SeeBeyond ICAN Suite

e*Way Intelligent Adapter for SOAP User's Guide

Release 5.0.5 for Schema Run-time Environment (SRE)

SEEBEYOND®

Contents

Chapter 1

Introduction

This chapter introduces you to SeeBeyond™ Technology Corporation's (SeeBeyond™) e*Way™ Intelligent Adapter for SOAP (SOAP e*Way). It also provides an overview of the Simple Object Access Protocol (SOAP) and how to use this e*Way.

1.1 SOAP e*Way: Overview

The SOAP e*Way enables the e*Gate system to exchange data with Internet and Web Services applications that exchange information using SOAP. The e*Way provides SOAP receiver and sender synchronous (RPC) and asynchronous messaging support for the messaging framework, using HTTP transport bindings.

Note: *This e*Way is enabled by the Java programming language.*

1.2 Introduction to SOAP

SOAP, based on the Extensible Markup Language (XML), is a lightweight protocol for the exchange of information in a distributed, decentralized environment. SOAP specifies how to create an XML file and the encoding for HyperText Transfer Protocol (HTTP). This protocol enables an application to communicate over the Internet regardless of the operating system (OS), object model, or implementation language.

SOAP is similar to IIOP, CORBA, and RMI. However, in contrast to these protocols, SOAP has been designed to be fire-wall friendly, lightweight, and easy to implement.

1.2.1 Conventions and Specifications

SOAP defines a set of conventions for the following purposes:

- To format its own messages
- To contain rules for carrying a SOAP message within or on top of another protocol
- To process SOAP messages along the SOAP message path

SOAP specifications can be found on World Wide Web Consortium (W3C) Web site as follows:

- SOAP Version 1.2, Part 1: Messaging Framework:

  http:\\www.w3.org\TR\2001\WD-soap12-part1-20011002

- SOAP Version 1.2, Part 2: Adjuncts:

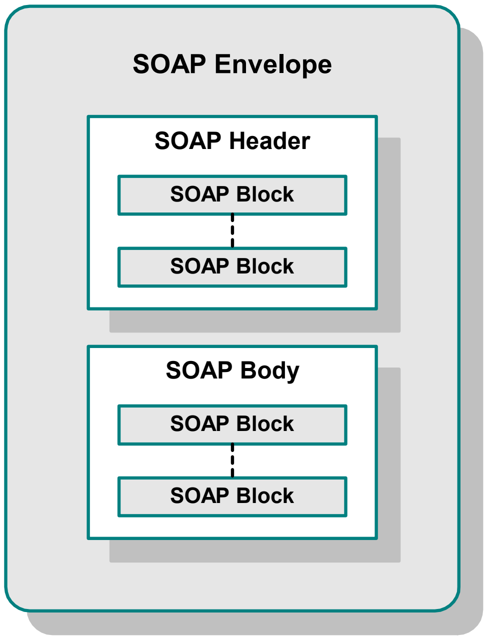
  http:\\www.w3.org\TR\soap12-part2\

Note: *The SOAP e*Way is compatible with both SOAP versions 1.1 and 1.2.*

1.2.2 SOAP Messaging

Figure 1 shows a diagram of the SOAP message components.

Figure 1 Soap Message Components



SOAP Message Structure

SOAP messages (see Figure 1) consist of the following major parts:

- A required SOAP envelope that marks the start and end of the SOAP message and defines a framework for describing what is in a message and how to process it
- An optional SOAP header that carries general information about the SOAP message in one or more header blocks
- A required SOAP body made up of one or more blocks that carry the actual message payload

SOAP messages also specify:

- A set of encoding rules for expressing instances of application-defined data types
- A convention for representing remote procedure calls and responses

Additionally, a special type of SOAP body block, a SOAP fault, is used to carry error and/or status information. If it is present, a SOAP fault (body block) occurs only once.

Example: SOAP Message

Sample request envelope

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getQuote xmlns:ns1="urn:xmethods-delayed-quotes" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<symbol xsi:type="xsd:string">IBM</symbol>
</ns1:getQuote>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Sample response envelope

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getQuoteResponse xmlns:ns1="urn:xmethods-delayed-quotes" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">133.625</return>
</ns1:getQuoteResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A universal SOAP standard is in “working draft” status with W3C, but it is backed by many leading organizations. SOAP is a major building block for emerging Web Services. Web Services are the next generation model for businesses using the Internet to allow business services and functions to be accessed by other applications across the Internet.

You can learn more about SOAP by visiting the following Web sites:

- **http:\\www.xmethods.net**
- **http:\\www.webservices.org**
- **http:\\www.ibm.com\developerworks\webservices\**
- **http:\\msdn.microsoft.com\soap**
- **http:\\www.develop.com\soap**
- **http:\\www.soapware.org**

1.3 e*Way Components and Features

This section provides an overview of the SOAP e*Way including its basic components and features.

1.3.1 Supported Features

This version of the SOAP e*Way supports the following features:

- SOAP version 1.2 and 1.1 messaging
- Message transport on top of HTTP(S)
- Messages with attachments
- Digital signatures
- Web Services Description Language (WSDL)

Note: *When referring specifically to HTTP clear, this guide uses the term HTTP. For HTTP over SSL, that is, secure HTTP, it uses the term HTTPS. For generic HTTP that can be either clear or secure, it uses the term HTTP(S).*

See **Chapter 6** for more information on the SOAP e*Way's additional features.

1.3.2 Basic Components

The SOAP e*Way includes the following components:

- **stceway.exe** file for the Multi-Mode e*Way (core e*Gate component), which provides all the basic e*Way functions and invokes **stcsoap.jar**; for more information on the Multi-Mode e*Way, see **Chapter 3**.
- Setup **.jar** files that contain the logic required to implement SOAP e*Way features
- Third-party libraries, including **activation.jar**, **mail.jar**, **jsafe.jar**, **certj.jar**, and **xalan.jar**, listed in **Table 1 on page 18**.

1.4 Supported Operating Systems

The SOAP e*Way is available on the following operating systems:

- Windows 2000, Windows XP, and Windows Server 2003
- HP Tru64 V5.1A
- HP UX 11.0 and 11i (11.11) on PA-RISC, and 11i v2.0 (11.23)
- IBM AIX 5.1L and 5.2
- Sun Solaris 8 and 9

1.5 System Requirements

To use the SOAP e*Way, you need to meet the following requirements:

- An e*Gate Participating Host
- A TCP/IP network connection

The e*Way must be configured and administered using the e*Gate Schema Designer.

Note: *Additional disk space can be required to process and queue the data that this e*Way processes. The amount necessary can vary based on the type and size of the data being processed and any external applications doing the processing.*

1.6 External System Requirements

The SOAP e*Way supports the following external systems:

- To use the e*Way, you need a SOAP service on the network/Internet
- To use the SOAP e*Way sample schemas, you need access to the following Web site:

  www.xmethods.net

Additional Requirements

Use of the SOAP e*Way requires using the e*Gate API Kit and the HTTP(S) e*Way Intelligent Adapter. See the *e*Gate API Kit User's Guide* and the *HTTP(S) e*Way Intelligent Adapter User's Guide* for more information.

Chapter 2

Installation

This chapter explains how to install the e*Way Intelligent Adapter for SOAP.

2.1 Windows

2.1.1 Pre-installation

- Exit all Windows programs before running the setup program, including any anti-virus applications.
- You must have Administrator privileges to install this e*Way.

2.1.2 e*Way Installation Procedure

To install the SOAP e*Way on Windows Systems

1. Log in as an Administrator on the workstation where you want to install the e*Way.
2. Insert the e*Way installation CD-ROM into the CD-ROM drive.
3. If the CD-ROM drive's **Auto-run** feature is enabled, the setup application should launch automatically; skip ahead to step 4. Otherwise, use Windows Explorer or the Control Panel's **Add/Remove Applications** feature to launch the **setup.exe** file on the CD-ROM drive.
4. After the **InstallShield** setup application launches, follow the on-screen instructions to install the e*Way.

Note: *When you select the SOAP e*Way, the installation automatically selects the HTTP(S) e*Way and the e*Gate Integrator API Kit for installation.*

Be sure to install the e*Way files in the suggested **\client** installation directory. The installation utility detects and suggests the appropriate installation directory.

Caution: *Unless you are directed to do so by SeeBeyond support personnel, do not change the suggested installation directory setting.*

Once you have installed and configured this e*Way, you must incorporate it into a schema by defining and associating the appropriate Collaborations, Collaboration Rules, Intelligent Queues (IQs), and Event Types before this e*Way can perform its intended functions.

2.2 UNIX

2.2.1 Pre-installation

You do not require root privileges to install this e*Way. Log in under your name with which you wish to own the e*Way files. Be sure that this user has sufficient privileges to create files in the e*Gate directory tree.

2.2.2 Installation Procedure

To install the SOAP e*Way on a UNIX system

1. Log in on the workstation containing the CD-ROM drive, and insert the CD-ROM into the drive.
2. If necessary, mount the CD-ROM drive.
3. At the shell prompt, type:

   cd /cdrom/setup
4. Start the installation script by typing:

   setup.sh
5. A menu of options appears. Select the **e*Gate Add-on Applications** option. Then, follow any additional on-screen directions.

Note: *When you select the SOAP e*Way, the installation automatically selects the HTTP(S) e*Way and the e*Gate Integrator API Kit for installation.*

Be sure to install the e*Way files in the suggested **\client** installation directory. The installation utility detects and suggests the appropriate installation directory.

Caution: *Unless you are directed to do so by SeeBeyond support personnel, do not change the suggested installation directory setting.*

Once you have installed and configured this e*Way, you must incorporate it into a schema by defining and associating the appropriate Collaborations, Collaboration Rules, Intelligent Queues (IQs), and Event Types before this e*Way can perform its intended functions.

2.3 After Installation

The SOAP e*Way installation automatically installs the e*Gate API Kit. After installing the e*Way, you must follow the instructions in the *e*Gate API Kit User's Guide* and copy the appropriate files to their Web server directories before you can use the SeeBeyond JMS IQ Service.

Note: *See the **e*Gate API Kit User's Guide** for more information on this feature.*

In addition, to run some of the sample e*Gate schemas, you must place the appropriate ASP files included in the sample schema **.zip** file in the correct directories. The configuration of your Web server determines these directory locations.

2.4 Files/Directories Created by the Installation

Whether for Windows or UNIX, the SOAP e*Way installation installs the files shown in Table 1 within the e*Gate directory tree. Files are installed within the **eGate** directory on the Participating Host and committed to the "default" schema on the Registry Host.

Table 1 Installation Files and Directories

e*Gate Directory	File
client\classes\	stcBasicXsdTypes.jar stcsoap.jar stcutil.jar stcWebServices.jar stcWebServicesBinders.jar stcWebServicesMsghandler.jar stcWebServicesTransport.jar stcwsdl.jar
client\etd	soap.ctl soapwizard.ctl stcewsoap.ctl
client\etd\ewsoap	SOAPSimple.xsc SOAPSimple.jar
client\Thirdparty\jaf-1.0.1\classes\	activation.jar
client\Thirdparty\javamail-1.2\classes\	mail.jar
client\Thirdparty\RSA\certJ_2.0.1\classes\	certj.jar xalan.jar
client\Thirdparty\RSA\cryptoj_3.3\classes\	jsafe.jar
server\registry\repository\default\	addonconnpt.ini
server\registry\repository\default\classes\	stcsoap.jar stcutil.jar

Table 1 Installation Files and Directories (Continued)

e*Gate Directory	File
server\registry\repository\default\configs\ewsoap\	ewsoap.def
server\registry\repository\default\Thirdparty\jaf-1.0.1\classes\	activation.jar
server\registry\repository\default\Thirdparty\javamail-1.2 \classes\	mail.jar
server\Thirdparty\server\repository\default\Thirdparty\RSA \certJ.2.0.1\classes	certj.jar xalan.jar
server\Thirdparty\server\repository\default\Thirdparty\RSA \cryptoj_3.3\classes	jsafe.jar

Chapter 3

Multi-Mode e*Way Configuration

This chapter describes how to configure the e*Gate Integrator's Multi-Mode e*Way Intelligent Adapter.

3.1 Multi-Mode e*Way Properties

Set the Multi-Mode e*Way properties using the e*Gate Schema Designer.

To set properties for a new Multi-Mode e*Way

1. Select the Navigator pane's Components tab in the Main window of the Schema Designer.
2. Open the host and Control Broker where you want to create the e*Way.
3. On the Palette, click on the icon to create a new e*Way.
4. Enter the name of the new e*Way, then click **OK**.
5. Select the new component, then click the Properties icon to edit its properties.

   The e*Way Properties dialog box opens
6. Click **Find** beneath the **Executable File** field, and select an executable file (**stceway.exe** is located in the **bin** directory).
7. Under the **Configuration File** field, click **New**.

   The e*Way Configuration Editor window opens.
8. When the **Settings** page opens, set the configuration parameters for this e*Way's configuration file (see **"JVM Settings" on page 21** and **"General Settings" on page 24** for details).
9. After selecting the desired parameters, click **Save** on the **File** menu to save the configuration (**.cfg**) file.
10. Close the **.cfg** file and e*Way Configuration Editor.
11. Set the properties for the e*Way in the **e*Way Properties** dialog box.
12. Click **OK** to close the dialog box and save the properties.

3.2 JVM Settings

To correctly configure the Multi-Mode e*Way for the e*Way Intelligent Adapter for SOAP, you must configure the Java Virtual Machine (JVM) settings. This section explains the configuration parameters in the e*Way Configuration Editor, which controls these settings.

JNI DLL Absolute Pathname

Description

Specifies the absolute path name to where the JNI **.dll** (Windows) or shared library (UNIX) file is installed by the Java SDK on the Participating Host. This parameter is *mandatory.*

Required Values

A valid path name.

Additional Information

The JNI **.dll** or shared library file name varies, depending on the current operating system (OS). The following table lists the file name by OS:

Operating System	Java 2 JNI .dll or Shared Library Name
Windows systems	jvm.dll
Solaris	libjvm.so
HP-UX	libjvm.sl
AIX	libjvm.a
Compaq	libjvm.so

The value assigned can contain a reference to an environment variable, by enclosing the variable name within a pair of “%” symbols, for example:

```
%MY_JNIDLL%
```

Such variables can be used when multiple Participating Hosts are used on different OS/platforms.

Caution: *To ensure that the JNI **.dll** file loads successfully, the Dynamic Load Library search path environment variable must be set appropriately to include all the directories under the Java SDK installation directory, which contain shared library or **.dll** files.*

CLASSPATH Prepend

Description

Specifies the paths to be prepended to the CLASSPATH environment variable for the JVM.

Required Values

An absolute path or an environmental variable. This parameter is optional.

Additional Information

If left unset, no paths are prepended to the CLASSPATH environment variable.

Existing environment variables may be referenced in this parameter by enclosing the variable name in a pair of % signs. For example:

```
%MY_PRECLASSPATH%
```

CLASSPATH Override

Description

Specifies the complete CLASSPATH variable to be used by the JVM. This parameter is optional. If this parameter is left unset, an appropriate CLASSPATH environment variable (consisting of required e*Gate components concatenated with the system version of CLASSPATH) is set.

Note: *All necessary JAR and ZIP files needed by both e*Gate and the JVM must be included. It is advised that the* ***CLASSPATH Prepend*** *parameter should be used.*

Required Values

An absolute path or an environmental variable. This parameter is optional.

Additional Information

Existing environment variables may be referenced in this parameter by enclosing the variable name in a pair of % signs. For example:

```
%MY_CLASSPATH%
```

CLASSPATH Append From Environment Variable

Description

Specifies whether the path is appended for the CLASSPATH environmental variable to jar and zip files needed by the JVM.

Required Values

YES or **NO**. The configured default is YES.

Initial Heap Size

Description

Specifies the value for the initial heap size in bytes. If this parameter is set to 0 (zero), the preferred value for the initial heap size of the JVM is used.

Required Values

An integer between 0 and 2147483647. This parameter is optional.

Maximum Heap Size

Description

Specifies the value of the maximum heap size in bytes. If this parameter is set to 0 (zero), the preferred value for the maximum heap size of the JVM is used.

Required Values

An integer between 0 and 2147483647. This parameter is optional.

Maximum Stack Size for Native Threads

Description

Specifies the value of the maximum stack size in bytes for native threads. If this parameter is set to 0 (zero), the default value is used.

Required Values

An integer between 0 and 2147483647. This parameter is optional.

Maximum Stack Size for JVM Threads

Description

Specifies the value of the maximum stack size in bytes for JVM threads. If this parameter set to 0 (zero), the preferred value for the maximum heap size of the JVM is used.

Required Values

An integer between 0 and 2147483647. This parameter is optional.

Disable JIT

Description

Specifies whether the Just-In-Time (JIT) compiler is disabled.

Required Values

YES or **NO**.

Remote debugging port number

Description

Specifies whether to allow remote debugging of the JVM.

Required Values

YES or **NO**.

Suspend Option for Debugging

Description

Indicates whether to suspend Option for Debugging on JVM startup.

Required Values

YES or **NO**.

3.3 General Settings

This section contains the parameters for rollback wait and IQ messaging priority.

Note: *For more information on the **General Settings** configuration parameters see the **e*Gate Integrator User's Guide**.*

3.3.1 Rollback Wait Interval

Description

Specifies the time interval to wait before rolling back the transaction.

Required Values

A number within the range of 0 to 99999999, representing the time interval in milliseconds.

3.3.2 Standard IQ FIFO

Description

Specifies whether the highest priority messages from all SeeBeyond Standard IQs are delivered in the first-in-first-out (FIFO) order.

Required Values

Select **Yes** or **No**. **Yes** indicates that the e*Way retrieves messages from all SeeBeyond Standard IQs in the first-in-first-out (FIFO) order. **No** indicates that this feature is disabled; **No** is the default.

Chapter 4

Implementation

This chapter explains how to implement sample schemas for the e*Way Intelligent Adapter for SOAP, both the SOAP sender and receiver, in an e*Gate Integrator system environment.

4.1 SOAP e*Way: Architecture Overview

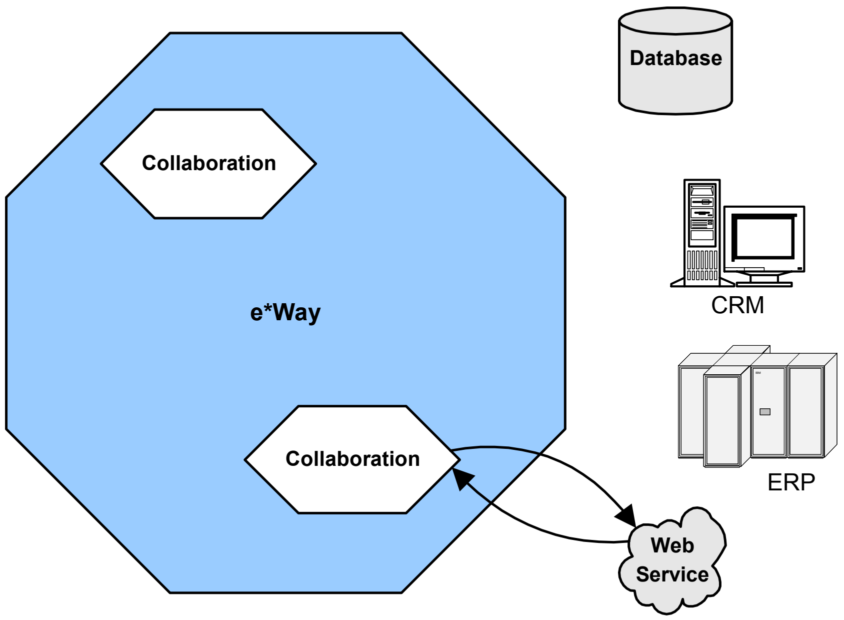
SOAP messaging is essentially the delivery of a message from a SOAP sender to the ultimate SOAP receiver. SOAP messages can be asynchronous, or they can be combined to form a SOAP request/response synchronous message exchange.

This section describes the architectural framework for the implementation of SOAP receivers and senders, regardless of the asynchronous or synchronous nature of the message exchange.

4.1.1 SOAP Sender

Figure 2 on page 26 shows a diagram of the basic SOAP sender setup in e*Gate.

Figure 2 SOAP Sender e*Gate Setup



An e*Gate Collaboration that sends a SOAP message (client mode) is implemented using an e*Gate Multi-Mode e*Way. The SOAP e*Way functionality is used by the Multi-Mode e*Way, along with a SOAP e*Way Connection.

Note: *See* **Chapter 3** *for more information on the Multi-Mode e*Way.*

You can specify the SOAP endpoint, or URL, in the e*Way Connection or dynamically in the Collaboration Rules. Additional HyperText Transfer Protocol (HTTP) binding and SOAP configuration parameters are specified in the e*Way Connection configuration. The SOAP e*Way supports sending request/reply messages, as well as "fire-and-forget" asynchronous messages.

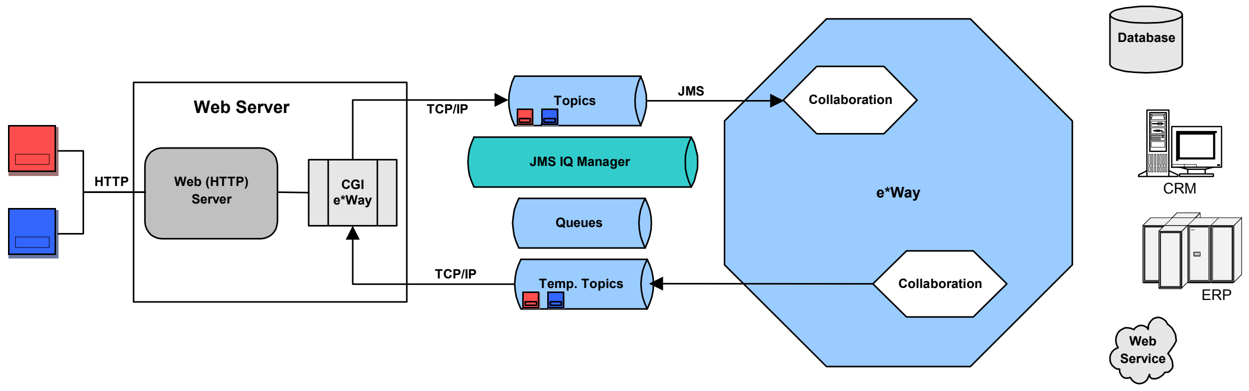
The installation of the SOAP e*Way includes all of the necessary files for sending SOAP messages. Included are the SOAP e*Way and wizard, the HTTP(S) e*Way, and third-party libraries.

Note: *See the* ***HTTP(S) e*Way Intelligent Adapter User's Guide (Java Version)*** *for complete information on this e*Way.*

4.1.2 SOAP Receiver

Figure 3 on page 27 shows a diagram of the basic SOAP receiver setup in e*Gate.

Figure 3 SOAP Receiver e*Gate Setup



In addition to the SOAP e*Way, implementing an e*Gate Collaboration that receives a SOAP message (server mode) requires:

- Use of an HTTP Web server, for example, Apache/Tomcat
- Use of the e*Way's built-in Common Gateway Interface (CGI) or the e*Gate API Kit

The e*Gate API Kit provides a Java Messaging Service (JMS) application programming interface (API) to the SeeBeyond JMS IQ Manager. In addition, the Web server (or application server) provides the HTTP daemon "listener" facility. You can use either a Web-server e*Way or the JMS IQ Manager (used in this chapter's example) to pass SOAP messages into the e*Gate system.

Figure 3 on page 27 illustrates the components of a SOAP request/response message where an e*Gate Collaboration is providing the SOAP service. External clients request the SOAP service using an end-point URL within the Web server space. Apache/Tomcat is an easily available Web server and is commonly used. Apache/Tomcat is included with some SeeBeyond products, but the SOAP Web server can be any HTTP server.

An e*Gate user implementing the SOAP service can use either:

- A plug-in provided with the SOAP e*Way installation, for example, a JavaServer Page (JSP) that implements the e*Gate API Kit for the JMS (API); see the example in this chapter
- User-created code as a base you can use to build custom applications

Web Server Logical Steps

The logical steps on the Web-server side are:

- Capture the HTTP data.
- Create a JMS temporary topic.
- Populate the **JMSReplyTo** header field with the temporary topic.
- Publish the HTTP data to a known topic or queue.
- Subscribe to the temporary topic.
- Wait for the reply message.
- Return the message as the HTTP response body.

Note: *JMS temporary topics and the **JMSReplyTo** header field are JMS features used in a request/reply solution. A temporary topic is a unique, dynamically created topic that is only active for the duration of the connection and is guaranteed to be unique across all connections. Temporary topics are associated only with the message server that the client is in session with. Any client can publish messages to a temporary topic, but only the client connection that created the temporary topic can subscribe to it.*

An e*Gate Collaboration subscribes to the known topic or queue, publishes to the temporary topic found in the **JMSReplyTo** header field, another IQ, or another Collaboration. Depending on the complexity of the service implementation, one or more Collaborations can be involved in processing a SOAP request.

A Collaboration uses a SOAP Event Type Definition (ETD) and the SOAP e*Way to unmarshal the received SOAP message. The Collaboration Rules for the Collaboration use the SOAP ETD to create the response SOAP message. This SOAP message is published to the JMS temporary topic found in the **JMSReplyTo** header field.

e*Gate System Logical Steps

The logical steps on the e*Gate side are:

- Create a schema.
- Create Event Types.
- Create the ETD that receives the message from the JMS IQ Manager.
- Create the SOAP ETD to be used for processing the request and response.
- Create the ETD that sends the reply to the temporary JMS topic.
- Create Collaboration Rules as follows:
    - Drag the source JMS payload onto the SOAP request.
    - Define Java rules to create the SOAP response.
    - Drag the SOAP response onto the JMS **ReplyTo** topic.
- Create e*Way Connections.

This section has described implementing the JMS interface as a request/reply schema. Request/reply is used for both synchronous SOAP request/response messages, as well as asynchronous SOAP messages that expect a SOAP response status message.

To receive true asynchronous "fire-and-forget" SOAP messages, you can implement the JMS publish/subscribe (pub/sub) schema. For more information on the e*Gate API Kit for the JMS (API), see the *e*Gate API Kit Developer's Guide*.

SOAP Services

The implementation of the SOAP e*Way begins with the selection of SOAP service. For the purposes of this chapter, a publicly available SOAP service is used to illustrate the configuration steps for implementing the SOAP e*Way in SOAP remote procedure call (RPC) style.

An RPC-style SOAP message is a synchronous request/reply process where the SOAP e*Way executes a remote SOAP service by passing input parameters and waiting for output parameters.

For a service list of the many publicly available SOAP services, see the following Web site:

www.xmethods.net

4.2 Using the SOAP ETD Wizard

The SOAP ETD wizard takes an Extensible Markup Language (XML) file and converts it to an **.xsc** file (e*Gate ETD) that contains the following elements:

- **SOAP request header element**: SOAP header of the request message
- **SOAP request body element**: SOAP body of the request message
- **SOAP response header element**: SOAP header of the response message
- **SOAP response body element**: SOAP body of the response message

You can use the SOAP ETD wizard to create a generic ETD using well-formed XML but no header or body format (BLOB option), or you can create an ETD from a specific Document Type Definition (DTD) file. If you choose the DTD file, the resulting ETD contains the precise format, syntax, and semantics for the SOAP request and response header and body.

You can also use the SOAP ETD wizard to convert a Web Services Description Language (WSDL) file to an **.xsc** file. WSDL is an XML format for describing network services. It is commonly used to describe the endpoints and message structure used with SOAP. For more information on the SOAP e*Way's WSDL support features, see **"Using the Web Services Description Language" on page 117**.

This section provides the following sets of wizard procedures:

- **Converting a basic SOAP XML file to an ETD**: **"SOAP ETD Wizard: Basic SOAP" on page 30**
- **Converting a SOAP WSDL file to an ETD**: **"SOAP ETD Wizard: WSDL" on page 34**

4.2.1 SOAP ETD Wizard: Basic SOAP

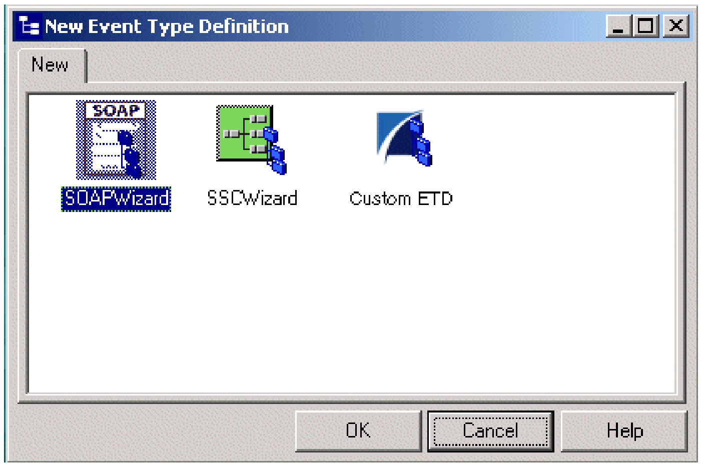
This section explains how to use the SOAP ETD wizard to create a basic SOAP e*Gate ETD (**.xsc**) file.

To convert a basic SOAP file to an ETD

1. From the e*Gate Schema Designer, display the ETD Editor. Be sure you have selected the Java editors as your default.
2. To access the SOAP ETD wizard, click **New** on the ETD Editor's **File** menu.

   The **New Event Type Definition** dialog box appears, displaying all installed ETD wizards (see the example in Figure 4).

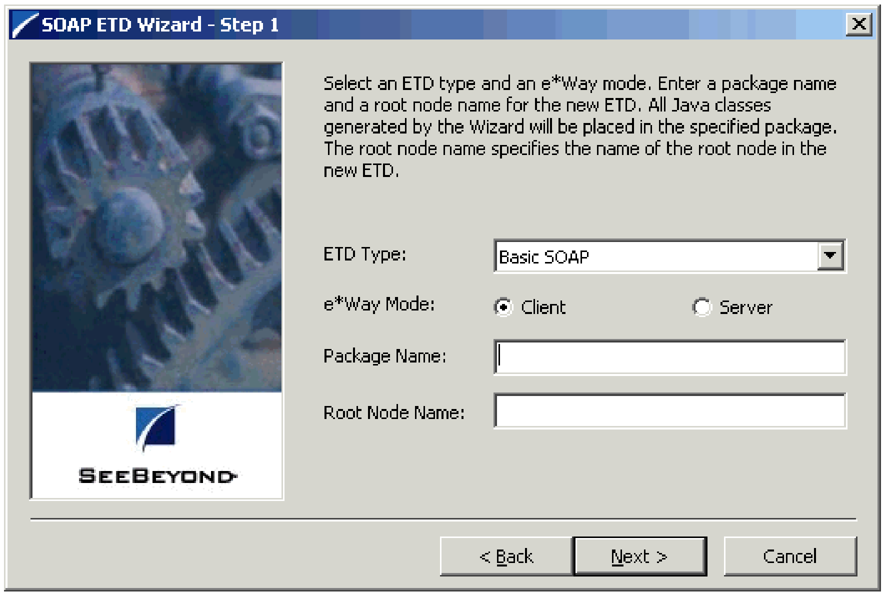
Figure 4 New Event Type Definition Dialog Box



3. Double-click the **SOAPWizard** icon.
4. Review the **SOAP ETD Wizard Introduction** dialog box, then click **Next**.

The **SOAP ETD Wizard - Step 1** dialog box appears (see Figure 5).

Figure 5 SOAP ETD Wizard - Step 1 Dialog Box



Select an ETD type and an e*Way mode. Enter a package name and a root node name for the new ETD. All Java classes generated by the Wizard will be placed in the specified package. The root node name specifies the name of the root node in the new ETD.

ETD Type: Basic SOAP

e*Way Mode: [✓ Client] Server

Package Name:

Root Node Name:

5 On **SOAP ETD Wizard - Step 1** do the following actions:

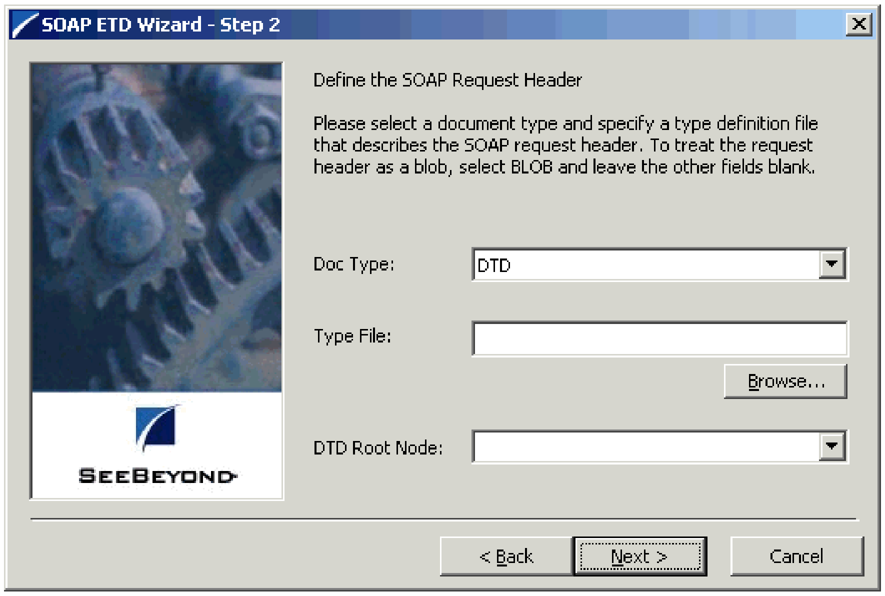
- Select the **Basic SOAP** ETD type (the default).
- Select the appropriate e*Way mode, **Client** or **Server**.
- Enter the desired package name for the container in which the wizard places the generated Java classes.
- Enter the desired root-node name of the ETD.

Note: *Observe Java naming rules in your entries. See the **e*Gate Integrator User's Guide** for details.*

6 Click **Next** to continue.

The **SOAP ETD Wizard - Step 2** dialog box appears (see Figure 6).

Figure 6 SOAP ETD Wizard - Step 2 Dialog Box (SOAP Request Header)



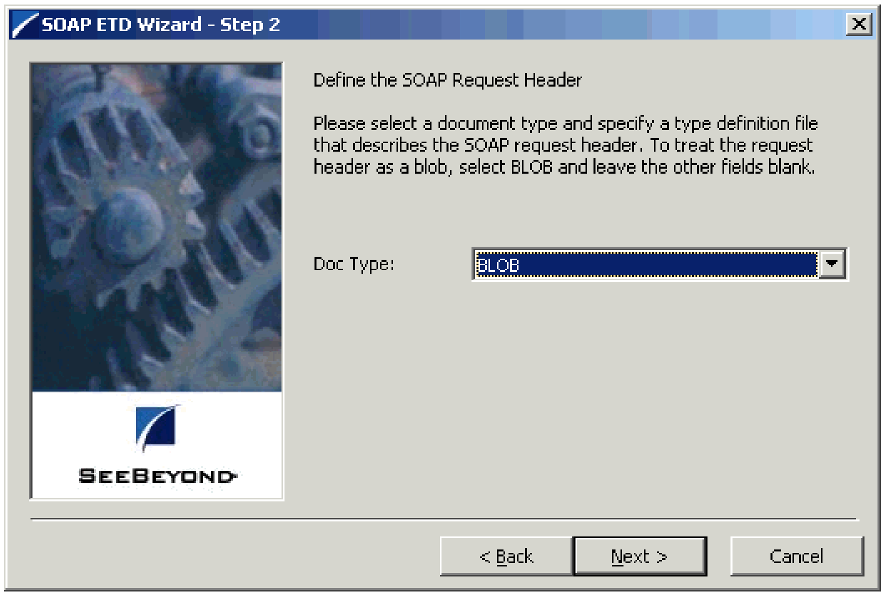
Define the SOAP Request Header

Please select a document type and specify a type definition file that describes the SOAP request header. To treat the request header as a blob, select BLOB and leave the other fields blank.

Doc Type: DTD

Type File:

DTD Root Node:



Define the SOAP Request Header

Please select a document type and specify a type definition file that describes the SOAP request header. To treat the request header as a blob, select BLOB and leave the other fields blank.

Doc Type: BLOB

Note: *There are two "Step 2" wizards, allowing you to define different elements of the SOAP request header.*

7 On the first **SOAP ETD Wizard - Step 2**, do one of the following operations to describe the contents of the SOAP request header element:

- To create a generic SOAP request header element, choose **BLOB** as the document type and continue to the next step.
- To create the SOAP request header element from a DTD file, choose **DTD** as the document type and enter the name of the DTD file. At this point, all top-level elements defined in the DTD file are listed in the **DTD root node** pull-down list. Select the element you want to use for the SOAP request header.

If you do not know the name of the DTD file, click **Browse** to navigate to the desired file. All the elements in the DTD file are listed as possible root-node names for this component.

8 When you are finished, click **Next** to continue.

The same dialog box reappears with only some text changes to allow you to enter new information.

9 On the next **SOAP ETD Wizard - Step 2**, do one of the following actions to describe the contents of the SOAP request body element:

- To create a generic SOAP request body element, choose **BLOB** as the document type and continue to the next step. A type file is not required.
- To create the SOAP request body element from a DTD file, choose **DTD** as the document type, enter the DTD file, and choose a DTD root-node name from the list.

If you do not know the name of the DTD file, click **Browse** to navigate to the appropriate file. All the elements in the DTD file are listed as possible root-node names for this component.

10 When you are finished, click **Next** to continue. The same dialog box reappears with only some text changes to allow you to enter new information.

11 On the next **SOAP ETD Wizard - Step 2**, do one of the following actions to describe the contents of the SOAP response header element:

- To create a generic SOAP response header element, choose **BLOB** as the document type and continue to the next step.
- To create the SOAP response header element from a DTD file, choose **DTD** as the document type, enter the DTD file, and choose a DTD root-node name from the list.

If you do not know the name of the DTD file, click **Browse** to navigate to the appropriate file. All the elements in the DTD file are listed as possible root-node names for this component.

12 When you are finished, click **Next** to continue. The same dialog box reappears with only some text changes to allow you to enter new information.

13 On the next **SOAP ETD Wizard - Step 2**, do one of the following actions to describe the contents of the SOAP response body element:

- To create a generic SOAP response body element, choose **BLOB** as the document type and continue to the next step.
- To create the SOAP response body element from a DTD file, choose **DTD** as the document type, enter the type file, and choose a DTD root-node name from the list.

If you do not know the name of the type file, click **Browse** to navigate to the appropriate file. All the elements in the DTD file are listed as possible root-node names for this component.

14 Click **Finish** when you are done with the wizard.

The structure of the ETD you have created appears in the ETD Editor's Main window.

15 Close the ETD Editor and save the ETD under the your desired name. This is your new SOAP ETD (.**xsc**) file.

4.2.2 SOAP ETD Wizard: WSDL

This section explains how to use the SOAP ETD wizard to convert a SOAP WSDL file to an e*Gate ETD (.**xsc**) file.

To convert a SOAP WSDL file to an ETD

1 Follow steps 1 through 4 in the **procedure on page 30**.

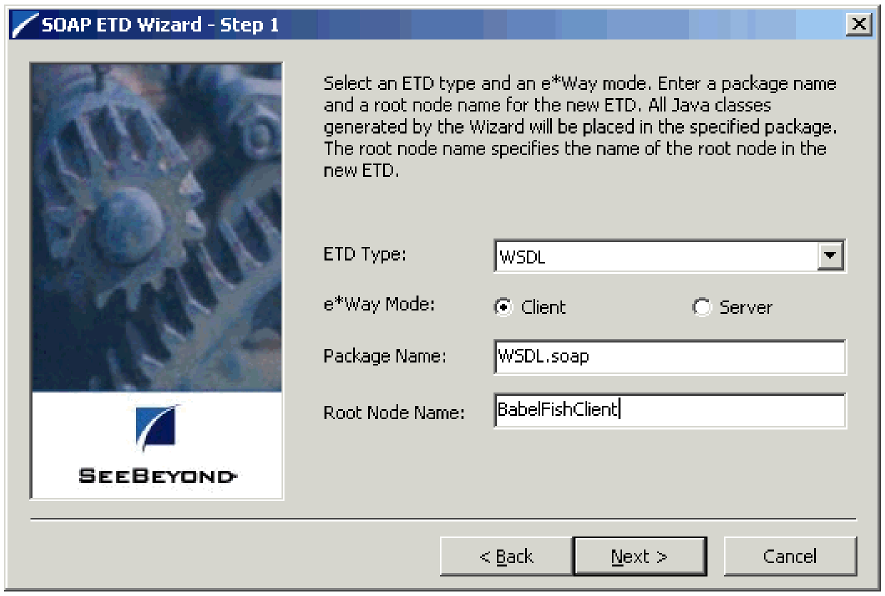
2 On **SOAP ETD Wizard - Step 1** do the following actions:

- Select the **WSDL** ETD type (the default).
- Select the appropriate e*Way mode, **Client** or **Server**.
- Enter the desired package name for the container in which the wizard places the generated Java classes.
- Enter the desired root-node name of the ETD.

Note: *Observe Java naming rules in your entries. See the **e*Gate Integrator User's Guide** for details.*

When you are finished, the dialog box appears as in the Figure 7 example.

Figure 7 SOAP ETD Wizard - Step 1 Dialog Box (WSDL)



Select an ETD type and an e*Way mode. Enter a package name and a root node name for the new ETD. All Java classes generated by the Wizard will be placed in the specified package. The root node name specifies the name of the root node in the new ETD.

ETD Type: WSDL

e*Way Mode: [✓ Client] Server

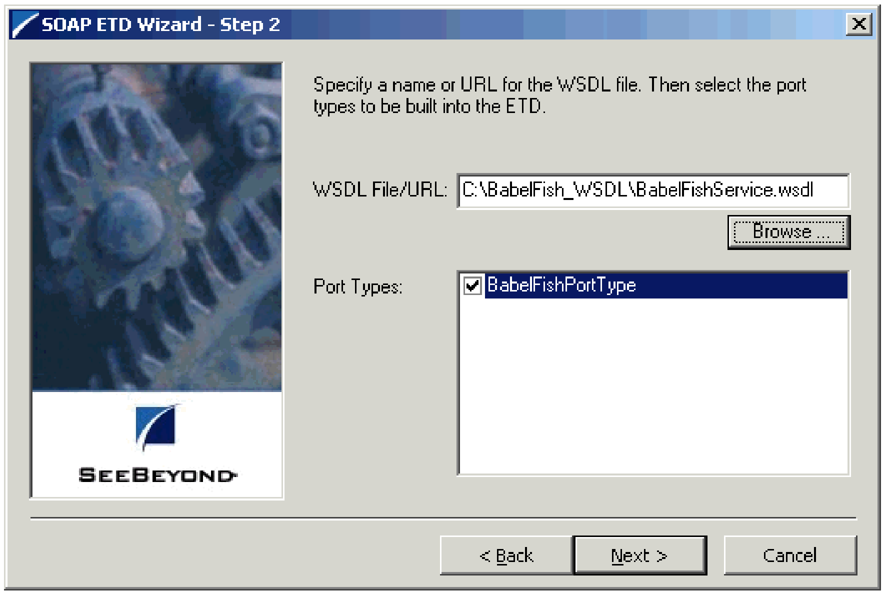
Package Name: WSDL.soap

Root Node Name: BabelFishClient

3 When you are finished, click **Next** to continue.

The **SOAP ETD Wizard - Step 2** dialog box (for WSDL) appears (see the example in Figure 8).

Figure 8 SOAP ETD Wizard - Step 2 Dialog Box (WSDL)



Specify a name or URL for the WSDL file. Then select the port types to be built into the ETD.

WSDL File/URL: C:\BabelFish_WSDL\BabelFishService.wsdl

Port Types: ✓ BabelFishPortType

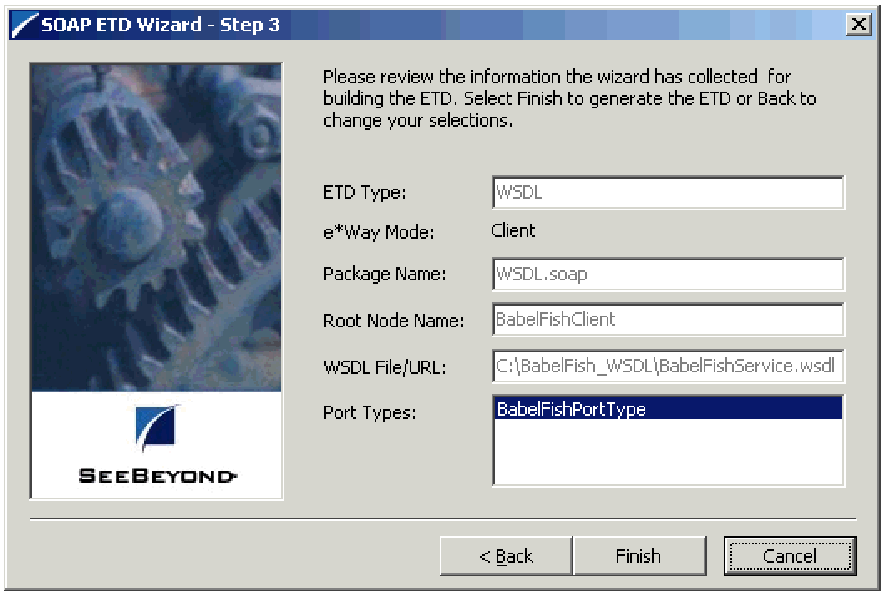
4 Click **Browse** to select the desired WSDL file or URL (see Figure 8). The wizard automatically displays the port types, reading the information from the file.

5 If there is only one port type listed in the file, the type is automatically selected and must be used. However, if more than one type is listed, the options display, and you must select the desired port type.

6 When you are finished, click **Next** to continue.

The **SOAP ETD Wizard - Step 3** dialog box (for WSDL) appears (see the example in Figure 8).

Figure 9 SOAP ETD Wizard - Step 3 Dialog Box (WSDL)



SOAP ETD Wizard - Step 3

Please review the information the wizard has collected for building the ETD. Select Finish to generate the ETD or Back to change your selections.

ETD Type:	WSDL
e*Way Mode:	Client
Package Name:	WSDL.soap
Root Node Name:	BabelFishClient
WSDL File/URL:	C:\BabelFish_WSDL\BabelFishService.wsdl
Port Types:	BabelFishPortType

7 Carefully review the information you have supplied. If it is correct, click **Finish** to create your ETD.

The structure of the ETD you have created appears in the ETD Editor’s Main window.

8 Close the ETD Editor and save the ETD under the your desired name. This is your new SOAP WSDL ETD (**.xsc**) file.

4.3 SOAP Sender Implementation

This section explains how to implement a sample schema for the SOAP e*Way, for the basic SOAP sender. This schema demonstrates how the SOAP e*Way works as a client against a known SOAP service, Altavista's BabelFish Translator.

Note: *For more information on the sample implementations, see the* ***Readme.html*** *file that comes with your SOAP e*Way samples.*

4.3.1 Schema and File

The name of this schema is BabelFish. It is contained in the file **BabelFish.zip** in the following directory on the installation CD-ROM:

samples\ewsoap

All of the samples for this e*Way are located in this directory.

4.3.2 SOAP Sender Schema: Overview

This sample BabelFish schema illustrates the SOAP service interface to Altavista's BabelFish Translator service, which translates one language to another. The service entry is located at:

http:\\www.xmethods.net\detail.html?id=14

Schema Operation

The sample schema does the following operations:

- Retrieves text from a text file on any platform
- Transforms the text into a SOAP message
- Posts the SOAP message to a SOAP server that translates the text of the message into a different language (by default, the SOAP server translates an English message into French); for this sample, the SOAP server is the **www.xmethods.net** Web site
- Receives the translated text from the SOAP server
- Publishes the translated text to a different file

Schema Input Data

The following text is the input data used for this sample schema:

```
Good morning
```

Schema Output Data

The SOAP BabelFish service passes as request, or input parameters, a value for **translationmode** and **sourcedata**. The response, or output parameter, is the value for the return data, for example:

translationmode : en_fr

sourcedata : good morning

return : bonjour

Schema Components

See Table 3 on page 61 for the components' configuration settings. This sample BabelFish schema implementation consists of the following components:

e*Ways

- **Feeder** receives text from an external source, applies the e*Gate Pass Through Collaboration Service, and publishes the information to an Intelligent Queue (IQ) that stores inbound data.
- **SOAPBabelFishClient** applies extended Java Collaboration Rules to an inbound Event to perform the desired business logic. In this case, the e*Way translates the inbound Event into a SOAP message, posts the SOAP message to a SOAP server, receives a translated text response from the SOAP server, and publishes the response to an IQ.
- **Eater** receives the outbound message from an IQ and publishes it (via Pass Through again) to a file.

Event Types

- **Feeder_In_Event** contains raw data from the input file.
- **Feeder_Out_Event** contains raw data from the input file.
- **SOAP_BabelFish_Event** contains the request data, the response data (if any), and the methods used to manipulate the data.
- **Eater_In_Event** contains the translated data.
- **Eater_Out_Event** contains the translated data.

Collaboration Rules

- **FeederCollaboration** is associated with the **Feeder** e*Way and is used for receiving the input Event.
- **SOAPBabelFishClient** is associated with the **SOAPBabelFishClient** e*Way and is used to perform the transformation process, send the Event to the SOAP server, and receive a response from the SOAP server.
- **EaterCollaboration** is associated with the **Eater** e*Way and is used for sending the Event to the output file.

IQs

- **In_Q** receives data from the **Feeder** e*Way and sends it to the **SOAPBabelFishClient** e*Way.
- **Out_Q** receives data from the **SOAPBabelFishClient** e*Way and sends it to the **Eater** e*Way.

Location of Schema Files

To do this implementation, you first need to unzip the **BabelFish.zip** file that contains the schema. The files listed in Table 2 are contained within this file.

Table 2 BabelFish Schema Files

File Name	Description
BabelFish.zip	Export schema file.
BabelFishRequest.dtd	Document Type Definition (DTD) file that describes the BabelFish SOAP request.
BabelFishResponse.dtd	DTD file that describes the BabelFish SOAP response.
Text.~in	Input file.
Readme.txt	Information file.

To use this sample schema, the SOAP e*Way must be installed, the sample schema must be installed, all of the necessary files and scripts must be located in the default location, and the **www.xmethods.net** Web site must be available.

Schema Implementation

To implement this sample schema, you can do one of the following operations:

- To import the sample schema zip file, which automatically creates the sample schema components, see the instructions provided in **"Sample Sender Schema: Automatic Implementation" on page 39**.
- To manually create each of the components required to use the sample schema, see the instructions provided in **"Sample Sender Schema: Manual Configuration" on page 41**.

4.3.3 Sample Sender Schema: Automatic Implementation

This section explains how to automatically implement the SOAP e*Way within a sample sender schema.

Installing and Configuring the Schema

To install and configure the BabelFish sample schema

1. Copy the file named **BabelFish.zip** from the **samples\ewsoap** directory on the install CD-ROM to your desktop or to a temporary directory.
2. Start the e*Gate Schema Designer.
3. On the **Open Schema from Registry Host** dialog box, click **New**.
4. On the **New Schema** dialog box, click **Create from export**, and then click **Find**.
5. On the **Import from File** dialog box, browse to the directory that contains the sample schema, click **BabelFish.zip**, and then click **Open**.

   The sample schema is installed.
6. Configure the **Feeder** e*Way as follows:

   A From e*Gate Schema Designer, display the properties of the **Feeder** e*Way, then click **Edit**. The e*Way Configuration Editor appears; use this interface to configure or modify an e*Way.

   B In the **Goto Section** of the e*Way Configuration Editor, choose **Poller (inbound)** settings.

   C For the **Poll Directory** parameter, specify the path name of the directory that contains the sample input data. This directory is named **\INDATA,** and it is located in the directory where you installed the sample schema.
7. Configure the **Eater** e*Way as follows:

   A From the Schema Designer, display the properties of the **Eater** e*way, and then click **Edit**.

   B In the **Goto Section** of the e*Way Configuration Editor, choose **Outbound (send)** settings.

   C For the **OutputDirectory** parameter, specify the path name of the directory that contains the sample data. This directory is named **\data** and it is located in the directory in which you installed the sample schema.

Running the Schema

To run the BabelFish schema

1. From the command line prompt, enter:

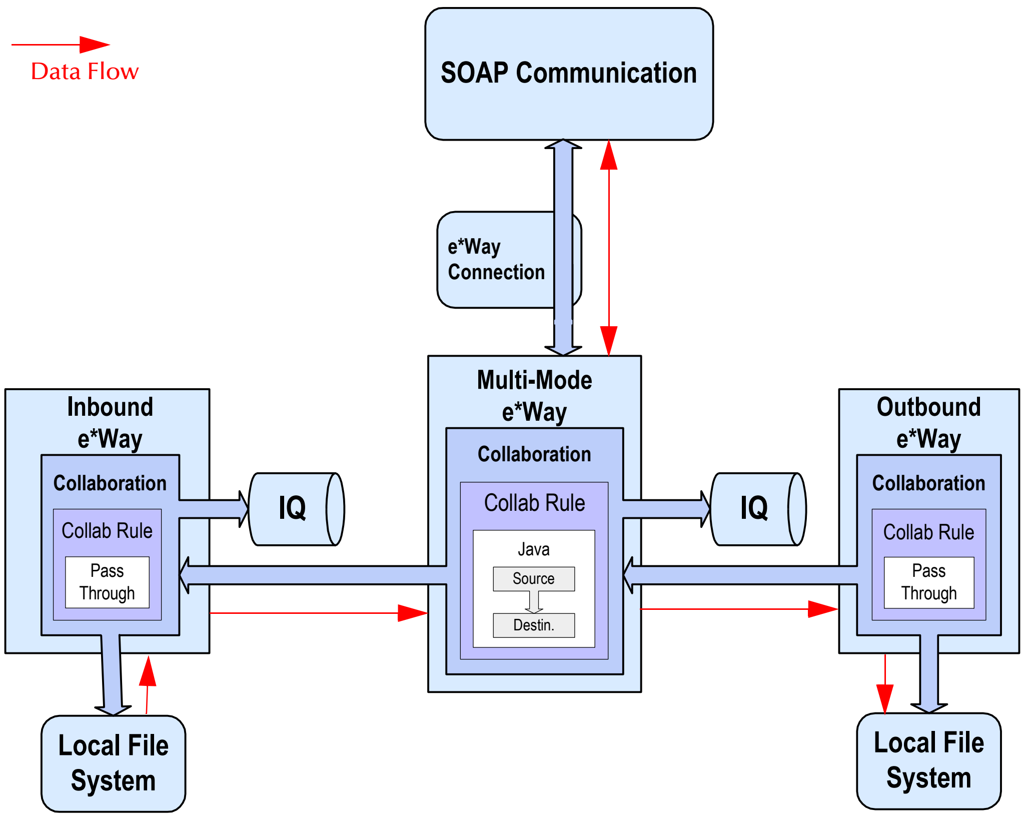
   ```
   stccb -rh hostname -rs schemaname -un username
   -up user password -ln hostname_cb
   ```

   Substitute ***hostname***, ***username***, ***schemaname***, and ***user password*** as appropriate.
2. Change the input file name extension to **.fin**.

The schema components start automatically. When there are no more run-time messages, check the output file. If the schema is operating correctly, this file contains the text translated into French.

Figure 10 shows an overview diagram of the BabelFish schema and how it operates. The blue arrows show publication/subscription (pub/sub) relationships between the components. Red arrows show the actual flow of data.

Figure 10 BabelFish Schema Diagram



4.3.4 Sample Sender Schema: Manual Configuration

This section explains how to configure the BabelFish sender schema manually in e*Gate, starting from the beginning.

Basic Implementation Steps

After you have located the SOAP service description, you must do the following steps:

1. Determine the SOAP endpoint URL.
2. Determine the format of the SOAP message.
3. Create a schema.
4. Create Event Types and Event Type Definitions (ETDs).
5. Add and configure e*Ways.
6. Create the e*Way Connection.
7. Create Collaboration Rules.
8. Add Intelligent Queues (IQs).
9. Create and configure Collaborations.
10. Check and test the schema.

The rest of this section explains each of the previous steps.

Note: *For a complete explanation of how to create an e*Gate schema and its components, see **Creating an End-to-End Scenario with e*Gate Integrator** and the **e*Gate Integrator User's Guide**.*

Determining the SOAP Endpoint URL

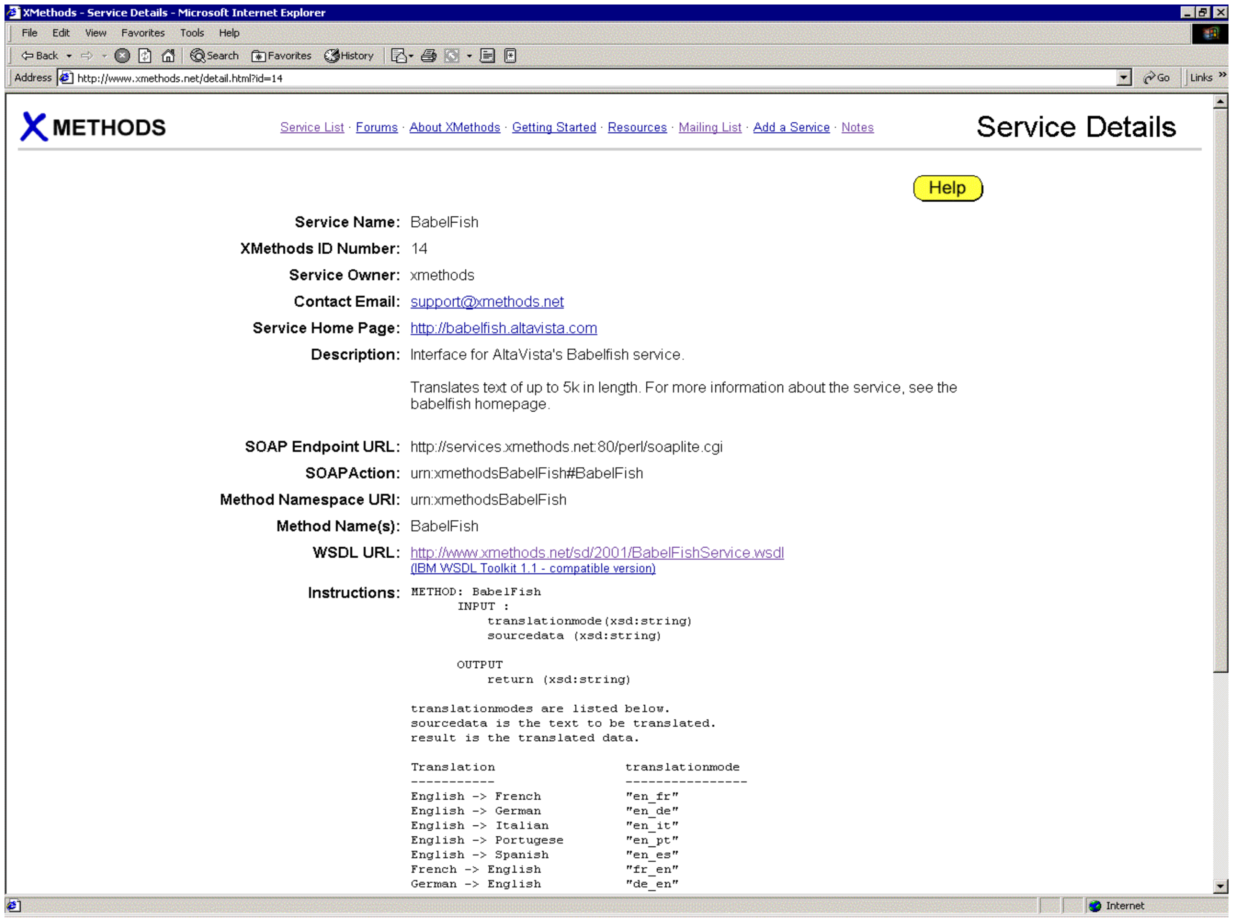
Each service entry contains information describing the service. Find the service entry detail and locate the SOAP endpoint URL. For the BabelFish service, the URL is:

http://www.xmethods.net/ve2/ViewListing.po?serviceid=14

For details, see **Figure 11 on page 43**.

Note: *Keep in mind that there are two BabelFish services, WSDL and non-WSDL. For this implementation, use the non-WSDL service.*

Figure 11 BabelFish Service Entry Detail


XMethods - Service Details - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.xmethods.net/detail.html?id=14
Go Links
METHODS
Service List · Forums · About XMethods · Getting Started · Resources · Mailing List · Add a Service · Notes
Service Details
Help
Internet

Service Name: BabelFish
XMethods ID Number: 14
Service Owner: xmethods
Contact Email: support@xmethods.net
Service Home Page: http://babelfish.altavista.com
Description: Interface for AltaVista's Babelfish service.

Translates text of up to 5k in length. For more information about the service, see the babelfish homepage.

SOAP Endpoint URL: http://services.xmethods.net:80/perl/soaplite.cgi
SOAPAction: urn:xmethodsBabelFish#BabelFish
Method Namespace URI: urn:xmethodsBabelFish
Method Name(s): BabelFish
WSDL URL: http://www.xmethods.net/sd/2001/BabelFishService.wsdl
(IBM WSDL Toolkit 1.1 - compatible version)
Instructions:

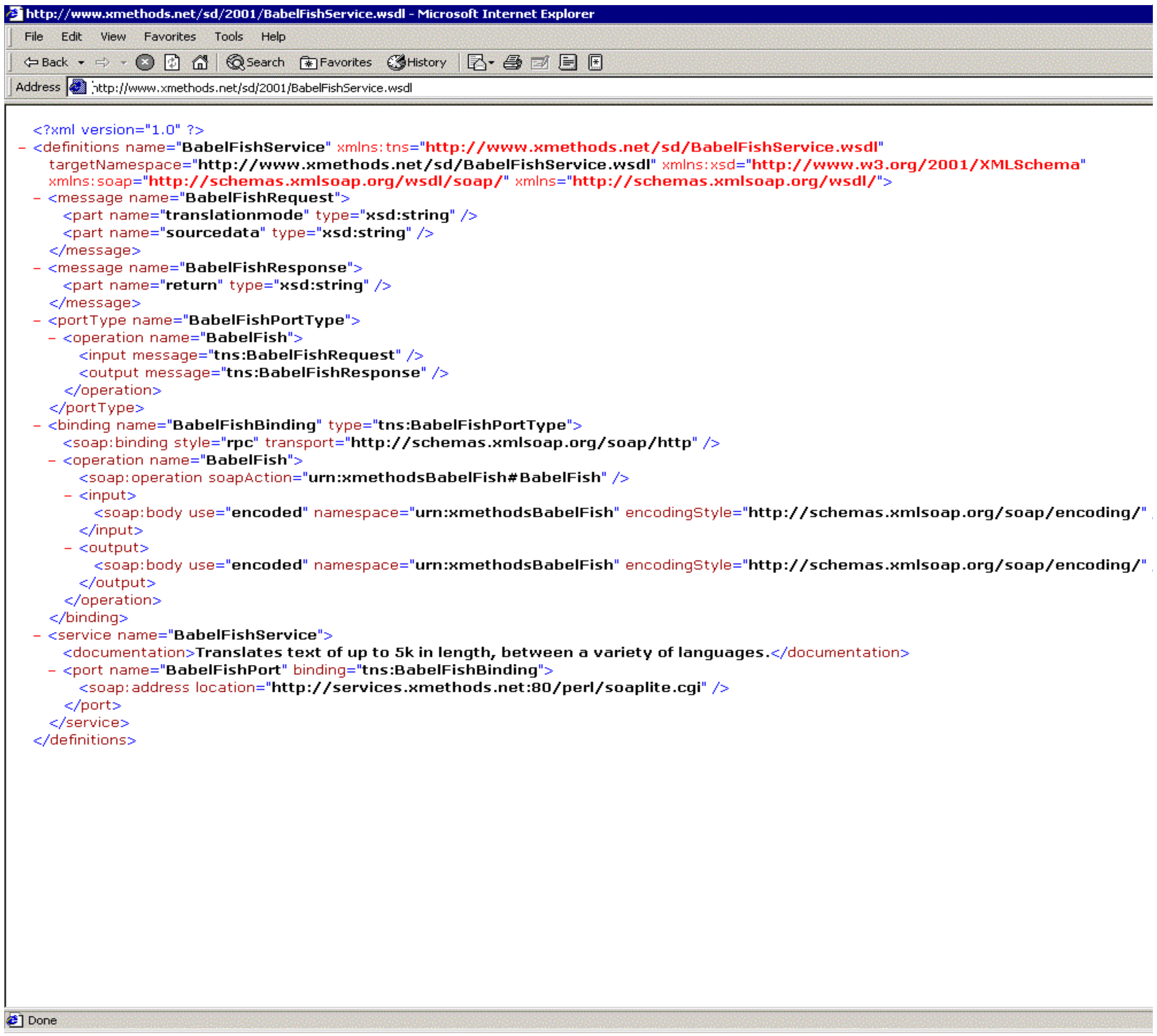
```
METHOD: BabelFish
      INPUT :
          translationmode(xsd:string)
          sourcedata (xsd:string)

      OUTPUT
          return (xsd:string)

translationmodes are listed below.
sourcedata is the text to be translated.
result is the translated data.

Translation                 translationmode
-----------                 ----------------
English -> French           "en_fr"
English -> German           "en_de"
English -> Italian          "en_it"
English -> Portugese        "en_pt"
English -> Spanish          "en_es"
French -> English           "fr_en"
German -> English           "de_en"
```

Determining the Format of the SOAP Message

The service entry typically provides either an example of the SOAP message format or WSDL URL. In this version of the SOAP e*Way and SOAP ETD wizard, you have the option of describing the SOAP message as a byte stream BLOB, or as structured data.

Using structured data allows the data to be marshaled and unmarshaled from the XML document and e*Gate Java Collaboration. To use structured data, DTDs are required to describe any SOAP request header and bodies, and SOAP response header and bodies (see **Figure 12 on page 44**).

Figure 12 BabelFish DTD


http://www.xmethods.net/sd/2001/BabelFishService.wsdl - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.xmethods.net/sd/2001/BabelFishService.wsdl
Done

```
  <?xml version="1.0" ?>
- <definitions name="BabelFishService" xmlns:tns="http://www.xmethods.net/sd/BabelFishService.wsdl"
    targetNamespace="http://www.xmethods.net/sd/BabelFishService.wsdl" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns="http://schemas.xmlsoap.org/wsdl/">
  - <message name="BabelFishRequest">
      <part name="translationmode" type="xsd:string" />
      <part name="sourcedata" type="xsd:string" />
    </message>
  - <message name="BabelFishResponse">
      <part name="return" type="xsd:string" />
    </message>
  - <portType name="BabelFishPortType">
    - <operation name="BabelFish">
        <input message="tns:BabelFishRequest" />
        <output message="tns:BabelFishResponse" />
      </operation>
    </portType>
  - <binding name="BabelFishBinding" type="tns:BabelFishPortType">
      <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
    - <operation name="BabelFish">
        <soap:operation soapAction="urn:xmethodsBabelFish#BabelFish" />
      - <input>
          <soap:body use="encoded" namespace="urn:xmethodsBabelFish" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        </input>
      - <output>
          <soap:body use="encoded" namespace="urn:xmethodsBabelFish" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        </output>
      </operation>
    </binding>
  - <service name="BabelFishService">
      <documentation>Translates text of up to 5k in length, between a variety of languages.</documentation>
    - <port name="BabelFishPort" binding="tns:BabelFishBinding">
        <soap:address location="http://services.xmethods.net:80/perl/soaplite.cgi" />
      </port>
    </service>
  </definitions>
```

The following examples illustrate DTDs that describe the SOAP message as documented in the BabelFish schema:

Example 1: BabelFishRequest.dtd

This DTD describes the SOAP request as follows:

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

<!ELEMENT ns1:BabelFish (translationmode, sourcedata)>
<!ATTLIST ns1:BabelFish SOAP-ENV:encodingStyle CDATA #REQUIRED>
<!ATTLIST ns1:BabelFish xmlns:SOAP-ENV CDATA #FIXED "http://
schemas.xmlsoap.org/soap/envelope/">
<!ATTLIST ns1:BabelFish xmlns:ns1 CDATA #FIXED
"urn:xmethodsBabelFish">

<!ELEMENT translationmode (#PCDATA)>
<!ATTLIST translationmode xsi:type CDATA #FIXED "xsd:string">
```

```
<!ATTLIST translationmode xmlns:xsi CDATA #FIXED "http://www.w3.org/
1999/XMLSchema-instance">

<!ELEMENT sourcedata (#PCDATA)>
<!ATTLIST sourcedata xsi:type CDATA #FIXED "xsd:string">
<!ATTLIST sourcedata xmlns:xsi CDATA #FIXED "http://www.w3.org/1999/
XMLSchema-instance">
```

Example 2: BabelFishResponse.dtd

This DTD describes the SOAP response as follows:

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

<!ELEMENT namesp1:BabelFishResponse ( return ) >
<!ATTLIST namesp1:BabelFishResponse SOAP-ENV:encodingStyle CDATA
#FIXED "http://schemas.xmlsoap.org/soap/encoding">
<!ATTLIST namesp1:BabelFishResponse xmlns:SOAP-ENV CDATA #FIXED
"http://schemas.xmlsoap.org/soap/envelope/">
<!ATTLIST namesp1:BabelFishResponse xmlns:namesp1 CDATA #FIXED
"urn:xmethodsBabelFish">

<!ELEMENT return ( #PCDATA ) >
<!ATTLIST return xsi:type CDATA #FIXED "xsd:string">
<!ATTLIST return xmlns:xsi CDATA #FIXED "http://www.w3.org/1999/
XMLSchema-instance">
```

Note: *The DTDs shown in the previous examples were created manually from information contained in the WSDL entry.*

Creating a Schema

Before creating a schema, first verify that you have the correct e*Gate installation and that it is operating correctly.

Verifying the e*Gate Installation

You can run this schema on a single machine. Before beginning the configuration process, you must verify that you have all the required software installed on the target machine.

Check the following e*Gate system components:

- Registry Host
- Participating Host
- Windows interfaces
  - e*Gate Schema Designer
  - Schema Manager

You can install all the software components shown in the previous list on the machine that runs this schema. See the *e*Gate Integrator Installation Guide* for instructions on how to install the e*Gate components and for e*Gate system requirements.

To create a new schema

1 Start the e*Gate Schema Designer and log in as **Administrator** (or another user with administrator privileges) to the appropriate Registry Host.

2 In the **Open Schema on Registry Host** dialog box, click **New**.

3 In the **Enter New Schema Name** box, type **BabelFish**, and then click **Open**.

The Schema Designer opens and displays the new **BabelFish** schema.

4 At the bottom of the Navigator (left) pane, click the **Components** tab.

You perform all configuration steps in this pane, on the **Components** tab.

Note: *When you create a new schema, by default, e*Gate automatically creates a Control Broker for the schema. The default name is **host-name_cb**, where **host-name** is the logical name of the current host machine. For this example, use these default Control Broker name and its default settings.*

Creating Event Types and Event Type Definitions

In this step, you create Event Types and Event Type Definitions (ETDs) that the e*Gate system uses to transport data.

Creating Event Types

An Event Type is a class of Events (packages of data) with a common data structure. The e*Gate system packages data within Events and categorizes them into Event Types. What these Events have in common defines the Event Type and comprises the ETD.

The following procedures show how to create an ETD using the an ETD wizard (see **"The SOAP ETD wizard takes an Extensible Markup Language (XML) file and converts it to an .xsc file (e*Gate ETD) that contains the following elements:" on page 29**).

To create Event Types

1 Highlight the **Event Types** folder on the **Components** tab of the e*Gate Schema Designer's Navigator pane (**Components** tab).

2 On the palette, click the **Create a New Event Type** button.

A dialog box opens allowing you to enter the name of the new Event Type.

3 Enter the name of the Event Type. For the purpose of this sample, the SOAP Event Type is named **SOAP_BabelFish_Event**.

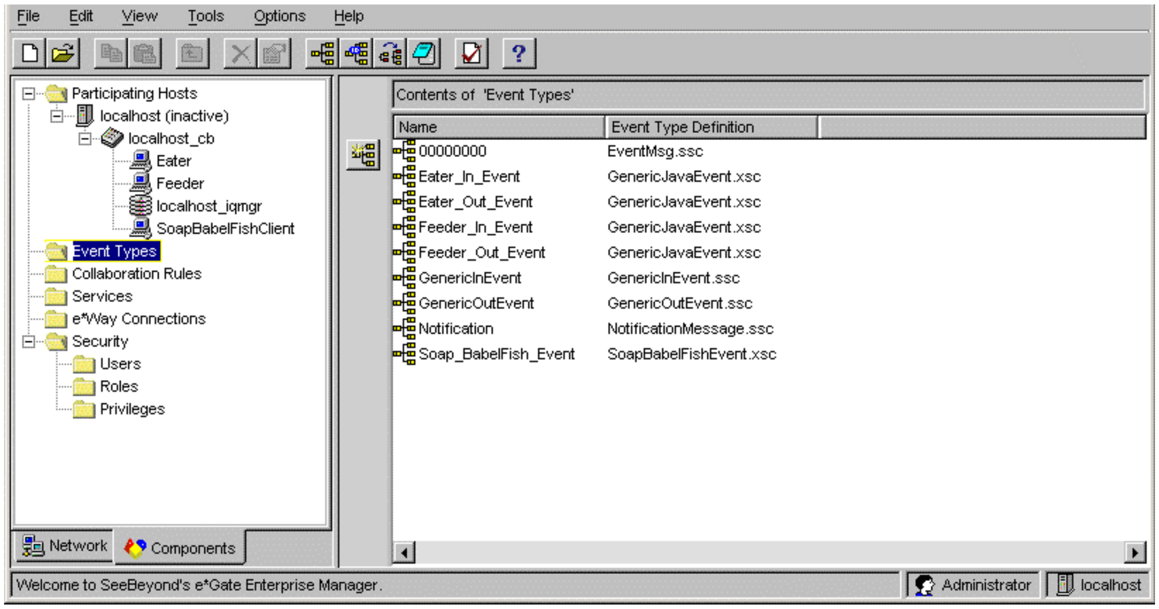
4 Click **OK.** The dialog box closes, and e*Gate saves the name of your new Event Type.

Using these steps, create the following Event Types:

- **SOAP_BabelFish_Event:** This Event Type contains the request data, the response data (if any), and the methods used to manipulate the data.
- **Feeder_In_Event:** This Event Type contains raw data from the input file.
- **Feeder_Out_Event:** This Event Type contains raw data from the input file.
- **Eater_In_Event:** This Event Type contains the translated data.
- **Eater_Out_Event:** This Event Type contains the translated data.

When you have finished, the e*Gate Schema Designer shows all of your created Event Types (see Figure 13).

Figure 13 e*Gate Schema Designer: Event Types



Contents of 'Event Types'

Name	Event Type Definition
00000000	EventMsg.ssc
Eater_In_Event	GenericJavaEvent.xsc
Eater_Out_Event	GenericJavaEvent.xsc
Feeder_In_Event	GenericJavaEvent.xsc
Feeder_Out_Event	GenericJavaEvent.xsc
GenericInEvent	GenericInEvent.ssc
GenericOutEvent	GenericOutEvent.ssc
Notification	NotificationMessage.ssc
Soap_BabelFish_Event	SoapBabelFishEvent.xsc

Welcome to SeeBeyond's e*Gate Enterprise Manager.

Defining the SOAP ETD

Next, you must define the **SOAP_BabelFish_Event** Event Type you have created, that is, create its ETD. To define this Event Type, you must create an ETD file that describes the SOAP message.

You can use any of the following methods to create this ETD:

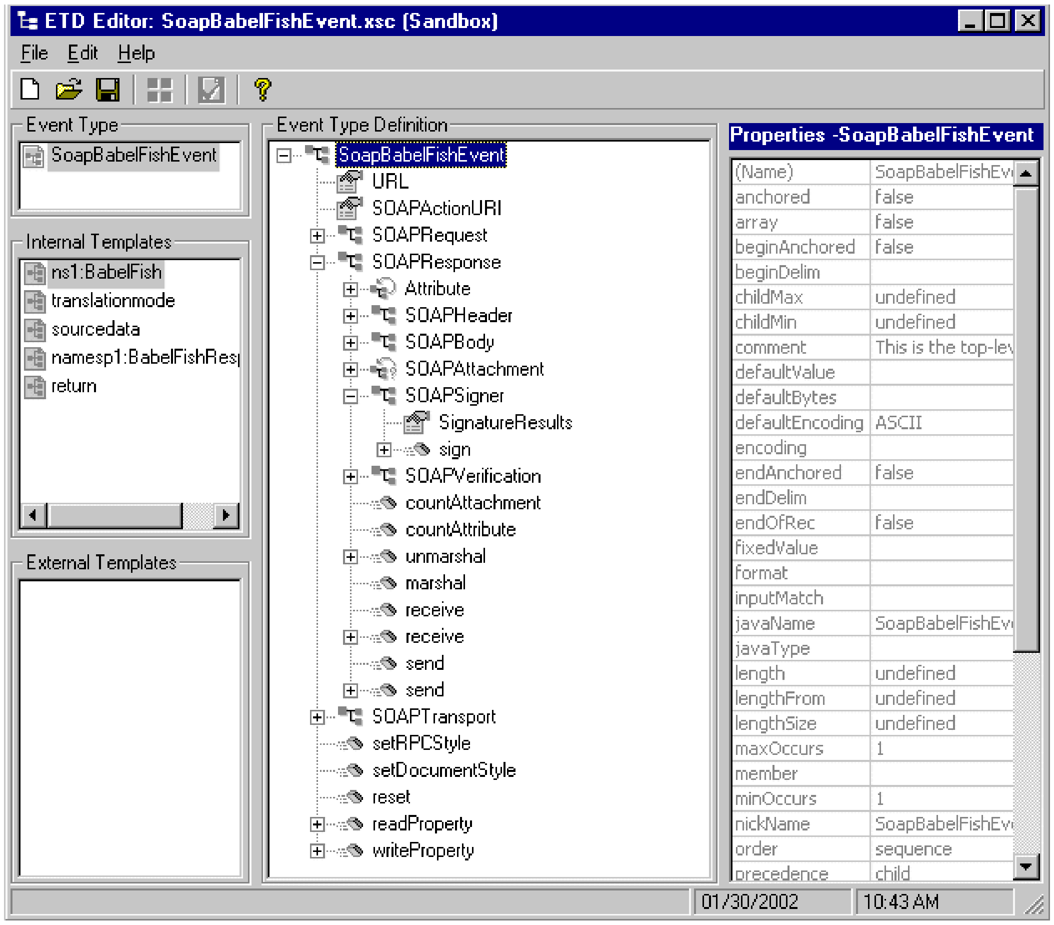
- Using the ETD **\eGate\client\etd\ewsoap\SOAPSimple.xsc**
- Modifying a copy of the ETD **\eGate\client\etd\ewsoap\SOAPSimple.xsc**
- Using the SOAP ETD wizard to create an ETD describing a byte stream BLOB
- Using the SOAP ETD wizard to create an ETD describing a structured XML document

Use the Schema Designer's ETD Editor to create and modify ETDs. The ETD Editor has convenient wizard features that help you to create ETDs. These sample procedures use an ETD created by the SOAP ETD wizard.

See **"SOAP ETD Wizard: Basic SOAP" on page 30** for a complete explanation of how to use the SOAP ETD wizard to create an ETD file from a structured DTD file (or basic XML).

When you have finished using the wizard, the ETD Editor opens the new **.xsc** file (the SOAP ETD) and shows its basic structure in the Main window (see **Figure 14 on page 48**).

Figure 14 ETD Editor: SOAP ETD



ETD Editor: SoapBabelFishEvent.xsc (Sandbox)

File Edit Help

Event Type
- SoapBabelFishEvent

Internal Templates
- ns1:BabelFish
- translationmode
- sourcedata
- namesp1:BabelFishRes
- return

External Templates

Event Type Definition

```
SoapBabelFishEvent
    URL
    SOAPActionURI
    SOAPRequest
    SOAPResponse
        Attribute
        SOAPHeader
        SOAPBody
        SOAPAttachment
        SOAPSigner
            SignatureResults
            sign
        SOAPVerification
        countAttachment
        countAttribute
        unmarshal
        marshal
        receive
        receive
        send
        send
    SOAPTransport
    setRPCStyle
    setDocumentStyle
    reset
    readProperty
    writeProperty
```

Properties -SoapBabelFishEvent

(Name)	SoapBabelFishEv
anchored	false
array	false
beginAnchored	false
beginDelim	
childMax	undefined
childMin	undefined
comment	This is the top-lev
defaultValue	
defaultBytes	
defaultEncoding	ASCII
encoding	
endAnchored	false
endDelim	
endOfRec	false
fixedValue	
format	
inputMatch	
javaName	SoapBabelFishEv
javaType	
length	undefined
lengthFrom	undefined
lengthSize	undefined
maxOccurs	1
member	
minOccurs	1
nickName	SoapBabelFishEv
order	sequence
precedence	child

01/30/2002 10:43 AM

The name of the ETD (**SOAPEventDefinition** in the example in Figure 14) shown in the ETD Editor Main window may not be the desired name. For the current example, the ETD name must be **BabelFish**. Change the name using either of the following steps:

- Click on the displayed ETD name twice in the **Event Type Definition** pane, type the new name (**BabelFish**), and press ENTER.
- Highlight the displayed ETD name under the **Abstract** tab in the **Properties** pane, type a new name **BabelFish**, and press ENTER.

File Name: Be sure to save and compile the new ETD file, then name it:

BabelFish.xsc

When you are done, close the ETD Editor and continue.

Additional Event Types

As explained previously, the BabelFish schema also uses the following Event Types:

- **Feeder_In_Event**
- **Feeder_Out_Event**
- **Eater_In_Event**
- **Eater_Out_Event**

Adding and Configuring e*Ways

For the BabelFish schema, you must create the following e*Ways:

- **Feeder** receives text from an external source, applies Java Pass Through Collaboration Rules, and publishes the information to an IQ that stores inbound data.
- **Eater** receives the outbound message from an IQ and publishes it to a file; it also uses the Java Pass Through Collaboration Rules.
- **SOAPBabelFishClient** applies extended Java Collaboration Rules to an inbound Event to perform the desired business logic. In this case, the e*Way translates the inbound Event into a SOAP message, sends the SOAP message to a SOAP service, receives a translated text response from the SOAP service, and publishes the response to an IQ.

These e*Ways use the executable file **stceway.exe**, causing them to become Multi-Mode e*Ways. For details on the Multi-Mode e*Way, see Chapter 3.

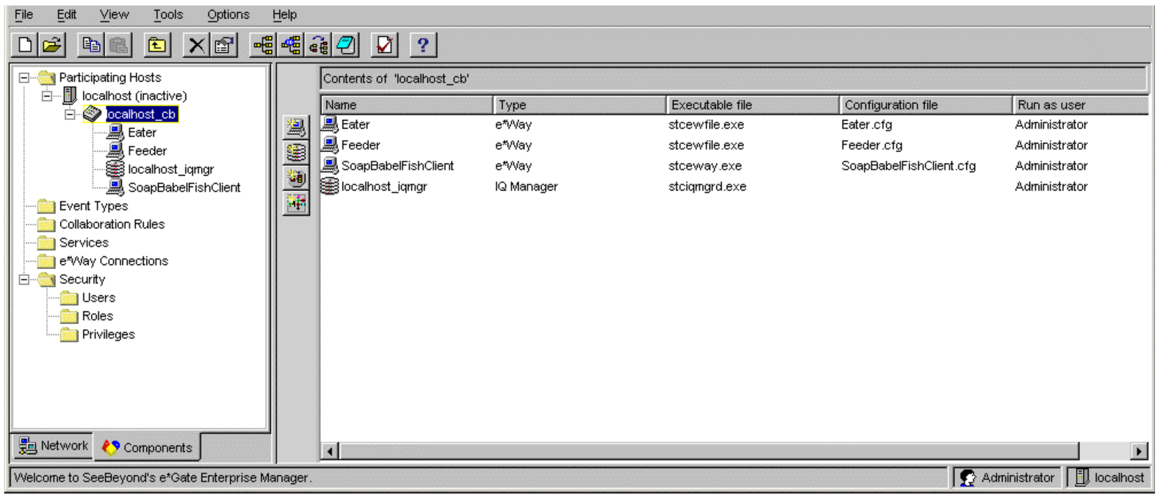
To create new e*Ways

1. In the **Navigator** pane (**Components** tab), select the desired Control Broker.
2. On the Palette, click the **Create a New e*Way** button.

   The **New e*Way Component** dialog box appears.
3. Enter the desired name (**Feeder**) for the new e*Way and click **Apply** to enter it into the system. The new name and an e*Way icon appear in both panes.
4. Name additional e*Ways as needed (**Eater** and **SOAPBabelFishClient**). Click **Apply** after you name each one.
5. When you are finished, click **OK** to close the dialog box.

The new e*Way icons appear in the Schema Designer Main window as shown in Figure 15.

Figure 15 e*Gate Schema Designer with e*Ways



- Participating Hosts
  - localhost (inactive)
    - localhost_cb
      - Eater
      - Feeder
      - localhost_iqmgr
      - SoapBabelFishClient
- Event Types
- Collaboration Rules
- Services
- e*Way Connections
- Security
  - Users
  - Roles
  - Privileges

Contents of 'localhost_cb'

Name	Type	Executable file	Configuration file	Run as user
Eater	e*Way	stcewfile.exe	Eater.cfg	Administrator
Feeder	e*Way	stcewfile.exe	Feeder.cfg	Administrator
SoapBabelFishClient	e*Way	stceway.exe	SoapBabelFishClient.cfg	Administrator
localhost_iqmgr	IQ Manager	stciqmgrd.exe		Administrator

Welcome to SeeBeyond's e*Gate Enterprise Manager.

To configure the Feeder e*Way

1. From e*Gate Schema Designer, double-click on the **Feeder** e*Way icon to display the properties of the e*Way.

   The **e*Way Properties** dialog box appears.
2. Select the executable file **stceway.exe**.
3. Click **New**.

   The e*Way Configuration Editor appears.
4. Configure the e*Way as desired for your schema setup.

*Note: For details on how to configure the Multi-Mode e*Way, see* **Chapter 3**.

5. Close the e*Way Configuration Editor and save the e*Way configuration file (**Feeder.cfg**).

To configure the Eater e*Way

1. From e*Gate Schema Designer, double-click on the **Eater** e*Way icon to display the properties of the e*way, then click **New**.

   The **e*Way Properties** dialog box appears.
2. Select the executable file **stceway.exe**.
3. Click **New**.

   The e*Way Configuration Editor appears.
4. Configure the e*Way as desired for your schema setup.
5. Close the e*Way Configuration Editor and save the e*Way configuration file (**Eater.cfg**).

To configure the SOAPBabelFishClient e*Way

1. From e*Gate Schema Designer, double-click on the **SOAPBabelFishClient** e*Way icon to display the properties of the e*Way, then click **New.**

   The **e*Way Properties** dialog box appears.

2. Select the executable file **stceway.exe**.
3. Click **New**.

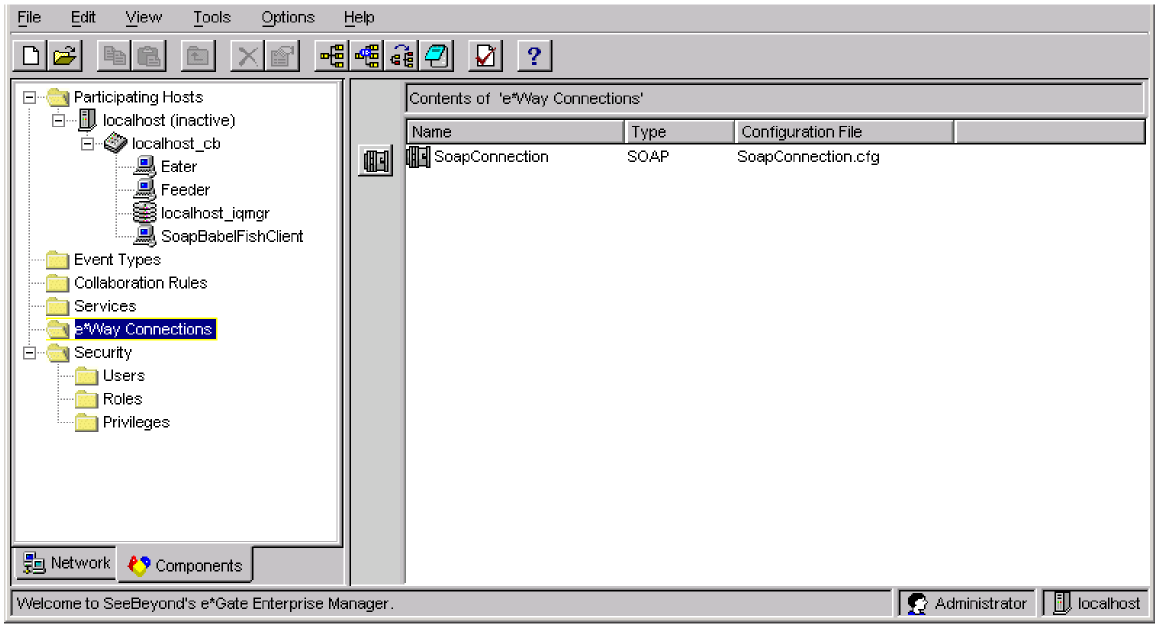
   The e*Way Configuration Editor appears.

4. Configure the e*Way as desired for your schema setup.
5. Close the e*Way Configuration Editor and save the e*Way configuration file (**SOAPBabelFishClient.cfg**).

Creating the e*Way Connection

See Chapter 5 for complete information on how to configure the SOAP e*Way Connection. Figure 16 on page 51 shows the created e*Way Connection **SoapConnection** in the e*Gate Schema Designer's Main window.

Figure 16 Schema Designer with e*Way Connection



Contents of 'e*Way Connections'

Name	Type	Configuration File
SoapConnection	SOAP	SoapConnection.cfg

Welcome to SeeBeyond's e*Gate Enterprise Manager.

Creating Collaboration Rules

In the e*Gate system, Events become subject to business logic via processing, transformation, or verification. e*Gate uses the following components to govern these operations:

- **Collaboration** is the necessary, configurable component of an e*Way that determines its operation; that is, the logical moving and transformation of Events.
- **Collaboration Rules** are the program logic that instructs a Collaboration how to execute the business logic required to support e*Gate's data transformation and routing.
- **Collaboration Service** is the program that defines the structure and operation of a Collaboration Rule's basic Event-handling processes. For example, Java Collaborations use the Java Collaboration Service.
- **Collaboration Rules Script** contains the specific operations (written in the Monk programming language) that are used to govern Event-transformation processes within a Collaboration.
- **Business Rules** are the Java source code that creates the output Events that are a result of the Java Collaboration.

Collaboration **.class** files (Java) and Collaboration Rules scripts (Monk) are necessary if you want to have any data transformed and/or verified in some way as it passes through a Collaboration.

You must create Collaboration Rules before you create the Collaborations that use them. For details on how to create the BabelFish Collaborations, see **"Creating and Configuring Collaborations" on page 58**.

For the BabelFish schema, you must create the following Collaboration Rules:

- **Feeder:** Uses Java Pass Through Collaboration Rules and is associated with the **Feeder** e*Way.
- **Eater:** Uses Java Pass Through Collaboration Rules and is associated with the **Eater** e*Way.
- **SOAPBabelFishClient:** associated with the **SOAPBabelFishClient** e*Way and is used to perform the transformation process, send the Event to the SOAP service, and receive a response from the SOAP service.

The following pseudo-code helps to explain the actions of the **SOAPBabelFishClient** e*Way Collaboration/Collaboration Rules:

```
Set translation mode to en_fr
Populate the data source - an English sentence
Send SOAP request to the SOAP server and unmarshal the response into
SOAPResponse
Get the translated French sentence
```

To create Java Pass Through Collaboration Rules components

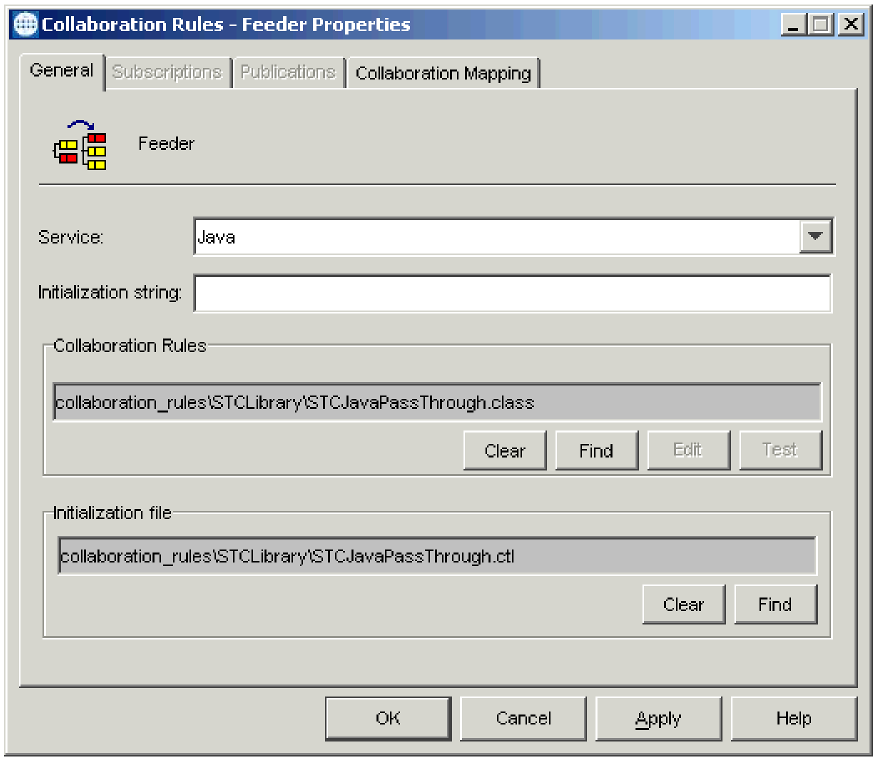
1. Select the Navigator pane's **Components** tab in the e*Gate Schema Designer.
2. In the **Navigator**, select the **Collaboration Rules** folder.

3 On the palette, click the **Create New Collaboration Rules** button.

4 Enter the name of the new Collaboration Rule Component, **Feeder**, then click **OK**.

5 Double-click the new Collaboration Rules Component icon. The **Collaboration Rules Properties** dialog box opens (see Figure 17 on page 53).

Figure 17 Collaboration Rules Properties Dialog Box: Java Pass Through



Feeder

Service: Java

Initialization string:

Collaboration Rules

collaboration_rules\STCLibrary\STCJavaPassThrough.class

Initialization file

collaboration_rules\STCLibrary\STCJavaPassThrough.ctl

6 Select **Java** for the **Service** and **STCJavaPassThrough.class** for the Collaboration Rules file.

Note: *With Java Pass Through Collaboration Rules, the information on the Subscriptions and Publications tabs is configured automatically.*

7 Click **OK** to close the **Collaboration Rules Properties** dialog box.

8 Repeat this same procedure for the **Eater** Collaboration Rule.

To create and edit the Java Collaboration Rules component

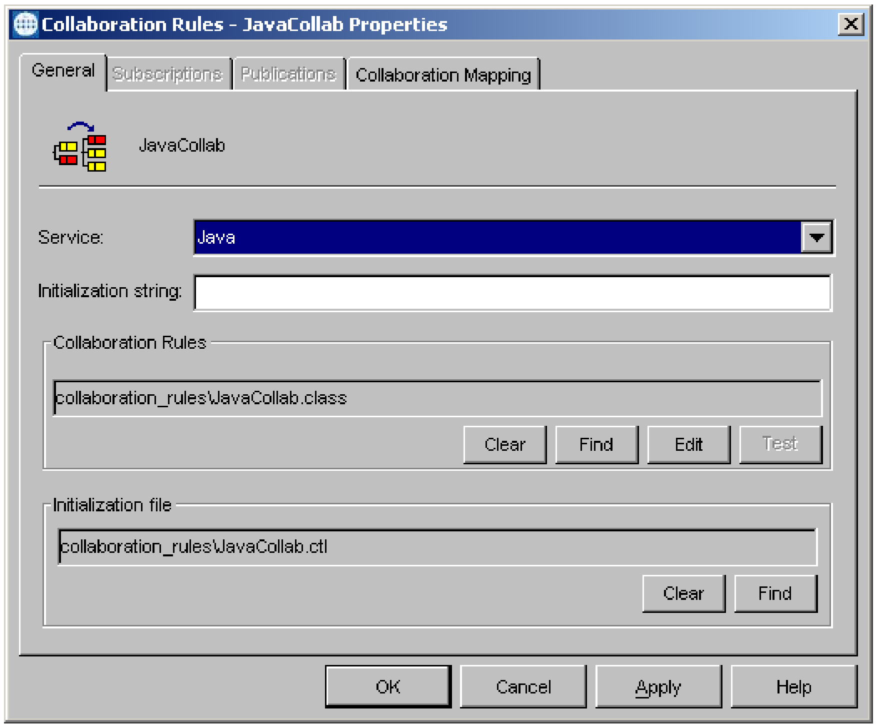
1 Use the Schema Designer to create the new Java Collaboration Rules component in the same way as you did the Java Pass Through Collaboration Rules components.

2 Double-click the new Collaboration Rules component icon to edit its properties.

The **Collaboration Rules Properties** dialog box opens (see Figure 17).

3 From the **Service** field drop-down box, select **Java**. The **Collaboration Mapping** tab is now enabled, and the **Subscriptions** and **Publications** tabs are disabled (see Figure 18 on page 54).

Figure 18 Collaboration Rules Properties Dialog Box: Java



JavaCollab

Service: Java

Initialization string:

Collaboration Rules: collaboration_rules\JavaCollab.class

Initialization file: collaboration_rules\JavaCollab.ctl

4 In the **Initialization string** field, enter any required initialization string for the Collaboration. If none is needed, you can skip this step.

5 Select the **Collaboration Mapping** tab.

6 Using the **Add Instance** button, create instances to coincide with the Event Types as follows:

- In the **Instance Name** column, enter **In** for the instance name.
- Click **Find**, navigate to **etd\BabelFish.xsc**, double-click to select. **BabelFish.xsc** is added to the **ETD** column of the instance row.
- In the **Mode** column, select **In** from the drop–down menu available.
- In the **Trigger** column, click the box to enable trigger mechanism.

7 Repeat the actions listed under step 6 using the following values:

- Instance Name: **Out**
- ETD: **BabelFish.xsc**
- Mode: **Out**

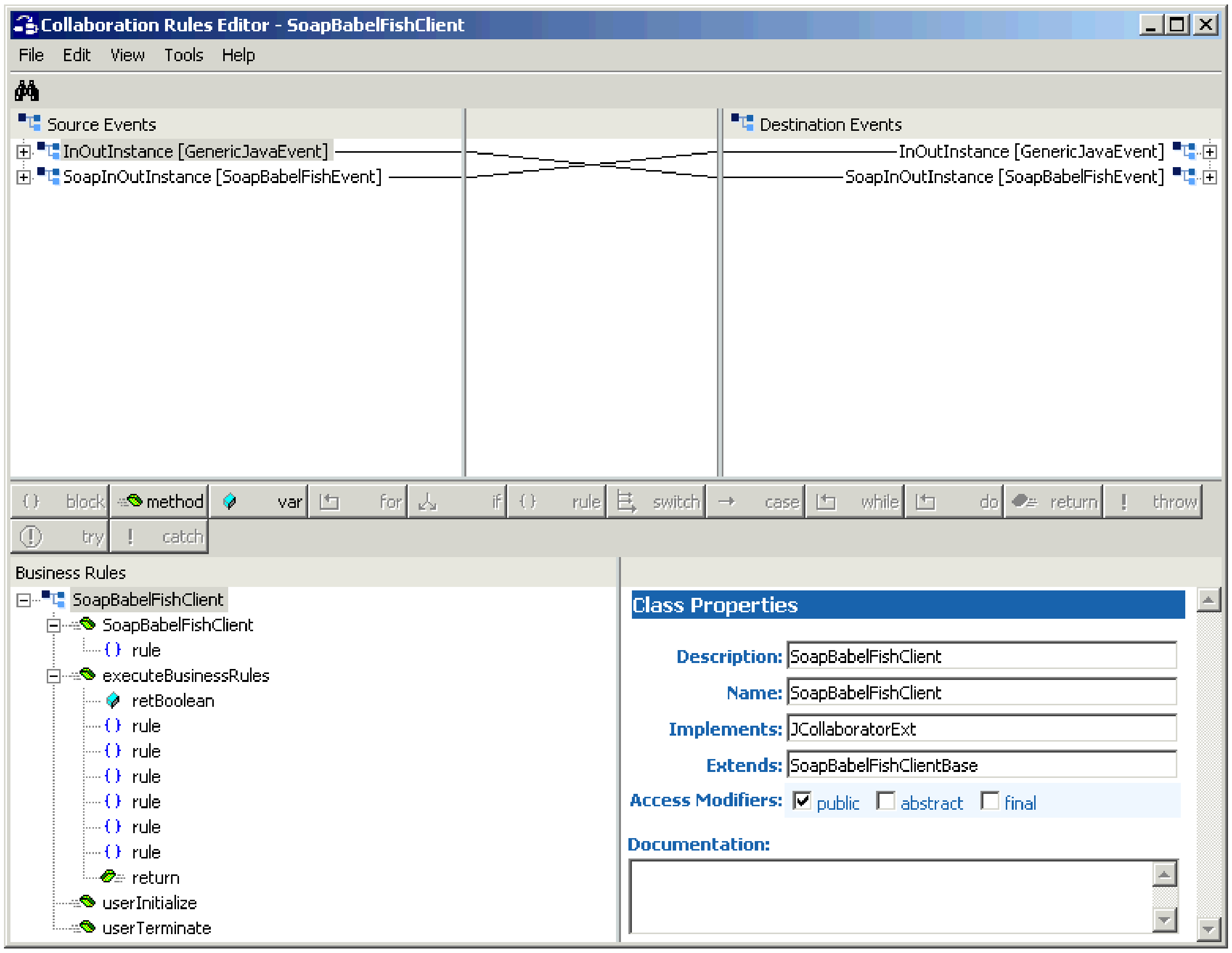
Note: *At least one of the ETD instances used by the Collaboration must be checked as the trigger. For specific information on creating and configuring Collaboration Rules, see the* ***e*Gate Integrator User's Guide****.*

8 Select the **General** tab, under the Collaboration Rule box, select **New**. The Collaboration Rules Editor opens.

9 Expand the Collaboration Rules Editor to full size, expanding the Source and Destination Events panes as well, then create the Collaboration Rule.

Note: *The example in Figure 19 shows the opened **SoapBabelFishClient.xpr** file from the sample.*

Figure 19 Collaboration Rules: Collaboration Rules Editor


Collaboration Rules Editor - SoapBabelFishClient
File Edit View Tools Help
Source Events
InOutInstance [GenericJavaEvent]
SoapInOutInstance [SoapBabelFishEvent]
Destination Events
InOutInstance [GenericJavaEvent]
SoapInOutInstance [SoapBabelFishEvent]
block method var for if rule switch case while do return throw
try catch
Business Rules
SoapBabelFishClient
SoapBabelFishClient
rule
executeBusinessRules
retBoolean
rule
rule
rule
rule
rule
rule
return
userInitialize
userTerminate
Class Properties
Documentation:

Description: SoapBabelFishClient
Name: SoapBabelFishClient
Implements: JCollaboratorExt
Extends: SoapBabelFishClientBase
Access Modifiers: [✓ public] abstract final

Using the Collaboration Rules Editor

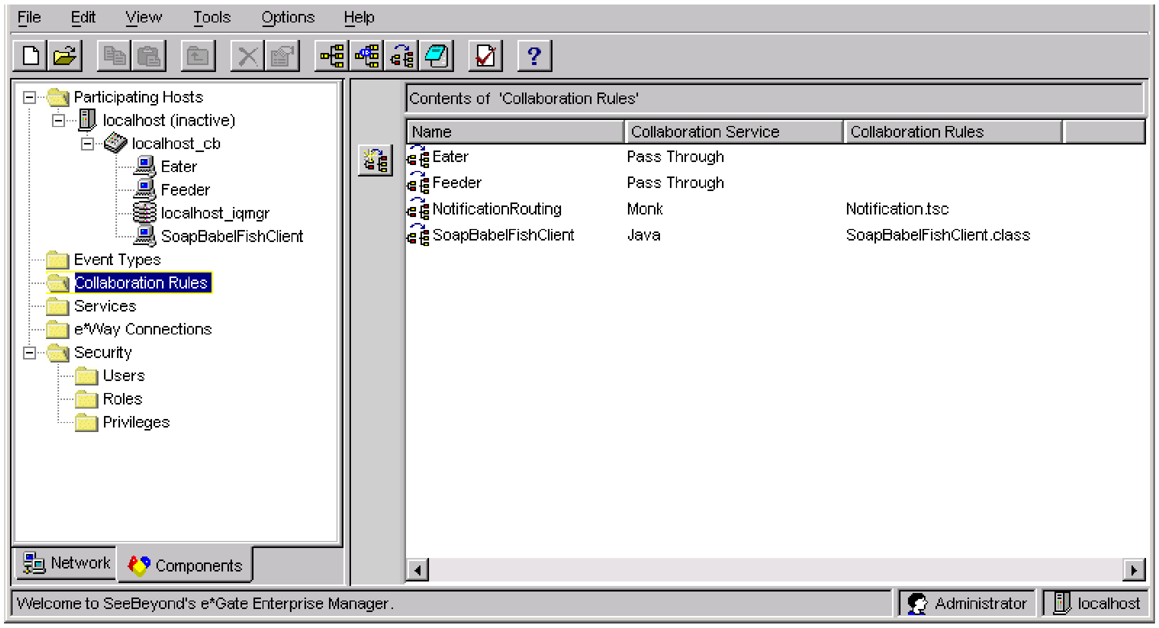
The next part of this step is to define the business logic using the Collaboration Rules Editor. Use the Collaboration Rules Editor to create and modify your Business Rules.

A Java Collaboration Rule is created by designating one or more source ETDs and one or more destination ETDs then setting up rules governing the relationship between fields in the two ETDs. Use the Collaboration Rules Editor to tell e*Gate how you want data taken from the source ETD, then manipulated and placed in the destination ETD.

Note: *To create the Collaboration Rule for SoapBabelFishClient, open the **.xpr** file from the sample and configure your new Collaboration Rule in the same way. For complete information on creating Collaboration Rules using the Java Collaboration Rules Editor see the **e*Gate Integrator User's Guide**.*

Figure 20 shows the created Collaboration Rules in the e*Gate Schema Designer.

Figure 20 e*Gate Schema Designer with Collaboration Rules



Contents of 'Collaboration Rules'

Name	Collaboration Service	Collaboration Rules
Eater	Pass Through	
Feeder	Pass Through	
NotificationRouting	Monk	Notification.tsc
SoapBabelFishClient	Java	SoapBabelFishClient.class

Welcome to SeeBeyond's e*Gate Enterprise Manager.

Creating Intelligent Queues

The next step in setting up the BabelFish schema is to create the IQs. IQs manage the exchange of information between components within the e*Gate system, providing non-volatile storage for data as it passes from one component to another.

IQs use IQ Services to transport data. IQ Services provide the mechanism for moving Events between IQs and handling the low-level implementation of data exchange (such as system calls to initialize or reorganize a database).

For the BabelFish schema, you must create the following IQs:

- **In_Q** receives data from the **Feeder** e*Way and sends it to the **SOAPBabelFishClient** e*Way.
- **Out_Q** receives data from the **SOAPBabelFishClient** e*Way and sends it to the **Eater** e*Way.

To create and modify the IQs

1. Select the **Navigation** pane's **Components** tab.
2. Open the host where you want to create the IQ.
3. Open the desired **Control Broker**.
4. Select the desired **IQ Manager**.

5 On the palette, click the **Create a New IQ** button.

6 Enter the name of the new IQ (in this case, **In_Q**), then click **OK.**

7 Double-click the new IQ's icon in the **Navigation** pane to edit its properties.

The **IQ Properties** dialog box appears.

8 On the **General** tab, specify the **Service** and the **Event Type Get Interval**. Configure these settings as follows:

- The **STC_Standard** IQ Service provides sufficient functionality for most applications. If specialized services are required, you can create custom IQ Service **.dll** files.
- The default **Event Type Get Interval** of 100 ms is satisfactory for the purposes of this sample implementation.

9 On the **Advanced** tab, be sure that **Simple publish/subscribe** is checked under the **IQ behavior** section.

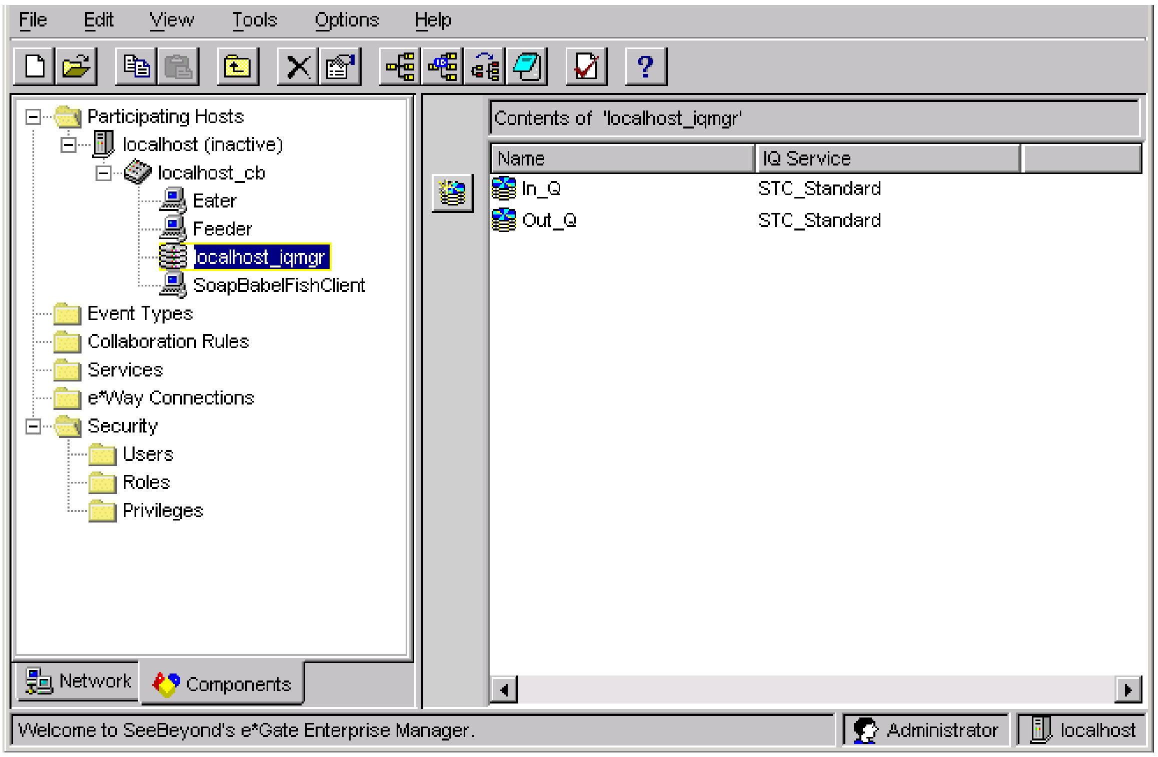
10 Click **OK** to close the **IQ Properties** dialog box.

11 For this schema, repeat this procedure to create an additional IQ (**Out_Q**).

The IQs you have created appear in the Schema Designer Main window (see Figure 21 on page 57)

Note: *For more details on this procedure, see **Creating an End-to-End Scenario with e*Gate Integrator** and/or the **e*Gate Integrator User's Guide**.*

Figure 21 e*Gate Schema Designer with IQs



Contents of 'localhost_iqmgr'

Name	IQ Service
In_Q	STC_Standard
Out_Q	STC_Standard

Welcome to SeeBeyond's e*Gate Enterprise Manager.

Creating and Configuring Collaborations

You must create the following Collaborations:

- **Feeder:** associated with the **Feeder** e*Way and is used for receiving input Events into e*Gate; uses the Feeder Collaboration Rule you created previously.
- **SOAPBabelFishClient:** associated with the **SoapBabelFishClient** e*Way and is used to perform the transformation process, send the Event to the SOAP service, and receive a response from the SOAP service; uses the SOAPBabelFishClient Collaboration Rule you created previously.
- **Eater:** associated with the **Eater** e*Way and is used for sending Events out of e*Gate; uses the Eater Collaboration Rule you created previously.

To create the Collaborations

1. In the **Navigator** pane (**Components** tab), select the desired Control Broker.
2. Select the desired e*Way component (**Feeder**).
3. On the Palette, click the **Create a New Collaboration** button.

   The **New Collaboration Component** dialog box appears.
4. Enter the desired name (**Feeder**) for the new Collaboration and click **OK** to enter it into the system. The new name and a **Collaboration** icon appear in the Editor (right) pane.
5. Repeat these procedures to create the **Eater** and **SOAPBabelFishClient** Collaborations. Click **OK** after you name each one.

To configure the Collaborations

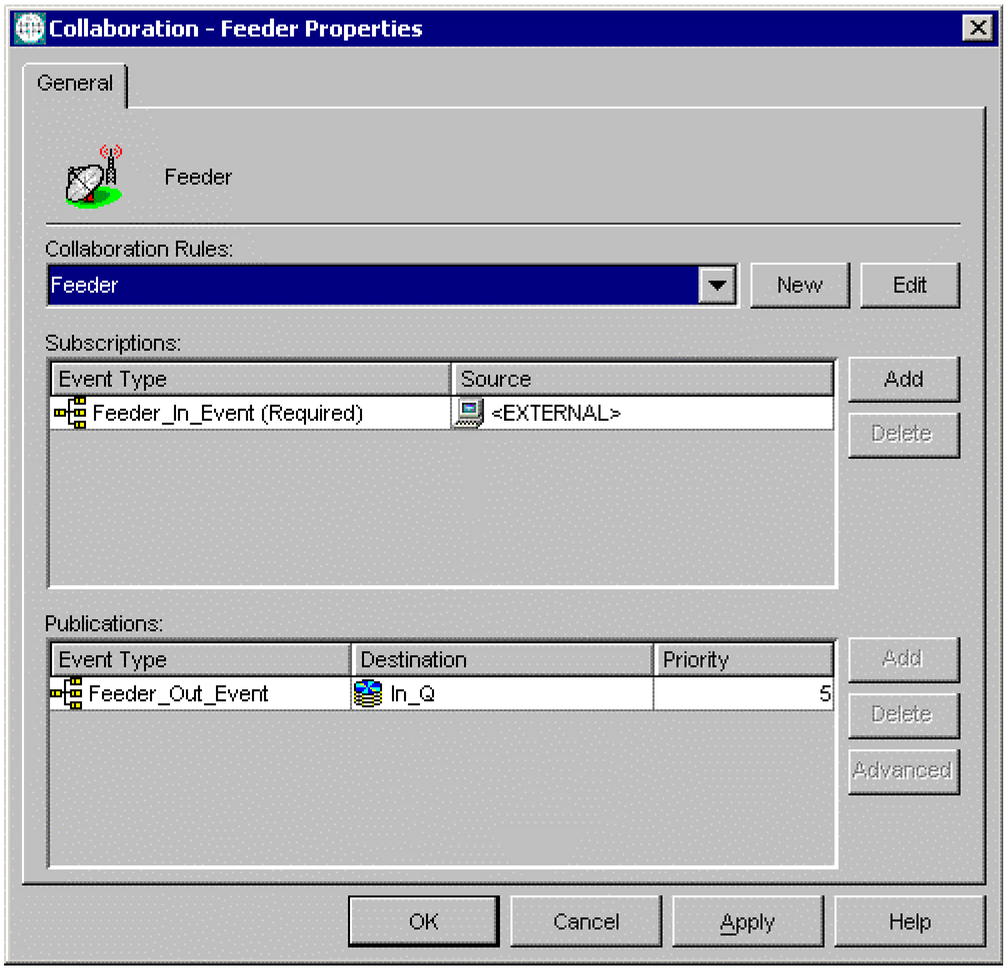
1. Double-click on the icon for the desired Collaboration (for this example, choose **Feeder** first).

   The **Collaboration Properties** dialog box appears.
2. Configure the properties for the Collaboration.

Note: *The properties and settings for all Collaborations are shown in* **Figure 22 on page 59** *through* **Figure 24 on page 61**.

3. When you are finished with **Feeder**, configure **SOAPBabelFishClient** and **Eater**. Be sure to choose the appropriate Collaboration Rule for each Collaboration.
4. When you are finished, click **OK** to save each configuration and close each dialog box.

Figure 22 BabelFish Feeder Collaboration Properties



Feeder

Collaboration Rules: Feeder

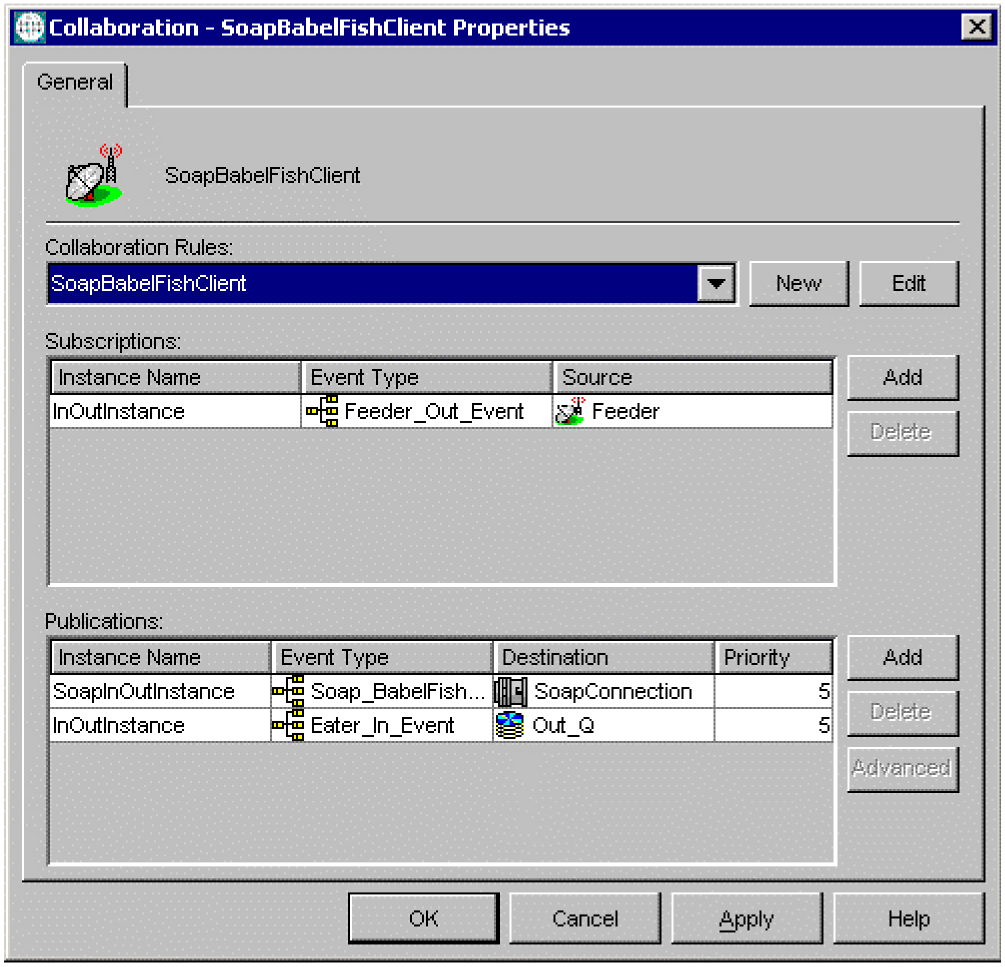
Subscriptions:

Event Type	Source
Feeder_In_Event (Required)	<EXTERNAL>

Publications:

Event Type	Destination	Priority
Feeder_Out_Event	In_Q	5

Figure 23 BabelFish SOAPBabelFishClient Collaboration Properties



SoapBabelFishClient

Collaboration Rules: SoapBabelFishClient

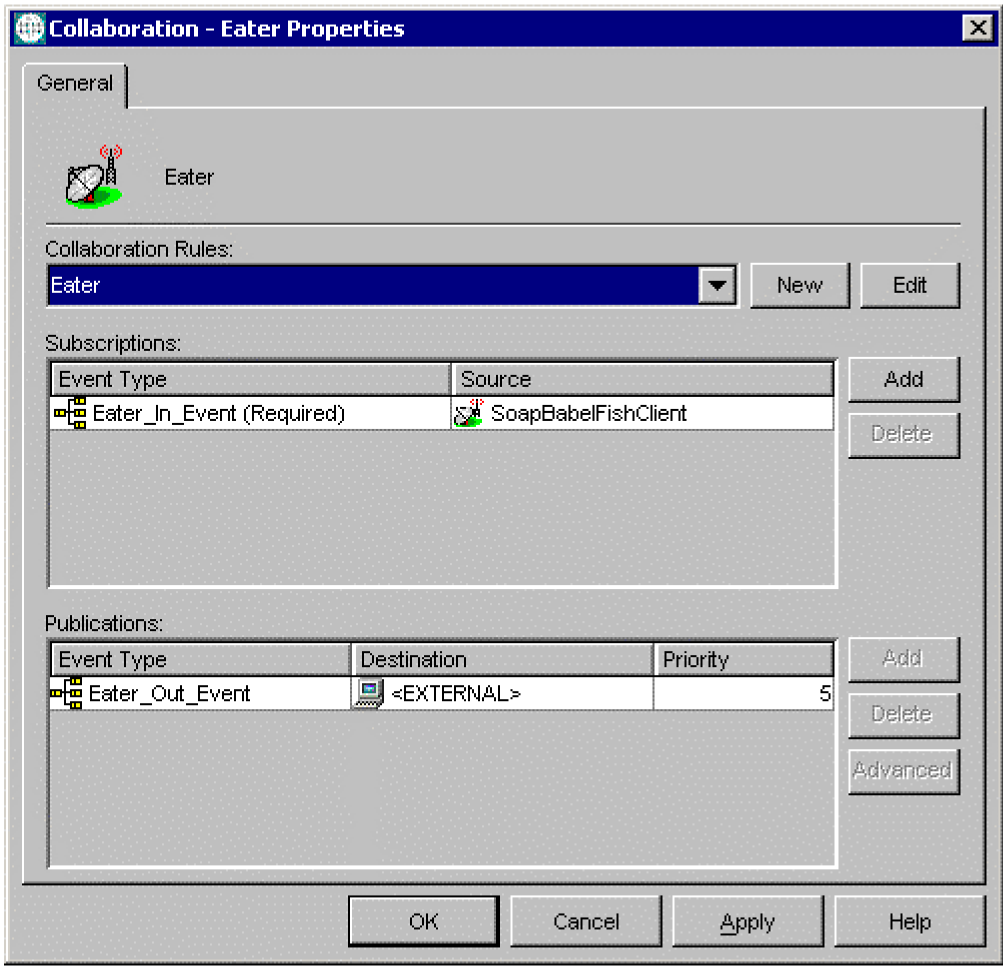
Subscriptions:

Instance Name	Event Type	Source
InOutInstance	Feeder_Out_Event	Feeder

Publications:

Instance Name	Event Type	Destination	Priority
SoapInOutInstance	Soap_BabelFish...	SoapConnection	5
InOutInstance	Eater_In_Event	Out_Q	5

Figure 24 BabelFish Eater Collaboration Properties



Eater

Collaboration Rules: Eater

Subscriptions:

Event Type	Source
Eater_In_Event (Required)	SoapBabelFishClient

Publications:

Event Type	Destination	Priority
Eater_Out_Event	<EXTERNAL>	5

Checking and Testing the Schema

Table 3 lists all the components for the schema. Check all the settings to be sure you have configured the components correctly.

Table 3 BabelFish Schema Components

Component	Logical Name	Settings
Schema	BabelFish	
Control Broker	localhost_cb	
IQ Manager	localhost_iqmgr	Start Up = Auto

Table 3 BabelFish Schema Components (Continued)

Component	Logical Name	Settings
Event Type	Feeder_In_Event	
	Feeder_Out_Event	
	Eater_In_Event	
	Eater_Out_Event	
	SOAP_BabelFish_Event	
Java ETD	BabelFish.xsc	▪ Package Name = pkgSOAPSample
Collaboration Rules	Feeder	▪ Service = Java ▪ Collaboration Rules File = STCJavaPassThrough.class
	Eater	▪ Service = Java ▪ Collaboration Rules File = STCJavaPassThrough.class
	SOAPBabelFishClient	▪ Service = Java ▪ Subscription = InOutInstance; Feeder_Out_Event (In; Trigger) ▪ Publication = SoapInOutInstance; SOAP_BabelFish_Event (Out) InOutInstance; Eater_In_Event (Out)
e*Way Connection	SoapConnection	▪ -1 for Event Type “get” interval
e*Ways	Feeder	▪ Executable = stceway.exe ▪ Configuration file = Feeder.cfg ▪ Start Up = Auto ▪ Collaboration = Feeder
	Eater	▪ Executable = stceway.exe ▪ Configuration file = Eater.cfg ▪ Start Up = Auto ▪ Collaboration = Eater
	SOAPBabelFishClient	▪ Executable = stceway.exe ▪ Configuration file = SOAPBabelFishClient.cfg ▪ Start Up = Auto ▪ Collaboration = SOAPBabelFishClient
IQ	In_Q	▪ Service = STC_Standard
	Out_Q	▪ Service = STC_Standard

Table 3 BabelFish Schema Components (Continued)

Component	Logical Name	Settings
Collaboration	Feeder	▪ Collaboration Rule = Feeder ▪ Subscription = Feeder_In_Event from <EXTERNAL> ▪ Publication = Feeder_Out_Event to In_Q
	Eater	▪ Collaboration Rule = Eater ▪ Subscription = Eater_In_Event from SOAPBabelFishClient ▪ Publication = Eater_Out_Event to <EXTERNAL>
	SOAPBabelFishClient	▪ Collaboration Rule = SOAPBabelFishClient ▪ Subscription = InOutInstance and Feeder_Out_Event from Feeder ▪ Publication = ♦ SoapInOutInstance and SOAP_BabelFish_Event to SoapConnection ♦ InOutInstance and Eater_In_Event to Out_Q

To run the BabelFish schema

1. From the command line prompt, enter:

   ```
   stccb -rh hostname -rs schemaname -un username
   -up user password -ln hostname_cb
   ```

   Substitute ***hostname***, ***username***, ***schemaname***, and ***user password*** as appropriate.

2. Change the input file name extension to **.fin**.

The schema components start automatically. When there are no more run-time messages, check the output file. If the schema is operating correctly, you can see that this file contains the input text (good morning) translated into French (*bon jour*).

4.4 SOAP Receiver Implementation

This section explains how to implement a sample schema for the SOAP e*Way, for the basic SOAP receiver. This schema, named SecureSOAP, demonstrates the use of the SOAP e*Way in implementing a Web client, as well as a SOAP service, and communicating with a SOAP server.

Note: *Implementing the SOAP receiver/server examples requires use of the CGI Web Server e*Way Intelligent Adapter and its sample schemas. See the* ***CGI Web Server e*Way Intelligent Adapter User's Guide (Java Version)*** *for details on implementing sample schemas for that e*Way.*

4.4.1 SOAP Receiver Schema: Overview

The SecureSOAP sample schema demonstrates the use of the SOAP e*Way in implementing a Web server, a SOAP client, and a SOAP server.

Note: *Implementing and testing this schema requires the use of an HTTP server, for example, Apache/Tomcat.*

Schema and File

The name of this schema is SecureSOAP. It is contained in the file **SecureSOAP.zip** in the following directory on the installation CD-ROM:

samples\ewsoap

Schema Operation

The sample schema contains elements that do the following operations:

- A file e*Way that communicates with the SOAP client e*Way and an external system via an IQ
- A SOAP client e*Way that posts data to and receives it from an HyperText Markup Language (HTML) page on an HTTP Web server
- A SOAP e*Way that implements the SOAP service
- A JMS IQ Manager that exchanges data with a SOAP server and the Web server

The SOAP server implements an “add two numbers” service in an e*Gate Java Collaboration Rule within its Collaboration. All elements outlined in the previous paragraph, except the HTML page and external system, are within the e*Gate SecureSOAP schema.

Schema Input Data

The text of the input data file is:

```
<number1>5</number1>
<number2>11</number2>
<sum>0</sum>
```

Schema Output Data

The **Feeder_eater** e*Way passes the input data to the **SOAPClient** e*Way which, in turn, sends the SOAP request to the **SOAPServiceImpl** e*Way (SOAP server) via the HTML page. The **JMS_CONN** e*Way Connection receives the SOAP response and passes it back to **Feeder_eater** via the Web server and **SOAPClient**. **Feeder_eater** then produces the output file.

The SOAP server adds the two numbers and returns the sum as follows:

```
<number1>5</number1>
<number2>11</number2>
<sum>16</sum>
```

Schema Components

See **Table 5 on page 68** for the components' configuration settings. This sample SecureSOAP schema implementation consists of the following components:

e*Ways

- **Feeder_eater** file e*Way reads text from an external source, applies a Pass Through Collaboration Rule, and publishes the information to an IQ that stores inbound data. It also receives the outbound message from the same IQ and publishes it externally to a file.
- **SOAPClient** Multi-Mode e*Way applies extended Java Collaboration Rules to an inbound Event to translate the input data into SOAP and back again. In this case, the e*Way sends a SOAP message to a Web server, receives a processed text response from the SOAP service (via the Web server), and publishes the response to an IQ.
- **SOAPServiceImpl** Multi-Mode e*Way implements the SOAP service, also applying Java Collaboration Rules. It receives a SOAP message from the JMS IQ Manager and returns the processed Event to the SOAP client via the **JMS_CONN** e*Way Connection and Web server.

Event Types

- **GenericInEvent** contains raw data from the input file.
- **GenericOutEvent** contains the processed data output file.
- **TopicRequest** contains the known topic used by the HTML page to publish the SOAP message to the **JMS_CONN** e*Way Connection.

Collaboration Rules

- **feed** is associated with the **Feeder_eater** e*Way and is used for polling the input Event.
- **eat** is associated with the **Feeder_eater** e*Way and is used for sending the processed Event to the output file.
- **SOAPClient** is associated with the **SOAPClient** e*Way and is used to perform a transformation process (translating the input data into SOAP and back again), send the SOAP request to the Web server, receive a SOAP response from the Web server, and send that response to the **eat** Collaboration.
- **SOAPServiceImpl** is associated with the **SOAPServiceImpl** e*Way and is also used to do a transformation process, implementing the SOAP service. This Collaboration implements the SOAP service. **SOAPServiceImpl** receives the Event from the **JMS_CONN** e*Way Connection, does the calculation, and sends the Event back to **JMS_CONN**.

IQ

- **The_IQ** receives data from the **Feeder_eater** e*Way and sends it to the **SOAPClient** e*Way. It also sends and receives in the reverse direction.

Schema Configuration Notes

You must configure the **SOAPClient** e*Way and the JMS IQ Manager, then modify the **.html** file so all of them can find each other. Ensure the following operations:

- The **SOAPClient** e*Way must publish to the URL for the **.html** file.
- The **.html** file must refer to the correct host name and port number of the JMS IQ Manager.

Location of Schema Files

The completed SecureSOAP schema is included on the installation CD-ROM at the following location:

\samples\ewSOAP\SecureSOAP.zip

To do this implementation, you first need to unzip this **SecureSOAP.zip** file. The files listed in Table 4 are the schema implementation files contained within **SecureSOAP.zip**.

Table 4 SecureSOAP Schema Files

File Name	Description
SecureSOAP.zip	Export schema file (do not confuse with the main .zip file).
dtds\add2numsRequestBodyBlob.dtd	DTD that describes the BabelFish SOAP request; used to create the specific SOAP ETD for the sample
dtds\add2numsResponseBodyBlob.dtd	DTD that describes the BabelFish SOAP response; used to create the specific SOAP ETD for the sample
keystore.bin	SSL KeyStore file
keystore.dsa	SSL KeyStore file
data\input\data.fin	Input file

To use this sample schema, the SOAP e*Way must be installed, the sample schema must be installed, and all of the necessary files and scripts must be located in the default location.

Schema Implementation

To implement this sample schema, you can do one of the following operations:

- To import the sample schema zip file, which automatically creates the sample schema components, see **"Sample Receiver Schema: Automatic Implementation" on page 67**.
- To manually create each of the components required to use the sample schema, see the instructions provided in **"Sample Receiver Schema: Manual Configuration" on page 68**.

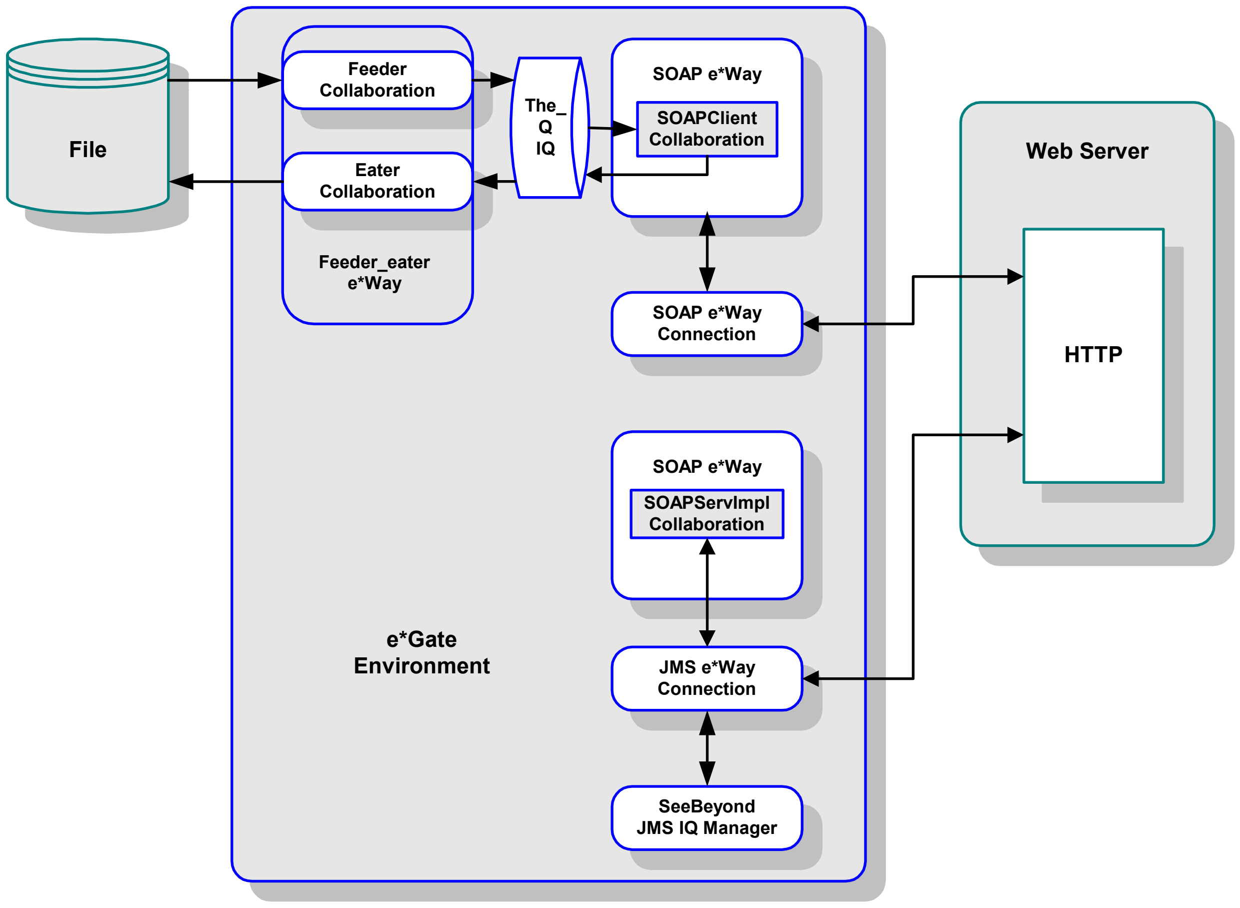
4.4.2 Sample Receiver Schema: Automatic Implementation

Install, configure, and run the SecureSOAP sample schema in the same way as you did the BabelFish sample schema explained under **"Sample Sender Schema: Automatic Implementation" on page 39**.

The schema components start automatically. When there are no more run-time messages, check the output file. If the schema is operating correctly, this file contains the sum of the two numbers, 16.

Figure 25 shows an overview diagram of the SecureSOAP schema and how it operates.

Figure 25 SecureSOAP Schema Overview



The Participating Host component that communicates with the server is a Multi-Mode e*Way. This e*Way uses a proprietary IP-based protocol to multi-thread Event exchanges between the SOAP e*Way and external systems or other e*Gate components.

4.4.3 Sample Receiver Schema: Manual Configuration

This section explains how to configure the SecureSOAP receiver schema manually in e*Gate.

Basic Implementation Steps

After you have located the SOAP service description, you are ready to implement the sample, following the basic steps outlined under **"Basic Implementation Steps" on page 42**.

See **"Sample Sender Schema: Manual Configuration" on page 41** for details on each of these steps.

Note: *For complete explanations of how to set up an e*Gate schema, see **Creating an End-to-End Scenario with e*Gate Integrator** and the **e*Gate Integrator User's Guide**.*

Creating the ETD

For complete instructions for how to create the SOAP receiver ETD, see **"Using the SOAP ETD Wizard" on page 29**.

Schema Component Configuration

Table 5 lists all the components for the SecureSOAP schema. Check all the settings to be sure you have configured all the components correctly.

Table 5 SecureSOAP Schema Components

Component	Logical Name	Settings
Schema	SecureSOAP	
Control Broker	localhost_cb	
IQ Manager	localhost_iqmgr	Start Up = Auto
JMS IQ Manager	SBYN_JMS_QMGR	Start Up = Auto
Event Type	GenericInEvent	
	GenericOutEvent	
	TopicRequest	
Java ETD	add2numbers.xsc	▪ Package Name = pkgSOAPSample
	TopicRequest.xsc	▪ Package Name = pkgSOAPSample
	SoapEvent.xsc	▪ Package Name = pkgSOAPSample

Table 5 SecureSOAP Schema Components (Continued)

Component	Logical Name	Settings
Collaboration Rules	feed	▪ Service = Pass Through ▪ Subscription = GenericInEvent ▪ Publication = GenericInEvent
	eat	▪ Service = Pass Through ▪ Subscription = GenericOutEvent ▪ Publication = GenericOutEvent
	SOAPClient	▪ Service = Java ▪ Subscription = in; GenericInEvent (In/Out; Trigger) ▪ Publication = soap; GenericInEvent; SOAP (In; Trigger) in; GenericOutEvent; The_Q (In/Out; Manual Publish)
	SOAPServiceImpl	▪ Service = Java ▪ Subscription = data; TopicRequest; JMS_CONN (In/Out; Trigger) ▪ Publication = soap; GenericInEvent; SOAP (In/Out; Manual Publish)
Java Collaboration Rule Class	SoapClient.class	▪ Source = in ▪ Destination = soap
	SOAPServiceImpl.class	▪ Source = data ▪ Destination = soap
e*Way Connection	SOAP	▪ 0 for Event Type "get" interval
	JMS_CONN	▪ 100 for Event Type "get" interval
Inbound/Outbound e*Way	Feeder_eater	▪ Executable = stcewfile.exe ▪ Configuration file = feeder_eater.cfg ▪ Start Up = Auto ▪ Collaborations = feed and eat
Multi-Mode e*Way	SOAPClient	▪ Executable = stceway.exe ▪ Configuration file = SOAPClient.cfg ▪ Start Up = Auto ▪ Collaboration = SOAPClient
	SOAPServiceImpl	▪ Executable = stceway.exe ▪ Configuration file = SOAPServiceImpl.cfg ▪ Start Up = Auto ▪ Collaboration = SOAPServiceImpl
IQ	The_Q	▪ Service = STC_Standard (see **"IQ" on page 65**)

Table 5 SecureSOAP Schema Components (Continued)

Component	Logical Name	Settings
Collaboration	feed	▪ Collaboration Rule = feed ▪ Subscription = GenericInEvent from <EXTERNAL> ▪ Publication = GenericInEvent to The_Q
	eat	▪ Collaboration Rule = eat ▪ Subscription = GenericOutEvent from SOAPClient ▪ Publication = GenericOutEvent to <EXTERNAL>
	SOAPClient	▪ Collaboration Rule = SOAPClient ▪ Subscription = in and GenericInEvent from feed ▪ Publication = ♦ soap and GenericInEvent to SOAP ♦ in and GenericOutEvent to The_Q
	SOAPServiceImpl	▪ Collaboration Rule = SOAPServiceImpl ▪ Subscription = data and TopicRequest from JMS_CONN ▪ Publication = soap and GenericInEvent to SOAP

Additional Information

Running the SecureSOAP Schema

For details on how to run the SecureSOAP schema, see **"To run the BabelFish schema" on page 63**. The schema components start automatically. When there are no more run-time messages, check the output file. If the schema is operating correctly, you can see that this file contains the sum of the two numbers, 16, in the last line.

4.5 SOAP WSDL Implementation

The SOAP implementation examples with Web Services Description Language (WSDL) are similar to the previous examples. The main difference is that the WSDL examples the SOAP WSDL-enabled ETDs instead of the basic SOAP ETDs.

This section explains the WSDL-enabled SOAP ETDs and gives examples.

Note: *For more information on implementing SOAP with WSDL, see* **"Using the Web Services Description Language" on page 117**.

4.5.1 WSDL ETD Operation

This section explains the basic operation of the SOAP WSDL ETD and how the ETD's structure corresponds to WSDL.

Note: *To support XMLDSIG in an WSDL context, the e*Way has to implement Multipurpose Internet Mail Extensions (MIME) binding as defined in the WSDL specifications. For complete WSDL specifications, see the following Web site:*

http://www.w3.org/TR/wsdl

WSDL Operation Elements

To tie your messages together as a request-response pair corresponding to a method call, you must define operations using the WSDL **<operation>** element. A WSDL operation specifies which message is the *input* and which message is the *output*.

Note: *For details on using WSDL and its structure, see* **"Using the Web Services Description Language" on page 117**.

Inside the WSDL file's **<operation>** element**,** you specify your **<input>** and **<output>** elements. Each element refers to the corresponding message by its fully qualified name. The collection of all WSDL operations (that is, methods) exposed by your service is called a **portType** (called PortType in this guide) and is defined using the WSDL **<portType>** element.

The **<operation>** element is a child of **<portType>**. You can name the **<portType>** whatever you want. The port type **name** attribute provides a unique name among all the PortTypes defined within the enclosing WSDL file. Each WSDL operation is named via the **name** attribute.

Each operation within a WSDL ETD (like its WSDL file counterpart) uses one of the following operation modes for communication:

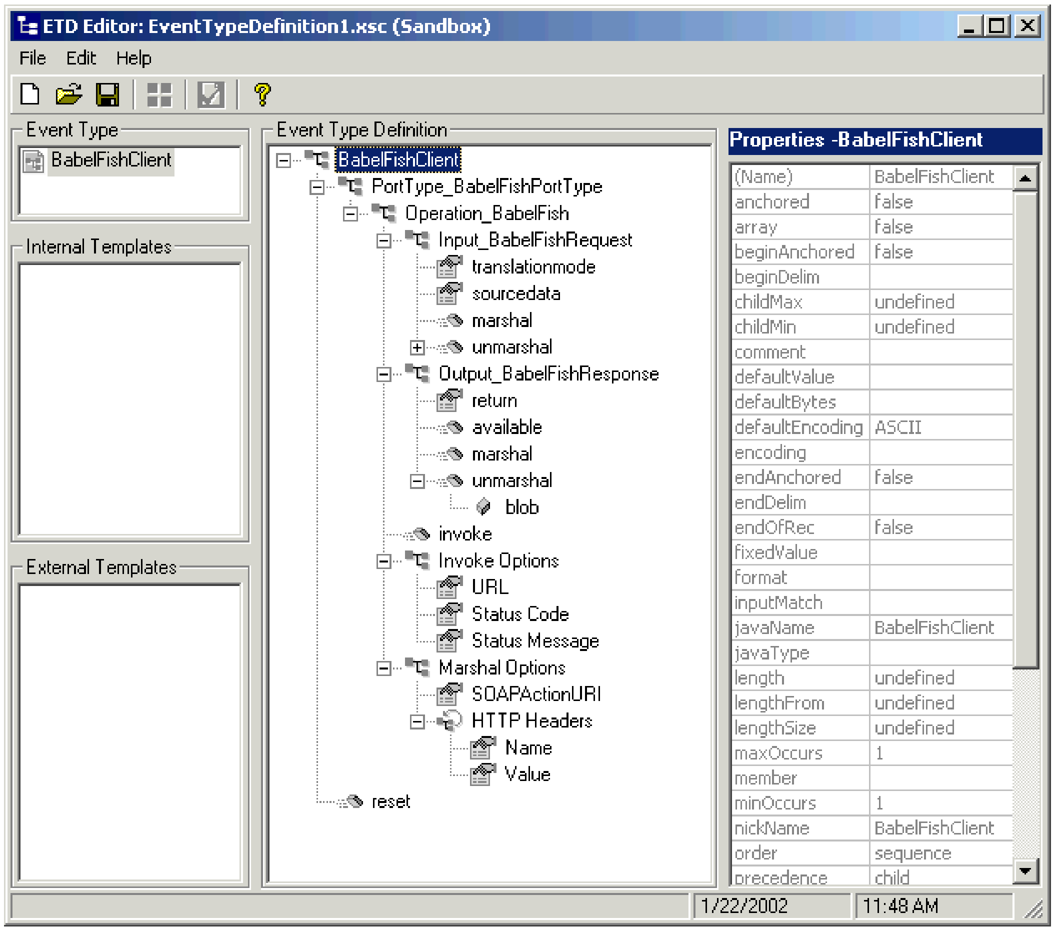
- **One-way:** The server receives a message from the client; also referred to as "fire and forget."
- **Request-response:** The server receives a message from the client and sends a correlated message back
- **Solicit-response:** The server sends a message to the client and receives a correlated message back.
- **Notification:** The server sends a message to the client.

Modes and Messages

The SOAP e*Way uses two types of WSDL ETDs, as you define using the SOAP ETD wizard (see **"SOAP ETD Wizard: WSDL" on page 34**). They are client and server. The wizard refers to these types as ETD e*Way modes.

Figure 26 shows a sample client-mode WSDL ETD, as it appears in the ETD Editor's Main window.

Figure 26 ETD Editor: BabelFish.xsc WSDL ETD



Event Type: BabelFishClient

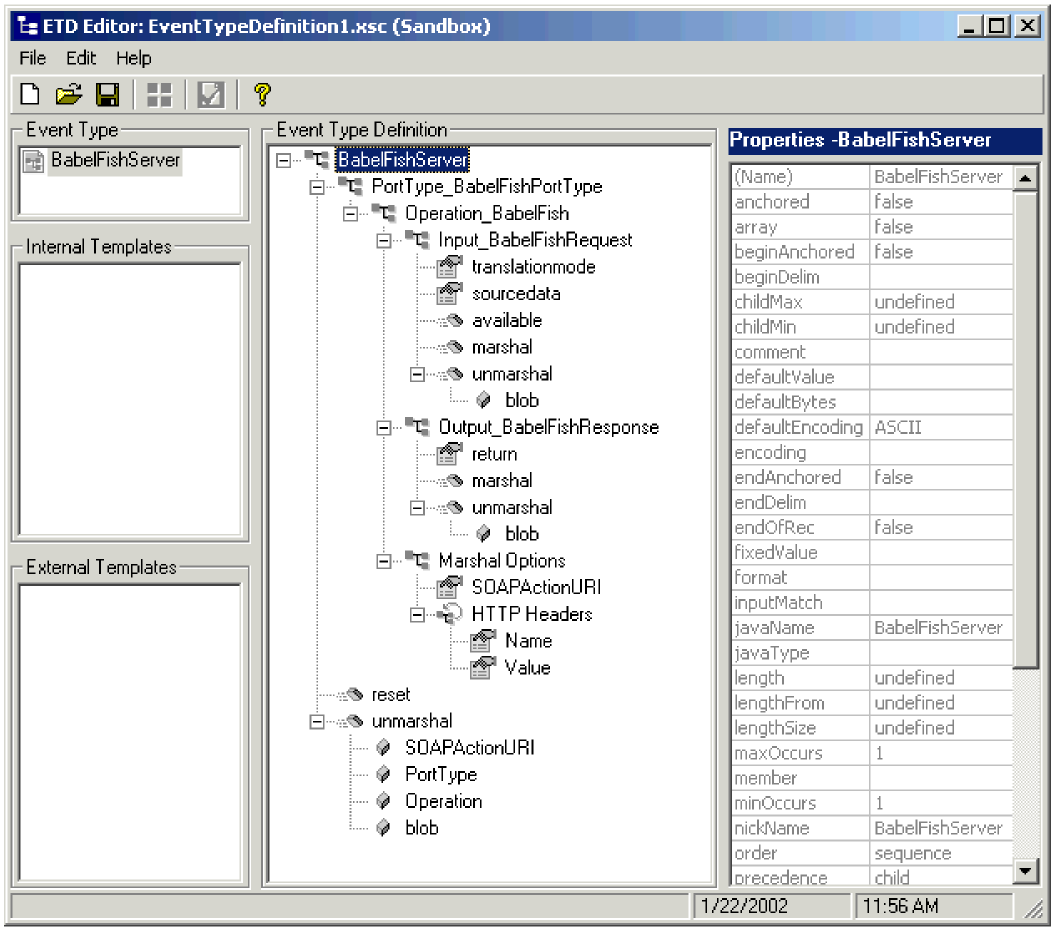
```
BabelFishClient
    PortType_BabelFishPortType
        Operation_BabelFish
            Input_BabelFishRequest
                translationmode
                sourcedata
                marshal
                unmarshal
            Output_BabelFishResponse
                return
                available
                marshal
                unmarshal
                    blob
            invoke
            Invoke Options
                URL
                Status Code
                Status Message
            Marshal Options
                SOAPActionURI
                HTTP Headers
                    Name
                    Value
    reset
```

(Name)	BabelFishClient
anchored	false
array	false
beginAnchored	false
beginDelim	
childMax	undefined
childMin	undefined
comment	
defaultValue	
defaultBytes	
defaultEncoding	ASCII
encoding	
endAnchored	false
endDelim	
endOfRec	false
fixedValue	
format	
inputMatch	
javaName	BabelFishClient
javaType	
length	undefined
lengthFrom	undefined
lengthSize	undefined
maxOccurs	1
member	
minOccurs	1
nickName	BabelFishClient
order	sequence
precedence	child

1/22/2002 11:48 AM

Figure 27 shows a sample server-mode WSDL ETD, as it appears in the ETD Editor's Main window.

Figure 27 ETD Editor: TopicRequest.xsc WSDL ETD



BabelFishServer

```
BabelFishServer
    PortType_BabelFishPortType
        Operation_BabelFish
            Input_BabelFishRequest
                translationmode
                sourcedata
                available
                marshal
                unmarshal
                    blob
            Output_BabelFishResponse
                return
                marshal
                unmarshal
                    blob
            Marshal Options
                SOAPActionURI
                HTTP Headers
                    Name
                    Value
    reset
    unmarshal
        SOAPActionURI
        PortType
        Operation
        blob
```

(Name)	BabelFishServer
anchored	false
array	false
beginAnchored	false
beginDelim	
childMax	undefined
childMin	undefined
comment	
defaultValue	
defaultBytes	
defaultEncoding	ASCII
encoding	
endAnchored	false
endDelim	
endOfRec	false
fixedValue	
format	
inputMatch	
javaName	BabelFishServer
javaType	
length	undefined
lengthFrom	undefined
lengthSize	undefined
maxOccurs	1
member	
minOccurs	1
nickName	BabelFishServer
order	sequence
precedence	child

1/22/2002 11:56 AM

WSDL ETD Structure

The SOAP e*Way classes for WSDL are generated and are specific to the original WSDL file. All the generated classes relating to the ETD have a similar structure, that is, they are based on a general template. The WSDL ETD has the following basic structure:

- **Root Node**
  - **PortType_XXX**
    - **Operation_XXX**
      - **Input_XXX**
      - **Output_XXX**
  - **PortType_XXX**
    - **Operation_XXX**
      - **Input_XXX**
      - **Output_XXX** (and so on)

Where **XXX** is the name for each element given in the original WSDL file.

For sample illustrations of how the ETDs are structured, see **Figure 26 on page 72** and **Figure 27 on page 73**.

Using WSDL ETDs

You can use and manipulate WSDL ETDs via the WSDL-specific Java classes and methods available with the e*Way. For details on each of these classes/methods, see **Chapter 7**.

The following methods are available with the ETD on a regular basis:

- **marshal** and **unmarshal** (blob) methods are available with each message.
- **marshal** options are available with each operation.

However, not all methods are available to use with every WSDL ETD. The following methods are available or not available, depending on the e*Way and operation modes:

- **invoke** and **invokeOptions** are available with an ETD operation only when it uses the following mode combinations:
  - **Client**: One way and request-response
  - **Server**: Solicit-response and notification

Note: *The **invoke**-related methods are only available in modes where the e*Way is initiating communication.*

- Top-level ETD node **unmarshal** methods are available only when one or more operations in the ETD are using the following mode combinations:
  - **Server**: One way and request-response
  - **Client**: Solicit-response and notification
- **available** methods only appear when the ETD is using the following mode and input/output message combinations:
  - **Client**: Request-response, solicit-response, and notification; for output messages only
  - **Server**: One way, request-response, and solicit-response; for input messages only

Note: *A fault message has the **available** method only when it is used in the client and request-response modes or the server and solicit-response modes.*

Table 6 lists the mode combinations and available methods shown in the previous list. The method names are called out in **boldface** type.

Table 6 ETD Mode Combination/Available Methods Matrix

Operation Mode	Using Client e*Way Mode	Using Server e*Way Mode
One way	▪ **invoke** method with operations	▪ Top-level method **unmarshal** ▪ **available** method for input message
Request-response	▪ **invoke** method with operations ▪ **available** method for output message	▪ Top-level method **unmarshal** ▪ **available** method for input message
Solicit-response	▪ Top-level method **unmarshal** ▪ **available** method for output message	▪ **invoke** method with operations ▪ **available** method for input message
Notification	▪ Top-level method **unmarshal** ▪ **available** method for output message	▪ **invoke** method with operations

4.5.2 Sender: WSDLBabelFish Schema

This schema demonstrates the use of the SOAP e*Way in implementing a Web client against the BabelFish Web Service. The schema uses a **BabelFishService.xsc** ETD file generated from the included **BabelFish.wsdl** file.

Schema and File

The name of this schema is WSDLBabelFish. It is contained in the file **WSDLBabelFish.zip** in the following directory on the installation CD-ROM:

samples\ewsoap

Table 7 lists the contents of the **WSDLBabelFish.zip** file.

Table 7 SecureSOAP Schema Files

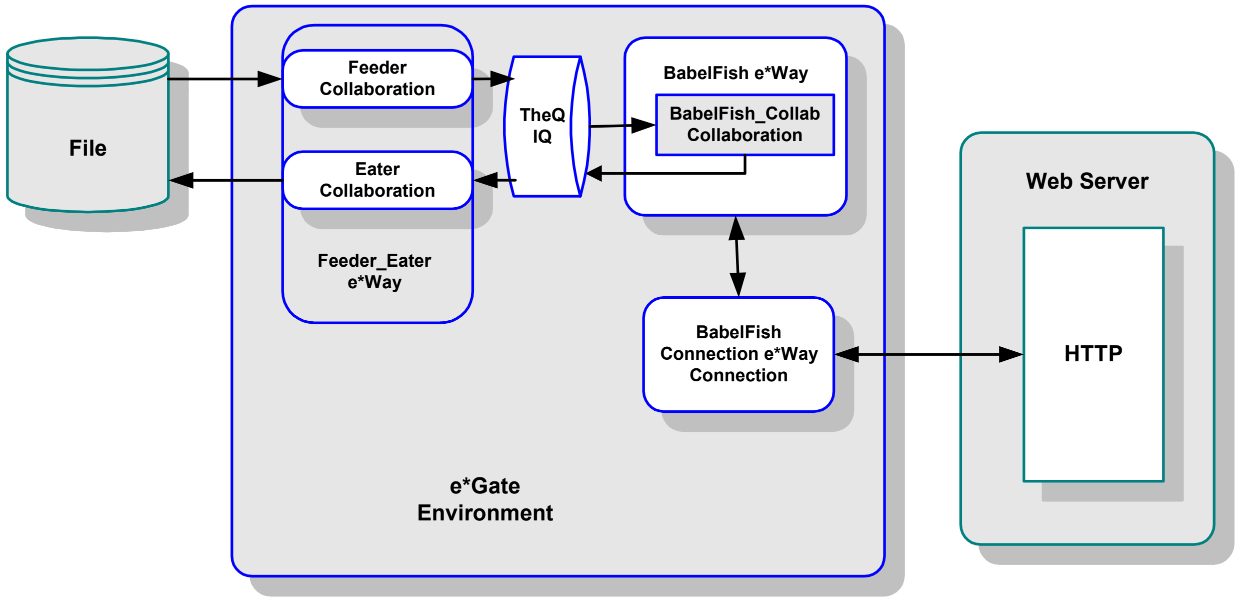
File Name	Description
WSDLBabelfish.zip	The actual schema that can be imported into e*Gate.
data/input/data.fin	A sample inbound data file.
wsdl/BabelFish.wsdl	The WSDL file used to generate the BabelFishService.xsc ETD.

There is no configuration needed for this schema. Simply run the schema in the same way as you would any other. Change the name of the input file to one with the appropriate extensions. The eGate system picks up the file, processes it, and sends it to the BabelFish SOAP Service.

Note: *See the information given under* **"SOAP Sender Implementation" on page 37** *for exact procedures on how to create this schema.*

Figure 28 shows a diagram of how this schema is set up.

Figure 28 WSDLBabelFish Schema Diagram



Schema Components

This sample WSDLBabelFish schema implementation consists of the following components:

e*Ways

- **Feeder_Eater** file e*Way reads text from an external source, applies a Pass Through Collaboration Rule, and publishes the information to an IQ that stores inbound data. It also receives the outbound message from the same IQ and publishes it externally to a file.
- **BabelFish** Multi-Mode (SOAP) e*Way applies extended Java Collaboration Rules to an inbound Event to translate the input data into SOAP and back again. In this case, the e*Way sends a SOAP message to a Web server via an e*Way Connection, receives a processed text response from the SOAP service (via the Web server), and publishes the response to an IQ.

e*Way Connection

- **BabelFishConnection** e*Way Connection provides the external connection between the Multi-Mode (SOAP) e*Way and the external Web server.

Event Types

- **GenericInEvent** contains raw data from the input file.
- **GenericOutEvent** contains the processed data output file.

Collaboration Rules

- **Feeder_CollabRules** is associated with the **Feeder_Eater** e*Way and **FeederCollab** Collaboration.
- **Eater_CollabRules** is associated with the **Feeder_Eater** e*Way and **EaterCollab** Collaboration.
- **BabelFish_CollabRules** is associated with the **BableFish** e*Way and **BabelFishCollab** Collaboration and is used to perform a transformation process, that is, translating the input data into SOAP and back again.

Collaborations

- **FeederCollab** is associated with the **Feeder_eater** e*Way and is used for polling the input Event (uses **Feeder_CollabRules**).
- **EaterCollab** is associated with the **Feeder_eater** e*Way and is used for sending the processed Event to the output file (uses **Eater_CollabRules**).
- **BabelFishCollab** is associated with the **BabelFish** e*Way and utilizes the **BabelFish_CollabRules**, sends the SOAP request to the Web server, receives a SOAP response from the Web server, and sends that response to the **EaterCollab** Collaboration.

IQ

- **TheIQ** receives data from the **Feeder_Eater** e*Way and sends it to the **BabelFish** e*Way. It also sends and receives data in the reverse direction.

WSDLBabelFish ETD

To create its SOAP ETD, the SOAP sender example WSDLBabelFish schema uses an input file named **BabelFishService.wsdl**. Use the SOAP ETD Wizard (WSDL) to convert this file to an ETD named **BabelFish.xsc**. For details on this operation, see **"SOAP ETD Wizard: WSDL" on page 34**.

Client WSDL File

The following example shows the text of the file **BabelFishService.wsdl**:

```
<?xml version="1.0"?>
<definitions name="BabelFishService" xmlns:tns="http://
www.xmethods.net/sd/BabelFishService.wsdl" targetNamespace="http://
www.xmethods.net/sd/BabelFishService.wsdl" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/
wsdl/soap/" xmlns="http://schemas.xmlsoap.org/wsdl/">
    <message name="BabelFishRequest">
        <part name="translationmode" type="xsd:string"/>
        <part name="sourcedata" type="xsd:string"/>
    </message>
    <message name="BabelFishResponse">
        <part name="return" type="xsd:string"/>
    </message>
    <portType name="BabelFishPortType">
        <operation name="BabelFish">
            <input message="tns:BabelFishRequest" />
            <output message="tns:BabelFishResponse" />
        </operation>
    </portType>
    <binding name="BabelFishBinding" type="tns:BabelFishPortType">
        <soap:binding style="rpc" transport="http://
schemas.xmlsoap.org/soap/http"/>
        <operation name="BabelFish">

<soap:operation soapAction="urn:xmethodsBabelFish#BabelFish"/>
            <input>

<soap:body use="encoded" namespace="urn:xmethodsBabelFish" encodingSt
yle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>

<soap:body use="encoded" namespace="urn:xmethodsBabelFish" encodingSt
yle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
    </binding>
    <service name="BabelFishService">

<documentation>Translates text of up to 5k in length, between a varie
ty of languages.</documentation>
        <port name="BabelFishPort" binding="tns:BabelFishBinding">
            <soap:address location="http://services.xmethods.net:80/
perl/soaplite.cgi"/>
        </port>
    </service>
</definitions>
```

ETD Structure

Figure 26 on page 72 shows the structure of the resulting **BabelFish.xsc** ETD, as it appears in the ETD Editor Main window.

Note: *When implementing the sample schema, be sure to name or rename the ETD file to **BabelFish.xsc**.*

Additional Information

Basic Implementation Steps

After you have located the SOAP service description, you are ready to implement the sample, following the basic steps outlined under **"Basic Implementation Steps" on page 42**.

See **"Sample Sender Schema: Manual Configuration" on page 41** for details on each of these steps.

Running the WSDLServer Schema

For details on how to run the WSDLServer schema, see **"To run the BabelFish schema" on page 63**.

4.5.3 Receiver: WSDLServer Schema

This schema implements a SOAP receiver setup, using WSDL. In most ways, this schema is similar to the SecureSOAP schema (for details, see **“SOAP Receiver Implementation” on page 63**).

Note: *Implementing and testing this schema requires the use of an HTTP server, for example, Apache/Tomcat.*

Schema and File

The name of this schema is WSDLServer. It is contained in the file **WSDLServer.zip** in the following directory on the installation CD-ROM:

samples\ewsoap

Table 8 lists the contents of the **WSDLServer.zip** file.

Table 8 WSDLServer Schema Files

File Name	Description
WSDLServer.zip	The actual schema that can be imported into e*Gate.
data/input/test.fin	A sample inbound data file.
wsdl/BabelFishServerClient.wsdl	The WSDL file used to generate the server and client ETDs.

Schema Operation

The sample schema contains the following elements/operations:

- A file e*Way that communicates with the SOAP client e*Way and an external system via an IQ
- A SOAP client e*Way that posts data to and receives it from an HTML page on an HTTP Web server
- A SOAP e*Way that implements the SOAP service
- A JMS IQ Manager that exchanges data with a SOAP server and the WSDL Web server

Schema Setup

The WSDLServer schema demonstrates the use of the SOAP e*Way in implementing a Web client, as well as a WSDL Web service. It contains a WSDL client that posts to and receives from an HTML page under an HTTP Web server, which in turns talks to a JMS IQ Manager, which in turn talks to a WSDL server.

The WSDL client communicates with a file e*Way with **feed** and **eat** Collaborations. The WSDL Web service implements a “dumb” BabelFish interpreter. No matter what is sent to the service, it always returns "Hello, world" in French.

The schema contains all items described previously, except the server and HTML page.

Configuring the Schema

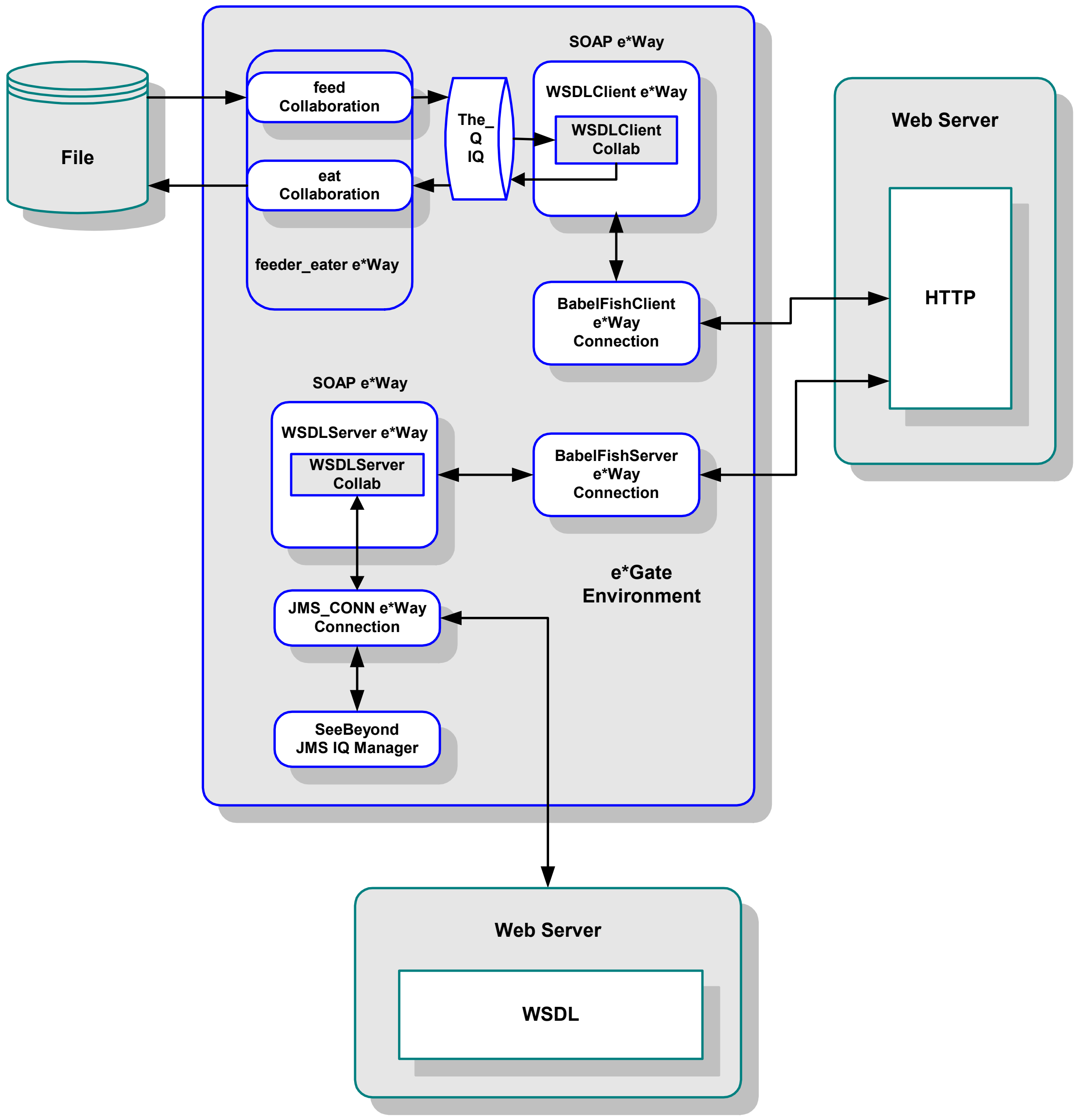
See information given under **“SOAP Receiver Implementation” on page 63**, on the SecureSOAP schema, for additional information on configuring this schema.

Note: *Implementing the SOAP receiver/server examples requires use of the CGI Web Server e*Way and its sample schemas. See the* ***CGI Web Server e*Way Intelligent Adapter User’s Guide (Java Version)*** *for details on implementing sample schemas for that e*Way.*

You can redefine the default WSDL Web service URL configured in the WSDL file and regenerate the client and server ETDs. As another option, you can also override the location in the Collaboration by setting the URL parameter under the ETD’s **InvokeOptions** node.

Figure 29 on page 81 shows a diagram of how this schema is set up.

Figure 29 WSDLServer Schema Diagram



Schema Components

This sample WSDLServer schema implementation consists of the following components:

e*Ways

- **feeder_eater** file e*Way reads text from an external source, applies a Pass Through Collaboration Rule, and publishes the information to an IQ that stores inbound data. It also receives the outbound message from the same IQ and publishes it externally to a file.
- **WSDLClient** Multi-Mode (SOAP) e*Way applies extended Java Collaboration Rules to an inbound Event to translate the input data into SOAP and back again. In this case, posts to an HTML page on an HTTP server, receives from the same HTML page, and publishes the response to an IQ.
- **WSDLServer** Multi-Mode (SOAP) e*Way implements the SOAP service, also applying Java Collaboration Rules. It receives a SOAP message from the JMS IQ Manager and returns the processed Event to the SOAP client via the **JMS_CONN** and **BabelFishServer** e*Way Connections and Web server.

e*Way Connections

- **BabelFishClient** provides the external connection between the **WSDLClient** e*Way and the HTTP Web server.
- **BableFishServer** provides the external connection between the **WSDLServer** e*Way and the HTTP Web server.
- **JMS_CONN** provides the external connection between the **WSDLServer** e*Way and the WSDL Web server, as well as the connection between that e*Way and the JMS IQ Manager.

Event Types

- **GenericInEvent** contains raw data from the input file.
- **GenericOutEvent** contains the processed data output file.
- **TopicRequest** contains the known topic used by the HTML page to publish the SOAP message to the **JMS_CONN** e*Way Connection.

Collaboration Rules

- **feed** uses the Pass Through Service and is associated with the **feeder_eater** e*Way and **feed** Collaboration.
- **eat** uses the Pass Through Service and is associated with the **feeder_eater** e*Way and **eat** Collaboration.
- **WSDLClientCollabRules** is associated with the **WSDLClientCollab** Collaboration and is used to perform a transformation process that translates the input data into SOAP and back again.
- **WSDLServerCollabRules** is associated with the **WSDLServerCollab** Collaboration and is also used to do a transformation process, implementing the SOAP service by doing the required calculation.

Collaborations

- **feed** is associated with the **feeder_eater** e*Way and **feed** Collaboration Rule and is used for polling the input Event.
- **eat** is associated with the **feeder_eater** e*Way and **eat** Collaboration Rule and is used for sending the processed Event to the output file.
- **WSDLClientCollab** is associated with the **WSDLClient** e*Way; this Collaboration performs a data transformation (via its Collaboration Rules), sends the SOAP request to the Web server, receives a SOAP response from the Web server, and sends that response to the **eat** Collaboration
- **WSDLServerCollab** is associated with the **WSDLServer** e*Way; this Collaboration implements the SOAP service via its Collaboration Rules, receives the Event from the **JMS_CONN** e*Way Connection, performs a data transformation (via Collaboration Rules), and sends the Event back to **JMS_CONN**.

IQ

- **The_IQ** receives data from the **feeder_eater** e*Way and sends it to the **WSDLClient** e*Way. It also sends and receives in the reverse direction.

WSDLServer ETD

To create its SOAP ETD, the SOAP receiver example WSDLServer schema uses an input file named **BabelFishServerClient.wsdl**. Use the SOAP ETD wizard (WSDL option) to convert this file to an ETD named **TopicRequest.xsc**.

See **"SOAP ETD Wizard: WSDL" on page 34** for procedures on how to use the SOAP ETD wizard and its WSDL option.

Server WSDL File

The following example shows the text of the file **BabelFishServerClient.wsdl**:

```
<?xml version="1.0"?>
<definitions name="BabelFishService" xmlns:tns="http://
www.xmethods.net/sd/BabelFishService.wsdl" targetNamespace="http://
www.xmethods.net/sd/BabelFishService.wsdl" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/
wsdl/soap/" xmlns="http://schemas.xmlsoap.org/wsdl/">
    <message name="BabelFishRequest">
        <part name="translationmode" type="xsd:string"/>
        <part name="sourcedata" type="xsd:string"/>
    </message>
    <message name="BabelFishResponse">
        <part name="return" type="xsd:string"/>
    </message>
    <portType name="BabelFishPortType">
        <operation name="BabelFish">
            <input message="tns:BabelFishRequest" />
            <output message="tns:BabelFishResponse" />
        </operation>
    </portType>
    <binding name="BabelFishBinding" type="tns:BabelFishPortType">
        <soap:binding style="rpc" transport="http://
schemas.xmlsoap.org/soap/http"/>
        <operation name="BabelFish">
```

```
            <soap:operation
soapAction="urn:xmethodsBabelFish#BabelFish"/>
            <input>
                <soap:body use="encoded"
namespace="urn:xmethodsBabelFish" encodingStyle="http://
schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
                <soap:body use="encoded"
namespace="urn:xmethodsBabelFish" encodingStyle="http://
schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
    </binding>
    <service name="BabelFishService">
        <documentation>Translates text of up to 5k in length, between
a variety of languages.</documentation>
        <port name="BabelFishPort" binding="tns:BabelFishBinding">
            <soap:address location="http://services.xmethods.net:80/
perl/soaplite.cgi"/>
        </port>
    </service>
</definitions>
```

ETD Structure

Figure 26 on page 72 shows the structure of the resulting **TopicRequest.xsc** ETD, as it appears in the ETD Editor Main window.

Note: *When implementing the sample schema, be sure to name or rename the ETD file to **TopicRequest.xsc**.*

Additional Information

Basic Implementation Steps

After you have located the SOAP service description, you are ready to implement the sample, following the basic steps outlined under **"Basic Implementation Steps" on page 42**.

See **"Sample Sender Schema: Manual Configuration" on page 41** for details on each of these steps.

Running the WSDLServer Schema

For details on how to run the WSDLServer schema, see **"To run the BabelFish schema" on page 63**. The schema components start automatically. When there are no more run-time messages, check the output file. If the schema is operating correctly, you can see that this file contains the translation of "Hello, world" into French.

4.6 Generic SOAP Schema

This schema implements a simple, generic SOAP setup, as explained in this section.

4.6.1 Schema and File

The name of this schema is GenericSOAP. It is contained in the file **GenericSOAP.zip** in the following directory on the CD-ROM:

samples\ewsoap

Table 9 lists the contents of the **GenericSOAP.zip** file.

Table 9 GenericSOAP Schema Files

File Name	Description
GenericSOAPSchema.zip	The schema that can be imported into e*Gate.
indata/gps.~in	The input file for the GPS service.
indata/msg2.~in	The input file for the weather/temperature service.
data	The directory where the output goes.

4.6.2 Configuring the Schema

Two data files are included with the schema sample. They connect with the following services:

- Global Positioning Service (GPS)
- Weather/temperature service

Import this schema into the e*Gate Schema Designer. When you examine it, you can see that the components, including the e*Way Connection, have already been set up for you.

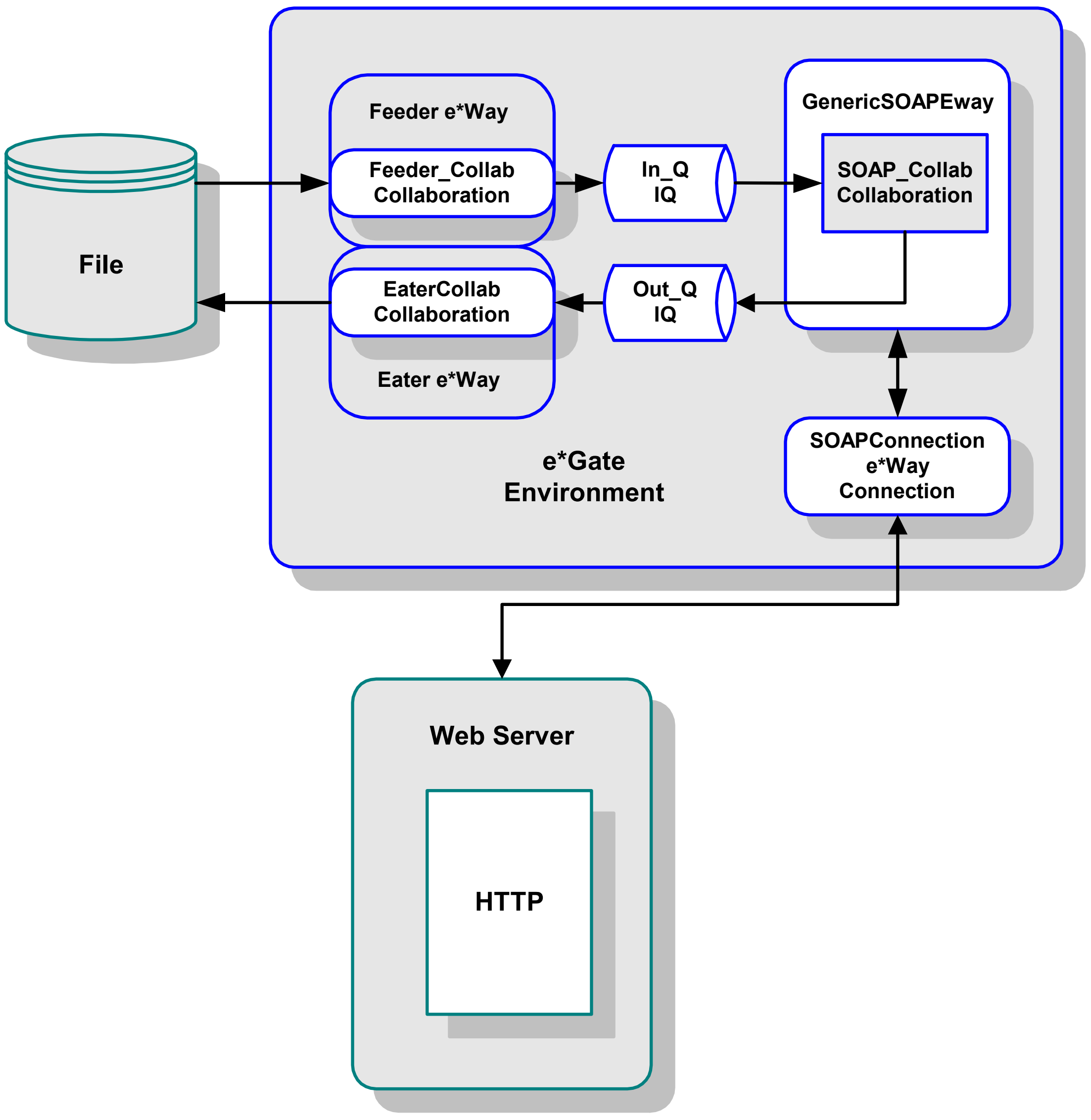
Note: *For more information on the sample implementations, see the* ***Readme.html*** *file that comes with your SOAP e*Way samples. Also, see* **"Defining Services" on page 118** *for more information on implementing a weather service accessed via the Web.*

Schema Operation

This is a generic schema that reads valid SOAP messages and sends them to any SOAP server. The ETD used with this schema is installed with e*Gate by default.

Figure 30 on page 86 shows a diagram of how this schema is set up.

Figure 30 GenericSOAP Schema Diagram


File
Feeder e*Way
Feeder_Collab Collaboration
EaterCollab Collaboration
Eater e*Way
In_Q IQ
Out_Q IQ
GenericSOAPEway
SOAP_Collab Collaboration
SOAPConnection e*Way Connection
e*Gate Environment
Web Server
HTTP

Schema Components

This sample GenericSOAP schema implementation consists of the following components:

e*Ways

- **Feeder** file e*Way reads text from an external source, applies Java Pass Through Collaboration Rules, and publishes the information to the **In_Q** IQ that stores inbound data.
- **Eater** file e*Way applies Java Pass Through Collaboration Rules, receives the outbound information from the **Out_Q** IQ, and publishes it externally to a file.

- **GenericSOAPEway** Multi-Mode (SOAP) e*Way applies extended Java Collaboration Rules to an inbound Event to translate the input data into SOAP and back again. In this case, the e*Way sends a SOAP message to a Web server via an e*Way Connection, receives a processed text response from the SOAP service (via the Web server), and publishes the response to the **Out_Q** IQ.

e*Way Connection

- **SOAPConnection** e*Way Connection provides the external connection between the **GenericSOAPEway** e*Way and the external Web server.

Event Types

- **GenericInEvent** contains raw data from the input file.
- **GenericOutEvent** contains the processed data output file.
- **SoapEvent** contains the information from the Web sites used to publish the SOAP message to the **SOAPConnection** e*Way Connection.

Collaboration Rules

- **JavaPass** is associated with the **Feeder** and **Eater** e*Ways, as well as the **Feeder_Collab** and **EaterCollab** Collaborations; this is a Java Pass Through Collaboration Rule.
- **GenericSOAPEway_Collab** is associated with the **GenericSOAPEway** e*Way and **SOAP_Collab** Collaboration and is used to perform a transformation process, that is, translating the input data into SOAP and back again.

Collaborations

- **Feeder_Collab** is associated with the **Feeder** e*Way and is used for polling the input Event (uses **JavaPass**).
- **EaterCollab** is associated with the **Eater** e*Way and is used for sending the processed Event to the output file (uses **JavaPass**).
- **SOAP_Collab** is associated with the **GenericSOAPEway** e*Way and utilizes the **GenericSOAPEway_Collab** Collaboration Rules, sends the SOAP request to the Web server, receives a SOAP response from the Web server, and sends that response to the **EaterCollab** Collaboration.

IQs

- **In_Q** receives data from the **Feeder** e*Way and sends it to the **GenericSOAPEway** e*Way.
- **Out_Q** receives data from the **GenericSOAPEway** e*Way and sends it to the **Eater** e*Way.

e*Way Connection Setup

When sending the weather/temperature SOAP message through the eGate system, the **SOAPConnection** (SOAP) e*Way Connection must be set to the following URL:

http://services.xmethods.net:80/soap/servlet/rpcrouter

When sending the GPS SOAP message through the eGate system, the **SOAPConnection** e*Way Connection must be set to the following URL:

http://216.101.160.38/xmlrpc/soap_api.php

The **SOAPConnection** e*Way Connection must be set to the appropriate URL (weather/temperature or GPS). Which URL you use depends on the data you feed through the schema.

Additional Information

Basic Implementation Steps

After you have located the SOAP service description, you are ready to implement the sample, following the basic steps outlined under **"Basic Implementation Steps" on page 42**.

See **"Sample Sender Schema: Manual Configuration" on page 41** for details on each of these steps.

Running the GenericSOAP Schema

For details on how to run the GenericSOAP schema, see **"To run the BabelFish schema" on page 63**. The schema components start automatically. When there are no more run-time messages, check the schema's output. If the schema is operating correctly, e*Gate sends the acquired weather or GPS information to the **data** directory.

Chapter 5

e*Way Connection Configuration

This chapter explains how to configure e*Way Connections for the e*Way Intelligent Adapter for SOAP.

5.1 Configuring e*Way Connections

Set up e*Way Connections using the e*Gate Schema Designer.

To create and configure e*Way Connections

1. In the Schema Designer's **Navigation** pane, select the **Component** tab.
2. Select the **e*Way Connections** folder.
3. On the palette, click on the icon to create a new **e*Way Connection**.

   The **New e*Way Connection Component** dialog box appears.
4. Enter a name for the **e*Way Connection**, then click **OK**. For the examples given in **Chapter 4**, the name is **SoapConnection.**

   An icon for your new e*Way Connection appears in the Navigation pane.
5. Double-click on the new **e*Way Connection** icon.

   The **e*Way Connection Properties** dialog box appears.
6. From the **e*Way Connection Type** drop-down box, select (for the examples) **SOAP**.
7. Enter **-1** for the **Event Type "get" interval** in the dialog box provided.
8. From the **e*Way Connection Configuration File**, click **New** to open the e*Way Configuration Editor.

Note: *To use an existing file, click **Find**.*

9. Use the e*Way Configuration Editor to create a new configuration file for this e*Way Connection. Do this operation by selecting the appropriate configuration parameters available in the interface.
10. When you are finished, close the e*Way Configuration Editor and save the new configuration file. For the examples given in **Chapter 4**, the file name is **SoapConnection.cfg.**

The rest of this chapter explains the SOAP e*Way Connection configuration parameters as follows:

- **"Connector" on page 90**
- **"WSDL Port Types" on page 91**
- **"Transport Binding" on page 92**
- **"Security" on page 92**
- **"Transport Level Retry" on page 94**
- **"HTTP" on page 95**
- **"Proxies" on page 96**
- **"HttpAuthentication" on page 98**
- **"SSL" on page 99**

5.2 Configuration Parameters

This section explains the configuration parameters for the SOAP e*Way Connection.

WSDL Parameters

Not all of the SOAP e*Way configuration parameters explained in this section are available when you are using WSDL ETDs. The following parameters are *not* available with WSDL:

- All parameters under **"Transport Binding" on page 92**
- All parameters under **"Security" on page 92**
- The following parameters under **"HTTP" on page 95**:
  - **"DefaultUrl" on page 95**
  - **"ContentType" on page 95**

Note: *To support XMLDSIG in an WSDL context, the e*Way has to implement Multipurpose Internet Mail Extensions (MIME) binding as defined in the WSDL specifications. For complete WSDL specifications, see the following Web site:*

http://www.w3.org/TR/wsdl

5.2.1 Connector

The parameters in the **Connector** section allow the Collaboration engine to identify the e*Way Connection.

Type

Description

Specifies the type of e*Way Connection.

Required Values

SOAP. The value defaults to **SOAP**.

Class

Description

Specifies the class name of the SOAP connector object.

Required Values

A valid package name. The default is **com.stc.eways.SOAP.SOAPConnector**.

Property.Tag

Description

Identifies the data source. This parameter is required by the current **EBobConnectorFactory**.

Required Values

A valid data source package name.

5.2.2 WSDL Port Types

Description

The parameter in the **WSDL Port Types** section configures values for the WSDL port-type bindings. When you are configuring the e*Way Connection, you can choose which **.def** file to use to create your current **.cfg** file.

This parameter is only available with WSDL ETDs.

Note: *You must first use the SOAP ETD wizard to create a WSDL ETD before these parameters become available in the e*Way Configuration Editor. This action creates an additional **.def** file for the e*Way's WSDL feature. The **.def** file takes its name from the name of the ETD, for example, **BabelFish.xsc** is accompanied by **BabelFish.def**. For details on the SOAP ETD wizard, see* **"Using the SOAP ETD Wizard" on page 29**.

Port Type Name

Description

Allows you to select the desired binding for the current port type. A configuration parameter appears for each port type contained in each ETD file. The e*Way Configuration Editor reads the available port types and bindings in each original WSDL file.

Required Values

Select the desired binding for each port type.

5.2.3 Transport Binding

Description

The parameters in the **Transport Binding** section configure the transport binding used by the SOAP e*Way when sending messages to the SOAP server.

Transport Type

Description

A transport binding to be used for posting SOAP messages.

Required Values

HTTP or **HTTPS**. The value defaults to **HTTP**.

SOAPAction URI

Description

This parameter specifies the **SOAPAction** URI header and is used only if the transport type is HTTP or HTTP(S).

Required Values

The value defaults to **com.stc.eways.soap.SOAP**, which is the only option.

SOAP Style

Description

This parameter specifies the SOAP style to use when interacting with a SOAP server.

Required Values

You can select either **RPC** or **Document** style. With **RPC** style, you can expect to receive a valid SOAP message or a valid MIME message (if the SOAP message has attachments). The valid message is unmarshaled into the **SOAPResponse** node of the SOAP ETD. With **Document** style, no response is expected. Calling **marshal** on the **SOAPResponse** node results in an empty SOAP document.

By default, **SOAP Style** is set to **RPC**. This value can be overridden by methods used in the SOAP ETD.

5.2.4 Security

Description

The parameters in this **Security** section allow you to specify the keys and certificates used by the SOAP e*Way to sign and verify SOAP messages.

KeyStore

Description

This parameter sets the default KeyStore file for use by the KeyManager. If the default KeyStore is not specified with this method, the KeyStore managed by KeyManager is empty.

Required Values

A valid KeyStore file name.

KeyStore Type

Description

This parameter sets the default KeyStore type. If the default KeyStore type is not set here, the default KeyStore type **JKS** is used. Other possible types include, for example, **PKCS12**.

Required Values

The name of a valid KeyStore type.

KeyStore Password

Description

This parameter sets the default KeyStore password. If the default KeyStore password is not set here, then the default KeyStore password is assumed to be " ".

Required Values

A valid KeyStore password.

Default Alias

Description

This parameter sets the alias name for the private key and the digital certificate. All entries in a KeyStore are identified by an alias. This parameter identifies the location of the private key and the digital certificate in the KeyStore. If **Default Alias** is not set, the default is assumed to be " ".

Required Values

A valid alias name for the private key and the digital certificate.

Signature Algorithm

Description

This parameter sets the signature algorithm to use when signing SOAP documents. One of these two algorithms *must* be set for the authentication to work. The default algorithm is **dsa-sha1**.

Required Values

The appropriate algorithm, either **dsa-sha1** or **rsa-sha1**.

5.2.5 Transport Level Retry

Description

The parameters in the **Transport Level Retry** section are related to the retry of transport posting. These parameters are used by the **SendToSOAPServer** function when it encounters errors at the transport level.

Timeout in Seconds

Description

This parameter is reserved for future use. Currently, the SOAP e*Way relies on the HTTP server to which the e*Way is posting to for time-out functionality.

Required Values

The number of seconds considered appropriate before timing out.

Retry Condition

Description

This parameter specifies the condition under which a retry of the transport posting is to be carried out. If **On Timeout Only** is chosen, the posting is retried only if the failure is due to a timeout on the connection. If **On Any Transport Failure** is chosen, the posting is retried on any transport failure.

Required Values

On Timeout Only or **On Any Transport Failure**. The default is **On Timeout Only**.

Number of Seconds to Wait Before Retry

Description

This parameter specifies the number of seconds to wait before the next retry of the transport posting. The e*Way sleeps through this period of time.

Required Values

The number of seconds considered appropriate before retrying the transport posting.

Maximum Retries

Description

This parameter specifies the maximum number of transport level retries the e*Way carries out before giving up and returning the appropriate status.

Required Values

The number of retries considered appropriate before giving up.

5.2.6 HTTP

This **HTTP** section contains a set of top-level parameters used by HTTP.

DefaultUrl

Description

Specifies the default URL to be used. If HTTPS protocol is specified, Secure Sockets Layer (SSL) must be configured. See the **"Using Secure Sockets Layer" on page 102**.

Required Values

A valid URL.

Additional Information

You must include the full URL, for example:

http://www.seebeyond.com

or

http://google.yahoo.com/bin/query

AllowCookies

Description

Specifies whether cookies sent from servers are stored and sent on subsequent requests. If cookies are not allowed, sessions are not supported.

Required Values

Yes or **No**.

ContentType

Description

Specifies the request content type.

Required Values

A string; the default is:

application/x-www-form-urlencoded

If you are sending other forms of data, set this parameter to the appropriate content type, for example:

text/xml

AcceptType

Description

Specifies the parameters for the **AcceptType** request header.

Required Values

A string. For example **text/html, text/plain, text/xml**, and so on.

5.2.7 Proxies

The **Proxies** parameters in this section specify the information required for the e*Way Connection to access the external systems through a proxy server.

Note: *When using proxy servers with Internet Information Services (IIS) Web servers, you must configure the proxy and IIS servers to release connections in a timely manner. Some proxies use **Keep-Alive** HTTP headers to keep connections open. If you cannot configure the proxy and IIS servers to release connections quickly, do not configure the IIS server with **Keep-Alive** headers. The SOAP e*Way does not use **Keep-Alive** headers and is therefore unaware when the proxy is keeping the connection alive.*

UseProxy

Description

Specifies whether an HTTP or HTTPS proxy is being used. If you set this parameter to HTTP, an HTTP proxy for a non-secured connection is used. If HTTPS is selected, an HTTPS proxy for a secured connection is used. Select **No** if a proxy is not used. See the following configuration parameters: **HttpProxyHost**, **HttpProxyPort**, **HttpsProxyHost**, **HttpsProxyPort**, **UserName**, and **PassWord** in this section.

Required Values

HTTP, **HTTPS**, or **No**.

HttpProxyHost

Description

Specifies the HTTP proxy host name to which to delegate requests to an HTTP server or reception of data from an HTTP server may be delegated to a proxy. This sets the proxy host for non-secured HTTP connections. To turn on proxy use, see the **UseProxy** configuration parameter.

Required Values

A valid HTTP proxy host name.

HttpProxyPort

Description

Specifies the HTTP proxy port to which requests to an HTTP server or reception of data from an HTTP server may be delegated to a proxy. This parameter sets the proxy port for non-secured HTTP connections. To turn on proxy use, see the **UseProxy** configuration parameter.

Required Values

A valid HTTP proxy port number.

HttpsProxyHost

Description

Specifies the HTTPS proxy host to which requests to an HTTP server or reception of data from an HTTP server may be delegated to a proxy. This sets the proxy port for secured HTTP connections. To turn on proxy use, see the **UseProxy** configuration parameter.

Required Values

A valid HTTPS proxy host number.

HttpsProxyPort

Description

Specifies the HTTPS proxy port to which requests to an HTTP server or reception of data from an HTTP server may be delegated to a proxy. This sets the proxy port for secured HTTP connections. To turn on proxy use, see the **UseProxy** configuration parameter.

Required Values

A valid HTTPS proxy port name.

UserName

Description

Specifies the user name necessary for authentication to access the proxy server. To turn on proxy use, see the **UseProxy** configuration parameter.

Required Values

A valid user name.

Additional Information

The user name is required by URLs that require **HTTP Basic Authentication** to access the site.

Important: *Enter a value for this parameter **before** you enter a value for the **PassWord** parameter.*

PassWord

Description

Specifies the password corresponding to the user name specified previously.

Required Values

The appropriate password.

Important: *Be sure to enter a value for the* ***UserName*** *parameter before entering the* ***PassWord*** *value.*

5.2.8 HttpAuthentication

The **HttpAuthentication** parameters in this section are used to perform HTTP authentication.

UseHttpAuthentication

Description

Specifies whether standard HTTP authentication is used. This is used when the Web site requires user name and password authentication. If this parameter is selected, the **UserName** and **PassWord** configuration parameters must be set. See **UserName** and **PassWord** configuration parameters in this section.

Required Values

Yes or **No**.

UserName

Description

Specifies the user name for standard HTTP authentication. See the **UseHttpAuthentication** configuration parameter.

Required Values

A valid user name.

Important: *Enter a value for this parameter* ***before*** *you enter a value for the* ***PassWord*** *parameter.*

PassWord

Description

Specifies the password associated with the specified user name for standard HTTP authentication. See **UseHttpAuthentication** configuration parameter.

Required Values

A valid password.

Important: *Be sure to enter a value for the **UserName** parameter before entering the **PassWord**.*

5.2.9 SSL

The parameters in this section control the information required to set up an SSL connection via HTTP.

UseSSL

Description

Specifies whether SSL needs to be configured in order to use the HTTPS protocol. If this parameter is set to **Yes**, at least **HttpsProtocolImpl** and **Provider** must be given.

Required Values

Yes or **No**.

HttpsProtocolImpl

Description

Specifies the package that contains the HTTP(S) protocol implementation. This specification adds the HTTP(S) **URLStreamHandler** implementation by including the handler's implementation package name to the list of packages searched by the Java URL class. The default value specified is the package that contains the Sun Microsystems reference implementation of the HTTPS **URLStreamHandler**.

Required Values

A valid package name. The default is **com.sun.net.ssl.internal.www.protocol**. This parameter is mandatory if you are using HTTP(S).

Provider

Description

Specifies the Cryptographic Service Provider. This parameter adds a JSSE provider implementation to the list of provider implementations. The default value specified is the Sun Microsystems reference implementation of the Cryptographic Service Provider **SunJSSE**.

Required Values

A valid provider name. The default is **com.sun.net.ssl.internal.ssl.Provider**. This parameter is mandatory if you are using HTTP(S).

X509CertificateImpl

Description

Specifies the implementation class of the **X509Certificate**.

Required Values

A valid package location. For example, if the implementation class is called, **MyX509CertificateImpl**, and it resides in the **com.radcrypto** package, specify **com.radcrypto.MyX509CertificateImpl**.

SSLSocketFactoryImpl

Description

Specifies the implementation class of the SSL socket factory.

Required Values

A valid package location. For example, if the implementation class is called **MySSLSocketFactoryImpl** and it resides in the **com.radcrypto** package, specify **com.radcrypto.MySSLSocketFactoryImpl**.

SSLServerSocketFactoryImpl

Description

Specifies the implementation class of the SSL server socket factory.

Required Values

A valid package location. For example, if the implementation class is called **MySSLServerSocketFactoryImpl** and it resides in **com.radcrypto** package, specify **com.radcrypto.MySSLServerSocketFactoryImpl**.

KeyStore

Description

Specifies the default KeyStore file for use by the KeyManager. If the default KeyStore is not specified with this method, the KeyStore managed by KeyManager is empty.

Required Values

A valid package location.

KeyStoreType

Description

Specifies the default KeyStore type. If the default KeyStore type is not set by this method, the default KeyStore type, **JKS** is used.

KeyStorePassword

Description

Specifies the default KeyStore password. If the default KeyStore password is not set by this method, the default KeyStore password is assumed to be “ “.

TrustStore

Description

Specifies the default TrustStore. If the default TrustStore is not set here, then a default TrustStore search is performed. If a TrustStore named **<java-home>/lib/security/jssecacerts** is found, it is used. If not, a search for a TrustStore name **<java-home>/lib/security/cacerts** is made, and used if located. If a TrustStore is not found, the TrustStore managed by the TrustManager is a new empty TrustStore.

Required Values

A valid TrustStore name.

TrustStoreType

Description

Specifies the default TrustStore type.

Required Values

A valid TrustStore type.

TrustStorePassword

Description

Specifies the default TrustStore password. If the default TrustStore password is not set by this method, the default TrustStore password is “ “.

KeyManagerAlgorithm

Description

Specifies the default KeyManager algorithm name to use. For example, the default KeyManager algorithm used in the Sun Microsystems reference implementation of JSSE is **SunX509**.

Required Values

A valid KeyManager algorithm name.

TrustManagerAlgorithm

Description

Specifies the default TrustManager algorithm name to use. For example, the default TrustManager algorithm used in the Sun Microsystems reference implementation of JSSE is **SunX509**.

Required Values

A valid TrustManager algorithm name.

Chapter 6

Additional Features

This chapter explains additional features available with the e*Way Intelligent Adapter for SOAP.

6.1 SOAP e*Way Special Features: Introduction

In addition to the standard e*Gate Integrator e*Way features, the SOAP e*Way offers you specialized features, including:

- **"Using Secure Sockets Layer" on page 102**
- **"Using SOAP Attachments" on page 110**
- **"Using Digital Signatures" on page 112**
- **"Using the Web Services Description Language" on page 117**

See the page/hyper-text references in the previous list for details on these features.

6.2 Using Secure Sockets Layer

The SSL feature offers hyper-text transfer protocol (HTTP) data exchanges security from interception, hackers, and other types of breaches. HTTP with SSL is called HTTP(S), meaning that SSL is enabled and provides security for any HTTP(S) data exchange.

The SSL feature is supported through the use of JSSE version 1.0.2. Currently, the JSSE reference implementation is used. JSSE is a provider-based architecture. Essentially, this means that there is a set of standard interfaces for cryptographic algorithms, hashing algorithms, secured-socket-layered URL stream handlers, and so on.

Because the user is interfacing with JSSE through these interfaces, the different components can be mixed and matched as long as the implementation is programmed under the published interfaces. However, some implementation cannot support a particular algorithm.

The JSSE 1.0.2 application programming interface (API) is capable of supporting SSL versions 2.0 and 3.0 and Transport Layer Security (TLS) version 1.0. These security protocols encapsulate a normal bidirectional stream socket and the JSSE 1.0.2 API adds transparent support for authentication, encryption, and integrity protection. The JSSE reference implementation implements SSL version 3.0 and TLS 1.0. It does not implement SSL 2.0.

For more information, see the following Web site:

http://java.sun.com/products/jsse/doc/guide/API_users_guide.html

Note: *See the JSSE documentation provided by Sun Microsystems for further details.*

6.2.1 KeyStores and TrustStores

JSSE makes use of files called KeyStores and TrustStores. A KeyStore is a database consisting of a private key and an associated certificate, or an associated certificate chain. The certificate chain consists of the client certificate and one or more certificate authority (CA) certificates.

A KeyStore contains a private key, in addition to the certificate, while TrustStore only contains the certificates trusted by the client (a "trust" store). The installation of the Java HTTP(S) e*Way installs a TrustStore file named **trustedcacertsjks**. This file can be used as the TrustStore for the e*Way.

A KeyStore is used by the e*Way for client authentication, while a TrustStore is used to authenticate a server in SSL authentication. Both KeyStore and TrustStores are managed via a utility called **keytool**, which is a part of the Java JDK installation.

Note: *To use **keytool**, you must set your CLASSPATH to **jcert.jar, jnet.jar**, and **jsse.jar**.*

The following line must also be added to the **jre\lib\security\java.security**:

```
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
```

See the installation manual for the JSSE version 1.0.2 for more information.

6.2.2 Methods for generating a KeyStore and TrustStore

This section explains steps on how to create a KeyStore and a TrustStore (or import a certificate into an existing TrustStore such as **trustedcacertsjks**). The primary tool used is **keytool**, but **openssl** is also used as a reference for generating **pkcs12** KeyStores. For more information on **openssl**, and available downloads, see the following Web site:

http://www.openssl.org.

Creating a TrustStore

For demonstration purposes, suppose you have the following CAs that you trust: **firstCA.cert**, **secondCA.cert**, and **thirdCA.cert**, located in the directory **C:\cascerts**. You can create a new TrustStore consisting of these three trusted certificates.

To create a new TrustStore

Use the following command:

```
keytool -import -file C:\cascerts\firstCA.cert -alias firstCA
    -keystore myTrustStore
```

You must enter this command two more times, but for the second and third entries, substitute **secondCA** and **thirdCA** for **firstCA**. Each of these command entries has the following purposes:

1. The first entry creates a KeyStore file name **myTrustStore** in the current working directory and imports the **firstCA** certificate into the TrustStore with an alias of **firstCA**. The format of **myTrustStore** is JKS.
2. For the second entry, substitute **secondCA** to import the **secondCA** certificate into the TrustStore, **myTrustStore**.
3. For the third entry, substitute **thirdCA** to import the **thirdCA** certificate into the TrustStore.

Once completed, myTrustStore is available to be used as the TrustStore for the e*Way. See **"TrustStore" on page 101** for more information.

Using an Existing TrustStore

This section explains how to use an existing TrustStore such as trustedcacertsjks. Notice that in the previous section, steps 2 and 3 were used to import two CAs into the TrustStore created in step 1.

For example, suppose you have a trusted certificate file named: **C:\trustedcerts\foo.cert** and want to import it to the **trustedcacertsjks** TrustStore.

If you are importing certificates into an existing TrustStore, use:

```
keytool -import -file C:\cacerts\secondCA.cert -alias secondCA
    -keystore trustedcacertsjks
```

Once you are finished, **trustedcacertsjks** can be used as the TrustStore for the e*Way. See **"TrustStore" on page 101** for more information.

6.2.3 Creating a KeyStore in JKS Format

This section explains how to create a KeyStore using the JKS format as the database format for both the private key, and the associated certificate or certificate chain. By default, as specified in the java.security file, **keytool** uses JKS as the format of the key and certificate databases (KeyStore and TrustStores). A CA must sign the certificate signing request (CSR). The CA is therefore trusted by the server-side application to which the e*Way is connected.

To generate a KeyStore

Use the following command:

```
keytool -keystore clientkeystore -genkey -alias client
```

You are prompted for several pieces of information required to generate a CSR. A sample key generation section follows:

```
Enter keystore password: seebeyond
What is your first and last name?
[Unknown]: development.seebeyond.com
What is the name of your organizational unit?
[Unknown]: Development
what is the name of your organization?
[Unknown]: SeeBeyond
What is the name of your City of Locality?
[Unknown]: Monrovia
What is the name of your State or Province?
[Unknown]: California
What is the two-letter country code for this unit?
[Unknown]: US
Is<CN=Foo Bar, OU=Development, O=SeeBeyond, L=Monrovia,
ST=California, C=US> correct?
[no]: yes

Enter key password for <client>
    (RETURN if same as keystore password):
```

If the KeyStore password is specified, then the password must be provided for the e*Way. Press RETURN when prompted for the key password (this action makes the key password the same as the KeyStore password).

This operation creates a KeyStore file **clientkeystore** in the current working directory. You must specify a fully-qualified domain for the “first and last name” question. The sample uses **development.seebeyond.com**. The reason for this use is that some CAs such as Verisign expect this parameter to be a fully qualified domain name.

There are CAs that do not require the fully qualified domain, but it is recommended to use the fully-qualified domain name for the sake of portability. All the other information given must be valid. If the information can not be validated, Certificate Authority such as Verisign does not sign a generated CSR for this entry.

This KeyStore contains an entry with an alias of **client**. This entry consists of the Generated private key and information needed for generating a CSR as follows:

```
keytool -keystore clientkeystore -certreq alias client -keyalg rsa
    -file client.csr
```

This command generates a certificate signing request which can be provided to a CA for a certificate request. The file **client.csr** contains the CSR in PEM format.

Some CA (one trusted by the Web server to which the e*Way is connecting) must sign the CSR. The CA generates a certificate for the corresponding CSR and signs the certificate with its private key. For more information, visit:

http://www.thawte.com

or

http://www.verisign.com

If the certificate is chained with the CA's certificate, perform step A; otherwise, perform step B in the following list:

A The following command assumes the client certificate is in the file **client.cer** and the CA's certificate is in the file **CARoot.cer**:

```
keytool -import -keystore clientstore -file client.cer -alias
    client
```

This command imports the certificate (which can include more than one CA in addition to the Client's certificate).

B The following command imports the CA's certificate into the KeyStore for chaining with the client's certificate:

```
keytool -import -keystore clientkeystore -file CARootcer -alias
    theCARoot
```

C The following command imports the client's certificate signed by the CA whose certificate was imported in the preceding step:

```
keytool -import -keystore clientkeystore -file client.cer -alias
    client
```

The generated file **clientkeystore** contains the client's private key and the associated certificate chain used for client authentication and signing. The KeyStore and/or **clientkeystore**, can then be used as the e*Way's KeyStore. See the **"KeyStore" on page 100** for more information.

6.2.4 Creating a KeyStore in PKCS12 Format

This section explains how to create a PKCS12 KeyStore to work with JSSE. In a real working environment, a customer could already have an existing private key and certificate (signed by a known CA). In this case, JKS format can not be used, because it does not allow the user to import/export the private key through **keytool**. It is necessary to generate a PKCS12 database consisting of the private key and its certificate.

The generated PKCS12 database can then be used as the e*Way's KeyStore. The **keytool** utility is currently lacking the ability to write to a PKCS12 database. However, it can read from a PKCS12 database.

Note: *There are additional third-party tools available for generating PKCS12 certificates, if you want to use a different tool.*

For the following example, **openssl** is used to generate the PKCS12 KeyStore:

```
cat.mykey.pem.txt mycertificate.pem.txt>mykeycertificate.pem.txt
```

The existing key is in the file **mykey.pem.txt** in PEM format. The certificate is in **mycertificate.pem.txt**, which is also in PEM format. A text file must be created which contains the key followed by the certificate as follows:

```
openssl pkcs12 -export -in mykeycertificate.pem.txt -out
    mykeystore.pkcs12 -name myAlias -noiter -nomaciter
```

This command prompts the user for a password. The password is required. The KeyStore fails to work with JSSE without a password. This password must also be supplied as the password for the e*Way's KeyStore password (see **"KeyStorePassword" on page 100**).

This command also uses the **openssl pkcs12** command to generate a PKCS12 KeyStore with the private key and certificate. The generated KeyStore is **mykeystore.pkcs12** with an entry specified by the **myAlias** alias. This entry contains the private key and the certificate provided by the **-in** argument. The **noiter** and **nomaciter** options must be specified to allow the generated KeyStore to be recognized properly by JSSE.

6.2.5 SSL Handshaking

There are two options available for setting up SSL connectivity with a Web server:

- **Server-side authentication:** The majority of eCommerce Web sites on the Internet are configured for server-side authentication. The e*Way requests a certificate from the Web server and authenticates the Web server by verifying that the certificate can be trusted. Essentially, the e*Way does this operation by looking into its TrustStore for a CA certificate with a public key that can validate the signature on the certificate received from the Web server.
- **Dual authentication:** This option requires authentication from both the e*Way and Web server. The server side (Web server) of the authentication process is the same as that described previously. However, in addition, the Web server requests a certificate from the e*Way. The e*Way then sends its certificate to the Web server. The server, in turn, authenticates the e*Way by looking into its TrustStore for a matching trusted CA certificate. The communication channel is established by the process of both parties' requesting certificate information.

For illustrations of both these types of authentication, see the following figures:

- **Figure 31 on page 108** shows a diagram of the SSL handshake dialog for server-side authentication.
- **Figure 32 on page 109** shows a diagram of the SSL handshake dialog for dual authentication.

Figure 31 Server-side Authentication

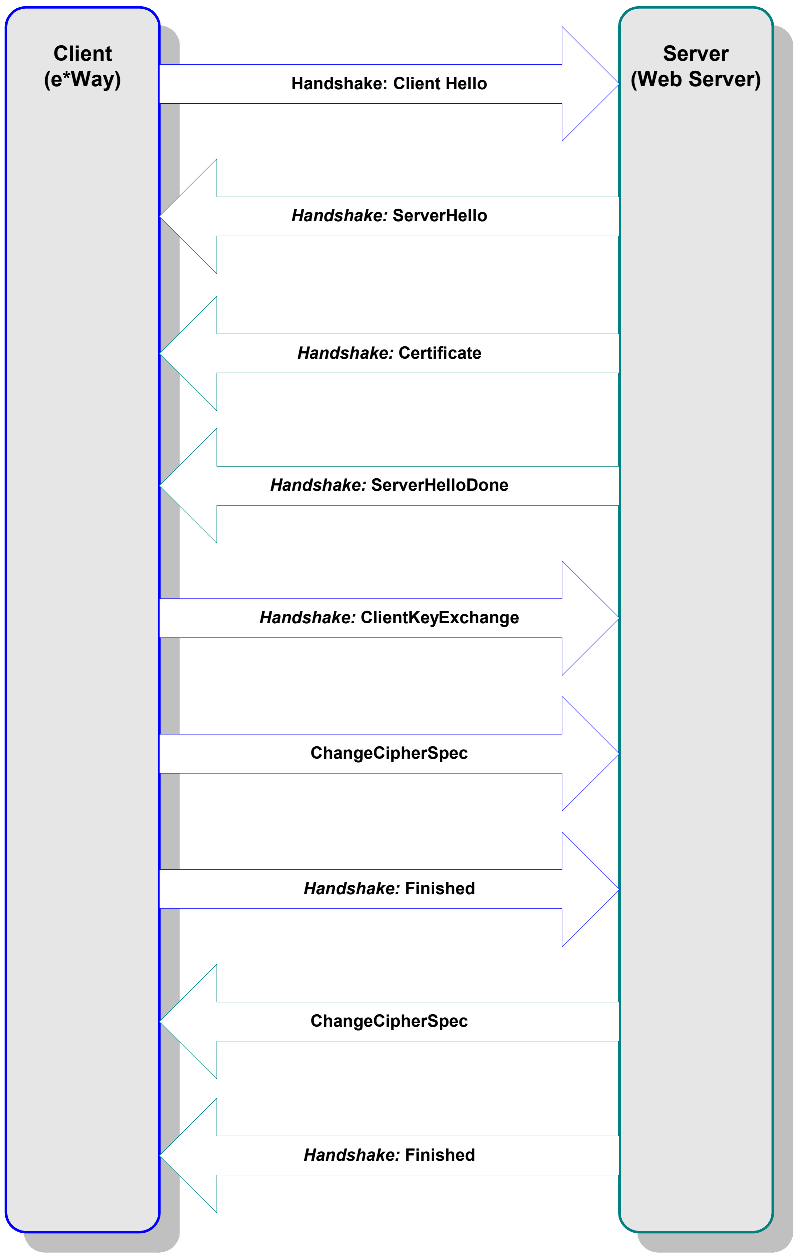


Figure 32 Dual Authentication

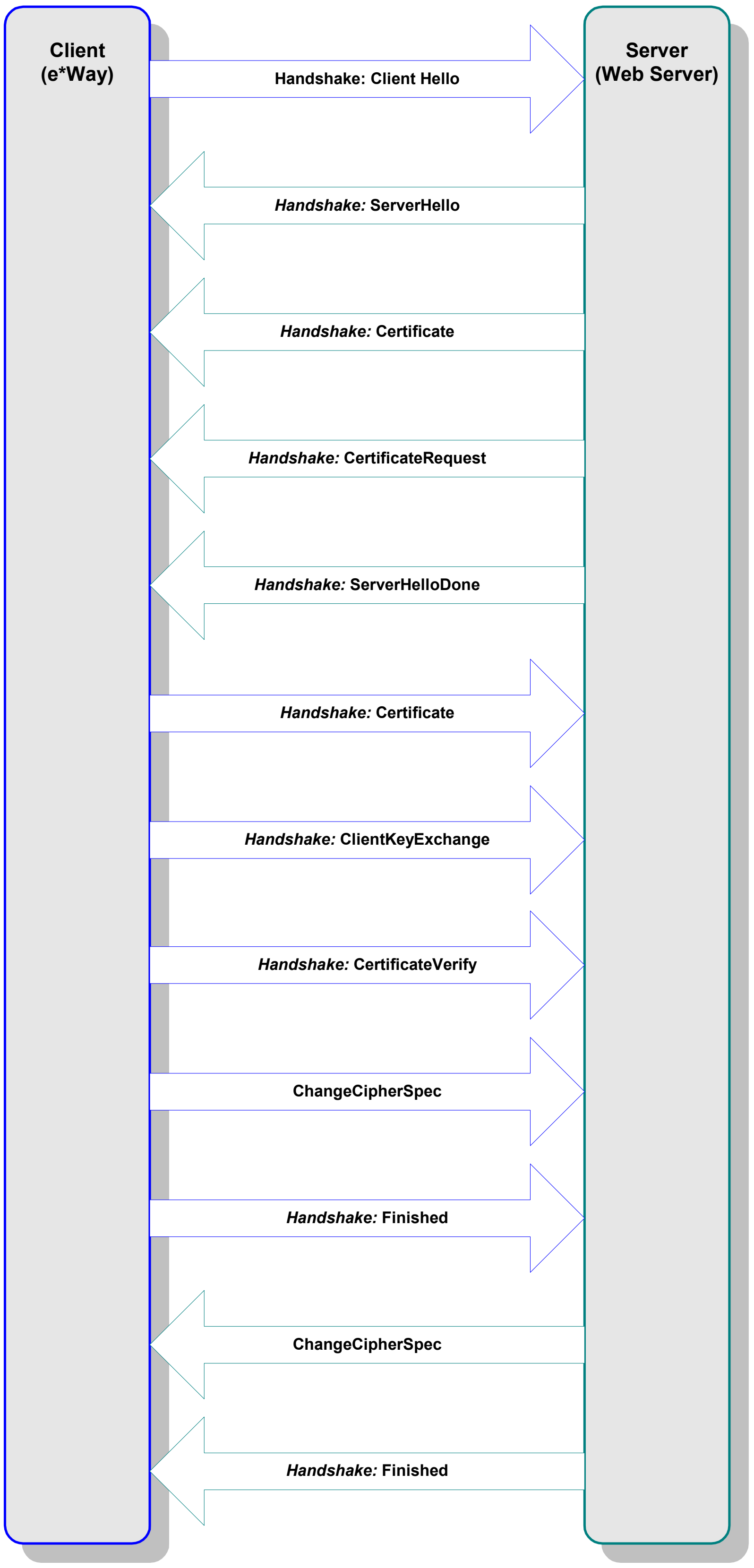
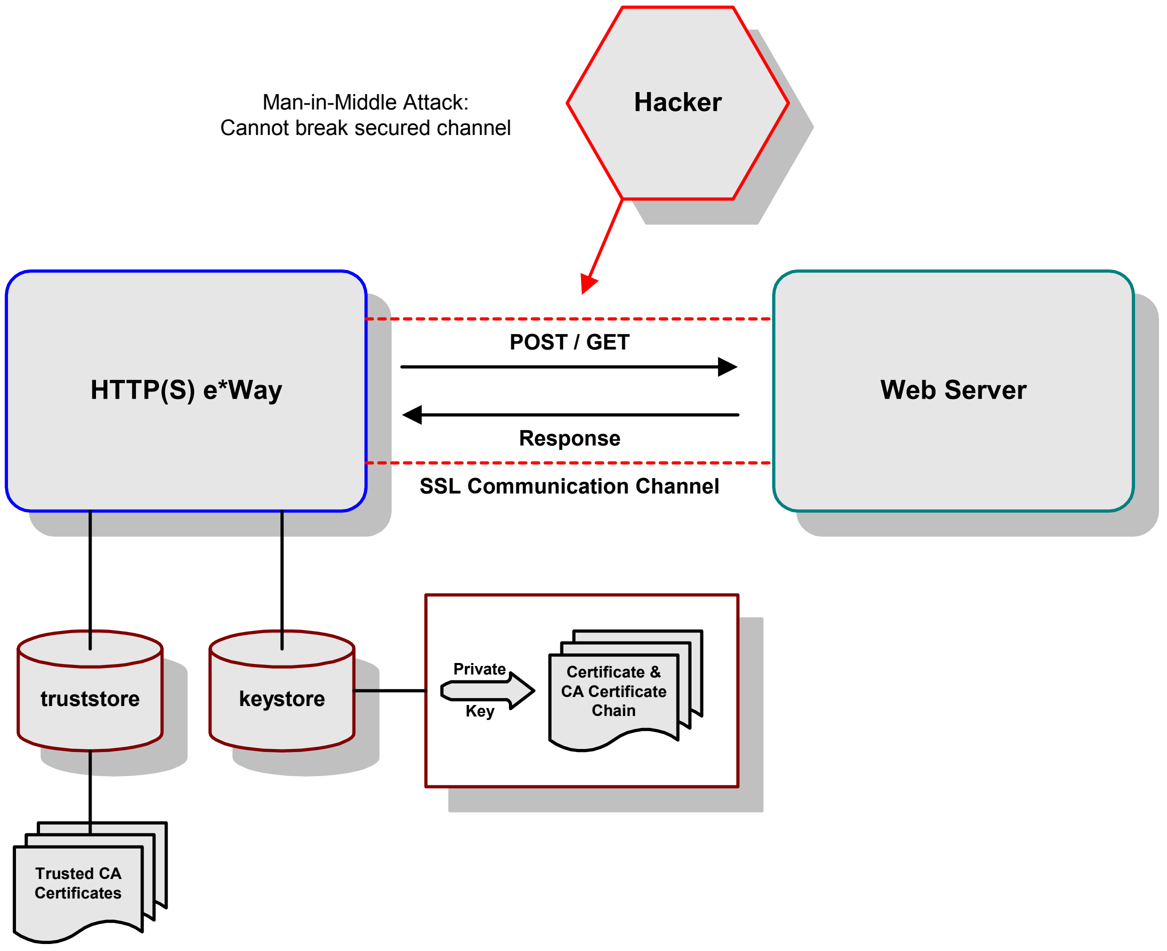


Figure 33 shows a diagram of general SSL operation with the HTTP(S) e*Way.

Figure 33 General SSL Operation: HTTP(S) e*Way



Note: *See the* ***HTTP(S) e*Way Intelligent Adapter User's Guide*** *for details on how to use this e*Way.*

6.3 Using SOAP Attachments

You can send a SOAP message together with attachments of various sorts, ranging from fax images to art drawings or images. These attachments are usually transmitted in some type of binary format. For example, most images on the Internet are transmitted using either the **.gif** or **.jpg** file data formats.

6.3.1 SOAP Attachments: Overview

Web service developers can specify that a Web service's methods are to use SOAP attachments to transport binary data or large Extensible Markup Language (XML) documents.

One typical use of SOAP attachments is for transporting intact, binary data such as image files. For another example, your system may need to transport XML documents to other parts of the system without the overhead of validating them. These XML documents that do not need to conform to your particular schema or Document Type Definition (DTD) can be passed as attachments.

By using SOAP attachments, the SOAP message body is much smaller because it contains only a reference to the data and not the data itself. Using attachments can be more efficient because smaller SOAP messages are processed more quickly than extremely large messages, and the translation of the data to Java objects is reduced for attachments.

SOAP Attachment Implementation

SOAP attachments are implemented by wrapping the SOAP message and one or more attachments in an envelope of Multipurpose Internet Mail Extensions (MIME). The developer uses, for example, the Web Service Builder (or an equivalent) to map method parameters or return value data types to MIME types.

Table 10 shows the Java data types that can be set to use SOAP attachments. The table also shows the MIME type that each Java type can be mapped to.

Table 10 Appropriate Mappings for Java Types to MIME Types

Java Data Type	MIME Type Mapping
java.lang.String	text/plain, text/xml
org.w3c.dom.Document	text/xml
byte[]	image/gif, image/jpeg, application/octet-stream
java.io.Serializable	application/x-java-serialized-object

To make a Web service use SOAP attachments

1. From the Web Service Builder (for example), select the method parameter or return value for which you want to use attachments.
2. Place a check next to **Part is Attachment**, if it is available.
3. Select from the list of **Available Types** an appropriate MIME type you want the application to use as an attachment.
4. Click **Add Type**.
5. If you want more than one type of attachment for your application, Repeat steps 2 and 3 for more types, as necessary. For example, you can allow attachments of both **image/gif** and **image/jpeg** MIME types.

After the Web service is deployed and the method that uses attachments is invoked, the MIME attachment is translated to a Java object by an appropriate data content handler, depending on the MIME data type. For example, the **text/xml MIME** type is translated to a **w3c.dom.Document** object.

6.3.2 Associating SOAP Messages and Attachments

This section explains a standard way to associate a SOAP message with one or more attachments in their native format in a multipart MIME structure for transport. The specification combines a specific usage of **Multipart/Related** MIME media type and the URI schemes explained on the World Wide Web Consortium (W3C) Web site, for referencing MIME parts.

For additional details, see the SOAP-related pages of the W3C Web site at the following URL:

http://www.w3c.org

The processes explained in this section treat the multipart MIME structure as essentially a part of the transfer protocol binding, that is, on par with the transfer protocol headers, as far as the SOAP message is concerned. This multipart structure is called the SOAP message package.

The purpose of this section is to show how to use existing facilities in SOAP, as well as standard MIME mechanisms, to carry and reference attachments, using existing standards and without inventing ways on your own. Most Internet communication protocols can transport MIME-encoded content, although some special considerations are required for the Hyper-text Transfer Protocol (HTTP).

6.4 Using Digital Signatures

The purpose of digital signatures is to allow the recipient of a data object, usually a message, to verify the data’s authenticity. Digital signatures are a value computed with a cryptographic algorithm and appended to a data object in such a way that any recipient of the data can use the signature to verify the data's origin and integrity.

This section explains the syntax and processing rules of a SOAP header entry to carry digital signature information within a SOAP 1.1 envelope.

6.4.1 Header Entry Syntax

This section explains the syntax for SOAP header entry.

Namespace

The XML namespace **[XML-ns]** URI that must be used by implementations of this specification and can be found at the following URI:

http://schemas.xmlsoap.org/soap/security/2000-12

The namespace prefix **SOAP-SEC** used in this specification is associated with this URI.

Signature Header Entry

The header entry **<SOAP-SEC:Signature>** is defined by the following schema:

[XML-Schema1], [XML-Schema2]

The **<SOAP-SEC:Signature>** element contains a single digital signature conforming to the XML-signature specification **[XML-Signature]** as follows:

```
<schema
  xmlns="http://www.w3.org/1999/XMLSchema"
  xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
  targetNamespace="http://schemas.xmlsoap.org/soap/security/2000-12"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

  <import namespace="http://www.w3.org/2000/09/xmldsig#"/>
  <import namespace="http://schemas.xmlsoap.org/soap/envelope/"/>

  <element name="Signature" final="restriction">
    <complexType>
      <sequence>
        <element ref="ds:Signature" minOccurs="1" maxOccurs="1"/>
      </sequence>
      <attribute name="id" type="ID" use="optional"/>
      <attribute ref="SOAP-ENV:actor" use="optional"/>
      <attribute ref="SOAP-ENV:mustUnderstand" use="optional"/>
    </complexType>
  </element>

  <attribute name="id" type="ID"/>

</schema>
```

SOAP-SEC:id Attribute

The **<ds:Reference>** element *must* refer to the signed part of the SOAP envelope. This reference can be achieved through the use of XML identifiers. Applications are responsible for determining which attributes are of the type **ID**.

To help applications identify attributes of the type **ID**, this specification defines the **SOAP-SEC:id** global attribute. This attribute can be used for referencing the signed part of the SOAP envelope.

Example of SOAP Message with Signature Header

Here is an example of a SOAP message with a signature header entry, where the SOAP body is signed and the resulting signature **<ds:Signature>** is added to the **<SOAP-SEC:Signature>** header entry. Note that the **URI** attribute of the **<ds:Reference>** element refers to the **<SOAP-ENV:SOAP-Body>** element. The example follows:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <SOAP-SEC:Signature
      xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-
12"
      SOAP-ENV:actor="some-URI"
      SOAP-ENV:mustUnderstand="1">
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod
            Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-
20001026">
          </ds:CanonicalizationMethod>
          <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/
xmldsig#dsa-sha1"/>
          <ds:Reference URI="#Body">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/TR/2000/CR-
xml-c14n-20001026"/>
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/
xmldsig#sha1"/>
            <ds:DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</
ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>MC0CFFrVLtRlk=...</ds:SignatureValue>
      </ds:Signature>
    </SOAP-SEC:Signature>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body
    xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
    SOAP-SEC:id="Body">
    <m:GetLastTradePrice xmlns:m="some-URI">
      <m:symbol>IBM</m:symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

6.4.2 Processing Rules

The **Signature** header entry is used to carry a signature compliant with the XML-signature specification **[XML-Signature]** within a SOAP envelope. It can be used for signing one or more elements in the SOAP envelope. Multiple signature header entries may be added into a single SOAP envelope with either disjoint or overlapping signed elements.

Note: *A future version of this specification may allow signature syntax other than using* ***[XML-Signature]****, through extension* ***[XML-Schema1]*** *of the content model of* ***<SOAP-SEC:Signature>****.*

SOAP applications conforming to this specification *must* satisfy the following conditions:

- The application must be capable of processing an XML signature as defined in the XML-signature specification **[XML-Signature]**.
- If a conforming SOAP application is to add a **<SOAP-SEC:Signature>** header entry in the SOAP Header, the header entry must have a **<ds:Signature>** element conforming to **[XML-Signature]**. All the **<ds:Reference>** elements contained in the signature must refer to a resource within the enclosing SOAP envelope or to a resource in the enclosing SOAP message package **[SOAP-attachment]** if the envelope is the primary SOAP 1.1 message **[SOAP-attachment]** of the package.
- When a conforming SOAP application receives a SOAP message containing one or more **<SOAP-SEC:Signature>** header entries intended for the application (either it is explicitly specified by the SOAP **actor** attribute, or the application is the ultimate destination), for each such header entry, the application must perform the following steps:
    - Decide whether to process the header entry (either forced by the **mustUnderstand="1"** attribute or voluntarily).
    - If it is to be processed, the application must try to validate the signature using the processing model of **[XML-Signature]**.

The XML canonicalization **[XML-C14N]** of **<ds:SignedInfo>** and other signed resources must each be done within its own context. In other words, the canonical form **[XML-C14N]** of **<ds:SigndInfo>** always inherits the namespace declarations for **SOAP-ENV** and **SOAP-SEC**.

Signature Header Entry Generation

This section explains the operations you must perform for the signature header entry.

To create a <SOAP-SEC:Signature> header

1. Prepare the target SOAP envelope with the body and necessary headers.
2. Create a template of a **<ds:Signature>** element. The template is assumed to contain empty contents for **<ds:DigestValue>** or **<ds:SignatureValue>** elements, but contains appropriate values for the elements such as **<ds:SignatureMethod>** and **<ds:Reference>** required to calculate them.
3. Create a new header entry **<SOAP-SEC:Signature>** and add the template to this entry.
4. Add the header entry **<SOAP-SEC:Signature>** to the SOAP header.
5. Add the SOAP **actor** and **mustUnderstand** attributes to the entry, if necessary.
6. Calculate the **<ds:DigestValue>** and **<ds:SignatureValue>** elements according to the core generation of the XML-signature specification **[XML-Signature]**.

XPath filtering can be used to specify objects to be signed, as described in **[XML-Signature]**. However, since the SOAP message exchange model allows intermediate applications to modify the envelope (add or delete a header entry, for example), **XPath** filtering does not always result in the same objects after message delivery.

Take care in using **XPath** filtering so there is no subsequent validation failure because of such modifications. To do so, use the following transform:

http://www.w3.org/2000/09/xmldsig#enveloped-signature

It is defined in **[XML-Signature]** and is useful when signing the entire envelope, including other header entries (if any).

Signature Header Entry Validation

The validation of a **<SOAP-SEC:Signature>** header entry fails under the following conditions:

- The syntax of the content of the header entry does not conform to this specification.
- The validation of the signature contained in the header entry fails according to the core validation of the XML-signature specification **[XML-Signature].**
- The receiving application program rejects the signature for one of its own reasons (for example, the signature is created by an untrusted key).

Note: *If the validation of the signature header entry fails, applications usually report the failure to the sender. See the W3C SOAP 1.1 specifications for information on how to deal with validation failures.*

Security Considerations

The specifications provided in this section define the use of **[XML-Signature]** in SOAP 1.1 headers. As one of the building blocks for securing SOAP messages, **[XML-Signature]** is intended to be used in conjunction with other security techniques. Digital signatures must be understood in the context of other security mechanisms and possible security threats, then used accordingly.

For example, digital signatures alone do not provide message authentication. You can record a signed message and resend it (replay attack). To prevent this type of attack, digital signatures must be combined with an appropriate means to ensure the uniqueness of the message, for example, nonces or time stamps. One way to add this information is to place an extra **<ds:Object>** element as a child of the **<ds:Signature>**.

When digital signatures are used for verifying the identity of the sending party, the sender must prove the possession of the private key. One way to achieve this proof is to use a challenge-response type of protocol.

Implementers must also be aware of all the security implications resulting from the use of digital signatures in general and **[XML Signature]** in particular. For building trust into an application based on a digital signature, the following additional pieces of technology must be identified in relation to the signature:

- Well-defined certificate trust model, whether hierarchical or peer-to-peer
- Generation and maintenance of trusted key pairs and certificates
- Validation that a certificate has not been revoked

6.5 Using the Web Services Description Language

The SOAP e*Way allows your system to communicate using the Web Services Description Language (WSDL). This language is XML based and is used to define Web services and describe how to access them. Fortunately, you do not need to learn all the details of this language because there are tools that generate WSDL for you. However, you do need some basic WSDL knowledge to work with the medium.

For example, you need to know WSDL if your Web service methods accept objects as parameters because you need to define the data types corresponding to those objects in the service's WSDL file. This section provides a general overview of WSDL, how it is structured, and how it operates with SOAP.

Note: *To support XMLDSIG in an WSDL context, the e*Way has to implement Multipurpose Internet Mail Extensions (MIME) binding as defined in the WSDL specifications. For complete WSDL specifications, see the following Web site:*

http://www.w3.org/TR/wsdl

6.5.1 Basic WSDL Operation

To best use WSDL, you need to understand XML namespaces and the basics of XML Schema Definition Language. For more information on these topics, see the following Web sites:

- **XML Namespaces:**

  http://www.devxpert.com/tutors/xmlns/xmlns.asp

- **XML Schema Definition Language:**

  http://www.devxpert.com/tutors/xsd/xsdvb.asp

The rest of this section explains the elements of WSDL in more detail.

6.5.2 Defining Services

Because a sample e*Gate schema with this e*Way uses a weather service, this section uses a class is called **Weather** for an example of using WSDL. This sample class has one method (for now) called **GetTemperature**. The following example shows how this class/method looks in WSDL:

```
Public Function GetTemperature(ByVal zipcode As String, _
                            ByVal celsius As Boolean) As Single
    'just sends a harcoded value for now
    If celsius Then
        GetTemperature = 21.7
    Else
        GetTemperature = 71.06
    End If
End Function
```

You can think of the class as the Web service and the **GetTemperature** method as an operation on that service. To describe this service, use the WSDL **<service>** element. All WSDL elements belong to the WSDL namespace, which is defined as:

```
http://schemas.xmlsoap.org/wsdl/
```

As an example, consider a service for the previous example, called **weatherservice**. You can define this service using WSDL as follows:

```
<definitions name ='weatherservice'
    xmlns='http://schemas.xmlsoap.org/wsdl/'>
    <service name='WeatherService' >
......
</service>
   </definitions>
```

Note: *See* **"Generic SOAP Schema" on page 85** *for an explanation of the schema sample that uses a weather service. The section also explains the schema's implementation.*

The Element <definitions>

The **<definitions>** element is the root element of a WSDL document. This element declares the WSDL namespace as the default namespace for the document. Therefore, all elements belong to this namespace unless they have another namespace prefix (for clarity, all other namespace declarations in this example are omitted).

Each service is defined using a service element. Inside the service element, you specify the different ports on which this service is accessible. A port specifies the service address, for example:

```
http://localhost/demos/wsdl/devxpert/weatherservice.asp
```

The port definition in the example **Weather** could appear as follows:

```
<port name='WeatherSoapPort' binding='wsdlns:WeatherSoapBinding' >
    <soap:address
          location='http://localhost/demos/wsdl/devxpert/
weatherservice.asp' />
</port>
```

Ports and Port Names

Each port has a unique name and a binding attribute. When using SOAP, the port element contains a **<soap:address/>** element with the actual service address. Here, the SOAP namespace prefix refers to the following namespace:

```
http://schemas.xmlsoap.org/wsdl/soap/
```

This namespace is used for SOAP-specific elements within WSDL. Such elements are also known as WSDL SOAP extension elements.

Note: *There are more examples of WSDL extension elements given throughout this section.*

A Web service does not have to be exposed using SOAP. For example, if your Web service is exposed via HTTP GET, the port element contains an **<http:address/>** element similar to:

```
<http:address location="http://localhost/demos/wsdl/devxpert
    weatherGET.asp"/>
```

A Web service can be accessible on many ports. For example, you could make your service available via SOAP and HTTP GET and possibly even via SMTP. For this Web service, you could have three ports, each one with a different name.

Note: *The current version of the SOAP e*Way only supports exposure of the Web service via SOAP.*

6.5.3 Defining Request and Response Messages

A message is protocol independent, that is, a message can be used with SOAP, HTTP GET, or any other protocol. To use Web services in a remote procedure call (RPC) model, you must describe the following messages:

- **Input** or request message, which is sent from the client to the service
- **Output** or response message, which is sent back the opposite way

Message as Payload

When you are using SOAP, keep in mind that the word *message* here refers to the payload of the SOAP request or response. That is, the message does not include the SOAP envelope, headers, or fault. The WSDL specification does not specify a naming convention for messages. You can call the messages whatever you like using their name attribute. You can use a tool to generate the WSDL and that tool probably follows its own naming convention for messages.

To describe the message structures, use the WSDL **<message>** element. Each **<message>** contains zero or more **<part>** elements. A **<part>** corresponds to a parameter or a return value in the RPC call.

The request message contains all **ByVal** and **ByRef** parameters, and the response message contains all **ByRef** parameters, as well as the return value, if the service returns something (that is, if it is a Function not a Sub). Each **<part>** must have the same name and data type as the parameter it represents (this naming rule is part of the SOAP specification and not WSDL).

Using Two Messages

The **GetTemperature** method in the previous example corresponds to two messages: a request message sent from client to server and a response message sent back to the client. You can see these messages in the following example:

```
<message name='Weather.GetTemperature'>
    <part name='zipcode' type='xsd:string'/>
    <part name='celsius' type='xsd:boolean'/>
  </message>
  <message name='Weather.GetTemperatureResponse'>
    <part name='Result' type='xsd:float'/>
  </message>
```

Note: *The data types are prefixed with the* ***.xsd*** *(XML Schema) namespace prefix, assuming it was declared earlier in the document. XML Schemas define many data types that you can draw from, when defining the message parts.*

Messages and Data Types

Table 11 lists the supported XML data types and their mapping to typical programming data types (with comments for additional information).

Table 11 XSD Types Mapped to Typical Data Types

XML (SOAP) Type	Data Type	Comments
anyURI	String	
base64Binary	Byte()	
boolean	Boolean	
byte	Integer	Range validated on conversion.
date	Date	Time set to 00:00:00.
dateTime	Date	
double	Double	
duration	String	No validation or conversion performed.
ENTITIES	String	No validation or conversion performed.
ENTITY	String	No validation or conversion performed.
float	Single	
gDay	String	No validation or conversion performed.
gMonth	String	No validation or conversion performed.
gMonthDay	String	No validation or conversion performed.

Table 11 XSD Types Mapped to Typical Data Types (Continued)

XML (SOAP) Type	Data Type	Comments
gYear	String	No validation or conversion performed.
gYearMonth	String	No validation or conversion performed.
ID	String	No validation or conversion performed.
IDREF	String	No validation or conversion performed.
IDREFS	String	No validation or conversion performed.
int	Long	
integer	Variant	Range validated on conversion.
language	String	No validation or conversion performed.
long	Variant	Range validated on conversion.
Name	String	No validation or conversion performed.
NCName	String	No validation or conversion performed.
negativeInteger	Variant	Range validated on conversion.
NMTOKEN	String	No validation or conversion performed.
NMTOKENS	String	No validation or conversion performed.
nonNegativeInteger	Variant	Range validated on conversion.
nonPositiveInteger	Variant	Range validated on conversion.
normalizedString	String	
NOTATION	String	No validation or conversion performed.
number	Variant	
positiveInteger	Variant	Range validated on conversion.
QName	String	No validation or conversion performed.
short	Integer	
string	String	
time	Date	Day set to December 30, 1899.
token	String	No validation or conversion performed.
unsignedByte	Byte	
unsignedInt	Variant	Range validated on conversion.
unsignedLong	Variant	Range validated on conversion.
unsignedShort	Long	Range validated on conversion.

The previous list of XML data types is sufficient for all your simple data-type needs. However, if your service uses user-defined data types, you need to define those types in WSDL yourself. You can define these types using XML, just as you would when creating a regular XML Schema.

6.5.4 WSDL Message Transmission

WSDL has the following message transmission primitives that an endpoint (server) can support:

- **One-way:** The endpoint (server) receives a message
- **Request-response:** The endpoint (server) receives a message and sends a correlated message
- **Solicit-response:** The endpoint (server) sends a message and receives a correlated message.
- **Notification:** The endpoint (server) sends a message.

WSDL refers to these primitives as operations. The type of operation is called the mode of operation, for example, one-way mode or notification mode of operation. Although request-response or solicit-response can be modeled abstractly using two one-way messages, it is useful to model these operations as separate primitive operation types.

Chapter 7

e*Way Java Methods

This chapter provides an overview of the Java classes and methods contained in the e*Way Intelligent Adapter for SOAP, which are used to extend the functionality of the e*Way.

7.1 SOAP e*Way Methods and Classes: Overview

For any e*Way, communication takes place both on the e*Gate Integrator system and the external system side. Communication between the e*Way and the e*Gate environment is common to all e*Ways, while the communication between the e*Way and the external system is different for each e*Way.

For the SOAP e*Way, the **stceway.exe** file (creates a Multi-Mode e*Way; see **Chapter 3**) is used to communicate between the e*Way and e*Gate. A Java Collaboration is utilized to keep the communication open between the e*Way and the external system or network.

7.2 Using Java Methods

Java methods have been added to make it easier to set information in the SOAP e*Way Event Type Definitions (ETDs), as well as get information from them. The nature of this data transfer depends on the configuration parameters (see **Chapter 5**) you set for the e*Way in the e*Gate Schema Designer's e*Way Configuration Editor window.

Note: *For more information on the* SOAP *e*Way ETD structures, their nodes, and attributes (within an e*Gate* **.xsc** *file), see* **Chapter 4**.

The Schema Designer's Collaboration Rules Editor window allows you to call Java methods by dragging and dropping an ETD node into the **Rules** scroll box of the **Rules Properties** window.

Note: *The node name can be different from the Java method name.*

After you drag and drop, the actual conversion takes place in the **.xsc** file. To view the **.xsc** file, use the Schema Designer's ETD Editor or Collaboration Rules Editor windows.

For example, if the node name is **Key**, the associated **javaName** is **Key**. If you want to get the node value, use the Java method called **getKey()**. If you want to set the node value, use the Java method called **setKey()**.

7.3 Attribute Class

The **Attribute** class represents an attribute of a SOAP ETD and is used to generate attributes of a SOAP header, SOAP body, or SOAP envelope node.

The **Attribute** class is defined as:

```
public class Attribute
```

The **Attribute** class extends **java.lang.Object.**

The **Attribute** class methods include:

- **getKey** on page 125
- **getValue** on page 125
- **setKey** on page 126
- **setValue** on page 126

getKey

Description

getKey gets the key to this attribute.

Syntax

```
public java.lang.String getKey()
```

Parameters

None.

Returns

java.lang.String
The key to this attribute; it can be null.

Throws

None.

getValue

Description

getValue retrieves the value of this attribute.

Syntax

```
public java.lang.String getValue()
```

Parameters

None.

Returns

java.lang.String
The value of this attribute; it can be null.

Throws

None.

setKey

Description

setKey sets the key to this attribute. **_key** must be a qualified name.

Syntax

```
public void setKey(java.lang.String _key)
```

Parameters

Name	Type	Description
_key	java.lang.String	The key to this attribute; it can be null.

Returns

Void.

Throws

java.lang.Exception when any generic error occurs.

setValue

Description

setValue sets the value to this attribute.

Syntax

```
public void setValue(java.lang.String _value)
```

Parameters

Name	Type	Description
_value	java.lang.String	The key to this attribute; it can be null.

Returns

Void.

Throws

None.

7.4 Input_XXX Class

The **Input_XXX** class is a generic template class for all input messages in an ETD generated from an original Web Services Description Language (WSDL) file. This class corresponds to an **Input** node under the **Operation** node.

Note: *XXX stands for the name given in the original WSDL file. For example, **Input_XXX** corresponds to **Input_BabelFish** where **BabelFish** is the name of the input message given in the WSDL file.*

The **Input_XXX** class is defined as:

```
public class Input_XXX
```

The **Input_XXX** class extends **com.stc.jcsre.SimpleETDImpl**.

The **Input_XXX** class methods include:

- **available** on page 127
- **marshal** on page 128
- **unmarshal** on page 128

Note: *For details on how the SOAP e*Way's WSDL ETD operates, see* **"WSDL ETD Operation" on page 71**. *For more information on WSDL and its use, see* **"Using the Web Services Description Language" on page 117**.

available

Description

available tests to see if the current message is available to be marshaled. When an ETD receives messages, the top-level (root node's) **unmarshal** method is called. This method delegates the unmarshaling to the **unmarshal** method in the appropriate individual input or output messages.

The **available** method determines whether a message has been unmarshaled. This method overrides **available** in the class **com.stc.jcsre.SimpleETDImpl**.

Syntax

```
public boolean available()
```

Parameters

None.

Returns

Boolean
true if the message is available, and **false** if it is not.

Throws

None.

marshal

Description

marshal marshals the message into a byte array. This method overrides the method marshal in the class **com.stc.jcsre.SimpleETDImpl.**

Syntax

```
public byte[] marshal()
```

Parameters

None.

Returns

byte[]
The message as a byte array.

Throws

com.stc.eways.WebServices.docs.MarshalException if it is unable to marshal the object.

unmarshal

Description

unmarshal unmarshals the byte array into the **message** node This method overrides the method **unmarshal** in the class **com.stc.jcsre.SimpleETDImpl.**

Syntax

```
public void unmarshal(byte[] _blob)
```

Parameters

Value	Type	Description
_blob	Byte array	The byte array.

Returns

Void.

Throws

com.stc.eways.WebServices.docs.UnmarshalException if it is unable to unmarshal the object.

7.5 InvokeOptions Class (Non-WSDL)

The non-WSDL **InvokeOptions** class encompasses all the information and methods needed to send or transport a SOAP request to a SOAP server, when you are *not* using WSDL. The class provides these options for the non-WSDL **invoke** method.

The non-WSDL **InvokeOptions** class is defined as:

```
public class InvokeOptions
```

The non-WSDL **InvokeOptions** class extends **java.lang.Object.**

The non-WSDL **InvokeOptions** class methods include:

- **getStatusCode** on page 129
- **getStatusMessage** on page 129
- **invoke** on page 178

getStatusCode

Description

getStatusCode retrieves the result of an HTTP post to the SOAP server.

Syntax

```
public int getStatusCode()
```

Parameters

None.

Returns

Integer
The status code for the last call to **SendToSOAPServer**.

Throws

java.lang.Exception when it is unable to retrieve the code.

getStatusMessage

Description

getStatusMessage retrieves any error messages from the last call to **SendToSOAPServer**.

Syntax

```
public java.lang.String getStatusMessage()
```

Parameters

None.

Returns

java.lang.String
The error message, if there is one.

Throws

java.lang.Exception when it is unable to retrieve the error message.

7.6 InvokeOptions Class (WSDL)

The **InvokeOptions** (WSDL ETD) class provides the following features for the WSDL **invoke** method:

- **URL**: determines the endpoint for the service. By setting this feature, the ETD overrides the default URL for the service defined in the original WSDL file.
- **Status Code**: holds information from the last call to invoke (initiate). The Status Code is transport-specific. For HTTP, it contains a valid HTTP return code such as 404, 200, or 500. The Status Message option holds more detailed information about the last call to invoke.
- **Status Message**: like Status Code, is transport-specific. For HTTP, the Status Message contains valid HTTP messages such as "File Not Found," "OK," and "Internal Server Error."

The WSDL **invoke** options appear only in certain operations of generated ETDs. For example, a client ETD that has an operation that works in one-way or request-response mode has an **invoke** method, and by extension, a **Retrieves** node.

A server ETD that has operations that work in solicit-response or notification mode also has the **invoke** method and the **Retrieves** node. In all other cases, **Retrieves** nodes do not appear, as it is not necessary for any other operation to have both an **invoke** method and **invoke** options.

Note: *See* **"WSDL ETD Operation" on page 71** *for a complete explanation of operation and e*Way modes, including an explanatory matrix in* **Table 6 on page 75**.

The WSDL **InvokeOptions** class is defined as:

```
public class InvokeOptions
```

The WSDL **InvokeOptions** class extends **java.lang.Object**.

The WSDL **InvokeOptions** class methods include:

- **getStatusCode** on page 131
- **getStatusMessage** on page 131
- **getURL** on page 131
- **setStatusCode** on page 132

- **setStatusMessage** on page 132
- **setURL** on page 133

getStatusCode

Description

getStatusCode retrieves the status code as a string. You must interpret the string in a transport-specific way. For example, an HTTP status code is a simple integer (for example, 200, 404, or 500). How the status code is used and/or interpreted is up to the user of this method.

Syntax

```
public java.lang.String getStatusCode()
```

Parameters

None.

Returns

java.lang.String
The status code.

Throws

None.

getStatusMessage

Description

getStatusMessage retrieves the status message as a string.

Syntax

```
public java.lang.String getStatusMessage()
```

Parameters

None.

Returns

java.lang.String
The status message.

Throws

None.

getURL

Description

getURL retrieves the URL as a string.

Syntax

```
public java.lang.String getURL()
```

Parameters

None.

Returns

java.lang.String
The URL string.

Throws

None.

setStatusCode

Description

setStatusCode sets the status code. This operation is usually done by the **invoke** method.

Syntax

```
public void setStatusCode(java.lang.String _statusCode)
```

Parameters

Value	Type	Description
_statusCode	java.lang.String	The transport-specific status code.

Returns

Void.

Throws

None.

setStatusMessage

Description

setStatusMessage sets the status message. This is usually done by the **invoke** method.

Syntax

```
public void setStatusMessage(java.lang.String _statusMessage)
```

Parameters

Value	Type	Description
_statusMessage	java.lang.String	Status message name.

Returns

Void.

Throws

None.

setURL

Description

setURL sets the URL

Syntax

```
public void setURL(java.lang.String _url)
```

Parameters

Value	Type	Description
_url	java.lang.String	The URL to set.

Returns

Void.

Throws

java.net.MalformedURLException if it is **_url** is not a valid URL.

7.7 MarshalOptions Class

The **MarshalOptions** (WSDL ETD) class provides **marshal** options for marshaling input and output messages. These options are transport-based and give information not defined by the World Wide Web Consortium (W3C) WSDL Specification.

Note: *See* **"Using the Web Services Description Language" on page 117** *for details on these specifications plus additional information on WSDL.*

The **SOAPActionURI** option overrides the **soapAction** attribute of the **soap:operation** element in the WSDL file. The **HTTP Headers** option allows you to define your own specific HTTP header when sending out a SOAP message.

The **MarshalOptions** class is defined as:

```
public class MarshalOptions
```

The **MarshalOptions** class extends **java.lang.Object**.

The **MarshalOptions** class methods include:

- **countHTTP_Headers** on page 134
- **getHTTP_Headers** on page 134
- **getSOAPActionURI** on page 135
- **setHTTP_Headers** on page 135
- **setSOAPActionURI** on page 136

countHTTP_Headers

Description

countHTTP_Headers retrieves the number of **HTTP_Headers** objects that are available.

Syntax

```
public int countHTTP_Headers()
```

Parameters

None.

Returns

Integer
The number of **HTTP_Headers** objects.

Throws

None.

getHTTP_Headers

Description

getHTTP_Headers retrieves an **HTTP_Headers** object. If one does not exist, a new one is created at the given index.

Syntax

```
public MarshalOptions.HTTP_Headers getHTTP_Headers(int _index)
```

Parameters

Value	Type	Description
_index	Integer	The index into the array for the HTTP_Headers object.

Returns

Object
An **HTTP_Headers** object.

Throws

None.

getSOAPActionURI

Description

getSOAPActionURI retrieves the **SOAPActionURI** as a string.

Syntax

```
public java.lang.String getSOAPActionURI()
```

Parameters

None.

Returns

java.lang.String
The **SOAPAction** URI.

Throws

None.

setHTTP_Headers

Description

setHTTP_Headers sets the **HTTP_Headers** object at the given index. This object replaces any other **HTTP_Headers** object at the given index. If **_header** is null, a new, empty **HTTP_Headers** object is placed at the given index.

Syntax

```
public void setHTTP_Headers(int _index,
MarshalOptions.HTTP_Headers _header)
```

Parameters

Value	Type	Description
_index	Integer	The index number.
_header	Object	The HTTP_Headers object.

Returns

Void.

Throws

None.

setSOAPActionURI

Description

setSOAPActionURI sets the **SOAPActionURI**. You must ensure that the **SOAPActionURI** is a valid URI as defined by RFC 2396.

Syntax

```
public void setSOAPActionURI(java.lang.String _soapActionURI)
```

Parameters

Value	Type	Description
_soapActionURI	java.lang.String	The URI.

Returns

Void.

Throws

None.

7.8 Operation_XXX Class

The **Operation_XXX** (WSDL ETD) class is a generic template class for all WSDL operations. This class represents an **Operation** node in the ETD and contains all the methods and data described by the original WSDL file, for a given WSDL operation.

Note: *XXX stands for the name given in the original WSDL file. For example,* ***Operation_XXX*** *corresponds to* ***Operation_BabelFish*** *where* ***BabelFish*** *is the name of the operation given in the WSDL file.*

Certain methods may or may not appear in the generated WSDL node depending on:

- The node's WSDL operation mode: one-way, request-response, solicit-response, or notification
- The ETD's e*Way mode: client or server

All possible methods are listed in this section, for completeness, but depending on the contents of a given original WSDL file, not all methods are available at all times.

WSDL ETD nodes contain or do not contain any of these methods, depending on the following factors:

- For one-way operations, there is only an **Input_XXX** message, and thus only **get** and **set** methods for that **Input_XXX** message.
- For notification operations, there is only an **Output_XXX** message, and thus only **get** and **set** methods for that **Output_XXX** messages.
- Both request-response and solicit-response operations have **Input_XXX** and **Output_XXX** messages.
- All operations contained in the WSDL ETD have **marshal** options, and therefore have a **getMarshalOptions** and **setMarshalOptions** method.

The **invoke**, **getInvokeOptions**, and **setInvokeOptions** methods only appear in certain WSDL operations as follows:

- For one-way and request-response operations that are part of a client ETD, these methods do appear. This usage occurs because the client initiates (invokes) these types of operations by calling the **invoke** method.
- For notification and solicit-response operations that are part of a server ETD, these methods also appear. Again, this usage occurs because the server initiates (invokes) the operation.

Note: See **"WSDL ETD Operation" on page 71** *for a complete explanation of operation and e*Way modes, including an explanatory matrix in* **Table 6 on page 75**.

The **Operation_XXX** class is defined as:

```
public class Operation_XXX
```

The **Operation_XXX** class extends **java.lang.Object**.

The **Operation_XXX** class methods include:

- **getInput_XXX** on page 138
- **getInvokeOptions** on page 138
- **getMarshalOptions** on page 138
- **getOutput_XXX** on page 139
- **invoke** on page 139
- **setInput_XXX** on page 140
- **setInvokeOptions** on page 140
- **setMarshalOptions** on page 141
- **setOutput_XXX** on page 141

getInput_XXX

Description

getInput_XXX is a generic **get** method for an **Input_XXX** object.

Syntax

```
public Input_XXX getInput_XXX()
```

Parameters

None.

Returns

Object
The **Input_XXX** object.

Throws

None.

getInvokeOptions

Description

getInvokeOptions retrieves the **invoke** options.

Syntax

```
public InvokeOptions getInvokeOptions()
```

Parameters

None.

Returns

Object
The **invoke** options object.

Throws

None.

Note: *See* **"InvokeOptions Class (WSDL)" on page 130** *for more information.*

getMarshalOptions

Description

getMarshalOptions retrieves the **marshal** options for the current operation.

Syntax

```
public MarshalOptions getMarshalOptions()
```

Parameters

None.

Returns

Object
The **marshal** options object.

Throws

None.

getOutput_XXX

Description

getOutput_XXX is a generic **get** method for an **Output_XXX** object.

Syntax

```
public Output_XXX getOutput_XXX()
```

Parameters

None.

Returns

Object
The **Output_XXX** object.

Throws

None.

invoke

Description

invoke initiates the current WSDL operation. This method marshals the correct **Message** node (**Input** or **Output**) based on the binding information defined in the e*Way Connection configuration. The method then sends the message to the recipient and waits for a response.

After receiving the response, the method unbinds the message then unmarshals it to the correct message node (**Input** or **Output**). The correct **Message** node is determined by the WSDL operation mode and the ETD's e*Way mode.

See **"Operation_XXX Class" on page 136** for more information.

Syntax

```
public boolean invoke()
```

Parameters

None.

Returns

Boolean

true if invoking this operation succeeds, and **false** if it fails.

Throws

java.lang.Exception if any error occurs.

setInput_XXX

Description

setInput_XXX is a generic **set** method for a **Input_XXX** object.

Syntax

```
public void setInput_XXX(Input_XXX _XXX)
```

Parameters

Value	Type	Description
_XXX	Object	The Input_XXX object.

Returns

Void.

Throws

None.

setInvokeOptions

Description

setInvokeOptions sets the **invoke** options.

Syntax

```
public void setInvokeOptions(InvokeOptions _invoke)
```

Parameters

Value	Type	Description
_invoke	Object	The invoke options object.

Returns

Void.

Throws

None.

Note: *See* **"InvokeOptions Class (WSDL)" on page 130** *for more information.*

setMarshalOptions

Description

setMarshalOptions sets the **marshal** options object.

Syntax

```
public void setMarshalOptions(MarshalOptions _options)
```

Parameters

Value	Type	Description
_options	Object	The marshal options object.

Returns

Void.

Throws

None.

setOutput_XXX

Description

setOutput_XXX is a generic **set** method for an **Output_XXX** object.

Syntax

```
public void setOutput_XXX(Output_XXX _XXX)
```

Parameters

Value	Type	Description
_XXX	Object	The Output_XXX object.

Returns

Void.

Throws

None.

7.9 Output_XXX Class

The **Output_XXX** (WSDL ETD) class is a generic template class for all output messages. This class corresponds to an **Output** node under the **Operation** (WSDL operation) node.

Note: *XXX stands for the name given in the original WSDL file. For example, **Output_XXX** corresponds to **Output_BabelFish** where **BabelFish** is the name of the output message given in the WSDL file.*

The **Output_XXX** class is defined as:

```
public class Output_XXX
```

The **Output_XXX** class extends **com.stc.jcsre.SimpleETDImpl**.

The **Output_XXX** class methods include:

- **available** on page 142
- **marshal** on page 143
- **unmarshal** on page 143

available

Description

available tests to find out whether the current message is available to be marshaled. When an ETD receives messages, the top-level (root node's) **unmarshal** method is called. This method delegates the unmarshaling to the **unmarshal** method in the individual input or output messages.

The **available** method determines whether a message has been unmarshaled. This method overrides the method **available** in the class **com.stc.jcsre.SimpleETDImpl.**

Syntax

```
public boolean available()
```

Parameters

None.

Returns

Boolean
true if the message is available, and **false** if it is not.

Throws

None.

marshal

Description

marshal marshals the message into a byte array. This method overrides the method **marshal** in the class **com.stc.jcsre.SimpleETDImpl.**

Syntax

```
public byte[] marshal()
```

Parameters

None.

Returns

byte[]
The message as a byte array.

Throws

com.stc.eways.WebServices.docs.MarshalException if it is unable to marshal the object.

unmarshal

Description

unmarshal unmarshals the byte array into the **message** node. This method overrides the method **unmarshal** in the class **com.stc.jcsre.SimpleETDImpl.**

Syntax

```
public void unmarshal(byte[] _blob)
```

Parameters

Value	Type	Description
_blob	Byte array	A byte representation of the message content.

Returns

Void.

Throws

com.stc.eways.WebServices.docs.UnmarshalException if it is unable to unmarshal the object.

7.10 PortType_XXX Class

The **PortType_XXX** (WSDL ETD) class is a generic template class for all PortType nodes. This class represents a WSDL PortType, contains **Operation** objects (representing WSDL operations), and is contained by the top-level (root) ETD node. There are **get** and **set** methods for each WSDL operation this class contains.

Note: *XXX stands for the name given in the original WSDL file. For example, **PortType_XXX** corresponds to **PortType_BabelFish** where **BabelFish** is the name of the PortType given in the WSDL file.*

The **PortType_XXX** class is defined as:

```
public class PortType_XXX
```

The **PortType_XXX** class extends **java.lang.Object**.

The **PortType_XXX** class methods include:

- getOperation_XXX on page 144
- setOperation_XXX on page 144

getOperation_XXX

Description

getOperation_XXX is a generic **get** method for a WSDL operation object. There is one such **get** method for every operation the current PortType contains.

Syntax

```
public Operation_XXX getOperation_XXX()
```

Parameters

None.

Returns

Object
The **Operation** object.

Throws

None.

setOperation_XXX

Description

setOperation_XXX is a generic **set** method for a WSDL operation object. There is one such **set** method for every WSDL operation the current PortType contains.

Syntax

```
public void setOperation_XXX(Operation_XXX _xxx)
```

Parameters

Value	Type	Description
_xxx	Object	The WSDL operation object.

Returns

Void.

Throws

None.

7.11 Service_XXX Class

The **Service_XXX** (WSDL ETD) class is a generic template class for the top-level (root) ETD node. This class represents a service in the original WSDL file and contains one or more PortType objects. There are **get** and **set** methods for each WSDL PortType this class contains.

Note: *XXX stands for the name of the top-level node. For example, **Service_XXX** corresponds to **Service_BabelFish** where **BabelFish** is the name of the top-level node.*

The **Service_XXX** class is defined as:

```
public class Service_XXX
```

The **Service_XXX** class extends **com.stc.jcsre.SimpleETDImpl**.

The **Service_XXX** class methods include:

- **getPortType_XXX** on page 145
- **reset** on page 146
- **setPortType_XXX** on page 146
- **unmarshal** on page 147

getPortType_XXX

Description

getPortType_XXX is a generic **get** method for a PortType object. There is one such **get** method for every PortType the current WSDL service contains.

Syntax

```
public PortType_XXX getPortType_XXX()
```

Parameters

None.

Returns

Object
The PortType object.

Throws

None.

reset

Description

reset resets the ETD. This method overrides the method **reset** in the class **com.stc.jcsre.SimpleETDImpl**

Syntax

```
public boolean reset()
```

Parameters

None.

Returns

Void.

Throws

None.

setPortType_XXX

Description

setPortType_XXX is a generic set method for a PortType object. There is one such **set** method for every PortType the current WSDL service contains.

Syntax

```
public void setPortType_XXX(PortType_XXX _xxx)
```

Parameters

Value	Type	Description
_xxx	Object	The PortType object.

Returns

Void.

Throws

None.

unmarshal

Description

unmarshal unmarshals a valid WSDL message based on the given parameters. This method basically delegates unmarshaling message to the **Input** or **Output** message node defined by **_portType** and **_operation**.

This method operates as follows:

- With an ETD in the server e*Way mode and WSDL operations that are one-way or request-response:
  - This **unmarshal** method delegates the unmarshaling of **_blob** to the node under the given **_portType** and **_operation**.
- With an ETD in the client e*Way mode and WSDL operations that are solicit-response or notification:
  - This **unmarshal** method delegates the unmarshaling of **_blob** to the **Output_XXX** node under the given **_portType** and **_operation**.

Note: *See* **"WSDL ETD Operation" on page 71** *for a complete explanation of operation and e*Way modes, including an explanatory matrix in* **Table 6 on page 75**.

Syntax

```
public void unmarshal(java.lang.String _soapActionURI,
    java.lang.String _portType,java.lang.String _operation,
    byte[] _blob)
```

Parameters

Value	Type	Description
_soapActionURI	java.lang.String	The name of the soapAction URI.
_portType	java.lang.String	The name of the WSDL PortType.
_operation	java.lang.String	The name of the WSDL operation.
_blob	Byte array	A byte representation of the message content.

Returns

Void.

Throws

com.stc.eways.WebServices.docs.UnmarshalException if any error occurs.

7.12 SOAP Class

Implementation of a SOAP ETD. This ETD allows for processing of SOAP messages. The SOAP ETD unmarshals valid Extensible Markup Language (XML) documents to its **SOAPRequest** node. A typical client use of this ETD would be to unmarshal a well-formed SOAP message (to the **SOAPRequest** node), call the **sendToSOAPServer** method, which unmarshals the response into the **SOAPResponse** node.

Another client use of the ETD could be to fill in key fields of the **SOAPRequest** node, then call the **sendToSOAPServer** method, which unmarshals the response into the **SOAPResponse** node.

A SOAP ETD can also be used to unmarshal a SOAP request and form a SOAP response, enabling implementation of SOAP services within a Collaboration.

The **SOAP** class is defined as:

```
public class SOAP
```

The **SOAP** class extends **com.stc.jcsre.MsgETDImpl.**

The **SOAP** class methods include:

getSOAPActionURI

Description

getSOAPActionURI gets the **SOAPAction** header used to send the request message.

Syntax

```
public java.lang.String getSOAPActionURI()
```

Parameters

None.

Returns

java.lang.String
The **SOAPAction** URI.

Throws

None.

getSOAPRequest

Description

getSOAPRequest gets the **SOAPRequest** object.

Syntax

```
public SOAPRequest getSOAPRequest()
```

Parameters

None.

Returns

Object
The **SOAPRequest** object.

Throws

None.

getSOAPResponse

Description

getSOAPResponse retrieves the **SOAPResponse** object.

Syntax

```
public SOAPResponse getSOAPResponse()
```

Parameters

None.

Returns

Object
The **SOAPResponse** object.

Throws

None.

getSOAPTransport

Description

getSOAPTransport retrieves the SOAPTransport object.

Syntax

```
public SOAPResponse getSOAPTransport()
```

Parameters

None.

Returns

Object
The **SOAPTransport** object.

Throws

None.

getURL

Description

getURL retrieves the URL to which the request message is sent.

Syntax

```
public java.lang.String getURL()
```

Parameters

None.

Returns

java.lang.String
The URL to which the request message is sent.

Throws

None.

marshal

Description

marshal marshals the data content of the ETD into a byte array. The default behavior is to marshal the **SOAPRequest** node. It overrides **marshal** in the class **com.stc.jcsre.SimpleETDImpl**.

Syntax

```
public byte[] marshal()
```

Parameters

None.

Returns

byte[]
Byte array of the blob result from marshaling.

Throws

com.stc.jcsre.MarshalException when it is unable to marshal the ETD.

reset

Description

reset resets the data content of an ETD. It overrides **reset** in the class **com.stc.jcsre.SimpleETDImpl**.

Syntax

```
public boolean reset()
```

Parameters

None.

Returns

Boolean
true if the reset clears data content of the ETD, and **false** if the ETD does not have meaningful implementation of **reset**, necessitating the creation of a new ETD.

setSOAPActionURI

Description

setSOAPActionURI sets the **SOAPAction** URI used in sending the request message.

Syntax

```
public void setSOAPActionURI(java.lang.String _soapActionURI)
```

Parameters

Name	Type	Description
_soapActionURI	java.lang.String	The name of the SOAPAction URI to set.

Returns

Void.

Throws

None.

setSOAPRequest

Description

setSOAPRequest sets the **SOAPRequest** object.

Syntax

```
public void setSOAPRequest(SOAPRequest _request)
```

Parameters

Name	Type	Description
_request	Object	The SOAPRequest object to set.

Returns

Void.

Throws

None.

setSOAPResponse

Description

setSOAPResponse sets the **SOAPResponse** object.

Syntax

```
public void setSOAPResponse(SOAPResponse _response)
```

Parameters

Name	Type	Description
_response	Object	The SOAPResponse object to set.

Return Values

Void.

Throws

None.

setURL

Description

setURL sets the URL to which the request message is sent.

Syntax

```
public java.lang.String setURL(java.lang.String _url)
```

Parameters

Name	Type	Description
`_url`	java.lang.String	The URL to set.

Returns

java.lang.String
The URL to which the request message is sent.

Throws

java.net.MalformedURLException when **_url** is an invalid URL.

unmarshal

Description

unmarshal unmarshals a byte array into the data content of an ETD. The default behavior is to unmarshal to the **SOAPRequest** node. It overrides **unmarshal** in the class **com.stc.jcsre.SimpleETDImpl**.

Syntax

```
public void unmarshal(byte[] _blob)
```

Parameters

Name	Type	Description
_blob	Byte array	Byte array representation of the XML document to be unmarshaled.

Returns

Void.

Throws

com.stc.jcsre.UnmarshalException when it is unable to unmarshal **_blob** into the ETD.

unmarshalResponse

Description

unmarshalResponse provides a convenient method to unmarshal a byte array into the **SOAPResponse** node. The byte array must be a valid XML document.

Syntax

```
public void unmarshalResponse(byte[] _blob)
```

Parameters

Name	Type	Description
byte[]	XML	Byte array representation of the XML document to unmarshal.

Returns

Void.

Throws

com.stc.jcsre.UnmarshalException if it is unable to unmarshal the **SOAPResponse** node.

7.13 SOAPAttachment Class

The **SOAPAttachment** class is an ETD node that represents an attachment to a SOAP message.

The **SOAPAttachment** class is defined as:

```
public class SOAPAttachment
```

The **SOAPAttachment** class extends **java.lang.Object.**

The **SOAPAttachment** class methods include:

- **addReference** on page 155
- **base64Encode** on page 155
- **getContentType** on page 156
- **getFileLocation** on page 156
- **getName** on page 156
- **getTransferEncoding** on page 157
- **getValue** on page 157
- **setContentType** on page 158
- **setFileLocation** on page 158
- **setName** on page 159
- **setTransferEncoding** on page 159
- **setValue** on page 160

addReference

Description

addReference adds a reference to **_obj**. The parameter **_obj** must implement the ETD interface and have the **setHref** method. In other words, the XML element that the ETD represents must have an **Href** attribute.

Syntax

```
public void addReference(com.stc.jcsre.ETD _obj)
```

Parameters

Name	Type	Description
_obj	Object	Object implementing the ETD interface.

Returns

Void.

Throws

- **java.lang.NoSuchMethodException** if a **setHref** method does not exist
- **java.lang.IllegalAccessException** if the **setHref** method does not have sufficient access privileges.
- **java.lang.reflect.InvocationTargetException**

base64Encode

Description

base64Encode encodes the current attachment in base 64. By default, the **Content-Transfer-Encoding** is set to **base64**.

Syntax

```
public void base64Encode()
```

Parameters

None.

Returns

Void.

Throws

java.io.IOException if it is unable to encode the contents of this attachment in base 64.

getContentType

Description

getContentType retrieves the **Content-Type** for the current attachment. By default, the content type of the attachment is set to **application/octet-stream** if no content type is specified.

Syntax

```
public java.lang.String getContentType()
```

Parameters

None.

Returns

java.lang.String
The content type for the current attachment.

Throws

None.

getFileLocation

Description

getFileLocation retrieves the file name of the current attachment.

Syntax

```
public java.lang.String getFileLocation()
```

Parameters

None.

Returns

java.lang.String
The file name of the current attachment.

Throws

None.

getName

Description

getName retrieves the name of the current attachment. This name is used to generate the Content-ID and must be unique across all attachments.

Syntax

```
public java.lang.String getName()
```

Parameters

None.

Returns

java.lang.String
The attachment name.

Throws

None.

getTransferEncoding

Description

getTransferEncoding retrieves the content transfer encoding for the current attachment. By default, **Content-Transfer-Encoding** is set to **binary** if no other **Content-Transfer-Encoding** is set.

Syntax

```
public java.lang.String getTransferEncoding()
```

Parameters

None.

Returns

java.lang.String
The content transfer encoding for the current attachment.

Throws

None.

getValue

Description

getValue retrieves the value of the attachment. If the value was set previously, it returns the value. If the value was not set, but a file location was specified, it retrieves the contents of the file and returns those contents as a byte array.

Syntax

```
public byte[] getValue()
```

Parameters

None.

Returns

byte[]
The value of the current attachment or the file contents as a byte array.

Throws

None.

setContentType

Description

setContentType sets the content type for the current attachment. The possible values are defined by RFC.

Syntax

```
public void setContentType(java.lang.String _contentType)
```

Parameters

Name	Type	Description
_contentType	java.lang.String	The content type for this attachment.

Returns

Void.

Throws

None.

setFileLocation

Description

setFileLocation sets the file name of the current attachment. This method checks the following properties before setting the file name:

- The file must exist.
- The file must be a valid file and not a directory or a symbolic link.
- The file must be readable.

This method clears out any previous setting of the value field.

Syntax

```
public void setFileLocation(java.lang.String _fileLocation)
```

Parameters

Name	Type	Description
_fileLocation	java.lang.String	The name of the file.

Returns

Void.

Throws

None.

setName

Description

setName sets the name of the current attachment. This name is used to generate the Content-ID and must be unique across all attachments.

Syntax

```
public void setName(java.lang.String _name)
```

Parameters

Name	Type	Description
_name	java.lang.String	The unique name for this attachment; it cannot be null.

Returns

Void.

Throws

NullPointerException if **_name** is null.

setTransferEncoding

Description

setTransferEncoding sets the content transfer encoding to **_transferEncoding**. The possible values are defined by RFC.

Syntax

```
public void setTransferEncoding(java.lang.String _transferEncoding)
```

Parameters

Name	Type	Description
_transferEncoding	java.lang.String	The content transfer encoding for this attachment.

Returns

Void.

Throws

None.

setValue

Description

setValue sets the value of the current attachment. If a file location was previously specified, it is set to the empty string.

Syntax

```
public void setValue(byte[] _value)
```

Parameters

Name	Type	Description
_value	Byte array	The value to set.

Returns

Void.

Throws

None.

7.14 SOAPBody Class

The **SOAPBody** class represents the **SOAPBody** element of the SOAP ETD, and is used to generate a body element in a SOAP message.

The **SOAPBody** class is defined as:

```
public class SOAPBody
```

The **SOAPBody** class extends **java.lang.Object.**

The **SOAPBody** class methods include:

- **getAttribute** on page 161
- **getBodyContents** on page 161
- **getNumberOfAttributes** on page 161
- **getSOAPFault** on page 162
- **setAttribute** on page 162
- **setBodyContents** on page 163
- **setSOAPFault** on page 163

getAttribute

Description

getAttribute allows a user to access a specific attribute.

Syntax

```
public Attribute getAttribute(int _index)
```

Parameters

Name	Type	Description
_index	Integer	The location of the attribute.

Returns

Attribute
The key-value pair that is the attribute.

Throws

None.

getBodyContents

Description

getBodyContents gets the contents of the specified **SOAPBody** object.

Syntax

```
public java.lang.String getBodyContents()
```

Parameters

None.

Returns

java.lang.String
A string representing the body contents of the **SOAPBody**. The contents must be well-formed XML.

Throws

None.

getNumberOfAttributes

Description

getNumberOfAttributes returns the number of attributes for this element.

Syntax

```
public int getNumberOfAttributes()
```

Parameters

None

Returns

Integer
The number of attributes for this element.

Throws

None.

getSOAPFault

Description

getSOAPFault retrieves the SOAP fault if one is generated.

Syntax

```
public SOAPFault getSOAPFault()
```

Parameters

None.

Returns

Object
The **SOAPFault** object; it can be null if no fault was set.

Throws

None.

setAttribute

setAttribute allows you to set the index to a specific attribute.

Syntax

```
public void setAttribute(int _index, Attribute _attribute)
```

Parameters

Name	Type	Description
_index	Integer	The location of the attribute.
_attribute	Attribute	The key-value pair mapping.

Returns

Void.

Throws

None.

setBodyContents

Description

setBodyContents sets the contents of the **SOAPBody** object to the given string.

Syntax

```
public void setBodyContents(java.lang.String _bodyContents)
```

Parameters

Name	Type	Description
_bodyContents	java.lang.String	A string representing the contents of the body of a SOAP message; the message must be well-formed XML.

Returns

Void.

Throws

java.lang.Exception when any generic error occurs.

setSOAPFault

Description

setSOAPFault sets the SOAP fault to the specified object.

Syntax

```
public void setSOAPFault(SOAPFault _soapFault)
```

Parameters

Name	Type	Description
_soapFault	Object	An object representing a SOAP fault.

Returns

Void.

Throws

None.

7.15 SOAPFault Class

The **SOAPFault** class represents the **SOAPFault** node in the SOAP ETD and is used to generate fault elements of a SOAP message.

The **SOAPFault** class is defined as:

```
public class SOAPFault
```

The **SOAPFault** class extends **java.lang.Object.**

The **SOAPFault** class methods include:

- **getDetail** on page 164
- **getFaultActor** on page 164
- **getFaultCode** on page 165
- **getFaultString** on page 165
- **setDetail** on page 166
- **setFaultActor** on page 166
- **setFaultCode** on page 166
- **setFaultString** on page 167

getDetail

Description

getDetail gets the detail information for this fault if it exists.

Syntax

```
public java.lang.String getDetail()
```

Parameters

None.

Returns

java.lang.String
The detail string.

Throws

None.

getFaultActor

Description

getFaultActor gets the fault actor URI.

Syntax

```
public java.lang.String getFaultActor()
```

Parameters

None.

Returns

java.lang.String
The fault actor URI.

Throws

None.

getFaultCode

Description

getFaultcode gets the fault code.

Syntax

```
public java.lang.String getFaultCode()
```

Parameters

None.

Returns

java.lang.String
The fault code.

Throws

None.

getFaultString

Description

getFaultString gets the fault string.

Syntax

```
public java.lang.String getFaultString()
```

Parameters

None.

Returns

java.lang.String
The fault string.

Throws

None.

setDetail

Description

getDetail sets the detail information for this fault if it exists.

Syntax

```
public void getDetail(java.lang.String _detail)
```

Parameters

Name	Type	Description
_detail	java.lang.String	The detail information as well-formed XML.

Returns

Void.

Throws

java.lang.Exception when **_detail** is not well-formed XML.

setFaultActor

Description

setFaultActor sets the fault actor.

Syntax

```
public void setFaultActor(java.lang.String _actor)
```

Parameters

Name	Type	Description
_actor	java.lang.String	The fault actor.

Returns

Void.

Throws

None.

setFaultCode

Description

setFaultCode sets the fault code.

Syntax

```
public void SetFaultCode(java.lang.String _code)
```

Parameters

Name	Type	Description
_code	java.lang.String	The fault code.

Returns

Void.

Throws

None.

setFaultString

Description

setFaultString sets the fault string.

Syntax

```
public void setFaultString(java.lang.String _string)
```

Parameters

Name	Type	Description
_string	java.lang.String	The fault string.

Returns

Void.

Throws

None.

7.16 SOAPHeader Class

The **SOAPHeader** class directly represents the **SOAPHeader** node of the SOAP ETD, and is used to generate the header element of the SOAP message.

The **SOAPHeader** class is defined as:

```
public class SOAPHeader
```

The **SOAPHeader** class extends **java.lang.Object.**

The **SOAPHeader** class methods include:

- **getAttribute** on page 168
- **getHeaderContents** on page 168
- **getNumberOfAttributes** on page 169
- **setAttribute** on page 169
- **setHeaderContents** on page 169

getAttribute

Description

getAttribute allows a user to access a specific attribute.

Syntax

```
public Attribute getAttribute(int _index)
```

Parameters

Name	Type	Description
_index	Integer	The location of the attribute.

Returns

Attribute
The key-value pair that is the attribute.

Throws

None.

getHeaderContents

Description

getHeaderContents gets the contents of the **SOAPHeader** node to the specified string.

Syntax

```
public java.lang.String getHeaderContents()
```

Parameters

None.

Returns

java.lang.String
The contents of the header as a string. The contents must be well-formed XML.

Throws

None.

getNumberOfAttributes

Description

getNumberOfAttributes returns the number of attributes for this element.

Syntax

```
public int getNumberOfAttributes()
```

Parameters

None

Returns

Integer
The number of attributes for this element.

Throws

None.

setAttribute

Description

setAttribute allows a user to set the index to a specific attribute.

Syntax

```
public void setAttribute(int _index, Attribute _attribute)
```

Parameters

Name	Type	Description
_index	Integer	The location of the attribute.
_attribute	Attribute	The key-value pair mapping.

Returns

Void.

Throws

None.

setHeaderContents

Description

setHeaderContents sets the contents of the **SOAPHeader** node. The contents must be well-formed XML.

Syntax

```
public void setHeaderContents(java.lang.String _headerContents)
```

Parameters

Name	Type	Description
_headerContents	java.lang.String	The contents of the header.

Returns

Void.

Throws

java.lang.Exception when any content error occurs.

7.17 SOAPMessage Class

The **SOAPMessage** class serves as a base class for all ETD nodes that are SOAP messages. This class is used to generate a complete SOAP envelope with required body and optional header and fault elements.

The **SOAPMessage** class is defined as:

```
public class SOAPMessage
```

The **SOAPMessage** class extends **java.lang.Object.**

The **SOAPMessage** class methods include:

- **getAttribute** on page 170
- **getNumberOfAttributes** on page 171
- **getSOAPBody** on page 171
- **getSOAPHeader** on page 172
- **marshal** on page 172
- **setAttribute** on page 172
- **setSOAPBody** on page 173
- **setSOAPHeader** on page 173
- **unmarshal** on page 174

getAttribute

Description

getAttribute allows you to access a specific attribute.

Syntax

```
public Attribute getAttribute(int _index)
```

Parameters

Name	Type	Description
_index	Integer	The location of the attribute.

Returns

Attribute
The key-value pair that makes up the attribute.

Throws

None.

getNumberOfAttributes

Description

getNumberOfAttributes returns the number of attributes for this element.

Syntax

```
public int getNumberOfAttributes()
```

Parameters

None

Returns

Integer
The number of attributes for this element.

Throws

None.

getSOAPBody

Description

getSOAPBody returns an ETD-specific **SOAPBody** object.

Syntax

```
public SOAPBody getSOAPBody()
```

Parameters

None.

Returns

Object
The body of this **SOAPMessage**.

Throws

None.

getSOAPHeader

Description

getSOAPHeader returns an ETD-specific **SOAPHeader** object.

Syntax

```
public SOAPHeader getSOAPHeader()
```

Parameters

None.

Returns

Object
The header of the **SOAPMessage.**

Throws

java.lang.Exception when any generic error occurs.

marshal

Description

marshal marshals the contents of the **SOAPMessage** class into a byte array.

Syntax

```
public byte[] marshal()
```

Parameters

None.

Returns

byte[]
Byte array representation of the current message.

Throws

com.stc.jcsre.MarshalException if it is unable to marshal the contents of this object into a byte array.

setAttribute

Description

setAttribute allows you to set the index to a specific attribute.

Syntax

```
public void setAttribute(int _index, Attribute _attribute)
```

Parameters

Name	Type	Description
_index	Integer	The location of the attribute.
_attribute	Attribute	The key-value pair mapping.

Returns

Void.

Throws

None.

setSOAPBody

Description

setSOAPBody sets the ETD-specific SOAP body.

Syntax

```
public void setSOAPBody(SOAPBody _soapBody)
```

Parameters

Name	Type	Description
_soapBody	SOAPBody	A new SOAPBody.

Returns

Void.

Throws

None.

setSOAPHeader

Description

setSOAPHeader sets the ETD-specific SOAP header.

Syntax

```
public void setSOAPHeader(SOAPHeader _soapHeader)
```

Parameters

Name	Type	Description
_soapHeader	SOAPHeader	A new SOAP header.

Returns

Void.

Throws

None.

unmarshal

Description

unmarshal unmarshals the byte array **_blob** into the internal structure of this **SOAPMessage** class. It is assumed that **_blob** is a valid, well-formed XML document conforming to the SOAP specification.

Syntax

```
public void unmarshal(byte[] _blob)
```

Parameters

Name	Type	Description
_blob	Byte array	The byte array representing a SOAP message.
SOAPAction URI	java.lang.String	The identifier from which the server is told what action to take.
Transport Type	java.lang.String	The transport mechanism, for example, HTTP, SMTP, or HTTP(S). Currently, HTTP is the only transport mechanism supported.

Returns

Void.

Throws

com.stc.jcsre.UnmarshalException if it is unable to interpret **_blob** into the internal class attributes.

7.18 SOAPNode Class

The **SOAPNode** class is a base class for all SOAP-specific nodes in an ETD. This class implements the ETD interface and extends **XMLETDImpl**. This class represents a generic, SOAP element from a SOAP XML document. For example, the **SOAPBody** class that extends from **SOAPNode** represents the element **SOAP-ENV:Body**.

SOAPNode is responsible for dealing with attributes and namespaces.

The **SOAPNode** class is defined as:

```
public class SOAPNode
```

The **SOAPNode** class extends **com.stc.jcsre.XMLETDImpl.**

The **SOAPNode** class methods include:

- **countAttribute** on page 175
- **getAttribute** on page 175
- **getLocalName** on page 176
- **setAttribute** on page 176
- **unmarshal** on page 177

countAttribute

Description

countAttribute retrieves the number of attributes for this element.

Syntax

```
public int countAttribute()
```

Parameters

None.

Returns

Integer
The number of attributes for this element.

Throws

None.

getAttribute

Description

getAttribute allows a user to access a specific attribute. If no attribute exists at the given location, an empty attribute is created and returned.

Syntax

```
public Attribute getAttribute(int _index)
```

Parameters

Name	Type	Description
_index	Integer	The location of the attribute.

Returns

Attribute
The key-value pair that is the attribute.

Throws

None.

getLocalName

Description

getLocalName retrieves the local name associated with the current XML element.

Note: *Every SOAPNode class in a SOAP ETD represents an XML element.*

Syntax

```
public abstract java.lang.String getLocalName()
```

Parameters

None.

Returns

java.lang.String
The local name of the current ETD node.

Throws

None.

setAttribute

Description

setAttribute allows you to set the index to a specific attribute.

Syntax

```
public void setAttribute(int _index,Attribute _attribute)
```

Parameters

Name	Type	Description
_index	Integer	The location of the attribute.
_attribute	Attribute	The key-value pair mapping.

Returns

Void.

Throws

None.

unmarshal

Description

unmarshal unmarshals the byte array **_blob** into the current object. It overrides **unmarshal** in the class **com.stc.jcsre.XMLETDImpl**.

Syntax

```
public void unmarshal(byte[] _blob)
```

Parameters

Name	Type	Description
_blob	Byte array	The byte array to unmarshal.

Returns

Void.

Throws

com.stc.jcsre.UnmarshalException if an error occurs in interpreting the bytes.

7.19 SOAPRequest Class

The **SOAPRequest** class represents a **SOAPMessage** that is sent to a SOAP Server. Fundamentally, a **SOAPRequest** is a simple wrapper around a **SOAPMessage** object. This class represents the SOAP request of a SOAP ETD.

The **SOAPRequest** class is defined as:

```
public class SOAPRequest
```

The **SOAPRequest** class extends **SOAPMessage.**

The following **SOAPRequest** class methods were inherited from the **SOAPMessage** class, and are described under **SOAPMessage Class** on page 170:

- **getAttribute** on page 170
- **getSOAPBody** on page 171
- **getSOAPHeader** on page 172
- **invoke** on page 178
- **marshal** on page 172
- **setAttribute** on page 172
- **setSOAPBody** on page 173
- **setSOAPHeader** on page 173
- **unmarshal** on page 174

7.20 SOAPResponse Class

The **SOAPResponse** class represents a **SOAPMessage** that has been received from a SOAP server. A SOAP response has the responsibility of dealing with possible SOAP exceptions and faults in addition to its regular responsibilities as a **SOAPMessage**.

The **SOAPResponse** class is defined as:

```
public class SOAPResponse
```

The **SOAPResponse** class extends **SOAPMessage.**

The following **SOAPResponse** class methods were inherited from the **SOAPMessage** class, and are described under that section (**SOAPMessage Class** on page 170):

- **getAttribute** on page 170
- **getSOAPBody** on page 171
- **getSOAPHeader** on page 172
- **invoke** on page 178
- **marshal** on page 172
- **setAttribute** on page 172
- **setSOAPBody** on page 173
- **setSOAPHeader** on page 173
- **unmarshal** on page 179

The rest of this section explains the **SOAPResponse** class method not inherited from the **SOAPMessage** class.

invoke

Description

invoke sends the SOAP message represented by the **SOAPRequest** node.

Syntax

```
public void invoke()
```

Parameters

The configuration parameters are derived from the SOAP e*Way Connection.

Name	Type	Description
URL	java.lang.String	The location to send the message or the identifier into which the SOAP message is posted.

Name	Type	Description
SOAPAction URI	java.lang.String	The identifier from which the server is told what action to take.
Transport Type	java.lang.String	The transport mechanism, for example, HTTP, SMTP, or HTTP(S). Currently, HTTP is the only transport mechanism supported.

Returns

Void.

Throws

java.lang.Exception when any generic error occurs.

unmarshal

Description

unmarshal behaves like a normal SOAP message, but be sure to check whether there are any faults. If any fault exists, it marshals that data to the fault node. It overrides **unmarshal** in the class **SOAPMessage**.

Syntax

```
public void unmarshal(byte[] blob)
```

Parameters

Name	Type	Description
blob	Byte array	The array that holds the XML document.

Returns

Void.

Throws

com.stc.jcsre.UnmarshalException if there is a fault.

7.21 SOAPSignature Class

The **SOAPSignature** class represents a **SOAP-SEC:Signature** element. This element encapsulates an XML signature element and is part of the header of the SOAP envelope.

Users of the SOAP ETD do not have to interact directly with this class. Through the use of the sign-and-verify method calls in **SOAPSigner** and **SOAPVerification**, respectively, objects of this class are generated automatically.

The **SOAPSignature** class is defined as:

```
public class SOAPSignature
```

The **SOAPSignature** class extends **SOAPNode.**

The **SOAPSignature** class methods include:

- **getLocalName** on page 180
- **getXMLSignature** on page 180
- **setXMLSignature** on page 181

getLocalName

Description

getLocalName allows you to retrieve the local name of the current EDT node. It overrides **getLocalName** in the class **SOAPNode** (see **"SOAPNode Class" on page 174**).

Syntax

```
public java.lang.String getLocalName()
```

Parameters

None.

Returns

java.lang.String
The local name of the current node.

Throws

None.

getXMLSignature

Description

getXMLSignature retrieves the XML signature as a string. The XML signature must be well-formed XML and conform to the official XML Digital Signature Specification.

Syntax

```
public java.lang.String getXMLSignature()
```

Parameters

None.

Returns

java.lang.String
The XML signature as a string.

Throws

None.

setXMLSignature

Description

setXMLSignature sets the XML signature. The XML Signature must be well-formed XML and conform to the official XML Digital Signature Specification.

Syntax

```
public void setXMLSignature(java.lang.String _digitalSignature)
```

Parameters

Name	Type	Description
_digitalSignature	java.lang.String	The XML signature as a string.

Returns

Void.

Throws

None.

7.22 SOAPSigner Class

The **SOAPSigner** class is a utility node in the SOAP ETD. This class signs key parts of the SOAP message, generating **SOAPSignature** objects.

The **SOAPSigner** class is defined as:

```
public class SOAPSigner
```

The **SOAPSigner** class extends **java.lang.Object.**

The **SOAPSigner** class methods include:

- **getSignatureResults** on page 182
- **getSignatures** on page 182
- **setSignatureResults** on page 182
- **setSignatures** on page 183
- **sign** on page 183

getSignatureResults

Description

getSignatureResults retrieves the results of the last call to sign. Its values can be:

- **Empty string** indicating success.
- **Exception stack trace** indicating failure

Syntax

```
public java.lang.String getSignatureResults()
```

Parameters

None.

Returns

java.lang.String
The signature results.

Throws

None.

getSignatures

Description

getSignatures retrieves the list of signatures.

Syntax

```
public java.util.List getSignatures()
```

Parameters

None.

Returns

List
The list of signatures.

Throws

None.

setSignatureResults

Description

setSignatureResults sets the signature results to **_signatureResults**.

Syntax

```
public void setSignatureResults(java.lang.String _signatureResults)
```

Parameters

Name	Type	Description
_signatureResults	java.lang.String	The signature results.

Returns

Void.

Throws

None.

setSignatures

Description

setSignatures sets the list of signatures.

Syntax

```
public void setSignatures(java.util.List _signatures)
```

Parameters

Name	Type	Description
_signatures	List	The data structure used to hold signatures; it cannot be null.

Returns

Void.

Throws

None.

sign

Description

sign signs the given **_etd** object. The **_etd** object must be an XML-based node. Moreover, the ETD object must have an identification (ID) attribute or be able to generically set attributes. For example, all SOAP-based nodes allow for generic attributes. After signing a SOAP-based node (for example, **SOAPRequest**, **SOAPResponse**, or **SOAPBody**) a new **ID** attribute is added.

If the **_etd** object is generated from a DTD, then the element represented by **_etd** must have an **ID** attribute. This element has a corresponding **setID** that is used to create the proper references.

Syntax

```
public void sign(com.stc.jcsre.ETD _etd)
```

Parameters

Name	Type	Description
_etd	Object	The ETD object to be signed.

Returns

Void.

Throws

None.

7.23 SOAPVerification Class

The **SOAPVerification** class is a utility class used to verify the SOAP message. This class examines all SOAP signature elements in the SOAP header and verifies that they match, based on the digest and signature algorithms.

The XML signatures must have the **KeyInfo** element. This element gives a representation of the key needed to validate the current document.

By default, verification is not done upon unmarshaling of the SOAP message. You must call the **verify** method within the Collaboration. You must then check the call's status to see whether to accept the message.

The **SOAPVerification** class is defined as:

```
public class SOAPVerification
```

The **SOAPVerification** class extends **java.lang.Object.**

The **SOAPVerification** class methods include:

- **getVerificationResults** on page 184
- **setVerificationResults** on page 185
- **verify** on page 185

getVerificationResults

Description

getVerificationResults retrieves the verification results as a string. The verification results can be one of the following values:

- **0**: Both the signature and reference validation were successful.
- **1**: Signature validation failed but the reference validation was successful.
- **2**: Signature validation was successful but the reference validation failed.

If you receive any other value, both the signature and reference validation failed.

Note: *See the official XML Digital Signature Specification for details on signature and reference validation.*

Syntax

```
public java.lang.String getVerificationResults()
```

Parameters

None.

Returns

java.lang.String
The results of the last call to verify.

Throws

None.

setVerificationResults

Description

setVerificationResults sets the verification results to the given string.

Syntax

```
public void setVerificationResults(java.lang.String
    _verificationResults)
```

Parameters

Name	Type	Description
_verificationResults	java.lang.String	The new results from a call to verify.

Returns

Void.

Throws

None.

verify

Description

verify verifies the signatures in the SOAP header.

Syntax

```
public void verify()
```

Parameters

None.

Returns

Void.

Throws

java.lang.Exception if any error occurs.

Index