock.

When extending `ApplicationLifecycleListener`, be sure to register your extending class in META-INF/weblogic-application.xml. The sample socket transport provides the following entry for its ApplicationListener class in ALSB_HOME/samples/servicebus/sample-transport/META-INF/weblogic-application.xml:

```
<weblogic-application>
  <listener>
    <!-- This class gives callbacks for the deployment lifecycle and socket
    transport is registered with ALSB whenever the application is started.
    -->
    <listener-class>com.bea.alsb.transports.sock.ApplicationListener
    </listener-class>
  </listener>
</weblogic-application>
```

Undeploying a Transport Provider

Once a transport provider has been registered with Oracle Service Bus, 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 Oracle 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.

UML Sequence Diagrams

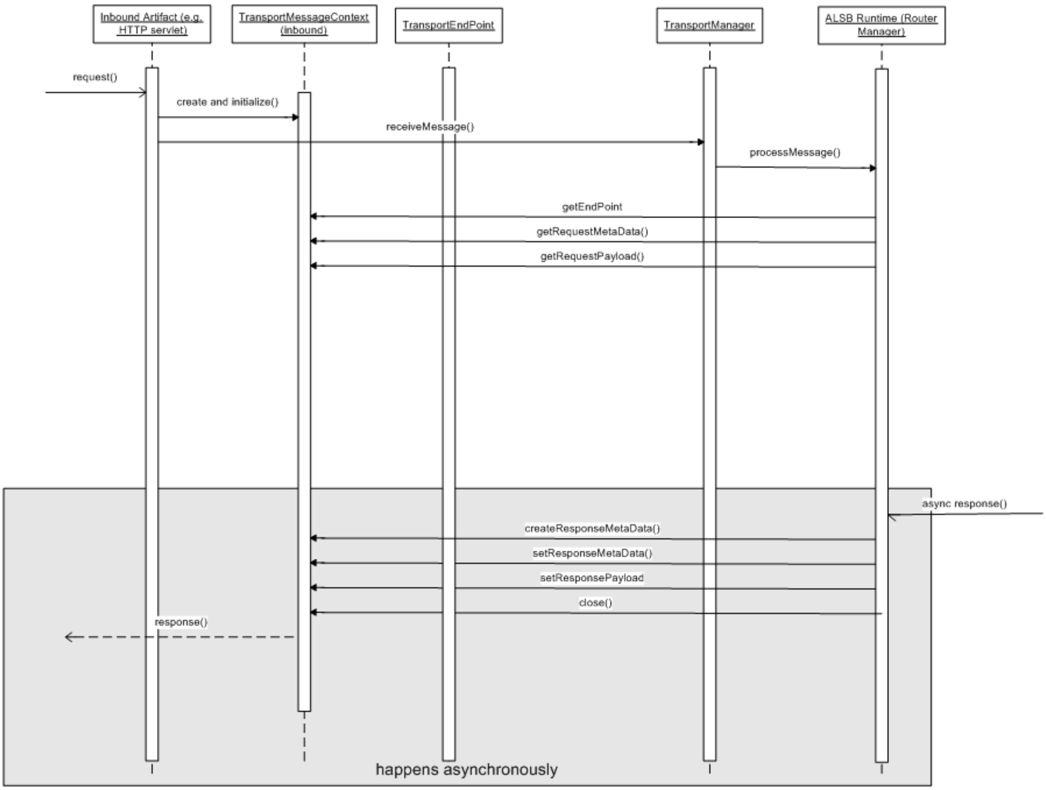
This chapter contains UML sequence diagrams that describe the flow of method calls through Oracle Service Bus runtime.

Oracle Service Bus Runtime Inbound Messages

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



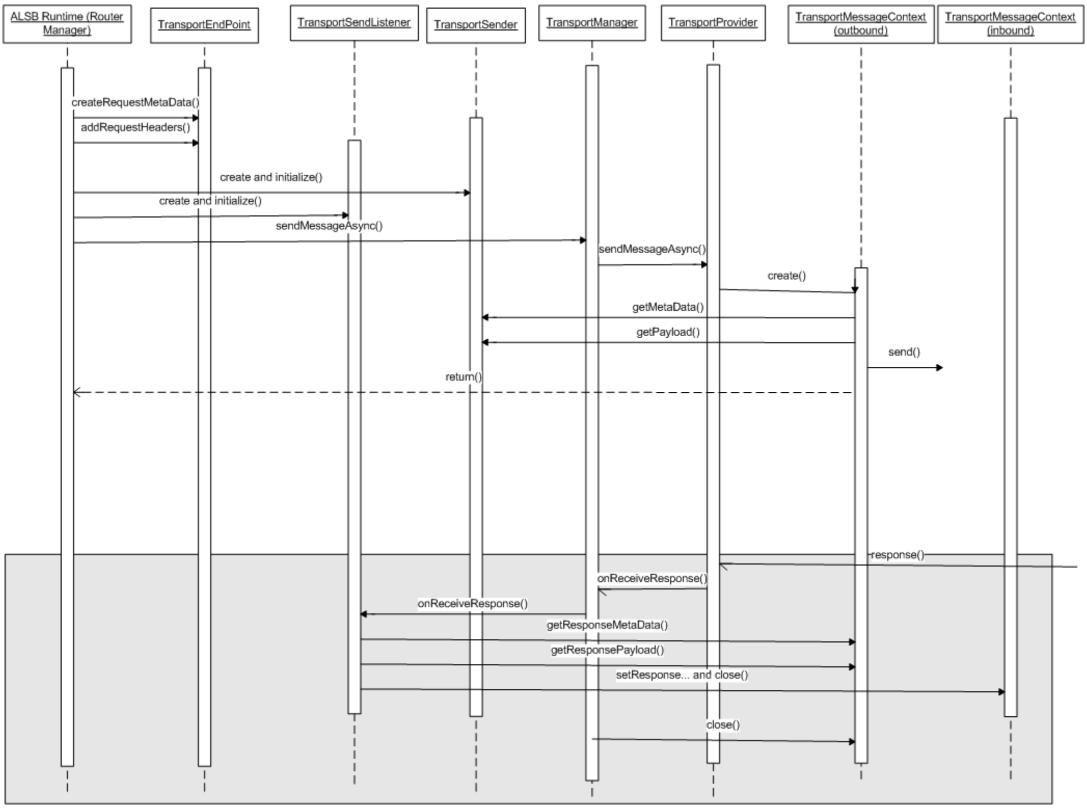
Oracle Service Bus Runtime Outbound Messages

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

The Oracle 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 Oracle 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 Oracle 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