Progress®



Develop an ABL SOAP Client

OpenEdge®

Copyright

© 2020 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, Icenium, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, iMail, JustAssembly, JustDecompile, JustMock, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

October 2020

Product version: Progress OpenEdge 12.3

Updated: 2021/07/22

Table of Contents

Preface....7

ABL SOAP Clients....9

Run the WSDL Analyzer....10
Examine the generated WDSL documentation....11
Develop code from sample snippets....12

Handling SOAP Message Headers in ABL....17

SOAP message headers: an overview....18
SOAP header structure....18
SOAP header object model....19
Accessing SOAP header entries....21
Specifying SOAP header callback procedures at run time....22
Using the SET-CALLBACK-PROCEDURE( ) method....23
Defining header handlers....23
Defining a response header handler....25
Defining a request header handler....25
Invoking a header handler....26
Creating and managing SOAP message headers....27
Reusing an unchanged SOAP response header....28
Modifying a reused SOAP response header....31
Using a client-created SOAP request header....37
Managing memory for SOAP headers....42
SOAP header object model and DOM relationships....42
Memory for the SOAP header object model and DOM....42
Attributes and methods for handling SOAP headers....43
SOAP header object attributes and methods....43
SOAP header-entryref object attributes and methods....44

Handling Errors in ABL Requests to OpenEdge SOAP Web Services....47

Handling SOAP faults....48
Detecting a SOAP fault....48
SOAP faults with traditional error handling....48
SOAP faults with structured error handling....49
Managing a SOAP fault....51
Examples of ABL accessing a SOAP fault....53
Traditional error handling example....53
Structured error handling example....56
Asynchronous call error handling example....58

Debugging ABL applications that call Web services....60
SOAP Viewers....60
Using WSAViewer....61
Using ProSOAPView....62

Appendix A: WSDL Analyzer....65

Appendix B: WSDL Details....69

ABL procedure prototype to WSDL operation....69
Array mapping in WSDL documents....71
Doc/Lit....72
RPC/Literal....73
RPC/Encoded....74
Define TABLE (static temp-table) parameters....75
Define TABLE-HANDLE (dynamic temp-table) parameters....77
Define DATASET (static ProDataSet) parameters....78
Include before-image data....83
Use NAMESPACE-URI attributes....84
Define DATASET-HANDLE (dynamic ProDataSet) parameters....85

Appendix C: ABL Elements to Consume SOAP Web Services....87

Handles to consume a web service....87
Statements to consume SOAP web services....89
Attributes, methods, and events to consume web services....91

Appendix D: Data Type Conversion Rules for ABL Calls to SOAP Web Services....101

Data type casting....102
CHARACTER or LONGCHAR....108
DATE....108
DATETIME....109
DATETIME-TZ....110
DECIMAL....111
INT64....112
INTEGER....114
LOGICAL....116
MEMPTR or RAW....116

Preface

Purpose

Progress OpenEdge enables ABL clients to access SOAP web services as SOAP clients. ABL developers can use existing SOAP services seemlessly from within their applications.

Audience

This content is intended for developers who want to connect to existing SOAP services.

Organization

This content is organized into the following topics:

- ABL SOAP Clients which introduces the basic resources needed to get started with SOAP clients including how to run the WSDL Analyzer, examine the generated WSDL and develop code from sample snippets.
- Reference information is available in the following topics:
  - WSDL Analyzer
  - WSDL Details
  - ABL Elements to Consume SOAP Web Services
  - Data Type Conversion Rules for ABL Calls to SOAP Web Services

Documentation conventions

See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content library.

1

ABL SOAP Clients

Progress OpenEdge enables ABL clients to access SOAP web services as SOAP clients. ABL developers can use existing SOAP services seemlessly from within their applications.

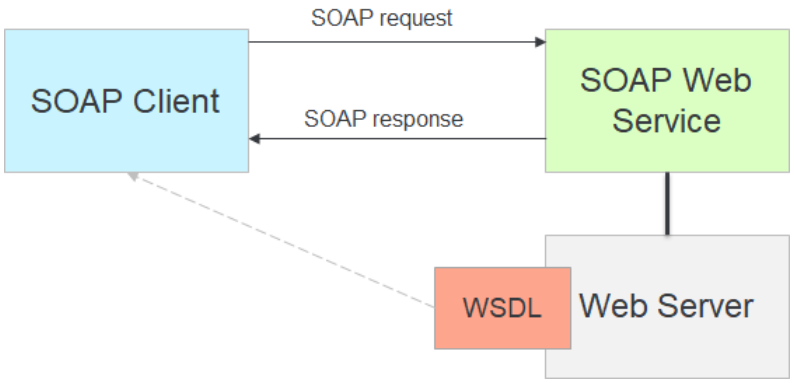
What is SOAP?

SOAP is an acronym for Simple Object Access Protocol. SOAP enables applications built on different platforms or in different programming languages to exchange messages in a standard way, primarily over HTTP.

The SOAP specification, which describes the protocol, is based on XML. It defines the structure of SOAP messages and the rules for processing a SOAP message.

A typical SOAP transaction involves the following components:

- A SOAP client application that initiates the transaction by sending a SOAP request message.
- A SOAP web service that processes the SOAP request and returns a SOAP response message. If an error occurs while processing the request, the web service returns a SOAP fault message.
- A Web Services Description Language (WSDL) document that describes the operations that are available on the SOAP web service and the request and response message schemas for each operation. SOAP client applications use the WSDL to determine which operations to invoke, generate request messages using the message schema in the WSDL, etc.



In this example, your ABL code is the SOAP client. To learn more about SOAP, see the following resources:

- SOAP specification: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/
- SOAP tutorials:
  - https://www.w3schools.com/xml/xml_soap.asp
  - https://www.tutorialspoint.com/soap/

Steps to build an ABL SOAP Client

To build an ABL SOAP client you will need to complete the following steps.

1. Run the WSDL Analyzer on page 10
2. Examine the generated WDSL documentation on page 11
3. Develop code from sample snippets on page 12

For details, see the following topics:

- Run the WSDL Analyzer
- Examine the generated WDSL documentation
- Develop code from sample snippets

Run the WSDL Analyzer

To develop ABL SOAP clients, you use a utility called `bprowsdldoc`, which is also referred to as the **WSDL Analyzer**. This utility reads a WSDL document and generates documentation that includes sample ABL code snippets that you can copy and modify as you develop an ABL SOAP client.

Run the WSDL Analyzer

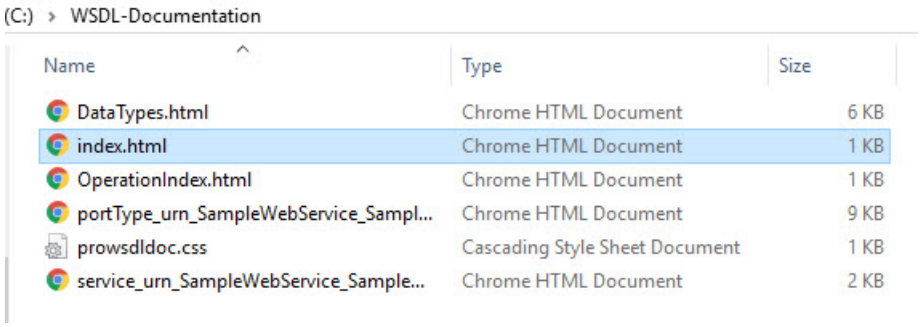
Perform the following steps to run the WSDL Analyzer:

1. Launch **Proenv**.
2. Enter `bprowsdldoc`, followed by:
   - The location of the WSDL file (if the WSDL file is stored locally) or the URL of the WSDL
   - An output directory where the WSDL documentation is generated
   - • The location of the WSDL file (if the WSDL file is stored locally) or the URL of the WSDL
     • An output directory where the WSDL documentation is generated

     For example:

     ```
     proenv>bprowsdldoc SampleWebService.wsdl "C:\WSDL-Documentation"
     ```

     For a more information on the `bprowsdldoc` command, see WSDL Analyzer on page 65.
3. The following files will be put in the directory.



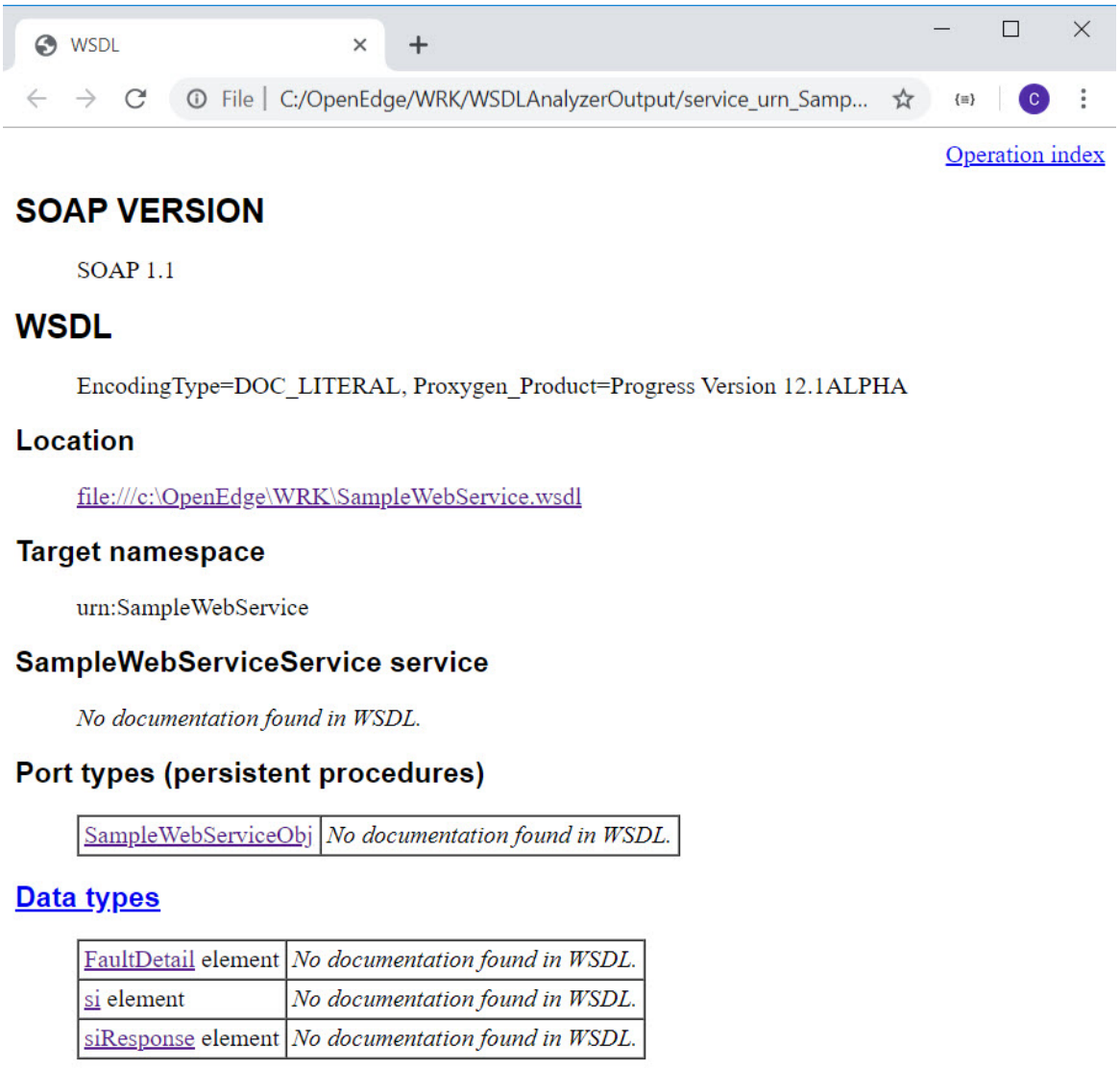
Examine the generated WDSL documentation

The WSDL Analyzer output is an HTML files in the designated directory.

Explore WSDL documentation

The `index.html` provides the top level listing for the WSDL documentation. The links allow you to drill down into the associated details.

1. Open the `index.html` to view the WSDL documentation.



Get more details

For a more information on how the output relates to the underlying WSDL, seeWSDL Details on page 69.

Develop code from sample snippets

Overview of client code

Use the generated WSDL documentation to complete the standard steps for consuming a SOAP web service which include the following:

1. Connects an ABL server object to the web service using the `CONNECT( )` method, specifying the location of the WSDL file for run-time access and the name of a port type on the server.
2. Creates a procedure object and associates it with the web service using the `RUN...SET` syntax.
3. Invokes a web service operation using the `RUN...IN` syntax, passing in any necessary parameters.

4. Handles any errors or return values.
5. Deletes the procedure object using `DELETE PROCEDURE` when it is no longer needed.
6. Unbinds the web service from the ABL server object using the `DISCONNECT( )` method when all processing is complete.
7. Deletes the ABL server object using `DELETE OBJECT` when it is no longer needed.

This comments in this code identify an example of each step.

```
DEFINE VARIABLE hWebService AS HANDLE NO-UNDO.
DEFINE VARIABLE hMySampleWebServiceObj AS HANDLE NO-UNDO.
DEFINE VARIABLE iCustNum AS INTEGER NO-UNDO.
DEFINE VARIABLE result AS CHARACTER NO-UNDO.
DEFINE VARIABLE cName AS CHARACTER NO-UNDO.

iCustNum = 5000.

CREATE SERVER hWebService.

//Step 1
hWebService:CONNECT("-WSDL
'http://machinename:8810/MySampleApp/soap/wsdl?targetURI=urn:MySampleWebService'").
IF NOT hWebService:CONNECTED() THEN DO:
    MESSAGE "SERVER: " SKIP
"http://machinename:8810/MySampleApp/soap/wsdl?targetURI=urn:MySampleWebService" SKIP

                       "is not connected"
        VIEW-AS ALERT-BOX INFO BUTTONS OK.
    RETURN.
END.

// Step 2
RUN MySampleWebServiceObj SET hMySampleWebServiceObj ON hWebService.
IF NOT VALID-HANDLE(hMySampleWebServiceObj) THEN DO:
MESSAGE "PortType: " VALID-HANDLE(hMySampleWebServiceObj) " is not valid"
        VIEW-AS ALERT-BOX INFO BUTTONS OK.
        RETURN.
END.

// Step 3
RUN serviceInterface IN hMySampleWebServiceObj(INPUT iCustNum, OUTPUT result, OUTPUT
cName).

// Step 4
IF ERROR-STATUS:ERROR THEN DO:
  DEFINE VARIABLE iCnt AS INTEGER NO-UNDO.
  DO iCnt = 1 TO ERROR-STATUS:NUM-MESSAGES:
      MESSAGE ERROR-STATUS:GET-MESSAGE(iCnt)
          VIEW-AS ALERT-BOX INFO BUTTONS OK.
 END.

// Step 4 continued
IF VALID-HANDLE(ERROR-STATUS:ERROR-OBJECT-DETAIL) THEN DO:
     DEFINE VARIABLE hXML        AS HANDLE     NO-UNDO.
     DEFINE VARIABLE mDoc        AS MEMPTR     NO-UNDO.
     CREATE X-DOCUMENT hXML.
     hXML:LOAD('LONGCHAR',
ERROR-STATUS:ERROR-OBJECT-DETAIL:SOAP-FAULT-DETAIL:GET-SERIALIZED(), FALSE).
     hXML:SAVE("memptr", mDoc).
     MESSAGE "Fault Code  : " ERROR-STATUS:ERROR-OBJECT-DETAIL:SOAP-FAULT-CODE  SKIP
                "Fault String: " ERROR-STATUS:ERROR-OBJECT-DETAIL:SOAP-FAULT-STRING
SKIP
               "Fault Actor : " ERROR-STATUS:ERROR-OBJECT-DETAIL:SOAP-FAULT-ACTOR SKIP

                "Error Type  : " ERROR-STATUS:ERROR-OBJECT-DETAIL:TYPE  SKIP SKIP
                "Fault Detail: " SKIP GET-STRING(mDoc,1)
        VIEW-AS ALERT-BOX INFO BUTTONS OK.
```

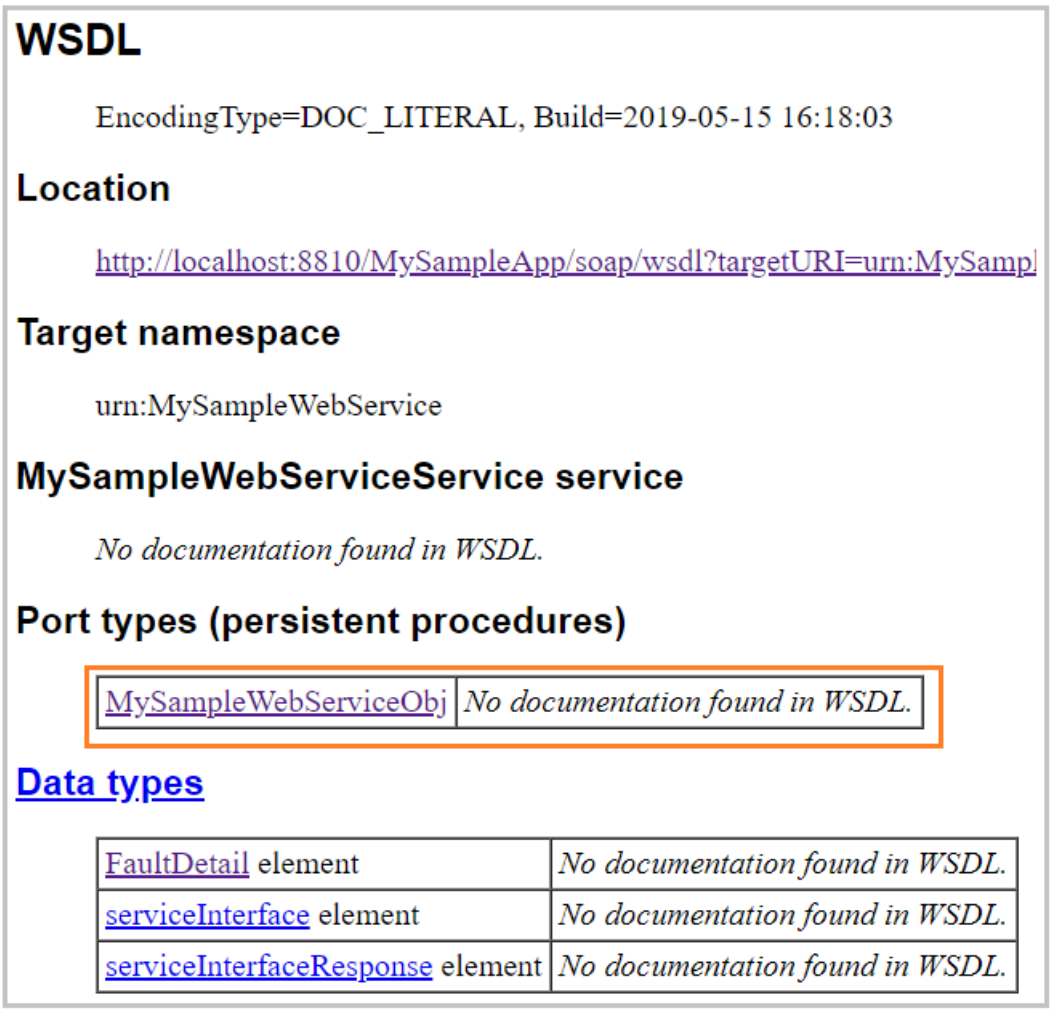
```
  END.

END.

MESSAGE cName.

// Step 5
DELETE OBJECT hMySampleWebServiceObj.

// Step 6
hWebService:DISCONNECT().

// Step 7
DELETE OBJECT hWebService.
```

For more information on the methods and statements listed above see, ABL Reference

Locate details for code in the WSDL documentation

To build the ABL SOAP client using code snippets generated by the WSDL Analyzer and available from the WSDL documentation, perform the following steps:

1. Create an ABL procedure file.
2. Open the `index.html` file in the WSDL documentation and click a link to a port type that you want to invoke.



This opens the port type documentation.

3. From the port type documentation, copy the example ABL code snippet that defines **connection parameters** and paste it into the ABL procedure file.

   Example

   This example shows the steps needed to prepare for calling one of the operations described below.

```
DEFINE VARIABLE hWebService AS HANDLE NO-UNDO.
DEFINE VARIABLE hMySampleWebServiceObj AS HANDLE NO-UNDO.


CREATE SERVER hWebService.


hWebService:CONNECT("-WSDL 'http://localhost:8810/MySampleApp/soap/wsdl?targetURI=urn:MySampleWe

RUN MySampleWebServiceObj SET hMySampleWebServiceObj ON hWebService.
```

4. Locate the example code snippet that defines **input and output variables** and copy those definitions into your procedure file.

   Example

```
DEFINE VARIABLE iCustNum AS INTEGER NO-UNDO.
DEFINE VARIABLE result AS CHARACTER NO-UNDO.
DEFINE VARIABLE cName AS CHARACTER NO-UNDO.


FUNCTION serviceInterface RETURNS CHARACTER
  (INPUT iCustNum AS INTEGER,
   OUTPUT cName AS CHARACTER)
  IN hMySampleWebServiceObj.


/* Function invocation of serviceInterface operation. */
result = serviceInterface(iCustNum, cName).


/* Procedure invocation of serviceInterface operation. */
RUN serviceInterface IN hMySampleWebServiceObj(INPUT iCustNum, OUTPUT result, OUTPUT cName).
```

5. From the same code snippet, copy and paste the line that **runs the operation**.

   Example

```
DEFINE VARIABLE iCustNum AS INTEGER NO-UNDO.
DEFINE VARIABLE result AS CHARACTER NO-UNDO.
DEFINE VARIABLE cName AS CHARACTER NO-UNDO.


FUNCTION serviceInterface RETURNS CHARACTER
  (INPUT iCustNum AS INTEGER,
   OUTPUT cName AS CHARACTER)
  IN hMySampleWebServiceObj.


/* Function invocation of serviceInterface operation. */
result = serviceInterface(iCustNum, cName).


/* Procedure invocation of serviceInterface operation. */
RUN serviceInterface IN hMySampleWebServiceObj(INPUT iCustNum, OUTPUT result, OUTPUT cName).
```

6. Assign values to input variables and add statements to process the output. (See the statement that runs the operation—`RUN serviceInterface` in this example—to know which variables are input and which ones are output).

   Input variables must be initialized before the operation is run. Statements to process the output from the web service should be placed after the RUN statement.

At this point, your ABL procedure file should resemble this example:

```
DEFINE VARIABLE hWebService AS HANDLE NO-UNDO.
DEFINE VARIABLE hMySampleWebServiceObj AS HANDLE NO-UNDO.
DEFINE VARIABLE iCustNum AS INTEGER NO-UNDO.
DEFINE VARIABLE result AS CHARACTER NO-UNDO.
DEFINE VARIABLE cName AS CHARACTER NO-UNDO.

iCustNum = 1.

CREATE SERVER hWebService.

hWebService:CONNECT("-WSDL
'http://machinename:8810/MySampleApp/soap/wsdl?targetURI=urn:MySampleWebService'").

RUN MySampleWebServiceObj SET hMySampleWebServiceObj ON hWebService.
RUN serviceInterface IN hMySampleWebServiceObj(INPUT iCustNum, OUTPUT result, OUTPUT
 cName).

MESSAGE cName.
```

7. You should now be able to test the web service by running the procedure (**CTRL+F11** in OpenEdge Developer Studio). Before you do so, you should modify your code to adhere to good programming practices. Write code to handle faults and other types of exceptions, delete the handles, and disconnect from the web service as shown in the overview example.

Additional information

- To learn more about data type conversions between ABL and SOAP web services, see Data Type Conversion Rules for ABL Calls to SOAP Web Services on page 101.
- To learn more about handles, statements, attributes, methods and events used to develop ABL SOAP clients, see ABL Elements to Consume SOAP Web Services on page 87.

2

Handling SOAP Message Headers in ABL

SOAP headers are an optional part of a SOAP message and the requirement to use them depends on the Web service. Many Web services require no client handling of SOAP headers.

If the Web service requires you to handle SOAP headers, especially if you need to modify or examine the contents of a SOAP header, you might need to understand ABL facilities for parsing and managing XML. This includes ABL support for parsing and directly manipulating an XML document in a Document Object Model (DOM) tree or parsing and writing the elements of an XML document using the Simple API for XML (SAX). For more information on these facilities, see *OpenEdge Development: Programming Interfaces.*

In rare instances, you might be able to access the data in a SOAP header as an ABL temp-table or ProDataSet, without the need to directly manage the XML. However, this is only possible if the WSDL Analyzer recognizes complex data in the header that it can map to a temp-table or ProDataSet definition. For more information, see Analyzing complex data.

The sections in this chapter describe how to define and use specified callback procedures (header handlers) that manage the SOAP headers for SOAP request and response messages.

For details, see the following topics:

- SOAP message headers: an overview
- Defining header handlers
- Invoking a header handler
- Creating and managing SOAP message headers
- Managing memory for SOAP headers
- Attributes and methods for handling SOAP headers

SOAP message headers: an overview

A SOAP request message or a SOAP response message can contain an optional <Header> element that passes non-parameter information about the Web service or the operation that generates the message. This header can contain one or more header entries. A *header entry* contains information that is necessary to properly invoke some or all of the Web service operations.

A header returned by the Web service in a SOAP response message is a *SOAP response header*. A header sent in a SOAP request message by invoking a Web service operation is a *SOAP request header*. There is no difference in the syntax for the two types. However, keeping them conceptually separate helps to clarify how you can manage them. The content of a SOAP request header for a given operation might vary from the content of a SOAP response header for that same operation.

Why are headers necessary if operations already pass data as parameters? Headers can simplify managing data in Web services that require context management, such as user authentication, during the course of Web service interaction. Session-managed OpenEdge Web services use this method for maintaining context. With the use of headers, the mainline of a client application need have no concern about maintaining these states. The Web service provider typically provides specific information about the content of headers and how they must be handled by client applications. The client can then provide special *header handlers*, which are callback procedures that manage the content of these headers for the Web service application.

SOAP header structure

The following XML contains a SOAP header as it might appear inside a SOAP message:

SOAP message with header

```
<soap:Envelope>
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   . . .
   <soap:Header>
      <q1:OrderInfoID
                  xmlns:q1="http://www.example.com/webservices/OrderInfo"
                  SOAP-ENV:mustUnderstand="1"
                  SOAP-ENV:actor="/WSA">
         <uuid xsi:type="xsd:string">12345-67890-98765-4321</uuid>
      </q1:OrderInfoID>
      <t:Transaction
                  xmlns:t="http://www.example.com/webservices/Transaction"
                  SOAP-ENV:mustUnderstand="1" t:stepnum="1">
         <t:TransID>5</t:TransID>
         <t:SubID>871</t:SubID>
      </t:Transaction>
   </soap:Header>
   <soap:Body ...>
      <parameter-element> ... </parameter-element>
      . . .
   </soap:Body>
</soap:Envelope >
```

As you can see, the <Header> element is only a container for other elements, the application-specific header entry elements. Any parameters generated for an operation in this SOAP message appear in the <Body> element in much the same way.

In this case, the SOAP header contains two header entries, named `q1:OrderInfoID` and `t:Transaction` (where `q1:` and `t:` are namespace prefixes). Each of these elements has a series of attributes and contains one or more child elements. The `<q1:OrderInfoID>` element contains one child element named `uuid`, and the `<t:Transaction>` element contains two child elements, named `t:TransID` and `t:SubID`. All of these element names are application-defined.

Accessing the SOAP header

To access the SOAP header and its elements from your ABL application, you define callback procedures using specific signatures that act as header handlers. When you need to access the headers, you need separate header handlers for each Web service procedure object. A header handler can handle either the SOAP request headers or the SOAP response headers for all operations maintained by that procedure object. A single header handler cannot handle both types of headers. You can specify the handlers for a Web service procedure object by invoking the `SET-CALLBACK-PROCEDURE( )` method on the corresponding procedure handle.

If you want to add additional handlers for the same port type managed by a Web service procedure object, you must map the port type using additional procedure objects.

Every time you invoke an operation for a port type where you have specified header handlers, the handlers execute as part of the operation invocation. For SOAP request messages, the handler intercepts and manages the SOAP headers before the message is sent to the Web service. For SOAP response messages, the handler intercepts and manages the SOAP headers before any message parameters are passed to complete invocation of the operation.

Requirements for header handlers

Using header handlers is not an absolute requirement. The design of the Web service drives the need for managing headers. The only absolute requirement for a header handler is the signature that you must use to define it. You can set and reset the callbacks for header handlers as needed. For example, you might make changes to do the following:

- Specify the correct handlers for different operations and port types
- Add additional header handlers for a single port type by mapping the port type to additional Web service procedure objects
- Remove any previously specified callbacks entirely when you have no need to handle the SOAP headers for an operation or port type

For more information on:

- How to specify callback procedures as header handlers, see Specifying SOAP header callback procedures at run time on page 22.
- The general cycle of operation for header handlers in ABL, see Invoking a header handler on page 26.
- Defining the callback procedures for header handlers, see Defining header handlers on page 23.

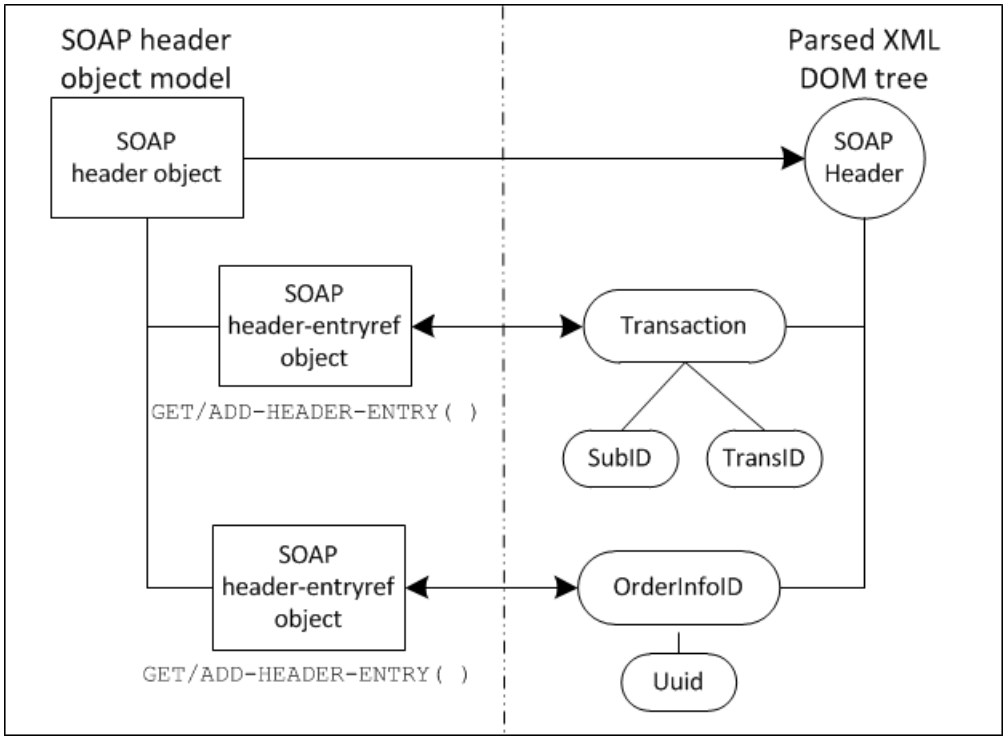
SOAP header object model

The most important part of a header handler signature is a `HANDLE` parameter that references the header of a SOAP message. This handle references a SOAP header object that maps to the parsed XML of the actual SOAP message header in the form of a DOM tree. The SOAP header object implements part of an object model that allows you to directly access the XML content of the SOAP header without having to parse it out of the SOAP message. Figure 1 shows the relationship between the SOAP header object model and the parsed XML DOM tree using the SOAP header described in the preceding section.

The complete SOAP header object model includes two types of objects:

- **SOAP header object** — References the `<Header>` element of a SOAP message.
- **SOAP header-entryref object** — References one of the header entry elements in a SOAP header as specified by the `GET-HEADER-ENTRY( )` method on the SOAP header object. The SOAP header entryref object can also reference a header entry element that you add to the SOAP header using the `ADD-HEADER-ENTRY( )` method on the SOAP header object.

Figure 1: Referencing a header entry in the SOAP header object model



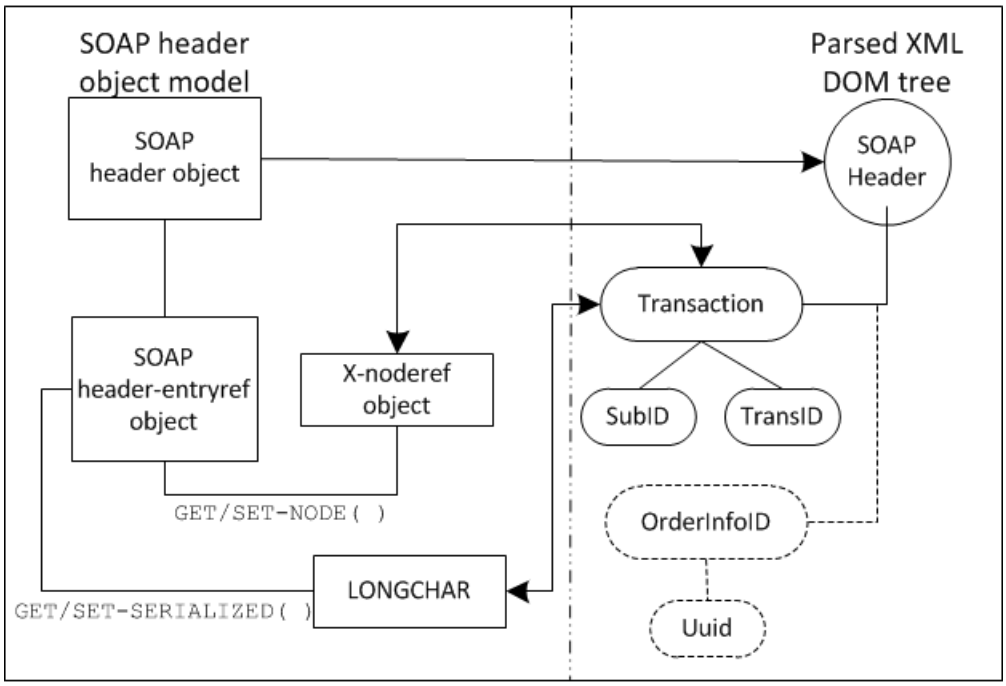
The SOAP header object handle also provides attributes that return information about the name of the header and the number of header entries that the header contains.

In the above figure, one SOAP header-entryref object references the `Transaction` header entry and another object references the `OrderInfoID` header entry. When working with a SOAP header, you can also use a single SOAP header-entryref object to reference all of the header entries in a SOAP header, one at a time. The SOAP header-entryref object handle provides several ABL attributes and methods that allow you to inspect and modify the contents of the header entry that it references. The attributes return the values of standard SOAP attributes contained in the header entry element, such as the name and namespace of the header entry.

Accessing SOAP header entries

The SOAP header-entryref object handle provides two methods for accessing the element subtree of the SOAP header entry that it references, as shown in the following figure.

Figure 2: Accessing entry elements in the SOAP header object model



The SOAP header-entryref object handle provides two methods to access the XML in the header entry that it references:

- `GET-NODE( ) method`—assigns an X-noderef object to reference the root node of the header entry in the DOM subtree of the header entry. You can then use this X-noderef object to examine the XML elements by walking and returning values for the nodes of the header entry DOM subtree.
- `GET-SERIALIZED( ) method`—assigns a `LONGCHAR` value that contains the entire header entry as a serialized XML string that you can parse yourself using either the ABL SAX reader or DOM parser.

The SOAP header-entryref object handle also provides two corresponding methods for replacing the entire content of the SOAP header entry that it references:

- `SET-NODE( ) method`—replaces the content of the header entry with the DOM subtree whose root node is currently referenced by a specified X-noderef object.
- `SET-SERIALIZED( ) method`—replaces the content of the header entry with a DOM subtree that it parses from a specified `LONGCHAR` value containing the serialized XML for the header entry that you can build and write using a DOM tree or the ABL SAX writer.

For more information on these objects and all the attributes and methods that they support, see Creating and managing SOAP message headers on page 27.

In the rare instances when the WSDL Analyzer identifies a temp-table or ProDataSet definition that maps to the header entry, you can use the `GET-SERIALIZED( )` and `SET-SERIALIZED( )` methods in conjunction with the `READ-XML( )` and `WRITE-XML( )` methods of the documented ABL object (temp-table or ProDataSet) to access or build the header entry.

Accessing a SOAP response header

To access a SOAP response header entry as a temp-table or ProDataSet:

1. Set a LONGCHAR with the header entry XML using the GET-SERIALIZED( ) method.
2. Load the temp-table or ProDataSet from the LONGCHAR using the READ-XML( ) method.

Creating a SOAP request header

To create a SOAP request header entry from a temp-table or ProDataSet:

1. Write the header entry from the temp-table or ProDataSet to a LONGCHAR using the WRITE-XML( ) method.
2. Replace the header entry with the serialized XML in the LONGCHAR using the GET-SERIALIZED( ) method.

For more information on temp-table and ProDataSet XML methods, see *OpenEdge Development: Programming Interfaces.*

Specifying SOAP header callback procedures at run time

For an AppServer, all information about remote requests is either passed with parameters or obtained using the attributes and methods of various handles related to the request. For a Web service, most information about Web service requests is passed with parameters, but some information is also contained within other elements of the SOAP message, such as the SOAP header. The SET-CALLBACK-PROCEDURE( ) method allows you to specify internal procedures that can access the content of the SOAP header element in a SOAP message as part of the invocation of a Web service operation that generates the message.

The SET-CALLBACK-PROCEDURE( ) method registers an internal procedure with a specified *ABL callback*, a designation that causes the internal procedure (*callback procedure*) to execute in response to some ABL core action, such as in this case, sending and receiving SOAP messages. For Web services, this method supports two types of ABL callbacks:

- **"REQUEST-HEADER"** — Identifies a SOAP request header callback procedure. This procedure executes just after you invoke a Web service operation and just before OpenEdge sends the SOAP request message to the Web service. It allows you to create or pass an existing SOAP header for use as the SOAP header of an out-going SOAP request message.
- **"RESPONSE-HEADER"** — Identifies a SOAP response header callback procedure. This procedure executes just after OpenEdge receives the message sent by the Web service in response to processing the Web service operation that you invoked, and just before it makes any output or return parameters available to the ABL calling context. It allows you to examine the SOAP header of an in-coming SOAP message.

Note: While this header callback procedure executes for either a normal SOAP response message or a SOAP fault message generated for the Web service request, a SOAP header is only available in a SOAP response that is not a SOAP fault.

A single Web service procedure object can support only one request header callback and one response header callback at one time. This means that for a single procedure object, you must use the same callbacks for all operations invoked on that procedure object. However, if you want greater flexibility to use different callbacks for the same set of operations, you can use the RUN*portTypeName* statement to create an additional Web service procedure object for the same port type. You can then invoke the SET-CALLBACK-PROCEDURE( ) method on the new procedure object to assign a different set of callback procedures for use by that same port type. By creating additional Web service procedure objects this way, you can then assign as many callback procedures for a single port type as you want.

Using the SET-CALLBACK-PROCEDURE( ) method

This is the syntax for the SET-CALLBACK-PROCEDURE( ) method:

Syntax

```
SET-CALLBACK-PROCEDURE (callback-name, internal-procedure
            [ , procedure-context ])
```

The parameters include:

- ***callback-name*** — A character string value of "REQUEST-HEADER" or "RESPONSE-HEADER" to identify the type of callback.
- ***internal-procedure*** — A character expression containing the name of the internal procedure you want executed for the specified callback. To unregister a callback definition, pass an empty string ("").
- ***procedure-context*** — (Optional) A procedure object handle that points to an active procedure context that contains the specified internal procedure. This is typically a persistent procedure instantiated in the current OpenEdge session or the method calling context (THIS-PROCEDURE). If not specified, it is the value of THIS-PROCEDURE.

In general, you can define the internal procedure for a callback in any procedure context that is guaranteed to be active at the time you execute the SET-CALLBACK-PROCEDURE( )method, and that remains active until after the callback procedure is executed by OpenEdge. The time of this execution depends on the type of request:

- **Synchronous Web service requests** — For either a SOAP request header callback or a SOAP response header callback, the procedure context must remain active until the operation invocation has completed execution.
- **Asynchronous Web service requests** — For a SOAP request header callback, the procedure context must remain active until the operation invocation has completed execution. For a SOAP response header callback, the procedure context must remain active until activation of the PROCEDURE-COMPLETE event for the request and any results have been returned to the PROCEDURE-COMPLETE event procedure.

Defining header handlers

An ABL application does most of the work for accessing or creating a SOAP header in the two header handlers that you can invoke for Web service operations. These include a callback procedure or method to handle the SOAP response message header and a callback procedure or method to handle the SOAP request message header for the operation. The syntax for defining the signatures for these request and response header callback handlers are specified below:

SOAP response header callback procedure or method

Method syntax

```
METHOD PUBLIC VOID response-header-method ( INPUT hSOAPHeader AS HANDLE,
    INPUT cOperationNamespace AS CHARACTER,
    INPUT cOperationLocalName AS CHARACTER):
END METHOD.
```

Procedure Syntax

```
PROCEDURE response-header-procname :
  DEFINE INPUT PARAMETER hSOAPHeader AS HANDLE .
  DEFINE INPUT PARAMETER cOperationNamespace AS CHARACTER .
  DEFINE INPUT PARAMETER cOperationLocalName AS CHARACTER .
END PROCEDURE .
```

SOAP request header callback procedure or method:

Method Syntax

```
METHOD PUBLIC VOID request-header-method ( OUTPUT hSOAPHeader AS HANDLE,
    INPUT  cOperationNamespace AS CHARACTER,
    INPUT  cOperationLocalName AS CHARACTER,
    OUTPUT lDeleteOnDone AS LOGICAL):
END METHOD.
```

Procedure Syntax

```
PROCEDURE request-header-procname :
                      DEFINE OUTPUT PARAMETER hSOAPHeader AS HANDLE .
                      DEFINE INPUT  PARAMETER cOperationNamespace AS CHARACTER .
                      DEFINE INPUT  PARAMETER cOperationLocalName AS CHARACTER .
                      DEFINE OUTPUT PARAMETER lDeleteOnDone AS LOGICAL .
                    END PROCEDURE .
```

These are the parameters:

- ***response-header-method*** —The name of a response header handler method. Specified as a character expression in the SET-CALLBACK( ) method's *routine-name* parameter representing the name of a method that resides within an object reference for a class instance along with the *callback-name* parameter set to "RESPONSE-HEADER".
- ***request-header-method*** —The name of a request header handler method. Specified as a character expression in the SET-CALLBACK( ) method's *routine-name* parameter representing the name of a method that resides within an object reference for a class instance along with the *callback-name* parameter set to "REQUEST-HEADER".
- ***response-header-procname*** — The name of a response header handler procedure. Specified as a character expression to the SET-CALLBACK( ) method's *routine-name* parameter along with the the *callback-name* parameter set to "RESPONSE-CALLBACK".
- ***request-header-procname*** — The name of a request header handler procedure. Specified as a character expression to the SET-CALLBACK( ) method's *routine-name* parameter along with the the *callback-name* parameter set to "REQUEST-CALLBACK" setting.
- ***hSOAPHeader*** — A handle to a SOAP header object that encapsulates the header of the SOAP message that is about to be sent (request header) or that has just been received (response header). In a response header handler, the SOAP header object has no content if the NUM-HEADER-ENTRIES attribute on the object handle returns the value 0; otherwise it contains one or more SOAP header entries. In a request header handler, this is an OUTPUT parameter, therefore if the outgoing SOAP message requires a SOAP header, you must either build a SOAP header for this parameter to reference or provide an existing SOAP header saved from a previous response callback.

- ***cOperationNamespace*** — Contains the namespace portion of the operation's qualified name. Use this parameter together with the *cOperationLocalName* parameter if you need to identify the operation for which the SOAP message is being sent or received.
- ***cOperationLocalName*** — Contains the local-name portion of the operation's qualified name. Use this parameter together with the *cOperationNamespace* parameter if you need to identify the operation for which the SOAP message is being sent or received.
- ***lDeleteOnDone*** — (Request callback only) Tells OpenEdge to delete the SOAP header object and all of the parsed XML after the SOAP header has been inserted into the out-bound SOAP message. For more information on the scope of these objects, see Managing memory for SOAP headers on page 42.

For both types of callback procedure you can use the INPUT parameters, *cOperationNamespace* and *cOperationLocalName*, to determine the Web service operation for which the message is generated. You might use this information either to determine how to parse the SOAP response header based on the invoked operation or to build a SOAP request header that is specific to the invoked operation.

If you need to pass context between the code that invokes a Web service operation and a header callback procedure, you can pass the context information as you might for any internal procedure:

- Use procedure variables global to the calling code, if the callback procedure is defined within the context of the calling code.
- Use the PRIVATE-DATA attribute on the procedure context handle (THIS-PROCEDURE) of the header handler.

Defining a response header handler

The response header handler receives a SOAP header object as an input parameter. This object references the SOAP header entries returned by the Web service for an operation. You might need to save the entire SOAP header for use in a subsequent request, or you might need to save specific information from one or more individual header entries for validation or some other application purpose. Such requirements dependent entirely on the Web service. However, when your application no longer needs the input header itself, you must delete the SOAP header object explicitly.

Caution: After receiving a response message for which a header handler has been defined, OpenEdge creates a SOAP header object for input to the header handler even if the message contains no SOAP header. Your application must manage the lifetime of this response header object even if it does nothing else except delete the object (typically done in the response header handler). To avoid having to manage SOAP header objects for response messages that do not contain headers, do not define a response header handler for these response messages or unregister any that you have defined. For more information, see Using the SET-CALLBACK-PROCEDURE( ) method on page 23.

Defining a request header handler

The main function of a request header handler is to build a SOAP <Header> element appropriate for the operation. You can do this by creating a SOAP header object and associating with it the required number of SOAP header entries. The handler has an OUTPUT parameter (*hSOAPHeader*) that you can use to pass the SOAP header that you have built to OpenEdge to insert into the outgoing SOAP message. Any changes you make in the handler to the SOAP header object or its constituent SOAP header-entryref objects are integrated into the SOAP message when the handler completes execution. Thus, if the SOAP header that you return in this procedure contains SOAP header entries, OpenEdge sends out the SOAP message containing a corresponding header element. If you do not set the SOAP header object OUTPUT parameter (*hSOAPHeader*), OpenEdge does not include a header in the outgoing SOAP message.

If you create a SOAP header in the handler, with potentially multiple SOAP header entries, you are responsible for cleaning up the memory that is used for it. As an aid for deleting this object, and its memory, you can set the *lDeleteOnDone* parameter for output. Setting this parameter to TRUE, directs OpenEdge to delete the SOAP header object after OpenEdge has copied the object's contents to the outgoing message. Otherwise, your application must manage the lifetime of this object and determine when the object gets destroyed.

Caution: Do not explicitly delete the header object within the context of the request header handler (before it completes). If you do this, OpenEdge never gets the header to include in the SOAP request message.

Invoking a header handler

The following procedure fragment shows how ABL might specify and invoke a header handler for the SOAP message described in the previous sections, in this case returned as a SOAP response message:

Invocation of a header handler

```
/*** PROCEDURE: Start transactions on an order. ***/
DEFINE INPUT PARAMETER OrderNum AS INTEGER.
DEFINE OUTPUT PARAMETER hOrderSvc AS HANDLE.
DEFINE OUTPUT PARAMETER gcUUID AS CHARACTER.

/* Create hOrderSvc server handle and connect to order Web service. */
. . .

/* Create hPortType procedure handle and map port type to it. */
. . .

/* Set up response header handler. */
hPortType:SET-CALLBACK-PROCEDURE( "RESPONSE-HEADER",
                                  "TransactionResponseHandler",
                                  THIS-PROCEDURE ).
/* Start a transaction on an order. */RUN startTransaction( "ORDER", OrderNum ) IN
hPortType NO-ERROR.

. . .
```

This fragment specifies the response header handler for an internal procedure named `TransactionResponseHandler` that is defined in the current external procedure context. Then, it invokes a `startTransaction` procedure as the first operation. As the name and parameters imply, this operation begins a transaction on the order (perhaps automatically retrieved or created, as necessary) with the specified order number.

Assume the SOAP response message returned by this operation has a header containing database object and transaction state information for the transaction that was started. The example SOAP message, in SOAP header structure on page 18, contains just such information, including an ID for the order (the `<q1:OrderInfoID>` element) and some values identifying the transaction that is managing the order (the `<t:Transaction>` element).

The following internal procedure defines the `TransactionResponseHandler` callback procedure for the header handler. In this case, the handler locates the `<uuid>` element in the `OrderInfoID` header entry within the SOAP header referenced by the SOAP header object handle parameter, `hSOAPHeader`. It then saves the `uuid` string value to an `OUTPUT` parameter(`gcUUID`)defined globally in the calling procedure context. This is all accomplished using methods and attributes of the SOAP header object, SOAP header entry object, and x-noderef objects to access the parsed XML DOM tree of the SOAP header:

Definition of a header handler procedure

```
/*
  This response handler looks for a header entry named "OrderInfoID" and
  assumes that it contains an element named "uuid". The handler saves away
  the value of "uuid". This routine assumes that the SOAP header is no longer
  needed after the callback completes.
*/
PROCEDURE TransResponseHandler:
  DEFINE INPUT PARAMETER hSOAPHeader AS HANDLE.
  DEFINE INPUT PARAMETER cOperationNamespace AS CHARACTER.
  DEFINE INPUT PARAMETER cOperationLocalname AS CHARACTER.

  DEFINE VARIABLE hsheEntry AS HANDLE.
  CREATE SOAP-HEADER-ENTRYREF hsheEntry IN WIDGET-POOL "soap".

  DEFINE VARIABLE hxnTemp AS HANDLE.
  DEFINE VARIABLE hxnWorkRoot AS HANDLE.
  DEFINE VARIABLE hxnTemp2 AS HANDLE.

  CREATE X-NODEREF hxnTemp IN WIDGET-POOL "soap".
  CREATE X-NODEREF hxnTemp2 IN WIDGET-POOL "soap".
  CREATE X-NODEREF hxnWorkRoot IN WIDGET-POOL "soap".

  /* Walk the SOAP-HEADER's list of header entries, */
  /* looking for the "OrderInfoID" header entry     */
  DEFINE VARIABLE idx AS INTEGER.
  REPEAT idx = 1 TO hSOAPHeader:NUM-HEADER-ENTRIES:
    hSOAPHeader:GET-HEADER-ENTRY(hsheEntry, idx).
    IF hsheEntry:LOCAL-NAME = "OrderInfoID" AND
       hsheEntry:NAMESPACE-URI =
                 "http://www.example.com/webservices/OrderInfo"
    THEN DO:
      /* Get the X-noderef side of the hsheEntry so we can navigate its
        body.    */
      hsheEntry:GET-NODE(hxnWorkRoot).
      hxnWorkRoot:GET-CHILD(hxnTemp, 1). /* hxnTemp is now uuid node */
      hxnTemp:GET-CHILD(hxnTemp2,1). /* hxnTemp2 is text node of uuid */
      gcUUID = hxnTemp2:NODE-VALUE.  /* save the text content */
    END.
  END.
  /* Delete all objects created in this procedure. */
  DELETE WIDGET-POOL "soap".
  /* Delete the SOAP header freeing all of its memory. */
  DELETE OBJECT hSOAPHeader.
END PROCEDURE.
```

Creating and managing SOAP message headers

Depending on the requirements of the Web service, you can work with SOAP headers in several different ways. Three basic use cases that a Web service might require for handling SOAP headers in a client application include:

1. Re-using a SOAP header for a response message that you must return unchanged in a subsequent request message
2. Re-using a SOAP header for a response message that you must return in a subsequent request message, but modified with additional or updated header entries
3. Building a unique SOAP header from scratch to provide as output from the request header handler

As described previously (see SOAP header object model on page 19), ABL represents a SOAP message header as:

- One SOAP header object
- One SOAP header-entryref object to access each SOAP header entry
- One X-noderef object to access a selected SOAP header entry as a DOM subtree, thus modeling the underlying XML of the header entry for access by ABL one node at a time

Depending on the situation, you might need to work only at the level of the SOAP header object with no need to inspect the contents, or you might need to access individual SOAP header entries and their individual XML elements.

Thus, the following sections describe examples of each of the three basic use cases presented in this section that you are most likely to encounter when managing SOAP headers as part of Web service access.

Note: All of these examples are based on a fictitious Web service (`HeaderExample`), using code that has been run against a working version of the Web service. In the code examples, listed steps refer to the numbers in comments, such as **`/*2b*/`** or **`/*9*/`**.

Reusing an unchanged SOAP response header

This example shows how you might handle a SOAP header that you first encounter in the response message returned from the Web service, then use unchanged as the SOAP header for the next request. This is an example of the header returned from the Web service:

SOAP response header to be reused

```
<soap:Envelope>
  . . .    <soap:Header>
     <AuthHeader xmlns="http://ServiceHost/SOAPHeader">
       <AccessID>XYZZY</AccessID>
     </AuthHeader>
   </soap:Header>
  . . .
</soap:Envelope >
```

It contains one header entry, `AuthHeader`, that contains a value used as an access key (`AccessID`). This type of header might be used when the Web service and client maintain a consistent context for each other between requests.

This is the mainline of a procedure that invokes the Web service to reuse the response header:

Invoking a request that reuses an unchanged SOAP response header

```
/* SOAPHeader1.p
 * Calls a fictitious Web service, first to request access, which gets back
 * a SOAP response header containing an AccessID, and sends the response
 * header back as part of a new request using the required access
 * credential that allows the Web service to respond appropriately to
 * the follow-up request.
 * The Web service has only one service and port available.   *//*1*//* Define local
variables */
DEFINE VARIABLE hWebSrvc    AS HANDLE.
DEFINE VARIABLE hPortType   AS HANDLE.
DEFINE VARIABLE cResponse   AS CHARACTER FORMAT "x(72)".
DEFINE VARIABLE g_header    AS HANDLE.

/* Create the Web service server object */
CREATE SERVER hWebSrvc.

/* Connect to the Web service */
hWebSrvc:CONNECT("-WSDL
                 http://ServiceHost/SOAPHeader/HeaderExample.asmx?wsdl").

/* Get the method, set the port type */
RUN HeadersSoap SET hPortType ON hWebSrvc.
/*2*/
/* Associate the req. & resp. callbacks with the port type */
hPortType:SET-CALLBACK-PROCEDURE("REQUEST-HEADER", "ReqHandler").
hPortType:SET-CALLBACK-PROCEDURE("RESPONSE-HEADER", "RespHandler").
/*3*/
/* Invoke the Web service with no header and display the results */
RUN OpenAccess IN hPortType (OUTPUT cResponse).
DISPLAY cResponse LABEL "WS response" WITH FRAME aaa.
/*4*/
/* Go again with the AccessID set from previous response header */
cResponse = "".
RUN HelloWorld IN hPortType (OUTPUT cResponse).
DISPLAY cResponse LABEL "WS response" WITH FRAME bbb.
```

```
/*5*/
DELETE OBJECT g_header.
DELETE OBJECT hPortType.
hWebSrvc:DISCONNECT().
DELETE OBJECT hWebSrvc.
/*************** Internal Procedures ***************/
```

The code in the preceding example:

1. Defines several mainline variables, including a global handle to reference the reused SOAP header (`g_header`).
2. Registers the request header (`ReqHandler`) and response header (`RespHandler`) handlers after connecting to the Web service and instantiating the `HeaderSoap` port type procedure object.
3. Runs the `OpenAccess` procedure to invoke the Web service operation that returns the `AccessID` value in the SOAP response header (see Response header handler for returning a header for reuse on page 30).
4. Runs the `HelloWorld` procedure to invoke the next Web service operation, passing back the SOAP response header to the Web service unchanged as the SOAP request header (see Request header handler for reusing a header on page 30).
5. Cleans up the global objects maintained in its context and disconnects from the Web service. Note that one of the objects it deletes is the original SOAP response header saved by the response header handler during execution of the `OpenAccess` procedure.

Response header handler for returning a header for reuse

This is the SOAP response header handler (RespHandler) that returns the header that is reused for passing around the AccessID value:

Response header handler saving a SOAP response header for reuse

```
PROCEDURE RespHandler: /*1*/
  DEFINE INPUT PARAMETER hHeader    AS HANDLE.
  DEFINE INPUT PARAMETER cNamespace AS CHARACTER.
  DEFINE INPUT PARAMETER cLocalNS   AS CHARACTER.
  /* If the g_header global variable is valid coming in, it has already been
    set in a previous response, therefore, delete the unnecessary response
    header object. Otherwise, set g-header to the response header object to
    pass back to the request header handler on subsequent requests. */

  IF NOT VALID-HANDLE(g_header) THEN /*2a*/ /* first response */
    g_header = hHeader.
  ELSE DO: /*2b*/ /* all subsequent responses */
    DELETE OBJECT hHeader.   END.
END PROCEDURE.
```

The code in the preceding example:

1. Receives the SOAP response header using the hHeader parameter
2. Tests if the global header handle (g_header) already references a valid object, and:
   a) If it does not reference an object, the handler must be running as part of the initial call to OpenAccess and thus saves the input SOAP header object (hHeader) to the global context (g_header) for use by subsequent requests. From this moment forward, all requests to the Web service discard the header object input to the response handler as unnecessary.
   b) If it does reference an object, the handle must already reference a SOAP response header returned in a prior request (the call to OpenAccess) and has no need of a subsequent response header. It therefore deletes the unnecessary SOAP header object returned to the handler through the hHeader parameter in order to prevent a memory leak accumulating in subsequent requests.

Request header handler for reusing a header

This is the SOAP request header handler (ReqHandler) that reuses the initial SOAP response header to pass the AccessID value between the client and Web service:

Request header handler reusing a saved SOAP response header

```
PROCEDURE ReqHandler: /*1*/  DEFINE OUTPUT PARAMETER hHeader    AS HANDLE.
  DEFINE INPUT  PARAMETER cNamespace AS CHARACTER.
  DEFINE INPUT  PARAMETER cLocalNS   AS CHARACTER.
  DEFINE OUTPUT PARAMETER lDeleteOnDone AS LOGICAL.

  /* The IF test determines if this is the first request. If it is, then
    g_header is not set and hHeader is set to ? to ensure that no header is
    sent. g_header gets set when the response header is returned, so a
    subsequent pass through this code takes the previous response header and
    sends it as the current request header. */

  IF NOT VALID-HANDLE (g_header) THEN DO: /*2a*/ /* first request */
    hHeader = ?.
    lDeleteOnDone = TRUE.
  END.
  ELSE DO: /*2b*/ /* all subsequent requests */
    hHeader = g_header.
    lDeleteOnDone = FALSE.
  END.
END PROCEDURE.
```

The code in the preceding example:

1. Sends the SOAP request header for the `HelloWorld` request (and any request run after running `OpenAccess`)
2. Tests if the global header handle (`g_header`) references a valid object, and:
   a) If it does not reference an object, the request handler must be running as part of the initial call to `OpenAccess` and sets the output parameters to ensure that no initial SOAP request header is sent.
   b) If it does reference an object, the handler passes the global header object as output using the request header parameter (`hHeader`) and ensures that the object is not deleted (saving it for use in any further request).

Modifying a reused SOAP response header

This example shows how you might handle a SOAP header that you first encounter in the response message returned from the Web service, then modify it as the SOAP header for the next request. The response header and its handler are identical to what is used in the header reuse example (see Reusing an unchanged SOAP response header on page 28). This is an example of the header returned from the Web service after a password is added:

SOAP request header built from a modified SOAP response header

```
<SOAP-ENV:Envelope
  . . .
    <SOAP-ENV:Header>
      <ns0:AuthHeader xmlns:ns0="http://ServiceHost/SOAPHeader">
        <ns0:AccessID>XYZZY</ns0:AccessID>
        <ns0:Password>Administrator</ns0:Password>
      </ns0:AuthHeader>
    </SOAP-ENV:Header>
  . . .
</SOAP-ENV:Envelope>
```

Note that the client adds the <Password> element as a sibling to the <AccessID> element in the existing AuthHeader header entry. Another approach is to create and add a new Password header entry as a sibling to the AuthHeader header entry itself. Again, the actual approach depends on the Web service itself, in this case the HeaderExample Web service. This type of header modification might be used when the Web service and client maintain a consistent context for each other between requests and the operation involved requires authorization or authentication or some other additional context information.

The following code is the mainline of a procedure that invokes the Web service to reuse the initial SOAP response header by adding a password node to it before passing it back as a SOAP request header:

1. Defines several mainline variables, including handles to access the global reused SOAP header (g_header) and its XML, and a variable to hold the password value (cPassword).
2. Registers the request header (ReqHandler) and response header (RespHandler) handlers after connecting to the Web service and instantiating the HeaderSoap port type procedure object.
3. Runs the OpenAccess procedure to invoke the Web service operation that returns the AccessID value in the SOAP response header (see Response header handler for returning a header for reuse on page 30).

Invoking a request that modifies a reused SOAP response header

```
/* SOAPHeader2.p
  An addition to SOAPHeader1.p.
  Calls a ficticious Web service. The first operation (OpenAccess) sends
  nothing in the request headers and gets back a SOAP response header
  containing an AccessID. The second operation (HelloWorld) sends the AccessID
  back in its request header. (No additional information is received in the
  response header.) The third operation (HelloSecureWorld) adds a Password
  node to the existing AccessID entry in its request header. This Password
  node is added as a sibling of the AccessID element and NOT as a new SOAP
  header entry. (Once again no additional information is received in the
  response header.) The Web service has only one service and port available.
*/
/*1*/
/* Define local variables */
DEFINE VARIABLE hWebSrvc      AS HANDLE.
DEFINE VARIABLE hPortType     AS HANDLE.
DEFINE VARIABLE cResponse     AS CHARACTER FORMAT "x(72)".
DEFINE VARIABLE hXdoc         AS HANDLE.
DEFINE VARIABLE hXnoderef1    AS HANDLE.
DEFINE VARIABLE hXnoderef2    AS HANDLE.
DEFINE VARIABLE hXtext        AS HANDLE.
DEFINE VARIABLE cPassword     AS CHARACTER INIT ?.
DEFINE VARIABLE g_header      AS HANDLE.

/* Create the Web service server object */
CREATE SERVER hWebSrvc.

/* Connect to the WS */
hWebSrvc:CONNECT("-WSDL
                  http://ServiceHost/SOAPHeader/HeaderExample.asmx?wsdl").

/* Get the method, set the port type */
RUN HeadersSoap SET hPortType ON hWebSrvc.
/*2*/
/* Associate the req. & resp. callbacks with the port type */
hPortType:SET-CALLBACK-PROCEDURE("REQUEST-HEADER", "ReqHandler").
hPortType:SET-CALLBACK-PROCEDURE("RESPONSE-HEADER", "RespHandler").
/*3*/
/* Invoke the Web service with no header and display the results */
RUN OpenAccess IN hPortType (OUTPUT cResponse).
DISPLAY cResponse LABEL "WS response" WITH FRAME aaa.
/*4*/
/* Go again with the AccessID set from previous response */
cResponse = "".
RUN HelloWorld IN hPortType (OUTPUT cResponse).
DISPLAY cResponse LABEL "WS response" WITH FRAME bbb.
/*5*/
/* Go again with the AccessID set from previous response */
/* header together with an added Username and Password   */
cResponse = "".
cPassword = "Administrator".
RUN HelloSecureWorld IN hPortType (OUTPUT cResponse).
DISPLAY cResponse LABEL "WS response" WITH FRAME ccc.
```

```
/*6*/
DELETE OBJECT g_header.
DELETE OBJECT hPortType.
hWebSrvc:DISCONNECT().
DELETE OBJECT hWebSrvc.

/**************** Internal Procedures ***************/
```

4. Runs the `HelloWorld` procedure to invoke the next Web service operation, passing back the SOAP response header to the Web service unchanged as the SOAP request header (see Request header handler for reusing and modifying a header on page 34).
5. Runs the `HelloSecureWorld` procedure to invoke the next Web service operation, passing back the password-modified SOAP response header as the Web service SOAP request header (see Request header handler for reusing and modifying a header on page 34).

   Note: The header handler processing for Step 4 is different from Step 5, to reflect that the initial SOAP response header is unchanged for one request and modified for the next.

6. Cleans up the global objects maintained in its context and disconnects from the Web service. Note that one of the objects it deletes is the original SOAP response header saved by the response header handler during execution of the `OpenAccess` procedure.

Request header handler for reusing and modifying a header

This is the SOAP request header handler (`ReqHandler`) that reuses the initial SOAP response header and adds a `Password` value to the existing `AuthHead` header entry to pass along with the `AccessID` value between the client and Web service:

Request header handler modifying a saved SOAP response header

```
PROCEDURE ReqHandler:   /*1*/
  DEFINE OUTPUT PARAMETER hHeader       AS HANDLE.
  DEFINE INPUT  PARAMETER cNamespace    AS CHARACTER.
  DEFINE INPUT  PARAMETER cLocalNS      AS CHARACTER.
  DEFINE OUTPUT PARAMETER lDeleteOnDone AS LOGICAL.

  /* The IF test determines if this is the first call through this code. If
    it is, then g_header is not set and hHeader is set to ? to ensure that no
    header is sent. g_header gets set when the response header is returned,
    so a subsequent pass through this code takes the previous response header
    and sends it as the current request header, possibly modified to
    authenticate a secure request. */

  IF NOT VALID-HANDLE (g_header) THEN DO:   /*2a*/ /* first request */
    hHeader = ?.
    lDeleteOnDone = TRUE.
  END.
  ELSE DO:   /*2b*/ /* all subsequent requests */
    hHeader = g_header.
    lDeleteOnDone = FALSE.
    /* Password node data are added to the existing SOAP header if a secure
      operation is being executed */
    IF cPassword <> ? THEN DO: /*3*/
      /*3a*/  /* Build a new x-doc to contain the Password data */
      CREATE X-DOCUMENT hXdoc.
      CREATE X-NODEREF  hXnoderef1.
      CREATE X-NODEREF  hXnoderef2.
      CREATE X-NODEREF  hXtext.

      /* Add the Password data as a child of the existing header entry */
      DEFINE VARIABLE hHeaderEntryref AS HANDLE.   /*3b*/
      DEFINE VARIABLE ClientNS AS CHARACTER
        INIT "http://ServiceHost/SOAPHeader".
      CREATE SOAP-HEADER-ENTRYREF hHeaderEntryref.

      hHeader:GET-HEADER-ENTRY(hHeaderEntryref, 1).   /*3c*/
      hHeaderEntryref:GET-NODE(hXnoderef1).

      hXdoc:IMPORT-NODE(hXnoderef2, hXnoderef1, TRUE).   /*3d*/
       IF hXnoderef2:NUM-CHILDREN > 1 THEN DO: /* Get rid of previous */
        hXnoderef2:GET-CHILD(hXnoderef1,2). /* Password data     */
        hXnoderef1:DELETE-NODE().
       END.

       hXdoc:APPEND-CHILD(hXnoderef2).   /*3e*/
       hXdoc:CREATE-NODE-NAMESPACE(hXnoderef1, ClientNS, "Password",
                                   "ELEMENT").

       hXnoderef2:APPEND-CHILD(hXnoderef1).   /*3f*/
      hXdoc:CREATE-NODE(hXtext, "", "text").
       hXnoderef1:APPEND-CHILD(hXtext).
      hXtext:NODE-VALUE = cPassword.

      /* Replace the existing header entry using a deep copy of the new version
        from the x-doc that has the Password node added */
      hHeaderEntryref:SET-NODE(hXnoderef2).   /*3g*/

      /* Procedure/header cleanup */
      cPassword = ?.   /*3h*/  /* Use current password until replaced */
      DELETE OBJECT hHeaderEntryref.
      DELETE OBJECT hXdoc.
      DELETE OBJECT hXnoderef1.
      DELETE OBJECT hXnoderef2.
      DELETE OBJECT hXtext.
    END.
  END. /* all subsequent requests */
END PROCEDURE.
```

The code in the preceding example:

1. Sends the SOAP request header for the `HelloWorld` and `HelloSecureWorld` requests (and any request run after running `OpenAccess`).
2. Tests if the global header handle (`g_header`) references a valid object, and:
   a. If it does not reference an object, the request handler must be running as part of the initial call to `OpenAccess` and sets the output parameters to ensure that no initial SOAP request header is sent
   b. If it does reference an object, the handler passes the global header object as output using the request header parameter (`hHeader`) and ensures that the object is not deleted (saving it for use in any further request)
3. Tests if a password has been specified for the current Web service request, indicated by any `cPassword` value that is not the `Unknown value (?)`. If the current Web service request is nonsecure (as with the `HelloWorld` operation), all work has been done and the request handler can end. If the current Web service request is secure (as with the `HelloSecureWorld` operation), the request handler adds the password information to the SOAP request header, as follows:

   Note: After the first secure request, all future requests (secure or nonsecure) send a request header that includes password information because the password information is never deleted until replaced by a newly-specified password.

   a. Creates the XML x-document and x-noderef objects to manipulate the SOAP header
   b. Creates the SOAP header entryref object (`hHeaderEntryref`) to access SOAP header entries and defines the namespace (`ClientNS`) used for defining the SOAP header entry for this request
   c. Returns the existing header entry from the saved global SOAP header object using the `GET-HEADER-ENTRY( )` method on the SOAP header object and accesses the XML root node of the entry using the `GET-NODE( )` method on the SOAP header entryref object

      Note: The handler is adding the password information to an existing SOAP header entry. If it was adding a new header entry to hold the information, it would invoke the `ADD-HEADER-ENTRY( )` method to add the header entry to contain the new XML for it.

   d. Imports the header entry root node into an x-document object in order to access and modify the XML for the header entry, and also deletes any password data from a previous secure request before adding the currently-specified password data
   e. Adds the <Password> element as a sibling of the <AccessID> element
   f. Adds the <Password> element value
   g. Replaces the entire existing header entry in the global SOAP header object with the header entry updated in the x-document object
   h. Sets the password value (`cPassword`) to unknown (?), which retains the current password in the header entry until it is explicitly changed in `cPassword`, then deletes all of the helper XML and SOAP header entryref objects created in the header handler

Using a client-created SOAP request header

This example shows how you might create a SOAP header internally to use as an initial SOAP request header, as opposed to recycling a header previously received in a SOAP response message (described in previous examples). This is an example of the header created by the client for the Web service:

SOAP request header created entirely by the client

```
<SOAP-ENV:Envelope
  . . .
    <SOAP-ENV:Header>
      <ns0:AuthHeader xmlns:ns0="http://ServiceHost/SOAPHeader">
        <ns0:UserName>Scott</ns0:UserName>
        <ns0:Password>Administrator</ns0:Password>
      </ns0:AuthHeader>
    </SOAP-ENV:Header>
  . . .
</SOAP-ENV:Envelope>
```

The client creates a header very similar to the headers described in the previous examples. The `<UserName>` and `<Password>` elements, in this case, provide user authentication for each Web service request. The Web service requires this authentication for every Web service request.

The following code is the mainline of a procedure that invokes the Web service to create the initial SOAP request header containing a username and password node. This code:

1. Defines several mainline variables, including handles to access the global SOAP header (`g_header`) created for requests and its XML, and variables to hold the username (`cUsername`) and password (`cPassword`) values.
2. Builds the global request header used for all requests (see Procedure to create a SOAP request header on page 39).
3. Registers the request header (`ReqHandler`) handler after connecting to the Web service and instantiating the `HeaderSoap` port type procedure object.
4. Runs the `HelloMyWorld` procedure to invoke a Web service operation, passing back the global SOAP request header created by the client (see Request header handler for passing a globally-created header object on page 41).
5. Cleans up the global objects maintained in its context and disconnects from the Web service. Note that one of the objects it deletes is the global SOAP request header create by the client.

Invoking a request that creates a SOAP request header on the client

```
/* SOAPHeader3.p
  Calls a ficticious web service, passes it a username and password through
  a SOAP message request header, and gets back a string. The Web service has
  only one service and port available. */

/*1*/
/* Define local variables */
DEFINE VARIABLE hWebSrvc      AS HANDLE.
DEFINE VARIABLE hPortType     AS HANDLE.
DEFINE VARIABLE cUsername     AS CHARACTER INIT "Scott".
DEFINE VARIABLE cPassword     AS CHARACTER INIT "Administrator" .
DEFINE VARIABLE cResponse     AS CHARACTER FORMAT "x(72)".
```

```
DEFINE VARIABLE hXdoc         AS HANDLE.
DEFINE VARIABLE hXnoderef1    AS HANDLE.
DEFINE VARIABLE hXnoderef2    AS HANDLE.
DEFINE VARIABLE hXAttribute   AS HANDLE.
DEFINE VARIABLE hXtext        AS HANDLE.
DEFINE VARIABLE g_header      AS HANDLE.
/*2*/
/* Build global SOAP request header */
RUN BuildRequestHeader (OUTPUT g_header).

/* Create the Web service server object */
CREATE SERVER hWebSrvc.

/* Connect to the WS */
hWebSrvc:CONNECT("-WSDL
                    http://ServiceHost/SOAPHeader/HeaderExample.asmx?wsdl").

/* Get the method, set the port type */
RUN HeaderSoap SET hPortType ON hWebSrvc.
/*3*/
/* Associate the request callback with the port type */
hPortType:SET-CALLBACK-PROCEDURE("REQUEST-HEADER", "ReqHandler").
/*4*/
/* Invoke the web service and display the results */
RUN HellowMyWorld IN hPortType (OUTPUT cResponse).
DISPLAY cResponse LABEL "WS Response" WITH FRAME aaa.
/*5*/
DELETE OBJECT g_header.
DELETE OBJECT hPortType.
hWebSrvc:DISCONNECT().
DELETE OBJECT hWebSrvc.

/*************** Internal Procedures ***************/
```

Procedure to create a SOAP request header

This is the procedure (`BuildRequestHeader`) that creates the global SOAP request header that the client sends in all requests to the Web service:

Procedure to create a SOAP request header

```
/*1*/
PROCEDURE BuildRequestHeader:
  /* Define procedure parameter */
  DEFINE OUTPUT PARAMETER hHeader AS HANDLE.
  /*2*/
  DEFINE VARIABLE hHeaderEntryref AS HANDLE.
  DEFINE VARIABLE ClientNS AS CHARACTER
    INIT "http://ServiceHost/SOAPHeader".
  /*3*/
  /* Create SOAP header and server objects */
    CREATE SOAP-HEADER hHeader.
    CREATE SOAP-HEADER-ENTRYREF hHeaderEntryref.
  /*4*/
  /* Create x-doc objects to build header */
    CREATE X-DOCUMENT hXdoc.
    CREATE X-NODEREF hXAttribute.
    CREATE X-NODEREF hXnoderef1.
    CREATE X-NODEREF hXnoderef2.
    CREATE X-NODEREF hXtext.
  /*5*/
  /* Create the header entry */
    hHeader:ADD-HEADER-ENTRY(hHeaderEntryref).
```

```
  /*6*/
  /* Create the header namespace data */
  hXdoc:CREATE-NODE-NAMESPACE(hXnoderef1, ClientNS, "AuthHeader",
                              "ELEMENT").
  hXdoc:CREATE-NODE-NAMESPACE(hXAttribute, "http://www.w3.org/2000/xmlns/",
                              "xmlns", "ATTRIBUTE").
 hXAttribute:NODE-VALUE = ClientNS.
  hXnoderef1:SET-ATTRIBUTE-NODE(hXAttribute).
  hXdoc:INSERT-BEFORE(hXnoderef1, ?).
  /*7*/
  /* Create the Username/Password data */
 hXdoc:CREATE-NODE-NAMESPACE(hXnoderef2, ClientNS, "UserName", "ELEMENT").
  hXnoderef1:APPEND-CHILD(hXnoderef2).
  hXdoc:CREATE-NODE(hXtext,"","text").
  hXnoderef2:APPEND-CHILD(hXtext).
 hXtext:NODE-VALUE = cUsername.
  /*8*/
 hXdoc:CREATE-NODE-NAMESPACE(hXnoderef2, ClientNS, "Password", "ELEMENT").
  hXnoderef1:APPEND-CHILD(hXnoderef2).
  hXdoc:CREATE-NODE(hXtext, "", "text").
  hXnoderef2:APPEND-CHILD(hXtext).
 hXtext:NODE-VALUE = cPassword.
  /*9*/
  /* Fill the header entry using a deep copy */
 hHeaderEntryref:SET-NODE(hXnoderef1).
  /*9*/
  /* Fill the header entry using a deep copy */
 hHeaderEntryref:SET-NODE(hXnoderef1).
  /*10*/
  /* Procedure/header cleanup */
 DELETE OBJECT hXdoc.
 DELETE OBJECT hXAttribute.
 DELETE OBJECT hXnoderef1.
 DELETE OBJECT hXnoderef2.
 DELETE OBJECT hXtext.
 DELETE OBJECT hHeaderEntryref.
END PROCEDURE.
```

The code in the preceding example:

1. Defines a single output parameter to return the global SOAP header object that it creates to the client mainline context
2. Defines a handle variable (`hHeaderEntryref`) to reference the SOAP header entryref object, and a variable (`ClientNS`) that specifies the namespace for the SOAP header entry that it creates
3. Creates the global SOAP header object that references the XML for the header using the `CREATE SOAP-HEADER` statement, and creates the SOAP header entryref object that references the XML for the header entry using the `CREATE SOAP-HEADER-ENTRYREF` statement
4. Creates the x-document and x-noderef objects required to build the XML to be added as the header entry for the global SOAP header
5. Adds a header entry to the newly created SOAP header object, referenced by the `hHeaderEntryref` object, using the `ADD-HEADER-ENTRY( )` method
6. Creates the root node (`<AuthHeader>` element) for the header entry in the working x-document object

   Note: The namespace attribute specifying `http://www.w3.org/2000/xmlns/` is a requirement of the DOM. For more information, see the information on XML support in *OpenEdge Development: Programming Interfaces.*

7. Creates the `<UserName>` element as a child node of the header entry
8. Creates the `<Password>` element as a second child node of the header entry
9. Assigns the header entry in the global SOAP header object to the XML for the `<AuthHeader>` element of the x-document object using the `SET-NODE( )` method

   Note: You must call the `ADD-HEADER-ENTRY( )` method before calling the `SET-NODE( )` method to populate an entry. If you do not, you are essentially overwriting the XML of the same entry over and over.

10. Cleans up by deleting all the helper objects created in the procedure before returning to the client mainline

Request header handler for passing a globally-created header object

This is the SOAP request header handler (`ReqHandler`) that passes the global SOAP request header created by the client and the Web service for each request:

Request header handler passing a client-created SOAP request header

```
PROCEDURE ReqHandler: /*1*/
  /* Define procedure parameters */
  DEFINE OUTPUT PARAMETER hHeader       AS HANDLE.
  DEFINE INPUT  PARAMETER cNamespace    AS CHARACTER.
  DEFINE INPUT  PARAMETER cLocalNS      AS CHARACTER.
  DEFINE OUTPUT PARAMETER lDeleteOnDone AS LOGICAL.
  /*2*/
  /* Pass in global header reused for every request */
  hHeader = g_header.
  lDeleteOnDone = FALSE.
END PROCEDURE.
```

The code in the preceding example:

1. Sends the SOAP request header for the `HelloMyWorld` request (and any subsequent request).
2. Passes the global SOAP header the request header output parameter and ensures that it is not deleted until the client mainline has finished with it.

Managing memory for SOAP headers

SOAP headers require two types of memory in ABL (see SOAP header object model on page 19):

- Memory used for the SOAP header and SOAP header entry data that SOAP header and SOAP header-entryref objects reference, that is, the underlying XML.
- Memory used by the SOAP header object, including the memory for SOAP header entries referenced by SOAP header-entryref objects, even if this header object memory does not reference any underlying XML.

SOAP header object model and DOM relationships

The manner in which a SOAP header object encapsulates the XML for a SOAP header is similar to how the ABL Document Object Model (DOM) allows an x-document object to encapsulate the XML of a general XML document. The manner in which a SOAP header-entryref object references a SOAP header entry is similar to how the ABL DOM allows an x-noderef object to reference the individual elements and attributes of a general XML document. Similar dependencies exist for managing the memory for objects in each reference model.

A SOAP header entry (XML sub-tree) must be associated with the SOAP header-entryref object handle. None of the object's methods (see Defining header handlers on page 23) can be called successfully unless there is an underlying SOAP header entry.

SOAP header objects can be created in the ABL application in two ways:

- Implicitly by OpenEdge for the header of an incoming SOAP response message
- Explicitly by the application using a variation of the `CREATE` statement

No matter how the application creates these objects, the application is responsible for explicitly deleting these and any other objects that it creates during a session, including the SOAP header-entryref objects used to reference the header entries in a SOAP header object and any X-document and X-noderef objects used to parse and build the XML structures that occupy memory for the underlying XML.

Memory for the SOAP header object model and DOM

The following table summarizes the available ABL elements and how they can help you to manage memory for SOAP headers.

Table 1: ABL to manage object memory for SOAP headers

ABL element	Applied to object	Deletes the ABL object	Deletes all underlying XML DOM objects
Setting the *lDeleteOnDone* OUTPUT parameter of the request header handler to TRUE (see Defining header handlers on page 23)	SOAP header object	Yes	Yes

ABL element	Applied to object	Deletes the ABL object	Deletes all underlying XML DOM objects
DELETE OBJECT statement	SOAP header object	Yes	Yes
	X-document object	Yes	Yes
	SOAP header-entryref object	Yes	No
	X-noderef object	Yes	No
DELETE-HEADER-ENTRY( ) method	SOAP header-entryref object	No	Yes
DELETE-NODE( ) method	X-noderef object	No	Yes[1]

Caution: Be sure that you always delete the underlying XML for a SOAP header-entryref object before you delete the SOAP header-entryref object itself. If you lose all reference to the underlying XML before deleting it, its memory becomes lost to your application. If this occurs as part of an iterative process, it represents a memory leak that could cause your application to crash.

Attributes and methods for handling SOAP headers

As described in previous sections, the two ABL objects provided for managing SOAP headers include:

- **SOAP header object** — For accessing the name and header entries of a SOAP header
- **SOAP header-entryref object** — For accessing each header entry in a SOAP header

Each of these objects provide attributes and methods that manage header access according to a SOAP header object model defined for ABL. This object model maps to the SOAP header XML in a manner analogous to the way x-document and x-noderef objects allow you to access XML through the Document Object Model (DOM) supported in ABL. In fact, the attributes and methods that implement the SOAP header object model provide access to the XML of a header by making it available to the standard XML support in ABL, such as the DOM, and by allowing the XML to be exchanged between the two SOAP header representations (SOAP object model and XML).

For more information on the SOAP header object model, see SOAP header object model on page 19. The following sections describe all of the attributes and methods available for each object in the SOAP header object model.

SOAP header object attributes and methods

The following table briefly describes all of the attributes on the SOAP header object that are unique to the object or have special application to the SOAP header object.

[1] This includes any child sub-trees of the deleted X-noderef object.

Table 2: SOAP header object attributes

Attribute	Type	Description
NAME	CHARACTER	Returns the qualified name of the SOAP header ("*namespacePrefix*:HEADER")
NUM-HEADER-ENTRIES	INTEGER	Returns the number of entries attached to the SOAP header
TYPE	CHARACTER	Returns the handle type, "SOAP-HEADER"

The following table briefly describes all of the methods on the SOAP header object.

Table 3: SOAP header object methods

Method	Type	Description
ADD-HEADER-ENTRY (*hHeaderEntryRef*)	LOGICAL	Adds a new entry to its list of pointers and associates it with the existing Header-EntryRef object specified. A specified SOAP header-entryref object handle (*hHeaderEntryRef*) references the new header entry.
GET-HEADER-ENTRY (*hHeaderEntryRef*, *index*)	LOGICAL	Associates a specified SOAP header-entryref object handle (*hHeaderEntryRef*) with an index-specified entry (*index*) in the associated SOAP header.

For more information on these attributes and methods, see *OpenEdge Development: ABL Reference.*

SOAP header-entryref object attributes and methods

The following table briefly describes all of the attributes on the SOAP header-entryref object that are unique to the object or have special application to the SOAP header-entryref object.

Note: As indicated in Table 4: SOAP header-entryref object attributes on page 44 and Table 5: SOAP header-entryref object methods on page 45, certain members are associated with a specific SOAP version. To determine which SOAP version is in use, use the SOAP-VERSION attribute of the server object handle.

Table 4: SOAP header-entryref object attributes

Attribute	Type	Description
ACTOR	CHARACTER	(SOAP Version 1.1 only) Returns the value of the actor attribute specified in the associated SOAP header entry, which identifies the recipient of the message. Replaced in SOAP Version 1.2 by the ROLE attribute, and assumes the value of that attribute if used in the context of a SOAP 1.2 connection.
LOCAL-NAME	CHARACTER	Returns the unqualified part of name specified for the associated SOAP header entry element.

Attribute	Type	Description
MUST-UNDERSTAND	LOGICAL	Returns the value of the mustUnderstand attribute specified in the associated SOAP header entry.
NAME	CHARACTER	Returns the qualified name of the SOAP header entry ("*namespacePrefix*:*localName*").
NAMESPACE-URI	CHARACTER	Returns the namespace URI prefixed to the associated SOAP header entry element's name.
ROLE	CHARACTER	(SOAP Version 1.2 only) Returns the value of the role attribute specified in the associated SOAP header entry, which identifies the recipient of the message. Replaces the ACTOR attribute used in SOAP Version 1.1.
TYPE	CHARACTER	Returns the handle type, "SOAP-HEADER-ENTRYREF".

The following table briefly describes all of the methods on the SOAP header-entryref object.

Table 5: SOAP header-entryref object methods

Method	Type	Description
DELETE-HEADER-ENTRY ( )	LOGICAL	Deletes the underlying SOAP header entry and all of its content, but does not delete the SOAP header-entryref object used to reference the deleted header entry.
GET-NODE (*handle*)	LOGICAL	Returns a handle (*handle*) to an X-noderef object that references the root node of a DOM tree containing the parsed XML for the underlying SOAP header entry.
GET-SERIALIZED ( )	LONGCHAR	Returns the XML for the underlying SOAP header entry in serialized form.
SET-ACTOR (*string*)	LOGICAL	(SOAP Version 1.1 only) Sets the value (*string*) of the actor attribute in the underlying SOAP header entry. With SOAP Version 1.2, use SET-ROLE.
SET-MUST-UNDERSTAND (*logical*)	LOGICAL	Sets the value (*logical*) of the mustUnderstand attribute in the underlying SOAP header entry.
SET-NODE (*hXnoderef*)	LOGICAL	Replaces the header entry referenced by this SOAP header-entryref object with a specified DOM sub-tree (parsed XML) that is assumed to represent a SOAP header entry element. The method performs a deep copy of the XML sub-tree specified by the X-noderef object handle (*hXnoderef*).

Method	Type	Description
SET-ROLE (*string*)	LOGICAL	(SOAP Version 1.2 only) Sets the value (*string*) of the role attribute in the underlying SOAP header entry. With SOAP Version 1.1, use SET-ACTOR.
SET-SERIALIZED (*longchar*)	LOGICAL	Replaces the header entry referenced by this SOAP header-entryref object with serialized XML (*longchar*) that is then parsed into a DOM sub-tree that represents a SOAP header entry element. The XML is assumed to be valid for a SOAP header entry.

Note: All of the methods and functions listed in Table 5: SOAP header-entryref object methods on page 45 return TRUE only if there is an underlying SOAP header entry (XML sub-tree) associated with the object handle.

For more information on these attributes and methods, see *OpenEdge Development: ABL Reference*.

3

Handling Errors in ABL Requests to OpenEdge SOAP Web Services

As described previously, the ERROR condition can result from a Web service request by:

1. A SOAP fault returned by the Web service
2. An ABL internal error raised by OpenEdge
3. An application error intentionally raised by the ABL application (using the RETURN ERROR statement) in a SOAP request header callback procedure or a SOAP response header callback procedure

The following chapter sections focus on how to handle SOAP faults and generally how to debug Web service requests.

For details, see the following topics:

- Handling SOAP faults
- Detecting a SOAP fault
- Managing a SOAP fault
- Examples of ABL accessing a SOAP fault
- Debugging ABL applications that call Web services

Handling SOAP faults

When a Web service operation fails after the request arrives at the Web service, it might generate a SOAP fault. This is a special SOAP response message that contains a single fault element in the body, and a namespace qualified and usually identified by a <SOAP:Fault> or <SOAP-ENV:Fault> tag. This is a typical SOAP fault message, with the most common SOAP fault elements:

Sample SOAP fault message

```
<SOAP-ENV:Body>
  <SOAP-ENV:Fault>
    <faultcode>VersionMismatch.Additionalinfo</faultcode>
    <faultstring>The message does not conform to expected
                 version</faultstring>
    <faultactor>http://www.stockvendor.com</faultactor>
    <detail>
      <e:badver xmlns:e=http://www.stockvendor.com/>
        <message>Expect version 2 request, received previous
                 version</message>
        <errorcode>523</errorcode>
      </e:badver>
    </detail>
  </SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

These three child elements of this SOAP fault are mostly application-defined, but typically provide the following information:

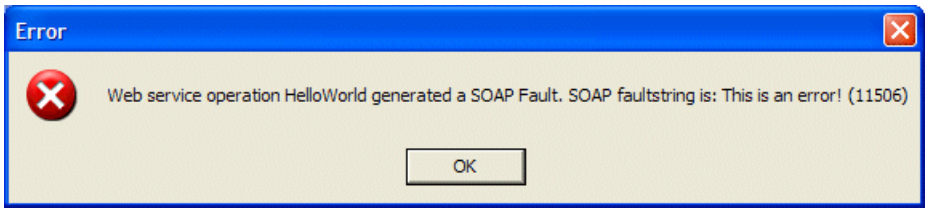
- **<faultcode>** — Coded fault identifier, such as an error number or specially formatted string to identify the fault.
- **<faultstring>** — Single string value containing a simple description of the fault.
- **<faultactor>** — (Optional) Single string value identifying the Universal Resource Identier (URI) of the Web service that caused the fault. This is less common, as it is usually apparent what Web service has generated the SOAP fault.
- **<detail>** — (Optional) Contains elements completely defined by the Web service, but which provide much more specific information about the fault.

Detecting a SOAP fault

ABL has two error handling models: traditional error handling and structured error handling. The sections below describe how to handle a SOAP fault using each model. For more complete information on error handling, see *OpenEdge Development: Error Handling*.

SOAP faults with traditional error handling

When the Web service returns a SOAP fault, OpenEdge responds depending on how the request is coded. As with ABL errors, if you do not specify the NO-ERROR option on a RUN statement that invokes a Web service operation or if the operation is invoked using a user-defined function, OpenEdge converts the SOAP fault into a standard ABL error message displayed as follows:



The error box displays the operation that generated the SOAP fault and the contents of the `<faultstring>` element in the SOAP fault message.

If the statement that invokes a Web service operation traps an error using the `NO-ERROR` option, this message and the information for any SOAP fault detail appears in the `ERROR-STATUS` system handle, and as with all ABL errors remains available until the next statement is invoked using the `NO-ERROR` option. The following table lists the `ERROR-STATUS` handle attribute that help you to detect and return the information for a SOAP fault. All other attributes and methods on this handle work as they do for any ABL error condition.

Table 6: ERROR-STATUS handle attributes

Attribute	Type	Description
`ERROR`	`LOGICAL`	Indicates that an `ERROR` condition was returned as a result of processing a Web service request
`ERROR-OBJECT-DETAIL`	`HANDLE`	References the SOAP-fault object for any Web service request that returns a SOAP fault trapped using ABL `NO-ERROR` option

Once you have detected an error (`ERROR-STATUS:ERROR=TRUE`), you can determine if it is the result of a SOAP fault using the `ERROR-OBJECT-DETAIL` attribute. If a SOAP fault caused the error, this attribute returns a handle to a SOAP-fault object containing the SOAP fault information. Otherwise, it has the `Unknown value (?)`.

Note that Web service operations that are invoked as user-defined functions **do not raise** the `ERROR` condition. For a SOAP fault resulting from an operation invoked as a user-defined function, `ERROR-STATUS:ERROR` is `FALSE`. But, in this case, `ERROR-STATUS:NUM-MESSAGES` is greater than zero.

SOAP faults with structured error handling

You can also trap the error using a `CATCH` block, which is the error handler for the ABL structured error handling model. In this scenario, the AVM generates an error object based on the built-in `Progress.Lang.SoapFaultError` class. The `SoapFault` property of the class contains a handle to the built-in SOAP-fault object. Recall that the SOAP-fault object is the ABL representation of a SOAP fault. Thus, `SoapFaultError` error object is a wrapper for the same SOAP-fault object you could also have accessed using the `NO-ERROR` option and the `ERROR-OBJECT-DETAIL` attribute of the `ERROR-STATUS` system handle.

This is the basic structure of a block with a CATCH block:

```
DO ON ERROR UNDO, THROW:
  /* Web service call, do not use the NO-ERROR option. */
  CATCH mySoapErrorObject AS Progress.Lang.SoapFaultError:
    /* Access and interrogate the SOAP-fault object wrapped by the error
       object. */
  END CATCH.
END /* DO */
```

Examples of ABL accessing a SOAP fault on page 53 provides more detailed information.

Structured error handling represents errors as objects. ABL includes a hierarchy of classes that allow all error types to be represented as objects. The following table summarizes the class representing all system errors and its subclass that represents SOAP errors.

Table 7: System error classes

Class	Members	Description
Progress.Lang.SysError	Inherits the Progress.Lang.ProError members: • CallStack property • NumMessages property • Severity property • GetMessage( ) method • GetMessageNum( ) method	A subclass of Progress.Lang.ProError that represents (with its subclasses) all ABL system errors
Progress.Lang.SoapFaultError	Inherits the Progress.Lang.ProError members and adds: • SoapFault	A subclass of Progress.Lang.SysError that contains a reference to a Web service SoapFault object

The following table describes the properties and methods of the Progress.Lang.SoapFaultError class.

Table 8: SoapFaultError class members

Member	Description
CallStack property	Returns the contents of the call stack at the time the error object was thrown as a string. If the ERROR-STACK-TRACE attribute of the SESSION handle is false, then this property returns the Unknown value (?). To enable the call stack, set SESSION:ERROR-STACK-TRACE property to TRUE directly, or use the -errorstack session startup parameter.

Member	Description
`NumMessages` property	In ABL, an error is represented as a pair of values. The message number is a unique number identifying the particular error. The error message is a string which describes the error. This property indicates how many error number and error message the error object contains.
`Severity` property	The `Severity` property is not used by ABL system errors. It is provided as a mechanism for you to use to assign severity rankings to your various application errors (`Progress.Lang.AppError`).
`GetMessage(` *`MessageIndex`* `)` method	Returns the error message for the indexed error in the error object, beginning with one (1). If there is no error message at the indicated index, the method returns the empty string.
`GetMessageNum(` *`MessageIndex`* `)` method	Returns the error message number associated with the indexed error in the error object. For `Progress.Lang.SysError` objects and subclasses, the method returns the Progress message number for the system generated error. If there is no error message at the index, the method returns the empty string.
`SoapFault`	Identifies the `SOAP-FAULT` object handle that contains a SOAP fault message detail. If the ABL application invokes a Web service operation that returns a SOAP fault message, the AVM creates a `SOAP-FAULT` object. Use the `SOAP-FAULT-DETAIL` attribute of the `SOAP-FAULT` object handle to access the SOAP fault message detail. see Managing a SOAP fault on page 51 for more information about the `SOAP-FAULT` object.

Managing a SOAP fault

On detecting a SOAP fault, OpenEdge automatically creates a SOAP-fault object for it. This object contains information from each child element of the response message `<Fault>` element. Like the `ERROR-STATUS` handle information, OpenEdge makes this object available only until the next statement executes with the `NO-ERROR` option.

Note: As indicated in the following table, certain members are associated with a specific SOAP version. To determine which SOAP version is in use, use the `SOAP-VERSION` attribute of the server object handle.

The following table lists the attributes on the SOAP-fault object, which contains no methods.

Table 9: SOAP-fault object attributes

Attribute	Type	Description
SOAP-FAULT-ACTOR	CHARACTER	(SOAP Version 1.1 only) Returns the value of the <faultactor> element of the SOAP fault message, which is a URI that identifies the Web service returning the fault. Replaced in SOAP Version 1.2 by the combination of SOAP-FAULT-ROLE and SOAP-FAULT-NODE.
SOAP-FAULT-CODE	CHARACTER	Returns the value of the <faultcode> element of the SOAP fault message, which identifies the fault.
SOAP-FAULT-DETAIL	HANDLE	References the SOAP fault-detail object, which contains more application-specific error information.
SOAP-FAULT-MISUNDERSTOOD- HEADER	CHARACTER	(SOAP 1.2 only) Returns the list of fully qualified names of SOAP headers, if any, for which mandatory processing (as designated by the SOAP-ENV:mustUnderstand attribute) failed.
SOAP-FAULT-NODE	CHARACTER	(SOAP 1.2 only) Returns the URI of the SOAP node that generated the SOAP-fault object, if available. In combination with SOAP-FAULT-ROLE, replaces the SOAP-FAULT-ACTOR attribute used in SOAP Version 1.1.
SOAP-FAULT-ROLE	CHARACTER	(SOAP 1.2 only) Returns the URI that identifies the role in which the node that generated the SOAP-fault object was operating when the fault occurred. In combination with SOAP-FAULT-NODE, replaces the SOAP-FAULT-ACTOR attribute used in SOAP Version 1.1.
SOAP-FAULT-STRING	CHARACTER	Returns the value of the <faultstring> element of the SOAP fault message, which provides a human-readable description of the fault.
SOAP-FAULT-SUBCODE	CHARACTER	(SOAP 1.2 only) Returns the list of fully qualified sub-code names for the SOAP-fault object.
TYPE	CHARACTER	Returns the handle type, "SOAP-FAULT".

As you can see, these attributes provide access to all the elements of a SOAP fault you might encounter.

Because the <detail> element has essentially no standard definition, and can contain any elements that the Web service chooses to generate, OpenEdge provides another object, the SOAP fault-detail object, to return this information to the ABL application. If the SOAP fault message contains a <detail> element, the SOAP-FAULT-DETAIL attribute on the SOAP-fault object handle returns a handle to the SOAP fault-detail object that is generated for it. Otherwise, this attribute returns the Unknown value (?). As is the case with the SOAP-fault object, OpenEdge makes the SOAP fault-detail object available only until the next ABL statement that executes with the NO-ERROR option.

The following table describes the single attribute of the SOAP fault-detail object:

Table 10: SOAP fault-detail object attributes

Attribute	Type	Description
TYPE	CHARACTER	Returns the handle type, "SOAP-FAULT-DETAIL"

The following table lists the methods of the SOAP fault-detail object.

Table 11: SOAP fault-detail object methods

Method	Type	Description
GET-NODE( )	LOGICAL	Returns a handle to an X-noderef object that references the root node (SOAP <detail> element) of a DOM tree containing the parsed XML for the underlying SOAP detail information
GET-SERIALIZED( )	LONGCHAR	Returns the XML for the underlying SOAP fault detail information in serialized form

The GET-NODE( ) and GET-SERIALIZED( ) methods provide access to the elements of the SOAP fault detail information in exactly the same way as they provide access to SOAP header entries for a SOAP header. For more information, see Handling SOAP Message Headers in ABL.

The general approach to managing SOAP fault detail elements is identical to retrieving and scanning the header entries of a SOAP response header. The structure of elements that make up the SOAP fault detail information is completely undefined. For more information, see the documentation available for the Web service you are accessing.

As with SOAP headers, if the WSDL Analyzer can identify a temp-table or ProDataSet definition that maps to the SOAP fault detail, you can use the GET-SERIALIZED( ) method in conjunction with the READ-XML( ) method of the documented ABL object (temp-table or ProDataSet) to access the SOAP fault detail data. However, this is unlikely, and you most often must access the data using a DOM tree or the ABL SAX reader. For more information on how the Analyzer might identify a temp-table or ProDataSet to access the SOAP fault detail, see WSDL Details. For more information on the ABL DOM, SAX, and temp-table/ProDataSet XML features, see Use XML with ABL Applications in the *OpenEdge Development:Use XML with ABL Applications* guide.

Examples of ABL accessing a SOAP fault

The first two examples run an operation on a fictitious Web service that returns a SOAP fault identical to the one described in the beginning of this section (see the Handling SOAP faults on page 48). They access and examine the SOAP fault message as described in the following steps. The first example uses traditional error handling, and the second uses structured error handling.

The third example shows how to access and examine a class-based error object returned from an asynchronous call.

Traditional error handling example

The code in the following example:

1. Uses the ABL VALID-HANDLE function to determine if a given ERROR condition (ERROR-STATUS:ERROR = TRUE) is caused by a SOAP fault by testing the validity of the handle returned by the ERROR-STATUS:ERROR-OBJECT-DETAIL attribute
2. Assigns a handle variable (hSoapFault) to any valid SOAP-fault object returned by the ERROR-STATUS:ERROR-OBJECT-DETAIL attribute for code readability

   Note: You can also use the ERROR-STATUS:ERROR-OBJECT-DETAIL handle attribute directly to work with the SOAP-fault object.

3. Examines the values of SOAP fault elements, as required, using appropriate attributes (SOAP-FAULT-CODE) on the SOAP-fault object handle
4. Uses the ABL VALID-HANDLE function to determine if this SOAP fault has SOAP fault detail by testing the validity of the handle returned by hSoapFault:SOAP-FAULT-DETAIL
5. Assigns a handle variable (hSoapFaultDetail) to the SOAP fault-detail object returned by the hSoapFault:SOAP-FAULT-DETAIL attribute for code readability

   Note: You can also use the ERROR-STATUS:ERROR-OBJECT-DETAIL:SOAP-FAULT-DETAIL handle attribute directly to work with the SOAP fault-detail object.

6. Returns the root node of the underlying SOAP fault <detail> element by using the hSoapFaultDetail:GET-NODE( ) method to assign the root node to the x-noderef object referenced by the handle variable hxnoderef
7. Can now use the methods and attributes of the x-noderef object handle (hxnoderef) and additional handle variables to walk the XML DOM subtree referenced by hxnoderef to examine the content of the SOAP fault <detail> element as specified by the WSDL for the Web service

Table 12: Sample SOAP fault procedure

```
DEFINE VARIABLE hWS           AS HANDLE.
DEFINE VARIABLE hStockPortType AS HANDLE.
DEFINE VARIABLE price          AS DECIMAL.

CREATE SERVER hWS.
/* Create a WebServicePortType object, using server & port information. */

hWS:CONNECT( "-WSDL http: //www.stockvend.com/application/wsdl/stock.wsdl
              -Service stockSVC
              -Port stockPort" ).
RUN stock SET hStockPortType ON SERVER hWS.

RUN getPrice IN hStockPortType( INPUT "error", OUTPUT price ) NO-ERROR.
IF ERROR-STATUS:ERROR THEN DO:
/*1*/
  /* Error occurred on the RUN. Did the Web service generate the error or was
     it an internal ABL error? */
  IF VALID-HANDLE( ERROR-STATUS:ERROR-OBJECT-DETAIL ) THEN DO:
    DEFINE VARIABLE hSoapFault as HANDLE.
/*2*/
    hSoapFault = ERROR-STATUS:ERROR-OBJECT-DETAIL.
/*3*/
    IF INDEX( "VersionMismatch", hSoapFault:SOAP-FAULT-CODE ) > 0 THEN DO:

/*4*/
      IF VALID-HANDLE( hSoapFault:SOAP-FAULT-DETAIL ) THEN DO:

        DEFINE VARIABLE hSoapFaultDetail as HANDLE.
/*5*/
        hSoapFaultDetail = hSoapFault:SOAP-FAULT-DETAIL.

        DEFINE VARIABLE hxnoderef AS HANDLE.
        CREATE X-NODEREF hxnoderef.
/*6*/
        hSoapFaultDetail:GET-NODE( hxnoderef ).
/*7*/
        /* From here the application can walk the detail XML and retrieve the
           relevant information. */

        . . .

      END. /* Examine SOAP-FAULT-DETAIL */
    END. /* Return SOAP-FAULT-CODE info */
  END. /* Examine ERROR-OBJECT-DETAIL */
END.

DELETE PROCEDURE hStockPortType.
hWS:DISCONNECT( ).
```

Structured error handling example

The code in the following example:

1. Encloses the logic in a simple block to demonstrate the placement and syntax of the CATCH block. In this simple example, if an error other than a system error occurred in the block, the error would be thrown to the enclosing block, which is the main block of the procedure (.p) file.
2. Handles any Progress.Lang.SoapFaultError error object generated by the AVM with the first CATCH block. This class is essentially a wrapper for the built-in SOAP-fault object.
3. Assigns a handle variable (hSoapFault) to the SOAP-fault object returned by the AVM for code readability.
4. Examines the values of SOAP fault elements, as required, using appropriate attributes (SOAP-FAULT-CODE) on the SOAP-fault object handle.
5. Uses the ABL VALID-HANDLE function to determine if this SOAP fault has SOAP fault detail by testing the validity of the handle returned by hSoapFault:SOAP-FAULT-DETAIL.
6. Assigns a handle variable (hSoapFaultDetail) to the SOAP fault-detail object returned by the hSoapFault:SOAP-FAULT-DETAIL attribute for code readability.
7. Returns the root node of the underlying SOAP fault <detail> element by using the hSoapFaultDetail:GET-NODE( ) method to assign the root node to the x-noderef object referenced by the handle variable hxnoderef.
8. Use the methods and attributes of the X-noderef object handle (hxnoderef) and additional handle variables to walk the XML DOM subtree referenced by hxnoderef to examine the content of the SOAP fault <detail> element as specified by the WSDL for the Web service.
9. Delete or throw the error object in the application code once it is handled by a CATCH block. Unhandled error objects are deleted automatically by the AVM.
10. Handles any system error (other than SoapFaultError) raised in the ON ERROR block with the second CATCH block. Because SoapFaultError is a subtype of SysError, the CATCH block handling SoapFaultError must occur before the more general CATCH block for all SysError objects.
11. Executes the FINALLY block whether the second CATCH block succeeded or failed. This makes the FINALLY block a good place to put clean up code.

Sample SOAP fault procedure

```
DEFINE VARIABLE hWS AS HANDLE.
DEFINE VARIABLE hStockPortType AS HANDLE.
DEFINE VARIABLE price AS DECIMAL.

/*1*/
DO ON ERROR UNDO, THROW:

  CREATE SERVER hWS.

  /* Create a WebServicePortType object, using server & port information. */
  hWS:CONNECT( "-WSDL http: //www.stockvend.com/application/wsdl/stock.wsdl
                -Service stockSVC
                -Port stockPort" ).

  RUN stock SET hStockPortType ON SERVER hWS.

  RUN getPrice IN hStockPortType( INPUT "error", OUTPUT price ).

/*2*/
  /* This CATCH handles SoapFaultErrors and ignores all other system errors.*/
```

```
  CATCH mySoapErrorObject AS Progress.Lang.SoapFaultError:

/*3*/
    DEFINE VARIABLE hSoapFault AS HANDLE.
    hSoapFault = mySoapErrorObject:SoapFault.

/*4*/
    IF INDEX("VersionMismatch", hSoapFault:SOAP-FAULT-CODE) > 0 THEN DO:

/*5*/
      IF VALID-HANDLE( hSoapFault:SOAP-FAULT-DETAIL ) THEN DO:

/*6*/
        DEFINE VARIABLE hSoapFaultDetail as HANDLE.
        ASSIGN hSoapFaultDetail = hSoapFault:SOAP-FAULT-DETAIL.

        DEFINE VARIABLE hxnoderef AS HANDLE.
        CREATE X-NODEREF hxnoderef.

/*7*/
        hSoapFaultDetail:GET-NODE( hxnoderef ).

/*8*/
        /* From here the application can walk the detail XML and retrieve the
           relevant information. */

/*9*/
        DELETE OBJECT mySoapErrorObject.
      END. /* Return SOAP-FAULT-CODE info */
    END. /* Examine SOAP-FAULT-DETAIL */
  END CATCH.

/*10*/
  CATCH mySystemErrorObject AS Progress.Lang.SysError:
    /* Handle any other system error. Since SysError is a superclass of
       SoapFaultError, this CATCH would also handle SoapFaultError if the
       more specific CATCH block did not come first. */
    DELETE OBJECT mySysErrorObject.
  END CATCH.

/*11*/
  FINALLY:
    DELETE PROCEDURE hStockPortType.
    hWS:DISCONNECT( ).
  END FINALLY.

END /* DO ON ERROR */
```

Asynchronous call error handling example

As mentioned in Results handling, class-based error objects returned from an asynchronous call are handled slightly differently:

1. Note that, in contrast to comment 2 in the Traditional error handling example, the code retrieves the error object from the `ERROR-OBJECT` attribute of the asynchronous call handle (`hRequest`).
2. The `ERROR-OBJECT` attribute returns a reference to an object that implements the `Progress.Lang.Error` interface. To access information beyond the attributes inherited from `Progress.Lang.Error`, the object referenced by `myerr` is cast to the more specific type of object returned, `Progress.Lang.SoapFaultError`.
3. Checks the `numMessages` property of `soaperr` (part of `Progress.Lang.Error`'s implementation) and displays messages as necessary.
4. Uses the ABL `VALID-HANDLE` function to determine the `SoapFault` property of `soaperr` points to a valid Soap-Fault object handle. If so, the information in the handle's attributes is displayed.
5. Uses the ABL `VALID-HANDLE` function again to determine if the SOAP fault has SOAP fault detail by testing the validity of the handle returned by h`SoapFault:SOAP-FAULT-DETAIL`. If so, it assigns a reference to the handle to the variable `SOAPFaultDetail`.

Sample asychronous error procedure

```
DEFINE VARIABLE hWS AS HANDLE.
DEFINE VARIABLE hWebSrvc AS HANDLE NO-UNDO.
DEFINE VARIABLE hPortType AS HANDLE NO-UNDO.
DEFINE VARIABLE hSOAPFault AS HANDLE NO-UNDO.
DEFINE VARIABLE hSOAPFaultDetail AS HANDLE NO-UNDO.
DEFINE VARIABLE i AS INTEGER NO-UNDO.
DEFINE VARIABLE cResponse AS CHARACTER NO-UNDO.
DEFINE VARIABLE hRequest AS HANDLE NO-UNDO.

CREATE SERVER hWebSrvc.
hWebSrvc:CONNECT("-pf soap.pf").

RUN ErrorServiceSoap SET hPortType ON hWebSrvc.

/* Get a SOAP fault */
/* HelloWorld2 always throws an exception with detailed contents. */

RUN HelloWorld2 IN hPortType ASYNCHRONOUS SET hRequest
  EVENT-PROCEDURE "procDone" IN THIS-PROCEDURE (OUTPUT cResponse).

WAIT-FOR PROCEDURE-COMPLETE OF hRequest.

DELETE OBJECT hRequest.
DELETE OBJECT hPortType.
hWebSrvc:DISCONNECT().
DELETE OBJECT hWebSrvc.

PROCEDURE procDone:
  DEFINE INPUT PARAMETER cResponse AS CHARACTER.
  DEFINE VARIABLE myerr AS Progress.Lang.Error.
  DEFINE VARIABLE soaperr AS Progress.Lang.SoapFaultError.

  IF hRequest:ERROR THEN DO:
    MESSAGE "An error occurred when running HelloWorld2".

/*1*/
    myerr = SELF:ERROR-OBJECT.
/*2*/
    soaperr = CAST (myerr, Progress.Lang.SoapFaultError).

/*3*/
    IF soaperr:NumMessages > 0 THEN DO:
      DO i = 1 TO soaperr:NumMessages:
        MESSAGE "SoapFault Error: " soaperr:GetMessage(i).
      END.
/*4*/
    IF VALID-HANDLE(soaperr:SoapFault) THEN DO:
      MESSAGE "soaperr:SOAPFAULT is a valid handle".
      hSOAPFault = soaperr:SoapFault.
      MESSAGE
        "Fault Code: "   hSOAPFault:SOAP-FAULT-CODE        SKIP
        "Fault String: " hSOAPFault:SOAP-FAULT-STRING      SKIP
        "Fault Actor: "  hSOAPFault:SOAP-FAULT-ACTOR       SKIP
        "Error Type: "   hSOAPFault:TYPE SKIP.

/*5*/
      IF VALID-HANDLE(hSOAPFault:SOAP-FAULT-DETAIL) THEN DO:
        SOAPFaultDetail = hSOAPFault:SOAP-FAULT-DETAIL.
      END.
    END.
  END.
  END.
END PROCEDURE.
```

Debugging ABL applications that call Web services

If an ABL application error appears (from error messages, for example) to result from attempts to access a Web service, you generally follow three phases of investigation to isolate the problem.

To investigate a problem related to using a Web service:

1. Determine if the WSDL file is still valid for the Web service you are accessing.

   This is particularly necessary if you access the WSDL file at a location other than the URL specified by the Web service provider, for example using a local file system that stores a separate copy that you have made from the original. It is also very possible that the Web service provider has changed the URL or target namespace to reference the WSDL file. If this is the case, you must run the WSDL Analyzer against the new WSDL file to identify any changes that might affect your code. For more information, see Using the WSDL Analyzer.

   This is a good first step to diagnose any Web service access failure.

2. If the WSDL file appears to be valid, use a SOAP message viewer at run time to compare the content of SOAP request and response messages with what the application expects and what the WSDL file specifies.

   OpenEdge provides a set of SOAP viewers included with the installation, and many other viewers are available. This chapter describes three of them to varying levels of detail.

3. If you suspect a problem with SOAP request and response messages, you can debug sample SOAP messages for selected operations by creating them in local files. Specify the SOAP addresses of these files using Microsoft UNC pathnames in the WSDL Bindings for the selected operations.
4. Debug the ABL code, possibly using the OpenEdge Debugger. For more information on the Debugger, see *OpenEdge Development: Debugging and Troubleshooting.*

SOAP Viewers

The following table lists some tools that you can use to view Web service messages as they are exchanged between the client and the Web service.

Table 13: Web service message viewers

This SOAP message viewer . . .	Is . . .
WSAViewer	Provided with the OpenEdge installation
ProSOAPView	Provided with the OpenEdge installation
Microsoft SOAP Toolkit	Available for free download from Microsoft's Web site

The Microsoft SOAP Toolkit contains a variety of tools for working with SOAP messages in Windows platforms. One advantage of WSAViewer and ProSOAPView is that they run on all supported OpenEdge platforms.

Using WSAViewer

WSAViewer is a basic tool for viewing the SOAP messages between a given source and destination, which you determine at startup. As such, it functions as a "man in the middle" between the client and the Web service. It is useful if you are only interested in viewing the content of the SOAP request and response. The simplicity of this tool makes it handy to use when debugging OpenEdge Web services for which viewing the well-defined SOAP formats is the primary focus. For more information on this tool, see Testing and Debugging OpenEdge SOAP Web Services.

Using WSAViewer with an ABL client

To use WSAViewer with an ABL client, you have two options:

- Change the connection parameters in your Web service `CONNECT( )` method to send all SOAP messages to WSAViewer. WSAViewer then forwards them to the actual Web service.
- Startup the ABL client using the `-proxyhost` and `-proxyport` startup parameters to redirect the SOAP messages to WSAViewer. Specify `-proxyhost` using the host name where the viewer is running (typically `localhost`), and specify `-proxyport` using the port on which WSAViewer listens for SOAP request messages (the *`listen-port`* value specified for WSAViewer as shown in the following syntax).

The syntax to start the viewer is the same when working with any Web service as it is when working with the OpenEdge Web Services Adapter (WSA):

Syntax

```
wsaviewer listen-portwebservice-hostwebservice-port
```

The *`listen-port`* is the port on which WSAViewer listens for SOAP request messages. The *`webservice-host`* is the host name of the Web service and the *`webservice-port`* is the host port on which the Web service listens for SOAP request messages.

Suppose you enter the following command line to start the viewer to listen on `localhost` at port 8080 and pass SOAP request messages to the Web service, `www.stockvend.com`, listening on port 80:

```
wsaviewer 8080 www.stockvend.com 80
```

You might code your OpenEdge Web service `CONNECT( )` method like this:

```
DEFINE VARIABLE hWS AS HANDLE.
CREATE SERVER hWS.
hWS:CONNECT("-WSDL http://www.stockvend.com/application/wsdl/stock.wsdl
             -Binding StockQuoteObj
             -SOAPEndpoint http://localhost:8080/application/stockquotes").
```

The `CONNECT( )` method still gets the WSDL file directly from `www.stockvend.com`, but all SOAP messages go through WSAViewer on their way to the Web service and back to the client.

Using ProSOAPView

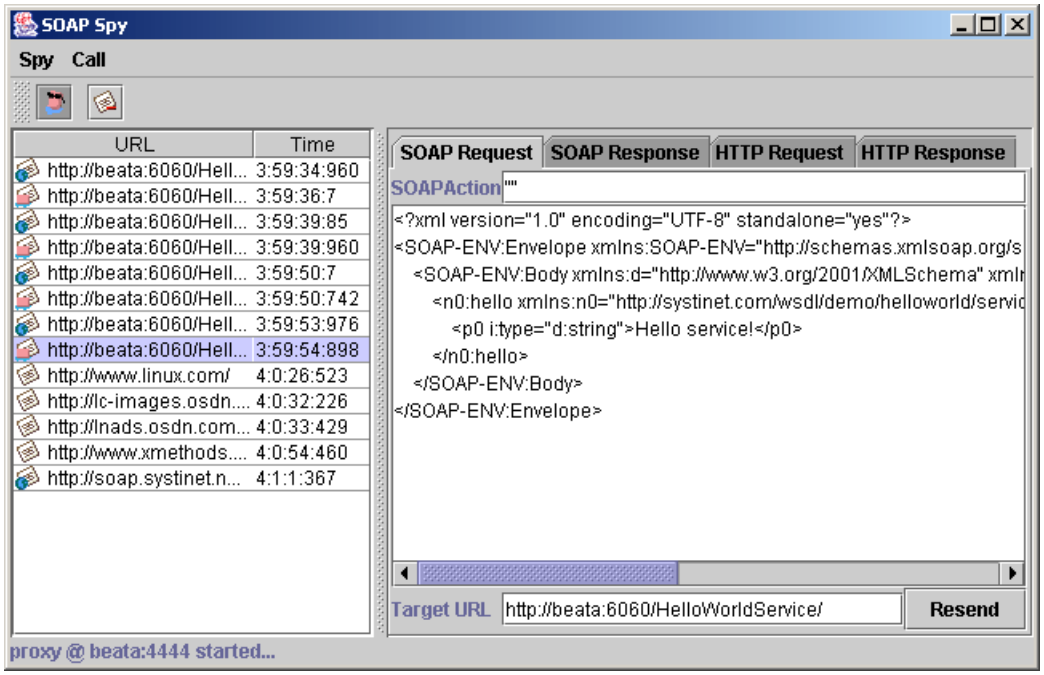
ProSOAPView is a more flexible tool than WSAViewer that allows you to view the following message content exchanged between a client and Web service:

- Request for a WSDL file
- SOAP request and response content
- HTTP request and response content
- The content of any other document exchanged between the client and Web service

This flexibility is especially helpful to debug industry standard Web services accessed from ABL (which can originate anywhere and contain a greater variety of SOAP formats).

Running ProSOAPView opens a **SOAPSpy** window, as shown in the following figure.

Figure 3: SOAPSpy window opened by ProSOAPView



SOAPSpy works both an HTTP proxy server and as an HTTP client, meaning that it can serve as a proxy between a Web service client and the Web service and also connect to its final destination through another proxy, all without changing any of the Web service client code.

The data stream that SOAPSpy can track includes binary data as well as readable text. However, binary data is represented schematically and all invalid text characters appear as the `'?'` character.

Running ProSOAPView (SOAPSpy)

You run ProSOAPView by setting it up as a proxy server for the ABL client.

To configure the ABL client and run ProSOAPView:

1. Start ProSOAPView by executing the `prosoapview` command in the OpenEdge environment using the following syntax:

Syntax

```
prosoapview [port-number]
```

By default the SOAPSpy executable listens on TCP/IP port 4444. You can specify the *port-number* value to change this port assignment.

2. To begin tracking HTTP and SOAP messages, be sure that the client application has not yet executed the Web service CONNECT( ) method, then choose **Start Spying** > **Spy** from the menu bar, as shown here, or click the spy icon in the **SOAPSpy** window.



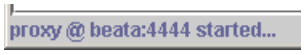
3. Start up the ABL client adding the following startup parameters to the command line:

```
-proxyhost localhost -proxyport port-number
```

Set *port-number* to the TCP/IP listening port used by SOAPSpy (4444 by default). For more information on these startup parameters and starting up an ABL client, see *OpenEdge Deployment: Startup Command and Parameter Reference.*

Note: You do not have to change any connection parameters coded for the Web service CONNECT( ) method in the ABL client application to use ProSOAPView. Proxy settings handle all redirection of message traffic that is required to inspect messages sent between the ABL client and the Web service.

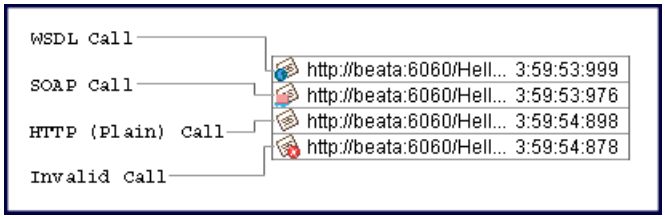
After the SOAPSpy window is opened and tracking messages has begun, the status bar indicates the proxy host and port as shown here, where the proxy host is name beata.



Tracking messages in the SOAPSpy window

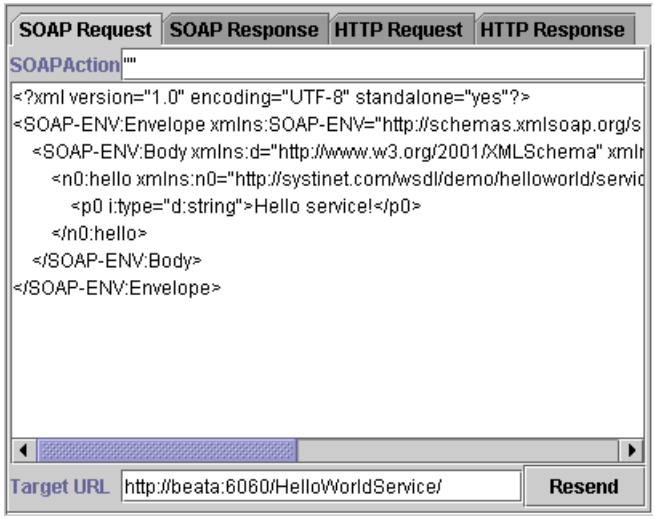
Once tracking begins, the SOAPSpy window shows a list of calls (Web service operation invocations) on the left and a tab viewer on the right showing message content (see Using ProSOAPView on page 62).

Each call in the list is identified by its URI, the time of the call, and an icon that identifies the type of call, as shown here.



When you select a call in the list, its message content appears in the tab viewer. You can view different message content for a call by selecting a tab in the viewer. The **HTTP Request** and **HTTP Response** tabs show the raw content of every call. The **SOAP Request** and **SOAP Response** tabs allow you to manipulate individual SOAP messages, including WSDL requests.

For example, this is the content of a SOAP request message in the **SOAP Request** tab.



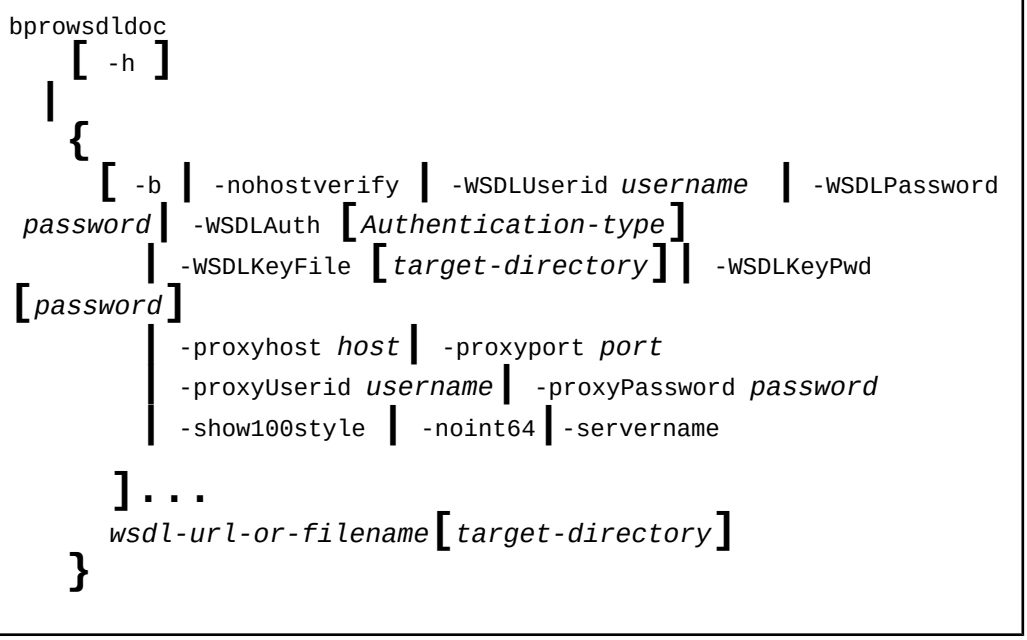
If you want to remove a call from the call list, select the call and choose **Remove Call**, either in the **Call** menu or in the context menu that pops up on the call.



A

WSDL Analyzer

Runs the WSDL Analyzer, which provides HTML documentation on the interface that a web service Description Language (WSDL) describes. This HTML documentation describes how an ABL developer can access the web service and its operations in ABL.

Operating system	Syntax
UNIX Windows	

`-h`

Displays a help message on the usage of this command.

`-b`

Forces documentation of binding names. A web service can offer two options for a web service client to connect (or bind) to it:

- Using services with ports (most common)
- Using binding names (the names of the WSDL <`binding`> elements) with SOAP endpoints that you must obtain separately (sometimes used to bind a SOAP viewer)

Normally, if a WSDL defines services, the Analyzer includes only the service and port names in the generated documentation. If the WSDL does not define any services, the Analyzer documents the binding names. This option tells the Analyzer to document the binding names even when services are defined in the WSDL.

`-nohostverify`

Turns off host verification for a Transport Layer Security (TLS) connection using HTTPS.

`-WSDLUserid` *username*

Specifies a username for accessing the WSDL file.

`-WSDLPassword` *password*

Specifies a password for accessing the WSDL file.

`-WSDLAuth` *password*

Specifies if the access to a WSDL file requires TLS client authentication. Set this to `ssl` to enable TLS client authentication for WSDL access. If you set this to `Basic`, the connect method ignores client authentication for WSDL access.

`-WSDLKeyFile` *password*

Specifies the location of the client certificate. If you do not specify an absolute path of the client certificate file, the connection operation searches the `$DLC/keys` folder for the *`client-certificate-file-name.pem`* file. This option must be updated only if `-WSDLAuth` parameter is set to `ssl`.

`-WSDLKeyPwd` *password*

Specifies, in clear text or encoded format, the password of the client certificate. This option must be updated only if `-WSDLAuth` parameter is set to `ssl`.

`-proxyhost` *host*

Specifies the name or IP address of the host where an HTTP-based proxy server required to access the WSDL file is located.

`-proxyport` *port*

Specifies the port on which the HTTP-based proxy server is listening for requests to download WSDL files.

`-proxyUserid` *username*

Specifies a username for accessing an HTTP-based proxy server.

`-proxyPassword` *password*

Specifies a password for accessing an HTTP-based proxy server.

`-show100style`

Shows procedure and function signatures as documented in the 10.0x releases of OpenEdge. With the release of 10.1A, some procedure signatures are changed for ease of use. For more information, see Analyzing wrapped document literal..

`-noint64`

Prior to OpenEdge Version 10.1B, the ABL `INT64` data type did not exist and the WDSL Analyzer mapped XML Schema types of `xsd:long` to the ABL `DECIMAL` data type. Use this option if you want to use the `xsd:long` to ABL `DECIMAL` mapping. Otherwise, `xsd:long` maps to `INT64`. The current version of OpenEdge continues to recognize existing mappings of `xsd:long` to `DECIMAL` as valid whether or not this option is specified.

wsdl-url-or-filename

Specifies a URL, Microsoft Uniform Naming Convention (UNC), or local pathname to the WSDL file. If the string does not specify a URL protocol ("file://", "http://", or "https://") or start with a UNC prefix ("\\"), it is assumed to be a a local file pathname.

target-directory

Specifies the target directory (to be created if necessary) where the Analyzer will write the generated documents. If not specified, the Analyzer assumes the current working directory. Choose a directory dedicated for the specified WSDL file, as some generated document filenames are the same for multiple WSDL files.

–servername

Specifies the server name in TLS connection initialization for client that it sends to the server as part of the TLS negotiation.

B

WSDL Details

This appendix includes expanded information on WSDL files. Only certain users need this level of detail.

For details, see the following topics:

- ABL procedure prototype to WSDL operation
- Array mapping in WSDL documents
- Define TABLE (static temp-table) parameters
- Define TABLE-HANDLE (dynamic temp-table) parameters
- Define DATASET (static ProDataSet) parameters
- Define DATASET-HANDLE (dynamic ProDataSet) parameters

ABL procedure prototype to WSDL operation

This is the ABL procedure prototype for the `FindCustomerByNum.p` external procedure:

ABL procedure prototype

```
/* FindCustomerByNum.p */

DEFINE INPUT PARAMETER CustomerNumber AS INTEGER.
DEFINE OUTPUT PARAMETER CustomerName AS CHARACTER.
```

The boldface elements in this prototype are the main information that ProxyGen maps into the corresponding Web service operation definition. These include the procedure name (filename for an external procedure) and for any parameters, the parameter mode (input or output), names, and data types.

Note: Some information can only be specified in ProxyGen, such as whether the ABL `RETURN-VALUE` is used and (for external procedures) what Open Client object this operation belongs to.

These are the `message` section definitions in the RPC/Encoded and RPC/Literal WSDL file for the `FindCustomerByNum` operation request and response messages:

WSDL message section defining request and response messages

```
<message name="OrderInfo_FindCustomerByNum">
  <part name="CustomerNumber" type="xsd:int"/>
</message>
<message name="OrderInfo_FindCustomerByNumResponse">
  <part name="CustomerName" type="xsd:string"/>
</message>
```

Note that the request message contains the input parameter, `CustomerNumber`, and the response message contains the output parameter, `CustomerName`, both defined by appropriate XML data types.

This is the definition for the `FindCustomerByNum` operation in the `portType` section of the WSDL:

WSDL portType section defining an operation signature on an object

```
<portType name="OrderInfoObj">
  <operation name="FindCustomerByNum"
             parameterOrder="CustomerNumber CustomerName">
    <input message="tns:OrderInfo_FindCustomerByNum"/>
    <output message="tns:OrderInfo_FindCustomerByNumResponse"/>
    <fault name="OrderInfoFault" message="tns:FaultDetailMessage"/>
  </operation>
  ...
</portType>
```

The `portType` section defines the object in which the operation is defined, in this case, the AppObject, `OrderInfo` (as specified in ProxyGen). Note that this definition groups together the request (input) and response (output) messages, along with a generic fault message as part of the operation definition.

This is the definition for the `FindCustomerByNum` operation in the Bindings section for the `OrderInfo` AppObject:

WSDL Bindings section defining an operation and object binding

```
<binding name="OrderInfoObj" type="tns:OrderInfoObj">
  <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>

  <operation name="FindCustomerByNum">
    <soap:operation soapAction="" style="rpc"/>
    <input>
      <soap:header message="tns:OrderInfoID" part="OrderInfoID"
                use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:OrderSvc:OrderInfo" wsdl:required="true">
      </soap:header>
      <soap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:OrderSvc:OrderInfo"/>
    </input>
    <output>
      <soap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:OrderSvc:OrderInfo"/>
    </output>
    <fault name="OrderInfoFault">
      <soap:fault name="OrderInfoFault" use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://servicehost:80/wsa/wsa1"/>
    </fault>
  </operation>
  ...
</binding>
```

Note that this definition specifies that the transport protocol as SOAP over HTTP, and goes on to define the content for SOAP messages request (input) and response (output) messages for the operation. This is where the object ID definitions are referenced for operations that require them.

This is the port definition in the `service` section for the object (`OrderInfo` AppObject) containing the `FindCustomerByNum` operation:

WSDL service section defining the Web service location

```
<service name="OrderInfoService">
  <port name="OrderInfoObj" binding="tns:OrderInfoObj">
    <documentation/>
    <soap:address location="http://servicehost:80/wsa/wsa1"/>
  </port>
  ...
</service>
```

Note that the URL for the web service is specified in `soap:address`.

Array mapping in WSDL documents

This section provides detailed information about how array parameters are represented in each of the WSDL formats. The description of each format includes an example based on an ABL procedure contained in the `CustomerA0` AppObject with the following signature:

arraySample procedure signature

```
/* arraySample*/

DEFINE INPUT  PARAMETER names     AS CHARACTER EXTENT.
DEFINE INPUT  PARAMETER hireDates AS DATETIME  EXTENT.
DEFINE OUTPUT PARAMETER quotas    AS INTEGER   EXTENT 12.
```

Doc/Lit

For the Doc/Lit format, an array parameter is represented as a sequence of its data type in an XML Schema type definition:

Array parameter definition for Doc/Lit WSDL

```
<complexType
  <sequence>
    <element name="paramName" type="XMLType"
             minOccurs="zero_or_extentval"
             maxOccurs="extentval_or_unbounded"/>
  </sequence>
</complexType>
```

The bolded elements refer to:

- ***paramName*** — The name of the parameter
- ***XMLType*** — The XML Schema type of the parameter
- ***zero_or_extentval*** — `0` or an integer for the minimum array size
- ***extentval_or_unbounded*** — An integer for the maximum array size or `unbounded`

The Doc/Lit WSDL for `arraySample` has the following `types` and `message` sections:

Doc/Lit WSDL schema example

```
<wsdl:types>
  <schema targetNamespace="urn:tempuri-org:CustomerAO"
          xmlns="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified">
    <element name="arraySample">
      <complexType>
        <sequence>
          <element name="names" type="xsd:string" minOccurs="0"
                   maxOccurs="unbounded" nillable="true">
          <element name="hireDates" type="xsd:dateTime" minOccurs="0"
                   maxOccurs="unbounded" nillable="true">
        </sequence>
      </complexType>
    </element>
    <element name="arraySampleResponse">
      <complexType>
        <sequence>
          <element name="quotas" type="xsd:int" minOccurs="12"
                   maxOccurs="12" nillable="true">
        </sequence>
      </complexType>
    </element>
  </schema>
</wsdl:types>
<wsdl:message name="CustomerAO_procName">
  <part name="parameters" element="S1:procName"/>
</wsdl:message>
<wsdl:message name="CustomerAO_procNameResponse">
  <part name="parameters" element="S1:procNameResponse"/>
</wsdl:message>
```

RPC/Literal

For the RPC/Literal format, an array parameter is represented as an unbounded sequence of its data type in an XML Schema type definition, where ***XMLType*** is the XML Schema type:

Array parameter definition for RPC/Literal WSDL

```
<complexType name="ArrayOf***XMLType***">
  <sequence>
    <element name="item" type="XMLType" minOccurs="0"
             maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

The WSDL file contains a separate schema for each object defined in ProxyGen. Each such schema includes one `complexType` definition for each data type used as an array parameter for any procedure or function in that object. For example, if the ABL defines one or more parameters in the object with the data type `CHARACTER EXTENT`, the schema includes a `complexType` named "`ArrayOfString`".

The RPC/Literal WSDL document for the `arraySample` procedure shown at the beginning of this section includes the following `types` and `message` sections:

RPC/Literal WSDL schema example

```
<wsdl:types>
  <schema targetNamespace="urn:tempuri-org:CustomerAO"
          xmlns="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="unqualified"
    <complexType name="ArrayOfInt">
      <sequence>
        <element name="item" type="xsd:int" minOccurs="0"
                 maxOccurs="unbounded"/>
      </sequence>
    </complexType>
    <complexType name="ArrayOfDateTime">
      <sequence>
        <element name="item" type="xsd:dateTime" minOccurs="0"
                 maxOccurs="unbounded"/>
      </sequence>
    </complexType>
    <complexType name="ArrayOfString">
      <sequence>
        <element name="item" type="xsd:string" minOccurs="0"
                 maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </schema>
</wsdl:types>

<wsdl:message name="CustomerAO_arraySample">
  <part name="names"     type="S1:ArrayOfString"/>
  <part name="hireDates" type="S1:ArrayOfDateTime"/>
</wsdl:message>
<wsdl:message name="CustomerAO_arraySampleResponse">
  <part name="quotas"    type="S1:ArrayOfInt"/>
</wsdl:message>
```

RPC/Encoded

For the RPC/Encoded format, an array parameter is represented as a SOAP array `complexType` in an XML Schema type definition, as shown:

Array parameter definition for RPC/Encoded WSDL

```
<complexType name="ArrayOfXMLType">
  <complexContent>
    <restriction base="soapenc:Array">
      <attribute ref=soapenc:arrayType" wsdl:arrayType="XMLType[]"/>
    </restriction>
  <complexContent>
</complexType>
```

The WSDL file contains a separate schema for each object defined in ProxyGen. Each such schema includes one SOAP array `complexType` definition for each data type used as an array parameter for any procedure or function in that object. For example, if the ABL defines one or more parameters in the object with the data type `CHARACTER EXTENT`, the schema includes a `complexType` named "`ArrayOfString`".

The RPC/Encoded WSDL document for the `arraySample` procedure shown at the beginning of this section includes the following `types` and `message` sections:

RPC/Encoded WSDL schema example

```
<wsdl:types>
  <schema targetNamespace="urn:tempuri-org:CustomerAO"
          xmlns="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="unqualified">
    <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
    <complexType name="ArrayOfInt">
      <complexContent>
        <restriction base="soapenc:Array">
          <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:int[]"/>
        </restriction>
      </complexContent>
    </complexType>
    <complexType name="ArrayOfDateTime">
      <complexContent>
        <restriction base="soapenc:Array">
          <attribute ref="soapenc:arrayType"
                     wsdl:arrayType="xsd:dateTime[]"/>
        </restriction>
      </complexContent>
    </complexType>
    <complexType name="ArrayOfString">
      <complexContent>
        <restriction base="soapenc:Array">
          <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:string[]"/>
        </restriction>
      </complexContent>
    </complexType>
  </schema>
</wsdl:types>

. . .

<wsdl:message name="CustomerAO_arraySample">
  <part name="names"     type="S1:ArrayOfString"/>
  <part name="hireDates" type="S1:ArrayOfDateTime"/>
</wsdl:message>
<wsdl:message name="CustomerAO_arraySampleResponse">
  <part name="quotas"    type="S1:ArrayOfInt"/>
</wsdl:message>
```

Define TABLE (static temp-table) parameters

TABLE parameters pass data only, because the static temp-table's schema is known at WSDL generation. OpenEdge Web services map a TABLE definition to a <complexType> consisting of a <sequence> of elements that represent a row (temp-table record). Each <element> in this sequence represents a column (temp-table field) of the row. For all SOAP formats, a TABLE parameter is defined as a <complexType> that references the corresponding row element <complexType>.

The following WSDL sample defines a TABLE row named, staticTT_ttEmpRow, with two columns, Name and Number:

TABLE parameter row schema for all SOAP formats

```
<complexType name="staticTT_ttEmpRow">
  <sequence>
    <element name="Name" nillable="true" type="xsd:string"/>
    <element name="Number" nillable="true" type="xsd:int"/>
  </sequence>
</complexType>
```

The following WSDL sample defines a temp-table parameter for the `ttEmp` row using the RPC/Encoded SOAP format. Note that the parameter is a SOAP array of rows:

TABLE parameter for RPC/Encoded

```
<complexType name="ArrayOfstaticTT_ttEmpRow">
   <complexContent>
      <restriction base="soapenc:Array">
         <attribute ref="soapenc:arrayType"
            wsdl:arrayType="S2:staticTT_ttEmpRow[]"/>
      </restriction>
   </complexContent>
</complexType>
```

The following WSDL sample defines a TABLE parameter using the Doc/Lit or RPC/Literal SOAP formats. Note that the parameter is a sequence of multiple rows:

TABLE parameter for RPC/Literal and Doc/Lit

```
<complexType name="staticTT_ttEmpParam">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="ttEmpRow"
             type="S2:staticTT_ttEmpRow" />
  </sequence>
</complexType>
```

For these SOAP formats, the row element name (`ttEmpRow`) is used to identify each element that holds a data row sent in SOAP messages that pass a TABLE parameter.

Each column of a TABLE row can hold any data type shown in the following table.

Table 14: XML data types for TABLE parameter columns

ABL data type	XML Schema data type
BLOB[2]	xsd:base64Binary
CHARACTER	xsd:string
CLOB[2]	xsd:string
COM-HANDLE	xsd:long

[2] BLOB and CLOB data types are designed to support very large objects. Use of these data types for table fields in Web services can result in a serious performance impact.

ABL data type	XML Schema data type
DATE	xsd:date
DATETIME	xsd:dateTime
DATETIME-TZ	xsd:dateTime
DECIMAL	xsd:decimal
INT64	xsd:long
INTEGER (32 bit)	xsd:int
LOGICAL	xsd:boolean
RAW	xsd:base64Binary
RECID (32 or 64 bit)	xsd:long
ROWID	xsd:base64Binary
WIDGET-HANDLE	xsd:long

Define TABLE-HANDLE (dynamic temp-table) parameters

TABLE-HANDLE parameters pass both the schema and data, because the dynamic temp-table schema is not known at compile time. Thus, for TABLE-HANDLE parameters, OpenEdge Web services map an ABL TABLE-HANDLE to a <complexType> containing a sequence of xsd:any. There is a single definition used for all TABLE-HANDLE parameters in all supported SOAP formats. The following WSDL sample shows this common TABLE-HANDLE parameter definition:

TABLE-HANDLE definition for all dynamic temp-table parameters

```
<complexType name="TableHandleParam">
  <sequence>
    <any namespace="##local"/>
  </sequence>
</complexType>
```

The client application must create (for input) and parse (for output) the XML Schema along with the data for the parameter. How the client inserts the input schema and data in request messages and how it parses the output schema and data from response messages is entirely dependent on the client toolkit.

Each column of a TABLE-HANDLE row can hold any data type shown in the following table.

XML data types for TABLE-HANDLE parameter columns

ABL data type	XML Schema data type
CHARACTER	xsd:string
DATE	xsd:date
DATETIME-TZ	xsd:dateTime
DECIMAL	xsd:decimal
INT64	xsd:long
INTEGER (32 bit)	xsd:int
LOGICAL	xsd:boolean
RAW	xsd:base64Binary

Define DATASET (static ProDataSet) parameters

DATASET parameters pass data only because the static ProDataSet's schema is known at WSDL generation. OpenEdge Web services map a DATASET definition to a <complexType> consisting of a <sequence> of elements that represent the ProDataSet's temp-tables. Each temp-table element includes a <complexType> describing the temp-table's fields. The definition also includes elements describing the data relations and indexes.

By default, a ProDataSet parameter includes only the current data. You must specify in ProxyGen the ProDataSet parameters for which you want to include before-image data. A ProDataSet parameter with before-image data is serialized as a proprietary OpenEdge datasetChanges document.

Nested and non-nested data relations produce different WSDL structures. For non-nested data relations, each temp-table description is separate. For nested data relations, the description of the child temp-table is embedded in the description of the parent temp-table.

Client-development toolkits typically define an object for every <complexType> in the WSDL. For an ABL client, the WSDL Analyzer produces definitions for a corresponding ProDataSet, its temp-tables, indexes, and data relations. A non-ABL client toolkit produces object definitions for the ProDataSet and for each of its temp-tables. Non-ABL clients toolkits do not produce code from the indexes and data relations.

The following code snippet defines a ProDataSet with nested and non-nested data relations:

```
/* getCustOrders.p */
DEFINE TEMP-TABLE ttCust     NO-UNDO
       FIELD      CustNum    AS INTEGER
       FIELD      Name       AS CHARACTER
       INDEX      CustNumIdx IS UNIQUE PRIMARY CustNum.

DEFINE TEMP-TABLE ttOrder     NO-UNDO
       FIELD      OrderNum    AS INTEGER
       FIELD      CustNum     AS INTEGER
       INDEX      OrderNumIdx IS UNIQUE PRIMARY OrderNum
       INDEX      CustOrdIdx  IS UNIQUE CustNum OrderNum.

DEFINE TEMP-TABLE ttOrderLine  NO-UNDO
       FIELD      OrderNum     AS INTEGER
       FIELD      LineNum      AS INTEGER
       INDEX      OrderLineIdx IS UNIQUE PRIMARY OrderNum LineNum.

DEFINE DATASET dsCustOrd FOR ttCust, ttOrder, ttOrderLine
       DATA-RELATION CustOrdRel FOR ttCust, ttOrder
         RELATION-FIELDS (CustNum, CustNum)
       DATA-RELATION OrdLinesRel FOR ttOrder, ttOrderLine
         RELATION-FIELDS (OrderNum, OrderNum) NESTED.

DEFINE INPUT  PARAMETER iCustNum AS INTEGER.
DEFINE OUTPUT PARAMETER DATASET FOR dsCustOrd.

/* fill dataset and return to caller */
...
```

The following WSDL sample defines the ProDataSet parameter for `getCustOrders.p` using the Doc/Lit SOAP format. Note the differences between the nested and non-nested data relations in this sample:

DATASET parameter for Doc/Lit

```
<!-- dataset definition -->
<element name="dsCustOrd" prodata:proDataSet="true">
  <complexType>
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="ttCust">
        <complexType>
          <sequence>
            <element name="CustNum" nillable="true" type="xsd:int"/>
            <element name="Name" nillable="true" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
      <element maxOccurs="unbounded" minOccurs="0" name="ttOrder">
        <complexType>
          <sequence>
            <element name="OrderNum" nillable="true" type="xsd:int"/>
            <element name="CustNum" nillable="true" type="xsd:int"/>
            <!-- nested data relation between ttOrder and ttOrderLine -->
            <element maxOccurs="unbounded" minOccurs="0"
                     name="ttOrderLine">
              <complexType>
                <sequence>
                 <element name="OrderNum" nillable="true" type="xsd:int"/>
                 <element name="LineNum" nillable="true" type="xsd:int"/>
                </sequence>
              </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
  <unique name="CustNumIdx" prodata:primaryIndex="true">
    <selector xpath="Sn:.//ttCust"/>
    <field xpath="Sn:CustNum"/>
  </unique>
  <unique name="OrderNumIdx" prodata:primaryIndex="true">
    <selector xpath=".//Sn:ttOrder"/>
    <field xpath="Sn:OrderNum"/>
  </unique>
  <unique name="CustOrdIdx">
    <selector xpath="Sn:.//ttOrder"/>
    <field xpath="Sn:CustNum"/>
    <field xpath="Sn:OrderNum"/>
  </unique>
  <unique name="OrderLineIdx" prodata:primaryIndex="true">
    <selector xpath=".//Sn:ttOrderLine"/>
    <field xpath="Sn:OrderNum"/>
    <field xpath="Sn:LineNum"/>
  </unique>
  <!-- non-nested data relation between ttCust and ttOrder -->
  <keyref name="CustOrdRel" refer="Sn:CustNumIdx">
    <selector xpath="Sn:.//ttOrder"/>
    <field xpath="Sn:CustNum"/>
  </keyref>
  <!-- nested data relation between ttOrder and ttOrderLine -->
  <keyref name="OrdLinesRel" prodata:nested="true" refer="Sn:OrderLineIdx">
    <selector xpath="Sn:.//ttOrderLine"/>
    <field xpath="Sn:OrderNum"/>
  </keyref>
</element>
```

Note: *Sn* refers to the namespace prefix for the XML Schema containing the dataset's definition.

For RPC styles, the name of the ProDataSet parameter follows the format, *PDSname*`Param`, where *PDSname* is the name of your ProDataSet. Both RPC styles use the same structure to define a ProDataSet parameter. The following WSDL sample defines the ProDataSet parameter using the RPC/Literal SOAP format. Note the differences between the nested and non-nested data relations in this sample:

DATASET parameter for RPC/Literal and RPC/Encoded

```
<!-- dataset definition -->
<complexType name="dsCustOrdParam" prodata:proDataSet="true">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="ttCust">
      <complexType>
        <sequence>
          <element name="CustNum" nillable="true" type="xsd:int"/>
          <element name="Name" nillable="true" type="xsd:string"/>
        </sequence>
      </complexType>
    </element>
    <element maxOccurs="unbounded" minOccurs="0" name="ttOrder">
      <complexType>
        <sequence>
          <element name="OrderNum" nillable="true" type="xsd:int"/>
          <element name="CustNum" nillable="true" type="xsd:int"/>
          <!-- nested data relation between ttOrder and ttOrderLine -->
          <element maxOccurs="unbounded" minOccurs="0" name="ttOrderLine">
            <complexType>
              <sequence>
                <element name="OrderNum" nillable="true" type="xsd:int"/>
                <element name="LineNum" nillable="true" type="xsd:int"/>
              </sequence>
            </complexType>
          </element>
        </sequence>
      </complexType>
    </element>
  </sequence>
  <annotation>
    <appinfo>
      <unique name="CustNumIdx" prodata:primaryIndex="true">
        <selector xpath=".//Sn:ttCust"/>
        <field xpath="Sn:CustNum"/>
      </unique>
      <unique name="OrderNumIdx" prodata:primaryIndex="true">
        <selector xpath=".//Sn:ttOrder"/>
        <field xpath="Sn:OrderNum"/>
      </unique>
      <unique name="CustOrdIdx">
        <selector xpath=".//Sn:ttOrder"/>
        <field xpath="Sn:CustNum"/>
        <field xpath="Sn:OrderNum"/>
      </unique>
      <unique name="OrderLineIdx" prodata:primaryIndex="true">
        <selector xpath=".//Sn:ttOrderLine"/>
        <field xpath="Sn:OrderNum"/>
        <field xpath="Sn:LineNum"/>
      </unique>
      <!-- non-nested data relation between ttCust and ttOrder -->
      <keyref name="CustOrdRel" refer="Sn:CustNumIdx">
        <selector xpath=".//Sn:ttOrder"/>
        <field xpath="Sn:CustNum"/>
      </keyref>
      <!-- nested data relation between ttOrder and ttOrderLine -->
      <keyref name="OrdLinesRel" prodata:nested="true"
              refer="Sn:OrderLineIdx">
        <selector xpath=".//Sn:ttOrderLine"/>
        <field xpath="Sn:OrderNum"/>
        <field xpath="Sn:LineNum"/>
      </keyref>
    </appinfo>
  </annotation>
</complexType>
```

Include before-image data

Before-image data is serialized in a proprietary OpenEdge `datasetChanges` document. The WSDL represents the parameter as arbitrary complex data with an <any> element. XML Schema attributes from the ABL-specific namespace identify the element as a `datasetChanges` document.

An ABL-based client can map the <any> element to a ProDataSet parameter and parse the OpenEdge `datasetChanges` document into a ProDataSet and its before-image data through the `READ-XML( )` method. For more information on how ABL handles before-image data in XML, see the chapter on reading and writing XML data from ProDataSets in *OpenEdge Development: Working with XML*. Non-ABL clients map the <any> element to an XML document that the client developer needs to parse with an XML API.

The following code snippet defines a ProDataSet with before-image data:

```
/* getCustOrdersBI.p */
DEFINE TEMP-TABLE    ttCust     NO-UNDO
       BEFORE-TABLE  ttCustBef
       FIELD         CustNum    AS INTEGER
       FIELD         Name       AS CHARACTER
       INDEX         CustNumIdx IS UNIQUE PRIMARY CustNum.

DEFINE TEMP-TABLE   ttOrder     NO-UNDO
       BEFORE-TABLE ttOrderBef
       FIELD        OrderNum    AS INTEGER
       FIELD        CustNum     AS INTEGER
       INDEX        OrderNumIdx IS UNIQUE PRIMARY OrderNum
       INDEX        CustOrdIdx  IS UNIQUE CustNum OrderNum.

DEFINE DATASET dsCustOrd FOR ttCust, ttOrder
       DATA-RELATION CustOrdRel FOR ttCust, ttOrder
         RELATION-FIELDS (CustNum, CustNum).

DEFINE INPUT  PARAMETER iCustNum AS INTEGER.
DEFINE OUTPUT PARAMETER DATASET  FOR dsCustOrd.

/* fill dataset and return to caller */
...
```

The following WSDL sample defines the ProDataSet parameter for `getCustOrdersBI.p` using the Doc/Lit SOAP format:

Doc/Lit WSDL for getCustOrdersBI.p

```
<!-- datasetChanges document -->
<complexType name="dsCustOrdChanges"
  prodata:datasetName="dsCustOrd"
  prodata:isDsChanges="true"
  prodata:namespace="WebServiceNameSpace:ObjectName">
  <sequence>
    <any />
  </sequence>
</complexType>
<!-- dataset definition -->
<element name="dsCustOrd" prodata:proDataSet="true">
  <complexType>
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="ttCust"
               prodata:beforeTable="BIttCust">
        <complexType>
          <sequence>
            <element name="CustNum" nillable="true" type="xsd:int"/>
            <element name="Name" nillable="true" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
      <element maxOccurs="unbounded" minOccurs="0" name="ttOrder"
               prodata:beforeTable="BIttOrder">
        <complexType>
          <sequence>
            <element name="OrderNum" nillable="true" type="xsd:int"/>
            <element name="CustNum" nillable="true" type="xsd:int"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
```

Note: *ObjectName* refers to the object's name as defined in ProxyGen. Each object gets its own XML Schema.

Note that the ProDataSet parameter is described in two parts:

- An arbitrary complex type (`<any />`) for the OpenEdge `datasetChanges` document
- The ProDataSet definition specifying the before-image tables with the `prodata:beforeTable` attribute

The RPC styles use the same structure for the dataset definition.

Use NAMESPACE-URI attributes

If you use ProDataSets with `NAMESPACE-URI` attributes, the WSDL contains a separate `<schema>` element in the `<types>` section for each namespace. The `<schema>` element for each namespace contains the definitions for all ProDataSets that share that namespace. In the Doc/Lit SOAP format, the primary `<schema>` element imports all the other namespaces.

Define DATASET-HANDLE (dynamic ProDataSet) parameters

DATASET-HANDLE parameters pass both the schema and data, because the dynamic ProDataSet schema is not known at compile time. Thus, for ProDataSet parameters, OpenEdge Web services map an ABL DATASET-HANDLE to an arbitrary complex type (<any>). There is a single definition used for all ProDataSet parameters in all supported SOAP formats. The following WSDL sample shows this common ProDataSet parameter definition:

Common DATASET-HANDLE definition for all SOAP formats

```
<complexType name="DataSetHandleParam">
  <annotation>
    <documentation>This is the schema definition for an OpenEdge dynamic
     ProDataSet parameter. The first element in this sequence must be a
     w3c XML Schema document describing the definition of the ProDataSet.
     The second element contains the serialized data.
    </documentation>
  </annotation>
  <sequence>
    <any maxOccurs="2" minOccurs="2"/>
  </sequence>
</complexType>
```

All dynamic ProDataSet parameters share the <complexType> definitions in the WSDL. Parameters without before-image data use one <complexType>, and parameters with before-image data use another <complexType>. The before-image version uses the following <complexType>:

```
<complexType name="DataSetHandleChangesParam" prodata:isDsChanges="true">
  <annotation>
    <documentation>This is the schema definition for an OpenEdge dynamic
     ProDataSet parameter. The first element in this sequence must be a
     w3c XML Schema document describing the definition of the ProDataSet.
     The second element contains the serialized data, including before-image
     data.
    </documentation>
  </annotation>
  <sequence>
    <any maxOccurs="2" minOccurs="2"/>
  </sequence>
</complexType>
```

The client application must create (for input) and parse (for output) the XML Schema along with the data for the parameter. How the client inserts the input schema and data in request messages and how it parses the output schema and data from response messages depends entirely on the client application.

C

ABL Elements to Consume SOAP Web Services

Many elements of ABL support consumption of OpenEdge SOAP Web services, including some that are extensions to other functionality and some that are unique for interacting with Web services. The following sections describe all of these elements, including.

- Handles to consume a web service
- Statements to consume SOAP web services
- Attributes methods and events for consuming Web services

For details, see the following topics:

- Handles to consume a web service
- Statements to consume SOAP web services
- Attributes, methods, and events to consume web services

Handles to consume a web service

The following table lists the ABL handles that are either valid only for consuming a Web service or have special application in Web service client programming.

Table 15: Handles to consume Web services

ABL handle	Description
Asynchronous request object handle	A type of handle that maintains the status of an asynchronous request in an ABL client application. This handle provides methods and attributes that allow you to check the status of an asynchronous Web service operation.
`ERROR-STATUS` system handle	Provides access to the error information returned from processing a Web service request. If the error resulted from a SOAP fault, the SOAP fault is returned as a SOAP-fault object.
Procedure object handle	A handle to a type of procedure object (*Web service procedure object*) that provides the interface to a port type of a Web service. port type operations are represented as ABL internal procedures and user-defined functions within the procedure object that represents the specified port type. A single Web service binding supports one associated port type.
`SELF` system handle	In the context of an asynchronous event procedure or method, returns the asynchronous request object handle of the completed request for which the event procedure or method is executing.
Server object handle	A handle to a server object that can provide a logical connection (or binding) to a Web service from an ABL client application. The handle provides methods and attributes that allow you to logically connect (or bind) the server object to a Web service and manage the Web service binding.
`SESSION` system handle	Maintains ABL session status, including a list of all server objects created by the session to bind Web services.
SOAP-fault object handle	A handle to an object that contains SOAP fault information for a Web service request. This system handle provides the SOAP-fault object handle as the value of its `ERROR-OBJECT-DETAIL` attribute.
SOAP fault-detail object handle	A handle to an object that references any `<detail>` element returned in a SOAP fault message generated for a Web service request. The SOAP-fault object handle provides any SOAP fault-detail object handle as the value of its `SOAP-FAULT-DETAIL` attribute.
SOAP header object handle	A handle to an object that contains the SOAP header for a SOAP request or response message generated for a Web service operation. This can be either an existing header from a SOAP response message, or a newly-created header for a pending SOAP request message.

ABL handle	Description
SOAP header-entryref object handle	A handle to an object that references an entry of a SOAP header, either an existing entry from the header of a SOAP response message, or a newly-created entry for a pending SOAP request message.
X-document object handle	A handle to an XML document object created for any SOAP message content that you must access as XML.
X-noderef object handle	A handle to any XML element (node object) within an XML document object that contains SOAP message content. This allows you to access the actual text of elements in the XML document retrieved as X-noderef objects.

Statements to consume SOAP web services

The following table lists the ABL statements that are either valid only for consuming a web service or have special application in web service client programming.

Table 16: Statements to consume web services

ABL statement	Description
`CREATE SERVER` *`server-handle`*	Creates a server object and stores a reference to it in the specified `HANDLE` variable. You can then bind (logically connect) the server object to a web service using the server handle `CONNECT( )` method.
`CREATE SOAP-HEADER` *`header-obj-handle`*	Creates a SOAP header object used to add a SOAP header to a pending SOAP request message generated by an OpenEdge web service invocation.
`CREATE SOAP-HEADER-ENTRYREF` *`soap-obj-handle`*	Creates a SOAP header-entryref object used to add SOAP header entries to an existing SOAP header.

ABL statement	Description
`DELETE OBJECT` *handle*	Deletes certain objects, including server objects, procedure objects, SOAP header objects, SOAP header-entryref objects, and asynchronous request objects.
`DELETE PROCEDURE` *procedure-handle*	Deletes the procedure object associated with a web service.
`FUNCTION` *operationName* [`RETURNS`] *dataType* [`(` *parameter* [`,` *parameter*]...] `IN` *hPortType*.	Defines a user-defined function prototype to map a web service operation as specified by the WSDL Analyzer, which determines the need for a function to return a value.
return `=` *operationName* [`(` *parameter* [`,` *parameter*]... `)`].	Invokes a web service operation defined as a user-defined function, where *return* is a variable to receive the value of the operation `<return>` parameter element, *operationName* is the name of the operation as specified in the WSDL file and whose ABL prototype is defined using the `FUNCTION` statement, and *parameter* is an ABL function parameter as required by the WSDL file for the operation. The operation can also be invoked as part of any other ABL statement that can invoke a user-defined function.

ABL statement	Description
PROCESS EVENTS	Handles any pending PROCEDURE-COMPLETE events for asynchronous requests by triggering execution of the event procedures for all completed asynchronous requests. You can also use any blocking I/O statement, such as the WAIT-FOR statement, to handle these events.
RUN *portTypeName* [SET *hPortType*] ON SERVER *server-handle* [NO-ERROR] .	Creates and associates a procedure object with a web service, where *portTypeName* is the name of a web service port type as specified in the WSDL file and whose operations this procedure object encapsulates, *hPortType* is a HANDLE variable that is set to the handle of the created procedure object, and *server-handle* is a handle to the server object that binds the web service.
RUN *operationName* IN *hPortType* [ASYNCHRONOUS [SET *asyncRequestHandle*] [EVENT-PROCEDURE *eventInternalProcedure* [IN *procedureContext*]] [EVENT-HANDLER *event-handler-name* [EVENT-HANDLER-CONTEXT *event-handler-context*]] [(*parameter* [, *parameter*] . . .)] [NO-ERROR] .	Invokes a web service operation that does not contain a <return> parameter element, where *operationName* is the name of a web service operation specified in the WSDL file, *hPortType* is a handle to the procedure object that encapsulates the operation, and *parameter* is an ABL procedure parameter as required by the WSDL for the operation. If the operation is invoked asynchronously, *asyncRequestHandle* is a handle that is set to the asynchronous request object created for the asynchronous request, *eventInternalProcedure* is the name of an ABL internal procedure defined to handle the results of the asynchronous request, and *procedureContext* is a handle to an active ABL procedure object that encapsulates the event internal procedure. *event-handler-name* is the name of an internal procedure or a method that is specified as a callback handler. The *event-handler-context* resolves to either a handle to an active procedure or the object-oriented ABL object's instance reference containing the method specified by *event-handler-name*.
WAIT-FOR ...	Handles any pending PROCEDURE-COMPLETE events for asynchronous requests by forcing the event procedures or methods for all completed asynchronous requests to execute. You can also use PROCESS EVENTS or any other blocking I/O statement, such as the PROMPT-FOR statement, to handle these events.

Attributes, methods, and events to consume web services

The following table lists the ABL handle attributes and methods that are either valid only for consuming a Web service or have special application in Web service client programming.

Table 17: Attributes, methods, and events to consume Web services

ABL attribute, method, or event	Description
ACTOR	(SOAP Version 1.1 only) A CHARACTER attribute on a SOAP header-entryref object handle that returns the value of the actor attribute specified in the associated SOAP header entry. Replaced in SOAP Version 1.2 by the ROLE attribute, and assumes the value of that attribute if used in the context of a SOAP 1.2 connection.
ADD-HEADER-ENTRY( )	A LOGICAL method on the SOAP header object handle that creates a new SOAP header entry and attaches it to the SOAP header. A specified SOAP header-entryref object handle references the new header entry.
ASYNC-REQUEST-COUNT	An INTEGER attribute that: • On a server object handle, returns the number of active asynchronous requests submitted to this Web service. • On a procedure handle, returns the number of currently outstanding asynchronous requests for this procedure object. Can be nonzero only if the PROXY and PERSISTENT attributes are both set to TRUE.
CANCELLED	A LOGICAL attribute on the asynchronous request object handle that indicates if the asynchronous request was cancelled using either the CANCEL-REQUESTS( ) method or the DISCONNECT( ) method on the associated server handle.
CANCEL-REQUESTS( )	A LOGICAL method on the server object handle that: • Terminates the socket connection to all currently running asynchronous requests for this Web service and raises the STOP condition in the event procedure context for each such request. • Purges the send queue of any asynchronous requests that have not yet been executed for this Web service.
CLIENT-CONNECTION-ID	A CHARACTER attribute on the server object handle that returns the connection ID for the connection associated with this server handle. For Web services, this is the empty string.
COMPLETE	A LOGICAL attribute on the asynchronous request object handle that indicates if the asynchronous request is completed and its result is processed on the client.
CONNECT ([connection-parms])	A LOGICAL method on the server object handle that logically connects (binds) and associates a Web service to the server handle.For complete list of parameters, see Connect() method (Web service)

ABL attribute, method, or event	Description
CONNECTED( )	A LOGICAL method on the server object handle that returns TRUE if the handle is currently bound to a Web service.
DELETE-HEADER-ENTRY( )	A LOGICAL method on the SOAP header-entryref object handle that deletes the underlying SOAP header entry and all of its content, but does not delete the SOAP header-entryref object used to reference the deleted header entry.
DISCONNECT( )	A LOGICAL method on the server object handle that disconnects from and removes all reference to the Web service currently associated with the server handle. Any running or pending asynchronous requests to the Web service submitted by this client are also cancelled.
ERROR	A LOGICAL attribute on the asynchronous request object handle that indicates that an ERROR condition was returned as a result of processing a Web service request.
ERROR-OBJECT	An attribute on the asynchronous request object handle that returns an object reference to an instance of a class that implements Progress.Lang.Error.
ERROR-OBJECT-DETAIL	A HANDLE attribute an the ERROR-STATUS system handle that references the SOAP-fault object for any Web service request that returns a SOAP fault that is trapped using the ABL NO-ERROR option.
EVENT-HANDLER	A character expression that results in the name of an internal procedure or a method to be run as the event procedure for this asynchronous event.
EVENT-HANDLER-CONTEXT	The event-handler-context value resolves to either a handle to an active procedure or the object-oriented ABL object's instance reference, where the event method for this asynchronous request is defined.
EVENT-PROCEDURE	A CHARACTER attribute on the asynchronous request object handle that contains the name of the internal procedure to be run as the event procedure for this asynchronous request.
EVENT-PROCEDURE-CONTEXT	A HANDLE attribute on the asynchronous request object handle that contains the procedure handle of the active procedure context where the event procedure for this asynchronous request is defined.
FIRST-ASYNC-REQUEST( )	A HANDLE method on the server object handle that returns the first entry in the list of all current asynchronous request handles for the specified Web service.

ABL attribute, method, or event	Description
FIRST-PROCEDURE	A HANDLE attribute on the server object handle that references the first procedure object associated with the Web service port type bound to this server object.
FIRST-SERVER	A HANDLE attribute on the SESSION system handle that returns the handle to the first entry in the chain of server handles for the session, including server handles for Web services.
GET-HEADER-ENTRY( )	A LOGICAL method on the SOAP header object handle that associates a specified SOAP header-entryref object handle with a specified entry in the associated SOAP header.
GET-NODE( )	A LOGICAL method that: • On the SOAP header-entryref object handle, returns a handle to an X-noderef object that is the root node of a DOM tree containing the parsed XML for the underlying SOAP header entry. • On the SOAP fault-detail object handle, returns a handle to an X-noderef object that is the root node of a DOM tree containing the parsed XML for the underlying SOAP detail information.
GET-SERIALIZED( )	A LONGCHAR method that: • On the SOAP header-entryref object handle, returns the XML for the underlying SOAP header entry in serialized form. • On the SOAP fault-detail object handle, returns the XML for the underlying SOAP fault detail information in serialized form.
LAST-ASYNC-REQUEST( )	A HANDLE method on the server object handle that returns the last entry in the list of all current asynchronous request handles for the specified Web service.
LAST-PROCEDURE	A HANDLE attribute on the server object handle that references the last procedure object associated with the Web service port type bound to this server object.
LAST-SERVER	A HANDLE attribute on the SESSION system handle that returns the handle to the last entry in the chain of server handles for the session, including server handles for Web services.
LOCAL-NAME	A CHARACTER attribute on the SOAP header-entryref object handle that returns the unqualified part of name specified for the associated SOAP header entry element.

ABL attribute, method, or event	Description
MUST-UNDERSTAND	A LOGICAL attribute on the SOAP header-entryref object handle that returns the value of the mustUnderstand attribute specified in the associated SOAP header.
NAME	A CHARACTER attribute that: • On a Web service server object handle, returns the name of the Web service for use in the OpenEdge Application Debugger. By default, this name is set to the URL of the Web service. • On a Web service procedure object handle, returns the name of the port type in the WSDL file that specifies the Web service operations encapsulated by this procedure object. This is also the value of *portTypeName* in the RUN portTypeName statement used to instantiate this procedure object. • On a SOAP header object handle, returns the qualified name of the SOAP header ("*namespacePrefix*:HEADER", where *namespacePrefix* is usually "SOAP" or "SOAP-ENV"). • On a SOAP header-entryref object handle, returns the qualified name of the SOAP header entry ("*namespacePrefix*:*localName*").
NAMESPACE-URI	A CHARACTER attribute on the SOAP header-entryref object handle that returns the namespace URI prefixed to the associated SOAP header entry element's name.
NEXT-SIBLING	A HANDLE attribute that: • On an asynchronous request object handle, returns the next entry in the list of asynchronous request handles for asynchronous operation requests submitted for the same Web service. • On a Web service procedure handle, returns the next entry in the list of all procedure objects bound to the same Web service port type. • On a server object handle, returns the next entry in the list of all server handles created for the current ABL session, regardless of subtype (see the SUBTYPE attribute).
NUM-HEADER-ENTRIES	An INTEGER attribute on the SOAP header object handle that returns the number of entries attached to the SOAP header.
PERSISTENT	A LOGICAL attribute on procedure handles that returns TRUE for a procedure object associated with a Web service.

ABL attribute, method, or event	Description
PERSISTENT-PROCEDURE	A HANDLE attribute on the asynchronous request object handle that returns the handle to the procedure object for the specified asynchronous Web service request.
PREV-SIBLING	A HANDLE attribute that: • On an asynchronous request object handle, returns the previous entry in the list of asynchronous request handles for asynchronous operation requests submitted for the same Web service. • On a Web service procedure handle, returns the previous entry in the list of all procedure objects bound to the same Web service port type. • On a server object handle, returns the previous entry in the list of all server handles created for the current ABL session, regardless of subtype (see the SUBTYPE attribute).
PROCEDURE-COMPLETE	The event returned for an asynchronous request object handle that indicates the associated Web service request has completed execution and, as a result, causes execution of the event procedure as specified by the EVENT-PROCEDURE and EVENT-PROCEDURE-CONTEXT attributes.
PROCEDURE-COMPLETE	The event returned for an asynchronous request object handle that indicates the associated Web service request has completed execution and, as a result causes execution of the: • event procedure, as specified by the EVENT-PROCEDURE and EVENT-PROCEDURE-CONTEXT attributes or • event method, as specified by the EVENT-HANDLER and EVENT-HANDLER-CONTEXT attributes.
PROCEDURE-NAME	A CHARACTER attribute on the asynchronous request object handle that provides the name of the Web service operation executed to instantiate this asynchronous request handle.
PROXY	A LOGICAL attribute on procedure object handles that is TRUE if the procedure handle references a procedure object associated with a Web service.
ROLE	(SOAP 1.2 only) A CHARACTER attribute on a SOAP header-entryref object handle that returns the value of the role attribute specified in the associated SOAP header entry. Replaces the ACTOR attribute used in SOAP 1.1.

ABL attribute, method, or event	Description
SERVER	A HANDLE attribute that: • On a procedure object handle, returns the handle to the Web service server object with which the procedure object is associated. Valid only if the PROXY and PERSISTENT attributes are both TRUE. • On an asynchronous request object handle, returns the server handle of the Web service where this asynchronous request was invoked.
SET-ACTOR( )	(SOAP 1.1 only) A LOGICAL method on the SOAP header-entryref object handle that sets the value of the actor attribute in the underlying SOAP header entry. This method completes and returns TRUE only if there is an underlying SOAP header entry (XML) associated with the object handle. Replaced by the SET-ROLE method in SOAP 1.2.
SET-CALLBACK( )	A LOGICAL method on the procedure object handle that associates a specified internal procedure or method with an ABL callback. For Web services, the two supported types of ABL callbacks, include: • **"REQUEST-HEADER"** — SOAP request header callback • **"RESPONSE-HEADER"** — SOAP Response header callback The internal procedures or methods associated with these callbacks provide access to the SOAP headers of the request and response messages sent for all the Web service operations encapsulated by the procedure object.
SET-CALLBACK-PROCEDURE( )	A LOGICAL method on the procedure object handle that associates a specified internal procedure with an ABL callback. For Web services, the two supported types of ABL callbacks, include: • **"REQUEST-HEADER"** — SOAP request header callback • **"RESPONSE-HEADER"** — SOAP Response header callback The internal procedures associated with these callbacks provide access to the SOAP headers of the request and response messages sent for all the Web service operations encapsulated by the procedure object.
SET-MUST-UNDERSTAND( )	A LOGICAL method on the SOAP header-entryref object handle that sets the value of the mustUnderstand attribute in the underlying SOAP header entry. This method completes and returns TRUE only if there is an underlying SOAP header entry (XML) associated with the object handle.

ABL attribute, method, or event	Description
SET-NODE( )	A LOGICAL method on the SOAP header-entryref object handle that replaces the header entry referenced by this SOAP header-entryref object with a specified DOM sub-tree (parsed XML) that is assumed to represent a SOAP header entry element.
SET-ROLE( )	(SOAP 1.2 only) A LOGICAL method on the SOAP header-entryref object handle that sets the value of the role attribute in the underlying SOAP header entry. This method completes and returns TRUE only if there is an underlying SOAP header entry (XML) associated with the object handle. Replaces the SET_ACTOR method used in SOAP 1.1.
SET-SERIALIZED( )	A LOGICAL method on the SOAP header-entryref object handle that replaces the header entry referenced by this SOAP header-entryref object with serialized XML that is assumed to parse into a DOM sub-tree that represents a SOAP header entry element.
SOAP-FAULT-CODE	A CHARACTER attribute on the SOAP-fault object handle that contains the value of the faultcode attribute for the SOAP fault, which identifies the fault.
SOAP-FAULT-STRING	A CHARACTER attribute on the SOAP-fault object handle that contains the value of the faultstring attribute for the SOAP fault, which provides a human-readable description of the fault.
SOAP-FAULT-ACTOR	(SOAP 1.1 only) A CHARACTER attribute on the SOAP-fault object handle that contains the value of the faultactor attribute for the SOAP fault, which is a URI that identifies the Web service returning the fault. Replaced in SOAP Version 1.2 by a combination of SOAP-FAULT-ROLE and SOAP-FAULT-NODE.
SOAP-FAULT-DETAIL	A HANDLE attribute on the SOAP-fault object handle that references the SOAP fault-detail object, which contains application-specific error information.
SOAP-FAULT-MISUNDERSTOOD-HEADER	(SOAP 1.2 only) A CHARACTER attribute on the SOAP-fault object handle that returns the list of fully qualified names of SOAP headers, if any, for which mandatory processing (as designated by the SOAP-ENV:mustUnderstand attribute) failed.
SOAP-FAULT-NODE	(SOAP 1.2 only) A CHARACTER attribute on the SOAP-fault object handle that returns the URI of the SOAP node that generated the SOAP-fault object, if available. In combination with SOAP-FAULT-ROLE, replaces the SOAP-FAULT-ACTOR attribute used in SOAP 1.1.

ABL attribute, method, or event	Description
SOAP-FAULT-ROLE	(SOAP 1.2 only) A CHARACTER attribute on the SOAP-fault object handle that contains the value of the faultrole attribute for the SOAP fault, which is a URI that identifies the Web service returning the fault. In combination with SOAP-FAULT-NODE, replaces the SOAP-FAULT-ACTOR attribute used in SOAP 1.1.
SOAP-FAULT-SUBCODE	(SOAP 1.2 only) A CHARACTER attribute on the SOAP-fault object handle that returns the list of fully qualified sub-code names for the SOAP-fault object.
SOAP-VERSION	A CHARACTER attribute on the server object handle that returns the version number of the SOAP connection current.
STOP	A LOGICAL attribute on the asynchronous request object handle that is set to TRUE, if the asynchronous request was executing when the client invoked a CANCEL-REQUESTS( ) method.
SUBTYPE	A CHARACTER attribute that returns the subtype of a handle. For a server object handle, this is: • **"WEBSERVICE"** — For a server object bound to a Web service The default value for an unbound server object handle is the Unknown value (?).
TYPE	A CHARACTER attribute that returns the handle type, which: • On a server object handle is "SERVER" • On a procedure object handle is "PROCEDURE" • On a SOAP header object handle is "SOAP-HEADER" • On a SOAP header-entryref object handle is "SOAP-HEADER-ENTRYREF" • On a SOAP-fault object handle is "SOAP-FAULT" • On a SOAP fault-detail object handle is "SOAP-FAULT-DETAIL" • On an X-document object handle is "X-DOCUMENT" • On an X-noderef object handle is "X-NODEREF" • On an asynchronous request object handle is "ASYNC-REQUEST"

D

Data Type Conversion Rules for ABL Calls to SOAP Web Services

This appendix describes how OpenEdge converts between the following ABL data types and XML Schema data types in parameters passed to SOAP Web services called from ABL. Any casting between an ABL type and an unsupported XML Schema type or an invalid match between an ABL type and a supported XML Schema type raises an ABL run-time error, as detailed in the following sections.

Note: The tables in this chapter describe data transformations for the ABL `INPUT` and `OUTPUT` parameter modes. The `INPUT-OUTPUT` parameter mode behaves in the same manner.

For details, see the following topics:

- Data type casting
- CHARACTER or LONGCHAR
- DATE
- DATETIME
- DATETIME-TZ
- DECIMAL
- INT64
- INTEGER
- LOGICAL

- MEMPTR or RAW

Data type casting

OpenEdge supports a set of alternative ABL data types (in addition to a suggested data type) to represent the value for an XML Schema data type in ABL. These alternative data types essentially force the Web service invocation to cast the value between the specified native ABL representation and the corresponding XML Schema data type. The result of this casting might not preserve as much accuracy as the suggested mapping.

The following table shows all the supported castings (alternative mappings) between the XML Schema and ABL data types. The suggested ABL data type mapping for each XML Schema type appears in bold font.

Note: OpenEdge supports no castings for the `RECID`, `ROWID`, or `HANDLE` ABL data types.

Table 18: Supported casts between data types

XML Schema data type	ABL data type
anyURI	**CHARACTER** LONGCHAR
base64Binary	CHARACTER LONGCHAR MEMPTR**RAW**
boolean	CHARACTER DECIMAL INT64 INTEGER **LOGICAL** LONGCHAR
byte	CHARACTER DECIMAL INT64 **INTEGER** LOGICAL LONGCHAR
date	CHARACTER **DATE** DATETIME DATETIME-TZ
dateTime	CHARACTER DATE DATETIME **DATETIME-TZ**
decimal	CHARACTER **DECIMAL** INT64 INTEGER LOGICAL LONGCHAR
double	**CHARACTER** DECIMAL INT64 INTEGER LOGICAL LONGCHAR
duration	

XML Schema data type	ABL data type
	CHARACTER LONGCHAR
ENTITIES	**CHARACTER** LONGCHAR
ENTITY	**CHARACTER** LONGCHAR
float	**CHARACTER** DECIMAL INT64 INTEGER LOGICAL LONGCHAR
gDay	**CHARACTER** LONGCHAR
gMonth	**CHARACTER** LONGCHAR
gMonthDay	**CHARACTER** LONGCHAR
gYear	**CHARACTER** LONGCHAR
gYearMonth	**CHARACTER** LONGCHAR
hexBinary	CHARACTER LONGCHAR MEMPTR**RAW**
ID	**CHARACTER** LONGCHAR
IDREF	

XML Schema data type	ABL data type
	CHARACTER LONGCHAR
IDREFS	**CHARACTER** LONGCHAR
int	CHARACTER DECIMAL INT64 **INTEGER** LOGICAL LONGCHAR
integer	CHARACTER **DECIMAL** INT64 INTEGER LOGICAL LONGCHAR
Language	**CHARACTER** LONGCHAR
long	CHARACTER DECIMAL **INT64** INTEGER LOGICAL LONGCHAR
Name	**CHARACTER** LONGCHAR
NCName	**CHARACTER** LONGCHAR
negativeInteger	CHARACTER **DECIMAL** INT64 INTEGER LONGCHAR
NMTOKEN	**CHARACTER** LONGCHAR

XML Schema data type	ABL data type
NMTOKENS	**CHARACTER** LONGCHAR
nonNegativeInteger	CHARACTER **DECIMAL** INT64 INTEGER LOGICAL LONGCHAR
nonPositiveInteger	CHARACTER **DECIMAL** INT64 INTEGER LOGICAL LONGCHAR
normalizedString	**CHARACTER** DATE DATETIME DATETIME-TZ DECIMAL INT64 INTEGER LOGICAL LONGCHAR
NOTATION	**CHARACTER** LONGCHAR
positiveInteger	CHARACTER **DECIMAL** INT64 INTEGER LONGCHAR
qName	**CHARACTER** LONGCHAR
short	CHARACTER DECIMAL INT64 **INTEGER** LOGICAL LONGCHAR
string	

XML Schema data type	ABL data type
	CHARACTER DATE DATETIME DATETIME-TZ DECIMAL INT64 INTEGER LOGICAL LONGCHAR
time	CHARACTER INT64 **INTEGER** LONGCHAR
token	**CHARACTER** LONGCHAR
unsignedByte	CHARACTER DECIMAL INT64 **INTEGER** LOGICAL LONGCHAR
unsignedInt	CHARACTER **DECIMAL** INT64 INTEGER LOGICAL LONGCHAR
unsignedLong	CHARACTER **DECIMAL** INT64 INTEGER LOGICAL LONGCHAR
unsignedShort	CHARACTER DECIMAL INT64 **INTEGER** LOGICAL LONGCHAR

CHARACTER or LONGCHAR

The following table describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL CHARACTER or LONGCHAR type and XML Schema.

Table 19: ABL data type conversion—CHARACTER or LONGCHAR

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
All XML Schema types where CHARACTER is the suggested ABL mapping (see "XML Schema and ABL Data Type Mappings" in *Use XML with ABL applications* guide).	INPUT[3]	Attempts to format the ABL string value to match the specified XML Schema type for inclusion in the SOAP request message.[4]
	OUTPUT[3]	Converts the value from the formatting of the specified XML Schema type before copying it into the ABL parameter as a string value.[4]
All XML Schema types where CHARACTER or LONGCHAR is **not** the suggested ABL mapping (see "XML Schema and ABL Data Type Mappings" in *Use XML with ABL applications* guide).	INPUT[3]	Copies the string value directly, with no attempt to format it, for inclusion in the SOAP request message. OpenEdge performs minimal validation to ensure that the string contains data formatted according to the rules of the XML Schema type.
	OUTPUT[3]	Copies the XML Schema value directly from the SOAP response message to the ABL parameter unchanged from its XML Schema format.

DATE

The following table describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL DATE type and XML Schema.

[4] The process of serializing and de-serializing CHARACTER data inserts and removes XML character references, for example, replacing "<" with "<".

[3] On both INPUT and OUTPUT parameters, OpenEdge translates the value between the cpinternal code page and Unicode.

Table 20: ABL data type cast—DATE

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
`string` `normalizedString`	`INPUT`	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.
	`OUTPUT`	De-serializes the string value from the SOAP response message into the ABL parameter as if the ABL `DATE( )` function was called on the string in an OpenEdge session started with the `–d ymd` startup parameter.
`dateTime`	`INPUT`	Serializes the ABL parameter according to the XML Schema serialization rules, with the time assumed to be 12:00 midnight local time.
	`OUTPUT`	De-serializes the XML Schema value into the ABL parameter according to the XML Schema serialization rules with the value set to the local time adjusted from any time and time zone information (which is stripped away). If there is no time and time zone information the value is stored as is. If you need to maintain the time information, use the ABL `DATETIME` data type. If you need to maintain the time zone information, use the ABL `DATETIME-TZ` data type.
`date`	`INPUT`	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules with no time zone.
	`OUTPUT`	De-serializes the XML Schema value into the ABL parameter with the date adjusted for any time zone information and the time zone information stripped away. If you need to maintain the time zone information, use the ABL `DATETIME-TZ` data type.

DATETIME

The following table describes the supported castings in each ABL parameter mode (`INPUT` and `OUTPUT`) between the ABL `DATETIME` type and XML Schema.

Table 21: ABL data type cast—DATETIME

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
`string` `normalizedString`	`INPUT`	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.
	`OUTPUT`	De-serializes the string value from the SOAP response message into the ABL parameter as if the ABL `DATETIME( )` function was called on the string.
`dateTime`	`INPUT`	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules without a time zone.
	`OUTPUT`	De-serializes the XML Schema value from the SOAP response message into the ABL parameter according to the XML Schema serialization rules with the value set to the local time adjusted from any time zone information (which is stripped away). If there is no time zone information the value is stored as is.
`date`	`INPUT`	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules. The time part of the `DATETIME` value is lost.
	`OUTPUT`	De-serializes the XML Schema value from the SOAP response message into the ABL parameter according to the XML Schema serialization rules with the date adjusted for any time zone information (which is stripped away). The ABL `DATETIME` value has the time set to 12:00 midnight.

DATETIME-TZ

The following table describes the supported castings in each ABL parameter mode (`INPUT` and `OUTPUT`) between the ABL `DATETIME-TZ` type and XML Schema.

Table 22: ABL data type cast—DATETIME-TZ

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
`string` `normalizedString`	`INPUT`	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.
	`OUTPUT`	De-serializes the string value from the SOAP response message into the ABL parameter according to the XML Schema serialization rules as if the ABL `DATETIME-TZ( )` function was called on the string.

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
dateTime	INPUT	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules including the time zone.
	OUTPUT	De-serializes the XML Schema value from the SOAP response message into the ABL parameter according to the XML Schema serialization rules.
date	INPUT	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules. The time part of the DATETIME-TZ value is lost, but the time zone part of the DATETIME-TZ value is retained.
	OUTPUT	De-serializes the XML Schema value from the SOAP response message into the ABL parameter according to the XML Schema serialization rules. The ABL DATETIME-TZ value has the time set to 12:00 midnight.

DECIMAL

The following table describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL DECIMAL type and XML Schema.

Table 23: ABL data type cast—DECIMAL

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
string normalizedString	INPUT	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.
	OUTPUT	De-serializes the string value from the SOAP response message into the ABL parameter as if the ABL DECIMAL( ) function was called on the string.
boolean	INPUT	Serializes an ABL DECIMAL value of 0 to false and all other values for the ABL parameter to true.
	OUTPUT	De-serializes a value of false to an ABL DECIMAL value of 0, and de-serializes a value of true to a DECIMAL value of 1.

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
decimal	INPUT	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.
	OUTPUT	De-serializes the XML Schema value into the ABL parameter according to the XML Schema serialization rules. If the XML Schema decimal value overflows the ABL DECIMAL, the AVM raises a run-time error.
float double	INPUT	Serializes the ABL DECIMAL value according to XML Schema serialization rules. Some of the least significant digits might be lost.
	OUTPUT	De-serializes the XML Schema value to the equivalent ABL DECIMAL value. If the value is outside the range of the ABL DECIMAL, the AVM raises a run-time error.
integer nonPositiveInteger negativeInteger long int short byte nonNegativeInteger unsignedLong unsignedInt unsignedShort unsignedByte positiveInteger	INPUT	Rounds the ABL DECIMAL value to an integer and serializes the result based on the serialization rules for the XML Schema type. Data might be lost. If the DECIMAL value is outside the valid range of the XML Schema type, the AVM raises a run-time error.
	OUTPUT	De-serializes the XML Schema value into the ABL parameter as if the ABL DECIMAL( ) function was called on the value. If the value in the SOAP response message is outside the range of the DECIMAL type, the AVM raises a run-time error.

INT64

The following table describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL INT64 type and XML Schema.

Table 24: ABL data type cast—INT64

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
string normalizedString	INPUT	Serializes the ABL parameter into the XML Schema value as if the ABL STRING( ) function was called on the INT64 value.
	OUTPUT	De-serializes the XML Schema value as if the INT64( ) function was called on string value.

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
boolean	INPUT	Serializes an INT64 value of 0 to false and all other values to true.
	OUTPUT	De-serializes a value of false to an ABL INT64 value of 0, and de-serializes a value of true to an INT64 value of 1.
decimal	INPUT	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.
	OUTPUT	De-serializes the XML Schema value as if the INT64( ) function was called on the serialized ABL DECIMAL value. This rounds any positions after the decimal point. If the XML Schema decimal value is outside the range of an ABL INT64, the AVM raises a run-time error.
float	INPUT	Translate the INT64 to a C value, cast to a C float value, and then serialize the ABL INT64 value to a float according to the XML Schema rules. Some of the least significant digits of the INT64 value are lost if the number of digits in the value is large.
	OUTPUT	De-serializes the XML Schema value into a C float value and then cast the C value to ABL INT64. If the XML Schema value overflows or underflows an ABL INT64, the AVM raises a run-time error.
double	INPUT	Translate the INT64 to a C value, cast to a C double value, and then serialize the value to a double according to the XML Schema rules.
	OUTPUT	De-serializes the XML Schema value into a C double value and cast it to an ABL INT64. If the XML Schema value overflows or underflows an ABL INT64, the AVM raises a run-time error.
integer nonPositiveInteger negativeInteger long nonNegativeInteger unsignedLong unsignedInt positiveInteger	INPUT	Serializes the ABL parameter into the SOAP request message according to the serialization rules for the XML Schema data type.
	OUTPUT	De-serializes the XML Schema value as if the ABL INT64( ) function was called on the value. If the XML Schema value is outside the valid range of an ABL INT64, the AVM raises a run-time error.

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
long	INPUT	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.
	OUTPUT	De-serializes the XML Schema value as if the ABL INT64( ) function was called on the value.
int short byte unsignedShort unsignedByte	INPUT	Serializes the ABL parameter into the SOAP request message according to the serialization rules for the XML Schema data type. If the ABL INT64 value is outside the valid range of the XML Schema type, the AVM raises a run-time error.
	OUTPUT	De-serializes the XML Schema value as if the ABL INT64( ) function was called on the value.

INTEGER

The following table describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL INTEGER type and XML Schema.

Table 25: ABL data type cast—INTEGER

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
string normalizedString	INPUT	Serializes the ABL parameter into the XML Schema value as if the ABL STRING( ) function was called on the INTEGER value.
	OUTPUT	De-serializes the XML Schema value as if the INTEGER( ) function was called on string value.
boolean	INPUT	Serializes an INTEGER value of 0 to false and all other values to true.
	OUTPUT	De-serializes a value of false to an ABL INTEGER value of 0, and de-serializes a value of true to an INTEGER value of 1.
decimal	INPUT	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.
	OUTPUT	De-serializes the XML Schema value as if the INTEGER( ) function was called on the serialized ABL DECIMAL value. This rounds any positions after the decimal point. If the XML Schema decimal value is outside the range of an ABL INTEGER, the AVM raises a run-time error.

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
float	INPUT	Serializes the ABL INTEGER value to a float according to the XML Schema rules. Some of the least significant digits of the INTEGER value are lost if the number of digits in the value is large.
	OUTPUT	De-serializes the XML Schema value into an ABL INTEGER. If the XML Schema value overflows or underflows an ABL INTEGER, the AVM raises a run-time error.
double	INPUT	Serializes the ABL INTEGER value to a double according to the XML Schema rules.
	OUTPUT	De-serializes the XML Schema value into an ABL INTEGER. If the XML Schema value overflows or underflows an ABL INTEGER, the AVM raises a run-time error.
integer nonPositiveInteger negativeInteger long nonNegativeInteger unsignedLong unsignedInt positiveInteger	INPUT	Serializes the ABL parameter into the SOAP request message according to the serialization rules for the XML Schema data type.
	OUTPUT	De-serializes the XML Schema value as if the ABL INTEGER( ) function was called on the value. If the XML Schema value is outside the valid range of an ABL INTEGER, the AVM raises a run-time error.
int	INPUT	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.
	OUTPUT	De-serializes the XML Schema value as if the ABL INTEGER( ) function was called on the value.
short byte unsignedShort unsignedByte	INPUT	Serializes the ABL parameter into the SOAP request message according to the serialization rules for the XML Schema data type. If the ABL INTEGER value is outside the valid range of the XML Schema type, the AVM raises a run-time error.
	OUTPUT	De-serializes the XML Schema value as if the ABL INTEGER( ) function was called on the value.
time	INPUT	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules without a time zone. The ABL INTEGER value is treated as the number of seconds since midnight.
	OUTPUT	De-serializes the XML Schema value to the number of seconds since midnight. The conversion loses fractions of seconds. If a time zone is specified in the value, the time is adjusted to the same instant in the local time zone.

LOGICAL

The following table describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL LOGICAL type and XML Schema.

Table 26: ABL data type cast—LOGICAL

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
string normalizedString	INPUT	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.
	OUTPUT	De-serializes the XML Schema string type as if it was a serialized boolean value.
boolean	INPUT	Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.
	OUTPUT	De-serializes the XML Schema value into the ABL parameter according to the XML Schema serialization rules.
decimal float double	INPUT	Serializes an ABL LOGICAL value of true to an XML Schema type value of 1, and serializes a LOGICAL value of false to an XML Schema type value of 0.
	OUTPUT	De-serializes an XML Schema value of 0 to an ABL LOGICAL value of false, and all other XML Schema values to a LOGICAL value of true.
integer nonPositiveInteger long int short byte nonNegativeInteger unsignedLong unsignedIn unsignedShort unsignedByte	INPUT	Serializes an ABL LOGICAL value of true to an XML Schema type value of 1 (or -1 for a nonPositiveInteger), and serializes a LOGICAL value of false to an XML Schema type value of 0.
	OUTPUT	De-serializes an XML Schema value of 0 to an ABL LOGICAL value of false, and all other XML Schema values to a LOGICAL value of true.

MEMPTR or RAW

The following table describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL MEMPTR or RAW type and XML Schema.

Table 27: ABL data type cast—RAW or MEMPTR

For this XML Schema type . . .	In this ABL parameter mode . . .	OpenEdge . . .
`base64Binary` `hexBinary`	`INPUT`	Serializes the value of each byte in the ABL value according to the XML Schema serialization rules for the data type. If the mapping is to a base64Binary value with the encoding attribute set, the value is base-64 encoded and copied directly from the ABL parameter without serialization.
	`OUTPUT`	De-serializes the value of each byte in the ABL value according to the XML Schema serialization rules for the data type. If the mapping is from a base64Binary value, it is always de-serialized into the ABL parameter.