Contents

Preface ............................................................... Preface–1

Part I Introduction

1. Web Services in OpenEdge ........................................... 1–1
   Web service basics .................................................. 1–2
   Industry standards supported by OpenEdge ......................... 1–2
   WSDL files .......................................................... 1–3
   SOAP messages ...................................................... 1–3
   XML Schema ......................................................... 1–4
   Creating OpenEdge Web services ................................... 1–5
   Defining requirements ............................................... 1–6
   Defining the Web service in ProxyGen ............................. 1–7
   Deploying the Web service ......................................... 1–8
   Putting the Web service into production .......................... 1–10
   Creating ABL clients to consume Web services .................... 1–11
   Obtain the Web service’s WSDL and documentation ............... 1–12
   Run the WSDL Analyzer ............................................. 1–12
   Build your client .................................................... 1–13
   Test your client ...................................................... 1–14
   Consuming Web service example .................................. 1–15
   Sample Web service applications ................................. 1–17

Part II Creating OpenEdge Web Services

2. Exposing AppServer-enabled Applications as OpenEdge Web Services .... 2–1
   Deciding how to expose your application .......................... 2–2
   Web service objects and the WSDL file ......................... 2–2
   Session Models and object IDs ................................... 2–3
   SOAP message formats ............................................ 2–6
   Preparing the AppServer-enabled application ..................... 2–7
3. Building Clients for OpenEdge Web Services
   - Creating client interfaces from WSDL
     - WSDL element overview for OpenEdge Web services
   - SOAP format impact on generated client interfaces
     - Method signatures
     - Method return values
   - Client programming for different session models
     - Programming clients for session-managed Web services
     - Programming clients for session-free Web services
   - Overview of calling methods on Web service objects
     - Client interface methods for session-managed AppObjects
     - Client interface methods for session-free AppObjects
     - Client interface methods for SubAppObjects and ProcObjects
   - Retrieving and sending object IDs—handling SOAP headers
     - Defining object IDs in WSDL
     - Using object IDs in a SOAP message
   - Defining and passing Web service method parameters
     - Client data type mapping
     - Relational data types
     - Date and time data
     - Dedicated ABL data types
     - Array parameters
   - Mapping relational data
     - Defining TABLE (static temp-table) parameters
     - Defining TABLE-HANDLE (dynamic temp-table) parameters
     - Defining DATASET (static ProDataSet) parameters
     - Defining DATASET-HANDLE (dynamic ProDataSet) parameters
     - Additional considerations
   - Mapping ABL procedures to SOAP messages
     - ABL procedure prototype to WSDL operation
     - WSDL operation to client method prototype
     - Client method call to SOAP request/response message pair
   - Handling Web service errors and SOAP faults
     - Client programming for SOAP faults

4. Sample Code with SOAP Messages for OpenEdge Web Services
   - Sample Web service specifications
   - Consuming a session-managed Web service
   - Consuming a session-free Web service
   - Running an ABL non-persistent procedure
   - Creating and using a ProcObject
     - ProcObject session context
     - ProcObject IDs
     - ProcObject class factory methods
   - Running an internal procedure or user-defined function
   - Releasing an object
   - Passing static and dynamic temp-table parameters
     - Invoking a method with a TABLE parameter
     - Invoking a method with a TABLE-HANDLE parameter
   - Receiving a SOAP fault message
5. **Testing and Debugging OpenEdge Web Services**  
   Testing the AppServer application as a Web service  
   Unit testing from ABL  
   Testing from a Web services client  
   Setting error information levels  
   WSA and Web service logging level properties  
   Logging information  
   Hints for setting the WSA loggingLevel property  
   Hints for setting the Web service serviceLoggingLevel property  
   Setting the logging level properties  
   Identifying relevant log files  
   Working with SOAP faults  
   Setting the serviceFaultLevel property  
   Typical SOAP fault response message  
   Handling SOAP faults programmatically  
   Using a SOAP message viewer  
   Identifying errors  
   How the WSA logs Web service information  
   Errors occurring in WSA administration requests  
   Errors occurring in Web service application requests  
   Sample error scenarios  
   Scenario 1: Web service deployed but not enabled  
   Scenario 2: ABL updated, but Web service not redeployed  
   Scenario 3: Web service deployed but AppServer not running  

6. **Creating an ABL Client from WSDL**  
   Basics of an ABL client to consume Web services  
   Using the WSDL Analyzer  
   Running the WSDL Analyzer  
   Understanding the WSDL Analyzer output  
   Analyzing wrapped document literal  
   Analyzing complex data  
   Mapping XML Schema data types to ABL data types  
   Simple data types  
   Arrays  
   Complex data types  

7. **Connecting to Web Services from ABL**  
   What is a Web service connection?  
   Binding to a Web service  
   Creating a server object handle  
   Binding to a server object handle  
   Managing Web service bindings  
   Accessing a port type for a Web service  
   Accessing multiple port types  
   Accessing multiple port types simultaneously.
### 8. Invoking Web Service Operations from ABL

- Preparing to invoke operations ........................................ 8–2
- Creating a Web service procedure object ......................... 8–2
- Invoking operations .................................................. 8–3
  - Using the RUN statement ........................................ 8–4
  - Using a user-defined function .................................... 8–5
- Managing operation parameters ...................................... 8–6
- Managing complex data .............................................. 8–7
  - Types of complex data ............................................ 8–7
  - Complex data example ............................................ 8–8
  - Coding options for wrapped document literal .................. 8–10
- Managing asynchronous requests .................................... 8–12

### 9. Handling SOAP Message Headers in ABL

- SOAP message headers—an overview .............................. 9–2
  - SOAP header structure .......................................... 9–2
  - SOAP header object model ...................................... 9–4
  - Accessing SOAP header entries ................................ 9–5
  - Specifying SOAP header callback procedures at run time .... 9–6
  - Using the SET-CALLBACK-PROCEDURE( ) method .......... 9–7
- Defining header handlers ............................................ 9–8
  - Defining a response header handler ............................... 9–9
  - Defining a request header handler ................................. 9–10
- Invoking a header handler .......................................... 9–11
- Creating and managing SOAP message headers ................. 9–13
  - Reusing an unchanged SOAP response header .................. 9–14
  - Modifying a reused SOAP response header ..................... 9–17
  - Using a client-created SOAP request header .................... 9–22
- Managing memory for SOAP headers ............................... 9–26
  - SOAP header object model and DOM relationships .......... 9–26
  - Memory for the SOAP header object model and DOM .......... 9–27
- Attributes and methods for handling SOAP headers ............ 9–28
  - SOAP header object attributes and methods .................... 9–28
  - SOAP header-entryref object attributes and methods ......... 9–29

### 10. Handling Errors in ABL Requests to Web Services

- Handling SOAP faults .............................................. 10–2
- Detecting a SOAP fault ............................................ 10–3
  - SOAP faults with traditional error handling ..................... 10–3
  - SOAP faults with structured error handling ..................... 10–4
- Managing a SOAP fault ............................................. 10–6
- Examples of ABL accessing a SOAP fault ......................... 10–8
  - Traditional error handling example ............................. 10–8
  - Structured error handling example ............................... 10–9
- Debugging ABL applications that call Web services .......... 10–12
  - SOAP Viewers .................................................... 10–12
  - Using WSAViewer .................................................. 10–13
  - Using ProSOAPView ............................................... 10–14
Part IV Appendices

A. Developing a .NET Client to Consume OpenEdge Web Services ............ A–1
What is Microsoft .NET? ................................................................. A–2
Using .NET as your Web services client ........................................ A–3
VB.NET sample Web service specifications ................................. A–4
Creating the VB.NET client interface ........................................ A–5
  Adding a Web reference .......................................................... A–5
  Reference.vb—Web service client interface ............................. A–6
  Reference.vb—Open Client objects .......................................... A–6
  .NET data type mapping ......................................................... A–6
  Accessing parameter data ..................................................... A–7
  TABLE (static temp-table) parameters .................................... A–7
  TABLE-HANDLE (dynamic temp-table) parameters ................. A–8
  Sample interface object prototypes in Reference.vb .............. A–9
  Sample common interface method prototypes in Reference.vb ... A–10
  Sample ABL and VB.NET interface method prototypes .......... A–11
Developing the VB.NET client application ................................ A–14
  Creating the Web service .................................................... A–15
  Running a non-persistent (external) procedure ....................... A–15
  Creating server-side context for a ProcObject
    (running a persistent procedure) ...................................... A–16
  Running an internal procedure or user-defined function ........ A–16
  Creating a SubAppObject .................................................... A–17
  Releasing an object .......................................................... A–17
  Running a procedure with a TABLE parameter ....................... A–18
  Processing the data from a TABLE parameter ....................... A–18
  Running a procedure with a TABLE-HANDLE parameter ........... A–18
  Preparing schema and data for input TABLE-HANDLE parameter . A–19
  Processing schema and data for output TABLE-HANDLE parameter A–20
  Extra processing for RPC/Encoded TABLE-HANDLE parameters ... A–20
  Handling errors on the client ............................................. A–21
Creating .NET DataSets from ProDataSet parameters .................. A–22
Learning more about writing .NET clients ................................ A–27

B. Developing a Java Client to Consume OpenEdge Web Services .......... B–1
Using Java client toolkits ....................................................... B–2
Java sample Web service specifications ..................................... B–3
Creating Java Axis client interface objects ............................ B–4
  Generating client interface objects .................................... B–4
  Proxy directory structure ................................................... B–5
  Proxy classes ................................................................. B–5
  Java data type mapping ..................................................... B–6
  Output parameters .......................................................... B–6
  ABL unknown values ......................................................... B–7
  TABLE (static temp-table) parameters .................................. B–7
  TABLE-HANDLE (dynamic temp-table) parameters ................... B–8
  Sample Java prototypes for common object methods ............... B–9
  Sample ABL and Java interface method prototypes ............... B–9
Developing the Axis client application ................................... B–12
  Setting up the Web service objects ..................................... B–12
  Using the sample PscObjectIDHandler and HandlerControlBlock
    classes ............................................................................. B–13
  Setting up the PscObjectIDHandler objects ......................... B–14
  Connecting to a Web service ................................................ B–15
<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1–1: OpenEdge Web services architecture</td>
<td>1–5</td>
</tr>
<tr>
<td>Figure 1–2: ABL clients consuming Web services</td>
<td>1–11</td>
</tr>
<tr>
<td>Figure 1–3: OpenEdge Web services sample applications</td>
<td>1–17</td>
</tr>
<tr>
<td>Figure 5–1: WSViewer main window</td>
<td>5–11</td>
</tr>
<tr>
<td>Figure 6–1: WSDL Analyzer index page for a single service only</td>
<td>6–7</td>
</tr>
<tr>
<td>Figure 6–2: WSDL Analyzer index page for multiple services</td>
<td>6–8</td>
</tr>
<tr>
<td>Figure 6–3: WSDL Analyzer service page for one of multiple services</td>
<td>6–8</td>
</tr>
<tr>
<td>Figure 6–4: WSDL Analyzer operation index page</td>
<td>6–9</td>
</tr>
<tr>
<td>Figure 6–5: WSDL Analyzer port type page (up to the Summary)</td>
<td>6–9</td>
</tr>
<tr>
<td>Figure 6–6: WSDL Analyzer port type page (connection details)</td>
<td>6–10</td>
</tr>
<tr>
<td>Figure 6–7: WSDL Analyzer port type page (operation detail)</td>
<td>6–11</td>
</tr>
<tr>
<td>Figure 6–8: WSDL Analyzer data types page (summary list)</td>
<td>6–12</td>
</tr>
<tr>
<td>Figure 6–9: WSDL Analyzer port type page (TABLE in signature)</td>
<td>6–15</td>
</tr>
<tr>
<td>Figure 8–1: WSDL Analyzer sample operation call</td>
<td>8–4</td>
</tr>
<tr>
<td>Figure 9–1: Referencing a header entry in the SOAP header object model</td>
<td>9–4</td>
</tr>
<tr>
<td>Figure 9–2: Accessing entry elements in the SOAP header object model</td>
<td>9–5</td>
</tr>
<tr>
<td>Figure 10–1: SOAPSpy window opened by ProSOAPView</td>
<td>10–14</td>
</tr>
</tbody>
</table>
Table D–8: ABL data type cast—INTEGER ........................................ D–13
Table D–9: ABL data type cast—LOGICAL ........................................ D–15
Table D–10: ABL data type cast—RAW or MEMPTR .......................... D–16
Table E–1: XML data types for TABLE parameter columns ............... E–9
Table E–2: XML data types for TABLE-HANDLE parameter columns .... E–10
Procedures

WSDL Containing an object ID definition ........................................ 3–13
SOAP header containing an object ID ............................................ 3–14
TABLE parameters—general Document (or RPC)/Literal format .......... 3–18
TABLE parameters—general RPC/Encoded format .............................. 3–18
TABLE-HANDLE definition for all dynamic temp-table parameters ....... 3–20
TABLE-HANDLE parameters—general format ................................ 3–20
DATASET parameters—general SOAP format ................................... 3–21
Common DATASET-HANDLE definition for all SOAP formats ............... 3–22
ABL procedure prototype ............................................................... 3–23
Interface method prototype generated from a WSDL operation definition 3–26
Interface method call generating SOAP messages .............................. 3–26
SOAP request message for method input parameters ......................... 3–27
SOAP response message for method output parameters ...................... 3–27
SOAP faults—general format .......................................................... 3–29
VB.NET method declarations on AppObject, OrderInfo ....................... 4–2
VB.NET method declarations on ProcObject, CustomerOrder ................. 4–3
VB.NET prototype for an AppObject Connect_Object method ................. 4–4
VB.NET client code for the connect method on AppObject, OrderInfo .... 4–5
SOAP connect request for AppObject, OrderInfo ................................ 4–5
SOAP response to OrderInfo AppObject connect request .................... 4–5
VB.NET client code to instantiate session-free AppObject, OrderInfo ...... 4–6
ABL prototype for a non-persistent external procedure ....................... 4–7
VB.NET prototype for a method that runs an ABL non-persistent procedure 4–7
VB.NET client code calling an external procedure method .................... 4–7
SOAP request from calling the FindCustomerByNum procedure method .... 4–7
SOAP response from calling the FindCustomerByNum method ................ 4–8
ABL prototype for a persistent procedure to implement a ProcObject .... 4–10
VB.NET prototype for the ProcObject CreatePO_CustomerOrder method .... 4–10
VB.NET client code to create the ProcObject, CustomerOrder ............... 4–10
SOAP request to create ProcObject, CustomerOrder .......................... 4–11
SOAP response from creating ProcObject, CustomerOrder ................... 4–11
ABL prototype for a user-defined function ...................................... 4–12
VB.NET prototype for an ABL user-defined function method ............... 4–12
VB.NET client code of method to run an ABL user-defined function ...... 4–12
SOAP request for user-defined function method, GetTotalOrdersByNumber 4–12
SOAP response from calling the GetTotalOrdersByNumber method ........ 4–13
VB.NET prototype for a Release_Object method ................................ 4–14
VB.NET client code calling methods to release objects ....................... 4–14
ABL prototype that passes a TABLE parameter ................................ 4–15
VB.NET declaration of interface method passing static TEMP-TABLE, ttEmp 4–15
VB.NET client code passing TABLE, ttEmp, to an interface method .......... 4–15
Doc/Lit SOAP request to pass a TABLE parameter ............................ 4–16
RPC/Encoded SOAP request to pass a TABLE parameter ...................... 4–16
ABL prototype that passes a TABLE-HANDLE parameter ..................... 4–17
VB.NET declaration of interface method passing TABLE-HANDLE, ttHandle 4–17
VB.NET client code passing TABLE-HANDLE, ttHandle, to interface method 4–18
Doc/Lit SOAP request for TABLE-HANDLE, ttHandle ........................ 4–18
TABLE-HANDLE <schema> element for an Doc/Lit SOAP request ....... 4–19
TABLE-HANDLE <Data> element for an Doc/Lit SOAP request ............. 4–19
Sample SOAP fault for disabled Web service .................................. 4–20
getCustOrders.p ........................................................................ 4–21
Sample C#.NET proxy code for getCustOrders .................................. 4–22
Sample C#.NET application for getCustOrders operation .................... 4–23
Sample C#.NET proxy code for getDynDs ...................................... 4–24
Preface

This Preface contains the following sections:

- Purpose
- Audience
- Organization
- Using this manual
- Typographical conventions
- Examples of syntax descriptions
- Example procedures
- OpenEdge messages
- Third party acknowledgements
Purpose

This manual describes how OpenEdge supports Web services. It describes the technology for exposing AppServer™ application services as Web services (OpenEdge Web services), and how to access industry Web services from ABL (Advanced Business Language) as a Web service client. It also defines the Web services industry standards that OpenEdge supports and offers basic information for programming industry Web service clients to access OpenEdge Web services.

This manual provides a brief overview of Web services architecture in OpenEdge. However, for a complete description of the Web services architecture in OpenEdge, including all components and how they work together, see OpenEdge Getting Started: Application and Integration Services.

This manual describes the general procedures and facilities for building OpenEdge Web services, as well as specific client requirements. For additional information on Web service development, deployment, and administration, see the following documentation:

- **Web service development:**
  - OpenEdge Application Server: Developing AppServer Applications
  - OpenEdge Development: Open Client Introduction and Programming

- **Web service deployment and administration:**
  - OpenEdge Application Server: Administration
  - Progress Explorer online help for the Web Services Adapter
  - OpenEdge Development: Messaging and ESB (for deploying and managing OpenEdge Web services for the Sonic ESB using the Sonic ESB Adapter)

Audience

This manual is intended for experienced ABL developers who want to either expose their AppServer-enabled applications as a Web services or consume Web services in their ABL applications.

Organization

Part I, Introduction

Chapter 1, “Web Services in OpenEdge”

Provides a general overview of Web services. It then briefly describes Web services in OpenEdge: how to develop, deploy, and manage OpenEdge Web services, and how you can access industry Web services from an ABL application. Finally, it provides information on the Web service sample applications available on the Documentation and Samples (doc_samples) directory of the OpenEdge product DVD.
Part II, Creating OpenEdge Web Services

Chapter 2, “Exposing AppServer-enabled Applications as OpenEdge Web Services”

Describes the initial decisions that you need to make when you transform an AppServer-enabled application into an OpenEdge Web service. This includes discussions of SOAP formats and session models.

Chapter 3, “Building Clients for OpenEdge Web Services”

Discusses how to build a client that consumes your OpenEdge Web service. This includes discussions of how SOAP formats, session models, and the structure of your ABL application impact the programming of the client.

Chapter 4, “Sample Code with SOAP Messages for OpenEdge Web Services”

Details the techniques for consuming OpenEdge Web service operations in a non-ABL client, showing the types of SOAP messages that are generated for various types of SOAP requests.

Chapter 5, “Testing and Debugging OpenEdge Web Services”

Describes how to test and debug OpenEdge Web services using log files, SOAP viewers, and basic SOAP fault handling techniques.

Part III, Creating ABL Clients to Consume Web Services

Chapter 6, “Creating an ABL Client from WSDL”

Describes how to access a Web service from an ABL application. This includes using the WSDL Analyzer to generate sample ABL code for connecting to the Web service and calling its operations. The chapter also describes how OpenEdge maps ABL data types to the XML Schema data types interpreted by Web services.

Chapter 7, “Connecting to Web Services from ABL”

Describes how to connect to and manage connections to a Web service from ABL.

Chapter 8, “Invoking Web Service Operations from ABL”

Describes how to invoke and manage Web service operations from ABL. This includes information on managing both simple and complex data in parameters and managing asynchronous requests.

Chapter 9, “Handling SOAP Message Headers in ABL”

Describes how to manage SOAP message headers in ABL for Web service operations that are configured for SOAP header management. This includes a detailed description of ABL techniques for accessing, manipulating, creating, and managing all elements of a SOAP message header.

Chapter 10, “Handling Errors in ABL Requests to Web Services”

Describes how to manage errors from ABL requests to Web services in general and for SOAP faults in particular. This includes a detailed description of ABL techniques for accessing all elements of a SOAP fault message.
Part IV, Appendices

Appendix A, “Developing a .NET Client to Consume OpenEdge Web Services”

Describes a sample scenario in which a .NET client application is developed to access a OpenEdge Web service.

Appendix B, “Developing a Java Client to Consume OpenEdge Web Services”

Describes a sample scenario in which a Java™ client application is developed to access a OpenEdge Web service.

Appendix C, “ABL Elements for Consuming Web Services”

Provides a quick reference to the basic ABL elements provided by OpenEdge to access Web services.

Appendix D, “Data Type Conversion Rules for ABL Calls to Web Services”

Describes in detail the key rules used to transform data for Web service operation parameters between alternative mappings of ABL data types and XML Schema data types.

Appendix E, “Understanding WSDL Details”

Provides a closer look at how certain ABL features are defined in the WSDL file that ProxyGen creates.

Appendix F, “Commands and Utilities”

Describes the syntax for commands and utilities documented in this manual. If this manual provides the primary documentation for a command or utility, the syntax for that command or utility appears in this appendix.

Using this manual

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is ABL (Advanced Business Language). With few exceptions, all keywords of the language appear in all UPPERCASE, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation updates see the OpenEdge Product Documentation Overview page on PSDN: http://communities.progress.com/pcom/docs/DOC-16074.

References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the ABL Virtual Machine (AVM). When the documentation refers to ABL source code compilation, it specifies ABL or the compiler as the actor that manages compile-time features of the language. When the documentation refers to run-time behavior in an executing ABL program, it specifies the AVM as the actor that manages the specified run-time behavior in the program.
For example, these sentences refer to the ABL compiler’s allowance for parameter passing and the AVM’s possible response to that parameter passing at run time: “ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error.” The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: “The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets.”

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all UPPERCASE, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever integer appears, this is a reference to the INTEGER or INT64 data type.
- Wherever character appears, this is a reference to the CHARACTER, LONGCHAR, or CLOB data type.
- Wherever decimal appears, this is a reference to the DECIMAL data type.
- Wherever numeric appears, this is a reference to the INTEGER, INT64, or DECIMAL data type.

References to built-in class data types appear in mixed case with initial caps, for example, Progress.Lang.Object. References to user-defined class data types appear in mixed case, as specified for a given application example.

Typographical conventions

This manual uses the following typographical conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Italic typeface indicates the title of a document, or signifies new terms.</td>
</tr>
<tr>
<td>SMALL, BOLD CAPITAL LETTERS</td>
<td>Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, <strong>GET</strong> and <strong>CTRL</strong>.</td>
</tr>
<tr>
<td>KEY1+KEY2</td>
<td>A plus sign between key names indicates a simultaneous key sequence: you press and hold down the first key while pressing the second key. For example, <strong>CTRL+X</strong>.</td>
</tr>
<tr>
<td>KEY1 KEY2</td>
<td>A space between key names indicates a sequential key sequence: you press and release the first key, then press another key. For example, <strong>ESCAPE H</strong>.</td>
</tr>
<tr>
<td>Convention</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Syntax:</strong></td>
<td></td>
</tr>
<tr>
<td>Fixed width</td>
<td>A fixed-width font is used in syntax statements, code examples,</td>
</tr>
<tr>
<td></td>
<td>system output, and filenames.</td>
</tr>
<tr>
<td>Fixed-width italics</td>
<td>Fixed-width italics indicate variables in syntax statements.</td>
</tr>
<tr>
<td>Fixed-width bold</td>
<td>Fixed-width bold indicates variables with special emphasis.</td>
</tr>
<tr>
<td>UPPERCASE fixed width</td>
<td>Uppercase words are ABL keywords. Although these are always shown in</td>
</tr>
<tr>
<td></td>
<td>uppercase, you can type them in either uppercase or lowercase in a procedure.</td>
</tr>
<tr>
<td><img src="three_arrows" alt="icon" /></td>
<td>This icon (three arrows) introduces a multi-step procedure.</td>
</tr>
<tr>
<td><img src="one_arrow" alt="icon" /></td>
<td>This icon (one arrow) introduces a single-step procedure.</td>
</tr>
<tr>
<td>Period (.) or colon (:)</td>
<td>All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a</td>
</tr>
<tr>
<td></td>
<td>period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with</td>
</tr>
<tr>
<td></td>
<td>either a period or a colon.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Large brackets indicate the items within them are optional.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Small brackets are part of ABL.</td>
</tr>
<tr>
<td>{ }</td>
<td>Large braces indicate the items within them are required. They are used to</td>
</tr>
<tr>
<td></td>
<td>simplify complex syntax diagrams.</td>
</tr>
<tr>
<td>{ }</td>
<td>Small braces are part of ABL. For example, a called external procedure</td>
</tr>
<tr>
<td></td>
<td>must use braces when referencing arguments passed by a calling procedure.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Ellipses indicate repetition: you can choose one or more of the preceding</td>
</tr>
<tr>
<td></td>
<td>items.</td>
</tr>
</tbody>
</table>
Examples of syntax descriptions

In this example, ACCUM is a keyword, and aggregate and expression are variables:

**Syntax**

```
ACCUM aggregate expression
```

FOR is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer:
  DISPLAY Name.
END.
```

In this example, STREAM stream, UNLESS-HIDDEN, and NO-ERROR are optional:

**Syntax**

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

**Syntax**

```
INITIAL [ constant [, constant ] ]
```

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

**Syntax**

```
{ &argument-name }
```

In this example, EACH, FIRST, and LAST are optional, but you can choose only one of them:

**Syntax**

```
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

**Syntax**

```
MAXIMUM ( expression , expression [, expression ] ... )
```
In this example, you must specify MESSAGE and at least one expression or SKIP \[ (n) \], and any number of additional expression or SKIP \[ (n) \] is allowed:

**Syntax**

```
MESSAGE \{ expression \| SKIP \[ (n) \] \} ...
```

In this example, you must specify \{ include-file, then optionally any number of argument or \&argument-name = "argument-value", and then terminate with \}:

**Syntax**

```
\{ include-file
    \[ argument \| \&argument-name = "argument-value" \] ...
\}
```

**Long syntax descriptions split across lines**

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, WITH is followed by six optional items:

**Syntax**

```
WITH \[ ACCUM max-length \] \[ expression DOWN \]
    \[ CENTERED \] \[ n COLUMNS \] \[ SIDE-LABELS \]
    \[ STREAM-IO \]
```

**Complex syntax descriptions with both required and optional elements**

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, ASSIGN requires either one or more field entries or one record. Options available with field or record are grouped with braces and brackets:

**Syntax**

```
ASSIGN \{ \[ FRAME frame \] \{ field \[ = expression \] \}
    \[ WHEN expression \] \} ...
    \| \{ record \[ EXCEPT field ... \] \}
```
Example procedures

This manual provides numerous example procedures that illustrate syntax and concepts. You can access the example files and details for installing the examples from the following locations:

- The Documentation and Samples located in the doc_samples directory on the OpenEdge Product DVD.
- The OpenEdge Product Documentation Overview page on PSDN:
  
  http://communities.progress.com/pcom/docs/DOC-16074

After you install the examples, you can find the Web services samples in src/samples/webservices.

OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.

- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.

- **Startup messages** inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.

- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.

- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.

- Terminates the current session.
OpenEdge messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.

**Obtaining more information about OpenEdge messages**

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose **Help → Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose **Help → Messages** and then type the message number to display a description of a specific OpenEdge message.
- In the Procedure Editor, press the **HELP** key or **F1**.

On UNIX platforms, use the OpenEdge pro command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

```
To use the pro command to obtain a message description by message number:
```

1. Start the Procedure Editor:

```
OpenEdge_install_dir/bin/pro
```

2. Press **F3** to access the menu bar, then choose **Help → Messages**.

3. Type the message number and press **ENTER**. Details about that message number appear.

4. Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose **File → Exit**.
Third party acknowledgements

OpenEdge includes AdventNet - Agent Toolkit licensed from AdventNet, Inc. [http://www.adventnet.com](http://www.adventnet.com). All rights to such copyright material rest with AdventNet.

OpenEdge includes ANTLR (Another Tool for Language Recognition) software Copyright © 2003-2006, Terence Parr. All rights reserved. Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes software developed by the Apache Software Foundation [http://www.apache.org/](http://www.apache.org/). Copyright © 1999 The Apache Software Foundation. All rights reserved (Xerces C++ Parser (XML) and Xerces2 Java Parser (XML)); Copyright © 1999-2002 The Apache Software Foundation. All rights reserved (Xerces Parser (XML)); and Copyright © 2000-2003 The Apache Software Foundation. All rights reserved (Ant). The names “Apache,” “Xerces,” “ANT,” and “Apache Software Foundation” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission of the Apache Software Foundation. For written permission, please contact apache@apache.org. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes Concurrent Java software Copyright 1994-2000 Sun Microsystems, Inc. All Rights Reserved. Neither the name of or trademarks of Sun may be used to endorse or promote products including or derived from the Java Software technology without specific prior written permission; and Redistributions of source or binary code must contain the above copyright notice, this notice and the following disclaimers: This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN MICROSYSTEMS, INC. AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN MICROSYSTEMS, INC. OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN MICROSYSTEMS, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

OpenEdge includes DataDirect software Copyright © 1991-2007 Progress Software Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect for JDBC Type 4 driver); Copyright © 1993-2009 Progress Software Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect for JDBC); Copyright © 1988-2007 Progress Software Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect for ODBC); and Copyright © 1988-2007 Progress Software
Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect64 for ODBC).

OpenEdge includes DataDirect Connect for ODBC and DataDirect Connect64 for ODBC software, which include ICU software 1.8 and later - Copyright © 1995-2003 International Business Machines Corporation and others All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in all copies of the Software.

OpenEdge includes DataDirect Connect for ODBC and DataDirect Connect64 for ODBC software, which include software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). Copyright © 1998-2006 The OpenSSL Project. All rights reserved. And Copyright © 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved.

OpenEdge includes DataDirect products for the Microsoft SQL Server database which contain a licensed implementation of the Microsoft TDS Protocol.

OpenEdge includes software authored by David M. Gay. Copyright © 1991, 2000, 2001 by Lucent Technologies (dtoa.c); Copyright © 1991, 1996 by Lucent Technologies (g_fmt.c); and Copyright © 1991 by Lucent Technologies (rnd_prod.s). Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OpenEdge includes software authored by David M. Gay. Copyright © 1998-2001 by Lucent Technologies All Rights Reserved (decsrttod.c; strtodg.c); Copyright © 1998, 2000 by Lucent Technologies All Rights Reserved (decstrtof.c; strtord.c); Copyright © 1998 by Lucent Technologies All Rights Reserved (dmisc.c; gdtoa.h; gethex.c; gmisc.c; sum.c); Copyright © 1998, 1999 by Lucent Technologies All Rights Reserved (gdtoa.c; misc.c; smisc.c; ulp.c); Copyright © 1998-2000 by Lucent Technologies All Rights Reserved (gdtoaimp.h); Copyright © 2000 by Lucent Technologies All Rights Reserved (hd_init.c). Full copies of these licenses can be found in the installation directory, in the c:/OpenEdge/licenses folder. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both the copyright notice and this permission notice appear in all supporting documentation, and that the name of Lucent or any of its entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
OpenEdge includes http package software developed by the World Wide Web Consortium. Copyright © 1994-2002 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All rights reserved. This work is distributed under the W3C® Software License [http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

OpenEdge includes ICU software 1.8 and later - Copyright © 1995-2003 International Business Machines Corporation and others All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.


OpenEdge includes Infragistics NetAdvantage for .NET v2009 Vol 2 Copyright © 1996-2009 Infragistics, Inc. All rights reserved.

OpenEdge includes JSTL software Copyright 1994-2006 Sun Microsystems, Inc. All Rights Reserved. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes OpenSSL software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). Copyright © 1998-2007 The OpenSSL Project. All rights reserved. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com). Copyright © 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes Quartz Enterprise Job Scheduler software Copyright © 2001-2003 James House. All rights reserved. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product. This product uses and includes within its distribution, software developed by the Apache Software Foundation (http://www.apache.org/).

OpenEdge includes code licensed from RSA Security, Inc. Some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/.

OpenEdge includes the RSA Data Security, Inc. MD5 Message-Digest Algorithm. Copyright ©1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.
OpenEdge includes Sonic software, which includes software developed by Apache Software Foundation (http://www.apache.org/). Copyright © 1999-2000 The Apache Software Foundation. All rights reserved. The names “Ant”, “Axis”, “Xalan,” “FOP,” “The Jakarta Project”, “Tomcat”, “Xerces” and/or “Apache Software Foundation” must not be used to endorse or promote products derived from the Product without prior written permission. Any product derived from the Product may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission. For written permission, please contact apache@apache.org.

OpenEdge includes Sonic software, which includes software developed by ExoLab Project (http://www.exolab.org/). Copyright © 2000 Intalio Inc. All rights reserved. The names “Castor” and/or “ExoLab” must not be used to endorse or promote products derived from the Products without prior written permission. For written permission, please contact info@exolab.org. Exolab, Castor and Intalio are trademarks of Intalio Inc.

OpenEdge includes Sonic software, which includes software developed by IBM. Copyright © 1995-2003 International Business Machines Corporation and others. All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

OpenEdge includes Sonic software, which includes the JMX Technology from Sun Microsystems, Inc. Use and Distribution is subject to the Sun Community Source License available at http://sun.com/software/communitysource.

OpenEdge includes Sonic software, which includes code licensed from Mort Bay Consulting Pty. Ltd. The Jetty Package is Copyright © 1998 Mort Bay Consulting Pty. Ltd. (Australia) and others.
OpenEdge includes Sonic software, which includes files that are subject to the Netscape Public License Version 1.1 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/NPL/. Software distributed under the License is distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is Mozilla Communicator client code, released March 31, 1998. The Initial Developer of the Original Code is Netscape Communications Corporation. Portions created by Netscape are Copyright 1998-1999 Netscape Communications Corporation. All Rights Reserved.

OpenEdge includes Sonic software, which includes software developed by the University Corporation for Advanced Internet Development http://www.ucaid.edu Internet2 Project. Copyright © 2002 University Corporation for Advanced Internet Development, Inc. All rights reserved. Neither the name of OpenSAML nor the names of its contributors, nor Internet2, nor the University Corporation for Advanced Internet Development, Inc., nor UCAID may be used to endorse or promote products derived from this software and products derived from this software may not be called OpenSAML, Internet2, UCAID, or the University Corporation for Advanced Internet Development, nor may OpenSAML appear in their name without prior written permission of the University Corporation for Advanced Internet Development. For written permission, please contact opensaml@opensaml.org.

OpenEdge includes the UnixWare platform of Perl Runtime authored by Kiem-Phong Vo and David Korn. Copyright © 1991, 1996 by AT&T Labs. Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED “AS IS”, WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHORS NOR AT&T LABS MAKE ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OpenEdge includes Vermont Views Terminal Handling Package software developed by Vermont Creative Software. Copyright © 1988-1991 by Vermont Creative Software.

OpenEdge includes XML Tools, which includes versions 8.9 of the Saxon XSLT and XQuery Processor from Saxonica Limited (http://www.saxonica.com/) which are available from SourceForge (http://sourceforge.net/projects/saxon/). The Original Code of Saxon comprises all those components which are not explicitly attributed to other parties. The Initial Developer of the Original Code is Michael Kay. Until February 2001 Michael Kay was an employee of International Computers Limited (now part of Fujitsu Limited), and original code developed during that time was released under this license by permission from International Computers Limited. From February 2001 until February 2004 Michael Kay was an employee of Software AG, and code developed during that time was released under this license by permission from Software AG, acting as a "Contributor". Subsequent code has been developed by Saxonica Limited, of which Michael Kay is a Director, again acting as a "Contributor". A small number of modules, or enhancements to modules, have been developed by other individuals (either written especially for Saxon, or incorporated into Saxon having initially been released as part of another open source product). Such contributions are acknowledged individually in comments attached to the relevant code modules. All Rights Reserved. The contents of the Saxon files are subject to the Mozilla Public License Version 1.0 (the "License"); you may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/ and a copy of the license can also be found in the
installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

OpenEdge includes XML Tools, which includes Xs3P v1.1.3. The contents of this file are subject to the DSTC Public License (DPL) Version 1.1 (the "License"); you may not use this file except in compliance with the License. A copy of the license can be found in the installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is xs3p. The Initial Developer of the Original Code is DSTC. Portions created by DSTC are Copyright © 2001, 2002 DSTC Pty Ltd. All rights reserved.

OpenEdge includes YAJL software Copyright 2007, Lloyd Hilaiel. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. Neither the name of Lloyd Hilaiel nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Part I

Introduction

Chapter 1, Web Services in OpenEdge
In OpenEdge®, you can:

- Produce Web services that you build from ABL (Advanced Business Language) application services running on the AppServer™. These OpenEdge Web services are consumable from any application that can access industry standard Web services as a client. For more information, see the chapters in Part II, “Creating OpenEdge Web Services.”

- Consume industry standard Web services and invoke Web service operations from any ABL application, and otherwise manage interactions with Web services from ABL in a manner consistent with industry Web service clients. For more information, see the chapters in Part III, “Creating ABL Clients to Consume Web Services.”

This chapter describes the general support for Web services in OpenEdge and contains the following sections:

- Web service basics
- Creating OpenEdge Web services
- Creating ABL clients to consume Web services
- Sample Web service applications
Web service basics

A Web service is an application that can be consumed (accessed and used) over the Internet (or an intranet) using industry-standard protocols. To tell users how to consume this application, the Web service provider creates a file that describes the application program interface (API) in the Web Services Description Language (WSDL). Using the WSDL file, a user can create a client in any appropriate language to consume the Web service.

The client invokes the Web service operations by sending messages in the SOAP protocol over HTTP/S. The Web service application executes the operations and returns the results to the client in the same way.

The following sections discuss the standards involved in Web services. For more information on the general architecture of Web services, see the chapter on Web services in OpenEdge Getting Started: Application and Integration Services. For further details of the ProxyGen utility, see OpenEdge Development: Open Client Introduction and Programming.

The mass of details about SOAP messages, WSDL files, and how everything fits together to create a working Web service can seem imposing at first. But, as client toolkits have improved, deploying Web services and developing clients to consume them has become more automated. Generally, you do not need to know the details of the WSDL and SOAP message structures. The client toolkits can usually extract the necessary information from the WSDL document and automatically generate the objects a client application needs to call the Web service.

The automation available through the client toolkits can insulate the user from many of the technical details of a Web service. While this manual provides some of those details, you might not need them. You might find that a client toolkit generates all the code that you need to make a Web service work for you.

Industry standards supported by OpenEdge

The key words in defining Web services are industry standards. Web services represent a widely accepted set of industry standards that allow distributed applications to communicate and exchange data without concern for each other’s specific platform requirements.

OpenEdge support of Web services is based on the following industry standards:

- **WSDL** — WSDL 1.1 is the de facto standard according to the W3C. For more information on WSDL, visit the following Web address:

  [http://www.w3.org/TR/wsd1](http://www.w3.org/TR/wsd1)

- **SOAP** — SOAP 1.1 HTTP Binding is the de facto standard according to the W3C. For more information on SOAP, visit the following Web address:

  [http://www.w3.org/TR/SOAP](http://www.w3.org/TR/SOAP)

- **XML Schema** — W3C 2001 XML Schema Recommendation. For an introduction to XML Schema, visit the following Web address:

  [http://www.w3.org/TR/xmlschema-0/](http://www.w3.org/TR/xmlschema-0/)
WSDL files

Most Web service providers publicize and describe the public interface to their Web services using WSDL. The WSDL file describes the Web service interface, including how to call Web service operations and the binding requirements to access the Web service. OpenEdge uses this standard, whether you are producing an OpenEdge Web service or consuming a Web service with an ABL client.

Accessing many Web services requires little, if any, knowledge of WSDL. For complex Web services, such as those requiring the use of SOAP headers, you should have a basic understanding of WSDL to make full use of the Web service and to access its data appropriately.

A WSDL file describes Web services in sections that define features of the Web service interfaces, from data and operations to the binding information needed to access the Web services on the network. Each section has a corresponding WSDL element:

- **Definitions** — Defines the Web service.
- **Types** — Defines the XML Schema for each Web service object and SOAP fault.
- **Messages** — Defines the request and response messages.
- **Port types** — Defines the signatures for all the Web service’s operations.
- **Bindings** — Defines the SOAP bindings for all the Web service objects. The bindings describe how operations in a port type are mapped to the SOAP protocol.
- **Services** — Defines the deployed locations of all the Web service objects.

**Note:** Not all Web services use these elements in exactly the same manner. For more information on WSDL files for OpenEdge Web services, see the “Web service objects and the WSDL file” section on page 2–2.

SOAP messages

SOAP is the protocol used to format message exchanges between Web service clients and Web services. The industry currently considers many scenarios for Web services communications (described at [http://www.w3.org/TR/ws-arch-scenarios/](http://www.w3.org/TR/ws-arch-scenarios/)). OpenEdge supports the request-response scenario.

In the request-response scenario, a client (Web service client) sends a SOAP request message to invoke a Web service operation on the Web service provider and the provider sends a response back to the client in the form of a SOAP response message (for a successful invocation) or in the form of a SOAP fault message (for an invocation that results in an error).

Given the automated resources available for producing and consuming Web services, you might not need to understand much about SOAP messages. The following sections give a brief overview of SOAP messages.
SOAP message formats

SOAP messages can be formatted using different SOAP message styles and encoding combinations. The style indicates how the SOAP message is structured. The encoding describes how data values are represented. Table 1–1 lists the SOAP message formats that OpenEdge supports.

Table 1–1: SOAP message formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document/Literal (Doc/Lit)</td>
<td>This is the recommended format for OpenEdge Web services. It works well for ABL and .NET clients. This includes the wrapped document literal (Wrapped Doc/Lit) convention (developed by Microsoft). Wrapped Doc/Lit is a convention using Doc/Lit that wraps all request parameters for an operation into one input XML complex type and wraps all response parameters into a separate output XML complex type.</td>
</tr>
<tr>
<td>RPC/Literal</td>
<td>Use this format for clients that do not support Doc/Lit.</td>
</tr>
<tr>
<td>RPC/Encoded</td>
<td>An earlier standard. The Web Services Interoperability Organization recommends against this format in their Basic Profile Version 1.0.</td>
</tr>
</tbody>
</table>

SOAP message document structure

SOAP messages have a common XML document structure that consists of the following ordered elements:

- The envelope is a mandatory element containing the optional header element and mandatory body element. It defines all the namespaces used in the document. Namespaces are identifiers that provide a way to define XML elements without name conflicts.
- The header is an optional element containing application-specific information, such as context and standard attributes that describe the operation.
- The body is a mandatory element containing data (for a request or response) or error information (for a fault).

XML Schema

Web services generally use XML Schema to describe both simple and complex data for their applications. OpenEdge Web services use the XML Schema standard to describe both simple data types (scalar parameters) and complex data types (array, temp-table, and ProDataSet parameters) in the WSDL file. An ABL Web service client maps XML Schema constructs into both simple and complex ABL data types.
Creating OpenEdge Web services

OpenEdge Web services are one of the interface types produced by the OpenEdge Open Client Toolkit. The Open Client Toolkit allows you to generate an interface that encapsulates remote ABL procedures and functions supported on an AppServer. A client application, either ABL or non-ABL, can then access these AppServer procedures and functions through the methods of the generated Open Client interface.

To produce an OpenEdge Web service with the Open Client Toolkit, you create a definition of your Web service in ProxyGen, which then generates a client interface definition, the Web Service Mapping file (WSM). Using the WSM file, you can deploy your Web service on an OpenEdge Web Services Adapter (WSA).

Figure 1–1 shows the basic Web services architecture, highlighting the specific OpenEdge elements used to produce a Web service.

Figure 1–1: OpenEdge Web services architecture

As Figure 1–1 shows, producing an OpenEdge Web service only requires deploying an interface for an AppServer-enabled ABL application on the WSA. Both your application and the expected clients affect how you define that interface.

A typical set of tasks to create an OpenEdge Web service include:

1. Defining requirements
2. Defining the Web service in ProxyGen
3. Deploying the Web service
4. Putting the Web service into production
Defining requirements

The first step in creating a Web service is considering the requirements of the application and its intended clients. Knowing these requirements can decide certain details of a Web service design and implementation. Some important requirements are the session model, the SOAP message format, and the security options that the Web service uses.

Session models

The session model of your Web service matches the session model of the AppServer-enabled application. There are two session models:

- **Session-free** — This connectionless model uses an AppServer running in state-free mode. This is the recommended model for Web services.

- **Session-managed** — This connection-oriented model uses AppServers running in stateless, state-aware, or state-reset operating modes. This model should only be used when the application must maintain a private, persistent connection with a client.

The session-free model supports a connectionless and context-free interaction between the client and Web service application. This session-free model is far more scalable than the session-managed model because the AppServer accepts client requests from all clients as fast as it can execute them, and executes them in no particular order. No AppServer resource (memory or process) is reserved for indefinite use by a single client, as is the case with the session-managed model. With this model, the AppServer application must be written to be context-free, and all mechanisms otherwise available for context management on the AppServer are not supported.

The session-managed model provides varying types of support for maintaining and managing session context across client requests, throughout the life of the client connection. With this model, the type of context management supported depends on the actual operating mode of the AppServer, and the AppServer application must be written to maintain session context accordingly. The AppServer handles all requests from a single client sequentially in this model, completing each request before accepting another, which allows an orderly maintenance of context from one request to the next. To achieve this sequential handling, your Web service and its clients must use object IDs and SOAP headers to maintain context. Handling these elements increases the complexity of programming for a session-managed model. For more information on object IDs and SOAP headers, see the “Retrieving and sending object IDs—handling SOAP headers” section on page 3–13.

For information on configuring and programming AppServers, see OpenEdge Application Server: Administration and OpenEdge Application Server: Developing AppServer Applications. For more information on choosing a session model for your OpenEdge Web service, see the “Session Models and object IDs” section on page 2–3.

SOAP message formats

As described in the “SOAP messages” section on page 1–3, OpenEdge supports the following SOAP message formats:

- Document/Literal (Doc/Lit)
- RPC/Literal
- RPC/Encoded
In general, you can use the same application code to create Web services for clients that support any of these formats. Each Web service instance can only accept a single SOAP format within each target namespace. You must generate and deploy a separate Web service for each format. When you deploy the same Web service definition for multiple SOAP formats, Progress Software Corporation recommends that you specify a unique target namespace to deploy each format on the single WSA instance.

**Note:** Alternately, you might deploy the Web services for each SOAP format with the same target namespace, but on different WSA instances. However, reusing target namespaces can cause unexpected problems.

For more information on SOAP messages, see “SOAP message formats” section on page 2–6 and “SOAP format impact on generated client interfaces” section on page 3–5.

**Web service and server security options**

OpenEdge supports a variety of security options for the WSA that can be used to control access to its various administrative and run-time functions. Generally, these security options include:

- Requiring authorization to access WSA administration functions
- Restricting WSA instance administration to a selection of authorized functions
- Requiring authorization to retrieve WSDL files
- Requiring authorization for clients to access Web services
- Enabling or disabling access to WSA administration
- Enabling or disabling client access to Web services
- Securing the data connection between a Web service and the AppServer using SSL tunneling

For more information on security options for managing WSA configuration and administration and on managing access control for Web services, see *OpenEdge Application Server: Administration*. For more information on how OpenEdge supports security for OpenEdge Web services, see *OpenEdge Getting Started: Core Business Services*.

**Defining the Web service in ProxyGen**

After you have built the AppServer application and compiled its r-code, you can use ProxyGen to prepare the application for deployment as a Web service. As you define the Web service, your knowledge of the application and client requirements helps to decide how best to organize and include the procedures of your application as Open Client objects and methods. ProxyGen saves information needed to create the Web service definition as a WSM file.

For information on how to use ProxyGen to define the Web service, see *OpenEdge Development: Open Client Introduction and Programming*.
Deploying the Web service

Use Progress Explorer or WSAMAN to deploy Web services to a particular Web Service Adapter (WSA) instance and to manage deployed Web services at run time. The WSA is a Java servlet running on a Web server or stand-alone Java Servlet Engine (JSE). Situated between the client and AppServer, this Java servlet understands how to pass communications at run time between a Web service client and the AppServer. Each deployed Web service behaves according to a predefined set of run-time properties. These properties are initially set to default values for all Web services deployed to a WSA instance. You can set and reset their values for each Web service individually, under prescribed run-time conditions.

**Caution:** You can update a deployed Web service to change its deployment information and object definitions. However, you should not update a Web service that is enabled for live access by production clients. Instead, create a new version of the Web service with the intended updates and deprecate the old version of the Web service once all clients have moved to the new version.

For more information on deploying and managing a deployed Web service, see *OpenEdge Application Server: Administration*. For information on configuring a WSA instance, see the “Configuring a Web Service Adapter instance” section on page 2–8.

Writing a client and testing the Web service

Before enabling your Web service for public access, you should test it by building test clients to access it. To begin writing a client, you need the WSDL file. When you define an OpenEdge Web service in ProxyGen, you can select an option to have ProxyGen generate a test WSDL file. You should build your initial test clients using only the information provided in the WSDL. Starting from this information enables you to identify what other documentation users who are building clients without intimate knowledge of your application might need.

Here are three common examples of how client toolkits work:

- **Microsoft® Visual Studio** — Adds a Web Reference that points to the WSDL to your .NET project. Visual Studio creates proxy code from the WSDL and stores the proxies in a references file. When you reference the Web service in your code, Visual Studio offers these proxy objects as appropriate. See Appendix A, “Developing a .NET Client to Consume OpenEdge Web Services” for detailed information on creating .NET for OpenEdge Web services.

- **Apache® Axis** — Runs the wsdl2java command on the WSDL. The command generates the proxy code in several packages in the current directory. See Appendix B, “Developing a Java Client to Consume OpenEdge Web Services” for detailed information on creating Java clients for OpenEdge Web services.

- **OpenEdge** — Uses the WSDL Analyzer. The WSDL Analyzer outputs a series of HTML pages with sample code for invoking the Web service’s operations.

**Caution:** You should test a Web service by deploying it to a WSA instance that is isolated from the intended client domain. This allows you to enable the Web service for access by clients doing preproduction testing without making it visible and “live” to production clients. Once testing is complete, you can redeploy or import the Web service to the context of the production WSA instance.
For more information on writing client applications for an OpenEdge Web service, see Chapter 3, “Building Clients for OpenEdge Web Services.” For more information on testing and debugging Web services, see Chapter 5, “Testing and Debugging OpenEdge Web Services.”

**Documenting client programming requirements**

You might find that your Web service interface can benefit from additional client programmer documentation along with the WSDL file to aid in easily building a client. This information describes any special requirements for Web service operation parameters and other requirements for Web service binding and programming. The requirements for most parameters and their data types are based on standards supported in the WSDL file, and most Web service client toolkits support them without additional programming.

However, there are some unique features of OpenEdge support for building Web services that it might be helpful or necessary for client programmers to know:

- **Open Client object model** — The Open Client object model supports certain built-in object management methods whose availability and use depends on the session model of the Web service application and the particular type of Open Client object in use. For example, in a session-managed application, the instantiation of each object returns a unique ID that identifies that object and must be maintained and supplied in all method calls on that object. In general, the client programmer needs to understand when they can and should use these built-in methods on an object and how to use them.

- **SOAP format** — All Web service client platforms support a limited choice of SOAP formats, often fewer than the three supported by OpenEdge. The client programmer needs to know the SOAP format that a particular Web service uses in order to know whether their client platform supports it. This information is certainly available in the WSDL, but it might be helpful to know before downloading the WSDL. Also, you might provide versions of the Web service for each supported SOAP format and identify them accordingly.

- **Relational data** — OpenEdge allows you to build Web services that pass complex data types as input and output parameters, thus supporting the exchange of relational data (such as ABL temp-tables and ProDataSets). For many OpenEdge Web services, the WSDL supplies all the schema information required to access these relational data, particularly for static temp-tables and ProDataSets. However, for Web services that pass dynamic temp-tables and ProDataSets, where the schema is not defined at compile-time, the client programmer needs additional documentation on the schema required to pass dynamic temp-tables and ProDataSets as input parameters.

- **SOAP faults** — The client programmer needs to understand the format of error messages that are returned from OpenEdge Web services.

For more information on all these requirements (and more) for programming client applications to access OpenEdge Web services, see Part II, “Creating OpenEdge Web Services.”
Putting the Web service into production

When you initially deploy an OpenEdge Web service, it is disabled from access by clients over the network. You can adjust Web service property values and perform other administrative tasks before making it visible and accessible to the intended network clients. When the Web service is ready for production, you can enable it for client access through Progress Explorer or the WSAMAN utility.

Tuning the Web service and its WSA instance

After your Web service is in production, you can monitor various run-time statistics accumulated for both the WSA instance and the deployed Web service. Using these, you should adjust WSA instance and Web service property values to yield the best performance metrics.

Note: Remember also to monitor and tune the AppServer configurations that support your Web services along with monitoring and tuning the Web services themselves.

For information on enabling and disabling Web services for client access and on monitoring and tuning AppServer configurations, WSA instances, the Web services that are deployed to WSA instances, see OpenEdge Application Server: Administration.
Creating ABL clients to consume Web services

An ABL client requires the target Web service’s WSDL file, both at design time and run time. OpenEdge provides a utility, the WSDL Analyzer, that reads the WSDL file and generates documentation and sample code for invoking and managing the Web service’s operations in ABL. OpenEdge uses the WSDL file at run time to interpret ABL calls to Web service operations and to manage the exchange of data between the ABL client and the Web service.

ABL views a Web service much like an application service accessed through an AppServer. In addition to the ABL elements used in any AppServer-enabled application, ABL includes some elements specifically designed to manage calls to Web services. In general, all ABL network deployment models support inherent access to Web services.

Figure 1–2 shows the basic industry standard Web services architecture and how an ABL client application communicates with a Web service.

Figure 1–2: ABL clients consuming Web services

The darker gray components are the additional pieces that the ABL client uses for consuming Web services. OpenEdge has a built-in SOAP processor to convert ABL-like procedure invocations into SOAP messages. OpenEdge supports both nonsecure HTTP and SSL using its HTTP/S processor to manage the transport of these SOAP messages across the wire.

Using this interface, you can use ABL code to interact with a Web service in a similar manner to how you interact with an AppServer-enabled application. A typical set of tasks to create an ABL client for a Web service include:

1. Obtain the Web service’s WSDL and documentation
2. Run the WSDL Analyzer
3. Build your client
4. Test your client
Obtain the Web service’s WSDL and documentation

The first step in building a client to access a Web service is to obtain the WSDL and any other documentation the Web service’s producer makes available. You need the WSDL to describe the Web service’s API and connection requirements. An ABL client requires the WSDL, both at design time and run time. The Web service’s producer might make other documentation available to describe exactly how you should use the API described in the WSDL. Make sure that the Web service uses one of the SOAP formats that OpenEdge supports.

Run the WSDL Analyzer

After obtaining the WSDL file, run the WSDL Analyzer on it. The WSDL Analyzer makes accessing Web services from ABL much easier by reading the WSDL file and producing a reference guide that documents how to use ABL to access the Web service. This guide is a series of linked HTML documents that define the Web service operations and their ABL interfaces, including how complex data types are represented in the WSDL. It also provides the binding information necessary to access the Web service on the Internet. Finally, it includes any internal WSDL file comments as documentation from the Web service provider.

For example, the WSDL Analyzer outputs the following index page when run on the WSDL for the NewCo Web service discussed in the “Consuming Web service example” section on page 1–15:
All of the information for writing the example code in the “Consuming Web service example” section on page 1–15 is found in the WSDL file. When you run the WSDL Analyzer on the WSDL file it generates interface documentation that contains information on how to connect to and invoke operations in the Web service, such as the:

- Connection parameters for the `CONNECT( )` method
- Name of a Web service port type
- Prototypes for ABL procedures and user-defined functions used to invoke Web service operations for a given port type
- Any complex types defined for use in Web service operation parameters
- Other information required to use the Web service, such as what Web service errors (SOAP faults) might be returned to the client

For more information on running the WSDL Analyzer and the documentation that it generates, see Chapter 6, “Creating an ABL Client from WSDL.”

**Build your client**

The output from the WSDL Analyzer provides the starting point to build your client. The output gives you the code to connect to the Web service and call its operations. You need to create the logic to provide data to the Web service and manipulate the data the Web service provides to complete your client.

The general procedure for accessing a Web service to invoke its operations is similar to accessing an AppServer and invoking its remote procedures and user-defined functions. For information on accessing an AppServer, see *OpenEdge Application Server: Developing AppServer Applications*.

**Comparing access to AppServers and Web services**

Table 1–2 lists and compares ABL elements used for session-free AppServer access and Web service access, listed in rough order of use.

<table>
<thead>
<tr>
<th>Session-free AppServer clients use the . . .</th>
<th>Web service clients use the . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server object handle to access the AppServer application service.</td>
<td>Server object handle to access the Web service.</td>
</tr>
<tr>
<td><code>CONNECT( )</code> method on the server handle to logically connect (bind) the server object to the application service.</td>
<td><code>CONNECT( )</code> method on the server handle to logically connect (bind) the server object to the Web service.</td>
</tr>
<tr>
<td><code>RUN</code> statement to instantiate a remote persistent procedure on an AppServer and map it to a proxy persistent procedure handle.</td>
<td><code>RUN</code> statement to access a port type in the Web service and map it to a Web service proxy procedure handle.</td>
</tr>
</tbody>
</table>

**Note:** While the syntax to access a port type is similar to the instantiation of a persistent procedure, it does not instantiate anything persistently.
Test your client

As with all applications, you need to test your Web service client. Before using your client, create some realistic test cases and make sure the client and Web service behave as expected. In addition to the normal OpenEdge tools for debugging applications, the ProSOAPView utility can be useful in testing your client. It provides a way to view SOAP messages. For more information on testing clients, see Chapter 5, “Testing and Debugging OpenEdge Web Services” and the “Examples of ABL accessing a SOAP fault” section on page 10–8.
Consuming Web service example

The Progress eLearning Community maintains a Web service for a fictional company, NewCo. The Web service enables users to look up customers, place orders, and receive invoices. The application runs off the Sports2000 database. The WSDL file for the Web service is available at the following address:

http://wstraining.progress.com/index.html

Sample of consuming a Web service

One of the operations available from NewCo Web service enables you to get the customer name for a given customer number. The Progress eLearning Community Web page includes a link to a program, InvokeValidCustomer.w, that uses that operation:

The following excerpts from InvokeValidCustomer illustrate how to consume a Web service from ABL. In the main block, the application first creates a server and connects to the Web service by specifying the location of the WSDL file for run-time access and the name of a port type (persistent procedure) on the server:

```
CREATE SERVER hServer.
-Port NewCoServiceObj").

IF 1Return = NO THEN DO:
  MESSAGE
  "Could not connect to WebService server"
  VIEW-AS ALERT-BOX INFO BUTTONS OK.
  APPLY "CLOSE":U TO THIS-PROCEDURE.
  RETURN.
END.
```

Note: You only need to specify -Port when the WSDL file contains more than one valid service and port. For information on the full range of binding options for Web services, see Chapter 7, “Connecting to Web Services from ABL.”
If the server connects successfully, the application uses the `RUN` statement syntax to create the Web service proxy procedure object and map it to the port type (`NewCoServiceObj`) that defines the `ValidateCustomer` operation for the Web service:

```
RUN NewCoServiceObj SET hPortType ON SERVER hServer.

IF NOT VALID-HANDLE(hPortType) THEN DO:
  MESSAGE
    "Could not establish portType"
  VIEW-AS ALERT-BOX INFO BUTTONS OK.
  APPLY "CLOSE":U TO THIS-PROCEDURE.
  RETURN.
END.
```

When you click **Enter**, the application invokes the `ValidateCustomer` operation, passing in the value in the customer number field and displaying the return value in the customer name field:

```
ON CHOOSE OF bDoWork IN FRAME DEFAULT-FRAME DO:
  ASSIGN iCustNum.

  /* Invoke ValidateCustomer */
  RUN ValidateCustomer IN hPortType (INPUT iCustNum, OUTPUT cCustName) NO-ERROR.

  /* this procedure checks for errors in the previous call */
  RUN ErrorInfo (OUTPUT lerr).

  IF NOT lerr THEN DO:
    DISPLAY cCustName WITH FRAME default-frame.
  END.
END.
```

For more information on mapping Web service port types and invoking operations on them, see Chapter 8, “Invoking Web Service Operations from ABL.”

When you click **Done**, the application deletes the `hPortType` procedure object, unbinds the Web service from the server, and then deletes the server object:

```
DELETE PROCEDURE hPortType.

hServer:DISCONNECT().
DELETE OBJECT hServer.
```

For more information on managing Web service bindings, see the Chapter 7, “Connecting to Web Services from ABL.”

This sample shows the similarity between basic AppServer and Web service access using synchronous interaction with the client. Another common feature between client interaction with an AppServer and client interaction with a Web service is the ability to invoke procedures asynchronously. For information on when and what you need to do to invoke Web service requests asynchronously, see the “Managing asynchronous requests” section on page 8–12.

**Note:** This sample is not available on the Documentation and Samples (`doc_samples`) directory of the OpenEdge product DVD or the Progress Documentation Web site.
Sample Web service applications

Sample Web service applications are located in a directory on the Documentation and Samples (doc_samples) directory of the OpenEdge product DVD and the Progress Documentation Web site. For more information on accessing this directory, see the “Example procedures” section on page Preface–9. This samples directory contains both sample OpenEdge Web services and sample ABL client applications accessing Web services. The directory contains separate subdirectories for different sets of samples and support files. Each sample subdirectory contains a readme.txt file with instructions on building and running the sample. You must install these samples before accessing them.

This book references some samples on the Documentation and Samples (doc_samples) directory of the OpenEdge product DVD, but also contains many code samples and examples that are not available on this DVD.

Sample application directory structure

After you install the samples, you can find three sample OpenEdge Web services and several related sample client applications written using the Microsoft .NET and Java Axis client platforms. Figure 1–3 shows the directory structure as it might appear if you install the samples into the OpenEdge installation directory (C:\Progress\dlc\ in the figure).

Figure 1–3: OpenEdge Web services sample applications
Thus, the Web service samples include the following:

- **AppService** — The three Web services, including the ProxyGen project files to generate the Web service definitions. These samples include one Web service that passes a dynamic temp-table.

- **Java-Axis** — Java Axis client samples, including applications for sample session-free and session-managed Web services, and in several choices of SOAP format. Also included with these samples are sample Java helper classes for managing object IDs and dynamic temp-table schemas and data.

- **Microsoft** — .NET client samples, including a C#.NET application for a session-free Web service and VB.NET application for session-free and session-managed Web services. These samples include the Web-services-tools-supported SOAP format choices that are also supported by .NET.

**Using the sample applications**

For information on how to access and use these samples, see the README files provided in the Java-Axis and Microsoft folders for the respective client platforms. For an introduction to the available Java helper classes, see the “Developing the Axis client application” section on page B–12.
Part II

Creating OpenEdge Web Services

Chapter 2, Exposing AppServer-enabled Applications as OpenEdge Web Services
Chapter 3, Building Clients for OpenEdge Web Services
Chapter 4, Sample Code with SOAP Messages for OpenEdge Web Services
Chapter 5, Testing and Debugging OpenEdge Web Services
Exposing AppServer-enabled Applications as OpenEdge Web Services

As discussed in the “Creating OpenEdge Web services” section on page 1–5, the general pattern for creating an OpenEdge Web service begins with the following steps:

1. Defining your requirements
2. Defining the Web service in ProxyGen
3. Configuring a Web Service Adapter instance
4. Deploying the Web service

This chapter discusses these steps in the following sections:

- Deciding how to expose your application
- Preparing the AppServer-enabled application
- Defining and deploying a Web service definition
- Distributing your WSDL file
Deciding how to expose your application

The first step in creating a Web service is considering the requirements of the application and its intended clients. Knowing these requirements can decide certain details of a Web service design and implementation. These requirements include the session model, the SOAP message format, and how you group operations into Open Client objects.

Security considerations also have a major impact on how you expose your application. For more information on security options for managing WSA configuration and administration and on managing access control for Web services, see *OpenEdge Application Server: Administration*. For more information on how OpenEdge supports security for OpenEdge Web services, see *OpenEdge Getting Started: Core Business Services*.

Web service objects and the WSDL file

During Web service deployment, the WSA generates a WSDL file that defines one or more Web service objects to describe the Web service. These Web service objects conform to the Open Client object model. There must be a single AppObject and, if appropriate, there might also be SubAppObjects and ProcObjects. The Web service objects include all operations—procedures, user-defined functions, and built-in operations for object management—that define the Web service interface. The WSDL file might also define an object identifier (*object ID*) for each Open Client object. The WSDL defines this object ID for most Open Client object types, depending on the session model.

For more information on the Open Client object model and its architecture, see *OpenEdge Development: Open Client Introduction and Programming*. For more information on how the Open Client object model supports OpenEdge Web services, see Chapter 3, “Building Clients for OpenEdge Web Services.”

Object representation

WSDL does not have an inherent concept of objects for encapsulating operations. However, OpenEdge Web services follow common usage in representing each Web service object within a unique namespace of the WSDL file. Typically, interface generators code each such object defined in the WSDL file as a corresponding object in the client interface. However, the interface generator on a particular client platform ultimately determines how each WSDL object is represented in the client development language.

Object binding

In addition to defining the Web service interface, the WSDL file specifies how this interface is bound to the application server executing the operations. For OpenEdge Web services, this binding information specifies how to generate and handle SOAP messages exchanged over HTTP/S between the client and the WSA instance managing the Web service. This includes information to identify the Web service and its WSA instance, as well as to format and send the SOAP messages. The Web service deployer specifies this binding and implementation information (*deployment information*) during Web service deployment.
Deciding how to expose your application

Deployment information

This deployment information consists of the data stored in WSDL elements as follows:

- **Web service namespace** — A namespace used to uniquely identify the Web service to the WSA instance where the Web service is deployed. In the WSDL, this value is specified by the `targetNamespace` attribute of the `<definitions>` element. It is typically specified as a Universal Resource Identifier (URI), usually in the form of a Universal Resource Locator (URL) or Universal Resource Name (URN). The value can be specified in ProxyGen, during Web service definition, or in Progress Explorer, during Web service deployment.

- **WSA URL** — The URL for the WSA instance where the Web service is deployed, and that forms the root for all URLs that access the Web services it manages. In the WSDL, this is specified by the `location` attribute of the `<soap:address>` element within the `<services>` element.

- **SOAP action** — A string (if specified) that the client application must place in the SOAPAction HTTP header when it invokes operations on the Web service. In the WSDL, this value is specified using the `soapAction` attribute of the `<soap:operation>` element, and the value is included in the SOAPAction header of all HTTP messages that transport SOAP messages for the Web service.

  The SOAPAction HTTP header is normally required for all HTTP messages that carry SOAP messages and is used by intervening security servers (such as firewalls) to determine if each HTTP message is allowed to pass through to its destination. The default is a blank string, but can be any string specified at deployment that is required by the intervening security servers on the network.

- **SOAP format** — Two string values that specify the format of SOAP messages exchanged between the client and the Web service. In the WSDL, these are specified by the `style` attribute in the `<soap:binding>` and `<soap:operation>` elements, and by the `use` attribute in the `<soap:header>`, `<soap:body>`, and `<soap:fault>` elements. While these values are specified in many places within the WSDL, the same style and use values are used everywhere in the file, ensuring that all SOAP messages generated for the Web service use the same SOAP format. For more information on SOAP formats, see the “SOAP message formats” section on page 2–6.

  For more information on specifying this deployment information in a WSDL, see *OpenEdge Development: Open Client Introduction and Programming* for the optional WSDL typically used for development testing, and see the Web service deployment chapters in *OpenEdge Application Server: Administration* for the WSDL deployed to a WSA instance. For more information on how the WSDL file specifies this information, see the “WSDL element overview for OpenEdge Web services” section on page 3–3.

Session Models and object IDs

The session model for an OpenEdge Web service must match the session model of the AppServer that hosts your application. If you are exposing an existing AppServer application as an OpenEdge Web service, this choice is already made for you. If you are building a new Web service application, choosing a session model for your Web service defines how you structure your application. The choice of session model affects how the AppObject for a Web service is defined in the WSDL, how the WSA manages the Web service at run time, and how the client must access the Web service objects.
Session models

Table 2–1 describes the available session models.

<table>
<thead>
<tr>
<th>Model</th>
<th>AppServer operating modes</th>
<th>How clients interact with Web service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session managed</td>
<td>Stateless</td>
<td>Client maintains a persistent connection to the AppServer, maintaining session context between requests for a single client application. This allows a client to carry out complex transactions that span multiple requests, but often requires extra code management, on the part of the client, for each request.</td>
</tr>
<tr>
<td></td>
<td>State-aware</td>
<td></td>
</tr>
<tr>
<td></td>
<td>State-reset</td>
<td></td>
</tr>
<tr>
<td>Session free</td>
<td>State-free</td>
<td>Client sends requests without any connection to an AppServer. This allows more clients to simultaneously access the Web service with minimal code management, but each request returns a complete result that has no certain dependency on the results of any previous requests.</td>
</tr>
</tbody>
</table>

For more information on how the AppServer supports session models, see *OpenEdge Application Server: Developing AppServer Applications*.

How object IDs support session models

OpenEdge Web services use object IDs to maintain context for clients across multiple operations of a Web service. Session-free AppObjects do not use object IDs. All other Open Client objects use object IDs. These objects require client applications to obtain the object ID from the WSA for each such object when they create it. Each request on the object must send the object’s object ID to the WSA.

For session-managed Web services, these object IDs allow each client to maintain communications with its own set of Web service objects, and thus interact with the context maintained by the AppServer for these objects. For session-free Web services, these object IDs allow the use of SubAppObjects and ProcObjects, when it is necessary to do so.

Note: SubAppObjects and ProcObjects have limited utility in the session-free model and generally should not be defined for session-free Web services. See the “How session models affect Web service objects” section on page 2–5, for more information.
How session models affect Web service objects

A session-managed AppObject establishes a connection with a single AppServer dedicated to a single client application. This AppServer services all requests for operations invoked on that AppObject until the AppObject is released. The Web service’s SubAppObjects and ProcObjects share the same connection. The connection is not freed until all the Web service objects have been released. In effect, the WSA creates copies (server instances) of these objects for each client application that accesses the same session-managed Web service, and maintains a unique object ID for each instance.

A session-free AppObject establishes no connection to an AppServer. Instead, the WSA maintains a pool of AppServer TCP/IP connections (connection pool) that it uses to send requests from all clients to any available AppServer. Requests from operations of all client applications invoked on a session-free AppObject are executed as the WSA receives them and resources are available to handle them. In effect, all clients of a session-free Web service share the same server instance of the AppObject. Requests sent to session-free SubAppObjects use any available AppServer connection in the Web service connection pool, but the WSA maintains a separate server instance of the SubAppObject for each client that accesses it using its own unique object ID until the client releases it. The WSA also maintains a separate server instance of any ProcObject accessed by a client for a session-free Web service, and each such ProcObject reserves a single AppServer connection from the connection pool dedicated for the sole use of its client until the client releases the ProcObject.

Thus, for session-managed Web service objects, the AppServer can maintain session context between requests, enabling the possibility of fine-grained, multi-request transactions on a server-side database. For session-free Web service AppObjects and SubAppObjects, there is no single AppServer to maintain session context, and any transactions on a server-side database must be coarse-grained so that they begin and end for each request before the AppServer returns the response to the client.

Note: SubAppObjects provide no additional functionality for a session-free Web service that is not already provided by the AppObject, except to organize Web service operations into developer-defined categories, much like using folders to organize files. The increased complexity of the client application required to manage the object IDs likely overwhelms any advantage of using SubAppObjects.

A ProcObject of a session-free Web service does maintain context on a single AppServer that can be used to manage finer-grained, multiple-request database transactions. But because a ProcObject reserves its AppServer connection until the object is released, that resource is not available for other requests on objects of the session-free Web service.

Note: Progress Software Corporation recommends that you avoid defining ProcObjects for session-free Web services. Each instantiated ProcObject degrades the performance of a session-free Web Service by taking AppServer connections out of the connection pool that might otherwise be used by all other requests to the Web service.
SOAP message formats

OpenEdge Web services use the request/response model for invoking the Web service and handling the results. The client sends a request message to the Web service, and the Web service returns a response message back to the same client.

Usually, client interfaces expose this request/response pair as a single object method. When the client executes the method, the underlying interface sends a SOAP request message containing any method input parameters and receives a SOAP response message containing any method output parameters and any return value.

SOAP formats supported by OpenEdge Web services

SOAP messages can be formatted using different SOAP message styles and encoding combinations. The style indicates how the SOAP message is structured. The encoding describes how data values are represented. Table 2–2 lists the SOAP message formats that OpenEdge supports. You should choose the format with which your intended clients work well.

Table 2–2: SOAP message formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document/Literal (Doc/Lit)</td>
<td>The entire message is defined as a single entity and the messages are represented literally using XML Schema standards. This is the recommended format for OpenEdge Web services. It works well for ABL and .NET clients. OpenEdge Web services use the wrapped document literal (Wrapped Doc/Lit) convention (developed by Microsoft). Wrapped Doc/Lit is a convention using Doc/Lit that wraps all request parameters for an operation into one input XML complex type and wraps all response parameters into a separate output XML complex type.</td>
</tr>
<tr>
<td>RPC/Literal</td>
<td>Each parameter is defined as a separate entity and the formatting of the messages is represented literally using XML Schema standards. Use this format for clients that do not support Doc/Literal.</td>
</tr>
<tr>
<td>RPC/Encoded</td>
<td>An earlier standard. Each parameter is defined as a separate entity and the messages to be encoded are formatted according to a set of encoding rules. This format is not recommended. The Web Services Interoperability Organization recommends against this format in their Basic Profile Version 1.0.</td>
</tr>
</tbody>
</table>

In practical terms, there is little difference between using RPC or Document style when exchanging request/response messages. The most significant difference between the supported formats is between RPC/Encoded (SOAP encoding) and Doc/Lit. Doc/Lit messages have the advantage that they can be validated by XML Schema. While SOAP encoding allows for a simpler representation of complex, object-oriented data structures than is possible with Literal encoding, this advantage usually does not offset the increased complexity needed to handle RPC/Encoded.
Preparing the AppServer-enabled application

Once you understand the requirements of your OpenEdge Web service, you can either create a new AppServer application or enable an existing AppServer application as a Web service. The decisions that you made about security, session model, and the structure of Web service objects should help you decide which is the best approach for your situation.

Realize that you do not have to include the complete interface of your AppServer-enabled application in your OpenEdge Web service. You might find that only enabling a few interface components of an existing application provides all the operations that you want in your Web service. The session model that the AppServer must support for a new application also determines the complexity of its programming.

For information on configuring and programming AppServer applications, see *OpenEdge Application Server: Administration* and *OpenEdge Application Server: Developing AppServer Applications*.

Once your application is ready, you must compile it into r-code in preparation for the next step in the process.
Defining and deploying a Web service definition

After you have built and compiled the ABL r-code for the AppServer application, you use ProxyGen to assemble a definition of your Web service. That definition enables you to define WSA instances to host your Web service and to generate a WSDL file that end users can use to build clients for your Web service.

Creating Web service definitions follows the Open Client model. For instructions on creating Web service definitions, see the chapter on generating proxies and Web service definitions in *OpenEdge Development: Open Client Introduction and Programming*. For more information on working with ProxyGen, see the online help.

Once you have a definition for your OpenEdge Web service, you need to deploy it to a Web Service Adapter (WSA) instance for testing. Chapter 3, “Building Clients for OpenEdge Web Services,” discusses how to create clients to test your Web service.

Configuring a Web Service Adapter instance

The WSA is a Java servlet running on a Web server or stand-alone Java Servlet Engine (JSE). Situated between the client and AppServer, this Java servlet understands how to communicate at run time with:

- A Web service client using SOAP messages
- An AppServer using the ABL interface to the application service

The WSA generates the WSDL file needed to design a client for your Web service. The WSA maintains the status of all OpenEdge Web services that have been deployed to its Web application context. It also manages the exchange of Web service SOAP requests and generates SOAP responses between Web service clients and AppServers at run time.

Each WSA instance can support multiple Web services. But, you can define multiple WSA server instances (Java servlets), each with different properties. For example, you might create separate WSA instances to deploy the same Web service using different SOAP message formats. For more information on WSA configuration and administration and on WSA deployment and Web service run-time support, see *OpenEdge Application Server: Administration*.

At any time in the cycle of Web service design and development, you can create and configure a WSA instance on which to deploy your Web service. When you do, think carefully about security requirements for the WSA instance and the Web services you plan to deploy to it.

Creating and configuring a WSA instance is a two-step process. You must configure the WSA servlet and the JSE that executes the WSA. You first use either OpenEdge Explorer and Progress Explorer or the mergeprop utility to define and configure the WSA instance. While OpenEdge Explorer and Progress Explorer supports most WSA configuration options, you must use the mergeprop utility to edit some WSA configuration properties in the local ubroker.properties file, particularly those that define WSA administration roles and their associated access permissions.

You can use the OpenEdge Explorer and Progress Explorer or the WSAMAN utility to temporarily change a selected set of WSA instance properties at run time (WSA Run-time Properties). This can be helpful for debugging or testing certain settings before making them permanent in your WSA configuration.
After configuring the WSA instance, you use the appropriate tools to define the WSA instance as a Java servlet to your JSE. After completing the creation and configuration of a WSA instance, you might have to restart the JSE to start up the WSA instance before deploying Web services to it.

For specific information on creating and configuring WSA instances, see OpenEdge Application Server: Administration. For information on using the mergeprop utility, see OpenEdge Getting Started: Installation and Configuration.

Installing WSA on a DMZ server

A DMZ server is a server that might not allow you to open untrusted network ports. Since the AdminServer uses ports which might be interpreted as untrusted, the standard WSA configuration might run on such a server. The WSA supports a “remote” configuration to work around this issue. The remote configuration installs only enough of OpenEdge on the remote server to run the WSA. This installation includes the WSA Java Servlet and the files used to support its local configuration.

The remote WSA uses a ubroker.properties file with a limited set of static startup parameters. You must manually edit this file to change the WSA instance's static configuration properties.

You can continue to administer the remote WSA's dynamic run-time properties and deploy and manage SOAP services by mapping it to an AdminServer on the Intranet behind the DMZ server. The Intranet AdminServer uses HTTP/S to pass WSA administration operations as SOAP messages to the WSA’s Administration service.

To run a remote WSA configuration:

1. Install the remote WSA on the DMZ server. Manually edit the static startup parameters in its ubroker.properties file and install the WSA as a Web application in the local Web server's JSE.

2. Start the WSA the Web server's JSE and verify that a browser can access its URL from the Internet.

3. Connect to an AdminServer on the Intranet and configure a “remote” WSA instance using OpenEdge Explorer and Progress Explorer or the WSAMAN utility by specifying the remote WSA's URL.

   The Intranet AdminServer creates a WSA mapping entry in its local ubroker.properties file with the URL of the remote WSA’s administration SOAP service.

4. Use OpenEdge Explorer and Progress Explorer or the WSAMAN utility to manage the “remote” WSA’s dynamic run-time properties and deploy and manage the WSA's SOAP services.

OpenEdge Explorer and Progress Explorer connects to the Intranet AdminServer where you mapped the WSA and then connects to the remote WSA’s SOAP administration service. When OpenEdge Explorer and Progress Explorer sends an operation to the Intranet AdminServer, the AdminServer turns it into a SOAP message and forwards it to the WSA’s SOAP administration service which executes the operation.
Distributing your WSDL file

Eventually, you should also decide how you want to distribute the WSDL file for your Web service to your end users. Some methods are more public than others. How end users access the WSDL can affect the security of your Web service. Some of the ways that you can distribute a WSDL file are the following:

- Use e-mail
- Post it on a Web server
- Host it in a UDDI directory

**Note:** When the WSDL specification was first proposed, UDDI directories were intended to be the primary method of distributing the WSDL files for public Web services. Most major companies, like Microsoft and IBM, have since abandoned the concept.

You can also host the file on a network using a WSA by deploying the Web service using the OpenEdge Web services tools. This deployed WSDL file resides at a URL determined by the WSA installation and other information specified for the Web service at deployment. Once the Web service is deployed, the WSA administrator can make the WSDL available by enabling access to WSDL listings on the network. For more information on Web service deployment, see the Web service chapters of *OpenEdge Application Server: Administration*.

To obtain a WSDL file deployed to a WSA instance, you can download it over the network through its URL, specified using the following syntax:

**Syntax**

```
http[s]://host:port/web-server-context/wsapp-context/ws-instance/wsdl?targetURI=web-service-name
```

**Note:** The syntax is case sensitive.

For example, here is a WSDL retrieval URL for a Web service with the friendly name, OrderInfo, that is deployed to a WSA instance (wsa1) running in the WSA Web application context (wsa):

```
http://servicehost:80/wsa/wsa1/wsdl?targetURI=OrderInfo
```

For more information on the syntax of the WSDL retrieval URL, see the chapters on the WSA in *OpenEdge Application Server: Administration*.

**Note:** To aid developing client code for testing Web services under development, you can optionally generate a WSDL file from ProxyGen when generating pre-deployment versions of the Web service definition. ProxyGen writes this file to the specified output directory. For more information, see *OpenEdge Development: Open Client Introduction and Programming*.
After creating a Web service definition in ProxyGen for your AppServer application, you must test the Web service before deploying it into a production environment. To test your Web service, you must build a client to access it. If you are hosting the Web service on an Intranet, you might only need a single client for all your end users. But, if you intend to make your Web service publicly available, you should build several clients in various languages so you can test all the common possibilities.

**Note:** For more information on creating Web service definitions, see the chapter on generating proxies and Web service definitions in *OpenEdge Development: Open Client Introduction and Programming*.

To build clients to test your Web service, you must understand how clients and your Web service interact. The choices that you made while creating your Web service definition impact how clients use your Web service. The chapter discusses the general programming model for building clients to consume OpenEdge Web services.

The chapter covers general information that applies to any Web service as well as information specific to consuming OpenEdge Web services, in the following sections:

- Creating client interfaces from WSDL
- SOAP format impact on generated client interfaces
- Client programming for different session models
- Overview of calling methods on Web service objects
- Retrieving and sending object IDs—handling SOAP headers
- Defining and passing Web service method parameters
• Mapping relational data
• Mapping ABL procedures to SOAP messages
• Handling Web service errors and SOAP faults
Creating client interfaces from WSDL

As mentioned in the “Web service basics” section on page 1–2, advances in client toolkits have nearly eliminated the need to manually write a Web service client from the WSDL file. There are client toolkits available that can generate most of the client interface directly from the WSDL for most major platforms, including:

- Microsoft .NET
- Apache Axis
- OpenEdge

The proxies from these toolkits provide an API that generally appears as an interface to application functions. In fact, they provide an interface to messaging operations that hide the complexity of exchanging SOAP messages over HTTP.

Each client toolkit generates these proxies in a unique manner, but most create objects with methods. Usually, they create one object for each `<portType>` and one object for each `<complexType>` specified in the WSDL. Each client toolkit creates methods to generate and interpret SOAP messages appropriately for their platform.

All major client toolkits can generate a client interface from an OpenEdge Web service’s WSDL file.

WSDL element overview for OpenEdge Web services

Previous chapters have discussed the WSDL file in general. A brief look at the parts of a WSDL file is now appropriate. A WSDL file is an XML document consisting of a series of ordered sections. These sections contain the elements defining the Web service objects and their bindings. In general, each succeeding section element references elements defined by preceding sections. This ensures the object definition and its operations are associated with the required bindings for the object.

Thus, the WSDL files generated by client toolsets follow the W3C WSDL 1.1 specification and include the element sections listed in Table 3–1.

Table 3–1: WSDL element overview

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>definitions</strong></td>
<td>Defined by one occurrence of the <code>&lt;definitions&gt;</code> element. This section identifies the Web service, defines all valid namespaces for the WSDL file, and contains all other WSDL sections.</td>
</tr>
<tr>
<td><strong>types</strong></td>
<td>Defined by one occurrence of the <code>&lt;types&gt;</code> element. This section includes a separate schema for each Web service object. There is a separate schema that defines the SOAP fault <code>&lt;detail&gt;</code> element. These schema define the input (request) and output (response) parameters for the Web service object. <strong>Note:</strong> The definitions for built-in XML Schema data types are not listed in the types section.</td>
</tr>
</tbody>
</table>
For more information on WSDL and its meaning, see the WSDL specification at the following URL:

http://www.w3.org/TR/wsdl
SOAP format impact on generated client interfaces

The SOAP format of your Web service affects how a client toolkit defines the objects and methods for interacting with the Web service. The following sections compare how the SOAP format affects the representation of various Web service components.

For examples of how some of these components appear in SOAP, see Chapter 4, “Sample Code with SOAP Messages for OpenEdge Web Services.”

Method signatures

Client interface method signatures typically map to ABL prototypes as follows:

- **Doc/Lit** — Method signatures do not necessarily match the prototypes in the ABL
- **RPC/Literal** — Method signatures normally match the prototypes in ABL
- **RPC/Encoded** — Same as RPC/Literal

Method return values

Client interface methods typically provide return values for ABL procedures and user-defined functions as follows:

- **Doc/Lit** — Methods provide return values for:
  - User-defined functions (always). The result is either the method return value or the first output parameter.
  - Procedures (always). The result will either be the method return value or the first output parameter. If `Return ABL RETURN-VALUE` is specified in ProxyGen, the result is the ABL RETURN-VALUE. If `Return ABL RETURN-VALUE` is not specified in ProxyGen, the result is a null string.
- **RPC/Literal** — Methods provide return values for:
  - User-defined functions (always)
  - Procedures only when `Return ABL RETURN-VALUE` is specified in ProxyGen
- **RPC/Encoded** — Same as RPC/Literal
Client programming for different session models

You must program Web service clients differently, depending on the Web service session model. The following sections summarize these differences.

Caution: For any session model, failing to call the release method on any Open Client object that a Web service client uses (except a session-free AppObject) leaves the resources reserved for that object unavailable. In effect, this creates a memory leak in the WSA. The Web service deployer can manage these “orphaned” resources by property settings on the Web service. For more information, see the Web service properties reference sections of OpenEdge Application Server: Administration.

Programming clients for session-managed Web services

The programming model for session-managed Web services is similar to programming all other Open Clients, except for the use of object IDs:

- Before invoking any other method on a session-managed AppObject, you must instantiate (create) and call a connect method on the object to establish a connection to the AppServer context. The client must obtain the object ID for the AppObject after it is created. After that, you can invoke any method on the AppObject. The subsequent calls to all AppObject methods are handled sequentially using the established connection. You must call the release method on an AppObject when you no longer need the object (Web service).

- You can create ProcObjects and SubAppObjects using class factory methods on the AppObject or create ProcObjects on another SubAppObject. Once created, all objects share the same connection. This connection terminates only after you have called the release method on all Web service objects that you create. You must call each object's release method when you no longer need the object.

- You can call methods on a ProcObject as soon as you create it and after the client obtains the object ID for it. Calls to all ProcObject methods (internal procedures and user-defined functions) are sequential using the connection established by the AppObject. You must call the release method on a ProcObject when you no longer need the object.

- You can call methods on a SubAppObject as soon as you create it and after the client obtains the object ID for it. Calls to all SubAppObject methods are sequential using the connection established for the AppObject. You must call the release method on a SubAppObject when you no longer need the object.

- Once connected to a session-managed Web service, you must always send object IDs with each method call.
Programming clients for session-free Web services

The programming model for session-free Web Services differs significantly from the session-managed model:

- After you instantiate (create) a session-free AppObject, you can call any method on the object and you never connect to an AppServer. Calls to all AppObject methods are executed in parallel (calls execute simultaneously) using the available AppServer connections in the Web service connection pool (a pool of AppServer connections maintained for each session-free Web service). Each call to the AppObject is independent of every other call and can come from any client, in any order.

- You can create SubAppObjects and ProcObjects using class factory methods. Although no connection is established for the AppObject, SubAppObjects and ProcObjects are maintained uniquely for each client, and ProcObjects reserve their own AppServer connections from the Web service connection pool. You must call each object’s release method when you no longer need the object and to return each object’s AppServer connections to the connection pool.

- You can call methods on a ProcObject as soon as you create it and after the client obtains the object ID for it. Calls to all ProcObject methods (internal procedures and user-defined functions) are sequential using the connection established for the ProcObject. You must call the release method on a ProcObject when you no longer need the object and to return the object’s AppServer connection to the connection pool.

- You can call methods on a SubAppObject as soon as you create it and after the client obtains the object ID for it. Calls to all SubAppObject methods are executed in parallel. You must call the release method on a SubAppObject when you no longer need the object, to remove it from the list of client SubAppObjects maintained by the WSA and to return its AppServer connection to the connection pool.

- Once an object is created, you must always send object IDs with each method call on a session-free SubAppObject or ProcObject, but you never send object IDs on method calls to a session-free AppObject.

Note: SubAppObjects and ProcObjects have limited utility in the session-free model and generally should not be defined for session-free Web services. For more information, see the “How session models affect Web service objects” section on page 2–5.
Overview of calling methods on Web service objects

The following sections provide an overview of the methods supported on Web service objects, and how to use them based on the session model of the Web service. The bold-italicized variable names are replaced by real object and method names during WSDL file generation:

- **AppObject** — The name of the AppObject for which the named item is defined or associated
- **SubAppObject** — The name of the SubAppObject for which the named item is defined or associated
- **ProcObject** — The name of the ProcObject for which the named item is defined or associated
- **ProcName** — The name of a method that executes an ABL non-persistent procedure or internal procedure on the AppServer
- **FuncName** — The name of a method that executes an ABL user-defined function on the AppServer

**Note:** Open Client object names have fewer legal characters to match restrictions in non-ABL environments. Avoid special characters and hyphens in object names.

Many of the methods referenced here are standards from the Open Client programming model. The precise syntax for invoking these methods differs depending on the client development language used to access Web services; however, the basic signatures for these methods are the same in all languages. For more information on how these methods are used in Web service clients, see Chapter 4, “Sample Code with SOAP Messages for OpenEdge Web Services.”

The following sections describe usage for:

- Client interface methods for session-managed AppObjects
- Client interface methods for session-free AppObjects
- Client interface methods for SubAppObjects and ProcObjects

**Client interface methods for session-managed AppObjects**

A session-managed AppObject provides the following methods:

- `Connect_AppObject(...)
- Release_AppObject()
- CreateAO_SubAppObject()
- CreatePO_ProcObject(...)
- `ProcName(...)`
Overview of calling methods on Web service objects

Use this method to connect to an AppServer:

```
Connect_AppObject ( )
```

This method must be executed before any other method can be called. The SOAP response header returned by this method contains an `AppObjectID` element whose value must be sent in the SOAP request header for all other methods invoked on this AppObject.

The AppServer Connect procedure has the ability to return user-defined strings to clients using the WSA or the Sonic ESB Adapter on success or failure. The ABL Connect procedure uses the `ABL RETURN ERROR string` or `RETURN string` statements to specify the return strings. By default, the strings are returned to the WSA or Sonic ESB Adapter and logged.

To access the strings on the actual client, the proxy must have been generated using the `Return ABL RETURN-VALUE on Connect` option on the `General` tab in ProxyGen.

On a successful connection, the client can access the returned string by the `result OUTPUT` parameter on the associated `Connect_AppObject()` method or as the return value of `Connect_AppObject()` if it is run as a function. On a failed connection, the client can access the returned string in the `<faultstring>` element of the returned SOAP fault. If no user-defined string was returned from the Connect procedure, then parameter value or SOAP element will be an empty string.

**Note:** ProxyGen normally configures a `Connect_AppObject()` method with three parameters (`string userid`, `string password`, and `string AppServerInfo`). Enabling the `Return ABL RETURN-VALUE on Connect` option adds the output parameter `string result`, which contains the return value of the Connection procedure.

This option does not affect the normal process of returning SOAP faults when a connection attempt fails from a client using a Web Services Adapter. That is, when the Connect procedure fails and returns a string value, that string value will be sent to the client in the `<faultstring>` element of a SOAP fault. If the Connect procedure does not return a string, then the normal SOAP fault will be returned to the client.

Use this method to release the AppObject connection:

```
Release_AppObject ( )
```

Once this method is executed, no other methods on the AppObject can be called. The SOAP request header must contain the value of the `AppObjectID` element. If other objects (SubAppObjects or ProcObjects) are using the same connection, the connection is not terminated until the corresponding release method is called on every object.

Use this method to create a SubAppObject:

```
CreateAO_SubAppObject ( )
```

This method must be executed before calling any other methods on the SubAppObject. The SOAP request header must contain the value of the `AppObjectID` element. The SOAP response header returned by this method contains a `SubAppObjectID` element whose value must be sent in the SOAP request header for all methods invoked on this SubAppObject.
Use this method to create a ProcObject and execute the corresponding persistent procedure:

\[ \text{CreatePO}_{-} \text{ProcObject} (\ldots) \]

This method must be executed before calling any other methods on the ProcObject. The SOAP request header must contain the value of the AppObject ID. The SOAP response header returned by this method contains a ProcObject ID whose value must be sent in the SOAP request header for all methods invoked on this ProcObject.

Call a ProcName method on the AppObject to execute a corresponding non-persistent external procedure typically identified by ProcName:

\[ \text{ProcName} (\ldots) \]

The SOAP request header must contain the value of the AppObject ID element.

**Client interface methods for session-free AppObjects**

A session-free AppObject provides the following methods:

- ProcName (\ldots)
- CreateAO_{-}SubAppObject ()
- CreatePO_{-}ProcObject (\ldots)

Call a ProcName method on the AppObject to execute a corresponding non-persistent external procedure typically identified by ProcName:

\[ \text{ProcName} (\ldots) \]

Because no context is maintained for a session-free AppObject, calls to these methods require no object IDs to be sent in the SOAP request header.

Use this method to create a SubAppObject:

\[ \text{CreateAO}_{-} \text{SubAppObject} ( ) \]

**Note:** SubAppObjects are not recommended for use in session-free Web services.

This method must be executed before calling any other methods on the SubAppObject. The SOAP response header returned by this method contains a SubAppObject ID element whose value must be sent in the SOAP request header for all methods invoked on this SubAppObject. Because no context is maintained for a session-free AppObject, calls to this method require no object ID to be sent in the SOAP request header.
Use this method to create a ProcObject and execute the corresponding persistent procedure:

```
CreatePO_ProcObject (***)
```

**Note:** ProcObjects are not recommended for use in session-free Web services.

This method must be executed before calling any other methods on the ProcObject. The SOAP response header returned by this method contains a `ProcObjectID` element whose value must be sent in the SOAP request header for all methods invoked on this ProcObject. Because no context is maintained for a session-free AppObject, calls to this method requires no object ID to be sent in the SOAP request header.

### Client interface methods for SubAppObjects and ProcObjects

You use the methods for ProcObjects and SubAppObjects in the same way regardless of the session model. The value for the `ProcObjectID` element or `SubAppObjectID` element that you send in the SOAP request header for methods invoked on each object is initially returned in the SOAP response header of the `CreatePO_ProcObject` method or `CreateAO_SubAppObject` method that you called to instantiate the object.

#### ProcObject methods

A ProcObject provides the following methods:

- `Release_ProcObject ()`
- `ProcName (***)`
- `FuncName (***)`

Use this method to terminate and remove the context of the persistent procedure that is managed by the ProcObject:

```
Release_ProcObject ( )
```

Once this method is executed, no other methods on the ProcObject can be called. The SOAP request header must contain the value of the `ProcObjectID` element. If other objects (AppObject or SubAppObjects) are sharing the same connection, the connection does not terminate until you call the release method for all the objects sharing the connection.

Call a `ProcName` method on the ProcObject to execute a corresponding internal procedure in the persistent procedure that is managed by the ProcObject:

```
ProcName (***)
```

The SOAP request header must contain the value of the `ProcObjectID` element.
Call a `FuncName` method on the ProcObject to execute a corresponding user-defined function in the persistent procedure that is managed by the ProcObject:

```
FuncName (...)
```

The SOAP request header must contain the value of the `ProcObjectID` element.

### SubAppObject methods

A SubAppObject provides the following methods:

- `Release_SubAppObject()`
- `ProcName(...)`
- `CreatePO_ProcObject(...)`

Use this method to release a SubAppObject:

```
Release_SubAppObject()
```

Once this method is executed, no other methods on the SubAppObject can be called. The SOAP request header must contain the value of the `SubAppObjectID` element. If other objects (AppObject or ProcObjects) are sharing the same connection, the connection does not terminate until you call the release method for all the objects sharing the connection.

Call a `ProcName` method on the SubAppObject to execute a corresponding non-persistent external procedure typically identified by `ProcName`:

```
ProcName (...)
```

The SOAP request header must contain the value of the `SubAppObjectID` element.

Use this method to create a ProcObject and call the corresponding persistent procedure:

```
CreatePO_ProcObject (...)
```

This method must be executed before calling any other methods on the ProcObject. The SOAP request header must contain the value of the `SubAppObjectID` element. The SOAP response header returned by this method contains a `ProcObjectID` element whose value must be sent in the SOAP request header for all methods invoked on this ProcObject.
Retrieving and sending object IDs—handling SOAP headers

The “Session Models and object IDs” section on page 2–3 explains that object IDs must be retrieved from and sent in SOAP headers for methods on the following Open Client objects:

- Session-managed AppObjects
- ProcObjects
- SubAppObjects

Each client toolkit provides different means to access SOAP headers. In some environments, such as Microsoft .NET, the interface automatically moves information from the response header to subsequent request headers within the same object. You do have to do a little work to move the information between objects (for example, copying a ProcObject ID from the AppObject that creates a ProcObject to the ProcObject itself). In other environments, you are responsible for extracting the information from the response header and sending it in the SOAP headers for subsequent requests.

The WSDL defines the SOAP elements that hold the object ID using a `<complexType>` declaration in the `types` section for each Web service object schema that requires it. The headers in the SOAP messages that are sent and received for a given object then contain the required object ID values.

Defining object IDs in WSDL

The follow sample shows the `types` section that might appear in the WSDL for many of the sample Web services in this chapter. (This example is for Doc/Lit, but similar for all SOAP formats.) For example:

WSDL Containing an object ID definition

```xml
<types>
  <schema elementFormDefault="qualified"
    targetNamespace="urn:OrderSvc:OrderInfo"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <complexType name="OrderInfoID">
      <sequence>
        <element name="UUID" type="xsd:string"/>
      </sequence>
    </complexType>
  </schema>
  <!-- Additional schema declarations here (if any) -->
</types>

<!-- Additional Object Schema Declarations (if any) -->
```

Note: The samples in this chapter are not all available on the Documentation and Samples (doc_samples) directory of the OpenEdge product DVD or Progress Documentation Web site.
The WSDL schema definition for each Web service object (except a session-free AppObject) defines the SOAP header element that contains the value of its object ID as a `<complexType>`. As shown in the example, the name for the SOAP header element is the name of the `<complexType>`, in this example, "OrderInfoID". The name of the element that contains the value of the object ID is "UUID", defined as an element of type "xsd:string". This definition is the same for the object IDs of all Web service objects, differing only in the name of the object ID element.

### Using object IDs in a SOAP message

This is a sample header of a SOAP message containing an object ID for the object specified in the sample WSDL, using the WSDL-defined `<OrderInfoID>` SOAP element:

```xml
<soap:Header>
  <OrderInfoID xmlns="urn:OrderSvc:OrderInfo" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xsd:string">
    2e62cab6b81150d5:167f64e:f295e997b0:-8000:PX-000001|AO;M/IryPm3pi0Cf/W5DsH4GA==</UUID>
  </OrderInfoID>
</soap:Header>
```

The value of the object ID appears in bold, contained by the `<UUID>` element.

The same basic `<OrderInfoID>` element contains the object ID in the SOAP response header that initially returns the `OrderInfoID` object ID, when the object is created, and is required in any subsequent SOAP request header sent by methods invoked on the `OrderInfo` object.

Thus, for all Web service objects that require object IDs, your client application must parse out the object ID value in the `<UUID>` element contained by the object ID element (`<OrderInfoID>` in the example) and returned in the SOAP response header at object creation, whether it happens automatically or you must program it explicitly. Similarly, for every SOAP request sent by a method invoked on this same object, your client application must create an object ID element containing a `<UUID>` element that holds this identical object ID value in the SOAP header of the request message.

For examples of SOAP headers that contain object IDs and sample code that handles object IDs, see the following chapters:

- Chapter 4, “Sample Code with SOAP Messages for OpenEdge Web Services”
- Appendix A, “Developing a .NET Client to Consume OpenEdge Web Services”
- Appendix B, “Developing a Java Client to Consume OpenEdge Web Services”

For examples of client applications that handle object IDs, see the sample OpenEdge Web service applications available on the Documentation and Samples (doc_samples) directory of the OpenEdge product DVD. For information on how to access these applications, see the “Sample Web service applications” section on page 1–17.
Defining and passing Web service method parameters

Most parameters are defined and passed in SOAP messages as XML Schema data types. OpenEdge Web services expose ABL parameter data types as XML data types in the WSDL based on the “2001 XML Schema: Data types” (referred to as xsd in Table 3–2).

Table 3–2 lists the data type mappings for parameters between ABL and XML data types. Most of these parameters can also be passed as arrays.

Table 3–2: Supported XML data types

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>XML Schema data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>xsd:string</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>xsd:long</td>
</tr>
<tr>
<td>DATE</td>
<td>xsd:date</td>
</tr>
<tr>
<td>DATETIME</td>
<td>xsd:dateTime</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>xsd:dateTime</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>xsd:decimal</td>
</tr>
<tr>
<td>INT64</td>
<td>xsd:long</td>
</tr>
<tr>
<td>INTEGER (32 bit)</td>
<td>xsd:int</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>xsd:boolean</td>
</tr>
<tr>
<td>LONGCHAR(^1)</td>
<td>xsd:string</td>
</tr>
<tr>
<td>MEMPTR(^1)</td>
<td>xsd:base64Binary</td>
</tr>
<tr>
<td>RAW</td>
<td>xsd:base64Binary</td>
</tr>
<tr>
<td>RECID (32 or 64 bit)</td>
<td>xsd:long</td>
</tr>
<tr>
<td>ROWID</td>
<td>xsd:base64Binary</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>xsd:long</td>
</tr>
<tr>
<td>TABLE (static temp-table)(^2)</td>
<td>complexType</td>
</tr>
<tr>
<td>TABLE-HANDLE (dynamic temp-table)(^2)</td>
<td>complexType (&lt;any&gt;)</td>
</tr>
<tr>
<td>DATASET (static ProDataSet)(^2)</td>
<td>complexType</td>
</tr>
<tr>
<td>DATASET-HANDLE (dynamic ProDataSet)(^2)</td>
<td>complexType (&lt;any&gt;)</td>
</tr>
</tbody>
</table>

1. LONGCHAR and MEMPTR data types are designed to support very large “strings” of data. Use of these data types as parameters in Web services can result in a serious performance impact.

2. TABLE, TABLE-HANDLE, DATASET, and DATASET-HANDLE data types cannot be passed as arrays. For more information on passing arrays, see the “Array parameters” section on page 3–17.

**Note:** ABL object type (class or interface) parameters and ABL BUFFER parameters are not supported.
Client data type mapping

Each client environment provides its own mapping from XML Schema data types to client language data types. For example, an ABL CHARACTER parameter is mapped to a java.lang.String class in Java and a System.String type in .NET.

Relational data types

OpenEdge WSDL represents TABLE, TABLE-HANDLE, DATASET, and DATASET-HANDLE parameters as <complexType> element definitions. The “Mapping relational data” section on page 3–18 describes these <complexType> elements.

Date and time data

The ABL DATE, DATETIME, and DATETIME-TZ data types are explicit with respect to the inclusion or exclusion of time and time zone values. When sending date/time values to an OpenEdge Web service, the client application must precisely match the expected format. The WSA returns a SOAP fault if the client request fails to include a time or time zone when that information is expected, or if the client includes unexpected time or time zone information.

In the case of a DATE field, the WSDL document maps the item unambiguously to the XML Schema data type xsd:date. Both DATETIME and DATETIME-TZ, however, map to the xsd:dateTime data type; therefore, the WSDL does not tell the client developer whether a time zone value is required. The developer must obtain this information by another means and must ensure that the client application sends the expected value.

Table 3–3 lists the XML date data formats that are valid as input for each of the ABL data types; use of any other input format results in a SOAP fault.

Table 3–3: Valid date input formats

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Valid input formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>CCYY-MM-DD</td>
</tr>
<tr>
<td>DATETIME&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CCYY-MM-DOTMm:ss</td>
</tr>
<tr>
<td></td>
<td>CCYY-MM-DOTMm:ssZ</td>
</tr>
<tr>
<td></td>
<td>CCYY-MM-DOTMm:ss-hh:mm</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>CCYY-MM-DOTMm:ssZ</td>
</tr>
<tr>
<td></td>
<td>CCYY-MM-DOTMm:ss+hh:mm</td>
</tr>
<tr>
<td></td>
<td>CCYY-MM-DOTMm:ss-hh:mm</td>
</tr>
</tbody>
</table>

<sup>1</sup> Any time zone information input to DATETIME is lost.

Dedicated ABL data types

The COM-HANDLE, RECID, ROWID, and WIDGET-HANDLE ABL data types are not meaningful outside the ABL environment. You can obtain their values from an ABL procedure (through one Web service method) and pass them back to another ABL procedure (through another Web service method), but there is no other practical use for them on the client side.
Array parameters

The WSA supports arrays (extents) as parameters for the data types listed in Table 3–2. These data types are mapped to XML Schema data types as indicated in this. In the WSDL document, the XML Schema data types are housed in a complex structure whose type definition depends on the encoding style.

**Note:** TABLE (temp-table), TABLE–HANDLE (temp-table handle), DATASET (ProDataSet), and DATASET–HANDLE (ProDataSet handle) parameters cannot be array parameters.

Fixed arrays

In cases where ABL refers to fixed-array parameters, only the Doc/Lit WSDL style/use format represents the fixed size of the array; the RPC/Encoded and RPC/Literal formats represent all arrays as unbounded. Therefore, when using either of the RPC WSDL style formats, the developer must know the size of each array parameter that is expected. The WSA returns a SOAP fault if the array sizes do not match.

**Note:** The Doc/Lit schema represents each parameter individually, and thus specifies the size for each fixed array parameter.

For details and examples of how WSDL represents arrays, see Appendix E, “Understanding WSDL Details.”
Mapping relational data

OpenEdge Web services support the passing of ABL static and dynamic temp-tables and ProDataSets as parameters. The WSDL Analyzer enables ABL clients to handle all these parameters simply. Non-ABL clients require more coding to handle dynamic temp-tables and ProDataSets and to handle ProDataSets with before-image data.

For examples of client applications using relational data, see Chapter 4, “Sample Code with SOAP Messages for OpenEdge Web Services.” For examples of how WSDL represents relational data, see Appendix E, “Understanding WSDL Details.”

Defining TABLE (static temp-table) parameters

TABLE parameters pass only the data, because the static temp-table’s schema is known at WSDL generation. In the WSDL, 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, client interfaces typically represent temp-table parameters as follows:

- For every temp-table, a row is represented by an object.
- Every temp-table parameter is represented as an array of that temp-table’s row objects.
- For an ABL client, the WSDL Analyzer transforms the WSDL definition of a temp-table into a matching ABL temp-table definition.

The following are general formats for TABLE parameters in SOAP messages. For an RPC/Literal or Doc/Lit Web service, the TABLE parameter is represented as a <sequence> of TABLE row elements. Each element is named after the row <element> in the <complexType> used to define the TABLE row for the parameter. This WSDL-named row <element> corresponds to the <Item> element used to represent SOAP array rows in RPC/Encoded Web services.

Thus, using the sample row element named ttEmpRow, a SOAP message contains a TABLE parameter for this row definition in the following form:

TABLE parameters—general Document (or RPC)/Literal format

```
<ttEmpRow> <!-- row instance 1 --> </ttEmpRow>
<ttEmpRow> <!-- row instance 2 --> </ttEmpRow>
<ttEmpRow> <!-- row instance 3 --> </ttEmpRow>
... 
```

For an RPC/Encoded Web service, the TABLE parameter is represented as a SOAP array of TABLE rows, where each row is encapsulated by an <Item> element:

TABLE parameters—general RPC/Encoded format

```
<Item> <!-- row instance 1 --> </Item>
<Item> <!-- row instance 2 --> </Item>
<Item> <!-- row instance 3 --> </Item>
... 
```
Each column of a TABLE row can hold any data type shown in Table 3–4.

### Table 3–4: XML data types for TABLE parameter columns

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>XML Schema data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB(^1)</td>
<td>xsd:base64Binary</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>xsd:string</td>
</tr>
<tr>
<td>CLOB(^1)</td>
<td>xsd:string</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>xsd:long</td>
</tr>
<tr>
<td>DATE</td>
<td>xsd:date</td>
</tr>
<tr>
<td>DATETIME</td>
<td>xsd:dateTime</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>xsd:dateTime</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>xsd:decimal</td>
</tr>
<tr>
<td>INT64</td>
<td>xsd:long</td>
</tr>
<tr>
<td>INTEGER (32 bit)</td>
<td>xsd:int</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>xsd:boolean</td>
</tr>
<tr>
<td>RAW</td>
<td>xsd:base64Binary</td>
</tr>
<tr>
<td>RECID (32 or 64 bit)</td>
<td>xsd:long</td>
</tr>
<tr>
<td>ROWID</td>
<td>xsd:base64Binary</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>xsd:long</td>
</tr>
</tbody>
</table>

1. 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.

### Defining 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. In the WSDL, OpenEdge Web services map an ABL TABLE-HANDLE to a `<complexType>` containing a sequence of `xsd:any`.

For all SOAP formats, client interfaces typically represent TABLE-HANDLE parameters as follows:

- For every TABLE-HANDLE parameter within a Web service object, there is a single object representing all TABLE-HANDLES.
- In both request messages and response messages, the schema of the TABLE-HANDLE must accompany the data.
- For every input TABLE-HANDLE, you must include the schema of the TABLE-HANDLE in the form of an XML Schema followed by the data in the form of an XML document fragment.
For every output TABLE-HANDLE, a non-ABL client must parse the XML Schema and data in the SOAP response message.

The WSDL Analyzer recognizes the WSDL definition of a TABLE-HANDLE parameter and maps it to a TABLE-HANDLE parameter for an ABL client.

The following WSDL sample shows this common TABLE-HANDLE parameter definition:

**TABLE-HANDLE definition for all dynamic temp-table parameters**

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

The non-ABL 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.

This is the general format in OpenEdge Web services for representing a TABLE-HANDLE in a SOAP message, where the schema is defined in a `<schema>` element and each row is encapsulated by an `<Item>` element within a `<Data>` element:

**TABLE-HANDLE parameters—general format**

```xml
<DataSet>
  <schema>
    <!-- TEMP-TABLE row definition in XML Schema -->
  </schema>
  <Data>
    <Item> <!-- row instance 1 --> </Item>
    <Item> <!-- row instance 2 --> </Item>
    <Item> <!-- row instance 3 --> </Item>
    ...
  </Data>
</DataSet>
```

Each column of a TABLE-HANDLE row can hold any data type shown in Table 3–5.

**Table 3–5: XML data types for TABLE-HANDLE parameter columns (1 of 2)**

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>XML Schema data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>xsd:string</td>
</tr>
<tr>
<td>DATE</td>
<td>xsd:date</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>xsd:dateTime</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>xsd:decimal</td>
</tr>
<tr>
<td>INT64</td>
<td>xsd:long</td>
</tr>
<tr>
<td>INTEGER (32 bit)</td>
<td>xsd:int</td>
</tr>
</tbody>
</table>
Defining DATASET (static ProDataSet) parameters

DATASET parameters pass only the data, because the static ProDataSet’s schema is known at WSDL generation. In the WSDL, 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.

For all SOAP formats, client interfaces typically represent dataset parameters as follows:

- For every dataset, each temp-table is represented by an object.
- Every dataset parameter is represented as arrays of the constituent temp-table objects.
- For an ABL client, the WSDL Analyzer transforms the WSDL definition of a ProDataSet parameter into a matching ProDataSet.

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. If the ProDataSet parameter includes before-image data, the before-image data is serialized in a proprietary OpenEdge `datasetChanges` document.

The following snippet shows the general format for DATASET parameters in SOAP messages:

**DATASET parameters—general SOAP format**

```xml
<Buffer1Name> <!-- TEMP-TABLE-1 row instance 1 --> </Buffer1Name>
<Buffer1Name> <!-- TEMP-TABLE-1 row instance 2 --> </Buffer1Name>
...
<Buffer2Name> <!-- TEMP-TABLE-2 row instance 1 --> </Buffer2Name>
<Buffer2Name> <!-- TEMP-TABLE-2 row instance 2 --> </Buffer2Name>
...
</PDSname>
```

---

Table 3–5: XML data types for TABLE-HANDLE parameter columns (2 of 2)

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>XML Schema data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGICAL</td>
<td><code>xsd:boolean</code></td>
</tr>
<tr>
<td>RAW</td>
<td><code>xsd:base64Binary</code></td>
</tr>
</tbody>
</table>
Defining 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. In the WSDL, 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.

For all SOAP formats, client interfaces typically represent DATASET-HANDLE parameters as follows:

- All DATASET-HANDLE parameters within a Web service object are represented as one of two objects, one for ProDataSets without before-image data and one for ProDataSets with before-image data.
- In both request messages (input) and response messages (output), the schema of the DATASET-HANDLE must accompany the data.
- For every input DATASET-HANDLE, you must include the schema of the DATASET-HANDLE in the form of an XML Schema followed by the data in the form of an XML document fragment.
- For every output DATASET-HANDLE, a non-ABL client must parse the XML Schema and data in the SOAP response message.
- For an ABL client, the WSDL Analyzer recognizes the WSDL definition of a DATASET-HANDLE parameter and maps it to a DATASET-HANDLE parameter for an ABL client.

The following WSDL sample shows this common ProDataSet parameter definition:

```
<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>
```

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.
This is the general format in OpenEdge Web services for representing a DATASET-HANDLE in a SOAP message:

### DATASET-HANDLE parameters—general format

```xml
<DataSet>
  <schema ...>
    <element ProDataSet>
      <!-- TEMP-TABLE-1 definition in XML Schema -->
      <!-- TEMP-TABLE-2 definition in XML Schema -->
      . . .
    </element>
    . . .
  </schema>
  <ProDataSet>
    <TEMP-TABLE-1> <!-- row instance 1 --> </TEMP-TABLE-1>
    <TEMP-TABLE-1> <!-- row instance 2 --> </TEMP-TABLE-1>
    . . .
    <TEMP-TABLE-2> <!-- row instance 1 --> </TEMP-TABLE-2>
    <TEMP-TABLE-2> <!-- row instance 2 --> </TEMP-TABLE-2>
    . . .
  </ProDataSet>
</DataSet>
```

### Additional considerations

When dealing with relational data in an OpenEdge Web service, you should also consider:

- Including before-image data
- Name collisions
- R-code header changes

#### Including 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. All SOAP formats represent this data as arbitrary complex data with XML Schema attributes that identify it as an OpenEdge 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. 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.
Name collisions

A ProxyGen AppObject or SubAppObject can contain multiple procedures, which might have ProDataSets with the same name. If such a collision occurs within an object, ProxyGen checks the ProDataSet’s definition and responds as follows:

1. If the definitions match, ProxyGen generates a single XML Schema definition for that ProDataSet name. All operations using that ProDataSet name reference the single XML Schema definition.

2. If the definitions do not match, ProxyGen renames the second and subsequent ProDataSets of that name. It renames them by adding a counter to the end of the ProDataSet name. This renaming shows up in different places depending on the style/encoding that you are using.

R-code header changes

ProxyGen uses information in the headers of r-code files to generate the WSM file. So, the WSDL definition of a ProDataSet is limited by the information written into the r-code header. Starting with OpenEdge R10.1C, the r-code header includes support for a temp-table or ProDataSet’s NAMESPACE-URI attribute and a temp-table field’s XML-NODE-TYPE attribute. The r-code header also includes support for the XML-NODE-NAME attribute on ProDataSets, temp-tables, and temp-table fields.

To accommodate the addition of the NAMESPACE-URI and XML-NODE-NAME attributes, the r-code header includes a header version number which is independent of the r-code version. The AVM adds this element to the r-code header only if the file contains a temp-table or ProDataSet with the NAMESPACE-URI or XML-NODE-NAME attributes. When this element is present, you cannot use the r-code file with the ProxyGen from releases before Release 10.1C. Previous versions of ProxyGen return an error if they encounter this attribute.
Mapping ABL procedures to SOAP messages

The mapping of an ABL procedure (or, similarly, a user-defined function) to its final representation as a pair of SOAP request/response messages follows a well-defined series of steps:

1. ProxyGen generates a Web service definition that includes the selected ABL procedure in the context of some Open Client object during Web service development (see OpenEdge Development: Open Client Introduction and Programming).

2. The WSA generates the WSDL file for the Web service during Web service deployment (see OpenEdge Application Server: Administration). This WSDL file includes elements that define a Web service operation that maps to the selected ABL procedure.

3. A developer uses a client toolkit to generate the source code for the client interface from the WSDL file. The client toolkit reads the WSDL file and writes out the source code for the client interface as a series of objects and methods, each of which represents a Web service operation.

Note: In OpenEdge, the WSDL Analyzer generates documentation on how to represent Web service operations in ABL. For more information, see the “Using the WSDL Analyzer” section on page 6–3.

4. The developer writes a client application using the client interface generated in Step 3. The application invokes the method mapped to the Web service operation defined in Step 2, which sends a SOAP request message to the WSA where the Web service is deployed.

5. The WSA receives the SOAP request and packages the input parameters in an AppServer message which results in executing the corresponding an ABL procedure on an AppServer.

6. The ABL procedure then returns any output parameters and any return value to the WSA.

7. The WSA packages these procedure results in a SOAP response message, which it returns to the client that originally sent the request.

In short, the process transforms:

1. ABL procedure prototype to WSDL operation

2. WSDL operation to client method prototype

3. Client method call to SOAP request/response message pair

The examples that follow show how these three transformations proceed.

Note: Some of these examples are available as samples on the Documentation and Samples (doc_samples) directory of the OpenEdge product DVD or Progress Documentation Web site. For more information on accessing them, see the “Sample Web service applications” section on page 1–17.
ABL procedure prototype to WSDL operation

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

ABL procedure prototype

```abl
/* FindCustomerByNum.p */

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

FIND FIRST Customer WHERE Customer.CustNum = CustomerNumber NO-ERROR.
IF AVAILABLE Customer THEN
  CustomerName = Customer.NAME.
ELSE
  CustomerName = ?.
```

ProxyGen can extract some of the needed information from this prototype to map it to a corresponding Web service operation definition, including the procedure name (filename for an external procedure) and the parameter mode (input or output), names, and data types of any parameters. However, you must specify some of the needed information directly in ProxyGen, such as whether the ABL RETURN-VALUE is used and (for external procedures) the Open Client object to which the operation belongs.

For an example of how WSDL represents a procedure call, see Appendix E, “Understanding WSDL Details.”

WSDL operation to client method prototype

This is a VB.NET method prototype for the FindCustomerByNum operation defined in the WSDL file:

Interface method prototype generated from a WSDL operation definition

```vbnet
Public Sub FindCustomerByNum(
  ByVal CustomerNumber As Integer,
  ByRef CustomerName As String)
```

This prototype has basically the same information as the original ABL procedure prototype in FindCustomerByNum.p. In this case, ByVal specifies a value for the CustomerNumber parameter used for input, and ByRef specifies a reference to a variable for the CustomerName parameter used for output. Also, when VB.NET generates the client interface object that contains this method, the information provided in the WSDL portType, bindings, and service sections specify the object on which this method is defined (<portType>), the format of SOAP messages for this method (<binding>), and the location of the WSA instance to which the Web service is deployed (<port> within <service>).
Client method call to SOAP request/response message pair

This is a sample VB.NET method call to the `FindCustomerByNum()` method defined in the client interface, where the `OrderInfo` AppObject instance on which the method is called is named `webService`:

```vbnet
Dim CustomerName As String
webService.FindCustomerByNum(3, CustomerName)
```

Thus, the call passes a value of 3 for the `CustomerNumber` parameter as input and receives the value returned by the `CustomerName` output parameter in a variable that happens also to be named `CustomerName`.

This is the SOAP request message sent out by the client after invoking the `FindCustomerByNum()` method. You can see that the value (3) for the input parameter, `CustomerNumber`, is passed in the SOAP message body:

```xml
<soap:Body>
  <FindCustomerByNum xmlns="urn:OrderSvc:OrderInfo">
    <CustomerNumber>3</CustomerNumber>
  </FindCustomerByNum>
</soap:Body>
</soap:Envelope>
```

This is the SOAP response message returned by the WSA with the value ("Hoops") for the output parameter, `CustomerName`:

```xml
<soap:Body>
  <FindCustomerByNumResponse xmlns="urn:OrderSvc:OrderInfo">
    <CustomerName>Hoops</CustomerName>
  </FindCustomerByNumResponse>
</soap:Body>
</soap:Envelope>
```

If an error occurred at any point after the SOAP request message was received by the WSA, a SOAP fault message would be returned instead of the SOAP response message shown in the example. For information on SOAP fault messages returned for a method, see the “Handling Web service errors and SOAP faults” section on page 3–28.
Handling Web service errors and SOAP faults

Clients receive Web service errors in the form of a SOAP fault. A SOAP fault specifies a particular format for the body of SOAP response messages that return errors instead of successful responses to SOAP requests. Errors are also reflected in log files for various components of the OpenEdge Web services tools and the Web server or JSE. Run-time errors, which are normally returned to the client application, can occur at the following points in a message exchange:

- **Client errors occur:**
  - Before a request is sent to the Web service. The nature of these errors and how they are propagated depends on the client platform.
  - While the request is being sent to the Web service. These errors can be written to a log and might result in a SOAP fault.

- **Server errors occur** after the Web service begins processing the request, preparing and sending the request to the AppServer. These errors can be written to a log and result in a SOAP fault.

Client programming for SOAP faults

Client interfaces typically convert SOAP faults to client exceptions. Client code should handle SOAP faults and alert the user. Client languages catch errors in different ways. One of the more common techniques is to use try...catch blocks such as those used in Java and C#.

**Note:** For ABL, OpenEdge translates SOAP faults into ABL errors and provides access to the SOAP fault information. For more information, see Chapter 10, “Handling Errors in ABL Requests to Web Services.”

**Caution:** In any catch block or other error routine where you exit the program, you must release all Web service objects that you have created in the program.
This is the general format for a SOAP fault, with content indicated by the XML comments:

**SOAP faults—general format**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope namespaces defined here…>
  <soap:Body>
    <soap:Fault>
      <faultcode> <!-- Client or Server Environment --> </faultcode>
      <faultstring> <!-- Basic Error Message (12345) --> </faultstring>
      <detail>
        <FaultDetail xmlns="http://... WSA root URL ...">
          <errorMessage xsi:type="xsd:string">
            <!-- Initial error message (99999) -->
          </errorMessage>
          <requestID> <!-- Unique request ID --> </requestID>
        </FaultDetail>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

For more information on catching and viewing SOAP faults and working with log files to debug OpenEdge Web services, see the “Working with SOAP faults” section on page 5–8.
Sample Code with SOAP Messages for OpenEdge Web Services

The following samples provide individual examples of how SOAP messages are sent between the client and Web service, and code that produces some of these messages. The code samples are written in VB.NET or C#.NET and are organized into the following sections:

- Sample Web service specifications
- Consuming a session-managed Web service
- Consuming a session-free Web service
- Running an ABL non-persistent procedure
- Creating and using a ProcObject
- Running an internal procedure or user-defined function
- Releasing an object
- Passing static and dynamic temp-table parameters
- Receiving a SOAP fault message
- Passing static and dynamic ProDataSet parameters

Note: You can find some of the sample applications in OpenEdge_install_dir/src/samples/webservices.
Sample Web service specifications

These examples rely on sample Web service specifications shown in Table 4–1.

Table 4–1: Sample Web service specifications

<table>
<thead>
<tr>
<th>Property or component</th>
<th>Configuration, value, or name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session model</td>
<td>Managed</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Some information and examples are also provided to show session-free interactions.</td>
</tr>
<tr>
<td>SOAP format</td>
<td>Doc/Lit</td>
</tr>
<tr>
<td>TargetNamespace</td>
<td>&quot;urn:OrderSvc&quot;</td>
</tr>
<tr>
<td>AppObject</td>
<td>OrderInfo</td>
</tr>
<tr>
<td>ProcObject</td>
<td>CustomerOrder</td>
</tr>
</tbody>
</table>

Some of the information presented in these examples provides a review of concepts introduced and described in previous chapters of this manual. Note that some SOAP messages and code samples are edited for brevity.

Client references to the sample VB.NET interface include the following method declarations for these interface objects:

- Interface AppObject, OrderInfo:

**VB.NET method declarations on AppObject, OrderInfo**

```vbnet
Public Sub Connect_OrderInfo(
    ByVal userId As String,
    ByVal password As String,
    ByVal appServerInfo As String)
Public Function FindCustomerByNum(
    ByVal CustomerNumber As Integer,
    ByRef CustomerName As String) As String
Public Function CreatePO_CustomerOrder(
    ByVal custNum As Integer) As String
Public Sub Release_OrderInfo( )
```

The `FindCustomerByNum` and `CreatePO_CustomerOrder` methods are represented as functions because methods in Doc/Lit WSDLs are always defined to return a value, regardless of whether the procedure defined in ProxyGen is specified to return the ABL RETURN-VALUE. If `Return ABL RETURN-VALUE` is not specified, the return value for the function will be a null String. For more information, see *OpenEdge Development: Open Client Introduction and Programming*. 
• Interface ProcObject, CustomerOrder:

**VB.NET method declarations on ProcObject, CustomerOrder**

```vbnet
Public Function GetOrderDetails(    
    ByRef OrderDetails() As OrderDetailsRow) As String
Public Function GetTotalOrdersByNumber(    
    ByVal Threshold As Decimal) As Integer
Public Sub Release_CustomerOrder( )
```

**Note:** ProxyGen normally configures a `Connect_AppObject()` method with three parameters (string `userid`, string `password`, and string `appServerInfo`). Enabling the **Return ABL RETURN-VALUE on Connect** option adds the output parameter `string result`, which contains the return value of the Connection procedure.
Consuming a session-managed Web service

To access a session-managed Web service, you maintain connection context by passing object ID’s associated with each request.

To begin using a session-managed Web service:

1. Instantiate (create) the AppObject as appropriate for the client platform.

2. Connect to an AppServer by calling the connect method on the AppObject before calling any other Web Service (AppObject) method.

3. Obtain the AppObject ID value from the SOAP response header for the connect method and use it for all subsequent calls to methods on the session-managed AppObject.

4. Invoke any available methods on the AppObject, as required.

5. Ensure that the last method you invoke on the AppObject is the object’s release method. For more information, see the “Releasing an object” section on page 4–14.

As with other Open Clients, there is no ABL involved in implementing the connect method on an AppObject. For Web services, however, the operation is an object method that is required by the WSA.

This is a VB.NET declaration for an AppObject connect method, Connect_OrderInfo():

VB.NET prototype for an AppObject Connect_OrderInfo method

```vbnet
Public Sub Connect_OrderInfo(
    ByVal userId As String,
    ByVal password As String,
    ByVal appServerInfo As String)
```

**Note:** For more information on the parameters to this method, see the sections on connecting to an AppServer in *OpenEdge Development: Open Client Introduction and Programming*. Note that there is no AppServer URL parameter. For a Web service (unlike other Open Client applications), the deployer manages the AppServer connection information for each Web service through the WSA instance where the Web service is deployed. Each deployed Web service has writable properties to specify this information. For more information, see the sections on Web service deployment in *OpenEdge Application Server: Administration*.

When the client executes the connect method, the SOAP response message contains a SOAP header with the AppObject ID. You must send this AppObject ID in the SOAP header of the request message for every subsequent method call on the AppObject. How you handle the SOAP header and AppObject ID depends on your client type. For some clients, such as .NET, this process is automated. For other clients, such as ABL and Java, you need to create the code to handle this yourself. For more information on handling SOAP headers and object IDs, see Chapter 9, “Handling SOAP Message Headers in ABL” and Appendix B, “Developing a Java Client to Consume OpenEdge Web Services.”
This is a sample instantiation and invocation of the connect method on the OrderInfo AppObject:

**VB.NET client code for the connect method on AppObject, OrderInfo**

```vbnet
webService = New OrderSvc.OrderInfoObj()
webService.Connect_OrderInfo("", "", "")
```

This is a sample Doc/Lit SOAP request message that might be generated from invoking the `Connect_OrderInfo()` method, as in the example:

**SOAP connect request for AppObject, OrderInfo**

```xml
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:urn="urn:OrderSvc:OrderInfo">
  <soap:Body>
    <Connect_OrderInfo xmlns="urn:OrderSvc:OrderInfo">
      <userId />
      <password />
      <appServerInfo />
    </Connect_OrderInfo>
  </soap:Body>
</soap:Envelope>
```

Note the data for the request highlighted in the example for parameters passed as empty strings.

This is a sample Doc/Lit SOAP response that might be generated by the WSA as a response to the `Connect_OrderInfo()` method:

**SOAP response to OrderInfo AppObject connect request**

```xml
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:urn="urn:OrderSvc:OrderInfo">
  <soap:Header>
    <OrderInfoID xmlns="urn:OrderSvc:OrderInfo">
      <UUID>2e62cab6b81150d5:167f64e:f295e997b0:-8000;</n      <OrderInfo|PX-000001|AO>;M/IryPm3p1DcF/W5DsH4GA==</UUID>
    </OrderInfoID>
  </soap:Header>
  <soap:Body>
    <Connect_OrderInfoResponse xmlns="urn:OrderSvc:OrderInfo" />
  </soap:Body>
</soap:Envelope>
```

Note the value for the returned AppObject ID, OrderInfo, highlighted in the example. Thus, the SOAP response header returns the following AppObject ID contained in the `OrderInfoID` element:

```
2e62cab6b81150d5:167f64e:f295e997b0:-8000;<OrderInfo|PX-000001|AO>;M/IryPm3p1DcF/W5DsH4GA==
```
Consuming a session-free Web service

Because the AppObject for a session-free Web service maintains no connection to the Web service, you can call the methods of a session-free AppObject with no preparation. When the WSA instance receives such a method request, it simply locates an available AppServer connection in the connection pool for that Web service and invokes the corresponding procedure or function on the available AppServer.

**Note:** All AppServers that participate in the connection pool for a session-free Web service are assumed to share the same version and capabilities and have access to the same set of databases and other shareable resources on a network. For more information, see *OpenEdge Application Server: Developing AppServer Applications*.

To use a session-free Web service:

1. Instantiate (create) the AppObject as appropriate for the client platform. For example:

   **VB.NET client code to instantiate session-free AppObject, OrderInfo**

   ```
   webService = New OrderSvc.OrderInfoObj()
   ```

2. Invoke any available methods on the AppObject, as required.
Running an ABL non-persistent procedure

Methods that run a non-persistent procedure can:

- Appear on any AppObject
- Require an object ID in the SOAP request header for each method invocation unless the object is a session-free AppObject

This is the ABL prototype for the sample `FindCustomerByNum()` method:

**ABL prototype for a non-persistent external procedure**

```abl
/* FindCustomerByNum.p */
DEFINE INPUT PARAMETER CustomerNumber AS INTEGER.
DEFINE OUTPUT PARAMETER CustomerName AS CHARACTER.
```

This is a sample VB.NET declaration for the ABL non-persistent procedure method, `FindCustomerByNum()`:

**VB.NET prototype for a method that runs an ABL non-persistent procedure**

```vbnet
Public Function FindCustomerByNum(
    ByVal CustomerNumber As Integer,
    ByRef CustomerName As String) As String
```

This is a sample call to this method:

**VB.NET client code calling an external procedure method**

```vbnet
Dim CustomerName As String
Dim retVal As String
retVal=webService.FindCustomerByNum(3, CustomerName)
```

This is a sample Doc/Lit SOAP request that might be generated from this `FindCustomerByNum()` method invocation:

**SOAP request from calling the FindCustomerByNum procedure method**

```xml
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns="urn:OrderSvc:OrderInfo"
xmlns:OrderInfo="urn:OrderSvc:OrderInfo">
<soap:Header>
<OrderInfoID xmlns="urn:OrderSvc:OrderInfo">
<UUID>2e62cab6b81150d5:167f64e:f295e997b0:-8000;PX-000001|AO>;M/IryPm3p1DcF/W5DsH4GA==</UUID>
</OrderInfoID>
</soap:Header>
<soap:Body>
<FindCustomerByNum xmlns="urn:OrderSvc:OrderInfo">
<CustomerNumber>3</CustomerNumber>
</FindCustomerByNum>
</soap:Body>
</soap:Envelope>
```
Note the following elements in the preceding SOAP message:

- AppObject ID (OrderInfoID) sent using the <UUID> element of the SOAP header
- Data for the request highlighted in the example, including the request for a CustomerNumber value of 3

This is a sample Doc/Lit SOAP response that is generated by the WSA from this invocation of the FindCustomerByNum( ) method:

**SOAP response from calling the FindCustomerByNum method**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope namespaces defined here...>
    <soap:Body>
        <FindCustomerByNumResponse xmlns="urn:OrderSvc:OrderInfo">
            <result xsi:nil="true"/>
            <CustomerName>Hoops</CustomerName>
        </FindCustomerByNumResponse>
    </soap:Body>
</soap:Envelope>
```

**Note:** The CustomerName value, "Hoops", returned and highlighted in the example.
Creating and using a ProcObject

Each AppObject that defines ProcObjects also defines a built-in class factory method specifically for each ProcObject. This method runs a persistent procedure on the AppServer that thereby establishes a persistent AppServer session context that is encapsulated by the ProcObject for access by the client.

To create and use a ProcObject:

1. Instantiate (create) the ProcObject as appropriate for the client platform.

2. Call the class factory method for this ProcObject on the parent AppObject or SubAppObject to run the persistent procedure on the AppServer.

3. Obtain the ProcObject ID value from the response header of the class factory method. Use this ProcObject ID for all subsequent calls to methods on the ProcObject.


5. Ensure that the last method you invoke on the ProcObject is the object’s release method. For more information, see the “Releasing an object” section on page 4–14.

Note: You can instantiate a ProcObject either before or after calling the class factory method and getting the ProcObject ID, but you must complete all of these steps before you invoke methods on the ProcObject.

ProcObject session context

For a session-managed ProcObject, the method establishes the persistent session context using the same AppServer connection as the parent AppObject. For a session-free ProcObject, the method establishes the persistent session context on any AppServer that the WSA makes available through the Web service connection pool, and the client connection to that context persists until the ProcObject is released. Note that for multiple ProcObjects defined for the same session-free Web service, no two objects ever share the same session context. While for a session-managed Web service, all defined ProcObjects always share the same session context.

ProcObject IDs

You use the same type of class factory method to create ProcObjects for both session-managed and session-free Web Services. For a ProcObject whose defining parent is a SubAppObject or a session-managed AppObject, you must send the object ID of the parent in the SOAP request message header. However, for a ProcObject whose defining parent is a session-free AppObject, you do not have an AppObject ID to send in the SOAP request header when you invoke the class factory method to create the ProcObject.

For every ProcObject, regardless of its parent, the SOAP response header for the class factory method returns a ProcObject ID that associates the ProcObject session context with the client. You must send this ProcObject ID in the SOAP request header for all other method calls on the ProcObject.
ProcObject class factory methods

This is the ABL prototype for the persistent procedure that runs for the sample ProcObject class factory method, CreatePO_CustomerOrder( ):

ABL prototype for a persistent procedure to implement a ProcObject

```abl
/* CustomerOrder.p */
DEFINE INPUT PARAMETER custNum AS INTEGER.
```

**Note:** The parameter list for the persistent procedure that runs for the ProcObject class factory method is AppServer application dependent, and is the basis for creating the parameter list of the ProcObject class factory method. A persistent procedure can also be specified in ProxyGen to return a string value using the ABL RETURN statement.

This is a VB.NET declaration for the ProcObject class factory method, CreatePO_CustomerOrder( ):

**VB.NET prototype for the ProcObject CreatePO_CustomerOrder method**

```vbnet
Public Function CreatePO_CustomerOrder(ByVal custNum As Integer) As String
```

**Note:** This method maps to a persistent procedure that has been specified to return the string from the ABL RETURN-VALUE function.

The following code snippet:

1. Instantiates the ProcObject on the client, enabling access to its methods
2. Calls the CreatePO_CustomerOrder( ) method on the AppObject, `webService`, to create the ProcObject, `CustomerOrder`, on the WSA and run `CustomerOrder.p` persistently
3. Copies the ProcObject ID to the ProcObject from the AppObject (`webService`) that creates the ProcObject

**VB.NET client code to create the ProcObject, CustomerOrder**

```vbnet
custOrder = New OrderSvc.CustomerOrderObj( )
custName = webService.CreatePO_CustomerOrder(3)
custOrder.CustomerOrderIDValue = webService.CustomerOrderIDValue
```
Creating and using a ProcObject

This is a Doc/Lit SOAP request generated by invoking the CreatePO_CustomerOrder() method to create the ProcObject, passing in a custNum value of 3:

SOAP request to create ProcObject, CustomerOrder

```xml
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:urn="urn:OrderSvc:OrderInfo">
  <soap:Header>
    <OrderInfoID xmlns="urn:OrderSvc:OrderInfo">
      <UUID>2e62cab6b81150d5:167f64e:f295e997b0:-8000;</UUID>
    </OrderInfoID>
  </soap:Header>
  <soap:Body>
    <CreatePO_CustomerOrder xmlns="urn:OrderSvc:OrderInfo">
      <custNum>3</custNum>
    </CreatePO_CustomerOrder>
  </soap:Body>
</soap:Envelope>
```

Note the value for the request highlighted in the example, especially the AppObject ID sent for the AppObject, OrderInfo, which is the parent of the ProcObject being created for customer number 3.

This is a sample Doc/Lit SOAP response that is generated by the WSA from this invocation of the CreatePO_CustomerOrder() method:

SOAP response from creating ProcObject, CustomerOrder

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope xmlns:urn="urn:OrderSvc:CustomerOrder">
  <soap:Header>
    <CustomerOrderID xmlns="urn:OrderSvc:CustomerOrder">
      <UUID>2e62cab6b81150d5:167f64e:f295e997b0:-8000;</UUID>
    </CustomerOrderID>
  </soap:Header>
  <soap:Body>
    <CreatePO_CustomerOrderResponse xmlns="urn:OrderSvc:OrderInfo">
      <result>Hoops</result>
    </CreatePO_CustomerOrderResponse>
  </soap:Body>
</soap:Envelope>
```

Note the value returned for the CustomerOrder ProcObject ID highlighted in the example. The Web service returns this ProcObject ID even if it is session free, to allow the ProcObject to access its own AppServer session context.

Thus, the SOAP response header returns the following ProcObject ID contained in the CustomerOrderID element:

```
2e62cab6b81150d5:167f64e:f295e997b0:-8000;<OrderInfo|PX-000002|PO>;G1Vc/vmohvLnxwQXwa6Cg==
```

Finally, note the ABL RETURN-VALUE result, returned from running the persistent procedure, which returns the customer name, "Hoops".
Running an internal procedure or user-defined function

Methods to run an ABL internal procedure and user-defined function of a persistent procedure (ProcObject) are indistinguishable in client code. These methods:

- Run only in the context of a defining ProcObject
- Require a ProcObject ID in the SOAP request header, which is the object ID of the defining ProcObject

This is the ABL prototype for the sample user-defined function, GetTotalOrdersByNumber( ):

```
ABL prototype for a user-defined function

/* CustomerOrder.p */
FUNCTION GetTotalOrdersByNumber RETURNS INTEGER
(Threshold AS DECIMAL):
```

This is a VB.NET declaration for the ABL user-defined function method, GetTotalOrdersByNumber( ):

```
VB.NET prototype for an ABL user-defined function method

Public Function GetTotalOrdersByNumber
(ByVal Threshold As Decimal) As Integer
```

The following is a sample method call for the user-defined function method, GetTotalOrdersByNumber, which is an interface method on the sample ProcObject, CustomerOrder:

```
VB.NET client code of method to run an ABL user-defined function

totNumber = custOrder.GetTotalOrdersByNumber(2150.99)
```

Note that user-defined function methods return a value whose data type maps to the ABL data type of the user-defined function’s return value.

This is a sample Doc/Lit SOAP request that might be generated from invoking the GetTotalOrdersByNumber( ) method to execute the ABL user-defined function, passing in a Threshold order value of 2150.99:

```
SOAP request for user-defined function method, GetTotalOrdersByNumber

<?xml version="1.0" encoding="utf-8" ?>
  <soap:Header>
    <CustomerOrderID xmlns="urn:OrderSvc:CustomerOrder">
      <UUID>2e62cab6b81150d5:167f64e:f295e997b0:-8000;
      <OrderInfo|PX-000002|PO>;G1Vc/vmohvLnwxQQXwA6Cg==</UUID>
    </CustomerOrderID>
  </soap:Header>
  <soap:Body>
    <GetTotalOrdersByNumber xmlns="OrderSvc:CustomerOrder">
      <Threshold>2150.99</Threshold>
    </GetTotalOrdersByNumber>
  </soap:Body>
</soap:Envelope>
```
Note the object ID for the ProcObject, CustomerOrder, sent to make the request on a method of the ProcObject.

This is the SOAP response returning a function value of 5, which is the total number of orders that satisfy the specified order Threshold value:

**SOAP response from calling the GetTotalOrdersByNumber method**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope xmlns:urn="urn:OrderSvc:CustomerOrder">
  <soap:Body>
    <GetTotalOrdersByNumberResponse>
      <result>5</result>
    </GetTotalOrdersByNumberResponse>
  </soap:Body>
</soap:Envelope>
```
Releasing an object

When you are finished using any Web service object (except a session-free AppObject), you must invoke the built-in release method to release that object and return its resources for use by other clients.

As with other Open Clients, there is no ABL involved in implementing the release method on an Open Client object. However, for Web services, the operation is an object method is required by the WSA.

This is a VB.NET declaration for the AppObject release methods:

**VB.NET prototype for a Release_Object method**

```vbnet
Public Sub Release_OrderInfo()  
Public Sub Release_CustomerOrder()
```

Although these methods take no parameters, you must remember to send the object ID for the object you are releasing in the SOAP header of the release request message. How you handle the SOAP header and object ID depends on your client type. For some clients, such as .NET, this process is automated. For other clients, such as ABL and Java, you need to create the code to handle this yourself. For more information on handling SOAP headers and object IDs, see Chapter 9, “Handling SOAP Message Headers in ABL” and Appendix B, “Developing a Java Client to Consume OpenEdge Web Services.”

These are sample calls to the release methods declared for the sample Web service:

**VB.NET client code calling methods to release objects**

```vbnet
custOrder.Release_CustomerOrder()  
webService.Release_OrderInfo()
```

**Note:** AppServer connections used by session-free methods are usually returned to the Web service connection pool after the SOAP response message for the release method is returned.
Passing static and dynamic temp-table parameters

The examples in the following sections show how parameters are passed for static and dynamic temp-tables.

Invoking a method with a TABLE parameter

To invoke a method that passes a TABLE parameter, the client application typically must:

- Create rows using the interface object that represents the TABLE row
- Insert the rows in an array that represents the TABLE parameter

This is the ABL prototype for a sample method, staticTT(), that passes a TABLE parameter:

**ABL prototype that passes a TABLE parameter**

```abl
/* staticTT.p */
DEFINE INPUT-OUTPUT PARAMETER TABLE FOR ttEmp.
```

This is the declaration for a VB.NET client interface method, staticTT(), which has a TABLE parameter, ttEmp, passed as the class, staticTT_ttEmpRow:

**VB.NET declaration of interface method passing static TEMP-TABLE, ttEmp**

```vbnet
Public Class staticTT_ttEmpRow
    Public Name As String
    Public Number As Integer
End Class

Public Sub staticTT(ByRef ttEmp() As staticTT_ttEmpRow)
```

The following client code defines a two row array (myTempTable) according to a defined schema (staticTT_ttEmpRow), then defines and creates two rows assigned with values. It then initializes the array (as TABLE, ttEmp) with the two rows and passes it to the staticTT() interface method, which passes the TABLE to the Web service:

**VB.NET client code passing TABLE, ttEmp, to an interface method**

```vbnet
Dim myTempTable(1) as webService.staticTT_ttEmpRow
Dim myRow1 as webService.staticTT_ttEmpRow = New webService.staticTT_ttEmpRow()
Dim myRow2 as webService.staticTT_ttEmpRow = New webService.staticTT_ttEmpRow()

myRow1.Name = "Fred"
myRow1.Number = 1
myRow2.Name = "Barney"
myRow2.Number = 2

myTempTable(0) = myRow1
myTempTable(1) = myRow2

webService.staticTT(myTempTable)
```
The following is a Doc/Lit SOAP message that this call to `staticTT()` might send:

**Doc/Lit SOAP request to pass a TABLE parameter**

```xml
<?xml version="1.0" encoding="utf-8" ?>
  <soap:Header>
    <EmployeeID xmlns="urn:EmployeeSrvc:Employee">2e62cab6b81150d5:-1380e8e:f27fb934f4:-8000;<Employee|PX-000004|AO>;5REiR9inxXQi4s6ghRwkfg==</EmployeeID>
  </soap:Header>
  <soap:Body>
    <staticTT xmlns="urn:EmployeeSrvc:Employee">
      <ttEmp>
        <ttEmpRow>
          <Name>Fred</Name>
          <Number>1</Number>
        </ttEmpRow>
        <ttEmpRow>
          <Name>Barney</Name>
          <Number>2</Number>
        </ttEmpRow>
      </ttEmp>
    </staticTT>
  </soap:Body>
</soap:Envelope>
```

**Note:** The `staticTT()` method must send the object ID for the Employee AppObject in the SOAP request header because `staticTT()` is a method on the session-managed AppObject.

As a point of comparison, the following is a RPC/Encoded SOAP message that this call to `staticTT()` might send:

**RPC/Encoded SOAP request to pass a TABLE parameter**

```xml
<?xml version="1.0" encoding="utf-8" ?>
  <soap:Header>
    <q1:EmployeeID id="h_id1" xmlns:q1="urn:EmployeeSrvc:Employee">2e62cab6b81150d5:-1380e8e:f27fb934f4:-8000;<Employee|PX-000004|AO>;5REiR9inxXQi4s6ghRwkfg==</q1:EmployeeID>
  </soap:Header>
  <soap:Body>
    <q1:staticTT xmlns="urn:EmployeeSrvc:Employee">
      <ttEmp href="#id1"/>
    </q1:staticTT>
    <soapenc:Array id="id1" xmlns:q2="urn:EmployeeSrvc:Employee" soapenc:arrayType="q2:staticTT_ttEmpRow[2]">
      <Item href="#id2"/>
      <Item href="#id3"/>
    </soapenc:Array>
    <q3:staticTT_ttEmpRow id="id2" xmlns:q3="urn:EmployeeSrvc:Employee">
      <Name xsi:type="xsd:string">Fred</Name>
      <Number xsi:type="xsd:int">1</Number>
    </q3:staticTT_ttEmpRow>
  </soap:Body>
</soap:Envelope>
```
Passing static and dynamic temp-table parameters

**Note:** The staticTT( ) method must send the object ID for the Employee AppObject in the SOAP request header because staticTT( ) is a method on the session-managed AppObject.

**Invoking a method with a TABLE-HANDLE parameter**

To invoke a method that passes a TABLE-HANDLE parameter:

- The client application must create and send an XML Schema along with the data to fully describe the dynamic temp-table in the SOAP request message. The TABLE-HANDLE parameter in a SOAP request or response consists of an XML `<DataSet>` element containing two child elements:
  - An XML Schema element, `<schema>`, representing the schema for the TABLE-HANDLE
  - An XML representation of data using an element, `<Data>`, with each row represented as a sub-element, `<Item>`

- A non-ABL client application must parse the XML Schema and data from the SOAP response message to make the TABLE-HANDLE accessible as native data within the application.

This is the ABL prototype for a sample method, dynttIO( ), that passes a TABLE-HANDLE parameter:

**ABL prototype that passes a TABLE-HANDLE parameter**

```ABL
/* dynttIO.p */
DEFINE INPUT-OUTPUT PARAMETER TABLE-HANDLE ttHandle.
```

This is the declaration for a VB.NET client interface method (dynttIO( )) which has a TABLE-HANDLE parameter, ttHandle, as a VB.NET object:

**VB.NET declaration of interface method passing TABLE-HANDLE, ttHandle**

```vbnet
Public Sub dynttIO(ByVal ttHandle As Object)
```
The following VB.NET client code passes the dyntt object representing a TABLE-HANDLE to the dynttIO( ) method:

**VB.NET client code passing TABLE-HANDLE, ttHandle, to interface method**

```vbnet
Dim dyntt as Object
'... Code to build up the dyntt Object (XML Schema and data)
webService.dynttIO(dyntt)
```

For more information on how you might manage TABLE-HANDLE parameters in VB.NET, see the subsections on handling TABLE-HANDLE parameters in Appendix A, “Developing a .NET Client to Consume OpenEdge Web Services,” and in Java see the subsections on handling TABLE-HANDLE parameters in Appendix B, “Developing a Java Client to Consume OpenEdge Web Services.”

This is the structure of the Doc/Lit SOAP request message that the sample dynttIO method sends to pass a dynamic temp-table that you create for the ttHandle TABLE-HANDLE parameter:

**Doc/Lit SOAP request for TABLE-HANDLE, ttHandle**

```xml
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope namespaces defined here...>
<soap:Header .../>
<soap:Body>
<dynttIO xmlns="urn:DynTTSrvc:DynTT">
<ttHandle>
<DataSet xmlns="">
<!-- Schema definition goes here -->
</schema>
<!-- Data goes here -->
</Data>
</DataSet>
</ttHandle>
</dynttIO>
</soap:Body>
</soap:Envelope>
```

The ttHandle TABLE-HANDLE parameter becomes an XML <ttHandle> element containing the <DataSet> element that contains the schema and the data.

**Note:** Not shown is any required object ID that must be sent in the SOAP header for the object on which dynttIO( ) is invoked.
This is a sample XML Schema created by VB.NET for the TABLE-HANDLE contained by the sample <DataSet> element in the sample Doc/Lit SOAP request:

**TABLE-HANDLE <schema> element for an Doc/Lit SOAP request**

```xml
<xs:element name="Data" msdata:IsDataSet="true">
    <xs:complexType>
        <xs:choice maxOccurs="unbounded">
            <xs:element name="Item">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="Name" type="xs:string" minOccurs="0"/>
                        <xs:element name="Number" type="xs:int" minOccurs="0"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:choice>
    </xs:complexType>
</xs:element>
</xs:schema>
```

Note the definition of the <data> element containing the data for the table, and how the column type information is specified within the <Item> element.

This is a sample <Data> element you would create to accompany the specified schema in the sample Doc/Lit SOAP request, including the column values for the two rows initialized in the sample VB.NET code:

**TABLE-HANDLE <Data> element for an Doc/Lit SOAP request**

```xml
<Data>
    <Item>
        <Name>Fred</Name>
        <Number>1</Number>
    </Item>
    <Item>
        <Name>Barney</Name>
        <Number>2</Number>
    </Item>
</Data>
```
Receiving a SOAP fault message

The following sample SOAP fault shows the response to a request sent for an object (OrderInfo) of a disabled Web service:

Sample SOAP fault for disabled Web service

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope namespaces defined here…>
  <soap:Body>
    <soap:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
      <faultstring>An error was detected ... request. (10893) </faultstring>
      <detail>
        <FaultDetail ... >
          <errorMessage>The urn:OrderSvc:OrderInfo service is unavailable (10921)</errorMessage>
          <requestID> 2e62cab6b81150d5:-17c6a3c:f1ddfbd11b:-8000#1f
        </requestID>
        </FaultDetail>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Caution: When exiting the program in response to a SOAP fault, you must release all Web service objects you created in the program.

Note: SOAP fault messages conform to SOAP 1.1.

The contents of the highlighted SOAP elements include some of the key components of the fault message, containing such error information as a general error code (<faultcode>) and message (<faultstring>). These elements are followed by more detailed information, including the message indicating a more precise cause for the error (<errorMessage>), and a unique identifier for the request that returned the error (<requestID>). The contents of the <detail> element is defined in the WSDL Types section. For more information on handling errors, including SOAP fault messages generated for WSA-managed Web services, see Chapter 5, “Testing and Debugging OpenEdge Web Services.”
Passing static and dynamic ProDataSet parameters

The examples in the following sections show how parameters are passed for static and dynamic ProDataSets.

These examples rely on the sample Web service specifications shown in Table 4–2.

Table 4–2: ProDataSet to .NET sample Web service specifications

<table>
<thead>
<tr>
<th>Property or component</th>
<th>Value or name</th>
<th>Object type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web service</td>
<td>CustOrders</td>
<td>–</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://servicehost:80/wsa/wsa1">http://servicehost:80/wsa/wsa1</a></td>
<td>–</td>
</tr>
<tr>
<td>Session model</td>
<td>Session-Free</td>
<td>–</td>
</tr>
<tr>
<td>SOAP format</td>
<td>Doc/Lit</td>
<td>–</td>
</tr>
<tr>
<td>TargetNamespace</td>
<td>urn:CustOrders</td>
<td>–</td>
</tr>
<tr>
<td>WSDL objects</td>
<td>CustOrdersObj</td>
<td>AppObject</td>
</tr>
<tr>
<td></td>
<td>dsCustOrd</td>
<td>ProDataSet dsCustOrd</td>
</tr>
</tbody>
</table>

Note: This section does not continue the previous examples. It uses C#.NET, rather than VB.NET like most of this chapter.

Invoking a method with a DATASET parameter

To invoke a method that passes a DATASET parameter, the client application typically must:

- Create rows using the interface object that represents the TABLE row
- Insert the rows in arrays that represent the constituent temp-tables in the ProDataSet

For example, the following snippet of code passes a ProDataSet parameter represents and operation in the Web service:

getCustOrders.p

```c#
/* 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.
```
When you add a Web Reference for the CustOrders Web service to Microsoft® Visual Studio, it creates the following proxies in a References file for the getCustOrders operation:

**Sample C#.NET proxy code for getCustOrders**

```csharp
public string getCustOrders(System.Nullable<int> iCustNum, out dsCustOrd dsCustOrd) {
    object[] results = this.Invoke("getCustOrders", new object[] {
        iCustNum
    });
    dsCustOrd = ((dsCustOrd)(results[1]));
    return ((string)(results[0]));
}
```

```csharp
public partial class dsCustOrd {
    private dsCustOrdTtCust[] ttCustField;
    private dsCustOrdTtOrder[] ttOrderField;
    ...
}
```

```csharp
public partial class dsCustOrdTtCust {
    private System.Nullable<int> custNumField;
    private string nameField;
    ...
}
```

```csharp
public partial class dsCustOrdTtOrder {
    private System.Nullable<int> orderNumField;
    private System.Nullable<int> custNumField;
    <!-- nested data relation with ttOrderLine -->
    private dsCustOrdTtOrderTtOrderLine[] ttOrderLineField;
    ...
}
```
When you reference the Web service in your code, Microsoft Visual Studio offers these proxy objects as appropriate. You can then create code to access the ProDataSet parameter like the following:

**Sample C#.NET application for getCustOrders operation**

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace getCustOrders
{
    class Program
    {
        static void Main(string[] args)
        {
            int iCustNum;
            wsCustOrders.CustOrdersService ws = new wsCustOrders.CustOrdersService();
            wsCustOrders.dsCustOrd dsCustOrd;
            iCustNum = 1;
            ws.getCustOrders(iCustNum, out dsCustOrd);
            FileStream fs = new FileStream("getCustOrder.out", FileMode.Create);
            StreamWriter w = new StreamWriter(fs);

            //temp-table ttCust
            for (int i = 0; i < dsCustOrd.ttCust.Length; i++)
            {
                w.Write(dsCustOrd.ttCust[i].CustNum);
                w.Write(dsCustOrd.ttCust[i].Name);
                w.WriteLine();
            }

            //temp-table ttOrder
            for (int i = 0; i < dsCustOrd.ttOrder.Length; i++)
            {
                w.Write(dsCustOrd.ttOrder[i].CustNum);
                w.Write(dsCustOrd.ttOrder[i].OrderNum);
                w.WriteLine();
            }
            //nested temp-table ttOrderLine
            for (int j = 0; j < dsCustOrd.ttOrder[i].ttOrderLine.Length; j++)
            {
                w.Write(dsCustOrd.ttOrder[i].ttOrderLine[j].LineNum);
                w.Write(dsCustOrd.ttOrder[i].ttOrderLine[j].OrderNum);
            }
        }
        w.Close();
    }
}
```
Invoking a method with a DATASET-HANDLE parameter

To invoke a method that passes a DATASET-HANDLE parameter:

- The client application must create and send an XML Schema along with the data to fully describe the dynamic temp-table in the SOAP request message. The DATASET-HANDLE parameter in a SOAP request or response consists of an XML element containing two child elements:
  - An XML schema element representing the schema for the DATASET-HANDLE
  - An XML representation of data, using a data element with child elements representing individual rows for each constituent temp-table

- The client application must parse the XML Schema and data from the SOAP response message to make the DATASET-HANDLE accessible as native data within the application.

For example, the following snippet of code is from a procedure that passes a dynamic ProDataSet parameter:

```/* getDynDs.p */
DEFINE INPUT PARAMETER criteria as CHARACTER.
DEFINE OUTPUT PARAMETER DATASET-HANDLE hDset.

/* Create dataset based on criteria
** fill dataset and return to caller */
...
```

When you add a Web Reference for the Web service to Microsoft® Visual Studio, it creates the following proxies in a References file for the getDynDs operation:

**Sample C#.NET proxy code for getDynDs**

```public string getDynDs(
    string criteria,
    out DataSetHandleParam hDset) {
    object[] results = this.Invoke("getDynDs", new object[] {
        criteria});
    hDset = ((DataSetHandleParam)(results[1]));
    return ((string)(results[0]));
}
...
```

```public partial class DataSetHandleParam {
    private System.Xml.XmlElement[] anyField;
    ...
}```
When you reference the Web service in your code, Microsoft Visual Studio offers these proxy objects as appropriate. You can then create code to access the ProDataSet parameter like the following:

**Sample C#.NET code for invoking getDynDs**

```csharp
sampleDynDS.sampleDynDSService mySvc = new sampleDynDS.sampleDynDSService();
sampleDynDS.DataSetHandleParam dsh;
mySvc.getDynDs(textBox1.Text, out dsh);
```
Testing and Debugging OpenEdge Web Services

This chapter explains how to test and debug OpenEdge Web services. It assumes you are familiar with Web services and the Web Services tools in OpenEdge, as described in the preceding chapters of this manual. Information in this chapter is useful for both Web service and client developers to troubleshoot OpenEdge Web services and client applications that access them.

This chapter contains the following sections:

• Testing the AppServer application as a Web service
• Setting error information levels
• Identifying relevant log files
• Working with SOAP faults
• Using a SOAP message viewer
• Identifying errors
• Sample error scenarios
Testing the AppServer application as a Web service

You can generally use two different approaches for testing an AppServer application written to support an OpenEdge Web service:

- **Unit testing from ABL** — From ABL, you can access AppServer functionality directly or as an OpenEdge Web service.

- **Testing from a Web service client** — From a Web service client using any platform, including ABL, you can thoroughly test the Web service as it might behave in the environment in which you will deploy it.

The approach that you take depends on your preferences, needs, and experience.

**Unit testing from ABL**

Accessing the AppServer directly is useful for doing unit testing without having to bother with the overhead and inconvenience of managing the Web service deployment environment. This form of testing also eliminates some of the ABL logic required to access a Web service. However, it can provide a good measure of confidence in the basic integrity of the application service itself.

**Testing from a Web services client**

This method allows you to create a client application environment and functionality exactly as you expect real Web service clients to be created, and you can test the application end-to-end using a complete Web service testing infrastructure.

The rapid prototyping capability of ABL can also make it a convenient platform to test the application service as a Web service, depending on the requirements of the Web service. Some client platforms might handle certain types of Web service interaction more easily than others. However, especially where ABL most easily handles a particular Web service application feature, modifying an existing ABL unit test bed to access the AppServer as a Web service is a natural next step in testing Web service functionality.

Otherwise, especially if you expect your primary client base to use another platform, you might want to test on that platform to fully understand what users of that platform need to do in order to use the Web service that you plan to deploy.

**To test from a Web services client:**

1. Configure the AppServer to support the required application session model:
   - For information on configuring the session-managed operating modes, see *OpenEdge Application Server: Developing AppServer Applications* and the OpenEdge Explorer and Progress Explorer online help.
   - For information on configuring the state-free operating mode (session-free model) see the Progress Explorer online help.

2. Define the Web service and generate the Web service mapping (WSM) file using ProxyGen. For more information, see *OpenEdge Development: Open Client Introduction and Programming.*
3. Generate a test WSDL file from ProxyGen when you generate the WSM file, or deploy the WSM file to a WSA instance to obtain a deployed WSDL file. For more information on deploying a Web service, see the sections on deploying and managing Web services in *OpenEdge Application Server: Administration*.

4. Write the test client application, using the WSDL file to help generate client interfaces or to manually write the code to invoke Web service requests. For more information, see Part II, “Creating OpenEdge Web Services.” For information on building an ABL client to access your Web service, see Part III, “Creating ABL Clients to Consume Web Services.”

5. Once you have an executable test client, ensure that the AppServer and WSA are running, and the WSA instance where you deployed the Web service is enabled to allow client access to the Web service. For more information on running and enabling the WSA for Web service client access, see the sections on working with the WSA and deploying Web services in *OpenEdge Application Server: Administration*.

6. Begin testing the client.
Setting error information levels

To make it easier to diagnose errors, you can control the amount of information provided for the following sources of Web service error information:

- WSA log file
- Web service SOAP fault response messages
- AdminService log file
- AppServer log file
- NameServer log file

For information on managing logging for the AdminService, NameServer, and in general for all OpenEdge servers managed by the Progress Explorer Framework, see *OpenEdge Getting Started: Installation and Configuration*. For more information on managing logging for an AppServer, see *OpenEdge Application Server: Administration*. The following sections describe how to control logging and SOAP fault information output for the WSA and Web services.

WSA and Web service logging level properties

Table 5–1 shows the property files where these settings occur and the default installation settings for basic properties that affect output for each of these information sources.

<table>
<thead>
<tr>
<th>Affected error output</th>
<th>Property files</th>
<th>Default property setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSA log file</td>
<td>ubroker.properties (WSA properties)</td>
<td>loggingLevel=2</td>
</tr>
<tr>
<td></td>
<td>*.props (Web service properties)</td>
<td>serviceLoggingLevel=2</td>
</tr>
<tr>
<td>SOAP fault response messages</td>
<td>*.props (Web service properties)</td>
<td>serviceFaultLevel=2</td>
</tr>
</tbody>
</table>

**Note:** For complete information on the log file settings for the WSA log file, including more advanced logging properties, see the Progress Explorer online help and the unbroker.properties.README file in the properties directory under the OpenEdge installation directory.
Logging information

The properties in Table 5–1 have the following functions and value ranges (the greater the value, the more information is provided):

- **loggingLevel** — Affects information logged for a single WSA instance. Set this property to change the logging detail for WSA administration, WSDL document retrieval, user authorization, and HTTP request handling. Valid values range between 1 and 4.

- **serviceLoggingLevel** — Affects information logged for a single deployed Web service. Set this property to change the logging detail for the processing of a Web service client request. Valid values range between 1 and 4.

- **serviceFaultLevel** — Affects the information returned in all SOAP faults from a single deployed Web service. Set this property to change the level of detail returned for client exceptions. Valid values are 2 or 3.

Hints for setting the WSA loggingLevel property

Set the `loggingLevel` property for the WSA instance to the following value:

- **3** — To get information about user authorization errors or to log the start of a new client request for any Web service deployed to this WSA instance

- **4** — To get detailed information about WSA startup parameters

Hints for setting the Web service serviceLoggingLevel property

Set the `serviceLoggingLevel` property for the Web service to the following value:

- **4** — To record Web service request parameter values.
- **3** — To track Web service request execution. Use this value during Web service development to provide more information for each Web service request error.
- **2** — To reduce logging overhead when the Web service is running in production. This value only allows severe errors to be recorded that affect the Web service as a whole, and no errors to be recorded for single Web service requests.

During production run time, you might temporarily set the value to 3 or 4 when working with a specific client to resolve an error, then return the logging level to 2 for general client access when the problem has been resolved.
Setting the logging level properties

Settings to the serviceLoggingLevel and serviceFaultLevel take effect immediately at run time. You can also set an immediate run-time value for the WSA loggingLevel property, as well as a persistent value that takes affect the next time the WSA is started.

To change these settings, use the Progress Explorer or the wsaman utility. If you are working with a WSA installation on a system that does not have an AdminServer installed, you must edit the ubroker.properties file to change the persistent value of loggingLevel property.

For more information on setting these logging level properties and their appropriate values using the OpenEdge Explorer and Progress Explorer, see the online help. You can also set the LoggingLevel property if you are using the wsaman utility, see the sections on the wsaman utility in OpenEdge Application Server: Administration. For more information on the serviceLoggingLevel and serviceFaultLevel properties, see the Web service properties reference sections of OpenEdge Application Server: Administration.

For complete information on OpenEdge general logging capabilities, see OpenEdge Development: Debugging and Troubleshooting.

For more information on using log files, see the “Identifying relevant log files” section on page 5–7. For more information on handling SOAP faults, see the “Working with SOAP faults” section on page 5–8.
Identifying relevant log files

A common way to detect software errors of all kinds, not just Web service errors, is to examine log files for error messages. Table 5–2 lists OpenEdge log files providing Web-service-related messages. These files all reside in your OpenEdge work directory.

Table 5–2: OpenEdge log files providing Web service-related messages

<table>
<thead>
<tr>
<th>This component . . .</th>
<th>Logs messages here . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSA instance</td>
<td>instance-name.wsa.log</td>
</tr>
<tr>
<td>Web service</td>
<td>instance-name.wsa.log</td>
</tr>
<tr>
<td>AdminServer</td>
<td>adminserv.log</td>
</tr>
<tr>
<td>AppServer</td>
<td>instance-name.broker.log</td>
</tr>
<tr>
<td></td>
<td>instance-name.server.log</td>
</tr>
<tr>
<td>NameServer</td>
<td>instance-name.NS.log</td>
</tr>
</tbody>
</table>

For information on managing logging for the AdminService, NameServer, and in general for all OpenEdge servers managed by the Progress Explorer Framework, see OpenEdge Getting Started: Installation and Configuration. For more information on managing logging for an AppServer, see OpenEdge Application Server: Developing AppServer Applications.
Working with SOAP faults

When an error occurs while processing a Web service client request, the WSA generates a SOAP fault message and returns it to the client application. The client application should catch all SOAP faults and handle them programmatically, perhaps by displaying an error message to the user or by logging the fault information for review off line. If the client application does not catch the SOAP fault, the behavior of the client application depends on the default error handling of the client software development kit.

Setting the serviceFaultLevel property

You can control the amount of information provided with SOAP fault messages by how you set the serviceFaultLevel property for a Web service. Any change in value takes effect immediately at run time.

Set the serviceFaultLevel property to the following value:

- **2** — To log general error information in a production environment
- **3** — To log error information in a development environment or in a production environment where more detailed information is required

Note that all Web service logging and fault levels are controlled individually for each Web service to further reduce processing overhead. For more information on setting this property, see the “Setting error information levels” section on page 5–4.

Typical SOAP fault response message

The following is the content of a typical SOAP fault response that appears within the body of a WSA-generated SOAP fault message:

```xml
<soap:Fault>
  <faultcode>SOAP-ENV:Server</faultcode>
  <faultstring>An error was detected ... request. (10893) </faultstring>
  <detail>
    <FaultDetail xmlns="http://servicehost:8080/wsa/wsa1">
      <errorMessage>The urn:OrderSvc:OrderInfo service is unavailable (10921) </errorMessage>
      <requestID>2e62cab6b81150d5:-17c6a3c:f1dfffbd11b:-8000#1f</requestID>
    </FaultDetail>
  </detail>
</soap:Fault>
```

Note that SOAP faults can be returned by the client’s own SOAP libraries and might have a different XML definition for the contents encapsulated by the <detail> element, depending on the Web service client toolkit that is used. However, a SOAP fault generated by the OpenEdge WSA always includes an <error Message> and <requestID> element within the <detail> element section.
WSA's `<errorMessage>` element

The SOAP `<faultstring>` element provides only a very general description of the error. For the most effective diagnostic detail, always look at the information provided by the `<errorMessage>` element.

WSA's `<requestID>` element

Use the information provided by the `<requestID>` element when you contact the Web service deployer for assistance with an error. This information tells the Web service deployer whether more information can be obtained from the WSA log file. For more information on how the WSA uses the information in the `<requestID>` element to log information, see the “How the WSA logs Web service information” section on page 5–12.

Handling SOAP faults programmatically

The client application should handle all SOAP faults. Many SOAP client toolkits convert SOAP faults to client exceptions that can be handled in a `try...catch` block. The following code is a VB.NET fragment from the sample applications that are available on the Documentation and Samples (doc_samples) directory of the OpenEdge product DVD:

```vbnet
Try
   ' Code to access the Web service
   ...
       Dim detail As String, reqId As String
       detail = parseSoapException(soapEx.Detail, reqId)
       MsgBox(detail, MsgBoxStyle.Critical, soapEx.Message)
End Try
```

In this example, `parseSoapException( )` is a client helper function to parse the SOAP fault detail provided in .NET sample code. You can find more examples like this in both the .NET and Java sample client applications. For more information on these sample applications, see the “Sample Web service applications” section on page 1–17.
Using a SOAP message viewer

To view SOAP messages as they are sent and received, use a SOAP message viewer. Table 5–3 lists some examples.

Table 5–3: Some SOAP message viewers

<table>
<thead>
<tr>
<th>This SOAP message viewer . . .</th>
<th>Is . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSAViewer</td>
<td>Provided with the OpenEdge installation</td>
</tr>
<tr>
<td>ProSOAPView</td>
<td>Provided with the OpenEdge installation</td>
</tr>
<tr>
<td>Microsoft SOAP Toolkit</td>
<td>Available for free download from Microsoft’s Web site</td>
</tr>
</tbody>
</table>

To use a SOAP message viewer:

1. Change the Web service client to send requests to the port of the SOAP message viewer by changing the SOAP end point’s URL host and port fields.

2. Configure the SOAP message viewer to forward requests to the host and port of the WSA, where the Web service is deployed, and run the viewer.

To configure and run WSAViewer:

1. Enter the `wsaviewer` command from a Proenv shell using the following syntax:

   **Syntax**

   ```
   wsaviewer listen-port WSA-host WSA-port
   ```

2. Change the Web service URL in the Web service client to access the WSAViewer. Assume that the original Web service URL in the client is as follows:

   ```
   http://bedrockquarry.com:8080/wsa/wsa1
   ```

   In the client, change this URL to use an agreed-upon viewer listening port:

   ```
   http://bedrockquarry.com:8081/wsa/wsa1
   ```

3. To run the WSAViewer, enter this command in the Proenv shell on the WSA machine:

   ```
   wsaviewer 8081 localhost 8080
   ```
The WSAViewer runs configured to listen for requests from the reconfigured client interface on port 8081 and forward those requests to port 8080. SOAP responses from the WSA, then, move in the reverse direction.

Figure 5–1 shows the main window of the WSAViewer when it first opens.

![WSAViewer main window](image)

**Figure 5–1: WSAViewer main window**

SOAP requests from the client appear in the left-hand viewer and SOAP responses from the WSA forwarded back to the client appear in the right-hand viewer. A continuous stream of requests and responses is recorded that you can review by using the vertical scroll bars.

To configure and run ProSOAPView, see the “Using ProSOAPView” section on page 10–14.

ProSOAPView is more general purpose than WSAViewer, and it allows you to trace not only SOAP messages, but also HTTP messages and other document contents exchanged between a client and server. ProSOAPView works as both an HTTP proxy server and an HTTP client that can connect to your choice of another proxy server or the final destination.
Identifying errors

This section describes how to identify errors and describes some of the more common errors that occur in WSA administration requests and in Web service application requests.

How the WSA logs Web service information

Each time the WSA starts up, each WSA instance receives a unique, alphanumeric identifier (WSA ID), which the WSA always places in a log file entry that indicates the WSA is starting up. This WSA ID appears in the log file similar to this example, always appearing after the “ID”:

```
ID 2bbafbc35852308b:7a8a02:f18e8ee918:-8000
```

When a WSA instance first receives a request for one of its deployed Web services, the WSA instance assigns the request an alphanumeric identifier (request ID) that is unique for the current session of the WSA instance. When the WSA logs information about a Web service request, whether the information concerns processing steps or errors, the WSA always includes its unique request ID in the log entry text using the following format, where `reqid-value` is the request ID for the Web service request to which the log entry applies:

```
(reqid:reqid-value)
```

When the WSA returns a SOAP fault to a client, it includes the request ID of the Web service request that triggered the SOAP fault, as well as the WSA ID of the WSA instance where the Web service is deployed. The WSA places these two IDs within the `<requestID>` element of the SOAP fault message according to the following format, where `wsaid-value` is the WSA ID and `reqid-value` is the request ID:

```
wsaid-value#reqid-value
```

The client can then obtain these two IDs from the `<requestID>` element of the SOAP fault and provide it to the WSA host operator to look up more detailed information in the corresponding entries of the WSA log file. The WSA operator then uses the WSA ID to locate the WSA log file for the corresponding WSA instance. They can now open the log file and search for log file entries that contain the “`(reqid:reqid-value)`” string to find all information related to the specific request and WSA instance.
Errors occurring in WSA administration requests

An error might occur in a WSA administration request. Such errors occur when you use the Progress Explorer or the wsaman utility. Both these tools are front ends to the AdminServer, which creates SOAP requests as a Web service client, then sends them to the WSA. When an error in a WSA administration request occurs, it is logged. Where it is logged depends on when it occurred, as shown in Table 5–4.

Table 5–4: WSA administration error logging

<table>
<thead>
<tr>
<th>If the error occurred . . .</th>
<th>The error is logged . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>While the request is being prepared in the AdminServer</td>
<td>In the AdminServer log</td>
</tr>
<tr>
<td>While the request is being processed in the WSA</td>
<td>In the WSA instance log</td>
</tr>
</tbody>
</table>

Errors occurring in Web service application requests

A Web service application request is initiated by a client, which creates a SOAP message and sends it to the WSA. The WSA processes the SOAP message, creates a response, and sends the response back to the client. An error can occur on the client side or the server side.

Client-side errors

A client-side error occurs before the request is sent or while it is being processed by the WSA.

An example of an error occurring before the request is sent is a mismatch in interface parameters. For these types of errors, the type of client response depends on the client platform.

Examples of errors occurring while the request is being sent (which involve a session-managed AppServer application) include:

- Forgetting to call the connect method first
- Not sending the correct Object ID in the SOAP header
- Web server not running
- Web service URL incorrectly specified (host or port)
- Network down
- Using HTTP instead of HTTPS when required

These types of errors result in a SOAP fault response and log file entries.
Server-side errors

A server-side error occurs while the request is being processed by the WSA and the error is not related to the contents of the SOAP request. Examples of server-side errors include:

- Forgetting to enable a specific deployed Web service
- Forgetting to enable client access to Web services for the WSA instance
- Configuring or starting an AppServer incorrectly—this might involve one or more of the following issues:
  - Invalid Propath on the AppServer
  - Invalid AppService (Application Service name)
  - Incompatible AppServer operating mode for the Web service session model
  - Missing ABL, or an incorrect version of ABL
- Configuring the Java Servlet Engine (JSE) incorrectly so it does not recognize the WSA

These types of errors result in a SOAP fault response and log file entries.
Sample error scenarios

This section discusses sample error scenarios. These scenarios involve a VB.NET client accessing a Web service whose specification is listed in Table 5–5.

Table 5–5: Sample Web service specification for error scenarios

<table>
<thead>
<tr>
<th>Web service specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>OrderInfo</td>
</tr>
<tr>
<td>TargetNamespace</td>
<td>urn:www-progress-com:samples:Orderinfo</td>
</tr>
<tr>
<td>AppObject</td>
<td>OrderInfo</td>
</tr>
<tr>
<td>ProcObject</td>
<td>CustomerOrder</td>
</tr>
<tr>
<td>Session Model</td>
<td>Managed</td>
</tr>
</tbody>
</table>

Note: These sample scenarios are not necessarily based on the samples available on the Documentation and Samples (doc_samples) directory of the OpenEdge product DVD or the Progress Documentation Web site.
Scenario 1: Web service deployed but not enabled

In this scenario, a Web service is deployed but not enabled. At this point:

1. The client sends the following connection request:

   ```
   Connect_OrderInfo("","","")
   ```

2. The WSA gets the request and determines the Web service is disabled. At this point:
   a. The WSA sends the client a SOAP response containing a SOAP fault.
   b. The log files do not show any error.

   The following is the SOAP response containing the SOAP fault:

   ```
   <?xml version="1.0" encoding="UTF-8" ?>
   <soap:Envelope  namespaces defined here..>
   <soap:Body>
   <soap:Fault>
     <faultcode>SOAP-ENV:Server</faultcode>
     <faultstring>An error was detected ... request. (10893)
   </faultstring>
   <detail>
     <FaultDetail xmlns="http://servicehost:8080/wsa/wsa1">
       <errorMessage>The urn:www-progress-com:samples:OrderInfo
OrderInfo service is unavailable (10921)
       </errorMessage>
       <requestID>2e62cab6b81150d5:-17c6a3c:f1dffbd11b:-8000#1f
       </requestID>
     </FaultDetail>
   </detail>
   </soap:Fault>
   </soap:Body>
   </soap:Envelope>
   ```

3. The interface translates the SOAP fault into a SOAP exception.

4. The client code catches the SOAP exception.
5. The client displays the following error message (assuming the client application has a user interface):

![Error Message](image)

**Note:** All client applications should always catch SOAP faults and handle them. For client applications without a user interface, all error information must be written to an application log file to ensure it is not lost. If a .NET client does not do this, the .NET Framework’s default exception handler displays the SOAP fault general error information. This might well hide and cause the loss of the more useful error detail information in the SOAP fault.

---

**Scenario 2: ABL updated, but Web service not redeploed**

In this scenario, ABL is updated with a different procedure prototype (signature), but the Web service is not updated accordingly. At this point:

1. The client sends the following method request:

   ```
   FindCustomerByNum(3, custNameVar)
   ```

2. The WSA gets the request and asks the AppServer to run the procedure `FindCustomerByNum.p`.

3. The AppServer:
   a. Tries to run `FindCustomerByNum.p`
   b. Detects a parameter-type mismatch and records the following error message in its log, `*.server.log`:

   ```
   [02/12/11@16:07:01.427-0500] P-000371 T-000370 0 AS --
   Mismatched parameter types passed to procedure
   OrderInfo/FindCustomerByNum.p. (3230)
   ```
   c. Returns the error to the WSA
4. The WSA:
   a. Records the following message in its log:

   ![Log Entry]

   b. Sends the following SOAP containing a SOAP fault back to the client:

   ```xml
   <?xml version="1.0" encoding="UTF-8" ?>
   <soap:Envelope  xmlns="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
   <soap:Fault>
   <faultcode>SOAP-ENV:Server</faultcode>
   <faultstring>An error was detected ... request. (10893)</faultstring>
   <detail>
   <FaultDetail xmlns="http://servicehost:8080/wsa/wsa1">
   <errorMessage>Error in SOAP request execution: ABL ERROR condition: Mismatched parameter types passed to procedure OrderInfo/FindCustomerByNum.p. (3230) (7211) (10926)</errorMessage>
   <requestID>2e62cab6b81150d5:-87f76e:f20f57227d:-8000#6</requestID>
   </FaultDetail>
   </detail>
   </soap:Fault>
   </soap:Body>
   </soap:Envelope>
   ```

   **Note:** This example uses a value of 3 for the serviceFaultLevel property.

   Notice the correspondence between the "(reqid:6)" string in the log file entry and the "<requestID> ...2e62 ...#6" string in the SOAP fault message, indicating information for the same request.

5. The interface translates the SOAP fault into a SOAP exception.

6. The client catches the SOAP exception.

7. The client displays the following error message (assuming the client application has a user interface):

   ![Error Message]

   An error was detected while executing the Web Service request (10893)
Scenario 3: Web service deployed but AppServer not running

In this scenario, a Web service is deployed, but its AppServer is not running. At this point:

1. The client sends the following connection request:

   ```plaintext
   Connect_Orderinfo('', '', '')
   ```

2. The WSA:
   a. Receives the request
   b. Determines that the AppServer is not available
   c. Records the following in its log:

   ```plaintext
   [02/12/11@16:17:27.349-0500] P-000120 T-Thread-11 2 OrderInfo
   ABL-Provider (reqid:8) Error in SOAP request execution: Connect
   Failure: Application Service asbroker1 Unknown at NameServer
   at Host servicehost Port 5162. (8234) (10926)
   ```
   d. Sends a SOAP message with a SOAP fault back to the client. The message is as follows:

   ```xml
   <?xml version="1.0" encoding="UTF-8" ?>
   <soap:Envelope xmlns:ns="http://servicehost:8080/wsa/wsa1">
     <soap:Body>
       <soap:Fault>
         <faultcode>SOAP-ENV:Server</faultcode>
         <faultstring>An error was detected in the request. (10893)
         </faultstring>
         <detail>
           <FaultDetail xmlns:ns="http://servicehost:8080/wsa/wsa1">
             <errorMessage>Error in SOAP request execution: Connect
             Failure: Application Service asbroker1 Unknown at
             NameServer at Host servicehost Port 5162. (8234) (10926)
             </errorMessage>
             <requestID>2e62cab6b81150d5:-87f6e:f20f57227d:-8000#8
             </requestID>
           </FaultDetail>
         </detail>
       </soap:Fault>
     </soap:Body>
   </soap:Envelope>
   ```
3. The interface translates the SOAP fault into a SOAP exception.

4. The client catches the SOAP exception.

5. The client displays this error message (assuming the client application has a user interface):

![Error Message](image.png)
Part III

Creating ABL Clients to Consume Web Services

Chapter 6, Creating an ABL Client from WSDL
Chapter 7, Connecting to Web Services from ABL
Chapter 8, Invoking Web Service Operations from ABL
Chapter 9, Handling SOAP Message Headers in ABL
Chapter 10, Handling Errors in ABL Requests to Web Services
Creating an ABL Client from WSDL

The chapter describes how to turn WSDL files into the basis for an ABL client that consumes Web services. This includes understanding how to generate ABL interface documentation from a WSDL file and how OpenEdge maps Web service parameters from the XML Schema types in WSDL to the ABL data types.

The sections in this chapter are:

- Basics of an ABL client to consume Web services
- Using the WSDL Analyzer
- Mapping XML Schema data types to ABL data types
Basics of an ABL client to consume Web services

You have obtained, by one of the methods described in the “Distributing your WSDL file” section on page 2–10, the WSDL file for the Web service that your ABL client will consume. You run the WSDL Analyzer on the file and get pages of sample ABL code. The samples cover all the necessary steps to consume the Web service.

As shown in the “Consuming Web service example” section on page 1–15, an ABL client does the following to consume a Web service:

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.

You can find sample code to cover this process in the WSDL Analyzer’s output. The connection details (Figure 6–6) and operation details (Figure 6–7) sections of the Port type page provide most of these code samples.
Using the WSDL Analyzer

After you know the location to a WSDL file for a target Web service, use the WSDL Analyzer to provide HTML documentation on the interface to the Web service. The documentation describes the requirements for consuming the Web service using ABL, and includes binding information and ABL prototypes for accessing operations in the Web services. The documentation also includes any internal WSDL comments by the Web service developer to explain elements of the Web service interface. The following sections describe:

- Running the WSDL Analyzer
- Understanding the WSDL Analyzer output
- Analyzing wrapped document literal
- Analyzing complex data

Running the WSDL Analyzer

To run the WSDL Analyzer, enter the `bprowsd1doc` command on your operating system command line using the following syntax:

**Syntax**

```plaintext
bprowsd1doc [ -h ] | { [ option ] . . . wsdl-url-or-filename [ target-directory ] }
```

**Note:** Use the command window (shell) opened by the `proenv` command tool to enter this command with the correct path settings.

- `h`
  Displays a help message on the usage of this command.

- `option`
  Specifies one of the following command options:

  **Syntax**

  ```plaintext
  -b
  | -nohostverify
  | -nosessionreuse
  | -WSDLUserid username
  | -WSDLPassword password
  | -proxyhost host
  | -proxyport port
  | -proxyUserid username
  | -proxyPassword password
  | -show100style
  | -noint64
  ```
Creating an ABL Client from WSDL

-b

Forces documentation of binding names.

-nohostverify | -nosessionreuse

Turns off host verification and reuse of session IDs for HTTPS connections.

-WSDLUserid username | -WSDLPassword password

Provides access to a secured WSDL file.

-proxyhost host | -proxyport port |
-proxyuserid username | -proxyPassword password

Specifies a proxy for the connection and provides access to a secure proxy.

-show100style

Shows procedure and function signatures as documented in the 10.0x releases of OpenEdge. For more information, see the “Analyzing wrapped document literal” section on page 6–13.

-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.

target-directory

Specifies a the target directory where the WSDL Analyzer writes the generated pages. This defaults to the current working directory.

For more information on the bprowsdlcmdoc command, see Appendix F, “Commands and Utilities.”
Understanding the WSDL Analyzer output

The WSDL Analyzer produces hyperlinked HTML pages in the specified target directory. The following sections describe the information for building your ABL client that you can find on each page. All the pages include hyperlinks to the other pages, so you can easily browse through the complete set.

Table 6–1 lists the information found on each page of the output.

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index and Service pages</td>
<td>WSDL</td>
<td>The document heading followed by any general comments from the WSDL describing the entire WSDL file.</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>The URL of the WSDL file. This is always a complete path name (absolute path) to the file.</td>
</tr>
<tr>
<td></td>
<td>Target namespace</td>
<td>The target namespace of the Web service.</td>
</tr>
<tr>
<td></td>
<td>Service-name service</td>
<td>The name of the service followed by any comments included in WSDL <code>&lt;documentation&gt;</code> elements about the service.</td>
</tr>
<tr>
<td></td>
<td>Port types (persistent procedures)</td>
<td>A hyperlinked list of the port types defined in the Web service, including any specific comments about them from the WSDL.</td>
</tr>
<tr>
<td></td>
<td>Data types</td>
<td>A hyperlinked list of the complex types defined for the Web service, including any specific comments about them from the WSDL.</td>
</tr>
<tr>
<td>Operation index page</td>
<td>Operation</td>
<td>An index table listing all operations described in the WSDL, including the port type and service for each operation and any specific comments about the operation from the WSDL. The operation names link to the operation detail in the port type page.</td>
</tr>
<tr>
<td>Port type page</td>
<td>Port type (persistent procedure)</td>
<td>The document heading. <strong>Note:</strong> The port type is modelled as a persistent procedure, but is not actually persistent.</td>
</tr>
<tr>
<td></td>
<td>Port-type-name</td>
<td>The defined name of the port type followed by comments in any WSDL <code>&lt;documentation&gt;</code> elements that describe the entire port type.</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>Basic connection (binding) information and a list of the operations defined by the port type, including comments in any WSDL <code>&lt;documentation&gt;</code> elements on each operation.</td>
</tr>
</tbody>
</table>
### Creating an ABL Client from WSDL

#### Port type page (continued)

**Connection details**

Provides information on how to connect to the Web service to use this port type, including the following sections:

- **Connection parameters** — ABL syntax for all of the Web service-related parameters that can be included in the connection parameters string passed to the ABL `CONNECT()` method, and an ABL example of how they might be used. Some of the same startup options used for execution of the WSDL Analyzer also appear in this section, such as those used to specify HTTPS and proxy server connections.

- **-Service and -Port descriptions** — A table listing the name of each service and the names of ports for each service that support this port type. For each service and port name, the table also includes comments in any WSDL `<documentation>` elements defined for each service and port. So, for example, if the port type is supported for different applications and in different locations, you might well have a corresponding choice of services and ports to make the connection.

- **Example** — Provides a real code example that illustrates how you might connect and set up the Web service to access the operations of the port type.

- **Binding descriptions** — If you specify the `-b` command-line option, this section appears in the document. It shows a table listing the name of each binding that supports this port type. For each binding, the table also includes comments in any WSDL `<documentation>` elements defined for each binding.

#### Operation (internal procedure) detail

A series of subsections describing each Web service operation defined for the port type mapped as an internal procedure (and user-defined function, if supported), including:

- The operation name
- Comments in any WSDL `<documentation>` elements
- ABL procedure prototypes (and function prototypes, if also supported)
- ABL example
- Descriptions of parameters and any return value (for a user-defined function mapping), including SOAP examples for complex types as appropriate for different service and port combinations
- Any SOAP request/response headers and SOAP faults as appropriate for different service and port combinations

---

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port type page</td>
<td>Connection details</td>
<td>Provides information on how to connect to the Web service to use this port type, including the following sections:</td>
</tr>
<tr>
<td>(continued)</td>
<td></td>
<td>- <strong>Connection parameters</strong> — ABL syntax for all of the Web service-related parameters that can be included in the connection parameters string passed to the ABL <code>CONNECT()</code> method, and an ABL example of how they might be used. Some of the same startup options used for execution of the WSDL Analyzer also appear in this section, such as those used to specify HTTPS and proxy server connections.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>-Service and -Port descriptions</strong> — A table listing the name of each service and the names of ports for each service that support this port type. For each service and port name, the table also includes comments in any WSDL <code>&lt;documentation&gt;</code> elements defined for each service and port. So, for example, if the port type is supported for different applications and in different locations, you might well have a corresponding choice of services and ports to make the connection.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>Example</strong> — Provides a real code example that illustrates how you might connect and set up the Web service to access the operations of the port type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>Binding descriptions</strong> — If you specify the <code>-b</code> command-line option, this section appears in the document. It shows a table listing the name of each binding that supports this port type. For each binding, the table also includes comments in any WSDL <code>&lt;documentation&gt;</code> elements defined for each binding.</td>
</tr>
<tr>
<td>Operation (internal</td>
<td></td>
<td>A series of subsections describing each Web service operation defined for the port type mapped as an internal procedure (and user-defined function, if supported), including:</td>
</tr>
<tr>
<td>procedure) detail</td>
<td></td>
<td>- The operation name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Comments in any WSDL <code>&lt;documentation&gt;</code> elements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ABL procedure prototypes (and function prototypes, if also supported)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ABL example</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Descriptions of parameters and any return value (for a user-defined function mapping), including SOAP examples for complex types as appropriate for different service and port combinations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Any SOAP request/response headers and SOAP faults as appropriate for different service and port combinations</td>
</tr>
</tbody>
</table>
Using the WSDL Analyzer

Index and Service pages

The index page, `index.html`, is the starting page of the documentation describing the Web service interface. This page takes one of two forms, depending on whether the WSDL file defines a single service or multiple services. The page for a single-service Web service includes the port types and data types for that service. The page for a multi-service Web service links to individual Service pages that provide this information separately for each service. Whether on a single page or multiple pages, they provide the information listed in Table 6–1.

Figure 6–1 shows an example of an index page for a WSDL containing only one service.

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data types page</td>
<td>Data types</td>
<td>The document heading followed by comments in any WSDL &lt;documentation&gt; elements that describe the entire types section</td>
</tr>
<tr>
<td>Datatype summary</td>
<td></td>
<td>A hyperlinked alphabetical listing of all the complex types defined for the Web service, including comments in any WSDL &lt;documentation&gt; elements as defined for each data type</td>
</tr>
<tr>
<td>Datatype detail</td>
<td></td>
<td>A series of subsections, one for each complex type defined for the Web service, listed in alphabetical order, and each one including comments in any WSDL &lt;documentation&gt; elements defined for the complex type, the XML &lt;schema&gt; element that defines the type, and an example of a SOAP instance</td>
</tr>
</tbody>
</table>

**Table 6–1: WSDL Analyzer output**

<table>
<thead>
<tr>
<th>Data types</th>
<th>No documentation found in WSDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>FacShipObj</td>
<td>No documentation found in WSDL</td>
</tr>
<tr>
<td>NewCoServiceObj</td>
<td>No documentation found in WSDL</td>
</tr>
</tbody>
</table>

**WSDL**

Author=Progress Education Services, Encoding=ENCODING, WSA_Product=10.1.01 - N/A

**Location**

http://wsisapi.progress.com/wsdl/wsdl?targetURI=wsa:OpenEdgeServices/NewCoService

**Target namespace**

wsa:OpenEdgeServices>NewCoService

**NewCoService Service**

No documentation found in WSDL

**Port types (persistent procedures)**

<table>
<thead>
<tr>
<th>FacShipObj</th>
<th>No documentation found in WSDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewCoServiceObj</td>
<td>No documentation found in WSDL</td>
</tr>
</tbody>
</table>

**Data types**

| ArrayOfInvoiceTotalsInfo_InvoiceTotalsRow | No documentation found in WSDL |
| FacShipDetail | No documentation found in WSDL |
| NewCoServiceObj | No documentation found in WSDL |
| ArrayOfInvoiceTotalsInfo_InvoiceTotalsRow | No documentation found in WSDL |
Figure 6–2 shows an example of an index page for a WSDL containing only one service.

**Figure 6–2: WSDL Analyzer index page for multiple services**

Figure 6–3 shows an example of a service page for one of the services shown in Figure 6–2.

**Figure 6–3: WSDL Analyzer service page for one of multiple services**
Operation index page

The operation index page contains a hyperlinked list of all operations from all port types in all services defined in the WSDL file. Figure 6–4 shows an example of this page.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Documentation</th>
<th>Port type</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>CancelOrder</td>
<td>No documentation found in WSDL</td>
<td>NewCoServiceObj</td>
<td>NewCoService</td>
</tr>
<tr>
<td>CreatePO Status</td>
<td>No documentation found in WSDL</td>
<td>NewCoServiceObj</td>
<td>NewCoService</td>
</tr>
<tr>
<td>CustomerName</td>
<td>No documentation found in WSDL</td>
<td>NewCoServiceObj</td>
<td>NewCoService</td>
</tr>
<tr>
<td>CustomerNumber</td>
<td>No documentation found in WSDL</td>
<td>NewCoServiceObj</td>
<td>NewCoService</td>
</tr>
<tr>
<td>GetCustomerNumber</td>
<td>No documentation found in WSDL</td>
<td>NewCoServiceObj</td>
<td>NewCoService</td>
</tr>
<tr>
<td>GetInvoice</td>
<td>No documentation found in WSDL</td>
<td>NewCoServiceObj</td>
<td>NewCoService</td>
</tr>
<tr>
<td>GetInvoiceTotal</td>
<td>No documentation found in WSDL</td>
<td>NewCoServiceObj</td>
<td>NewCoService</td>
</tr>
<tr>
<td>GetInvoiceTotalInv</td>
<td>No documentation found in WSDL</td>
<td>NewCoServiceObj</td>
<td>NewCoService</td>
</tr>
<tr>
<td>GetInvoiceOrderInv</td>
<td>No documentation found in WSDL</td>
<td>NewCoServiceObj</td>
<td>NewCoService</td>
</tr>
<tr>
<td>GetSalesRepPicture</td>
<td>No documentation found in WSDL</td>
<td>NewCoServiceObj</td>
<td>NewCoService</td>
</tr>
<tr>
<td>GetShipDate</td>
<td>No documentation found in WSDL</td>
<td>NewCoServiceObj</td>
<td>NewCoService</td>
</tr>
</tbody>
</table>

Figure 6–4: WSDL Analyzer operation index page

Port type page

The WSDL Analyzer generates a separate port type page for each port type defined in the WSDL file. The page contains detailed information on how to work with that port type. Figure 6–5 shows an example of the beginning of a port type page.

<table>
<thead>
<tr>
<th>Port type (persistent procedure)</th>
<th>No documentation found in WSDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewCoServiceObj</td>
<td></td>
</tr>
</tbody>
</table>

Summary

Connection information

The following connection parameters must be specified to use the operations described below. See the Connection Details topic below:

-WSDL 'http://wstraining.progress.com/wsa_wsdl?wsdlTargetURL=urn:OpenEdgeServices:NewCoService'

Operations (Internal procedures)

- PROCEDE CancelOrder:
  - BEFORE INPUT PARAMETER Customer AS INTEGER NO-UNDO.
  - BEFORE INPUT PARAMETER CID AS CHARACTER NO-UNDO.
  - BEFORE OUTPUT PARAMETER Results AS LOGICAL NO-UNDO.
  - END PROCEDURE.

- PROCEDE CreatePO Status:
  - END PROCEDURE.

- PROCEDE GetCustomerNumber:
  - BEFORE INPUT PARAMETER Customer AS CHARACTER NO-UNDO.
  - BEFORE OUTPUT PARAMETER Customer AS INTEGER NO-UNDO.
  - END PROCEDURE.

- PROCEDE GetInvoice:
  - END PROCEDURE.

Figure 6–5: WSDL Analyzer port type page (up to the Summary)
Figure 6–6 shows an example of the **Connection details** section of a port type page.

### Connection details

**Connection parameters**

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-WSDLPassword:</td>
<td>password</td>
</tr>
<tr>
<td>[ ]</td>
<td>-Service NewCoServiceService</td>
</tr>
<tr>
<td>[ ]</td>
<td>-ServiceNamespace urn:OpenEdgeServices:NewCoService</td>
</tr>
<tr>
<td>[ ]</td>
<td>-Port NewCoService</td>
</tr>
<tr>
<td>[ ]</td>
<td>-SOAPEndpointUser Id user</td>
</tr>
<tr>
<td>[ ]</td>
<td>-SOAPEndpointPassword password</td>
</tr>
<tr>
<td>[ ]</td>
<td>-TargetNamespace urn:OpenEdgeServices:NewCoService</td>
</tr>
<tr>
<td>[ ]</td>
<td>-NewConnections 9999999999</td>
</tr>
<tr>
<td>[ ]</td>
<td>-pf filename</td>
</tr>
<tr>
<td>[ ]</td>
<td>-subsystem1</td>
</tr>
<tr>
<td>[ ]</td>
<td>-resource1</td>
</tr>
</tbody>
</table>

**-Service and -Port descriptions**

The following service name may be used for the -Service connection parameter:

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewCoServiceService</td>
<td>No documentation found in WSDL. The following port name may be used for the -Port connection parameter, when using the NewCoServiceService service.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewCoService</td>
<td>This service provides customers with the capability of placing Orders with NewCo and receiving Invoices. It uses an EPC/Encoded binding</td>
</tr>
</tbody>
</table>

**Example**

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

#define variable hWebService AS HANDLE NO-UNDO.  
#define variable hNewCoServiceObj AS HANDLE NO-UNDO.  

CREATE SERVER hWebService.  


Figure 6–7 shows an example of how the Operation (internal procedure) detail section of a port type page describes a Web service operation.

<table>
<thead>
<tr>
<th>Operation (internal procedure) detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>CancelOrders</td>
</tr>
</tbody>
</table>

*No documentation found in WSDL.*

**Procedure prototype:**

``` abl
PROCEDURE CancelOrders;
    DEFINE INPUT PARAMETER iCustomer AS INTEGER NO-UNDO.
    DEFINE INPUT PARAMETER id AS CHARACTER NO-UNDO.
    DEFINE OUTPUT PARAMETER result AS LOGICAL NO-UNDO.
END PROCEDURE.
```

**Example:**

``` abl
DEFINE VARIABLE iCustomer AS INTEGER NO-UNDO.
DEFINE VARIABLE id AS CHARACTER NO-UNDO.
DEFINE VARIABLE result AS LOGICAL NO-UNDO.
RUN CancelOrders ON ShowUnCompleteOrders(INPUT iCustomer, INPUT id, OUTPUT result).
```

**Parameters:**

- iCustomer: This value is defined as an XML Schema int value.
- id: This value is defined as an XML Schema string value.
- result: This value is defined as an XML Schema boolean value.

**Fault details:**

The following XML fragments may be sent or received in the SOAP fault detail element. These fragments are accessible through the SOAP Fault handler:

- A fault defined as a FaultDetail data type value in the urn:schemas:soap:soap fault namespace.

**Example**

Construct the value as shown in the following example:

``` xml
<soap:detail xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:FaultDetail>
        <soap:Fault>
            <soap:Message>string-value</soap:Message>
            <soap:Detail>string-value</soap:Detail>
        </soap:Fault>
    </soap:FaultDetail>
</soap:detail>
```

**Figure 6–7: WSDL Analyzer port type page (operation detail)**

The WSDL Analyzer maps each Web service operation in a port type to an ABL internal procedure prototype. This allows all Web service operations to be invoked either synchronously or asynchronously as you require. For some Web service operations, the WSDL Analyzer also indicates a mapping to an ABL user-defined function prototype.

The WSDL Analyzer describes both the internal procedure mapping and the user-defined function mapping in the port type page. You can invoke them either as ABL procedures or user-defined functions, depending on your requirements at any given point in your application. However, as with user-defined functions in the native ABL, you cannot use the function to asynchronously invoke the Web service operation.
The port type page also describes how the WSDL Analyzer maps any complex data to ABL temp-tables or ProDataSets. For more information, see the “Analyzing complex data” section on page 6–14.

**Data types page**

For all the complex types defined in the WSDL file, the WSDL Analyzer generates a single data types page that contains detailed information on each complex type, whether or not they are actually used in the Web service. Figure 6–8 shows an example of the information from a data type page.

---

**Data types**

**Datatype summary**

<table>
<thead>
<tr>
<th>Datatype Name</th>
<th>Documentation in WSDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayOfGetInvoiceTotalInfo_InvoiceTotalRowsRow</td>
<td>No documentation found in WSDL</td>
</tr>
<tr>
<td>FaultDetail</td>
<td>No documentation found in WSDL</td>
</tr>
<tr>
<td>taxBillDetails</td>
<td>No documentation found in WSDL</td>
</tr>
<tr>
<td>GetInvoiceTotalInfo_InvoiceTotalRowsRow</td>
<td>No documentation found in WSDL</td>
</tr>
</tbody>
</table>

**Datatype detail**

**ArrayOfGetInvoiceTotalInfo_InvoiceTotalRowsRow**

No documentation found in WSDL

**Note:** The following schema definition and example may not agree. The example may indicate that all elements are optional, while the schema indicates otherwise. If this is the case, base the construction of your complex data on the example, not the schema fragment. This discrepancy is a result of the web service using SOAP encoding for its message serialization.

**Schema**

```xml
<schema elementFormDefault="unqualified" targetNamespace="urn:OpenEdgeServices:NewColService:NewColService"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.w3.org/2001/XMLSchema">
  <complexType name="ArrayOfGetInvoiceTotalInfo_InvoiceTotalRowsRow">
    <complexContent>
      <restriction base="arrayType">
        <attribute ref="soap:arrayType::arrayType" arrayType="ArrayOfGetInvoiceTotalInfo_InvoiceTotalRowsRow"/>
      </restriction>
    </complexContent>
  </complexType>
</schema>
```

**Example**

```xml
<complexType name="ArrayOfGetInvoiceTotalInfo_InvoiceTotalRowsRow">< xsd:complexType>
  <!-- The following element may appear 0 or more times.
  If minOccurs is specified above, the number of occurrences must agree with that value.
  If no value appears between the square brackets, then any number of occurrences may appear. -->
  <element minOccurs="0" name="totalOrder" type="xsd:decimal"/>

  <!-- The following element may have the "minInc="true" attribute (it is nullable). -->
  <element minOccurs="0" name="invoiceNumber" type="xsd:decimal"/>

  <!-- The following element may have the "minInc="true" attribute (it is nullable). -->
  <element minOccurs="0" name="totalCosts" type="xsd:decimal"/>

  <!-- The following element may have the "minInc="true" attribute (it is nullable). -->
  <element minOccurs="0" name="totalCostsProposal" type="xsd:decimal"/>
</complexType>
```

---

Figure 6–8: WSDL Analyzer data types page (summary list)
Note: This page only describes complex XML data types that cannot be represented as temp-tables or ProDataSets. It does not describe standard XML Schema types. For information on how XML Schema types map to ABL data types, see the “Mapping XML Schema data types to ABL data types” section on page 6–17. When the WSDL Analyzer identifies a complex type that maps to a temp-table or ProDataSet, it documents the ABL object and an example of its SOAP representation for the appropriate parameter, SOAP header entry, or SOAP fault detail in the port type page. For more information on how the WSDL Analyzer maps and documents temp-table and ProDataSet representations of complex types, see the “Analyzing complex data” section on page 6–14.

Analyzing wrapped document literal

The WSDL Analyzer recognizes and provides special documentation support for a narrow convention of the Doc/Lit SOAP format developed by Microsoft known as wrapped document literal (Wrapped Doc/Lit). A Web service operation defined according to the Wrapped Doc/Lit convention has a single request (input) parameter and single response (output) parameter. Each parameter in the operation is defined as a complex element that “wraps” one or more elements containing values of the same mode (input or output).

The WSDL Analyzer can document an ABL procedure or function that uses the Wrapped Doc/Lit convention according to one of two possible forms:

- Wrapped form
- Unwrapped form

Wrapped form

In a wrapped form, the procedure or function mirrors the Wrapped Doc/Lit convention and passes two LONGCHAR parameters, INPUT and OUTPUT, each of which contains the serialized XML data for the corresponding SOAP request or response message. Programming with this definition of the procedure or function requires that you parse or serialize XML for both parameters. For more information on coding with the wrapped form, see the “Programming options” section on page 6–14.

Unwrapped form

In an unwrapped form, the procedure or function passes individual INPUT, OUTPUT, or INPUT-OUTPUT parameters that reflect the individual data elements wrapped by the out element for these Wrapped Doc/Lit operation parameters. In this case, the WSDL Analyzer is likely to document each parameter as an ABL data type that maps to a simple XML Schema data type defined in the outer elements of the wrapped parameters. (For more information, see the “Mapping XML Schema data types to ABL data types” section on page 6–17.) If the definition for an unwrapped parameter embedded within a complex outer element is also a complex element, the WSDL Analyzer might have to document that unwrapped parameter as a complex XML Schema data type mapped to an ABL LONGCHAR. However, in many cases, only simple data types are required to implement all of the parameters for a Wrapped Doc/Lit operation. In this case, you have no need to parse any XML in ABL. For more information on coding with the unwrapped form, see the “Programming options” section on page 6–14.
**WSDL Analyzer documentation options**

By default, the WSDL Analyzer documents all Wrapped Doc/Lit operations using an unwrapped form. In most cases, this is the simplest form to work with in ABL.

However, versions of the WSDL Analyzer prior to OpenEdge Release 10.1 document Wrapped Doc/Lit operations using only the wrapped form. So, for backward compatibility, you can tell the WSDL Analyzer to document these operations using the wrapped form by specifying the `-show100style` command-line option when you run the WSDL Analyzer (see the “Running the WSDL Analyzer” section on page 6–3).

If any operations within a port type use the Wrapped Doc/Lit convention, the WSDL Analyzer writes a note at the top of the port type page, indicating that you can rerun the WSDL Analyzer with the `-show100style` option to generate the operation signatures as in OpenEdge releases prior to Release 10.1A.

**Programming options**

The ABL accepts, without restriction, both the wrapped and unwrapped forms for procedures or functions that map to Wrapped Doc/Lit operations. You can use either one or both of the wrapped and unwrapped forms as overloads of a Wrapped Doc/Lit operation (as documented by the WSDL Analyzer) to program that operation in ABL.

For more information on using the programming options that the WSDL Analyzer pages for Wrapped Doc/Lit Web services, see the “Coding options for wrapped document literal” section on page 8–10.

**Analyzing complex data**

Complex data consists of multiple data values treated as a group. A Web service can exchange complex data with a client using three different SOAP message data elements:

- A Web service operation parameter
- The entries of a SOAP header
- The detail of a SOAP fault

The techniques available for an ABL client to access this complex data depend on the structure and content of the data as determined by the WSDL Analyzer. If the WSDL is for an OpenEdge Web service that uses temp-table or ProDataSet parameters, the WSDL Analyzer indicates that you can access the complex data using an ABL temp-table or ProDataSet and how to define it in your code. If the WSDL Analyzer cannot identify a temp-table or ProDataSet definition to represent an instance of complex data, it provides an example of the serialized XML SOAP message element, so you can use ABL features to access the XML data as a Document Object Model (DOM) tree or to read and write the data using the Simple API for XML (SAX).

If the WSDL Analyzer identifies an appropriate temp-table or ProDataSet to represent the complex data, you can manage the data in your ABL program more easily than using the ABL DOM or SAX features. Access to such data relies entirely on ABL data types and requires no transformation to and from XML. You always have the option of using the DOM or SAX features to access the XML directly, but you are only required to do this when the WSDL Analyzer cannot identify a temp-table or ProDataSet definition to represent the complex data.
Figure 6–9 shows the **Operation (internal procedure) detail** section of a port type page for a procedure that is passing complex data parameters. The WSDL Analyzer has identified a static temp-table definition that can be used for a parameter. So, it specifies the TABLE parameter for the procedure signature.

---

### Operation (internal procedure) detail

**GetInvoiceTotalInfo**

No documentation found in WSDL.

**Procedure prototype**

```plaintext
PROCEDURE GetInvoiceTotalInfo:
  DEFINE INPUT PARAMETER invoiceNum AS INTEGER NO-UNDO.
  DEFINE OUTPUT PARAMETER TABLE FOR InvoiceTotals.
END PROCEDURE.
```

**Example**

```plaintext
DEFINE VARIABLE invoiceNum AS INTEGER NO-UNDO.
DEFINE TEMP-TABLE InvoiceTotals NO-UNDO
  NAMESPACE-URI "urn:OpenEdgeServices:NewCoService:NewCoService"
  FIELD invoiceNum AS INTEGER
  FIELD totalOrders AS DECIMAL
  FIELD shipCharge AS DECIMAL
  FIELD totalDue AS DECIMAL.

RUN GetInvoiceTotalInfo IN NewCoService object INPUT invoiceNum, OUTPUT TABLE InvoiceTotals).
```

**Parameters**

- **invoiceNum**
  - This value is defined as an XML Schema string value.

- **InvoiceTotals**
  - This value is defined as a TABLE.
  - This value can also be expressed as a **ArrayOfGetInvoiceTotalInfo_InvoiceTotalRow** value in the urn:OpenEdgeServices:NewCoService:NewCoService namespace. [View example](#)

**Fault details**

The following XML fragments may be sent or received in the SOAP fault detail element. These fragments are accessible through the SOAP Fault handler:

- A fault defined as a **FaultDetail** data type value in the urn:soap-fault-details namespace.

**Example**

Construct the value as shown in the following example:

```xml
<soap:Fault xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <faultcode>soap:Server</faultcode>
  <faultstring>Invalid parameter value</faultstring>
  <detail>
    <ArrayOfGetInvoiceTotalInfo_InvoiceTotalRow>
      <GetInvoiceTotalInfo_InvoiceTotalRow />
    </ArrayOfGetInvoiceTotalInfo_InvoiceTotalRow>
  </detail>
</soap:Fault>
```

---

**Figure 6–9: WSDL Analyzer port type page (TABLE in signature)**

### Web service operation parameter documentation

When the WSDL Analyzer identifies a Web service operation parameter as a static temp-table or ProDataSet, it documents the appropriate TABLE or DATASET parameter in the ABL procedure or user-defined function signature. It provides the ABL definition for the corresponding static temp-table or ProDataSet that you can use in your application. You can then access and manage the TABLE or DATASET parameter as in any ABL application.

6–15
If an OpenEdge Web service contains a dynamic temp-table or dynamic ProDataSet parameter, the WSDL Analyzer documents a corresponding TABLE-HANDLE or DATASET-HANDLE parameter in the ABL procedure or user-defined function signature. You can then access and manage the dynamic temp-table or ProDataSet parameter as in any ABL application.

If the WSDL Analyzer cannot identify any case for a temp-table or ProDataSet mapping, it documents the parameter as a LONGCHAR in the ABL procedure or user-defined function signature and provides an example SOAP message element for the complex data. In all cases where the WSDL Analyzer identifies a temp-table or ProDataSet mapping, it also provides a link to documentation containing an example SOAP message element, in case you prefer to work with the XML directly.

For more information on how to use ABL to work with complex data passed as a parameter, see the “Managing operation parameters” section on page 8–6.

**SOAP header entry and SOAP fault detail documentation**

For a SOAP header entry or SOAP fault detail, the WSDL Analyzer provides similar documentation for complex data. If it can identify a temp-table or ProDataSet that matches the SOAP header entry, it simply provides the appropriate static ABL definition. If a dynamic temp-table or ProDataSet mapping is possible, the WSDL Analyzer also documents this fact with any appropriate explanatory notes.

**Note:** SOAP headers rarely contain complex data that the WSDL Analyzer can map to a temp-table or ProDataSet.

For these SOAP elements, you never access a temp-table or ProDataSet directly. For a response header, you must access the header entry XML and use an ABL temp-table or ProDataSet READ-XML( ) method to transform the XML into temp-table or ProDataSet data, which you can then access as ABL data types. For a request header, you can create the header data in the corresponding temp-table or ProDataSet and transform it to the header entry XML using a temp-table or ProDataSet WRITE-XML( ) method. For SOAP fault detail, you must access the SOAP fault detail XML and use an ABL temp-table or ProDataSet READ-XML( ) method to transform the XML into temp-table or ProDataSet data, which you can then access as ABL data types. Again, if no temp-table or ProDataSet mapping is possible, you must use the ABL DOM or SAX features to access the XML for a header entry.

For more information on how to use ABL to work with complex data passed as a SOAP header entry, see Chapter 9, “Handling SOAP Message Headers in ABL.” For more information on how to use ABL to work with complex data passed as SOAP fault detail, see Chapter 10, “Handling Errors in ABL Requests to Web Services.”
Mapping XML Schema data types to ABL data types

The foundation for data types in WSDL is XML Schema, a standard that defines simple data types and a means to aggregate them into more complex types. OpenEdge supports a range of acceptable mappings for all XML Schema types. For a given parameter on a Web service operation, OpenEdge supports a range of ABL data types that can be automatically transformed between OpenEdge and the XML Schema representation. As part of the acceptable mappings, OpenEdge suggests (through the WSDL Analyzer) a recommended ABL data type to use when mapping an ABL parameter to a particular Web service parameter.

This section identifies the mapping options and how they work going from one domain (XML Schema) to the other (ABL) for the following types of data:

- Simple data types
- Arrays
- Complex data types

For more information on how to manipulate these data types (especially complex data types) in ABL, see Chapter 8, “Invoking Web Service Operations from ABL.”

Simple data types

Simple data types represent the foundation for all types of data. The mappings for simple data types and how they can be transformed between Web services and ABL provide the basic information for how to work with more complex types of data.

Suggested mappings

The WSDL Analyzer’s generated documentation provides recommended mappings between Web service parameters and ABL parameters. These are the mappings that are most semantically useful. Wherever the WSDL file indicates that a parameter of any data type is nillable, this means that on the Web service side the parameter can have the null value and on the ABL side the parameter can pass the ABL Unknown value (?).

Table 6–2 lists all the simple XML Schema data types in alphabetical order, showing for each one the suggested ABL data type for mapping it.

Table 6–2: Suggested XML Schema mappings to ABL data types (1 of 4)

<table>
<thead>
<tr>
<th>XML Schema data type</th>
<th>ABL data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>anyURI</td>
<td>CHARACTER</td>
<td>–</td>
</tr>
<tr>
<td>base64Binary</td>
<td>RAW</td>
<td>You can use the ABL BASE64–ENCODE and BASE64–DECODE functions to convert between Base 64 clear text (CHARACTER or LONGCHAR) and binary data (RAW or MEMPTR).</td>
</tr>
<tr>
<td>boolean</td>
<td>LOGICAL</td>
<td>–</td>
</tr>
<tr>
<td>byte</td>
<td>INTEGER</td>
<td>On INPUT, if the ABL INTEGER value is outside the valid range of the XML Schema byte type, it is a run-time error.</td>
</tr>
</tbody>
</table>
Creating an ABL Client from WSDL

On OUTPUT, any time zone information is lost. To retain it, use an ABL CHARACTER parameter.

The ABL DECIMAL type might not hold the entire XML Schema decimal value. However, the industry common practice is to map between XML Schema decimal and another decimal data type.

On OUTPUT, if the XML Schema decimal value overflows the ABL DECIMAL type, it is a run-time error.

On INPUT, if the ABL DECIMAL value is outside the valid range of the XML Schema integer type, it is a run-time error.

On OUTPUT, if the value of the XML Schema integer value is outside the valid range of the ABL DECIMAL type, it is a run-time error.

<table>
<thead>
<tr>
<th>XML Schema data type</th>
<th>ABL data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>DATE</td>
<td>On OUTPUT, any time zone information is lost. To retain it, use an ABL CHARACTER parameter.</td>
</tr>
<tr>
<td>dateTime</td>
<td>DATETIME-TZ</td>
<td></td>
</tr>
<tr>
<td>decimal</td>
<td>DECIMAL</td>
<td>The ABL DECIMAL type might not hold the entire XML Schema decimal value. However, the industry common practice is to map between XML Schema decimal and another decimal data type. On OUTPUT, if the XML Schema decimal value overflows the ABL DECIMAL type, it is a run-time error.</td>
</tr>
<tr>
<td>double</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>duration</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>ENTITIES</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>ENTITY</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>gDay</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>gMonth</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>gMonthDay</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>gYear</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>gYearMonth</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>hexBinary</td>
<td>RAW</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>IDREF</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>IDREFS</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>DECIMAL</td>
<td>The ABL DECIMAL type might not hold the entire XML Schema integer value, but it is a better choice than the ABL INTEGER type. On INPUT, if the ABL DECIMAL value is outside the valid range of the XML Schema integer type, it is a run-time error. On OUTPUT, if the value of the XML Schema integer value is outside the valid range of the ABL DECIMAL type, it is a run-time error.</td>
</tr>
</tbody>
</table>
### Table 6–2: Suggested XML Schema mappings to ABL data types

<table>
<thead>
<tr>
<th>XML Schema data type</th>
<th>ABL data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>CHARACTER</td>
<td>–</td>
</tr>
<tr>
<td>long</td>
<td>INT64</td>
<td>–</td>
</tr>
<tr>
<td>Name</td>
<td>CHARACTER</td>
<td>–</td>
</tr>
<tr>
<td>NCName</td>
<td>CHARACTER</td>
<td>–</td>
</tr>
<tr>
<td>negativeInteger</td>
<td>DECIMAL</td>
<td>The ABL DECIMAL type might not hold the entire XML Schema negativeInteger value, but it is a better choice than the ABL INTEGER type. On INPUT, if the ABL DECIMAL value is outside the valid range of the XML Schema negativeInteger type, it is a run-time error. On OUTPUT, if the value of the XML Schema negativeInteger value is outside the valid range of the ABL DECIMAL type, it is a run-time error.</td>
</tr>
<tr>
<td>NMTOKEN</td>
<td>CHARACTER</td>
<td>–</td>
</tr>
<tr>
<td>NMTOKENS</td>
<td>CHARACTER</td>
<td>–</td>
</tr>
<tr>
<td>nonNegativeInteger</td>
<td>DECIMAL</td>
<td>The ABL DECIMAL type might not hold the entire XML Schema nonNegativeInteger value, but it is a better choice than the ABL INTEGER type. On INPUT, if the ABL DECIMAL value is outside the valid range of the XML Schema nonNegativeInteger type, it is a run-time error. On OUTPUT, if the value of the XML Schema nonNegativeInteger value is outside the valid range of the ABL DECIMAL type, it is a run-time error.</td>
</tr>
<tr>
<td>nonPositiveInteger</td>
<td>DECIMAL</td>
<td>The ABL DECIMAL type may not hold the entire XML Schema nonPositiveInteger value, but it is a better choice than the ABL INTEGER type. On INPUT, if the ABL DECIMAL value is outside the valid range of the XML Schema nonPositiveInteger type, it is a run-time error. On OUTPUT, if the value of the XML Schema nonPositiveInteger value is outside the valid range of the ABL DECIMAL type, it is a run-time error.</td>
</tr>
<tr>
<td>normalizedString</td>
<td>CHARACTER</td>
<td>–</td>
</tr>
<tr>
<td>NOTATION</td>
<td>CHARACTER</td>
<td>–</td>
</tr>
</tbody>
</table>
Table 6–2: Suggested XML Schema mappings to ABL data types (4 of 4)

<table>
<thead>
<tr>
<th>XML Schema data type</th>
<th>ABL data type</th>
<th>Notes</th>
</tr>
</thead>
</table>
| positiveInteger      | DECIMAL       | The ABL DECIMAL type may not hold the entire XML Schema positiveInteger value, but it is a better choice than the ABL INTEGER type.  
  On INPUT, if the ABL DECIMAL value is outside the valid range of the XML Schema positiveInteger type, it is a run-time error.  
  On OUTPUT, if the value of the XML Schema positiveInteger value is outside the valid range of the ABL DECIMAL type, it is a run-time error. |
| qName                | CHARACTER     | --    |
| short                | INTEGER       | On INPUT, if the ABL INTEGER value is outside the valid range of the XML Schema short type, it is a run-time error. |
| string               | CHARACTER     | --    |
| time                 | INTEGER       | On INPUT, the value is converted with no time zone information. |
| token                | CHARACTER     | --    |
| unsignedByte         | INTEGER       | On INPUT, if the ABL INTEGER value is outside the valid range of the XML Schema unsignedByte type, it is a run-time error. |
| unsignedInt          | DECIMAL       | On INPUT, if the ABL DECIMAL value is outside the valid range of the XML Schema unsignedInt type, it is a run-time error.  
  On OUTPUT, if the value of the XML Schema unsignedInt value is outside the valid range of the ABL DECIMAL type, it is a run-time error. |
| unsignedLong         | DECIMAL       | On INPUT, if the ABL DECIMAL value is outside the valid range of the XML Schema unsignedLong type, it is a run-time error.  
  On OUTPUT, if the value of the XML Schema unsignedLong value is outside the valid range of the ABL DECIMAL type, it is a run-time error. |
| unsignedShort        | INTEGER       | On INPUT, if the ABL INTEGER value is outside the valid range of the XML Schema unsignedShort type, it is a run-time error. |

Note:  
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. For more information, see Appendix D, “Data Type Conversion Rules for ABL Calls to Web Services.”
Interacting with XML Schema data formats

You might need to work directly with an XML Schema value whose format is not represented among the supported formats for the suggested ABL data type mapping; for example, a 51-digit decimal value. To handle this requirement, you can pass an XML Schema-formatted string version of the value into or out of the Web service invocation. OpenEdge automatically passes the XML Schema-formatted value when you map either a CHARACTER or LONGCHAR data type to any XML Schema data type that has a suggested mapping (see Table 6–2) other than CHARACTER or LONGCHAR.

When you pass the XML Schema-formatted value as an INPUT parameter, the Web service invocation incorporates it directly into the generated SOAP message. For an OUTPUT parameter, the invocation copies the XML Scheme-formatted value directly from the SOAP message into the CHARACTER or LONGCHAR parameter.

When you pass an XML Schema-formatted value to a Web service, the invocation also validates the format of the value to ensure that it conforms to XML Schema formatting rules for the data type. Note that the invocation does not validate any facets declared in the <schema> element to constrain the value. It ensures that the format of the value you provide is valid according to its base XML Schema data type representation, but not, for example, if the value falls within a given range.

For example, if you pass an ABL CHARACTER as an INPUT parameter for an XML Schema int, the invocation checks to ensure that it contains a properly formatted 32-bit integer value. If you pass an ABL CHARACTER as an OUTPUT parameter for an XML Schema int, the invocation copies the Schema-formatted value directly from the SOAP message into the ABL CHARACTER parameter.

Arrays

OpenEdge maps arrays of simple types to ABL arrays. The WSDL Analyzer determines that it must map to an ABL array when it recognizes a Web service array declaration or the use of the minOccurs and maxOccurs XML Schema attributes in a parameter type definition. If a SOAP array is declared as unbounded, the WSDL Analyzer maps it to an ABL array with an indeterminate extent.

When it receives a SOAP response that includes a sparse array parameter, OpenEdge creates an ABL array of the appropriate size and fills it. OpenEdge sets each unused entry in the ABL array to the Unknown value (?).

OpenEdge never passes a sparse array as input to a Web service.

For arrays of complex types, the ABL represents them as one of the following data types, depending on the structure and content of the complex data, as determined by the WSDL Analyzer:

- A temp-table parameter, if the WSDL is for an OpenEdge Web service
- A ProDataSet parameter
- A LONGCHAR (LONGCHAR[ ]) or CHARACTER (CHARACTER[ ]) array parameter

Depending on how the WSDL Analyzer represents it, you might handle the entire array or individual elements as a complex data type (see the “Complex data types” section on page 6–22). For more information on how the WSDL Analyzer determines the representation of complex data, see the “Analyzing complex data” section on page 6–14.
Complex data types

Complex data types in XML appear in many forms, but share a common feature of representing multiple simple and other complex data types as an aggregate (group). One of the most common complex data types in XML is the XML Schema complex type. An XML Schema complex type is an aggregate of one or more simple types or complex types defined by an enclosing `<complexType>` XML Schema element.

When the WSDL Analyzer encounters a Web service operation parameter defined with a complex data type, it attempts to identify an ABL data type to represent the parameter that both maps to the parameter exactly and provides the easiest access to the data in ABL. Thus, the WSDL Analyzer might suggest any of the following parameter data types to represent:

- A static or dynamic temp-table (TABLE or TABLE-HANDLE) parameter, if the WSDL is for an OpenEdge Web service
- A static or dynamic ProDataSet (DATASET or DATASET-HANDLE) parameter
- A LONGCHAR or CHARACTER parameter, possibly in array form

Once in ABL, you can work with data from Web service complex data types using the features of the ABL data type used to represent it. For example, if the WSDL Analyzer determines that you can represent a Web service parameter as a temp-table or ProDataSet, you can access the data as a temp-table or ProDataSet with no need to manipulate the SOAP message XML. OpenEdge allows some flexibility in how the simple XML Schema data types of individual data elements map to the ABL fields in a temp-table. The casting rules for these mappings are similar to those for simple data type parameters (see the “Simple data types” section on page 6–17). For more information, see the sections on mapping between XML Schema and temp-table field data types, in OpenEdge Development: Programming Interfaces. For more information on how the WSDL Analyzer analyzes and documents complex data that maps to temp-tables and ProDataSets, see the “Analyzing complex data” section on page 6–14.

If the WSDL Analyzer can only represent the parameter as a LONGCHAR or CHARACTER (the most likely scenario), you must do one of the following:

- Directly manipulate the serialized XML, as specified by the WSDL Analyzer, in ABL CHARACTER or LONGCHAR variables.
- Convert the serialized XML to a parsed representation using the ABL SAX or DOM read and write methods. For more information, see OpenEdge Development: Programming Interfaces.

Whatever technique you use to work directly with the XML for a complex data type parameter in ABL, if you need to pass the parameter back as input to the Web service, you must pass it back as serialized XML. So, if you maintain the data for complex types in an ABL DOM tree (de-serialized) or read it as a stream using SAX, you must serialize it again to a string before passing it back to a Web service as a complex data type parameter. You can do this from a DOM tree, by using a DOM method to convert the DOM tree to a character string, and you can do this in SAX by using SAX write methods to create the XML character string that you use for input.

For more information on how to manage complex data types as input and output parameters of Web service operations and work with the data in ABL, see the “Managing operation parameters” section on page 8–6.
The first step in using a Web service is to connect (or bind) your application to it. Web service connections (or bindings) represent associations between your ABL application and the Web service, making it easier for you to access and manage your interaction with the Web service in ABL. They are not physical connections managed by your host or the network.

The following sections of this chapter describes the features of Web service connections and how you can create and manage them in ABL:

- What is a Web service connection?
- Binding to a Web service
- Managing Web service bindings

For more information on invoking operations in a Web service that you have connected, see Chapter 8, “Invoking Web Service Operations from ABL.”
What is a Web service connection?

Precisely speaking, there is no such thing as a Web service connection. All communications between a Web service client and the Web service is, from a network point of view, connectionless. That is, neither the client host nor the Web service host maintain any knowledge about each other except the other’s location, and then only when exchanging messages. Any context or state must be managed explicitly by the application.

However, from the client application point of view, there is definitely a defined relationship between itself and the Web service it is communicating with, and this relationship is more properly defined as a binding. A binding retains a logical connection between the client application and the Web service by maintaining a continuous association between specified client resources and specified Web service resources. This is the same type of “connection” that an ABL client application maintains between itself and a session-free AppServer with which it is communicating (see OpenEdge Application Server: Developing AppServer Applications).

This binding relationship makes it easier for the client to send requests to the Web service and to manage those requests within its own environment until such time as it has no more need for the bound Web service and its resources. At such a time, the client application can logically disconnect (unbind) the Web service from its environment in order to free up resources for other tasks.

While the Web service is bound, the process of invoking a Web service request includes telling the client host where to send the application message on the Internet so the Web service can receive and respond to it. This happens every time that the client invokes a Web service request because, as noted previously, the client host maintains no knowledge of the Web service host and its relationship to the client application. This lack of a physical connection provides performance benefits. Given sufficient AppServer resources, a client never needs to wait for an AppServer to service its Web service request, and whenever an AppServer completes a request, it is free to service a request from any other client. This arrangement offers maximum scalability for a given set of AppServer resources over an increasing client load.

To take further advantage of Web service availability on the network, OpenEdge allows the ABL client to make both synchronous and asynchronous requests to a Web service, providing greater flexibility in its own request management. For more information on asynchronous Web service requests, see the “Managing asynchronous requests” section on page 8–12.
Binding to a Web service

Binding to a Web service is a two-step process:

1. Create a server object handle.
2. Execute the CONNECT( ) method on the server object handle.

Creating a server object handle

To create a server handle for a Web service, you use the same procedure as creating one for an AppServer, as shown in the following example:

```plaintext
DEFINE VARIABLE hServer AS HANDLE.
CREATE SERVER hServer.
```

You can now use hServer to bind a Web service.

Binding to a server object handle

As for an AppServer, use the CONNECT( ) method on the server object handle to bind that server object handle to a Web service. This is the basic syntax:

**Syntax**

```plaintext
CONNECT ( connection-parameters )
```

A Web service binding requires and uses only the connection-parameters argument, which is a character string containing embedded parameter values to specify the binding. Note that the CONNECT( ) method is also used to connect AppServer application services. However, if the connection-parameters string contains any Web service parameters, OpenEdge assumes that the method is binding to a Web service and ignores any AppServer-specific connection parameters in the string.

Connection parameters

The primary function of the connection parameters is to specify the location and transport information for the Web service on the network. Web services provide two separate mechanisms for doing this. OpenEdge supports both mechanisms through the Web service connection parameters.
This is the Web service syntax for the connection-parameters string:

Syntax

```
-WSDL wsdl-document [ -WSDLUserid user-id [ -WSDLPassword password ] ]
{
    {
        [ -Service service-name [ -ServiceNamespace service-namespace ] ]
        -Port port-name
    }
    |
    {
        [ [ -Binding binding-name [ -BindingNamespace binding-namespace ] ]
        -SOAPEndpoint URL-endpoint ]
    }
}
-SOAPEndpointUserid user-id [ -SOAPEndpointPassword password ]
[TARGET_NAMESPACE targetNamespace]
-maxConnections num-connections
-connectionLifetime nSeconds
-nohostverify
-noSessionReuse
-pf filename
```

The only fully required element is the -WSDL parameter to identify the WSDL document for the Web service. This syntax might seem confusing, since the required elements, -Service and -Port or -Binding and -SOAPEndpoint, contain only optional elements. This situation arises because the WSDL file can directly supply some of this information.

Basically a complete binding can be specified by one of these mechanisms:

1. A valid service and port (transport and Web service URL location) specification (the most common mechanism)

2. A valid binding (transport) and SOAP endpoint (Web service URL location) specification (sometimes used to bind a SOAP viewer between the client and the Web service, or to provide a consortia of equivalent Web service locations)

The connection parameters only need to contain enough information to uniquely specify a target. If a WSDL only contains a single service, you can dispense with the -Service option. If a WSDL only contains a single port type, you can dispense with the -Port option. The same pattern applies for the -Binding/-SOAPEndpoint options. When a WSDL contains multiple choices for an option, the connection parameters must specify a unique value for that option.
A CONNECT( ) method fails, if any of the following happen:

- You include both valid -Service/-Port and valid -Binding/-SOAPEndpoint options in the connection parameters.
- The service, port, or binding options do not match the corresponding WSDL entries exactly with respect to namespace, local name, and letter case.
- The WSDL contains multiple services, ports, or bindings, and you fail to specify a valid, unique set of -Service/-Port or -Binding/-SOAPEndpoint options.

Table 7–1 provides a short description for each parameter in the Web service connection parameter string, in the order they appear in the syntax. For more information on these parameters, see the CONNECT( ) method entry in OpenEdge Development: ABL Reference.

### Table 7–1: Web service connection parameters

<table>
<thead>
<tr>
<th>Connection parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-WSDL wsdl-document</td>
<td>URL, UNC, or local file pathname (possibly relative) to the WSDL file that describes the Web service.</td>
</tr>
<tr>
<td>-WSDLUserid user-id</td>
<td>Username to authenticate access to the WSDL file (overridden by any username specified in the WSDL URL).</td>
</tr>
<tr>
<td>-WSDLPassword password</td>
<td>Password to authenticate access to the WSDL file (overridden by any password specified in the WSDL URL).</td>
</tr>
<tr>
<td>-Service service-name</td>
<td>Name of a &lt;service&gt; element in the WSDL.</td>
</tr>
<tr>
<td>-ServiceNamespace service-namespace</td>
<td>Namespace for service-name specified in the -Service parameter.</td>
</tr>
<tr>
<td>-Port port-name</td>
<td>Name of a &lt;port&gt; element defined in the specified &lt;service&gt; element.</td>
</tr>
<tr>
<td>-Binding binding-name</td>
<td>Name of a &lt;binding&gt; element in the WSDL.</td>
</tr>
<tr>
<td>-BindingNamespace binding-namespace</td>
<td>Namespace for binding-name specified in the -Binding parameter.</td>
</tr>
<tr>
<td>-SOAPEndpoint URL-endpoint</td>
<td>URL location for the Web service.</td>
</tr>
<tr>
<td>-SOAPEndpointUserid user-id</td>
<td>Username to authenticate access to the Web service endpoint (URL). This endpoint can be specified by the -SOAPEndpoint parameter. It can also be implied by the -Service parameter, the -Port parameter, or the -WSDL parameter (if there is only one service containing one port).</td>
</tr>
<tr>
<td>-SOAPEndpointPassword password</td>
<td>Password to authenticate access to the Web service endpoint.</td>
</tr>
</tbody>
</table>
Connecting to Web Services from ABL

When you use HTTPS to make a secure connection to a Web service, you might have to manage the OpenEdge certificate store to contain the necessary digital certificates for SSL verification. For information on managing the OpenEdge certificate store, see OpenEdge Getting Started: Core Business Services.

### Using HTTPS

<table>
<thead>
<tr>
<th>Connection parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-TargetNamespace <code>targetNamespace</code></td>
<td>The namespace that uniquely identifies a WSDL. It must match the target namespace declared in the document specified by the <code>-WSDL</code> parameter. This can be used as a verification check.</td>
</tr>
<tr>
<td>-maxConnections <code>num-connections</code></td>
<td>Maximum number of simultaneous (parallel) connections maintained between the client and Web service for asynchronous requests.</td>
</tr>
<tr>
<td>-connectionLifetime <code>nSeconds</code></td>
<td>The maximum number of seconds that a given connection can be reused for asynchronous requests before it is destroyed.</td>
</tr>
<tr>
<td>-nohostverify</td>
<td>If specified, turns off host verification for a Secure Socket Layer (SSL) connection using HTTPS. Without this parameter specified, the client compares the host name specified in the URL with the Common Name specified in the server certificate, and raises an error if they do not match. With this parameter specified, the client never raises the error. For more information, see the sections on managing the OpenEdge certificate store in OpenEdge Getting Started: Core Business Services.</td>
</tr>
<tr>
<td>-nosessionreuse</td>
<td>If specified, the connection does not reuse the SSL session ID when reconnecting to the same Web server using HTTPS.</td>
</tr>
<tr>
<td>-pf <code>filename</code></td>
<td>File containing any parameters specified in the CONNECT( ) method.</td>
</tr>
</tbody>
</table>

1. These connection parameters affect all SSL connections made using the Web service server handle, including any WSDL file access or Web service access that uses HTTPS.
Invoking the binding

The following example shows binding an actual Web service using service and port:

<table>
<thead>
<tr>
<th>Binding a Web service in ABL using service and port</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE VARIABLE hServer AS HANDLE.</td>
</tr>
<tr>
<td>DEFINE VARIABLE lReturn AS LOGICAL.</td>
</tr>
<tr>
<td>CREATE SERVER hServer.</td>
</tr>
<tr>
<td>lReturn = hServer:CONNECT(&quot;-WSDL <a href="http://wstraining.progress.com/wsa/wsa1/">http://wstraining.progress.com/wsa/wsa1/</a></td>
</tr>
<tr>
<td>wsd1?targetURI=urn:OpenEdgeServices:NewCoService</td>
</tr>
<tr>
<td>-Service NewCoServiceService</td>
</tr>
<tr>
<td>-Port NewCoServiceObj&quot;).</td>
</tr>
</tbody>
</table>

The following example shows what binding a Web service using binding and SOAP endpoint might look like:

<table>
<thead>
<tr>
<th>Binding a Web service in ABL using binding and SOAP endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE VARIABLE hServer AS HANDLE.</td>
</tr>
<tr>
<td>DEFINE VARIABLE lReturn AS LOGICAL.</td>
</tr>
<tr>
<td>CREATE SERVER hServer.</td>
</tr>
<tr>
<td>-Binding InventoryBinding</td>
</tr>
<tr>
<td>-SOAPEndpoint <a href="http://ws.acme.com:80soap/servlet/rpcrouter">http://ws.acme.com:80soap/servlet/rpcrouter</a>&quot;).</td>
</tr>
</tbody>
</table>

Binding results

After a successful attempt to bind a Web service, the server object is initialized for the Web service. This means that on the server object handle:

- The CONNECT( ) method returns the value, TRUE, during invocation
- The server object’s SUBTYPE attribute returns the value, "WEBSERVICE"
- The server object’s CONNECTED( ) method returns the value, TRUE

Note: A value of TRUE does not indicate the state of the HTTP/S connection between the client and Web service. Subsequent requests can fail if there is an HTTP failure at any point between the client and Web service. A TRUE value indicates that the WSDL document named in the -WSDL parameter was successfully read and any other parameter values (-Service, -Port, or -Binding) correctly match items in that WSDL document.

An attempt to bind a Web service can fail for any of the following reasons:

- A corrupt WSDL file
- A parameter specification in the connection-parameters string is invalid
- The value specified for the -TargetNamespace parameter does not match the target namespace specified in the WSDL document. (This is a version check.)
- The specified WSDL document is not found
- The value for the -WSDLuserid or the -WSDLpassword parameter is invalid
Managing Web service bindings

Once you bind a Web service to a server object handle using the CONNECT( ) method, the Web service remains bound (server-handle:CONNECTED( ) = TRUE) until you invoke the DISCONNECT( ) method on that handle to unbind it. Between these two points you can access the operations defined for the Web service.

To work with additional port types in the same Web service or in different Web services, you must create a separate Web service binding for each port type you want to access. Depending on how you choose to manage Web service binding resources, you can do all this using one or more pairs of server and procedure handles. If your application accesses many port types and Web services, especially if it disconnects and connects Web service bindings frequently, be sure to delete the created objects (especially the procedure objects) when you no longer need them in order to prevent memory leaks in your application.

Accessing a port type for a Web service

While the Web service is bound to a server object, you can create and map a procedure object to the port type supported by the binding in order to invoke its Web service operations using this syntax:

Syntax

```plaintext
RUN portTypeName [ SET hPortType ] ON SERVER hWebService .
```

As a result of executing a RUN statement using this syntax, you can access operations defined by the specified port type (portTypeName) in the Web service mapped to the server object handle, hWebService. You can invoke these operations by executing them as procedures or user-defined functions on the procedure object handle, hPortType, which is now mapped to the port type, portTypeName.

The RUN statement accomplishes this mapping by creating a procedure object for the port type and attaching it to the specified procedure object handle:

Accessing a port type for a Web service

```plaintext
DEFINE VARIABLE hServer AS HANDLE.
DEFINE VARIABLE hPortType AS HANDLE.
CREATE SERVER hServer.

RUN NewCoServiceObj SET hPortType IN hServer.
```

You can now invoke and manage all of the operations in the port type, NewCoServiceObj, defined for the Web service, NewCoServiceService. For more information on using procedure handles to invoke Web service operations, see the “Invoking operations” section on page 8–3.
Managing Web service bindings

Accessing multiple port types

If your client makes use of multiple port types from one or more Web services, you can either reuse a single server object or create multiple server objects. The first option limits the resources tied up in server objects. The second option enables you to access multiple port types simultaneously. You must decide which approach best suits your needs.

To reuse a server object to access a different port type, you must first disconnect the server object from its current binding.

- **To access a different port type in a bound Web service using the same server object handle:**
  1. Delete the procedure object mapped to the current port type, using the `DELETE PROCEDURE` or `DELETE OBJECT` statement.
  2. Unbind (logically disconnect) the server object handle from its current Web service binding using the `DISCONNECT()` method.
  3. Using the `CONNECT()` method on the same server object handle, bind the server object to the Web service using a binding that supports the next port type.
  4. Create and map a new procedure object to the next port type, using the `RUN` statement with the same server object handle, and, if you want, setting the same procedure object handle used to map the previous port type.

You can repeat this procedure for each port type you need to access in the same Web service, or you can map all of the Web service port types at one time using multiple server object and procedure object handles.

Accessing multiple port types simultaneously

If you want to maintain access to two or more port types at the same time (for the same or a different Web service), you must create and bind a Web service to separate server object handles for each port type that you want to access. You can then create and map separate procedure object handles to each port type.
Invoking Web Service Operations from ABL

Invoking a Web service operation can be a simple process of returning a value and using it directly, or it can require a more complex treatment. The more complex treatment can include interacting with the SOAP message headers, handling SOAP faults, parsing and building complex parameters, or simply managing the serialized XML of a simple parameter.

The following sections describe how to invoke and manage the data associated with an operation for all of these requirements:

- Preparing to invoke operations
- Invoking operations
- Managing operation parameters
- Managing complex data
- Managing asynchronous requests
Invoking Web Service Operations from ABL

Preparing to invoke operations

The ABL interface to Web service operations is a type of proxy procedure object that you map to a port type in a Web service binding (see Chapter 7, “Connecting to Web Services from ABL”). This Web service procedure object provides access to port type’s operations much like a proxy procedure object provides access to remote internal procedures and user-defined functions defined by a persistent procedure instantiated on an AppServer (see OpenEdge Application Server: Developing AppServer Applications).

Creating a Web service procedure object

You map the port type to the procedure object using a syntax similar to how you instantiate a remote persistent procedure:

Syntax

```
RUN portTypeName [ SET hPortType ] ON SERVER hWebService [ NO-ERROR ].
```

For an ABL remote persistent procedure, you actually execute the procedure by name persistently; for a Web service port type, you invoke the operation using the name of the port type from the WSDL (portTypeName) that defines the operations you want to run. Otherwise, you can set and access a Web service proxy procedure handle (hPortType) the same way as for a remote persistent procedure. Like an AppServer, you use the server object handle bound to the Web service (hWebService) to indicate the remote location to receive Web service requests.

Note the lack of parameters in this statement. Where you might pass parameters when instantiating a remote persistent procedure, you cannot pass any parameters when creating a Web service procedure object, and the AVM raises an error if you do so.

All other attributes and methods on a Web service procedure object work much like they do for any proxy procedure object. Any attribute or method that has no meaning in the context of a Web service returns the Unknown value (?) or an empty string.

Note: A Web service procedure object supports an additional method, SET-CALLBACK-PROCEDURE( ), that is not used on a proxy procedure object. Callback procedures are internal procedures that access information in a SOAP header. For more information on callback procedures, see the “Specifying SOAP header callback procedures at run time” section on page 9–6.
Invoking operations

After you have created a Web service procedure object, you can invoke the operations made available through that object. OpenEdge identifies Web service operations in the ABL by the same names and signatures used to define them in the WSDL file for the Web service, which you specify when you bind the Web service using the CONNECT() method (see Chapter 7, “Connecting to Web Services from ABL”).

The WSDL Analyzer determines if you can invoke the operation using only an ABL RUN statement, or if you can also invoke it with a user-defined function call:

- If the Analyzer determines that the operation does not return a value, it defines the operation as an internal procedure that you must call using the RUN statement.

- If the Analyzer determines that the operation returns a value, it defines the operation in both of the following ways:
  a. As a user-defined function that you forward reference and invoke as a function
  b. As an internal procedure that you can call using the RUN statement

**Note:** You cannot invoke native ABL procedures or user-defined functions interchangeably as one or the other. This feature only applies to Web service operations, and only those Web service operations as specified by the WSDL Analyzer.

The WSDL includes information that indicates if you can model a Web service as a user-defined function. If the WSDL does not indicate that or if there is a restriction in ABL that stops the operation from being called as a user-defined function, the WSDL Analyzer does not create a sample user-defined function. Any attempt to invoke a Web service operation using a method not specified by the WSDL Analyzer returns an error.
Figure 8–1 shows a sample operation listing from the WSDL Analyzer’s Port type page that can be used as either an internal procedure or as a user-defined function.

### Operation (internal procedure) detail

#### CustomerName

No documentation found in WSDL.

**Procedure prototype**

```abl
PROCEDURE CustomerName:
  DEFINE INPUT PARAMETER IdCustomer AS INTEGER NO-INDO.
  DEFINE OUTPUT PARAMETER result AS CHARACTER NO-INDO.
END PROCEDURE.
```

**Function prototype**

```abl
FUNCTION CustomerName RETURNS CHARACTER
  (INPUT IdCustomer AS INTEGER).
END FUNCTION.
```

#### Example

```abl
DEFINE VARIABLE IdCustomer AS INTEGER NO-INDO.
DEFINE VARIABLE result AS CHARACTER NO-INDO.

FUNCTION CustomerName RETURNS CHARACTER
  (INPUT IdCustomer AS INTEGER).
END FUNCTION.

/* Function invocation of CustomerName operation. */
result = CustomerName(IdCustomer).

/* Procedure invocation of CustomerName operation. */
RUN CustomerName IN hFunctionObject(INPUT IdCustomer, OUTPUT result).
```

**Parameters**

- `IdCustomer`
  - This value is defined as an XML Schema int type.
- `result`
  - This value is defined as an XML Schema string type.

**Returns**

See description of the result parameter.

---

**Figure 8–1:** WSDL Analyzer sample operation call

### Using the RUN statement

This is the general syntax for invoking Web service operations using the `RUN` statement:

**Syntax**

```abl
RUN operationName IN hPortType [ ASYNCHRONOUS ]
  [ ( parameter [ , parameter ] ... ) ]
  [ NO-ERROR ].
```
As indicated, this statement invokes a Web service operation with the specified name \( \text{operationName} \) defined in the port type mapped to the specified procedure object handle \( \text{hPortType} \), passing any parameters with data types specified in the order they are defined for the operation by the WSDL Analyzer. For an example of this syntax, see the sample WSDL Analyzer code in Figure 8–1.

Also, you can invoke any operation using the \texttt{RUN} statement either synchronously or asynchronously. For more information on asynchronous invocation for Web service operations, see the “Managing asynchronous requests” section on page 8–12. For complete information on the \texttt{RUN} statement syntax, see \textit{OpenEdge Development: ABL Reference}.

\section*{Using a user-defined function}

For a user-defined function invocation, you must first forward reference a prototype mapped to the appropriate Web service procedure handle definition, exactly as you do for a remote user-defined function prototype mapped to a Web service proxy procedure handle.

This is the general syntax for the forward reference to a function prototype:

\begin{verbatim}
FUNCTION operationName [ RETURNS ] dataType
    [ ( parameter [ , parameter ] ... ) ]
IN hPortType.
\end{verbatim}

This statement invokes a Web service operation with the specified name \( \text{operationName} \) defined in the port type mapped to the specified procedure object handle \( \text{hPortType} \), passing any parameters with data types specified in the order they are defined for the operation by the WSDL Analyzer, and passing the operation return parameter with the corresponding data type \( \text{dataType} \).

This is the syntax for simple invocation of one user-defined function in a single statement, the same for both Web service operations and user-defined functions:

\begin{verbatim}
return = operationName [ ( parameter [ , parameter ] ... ) ].
\end{verbatim}

This statement invokes a user-defined function with the specified name \( \text{operationName} \), passing any parameters with data types specified in the order they are defined for the operation by the WSDL Analyzer, and passing the operation return parameter as the value of the function, which is assigned to a variable of compatible data type. For an example of this syntax, see the sample WSDL Analyzer code in Figure 8–1.

As with any function, you can invoke it in any ABL statement where you need to use its value. However, you can only invoke a Web service user-defined function synchronously. For complete information on the \texttt{FUNCTION} statement prototype syntax, see \textit{OpenEdge Development: ABL Reference}. 

\textbf{Syntax}
Managing operation parameters

As described in the “Data type casting” section on page D–2, the data types that you specify for parameters in the ABL invocations of Web service operations must conform to a set of OpenEdge-supported castings established for corresponding XML Schema data types. The OpenEdge WSDL Analyzer provides all of the information you need to build interfaces between your ABL application and the Web service by documenting all of the Web service operations and how they can be mapped to the ABL, including the suggested (or recommended) ABL data types to use for XML Schema simple data type parameters.

Depending on the results you want to achieve, you might use one of the alternative castings supported by OpenEdge between XML Schema simple data types and ABL data types. Consistent with compatible data types in the ABL, OpenEdge implicitly converts between any XML Schema data type and any ABL data type included in the supported casting. If OpenEdge cannot complete the specified conversion, it generates a run-time error.

You can also map all XML Schema simple data types to the ABL CHARACTER or LONGCHAR type, because the SOAP messages used to exchange parameter values are XML character data. For XML Schema simple data types where CHARACTER is not the suggested ABL data type, using a CHARACTER or LONGCHAR causes OpenEdge to assume that you are passing the XML Schema-formatted string (the actual string value for the SOAP message) as the value for the parameter. This allows you to perform your own transformation on the value within your ABL application and to manage the value as you see fit. In doing this, you must format the CHARACTER or LONGCHAR values properly for any INPUT parameters to conform with the XML Schema data type defined for the Web service parameter by the WSDL. The only processing that OpenEdge does when passing XML Schema-formatted values is to perform any code page conversion required on the character data (for example, between the CPINTERNAL code page setting and UTF-8) and to validate that the format of the data is acceptable for the XML Schema data type. OpenEdge does no range or value checking that might be specified in the WSDL file.

XML Schema complex types are data types that contain multiple XML Schema data elements, possibly including other complex types. You can map complex types to either an ABL TABLE or DATASET parameter. To manage complex types that cannot be mapped to either an ABL temp-table or ProDataSet, you must work with the serialized XML directly, typically using the XML parsers in the ABL. For more information, see the “Managing complex data” section on page 8–7.

In general, if the invocation of a Web service operation fails for any reason, OpenEdge sets all OUTPUT and function return values to the Unknown value (?).
Managing complex data

Wherever you encounter complex data, for a Web service, you have these basic options for accessing and managing the data, depending on the documentation generated by the WSDL Analyzer:

- As an ABL temp-table or ProDataSet
- As a serialized XML string
- As individual simple data types

For more information on how the Analyzer determines the options you can use, see the “Analyzing complex data” section on page 6–14.

**Note:** The examples in this section are not available on the Documentation and Samples (doc_samples) directory of the OpenEdge product DVD or Progress Documentation Web site.

Types of complex data

Complex data that you might need to access includes:

- Complex data type parameters, including complex arrays
- SOAP header entries
- SOAP fault detail

**Complex data type parameters**

If the Analyzer specifies that you can use a temp-table or ProDataSet to access a Web service parameter, you can access (for OUTPUT) and write to (for INPUT) the parameter exactly as you do for any temp-table or ProDataSet parameter in the ABL. If the Analyzer provides the temp-table or ProDataSet definition, you can use a static instance and specify the parameter as a TABLE or DATASET. For a static OUTPUT parameter, OpenEdge ensures that the object is created and filled from the parameter element in the SOAP response message. For static INPUT parameter, you must create and fill the object, as appropriate, before passing the parameter.

If you must access the parameter as a dynamic temp-table or ProDataSet, you can specify the parameter as a TABLE-HANDLE or DATASET-HANDLE. For a dynamic OUTPUT parameter, OpenEdge ensures that the object is created and filled from the parameter element in the SOAP response message. For dynamic INPUT parameter, you must create and fill the object, as appropriate, before passing the parameter.

For an OUTPUT parameter that you access as a serialized XML string, you can work with serialized XML in the CHARACTER or LONCHAR parameter directly, using ABL string-manipulation statements and functions, or work with it in its parsed form using the ABL SAX reader or DOM parser. In any case, OpenEdge provides the entire <complexType> element for the value, exactly as it appears in the SOAP response message.

For an INPUT parameter that you write as a serialized XML string, you can build or maintain the value using the ABL SAX writer or DOM parser, and save the result as a LONCHAR value when you are ready to pass it as an INPUT parameter to the Web service.
For more information on using the SAX and DOM features in ABL, see *OpenEdge Development: Programming Interfaces*. For an example of how to handle complex data parameters in ABL, see the “Complex data example” section on page 8–8.

**Complex data in SOAP headers and SOAP faults**

You can also use techniques available for managing complex data parameters to access and manage the data for SOAP header entries and SOAP fault detail elements. You can get and set this data either as serialized character strings or as DOM trees. For more information, see Chapter 9, “Handling SOAP Message Headers in ABL” and Chapter 10, “Handling Errors in ABL Requests to Web Services.”

**Complex data example**

The following examples show how you might manage a complex parameter, or any complex data, in this case, an OUTPUT parameter as serialized XML. This is a procedure that maps to a Web service operation, `getAddress`. Given a social security number (`ssn`), the operation returns an address (`Address`) as a complex type:

**ABL prototype with a complex parameter accessed as serialized XML**

```abl
PROCEDURE getAddress:
  DEFINE INPUT PARAMETER ssn AS CHARACTER.
  DEFINE OUTPUT PARAMETER cmAddress AS LONGCHAR.
END PROCEDURE.
```

This is the schema for a `<complexType>` element that returns the address information to the caller. It contains five `xsd:string` data type elements representing the components of the address:

**Complex type in the WSDL**

```xml
<complexType name="Address">
  <sequence>
    <element name="name" type="xsd:string">
    <element name="street" type="xsd:string">
    <element name="city" type="xsd:string">
    <element name="state" type="xsd:string">
    <element name="zip-code" type="xsd:string">
  </sequence>
</complexType>
```

This sample ABL procedure demonstrates how you can manage this complex type in ABL as a DOM tree. The variable to receive the parameter value, `cmAddress`, is defined as a `LONGCHAR`. After the Web service operation returns a value for `cmAddress`, the `LOAD( )` method on the x-document handle, `hXDoc`, parses and loads the `<complexType>` element from `cmAddress` into the associated x-document object.

Because the schema of the complex type is known, the remaining x-document and x-noderef handle methods simply retrieve the root node from the “Address” DOM tree, and pick off the component value (text element) for each of the five component nodes that comprise the complex type, in order, assigning them to the corresponding fields of a database record.
This is the ABL example for handling the parameter as a DOM tree:

**Complex type managed in ABL as a DOM tree**

```abl
DEFINE VARIABLE hwS AS HANDLE.
DEFINE VARIABLE hAddrPortType AS HANDLE.
DEFINE VARIABLE cmAddress AS LONGCHAR.

CREATE SERVER hwS.

hwS:CONNECT ("-WSDL http://www.zzzcompany.org/ssn.wsdl
 -Service addressSVC
 -Port addressPort").

RUN addressPortType SET hAddrPortType ON SERVER hwS.

RUN getAddress IN hAddrPortType (INPUT "555-55-5555", OUTPUT cmAddress).

DEFINE VARIABLE hXDoc as HANDLE.
DEFINE VARIABLE hXRoot as HANDLE.
DEFINE VARIABLE hXNode as HANDLE.
DEFINE VARIABLE hXText as HANDLE.

CREATE X-DOCUMENT hXDoc.
CREATE X-NODEREF hXRoot.
CREATE X-NODEREF hXNode.
CREATE X-NODEREF hXText.

hXDoc:LOAD("LONGCHAR", cmAddress, FALSE).

hXDoc:GET-DOCUMENT-ELEMENT(hXRoot).

/* because we know the content, we are just moving straight ahead
 and getting each one of the nodes under the root, then getting its
 TEXT node to get the data we're interested in. */

hXRoot:GET-CHILD(hXNode, 1).

hXNode:GET-CHILD(hXText, 1).

/* let's assume we have a DB table with the appropriate fields */

myTable.name = hXText:NODE-VALUE.

/* ... */

hXRoot:GET-CHILD(hXNode, 5).

hXNode:GET-CHILD(hXText, 1).

myTable.zip-code = hXText:NODE-VALUE.

/* clean up */
/* ... */

hwS:DISCONNECT( ).
```
Coding options for wrapped document literal

For Wrapped Doc/Lit operations, ABL allows you to use one or both of two alternate forms for accessing the operation INPUT and OUTPUT parameters using ABL procedures and functions. You can either access them in wrapped form as XML strings containing the complex data for each SOAP message (request and response) or you can access them in unwrapped form as several ABL parameters, in many cases, with simple ABL data types. You can overload these forms. That is, you can use both forms for the same operation in the same client session.

**Note:** In most cases, you will find the unwrapped form easier to use. Using the wrapped form is supported for backwards compatibility.

Using the wrapped form

Suppose you have a Wrapped Doc/Lit operation, `foo`. The one complex input parameter contains a single element with two child elements, `bBoolean` and `iInteger`. The one complex output parameter also contains a single element with two child elements, `iInteger` and `cString`. Using the wrapped form, you might run the `foo` procedure with the INPUT parameter (`param1`) set to the XML string as shown in this example:

```
Wrapped Doc/Lit operation—wrapped form
```

```
DEFINE VARIABLE param1 AS LONGCHAR NO-UNDO.
DEFINE VARIABLE param2 AS LONGCHAR NO-UNDO.

.param1 = "
<ns0:foo xmlns:ns0='http://tempuri.org/'>
    <ns0:bBoolean>true</ns0:bBoolean>
    <ns0:iInteger>17</ns0:iInteger>
</ns0:foo>

RUN foo IN hWebService (INPUT param1, OUTPUT param2).

DISPLAY "Returned from web service call:" SKIP
    " param2 = " SKIP param2 NO-LABEL FORMAT "x(200)".
```

On the return, you might display an XML string in the OUTPUT parameter (`param2`) as follows:

```
On the return, you might display an XML string in the OUTPUT parameter (param2) as follows:

Returned from web service call:
    param2 =
    <ns0:fooResponse xmlns:ns0='http://tempuri.org/'>
        <ns0:iInteger>18</ns0:iInteger>
        <ns0:cString>Hello world!</ns0:cString>
    </ns0:fooResponse>
```

Clearly, to work with the individual values contained in these parameters, you must treat them as complex data (see the “Complex data example” section on page 8–8).
**Using the unwrapped form**

Using the unwrapped form for the same Wrapped Doc/Lit operation, `foo`, you can access all the parameter values individually using ABL data types. In this case, the WSDL Analyzer recognizes that there is a single INTEGER mapping for a value in both complex input and output parameters of the wrapped operation. So, it prescribes a single INPUT-OUTPUT parameter for that INTEGER mapping (iInteger). It prescribes individual INPUT and OUTPUT modes for the remaining values, as indicated in this example:

**Wrapped Doc/Lit operation—unwrapped form**

```abl
DEFINE VARIABLE bBoolean AS LOGICAL NO-UNDO.
DEFINE VARIABLE iInteger AS INTEGER NO-UNDO.
DEFINE VARIABLE cString AS CHARACTER NO-UNDO.

bBoolean = true.
iInteger = 17.

RUN foo IN hWebService (INPUT bBoolean, INPUT-OUTPUT iInteger,
OUTPUT cString).

DISPLAY "Returned from web service call:" SKIP
   " iInteger = " iInteger NO-LABEL SKIP
   " cString = " cString NO-LABEL FORMAT "x(12)".
```

On return, you might display the individual OUTPUT values (iInteger and cString) as follows:

```
Returned from web service call:
iInteger = 18
   cString = Hello world!
```

No further work is required to access these values in ABL.
Managing asynchronous requests

As indicated previously, you can invoke Web service requests asynchronously from ABL. This means that you can invoke a Web service request in such a way that the result is handled independently of the mainline flow of execution. That is, ABL statements following the asynchronous request execute immediately and in order without waiting for the result of the asynchronous request to be available. Instead, the asynchronous request specifies an internal event procedure to handle the result when the result for that asynchronous request becomes available, which is signalled by the activation of a PROCEDURE-COMPLETE event.

When it is time to handle the PROCEDURE-COMPLETE event, the specified event procedure executes and manages any OUTPUT or INPUT-OUTPUT parameters that were passed to the asynchronous request when it was invoked. These parameters are returned as INPUT parameters to the event procedure, which can store the parameter values or otherwise process the result for use by the mainline program. The mainline program can periodically inspect an asynchronous request object handle (also available to the event procedure), which it sets during invocation of the request, in order to determine if the specified asynchronous request has completed. If the request has completed, the program can then make use of the results as provided by the internal event procedure that handled them.

The model for asynchronous Web service request invocation is very similar to the model for asynchronous remote procedure invocation on a session-free AppServer. This section describes the asynchronous Web service calling model, noting any differences from the asynchronous session-free AppServer calling model. For information on the model for invoking asynchronous requests on a session-free AppServer, see *OpenEdge Application Server: Developing AppServer Applications*.

**Note:** For certain features that support Web service requests in ABL (for example, error handling), the feature behavior might differ depending on whether the request is synchronous or asynchronous.

### Supported asynchronous Web service requests

You can only invoke an asynchronous Web service request as a procedure using the RUN statement. This means that if a Web service operation is invoked as a user-defined function, you can only invoke it synchronously, because all operations that have a return value can only be invoked as user-defined functions, where the value must be made available at the point of invocation. This is consistent with how OpenEdge supports the invocation of asynchronous requests on the AppServer (remote procedures only).

### Order of completion

Similar to a session-free AppServer request, the order of completion for an asynchronous Web service request is determined solely by the Web service. The manner in which the ABL client send and response queues for an asynchronous request handle messages is identical for both session-free asynchronous AppServer requests and asynchronous Web service requests. That is, if ABL invokes multiple asynchronous Web service requests, the order of their completion cannot be determined on the Web service or session-free AppServer client. So, you must write your mainline program, and indeed your asynchronous event procedures, to make no assumptions about the order in which multiple asynchronous Web service requests complete.
A subtle distinction between a session-free asynchronous AppServer request and an asynchronous Web service request is the SOAP response callback procedure available for the Web service request. If a SOAP response callback procedure is set for an asynchronous Web service request, it executes after the SOAP response message arrives at the client but before the asynchronous event procedure executes to handle the PROCEDURE-COMPLETE event for the request.

**Asynchronous request object handle**

OpenEdge maintains an asynchronous request object for each asynchronous request, which maintains the status of the request, including whether and how it has completed (successfully or unsuccessfully). The handle to this asynchronous request object is available to the mainline program and to the event procedure for an asynchronous Web service request in the same way as for an asynchronous AppServer request. The Web service server object handle maintains its own chain of pending asynchronous request object handles (FIRST-ASYNC-REQUEST and LAST-ASYNC-REQUEST attributes), and the PROCEDURE-NAME attribute on each asynchronous request object handle returns the name of its Web service operation as it returns the name of the remote procedure executed for an asynchronous AppServer request.

Table 8–1 shows attributes on the asynchronous object handle that have special meaning for Web services.

**Table 8–1: Attributes on asynchronous request object handle**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning for Web service requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>Returns TRUE within the context of the event procedure when:</td>
</tr>
<tr>
<td></td>
<td>• You invoke the RETURN ERROR statement within the context of an associated SOAP response header callback procedure.</td>
</tr>
<tr>
<td></td>
<td>• A SOAP fault is returned for the asynchronous Web service request.</td>
</tr>
<tr>
<td>NEXT-SIBLING and</td>
<td>Maintains a chain of asynchronous request object handles for all pending asynchronous requests invoked in the same Web service</td>
</tr>
<tr>
<td>PREV-SIBLING</td>
<td></td>
</tr>
<tr>
<td>PERSISTENT-PROCEDURE</td>
<td>Returns a handle to the procedure object bound to the Web service port type</td>
</tr>
<tr>
<td>PROCEDURE-NAME</td>
<td>Returns the name of the Web service operation invoked asynchronously</td>
</tr>
<tr>
<td>QUIT</td>
<td>Always returns FALSE for an asynchronous Web service request</td>
</tr>
<tr>
<td>SERVER</td>
<td>Returns a handle to the server object for the Web service where the asynchronous request is executed</td>
</tr>
</tbody>
</table>
Results handling

A few differences exist in the results returned for an asynchronous Web service request compared to an asynchronous AppServer request, as shown in Table 8–2.

Table 8–2: Results for asynchronous Web service requests

<table>
<thead>
<tr>
<th>ABL result elements</th>
<th>Web service request result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR-STATUS system handle</td>
<td>If the ERROR attribute of the asynchronous request object handle is set to TRUE because of a SOAP fault, the ERROR-STATUS:ERROR-OBJECT-DETAIL attribute references a SOAP fault object with information on the SOAP fault.</td>
</tr>
<tr>
<td>RETURN-VALUE function</td>
<td>The value for ABL RETURN-VALUE function is never altered directly by the response from a Web service asynchronous request.</td>
</tr>
<tr>
<td>STOP condition</td>
<td>The client can raise the STOP condition by executing the CANCEL-REQUESTS() method on the Web service server object handle. All outstanding Web service requests are handled in a similar manner to those on an AppServer, except that for any currently executing Web service requests, the result is undefined. OpenEdge disconnects from all ports to currently executing Web service requests, and the exact response depends on how the Web service responds to the lost connection.</td>
</tr>
</tbody>
</table>
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 the “Analyzing complex data” section on page 6–14.

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:

- 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: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 ...>
    ...</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 the “Specifying SOAP header callback procedures at run time” section on page 9–6.
- The general cycle of operation for header handlers in ABL, see the “Invoking a header handler” section on page 9–11.
- Defining the callback procedures for header handlers, see the “Defining header handlers” section on page 9–8.
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 9–1 shows the relationship between the SOAP header object model and the parsed XML DOM tree using the SOAP header described in the previous 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.

![SOAP header object model diagram](image)

**Figure 9–1: 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 Figure 9–1, 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 Figure 9–2.

**Figure 9–2: 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 the “Creating and managing SOAP message headers” section on page 9–13.
In 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.

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.

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 \texttt{RUN portTypeName} statement to create an additional Web service procedure object for the same port type. You can then invoke the \texttt{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 \texttt{SET-CALLBACK-PROCEDURE( )} method:

**Syntax**

\[
\text{SET-CALLBACK-PROCEDURE (callback-name, internal-procedure [ , procedure-context ])}
\]

The parameters include:

- \textit{callback-name} — A character string value of "REQUEST-HEADER" or "RESPONSE-HEADER" to identify the type of callback.

- \textit{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 (""").

- \textit{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 \texttt{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 \texttt{PROCEDURE-COMPLETE} event for the request and any results have been returned to the \texttt{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 for handling the SOAP response message header and another callback procedure for handling the SOAP request message header for the operation. This is the syntax for defining the signatures for these two procedures:

- SOAP response header callback procedure:

```
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:

```
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-procname` — The name of a response header handler procedure. Specified as a character expression to the SET-CALLBACK-PROCEDURE( ) method along with the "RESPONSE-CALLBACK" setting.

- `request-header-procname` — The name of a request header handler procedure. Specified as a character expression to the SET-CALLBACK-PROCEDURE( ) method along with the "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.
Defining header handlers

- **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 the “Managing memory for SOAP headers” section on page 9–26.

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 depend 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 the “Using the **SET-CALLBACK-PROCEDURE( )** method” section on page 9–7.
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

```abl
/*** 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 the “SOAP header structure” section on page 9–2, 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**

```abl
/*
 * 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 the “SOAP header object model” section on page 9–4), 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:

- Reusing an unchanged SOAP response header
- Modifying a reused SOAP response header
- Using a client-created SOAP request header

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 (1 of 2)

```
/* SOAPHeader1.p
* Calls a ficticious 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.
```
Creating and managing SOAP message headers

Invoking a request that reuses an unchanged SOAP response header

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 the “Response header handler for returning a header for reuse” section on page 9–15).

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 the “Request header handler for reusing a header” section on page 9–16).

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.
Handling SOAP Message Headers in ABL

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 the “Reusing an unchanged SOAP response header” section on page 9–14). 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 the “Response header handler for returning a header for reuse” section on page 9–15).

**Invoking a request that modifies a reused SOAP response header**

```abl
/* SOAPHeader2.p
An addition to SOAPHeader1.p.
Calls a fictitious 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).
DISPAY cResponse LABEL "WS response" WITH FRAME aaa.
/*4*/
/* Go again with the AccessID set from previous response */
cResponse = "".
RUN HelloWorld IN hPortType (OUTPUT cResponse).
DISPAY 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).
DISPAY cResponse LABEL "WS response" WITH FRAME ccc.
```
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 the “Request header handler for reusing and modifying a header” section on page 9–19).

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 the “Request header handler for reusing and modifying a header” section on page 9–19).

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:

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

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

Invoking a request that modifies a reused 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 1DeleteOnDone 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 = ?.
        1DeleteOnDone = TRUE.
    END.
    ELSE DO: /*2b*/ /* all subsequent requests */
        hHeader = g_header.
        1DeleteOnDone = FALSE.
    /* Password node data are added to the existing SOAP header if a secure operation is being executed */
```

(2 of 2)
Handling SOAP Message Headers in ABL

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
Handling SOAP Message Headers in ABL

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**

```xml
<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 the “Procedure to create a SOAP request header” section on page 9–23).
3. Registers the request header (ReqHandler) handler after connecting to the Web service and instantiating the HeaderSoap port type procedure object.
4. Runs the HelloWorld procedure to invoke a Web service operation, passing back the global SOAP request header created by the client (see the “Request header handler for passing a globally-created header object” section on page 9–25).
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 (1 of 2)**

```abl
/* SOAPHeader3.p
   Calls a fictitious 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. */

/ */
/* 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)".
```
Invoking a request that creates a SOAP request header on the client

```
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:

```
/*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).
```
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 \(<\text{AuthHeader}\>\) 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 \(<\text{UserName}\>\) element as a child node of the header entry

8. Creates the \(<\text{Password}\>\) 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 \(<\text{AuthHeader}\>\) element of the x-document object using the \(\text{SET-NODE( )}\) method

**Note:** You must call the \(\text{ADD-HEADER-ENTRY()}\) method before calling the \(\text{SET-NODE()}\) 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 (\(\text{ReqHandler}\)) 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**

```method
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 \(\text{HelloMyWorld}\) 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 the “SOAP header object model” section on page 9–4):

- 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 Table 9–5) 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.
Managing memory for SOAP headers

Table 9–1 summarizes the available ABL elements and how they can help you to manage memory for SOAP headers.

Table 9–1: ABL to manage object memory for SOAP headers

<table>
<thead>
<tr>
<th>ABL element</th>
<th>Applied to object</th>
<th>Deletes the ABL object</th>
<th>Deletes all underlying XML DOM objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting the DeleteOnDone OUTPUT parameter of the request header handler to TRUE (see the &quot;Defining header handlers&quot; section on page 9–8)</td>
<td>SOAP header object</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DELETE OBJECT statement</td>
<td>SOAP header object</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>X-document object</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SOAP header-entryref object</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>X-noderef object</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DELETE-HEADER-ENTRY() method</td>
<td>SOAP header-entryref object</td>
<td>No</td>
<td>Yes¹</td>
</tr>
<tr>
<td>DELETE-NODE( ) method</td>
<td>X-noderef object</td>
<td>No</td>
<td>Yes¹</td>
</tr>
</tbody>
</table>

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

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 the “SOAP header object model” section on page 9–4. 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**

Table 9–2 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.

**Table 9–2: SOAP header object attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CHARACTER</td>
<td>Returns the qualified name of the SOAP header (“namespacePrefix:HEADER”)</td>
</tr>
<tr>
<td>NUM-HEADER-ENTRIES</td>
<td>INTEGER</td>
<td>Returns the number of entries attached to the SOAP header</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHARACTER</td>
<td>Returns the handle type, ”SOAP-HEADER”</td>
</tr>
</tbody>
</table>
Table 9–3 briefly describes all of the methods on the SOAP header object.

### Table 9–3: SOAP header object methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD-HEADER-ENTRY (h\text{HeaderEntryRef})</td>
<td>LOGICAL</td>
<td>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 (h\text{HeaderEntryRef}) references the new header entry.</td>
</tr>
<tr>
<td>GET-HEADER-ENTRY (h\text{HeaderEntryRef, index})</td>
<td>LOGICAL</td>
<td>Associates a specified SOAP header-entryref object handle (h\text{HeaderEntryRef}) with an index-specified entry (\text{index}) in the associated SOAP header.</td>
</tr>
</tbody>
</table>

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

### SOAP header-entryref object attributes and methods

Table 9–4 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.

### Table 9–4: SOAP header-entryref object attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTOR</td>
<td>CHARACTER</td>
<td>Returns the value of the actor attribute specified in the associated SOAP header entry</td>
</tr>
<tr>
<td>LOCAL-NAME</td>
<td>CHARACTER</td>
<td>Returns the unqualified part of name specified for the associated SOAP header entry element</td>
</tr>
<tr>
<td>MUST-UNDERSTAND</td>
<td>LOGICAL</td>
<td>Returns the value of the mustUnderstand attribute specified in the associated SOAP header entry</td>
</tr>
<tr>
<td>NAME</td>
<td>CHARACTER</td>
<td>Returns the qualified name of the SOAP header entry (&quot;namespacePrefix:localName&quot;)</td>
</tr>
<tr>
<td>NAMESPACE-URI</td>
<td>CHARACTER</td>
<td>Returns the namespace URI prefixed to the associated SOAP header entry element’s name</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHARACTER</td>
<td>Returns the handle type, &quot;SOAP-HEADER-ENTRYREF&quot;</td>
</tr>
</tbody>
</table>
Table 9–5 briefly describes all of the methods on the SOAP header-entryref object.

### Table 9–5: SOAP header-entryref object methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE-HEADER-ENTRY ()</td>
<td>LOGICAL</td>
<td>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.</td>
</tr>
<tr>
<td>GET-NODE (handle)</td>
<td>LOGICAL</td>
<td>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.</td>
</tr>
<tr>
<td>GET-SERIALIZED ()</td>
<td>LONGCHAR</td>
<td>Returns the XML for the underlying SOAP header entry in serialized form.</td>
</tr>
<tr>
<td>SET-ACTOR (string)</td>
<td>LOGICAL</td>
<td>Sets the value (string) of the actor attribute in the underlying SOAP header entry. This method returns TRUE only if there is an underlying SOAP header entry (XML sub-tree) associated with the object handle.</td>
</tr>
<tr>
<td>SET-MUST-UNDERSTAND (logical)</td>
<td>LOGICAL</td>
<td>Sets the value (logical) of the mustUnderstand attribute in the underlying SOAP header entry. This method returns TRUE only if there is an underlying SOAP header entry (XML sub-tree) associated with the object handle.</td>
</tr>
<tr>
<td>SET-NODE (hXnoderef)</td>
<td>LOGICAL</td>
<td>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).</td>
</tr>
<tr>
<td>SET-SERIALIZED (longchar)</td>
<td>LOGICAL</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Note:** All of the methods and functions listed in Table 9–5 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*. 

---

9–30
Handling Errors in ABL Requests to 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:

- 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 Identifier (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. Table 10–1 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 10–1: ERROR–STATUS handle attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>LOGICAL</td>
<td>Indicates that an ERROR condition was returned as a result of processing a Web service request</td>
</tr>
<tr>
<td>ERROR–OBJECT–DETAIL</td>
<td>HANDLE</td>
<td>References the SOAP fault object for any Web service request that returns a SOAP fault trapped using ABL NO–ERROR option</td>
</tr>
</tbody>
</table>

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 */
```

The “Examples of ABL accessing a SOAP fault” section on page 10–8 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. Table 10–2 summarizes the class representing all system errors and its subclass that represents SOAP errors.

Table 10–2: System error classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Progress.Lang.SysError</td>
<td>Inherits the Progress.Lang.ProError members:</td>
<td>A subclass of Progress.Lang.ProError that represents (with its subclasses) all ABL system errors</td>
</tr>
<tr>
<td></td>
<td>• CallStack property</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• NumMessages property</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Severity property</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• GetMessage( ) method</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• GetMessageNum( ) method</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SoapFault</td>
<td></td>
</tr>
</tbody>
</table>
Table 10–3 describes the properties and methods of the Progress.Lang.SoapFaultError class.

### Table 10–3: SoapFaultError class members

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallStack property</td>
<td>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.</td>
</tr>
<tr>
<td>NumMessages property</td>
<td>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.</td>
</tr>
<tr>
<td>Severity property</td>
<td>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).</td>
</tr>
<tr>
<td>GetMessage( MessageIndex ) method</td>
<td>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.</td>
</tr>
<tr>
<td>GetMessageNum( MessageIndex ) method</td>
<td>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.</td>
</tr>
<tr>
<td>SoapFault</td>
<td>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 the “Managing a SOAP fault” section on page 10–6 for more information about the SOAP-FAULT object.</td>
</tr>
</tbody>
</table>
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.

Table 10–4 lists the attributes on the SOAP fault object, which contains no methods.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAP-FAULT-CODE</td>
<td>CHARACTER</td>
<td>Returns the value of the &lt;faultcode&gt; element of the SOAP fault message, which identifies the fault</td>
</tr>
<tr>
<td>SOAP-FAULT-STRING</td>
<td>CHARACTER</td>
<td>Returns the value of the &lt;faultstring&gt; element of the SOAP fault message, which provides a human-readable description of the fault</td>
</tr>
<tr>
<td>SOAP-FAULT-ACTOR</td>
<td>CHARACTER</td>
<td>Returns the value of the &lt;faultactor&gt; element of the SOAP fault message, which is a URI that identifies the Web service returning the fault</td>
</tr>
<tr>
<td>SOAP-FAULT-DETAIL</td>
<td>HANDLE</td>
<td>References the SOAP fault-detail object, which contains more application-specific error information</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHARACTER</td>
<td>Returns the handle type, &quot;SOAP-FAULT&quot;</td>
</tr>
</tbody>
</table>

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 (?). Like 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.

Table 10–5 lists the single attribute of the SOAP fault-detail object.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>CHARACTER</td>
<td>Returns the handle type, &quot;SOAP-FAULT-DETAIL&quot;</td>
</tr>
</tbody>
</table>
Table 10–6 lists the methods of the SOAP fault-detail object.

**Table 10–6: SOAP fault-detail object methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET-NODE( )</td>
<td>LOGICAL</td>
<td>Returns a handle to an X-noderef object that references the root node (SOAP <code>&lt;detail&gt;</code> element) of a DOM tree containing the parsed XML for the underlying SOAP detail information</td>
</tr>
<tr>
<td>GET-SERIALIZED( )</td>
<td>LONGCHAR</td>
<td>Returns the XML for the underlying SOAP fault detail information in serialized form</td>
</tr>
</tbody>
</table>

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 Chapter 9, “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 the “Analyzing complex data” section on page 6–14. For more information on the ABL DOM, SAX, and temp-table/ProDataSet XML features, see *OpenEdge Development: Working with XML*. 
Examples of ABL accessing a SOAP fault

The following procedure fragment runs 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” section on page 10–2). It accesses and begins to examine the SOAP fault message as described in the following steps. There are two versions of the fragment: one uses traditional error handling and the other uses structured error handling.

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

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

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

Sample SOAP fault procedure

(1 of 2)

```
DEFINE VARIABLE hWS AS HANDLE.
DEFINE VARIABLE hStockPortType AS HANDLE.
DEFINE VARIABLE price AS DECIMAL.
CREATE SERVER hWS.
```
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 or not the second CATCH block succeeded or failed. This makes the FINALLY block a good place to put clean up code.

Sample SOAP fault procedure

```ABL
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/wsd1/stock.wsd1
    -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 ERROR-OBJECT-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 */
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 the “Using the WSDL Analyzer” section on page 6–3.

   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

Table 10–7 lists some tools that you can use to view Web service messages as they are exchanged between the client and the Web service.

<table>
<thead>
<tr>
<th>This SOAP message viewer . . .</th>
<th>Is . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSAViewer</td>
<td>Provided with the OpenEdge installation</td>
</tr>
<tr>
<td>ProSOAPView</td>
<td>Provided with the OpenEdge installation</td>
</tr>
<tr>
<td>Microsoft SOAP Toolkit</td>
<td>Available for free download from Microsoft’s Web site</td>
</tr>
</tbody>
</table>

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 Chapter 5, “Testing and Debugging OpenEdge 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-port webservice-host webservice-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 Figure 10–1.

Figure 10–1: 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 Spy → StartSpying 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 Figure 10–1).

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.
Appendices

Appendix A, Developing a .NET Client to Consume OpenEdge Web Services
Appendix B, Developing a Java Client to Consume OpenEdge Web Services
Appendix C, ABL Elements for Consuming Web Services
Appendix D, Data Type Conversion Rules for ABL Calls to Web Services
Appendix E, Understanding WSDL Details
Appendix F, Commands and Utilities
This appendix describes the basic tasks to develop a .NET client application to consume an OpenEdge Web service, as detailed in the following sections:

- What is Microsoft .NET?
- Using .NET as your Web services client
- VB.NET sample Web service specifications
- Creating the VB.NET client interface
- Developing the VB.NET client application
- Creating .NET DataSets from ProDataSet parameters
- Learning more about writing .NET clients
What is Microsoft .NET?

Briefly, Microsoft .NET supports the following features:

- As described by Microsoft, .NET is a robust platform for developing the next generation of applications.
- It is built using industry standards, such as XML and SOAP.
- The .NET platform supports many languages, including:
  - Visual Basic .NET
  - Visual C# .NET
- The .NET API, including data type support, is the same for all languages.
- .NET provides a single development environment, Visual Studio® .NET, to work with all its supported languages.
- .NET supports all of the SOAP formats that are supported by OpenEdge Web services.
Using .NET as your Web services client

To create a .NET Web service client application, you perform a standard set-up procedure at the start of application development.

To create your Web service client application using Visual Studio.NET:

1. Choose a .NET language and create a new project.

2. Add a Web reference for the Web service by specifying the WSDL file location. This automatically creates a .NET interface.

3. Add logic to the client to create interface objects and call interface methods.

Most sections of this chapter use Visual Basic (VB.NET) to demonstrate client development with .NET. The “Creating .NET DataSets from ProDataSet parameters” section on page A–22 uses C# instead. Compiling and running .NET applications with Visual Studio is straightforward. .NET compiles the application automatically when you run it from Visual Studio .NET.
## VB.NET sample Web service specifications

Table A–1 describes the specifications for a sample Web service and interface used to illustrate client development with VB.NET.

<table>
<thead>
<tr>
<th>Property or component</th>
<th>Value or name</th>
<th>Object type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web reference</td>
<td>OrderService</td>
<td>–</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://servicehost:80/wsa/wsa1">http://servicehost:80/wsa/wsa1</a></td>
<td>–</td>
</tr>
<tr>
<td>Session model</td>
<td>Managed</td>
<td>–</td>
</tr>
<tr>
<td>TargetNamespace</td>
<td>urn:OrderSvc:OrderInfo</td>
<td>–</td>
</tr>
<tr>
<td>WSDL objects</td>
<td>OrderInfoObj</td>
<td>AppObject</td>
</tr>
<tr>
<td></td>
<td>CustomerOrderObj</td>
<td>ProcObject</td>
</tr>
<tr>
<td></td>
<td>OrderInfoID</td>
<td>AppObject ID</td>
</tr>
<tr>
<td></td>
<td>CustomerOrderID</td>
<td>ProcObject ID</td>
</tr>
<tr>
<td></td>
<td>OrderDetailsRow</td>
<td>Temp-table OrderDetails</td>
</tr>
</tbody>
</table>

**Note:** This sample is available on the Documentation and Samples (doc_samples) directory of the OpenEdge product DVD or Progress Documentation Web site.
Creating the VB.NET client interface

The following sections describe tasks and features for creating the VB.NET client interface:

- Adding a Web reference
- Reference.vb—Web service client interface
- Reference.vb—Open Client objects
- .NET data type mapping
- Accessing parameter data
- TABLE (static temp-table) parameters
- TABLE-HANDLE (dynamic temp-table) parameters
- Sample interface object prototypes in Reference.vb
- Sample common interface method prototypes in Reference.vb
- Sample ABL and VB.NET interface method prototypes

These sections review information presented in previous sections of the manual, to provide context for understanding the VB.NET environment.

Adding a Web reference

Adding a Web reference essentially generates the client interface.

To generate a client interface:

1. Enter the URL for the WSDL file. The TargetURI can be set to either the Web service friendly name (AppObject name, by default) or the TargetNamespace for the Web service, as in the following examples:


2. Rename the Web reference you added to the project appropriately (for example, from servicehost to OrderService). Select the Show All Files toolbar button (or choose Project→Show All Files) to locate the Web reference file, Reference.vb under the renamed Web reference.
Reference.vb—Web service client interface

The client interface generated from the WSDL has objects for:

- Every Open Client object, for example, OrderInfo and CustomerOrder
- The object ID for each Open Client object (except a session-free AppObject); for example, OrderInfoID and CustomerOrderID
- Every temp-table, for example, OrderDetailsRow

Reference.vb—Open Client objects

Each Open Client object class usually contains:

- The URL of the WSA
- Methods defined in the WSDL for the object
- A member variable for its own object ID, such as OrderInfoID in the sample
- Member variables for the object IDs of any objects created using CreateXX_* methods (SubAppObjects and ProcObjects), such as CustomerOrderID in the sample

Each object is named as follows:

- It takes the WSDL service name if the Web service has only one object (AppObject), such as OrderInfoService.
- It takes the WSDL <binding> element names, such as OrderInfoObj and CustomerOrderObj, if the Web service has more than one object.

.NET data type mapping

Table A–2 lists the data-type mappings for parameters between ABL and .NET.

Table A–2: Supported .NET data types

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>.NET data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>String</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>Long</td>
</tr>
<tr>
<td>DATASET</td>
<td>Object</td>
</tr>
<tr>
<td>DATASET-HANDLE</td>
<td>Object</td>
</tr>
<tr>
<td>DATE</td>
<td>Date</td>
</tr>
<tr>
<td>DATETIME</td>
<td>DateTime</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>DateTime</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Decimal</td>
</tr>
<tr>
<td>INT64</td>
<td>Int64</td>
</tr>
</tbody>
</table>
Creating the VB.NET client interface

Accessing parameter data

The interface automatically generates parameter data for VB.NET client variables:

- For scalar parameters, the client does not need to do any special additional processing to use them.
- **TABLE, TABLE-HANDLE, DATASET and DATASET-HANDLE** parameters require extra processing for the client to use them.

**TABLE (static temp-table) parameters**

For **TABLE** parameters:

- Client interfaces contain an object for each WSDL `<complexType>` definition of a **TABLE** row. For example, the `OrderDetailsRow` class is declared as follows:

  ```
  Public Class OrderDetailsRow...
  ```

- The client interface represents the **TABLE** parameters as arrays of these row objects.

**Table A–3** lists the data type mappings for **TABLE** columns between ABL and .NET.

### Table A–2: Supported .NET data types

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>.NET data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER (32 bit)</td>
<td>Integer</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>Boolean</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>String</td>
</tr>
<tr>
<td>MEMPTR</td>
<td>Byte( )</td>
</tr>
<tr>
<td>RAW</td>
<td>Byte( )</td>
</tr>
<tr>
<td>RECID (32 or 64 bit)</td>
<td>Long</td>
</tr>
<tr>
<td>ROWID</td>
<td>Byte( )</td>
</tr>
<tr>
<td>TABLE</td>
<td>Object</td>
</tr>
<tr>
<td>TABLE-HANDLE</td>
<td>Object</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>Long</td>
</tr>
</tbody>
</table>

**Table A–3: .NET data types for TABLE parameter columns**

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>.NET data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>Byte( )</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>String</td>
</tr>
</tbody>
</table>
TABLE-HANDLE (dynamic temp-table) parameters

For TABLE-HANDLE parameters, the client must create (for input) and parse (for output) the XML Schema and data for the TABLE-HANDLE.

Table A–4 lists the typical data type mappings for TABLE-HANDLE columns between ABL and .NET.

Table A–4: .NET data types for TABLE-HANDLE parameter columns

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>.NET data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>String</td>
</tr>
<tr>
<td>DATE</td>
<td>Date</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Decimal</td>
</tr>
<tr>
<td>INT64</td>
<td>Int64</td>
</tr>
<tr>
<td>INTEGER (32 bit)</td>
<td>Integer</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>Boolean</td>
</tr>
<tr>
<td>RAW</td>
<td>Byte( )</td>
</tr>
</tbody>
</table>

1. This information also applies to the constituent temp-tables in a dynamic ProDataSet parameter.
Creating the VB.NET client interface

When passing a TABLE-HANDLE parameter, the interface or the client views it with different object types, depending on the Web service SOAP format:

- For RPC/Encoded:
  - The interface represents the parameter as an Object type
  - The client represents the parameter as a System.Array type

- For Doc/Lit, both the interface and the client represent the parameter as a System.Xml.XmlElement type.

Sample interface object prototypes in Reference.vb

The Web reference file (Reference.vb) defines the sample interface objects using the following Class declarations. Note the member variables:

- **Object ID classes**
  - **OrderInfoID**:
    ```vbes
    Public Class OrderInfoID Inherits SoapHeader
    Public UUID As String
    End Class
    ```

  - **CustomerOrderID**:
    ```vbes
    Public Class CustomerOrderID Inherits SoapHeader
    Public UUID As String
    End Class
    ```

- **AppObject class**
  - **OrderInfoObj**:
    ```vbes
    Public Class OrderInfoObj
    Public OrderInfoIDValue As OrderInfoID
    Public CustomerOrderIDValue As CustomerOrderID
    ...
    End Class
    ```

*Note:* The methods in this class declaration are not shown.
• ProcObject class

CustomerOrderObj:

```vbnet
Public Class CustomerOrderObj
    Public CustomerOrderIDValue As CustomerOrderID
    ...
End Class
```

**Note:** The methods in this class declaration are not shown.

**Sample common interface method prototypes in Reference.vb**

The Web reference file (Reference.vb) defines the sample common interface methods for managing Web service objects using the following Sub declarations:

• Common OrderInfoObj AppObject methods

  **Connect_OrderInfo( ):**

```vbnet
Public Sub Connect_OrderInfo(
    ByVal userId As String,
    ByVal password As String,
    ByVal appServerInfo As String)
```

  **Release_OrderInfo( ):**

```vbnet
Public Sub Release_OrderInfo( )
```

• Common CustomerOrderObj ProcObject methods

  **Release_CustomerOrder( ):**

```vbnet
Public Sub Release_CustomerOrder( )
```
Sample ABL and VB.NET interface method prototypes

The following examples show ABL prototypes and how the sample Reference.vb file maps them to the equivalent VB.NET method prototypes.

Note: The object ID in the SOAP headers are not shown in the VB.NET method prototypes.

Non-persistent procedure

- ABL:

```abl
/* FindCustomerByNum.p */
DEFINE INPUT PARAMETER CustomerNumber AS INTEGER.
DEFINE OUTPUT PARAMETER CustomerName AS CHARACTER.
```

- VB.NET OrderInfoObj(AppObject) method:

```vbnet
Public Sub FindCustomerByNum(
    ByVal CustomerNumber As Integer,
    ByRef CustomerName As String)
```

Persistent procedure

- ABL:

```abl
/* CustomerOrder.p */
DEFINE INPUT PARAMETER custNum AS INTEGER.
```

Note: This procedure returns a value using the ABL RETURN statement.

- VB.NET OrderInfoObj(AppObject) method:

```vbnet
Public Function CreatePO_CustomerOrder
    (ByVal custNum As Integer) As String
```

User-defined function

- ABL:

```abl
/* CustomerOrder.p */

FUNCTION GetTotalOrdersByNumber RETURNS INTEGER
    (Threshold AS DECIMAL):
• VB.NET CustomerOrderObj (ProcObject) method:

```vbnet
Public Function GetTotalOrdersByNumber
(ByVal Threshold As Decimal) As Integer
```

**Internal procedure passing a TABLE parameter**

• ABL:

```ABL
/* CustomerOrder.p */
PROCEDURE GetOrderDetails:
  DEFINE OUTPUT PARAMETER TABLE FOR OrderDetails.
```

• VB.NET CustomerOrderObj (ProcObject) method:

```vbnet
Public Sub GetOrderDetails(
  ByRef OrderDetails( ) As OrderDetailsRow)
```

**TABLE definition**

• ABL:

```ABL
/* CustomerTemp.p */
DEFINE TEMP-TABLE OrderDetails
  FIELD OrderNum LIKE Order.OrderNum
  FIELD SalesRep LIKE Order.SalesRep
  FIELD OrderDate LIKE Order.OrderDate
  FIELD ShipDate LIKE Order.ShipDate
  FIELD TotalDollars AS DECIMAL
  FIELD OrderStatus LIKE Order.OrderStatus.
```

• VB.NET OrderDetailsRow object:

```vbnet
Public Class OrderDetailsRow
  Public OrderNum As Integer
  Public SalesRep As String
  <...> Public OrderDate As Date
  <...> Public ShipDate As Date
  Public TotalDollars As Decimal
  Public OrderStatus As String
```
External procedure passing a TABLE-HANDLE parameter

- **ABL:**

```plaintext
/* DynTT.p */
DEFINE INPUT-OUTPUT PARAMETER TABLE-HANDLE ttHandle.
```

- **VB.NET RPC/Encoded method:**

```vbnet
Public Sub DynTT(ByRef ttHandle As Object)
```

- **VB.NET Doc/Lit and RPC/Literal method:**

```vbnet
Public Function DynTT(ByRef ttHandle As System.Xml.XmlElement) As String
```
Developing the VB.NET client application

Once you have added the Web reference and generated the client interface for the Web service, you can develop the client application. The typical tasks for developing a VB.NET client include:

- Creating the Web service
- Running a non-persistent (external) procedure
- Creating server-side context for a ProcObject (running a persistent procedure)
- Running an internal procedure or user-defined function
- Creating a SubAppObject
- Releasing an object
- Running a procedure with a TABLE parameter
- Processing the data from a TABLE parameter
- Running a procedure with a TABLE-HANDLE parameter
- Preparing schema and data for input TABLE-HANDLE parameter
- Processing schema and data for output TABLE-HANDLE parameter
- Extra processing for RPC/Encoded TABLE-HANDLE parameters
- Handling errors on the client

Notes: For more information on the concepts and procedures that underlie these tasks, see Part II, “Creating OpenEdge Web Services.”

OpenEdge comes installed with complete sample client applications that access OpenEdge Web services. For more information, see the “Sample Web service applications” section on page 1–17.
Creating the Web service

To create the Web service:

1. Create the AppObject:

```
' There is no communication with the WSA at this time
Dim WS_OrderService As OrderService.OrderInfoObj
WS_OrderService = New OrderService.OrderInfoObj()
```

2. Call the connect method on the AppObject (session managed only):

```
' Connect to the Progress Session Managed Web Service
WS_OrderService.Connect_OrderInfo("", "", "")
```

Note: Note that the value of the OrderInfoIDValue is extracted from the SOAP response header automatically by the interface. This value is then assigned to the OrderInfoIDValue member variable of the AppObject (WS_OrderService).

Running a non-persistent (external) procedure

The following example shows how you might invoke the sample FindCustomerByNum() method:

```
' Send a request to the AppServer to get the Customer Name
' and display in the UI
WS_OrderService.FindCustomerByNum(custNum, custName)
If custName <> Nothing Then
    TblCustName.Text = custName
```

Note: The OrderInfoIDValue is automatically inserted in the SOAP request header by the interface.
Creating server-side context for a ProcObject (running a persistent procedure)

To create the context for a ProcObject on the AppServer and run the persistent procedure:

1. Create the ProcObject by invoking the CreatePO_CustomerOrder() method on the AppObject:

   ```vba
   ' There is no communication with the WSA at this time
   Dim WS_custOrder As OrderService.CustomerOrderObj
   WS_custOrder = New OrderService.CustomerOrderObj()
   ' Run the persistent procedure CustomerOrder.p on the AppServer
   WS_OrderService.CreatePO_CustomerOrder(custNum)
   
   Note: OrderInfoIDValue is automatically inserted in the SOAP request header by the interface, and the AppObject CustomerOrderIDValue member variable is also filled in automatically by the interface.
   
2. Copy the ProcObject ID from the AppObject to the ProcObject:

   ```vba
   ' Save the ProcObjectID in the ProcObject - copy from AppObject
   WS_custOrder.CustomerOrderIDValue =
   WS_OrderService.CustomerOrderIDValue
   ```

Running an internal procedure or user-defined function

The following example shows how you might invoke the GetTotalOrdersByNumber() method on the ProcObject to run a user-defined function:

```vba
' Run the user-defined function GetTotalOrdersByNumber in
' CustomerOrder.p on the AppServer
intResult = WS_custOrder.GetTotalOrdersByNumber(1000000.0)
```

Note: The CustomerOrderIDValue is automatically inserted in the SOAP request header by the interface.

Methods for running an internal procedure are essentially the same as those for running a user-defined function.
Creating a SubAppObject

To create a SubAppObject:

1. Create the SubAppObject and evoke the CreateAO_Payroll() method:

   ```vbscript
   ' There is no communication with the WSA at this time
   Dim WS_Payroll As OrderService.PayrollObj
   WS_Payroll = New OrderService.PayrollObj()

   ' Create the SubAppObject
   WS_OrderService.CreateAO_Payroll()
   ```

   **Note:** The OrderInfoIDValue will automatically be put into the SOAP request header by the interface, and the PayrollIDValue AppObject member variable is filled in automatically by the interface.

2. Copy the SubAppObject ID from the AppObject to the SubAppObject:

   ```vbscript
   ' Save the SubAppObjectID in the SubAppObject copy from AppObject
   WS_Payroll.PayrollIDValue = WS_OrderService.PayrollIDValue
   ```

Releasing an object

For releasing an object:

- Every Open Client object except for a session-free AppObject has a release method.
- The syntax is the same for all objects, as in the following example that releases the CustomerOrder object, which happens to be a ProcObject:

   ```vbscript
   ' Tell the AppServer to release the Persistent Proc
   If Not (WS_custOrder.CustomerOrderIDValue Is Nothing) Then
     WS_custOrder.Release_CustomerOrder()
     WS_custOrder.CustomerOrderIDValue = Nothing
   End If
   ```
Running a procedure with a TABLE parameter

This example shows how you might invoke a method that passes a TABLE parameter, by calling the sample method, GetOrderDetails( ):

```vbnet
' Run the internal procedure GetOrderDetails in CustomerOrder.p. ' OUTPUT parameter is the OrderDetails TEMP-TABLE
Dim WS_OrderDetails(-1) As OrderService.OrderDetailsRow
WS_custOrder.GetOrderDetails(WS_OrderDetails)
```

**Note:** The CustomerOrderIDValue is automatically inserted in the SOAP request header by the interface.

Processing the data from a TABLE parameter

This example shows how you might process data from a TABLE parameter:

```vbnet
' Loop through the rows to obtain some of the column values
Dim i As Integer
Dim OrderNum As Integer
Dim OrderStatus As String
Dim TotalDollars As Decimal
For i = 0 To WS_OrderDetails.Length - 1
    OrderNum = WS_OrderDetails(i).OrderNum
    OrderStatus = WS_OrderDetails(i).OrderStatus
    TotalDollars = WS_OrderDetails(i).TotalDollars
Next i
```

Running a procedure with a TABLE-HANDLE parameter

This example shows how you might invoke a method on a Doc/Lit Web service that takes a TABLE-HANDLE parameter, by calling the sample method, DynTT( ):

```vbnet
Imports System.Xml

Dim dynTTEl as XmlElement
' Code to create XML document representing dynTTEl goes here

' Run the non-persistent procedure DynTT.p.
' INPUT-OUTPUT parameter is a TABLE-HANDLE
wsObj.DynTT(dynTTEl)
' Code to process the output TABLE-HANDLE from DynTT.p goes here
```
Preparing schema and data for input TABLE-HANDLE parameter

This example shows how you might prepare the schema and data for an input TABLE-HANDLE parameter by building up the XML element nodes for the SOAP message from existing XML Schema and XML data files:

```vbscript
Dim schemaDoc As XmlDocument = New XmlDocument()
Dim dataDoc As XmlDocument = New XmlDocument()
Dim dataSetDoc As XmlDocument = New XmlDocument()
Dim root As XmlElement
Dim schemaNode As XmlNode
Dim dataNode As XmlNode

' Load XML Schema(.xsd) and Data(.xml) files into XmlDocuments
schemaDoc.Load("ttEmp.xsd")
dataDoc.Load("empData.xml")

' Load the outer element into the dataSetDoc XmlDocument
dataSetDoc.LoadXml("<DataSet></DataSet>"
root = dataSetDoc.DocumentElement
root.SetAttribute("xmlns", "")

' Insert schema and data nodes as children of the dataSetDoc root node

root.AppendChild(schemaNode)
root.AppendChild(dataNode)

dynTTEl = dataSetDoc.DocumentElement()
```
Processing schema and data for output TABLE-HANDLE parameter

This example shows how you might process the data from an output TABLE-HANDLE parameter by walking the output XML Schema and XML data to create a .NET DataSet and bind the DataSet to a .NET DataGrid:

```vbnet
Dim schemaEl As XmlElement
Dim dataEl As XmlElement
Dim node As XmlNode
Dim type As XmlNodeType
...
' Extract the schema and data elements from the output
schemaEl = Nothing
dataEl = Nothing
For i = 0 To dynTTEl.ChildNodes.Count - 1
    node = dynTTEl.ChildNodes( i )
    type = node.NodeType
    If type = System.Xml.XmlNodeType.Element Then
        If schemaEl Is Nothing Then
            schemaEl = node
        ElseIf dataEl Is Nothing Then
            dataEl = node
        Else
            'Too many elements, something is wrong
        End If
    End If
Next i
' Load schema and data into a System.Data.DataSet, then bind to a
' System.Windows.Forms.DataGrid
mySchemaReader = New System.Xml.XmlNodeReader(schemaEl)
myDataReader = New System.Xml.XmlNodeReader(dataEl)
myDataSet.ReadXmlSchema(mySchemaReader)
myDataSet.ReadXml(myDataReader)
' Load myDataGrid with the output from DynTT
myDataGrid.SetDataBinding(myDataSet, "Item")
```

Extra processing for RPC/Encoded TABLE-HANDLE parameters

For RPC/Encoded Web services, a TABLE-HANDLE parameter is a System.Array containing the <DataSet> System.Xml.XmlElement. Process the array as follows for input and output:

- **Input** — Creates the System.Array with the XmlElement containing schema and data.
- **Output** — Extracts the last element of the System.Array, which is the XmlElement containing schema and data.
The following example shows the outlines of both procedures:

```
' This code goes after the code that creates XML document representing dynTTEl
' and before calling the method taking the TABLE-HANDLE parameter
...  
' Create a System.Array containing dynTTEl
dynTTArray = System.Array.CreateInstance(dynTTEl.GetType, 1)
dynTTArray(0) = dynTTEl

' Run DynTT.p
wsObj.DynTT(dynTTArray)

' Process the output TABLE-HANDLE from DynTT.p
dynTTEl = dynTTArray(dynTTArray.Length - 1)
' Continue with the rest of the code for processing the output
...
```

Handling errors on the client

To handle a SOAP fault in a VB.NET client, catch the SoapException and parse it, as shown:

```
Try
  ' Code to access the Web service
  ...
    Dim detail As String, reqId As String
    detail = parseSoapException(soapEx.Detail, reqId)
    MsgBox(detail, MsgBoxStyle.Critical, soapEx.Message)
End Try
```

**Caution:** In any catch block where you exit the program, you must release all Web service objects you created in the program.

The `parseSoapException( )` method in this example is a client function provided in the .NET samples installed with OpenEdge. It parses the detail error information from the SOAP fault message.

You can also use the Microsoft SOAP Toolkit to assist in debugging SOAP fault messages. For more information, see Chapter 5, “Testing and Debugging OpenEdge Web Services.”
Creating .NET DataSets from ProDataSet parameters

This section presents an example of how to create an .NET DataSet from a ProDataSet parameter. Because .NET has a proprietary method of recognizing and exposing .NET DataSets in WSDL documents, its toolkit cannot translate the WSDL definition of a ProDataSet directly into an .NET DataSet. To work around this limitation, a .NET client can walk the object arrays and populate a .NET DataSet with the data.

Note: This section does not use RPC/Encoded WSDLs and the VB.NET language. Instead, it uses Doc/Lit and C#.NET.

Table A–5 describes the specifications for the sample Web service and interface used in this section.

Table A–5: ProDataSet to .NET sample Web service specifications

<table>
<thead>
<tr>
<th>Property or component</th>
<th>Value or name</th>
<th>Object type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web service</td>
<td>CustOrdersService</td>
<td></td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://servicehost:80/wsa/wsa1">http://servicehost:80/wsa/wsa1</a></td>
<td></td>
</tr>
<tr>
<td>Session model</td>
<td>Session-Free</td>
<td></td>
</tr>
<tr>
<td>TargetNamespace</td>
<td>urn:CustOrders</td>
<td></td>
</tr>
<tr>
<td>WSDL objects</td>
<td>CustOrdersObj</td>
<td>AppObject</td>
</tr>
<tr>
<td></td>
<td>dsCustOrd</td>
<td>ProDataSet dsCustOrd</td>
</tr>
</tbody>
</table>

In general, the other information presented in the previous example for creating a client interface and developing a client application applies here as well. Except where the differences in the client language and session model affect things, you would complete the same tasks to build this sample.
Creating .NET DataSets from ProDataSet parameters

This Web service uses the following code to create and populate a static ProDataSet parameter:

```
/* getCustOrders.p */
DEFINE TEMP-TABLE ttCust NO-UNDO
   FIELD CustNum AS INTEGER
   FIELD Name AS CHARACTER
   FIELD Balance AS DECIMAL
   INDEX CustNumIdx IS UNIQUE PRIMARY CustNum.
DEFINE TEMP-TABLE ttOrder NO-UNDO
   FIELD OrderNum AS INTEGER
   FIELD CustNum AS INTEGER
   FIELD OrderDate AS DATE
   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 EXTENT 2.
DEFINE OUTPUT PARAMETER DATASET FOR dsCustOrd.
DEFINE VARIABLE hq1 AS HANDLE.
DEFINE VARIABLE hq2 AS HANDLE.
DEFINE VARIABLE lret AS LOGICAL.
DEFINE DATA-SOURCE dsCust FOR Customer.
DEFINE DATA-SOURCE dsOrder FOR Order.
/* fill dataset and return to caller */
CREATE QUERY hq1.
hq1:SET-BUFFERS(BUFFER Customer:HANDLE).
lret = hq1:QUERY-PREPARE("for each Customer where CustNum >= " + STRING(iCustNum[1]) + " AND CustNum <= " + STRING(iCustNum[2]))).
DATA-SOURCE dsCust:QUERY = hq1.
/* attach the data-sources to the dataset buffers */
BUFFER ttCust:HANDLE:ATTACH-DATA-SOURCE(DATA-SOURCE dsCust:HANDLE,?,?,?).
BUFFER ttOrder:HANDLE:ATTACH-DATA-SOURCE(DATA-SOURCE dsOrder:HANDLE,?,?,?).
MESSAGE "FILL() " DATASET dsCustOrd:FILL().
```

The Web service accepts an input array to specify a range of customer numbers. The service then outputs the data in the dsCustOrd ProDataSet.

When you add a Web Reference to the CustOrdersService Web service in Microsoft Visual Studio, the proxies in the Reference.cs file include this method to invoke the getCustOrders operation:

```
public string
getCustOrders([System.Xml.Serialization.XmlElementAttribute("iCustNum", IsNullable=true)] System.Nullable<int>[]> iCustNum,
out dsCustOrd dsCustOrd) {
    object[] results = this.Invoke("getCustOrders", new object[] {
        iCustNum});
    dsCustOrd = ((dsCustOrd)(results[1]));
    return ((string)(results[0]));
}
```
The Reference.cs file also includes the following partial classes to describe the ProDataSet and its constituent temp-tables:

```csharp
public partial class dsCustOrd {
    private dsCustOrdTtCust[] ttCustField;
    private dsCustOrdTtOrder[] ttOrderField;
    ...
}

public partial class dsCustOrdTtCust {
    private System.Nullable<int> custNumField;
    private string nameField;
    private System.Nullable<decimal> balanceField;
    ...
}

public partial class dsCustOrdTtOrder {
    private System.Nullable<int> orderNumField;
    private System.Nullable<int> custNumField;
    private System.Nullable<System.DateTime> orderDateField;
    ...
}
```

To access the Web service, you might build an interface like the following one:

The combo-boxes enable you to set the range of customer numbers for the input parameter. When you click the button, the client calls the Web service, retrieves the data for the specified range, and then displays it in the grid.
As shown in the following code, the button’s event logic invokes the Web service passing in the array for the CustNum range and retrieves the requested data:

```csharp
private void button2_Click(object sender, EventArgs e)
{
    string result = null;
    int cnt;
    int?[] CustNumRange = new int?[2];

    CustOrders.dsCustOrd dsCustOrd;
    CustOrders.CustOrdersService mySvc =
        new CustOrders.CustOrdersService();

    try
    {
        CustNumRange[0] = CustNumLow;
        CustNumRange[1] = CustNumHigh;

        if (CustNumHigh < CustNumLow)
            CustNumRange[1] = CustNumRange[0];

        result = mySvc.getCustOrders(CustNumRange, out dsCustOrd);
    }
    catch
    {
        // Error handling...
    }

    // Create a .Net Dataset based on this data
    DataSet x = new DataSet("dsCustOrd");
    DataTable t1 = x.Tables.Add("ttCust");
    DataTable t2 = x.Tables.Add("ttOrder");

    t1.Columns.Add("CustNum", typeof(int));
    t1.Columns.Add("Name", typeof(string));
    t1.Columns.Add("Balance", typeof(System.Decimal));

    t1.Columns[0].Unique = true;

    t2.Columns.Add("OrderNum", typeof(int));
    t2.Columns.Add("CustNum", typeof(int));
    t2.Columns.Add("OrderDate", typeof(DateTime));

    t2.Columns[0].Unique = true;

    DataColumn pCol = t1.Columns[0];
    DataColumn cCol = t2.Columns[1];

    x.Relations.Add(pCol, cCol);
}
```

Since you know the schema of the incoming DATASET parameter, the event logic can create a matching .NET Dataset to accept the incoming data, as follows:

```csharp
// Create a .Net Dataset based on this data
DataSet x = new DataSet("dsCustOrd");
DataTable t1 = x.Tables.Add("ttCust");
DataTable t2 = x.Tables.Add("ttOrder");

t1.Columns.Add("CustNum", typeof(int));
t1.Columns.Add("Name", typeof(string));
t1.Columns.Add("Balance", typeof(System.Decimal));

t1.Columns[0].Unique = true;

t2.Columns.Add("OrderNum", typeof(int));
t2.Columns.Add("CustNum", typeof(int));
t2.Columns.Add("OrderDate", typeof(DateTime));

t2.Columns[0].Unique = true;

DataColumn pCol = t1.Columns[0];
DataColumn cCol = t2.Columns[1];

x.Relations.Add(pCol, cCol);
```
Finally, the event logic fills the DataSet with the data from the incoming DATASET parameter and binds the DataSet to a grid, as follows:

```csharp
//Populate the dataset with data from the SOAP message
GetCustOrders.dsCustOrdTtCust[] ttCust = dsCustOrd.ttCust;
for (cntr = 0; cntr < ttCust.Length; ++cntr)
{
    Object[] ooCust = new Object[3];
    ooCust[0] = ttCust[cntr].CustNum;
    ooCust[1] = ttCust[cntr].Name;
    ooCust[2] = ttCust[cntr].Balance;
    t1.Rows.Add(ooCust);
}

GetCustOrders.dsCustOrdTtOrder[] ttOrder = dsCustOrd.ttOrder;
for (cntr = 0; cntr < ttOrder.Length; ++cntr)
{
    Object[] ooOrd = new Object[3];
    ooOrd[0] = ttOrder[cntr].OrderNum;
    ooOrd[1] = ttOrder[cntr].CustNum;
    ooOrd[2] = ttOrder[cntr].OrderDate;
    t2.Rows.Add(ooOrd);
}

//bind Dataset to Grid
myGrid.SetDataBinding(x, "ttCust");
} catch (Exception ex)
{
    MessageBox.Show("getCustOrders Failed: " + ex.Message);
}
```
Learning more about writing .NET clients

For more information on writing .NET clients, review the .NET samples provided with the OpenEdge, which highlight OpenEdge Web service functionality (see the “Sample Web service applications” section on page 1–17) and see the following sources for:

- General information on Microsoft .NET at the following URL:
  
  http://www.microsoft.com/net/

- Information on Microsoft Visual Studio .NET at the following URL:
  
  http://msdn.microsoft.com/vstudio/

- Information from other .NET developers at CodePlex, Microsoft’s open source project hosting Web site:
  
  http://www.codeplex.com
Developing a Java Client to Consume OpenEdge Web Services

This appendix describes the basic tasks to develop Java client applications that use an OpenEdge Web service, as detailed in the following sections:

- Using Java client toolkits
- Java sample Web service specifications
- Creating Java Axis client interface objects
- Developing the Axis client application
- ProDataSet parameters in Java Web service clients
Using Java client toolkits

Most major Java software vendors are developing Java client toolkits to support Web service client development. In general, these toolkits include features that help automate the generation of client interfaces to Web services and offer other development aids, such as SOAP viewers and other debugging aids. Among the more common Java client toolkits are the following:

- Axis from Apache
- JAX-RPC

The client development samples in the remainder of this section are provided using Axis as the client-side Web services toolkit.

If you are developing a secure Web service that relies on the secure socket layer (SSL) managed by the Web server, OpenEdge provides a utility (procertm) to manage the digital certificates for a Java client. This same utility is also used to manage digital certificates for Java Open Clients. For more information on this utility, see OpenEdge Development: Java Open Clients.
Java sample Web service specifications

Table B–1 describes the specifications for a sample Web service and interface used to illustrate client development with Axis.

Table B–1: Java sample Web service specifications

<table>
<thead>
<tr>
<th>Property or component</th>
<th>Value or name</th>
<th>Object type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web service</td>
<td>OrderService</td>
<td>–</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://servicehost:80/wsa/wsa1">http://servicehost:80/wsa/wsa1</a></td>
<td>–</td>
</tr>
<tr>
<td>Session model</td>
<td>Managed</td>
<td>–</td>
</tr>
<tr>
<td>TargetNamespace</td>
<td>urn:OrderSvc:OrderInfo</td>
<td>–</td>
</tr>
<tr>
<td>WSDL objects</td>
<td>OrderInfoObj</td>
<td>AppObject</td>
</tr>
<tr>
<td></td>
<td>CustomerOrderObj</td>
<td>ProcObject</td>
</tr>
<tr>
<td></td>
<td>OrderInfoID</td>
<td>AppObject ID</td>
</tr>
<tr>
<td></td>
<td>CustomerOrderID</td>
<td>ProcObject ID</td>
</tr>
<tr>
<td></td>
<td>OrderDetailsRow</td>
<td>Temp-table OrderDetails</td>
</tr>
</tbody>
</table>

Note: This sample is available on the Documentation and Samples (doc_samples) directory of the OpenEdge product DVD or Progress Documentation Web site.
Creating Java Axis client interface objects

Axis provides tools that help to automate the creation of the objects for a client interface. The following sections describe these tools and the generated interface objects:

- Generating client interface objects
- Proxy directory structure
- Proxy classes
- Java data type mapping
- Output parameters
- ABL unknown values
- TABLE (static temp-table) parameters
- TABLE-HANDLE (dynamic temp-table) parameters
- Sample Java prototypes for common object methods
- Sample ABL and Java interface method prototypes

To create client interfaces and client applications to access OpenEdge Web services, you must have Java SDK 1.3 or later and the Axis Toolkit. For more information, see the README files in the Java samples described in the “Sample Web service applications” section on page 1–17.

Generating client interface objects

You can generate client interface objects (classes) for the interface using the following command, where wsd1FileRef is a pointer to the WSDL file:

```
java org.apache.axis.wsdl.WSDL2Java wsd1FileRef
```

**Note:** Set the Classpath as specified in the Axis User’s Guide.

The value of wsd1FileRef can be:

- A local reference; for example:

  ```
  C:\OrderInfo.wsdl
  ```

- A URL to the WSDL; for example:

  ```
  http://servicehost:80/wsa/wsa1/wsd1?targetURI=OrderInfo
  ```

Each generated interface object is a class for an Open Client object and/ or some other component of the WSDL that is represented as an object in the Java interface.
Proxy directory structure

The Axis WSDL2Java command creates the client interface objects in Java packages that reflect the TargetNamespace value specified in the WSDL. Thus, for example, suppose the WSDL specifies a TargetNamespace as follows:

```
TargetNamespace="urn:OrderSvc:OrderInfo"
```

The relative directory structure created from this namespace is the following:

```
OrderSvc/OrderInfo
```

Proxy classes

For this sample, the Axis WSDL2Java command provides the classes for the Web service components shown in Table B–2. This represents one class for each specified Web service (WSDL) object or component.

**Note:** Unlike .NET, there are no Java objects created for object IDs.

<table>
<thead>
<tr>
<th>Table B–2: Proxy classes for sample Java client (RPC/Encoded)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Web service components</strong></td>
<td><strong>Class</strong></td>
</tr>
<tr>
<td>Each Open Client object</td>
<td>OrderInfoObjStub</td>
</tr>
<tr>
<td></td>
<td>CustomerOrderObjStub</td>
</tr>
<tr>
<td>One service locator object (Provides the URL to the WSA)</td>
<td>OrderInfoServiceLocator</td>
</tr>
<tr>
<td>Each temp-table input parameter</td>
<td>OrderDetailsRow</td>
</tr>
<tr>
<td>Two holder classes for each output temp-table</td>
<td>ArrayofOrderDetailsRowHolder</td>
</tr>
<tr>
<td></td>
<td>OrderDetailsRowHolder</td>
</tr>
<tr>
<td>SOAP fault detail object</td>
<td>FaultDetail</td>
</tr>
</tbody>
</table>
Java data type mapping

Table B–3 lists the data type mappings for parameters between ABL and Java.

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Java data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>String</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>long</td>
</tr>
<tr>
<td>DATASET (static ProDataSet)</td>
<td>Object (for every DATASET)</td>
</tr>
<tr>
<td>DATASET-HANDLE (dynamic ProDataSet)</td>
<td>DataSetHandleParam Object</td>
</tr>
<tr>
<td>DATE</td>
<td>java.util.Date</td>
</tr>
<tr>
<td>DATETIME</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>INT64</td>
<td>long</td>
</tr>
<tr>
<td>INTEGER (32 bit)</td>
<td>int</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>boolean</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>String</td>
</tr>
<tr>
<td>MEMPTR</td>
<td>byte[ ]</td>
</tr>
<tr>
<td>RAW</td>
<td>byte[ ]</td>
</tr>
<tr>
<td>RECID (32 or 64 bit)</td>
<td>long</td>
</tr>
<tr>
<td>ROWID</td>
<td>byte[ ]</td>
</tr>
<tr>
<td>TABLE (static temp-table)</td>
<td>Object (for every TABLE)</td>
</tr>
<tr>
<td>TABLE-HANDLE (dynamic temp-table)</td>
<td>TableHandleParam Object</td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td>long</td>
</tr>
</tbody>
</table>

Output parameters

The Java language does not provide a mechanism to pass output parameters. To work around this, client interfaces use the Java standard of representing output parameters as holder classes, which provide the reference space to return output parameter values. Axis provides holder classes for all standard Java data types. The Axis interface generator (WSDL2Java) creates holder classes for output TABLE, TABLE-HANDLE, DATASET, and DATASET-HANDLE parameters. For more information on Axis support for holder classes, see the Axis User’s Guide.
ABL unknown values

The Unknown value (?) appears as a null value in Java. For any ABL user-defined function that returns the Unknown value (?) as a primitive Java data type, this raises an exception in the interface method that calls this function. The client must compensate for this by catching the exception, as follows:

```java
catch (NullPointerException con) {
    System.err.println("The total Orders are: 0 (unknown)");
}
```

TABLE (static temp-table) parameters

For TABLE parameters, client interfaces:

- Contain a row object for any input parameter; for example:

  ```java
  public class OrderDetailsRow
  public class ArrayofOrderDetailsRowHolder
  public class OrderDetailsRowHolder
  ```

- Represent an input TABLE parameter as an array of row objects
- Represent output TABLE parameters using an array holder class and a row holder class for the row objects, as in the following RPC/Encoded example:

  ```java
  public class ArrayofOrderDetailsRowHolder
  public class OrderDetailsRowHolder
  ```

Table B–4 lists the data type mappings for TABLE columns between ABL and Java.

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Java data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>byte[]</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>String</td>
</tr>
<tr>
<td>CLOB</td>
<td>String</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>long</td>
</tr>
<tr>
<td>DATE</td>
<td>java.util.Date</td>
</tr>
<tr>
<td>DATETIME</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>INT64</td>
<td>long</td>
</tr>
<tr>
<td>INTEGER (32 bit)</td>
<td>int</td>
</tr>
</tbody>
</table>
TABLE-HANDLE (dynamic temp-table) parameters

For TABLE-HANDLE parameters:

- For every Web service object containing a method that passes a TABLE-HANDLE parameter, the WSDL contains a `<TableHandleParam>` element defined as a `<complexType>` definition. Therefore, a `TableHandleParam` class is created for every client object which contains a method that passes a TABLE-HANDLE parameter. For example:

```java
public class TableHandleParam {
    private org.apache.axis.message.MessageElement [] _any;
    ...}
```

- For input TABLE-HANDLE parameters, the client must create a `TableHandleParam` object, consisting of a `MessageElement` array containing the XML Schema and data for the TABLE-HANDLE. For output TABLE-HANDLE parameters, the client must parse the XML Schema and data in the `MessageElement` array.

Table B–5 lists the typical data type mappings for TABLE-HANDLE columns between ABL and Java.

Table B–5: Data types for TABLE-HANDLE parameter columns

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>Java data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>String</td>
</tr>
<tr>
<td>DATE</td>
<td>java.util.GregorianCalendar</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>INT64</td>
<td>long</td>
</tr>
<tr>
<td>INTEGER (32 bit)</td>
<td>int</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>boolean</td>
</tr>
<tr>
<td>RAW</td>
<td>byte[ ]</td>
</tr>
</tbody>
</table>

1. This information also applies to the constituent temp-tables in a static ProDataSet.
Sample Java prototypes for common object methods

This is an example of an AppObject connect method for the session-managed OrderInfo AppObject as defined for the OrderInfoObjStub class. It makes the logical connection to the AppServer:

```java
public void connect_OrderInfo
    (java.lang.String userID,
     java.lang.String password,
     java.lang.String appServerInfo)
```

This is an example of an AppObject release method for the OrderInfo AppObject as defined for the OrderInfoObjStub class:

```java
public void release_OrderInfo( )
```

**Note:** All interface object release methods have a similar definition.

Sample ABL and Java interface method prototypes

The following examples show ABL prototypes and how they map to equivalent Java interface method prototypes:

**Non-persistent procedure**

- **ABL:**

  ```abl
  /* FindCustomerByNum.p */
  DEFINE INPUT PARAMETER CustomerNumber AS INTEGER.
  DEFINE OUTPUT PARAMETER CustomerName AS CHARACTER.
  ```

- **Java OrderInfoObjStub method:**

  ```java
  public void findCustomerByNum
      (int customerNumber,
       javax.xml.rpc.holder.StringHolder customerName)
  ```

**Note:** The output parameter holder class shown in bold is provided by Axis.
Persistent procedure

- ABL:

```abl
/* CustomerOrder.p */
DEFINE INPUT PARAMETER CustNum AS INTEGER.
```

**Note:** This procedure returns a value using the ABL `RETURN` statement.

- Java `OrderInfoObjStub` method:

```java
public String createPO_CustomerOrder (int custNum)
```

User-defined function

- ABL:

```abl
/* CustomerOrder.p */
FUNCTION GetTotalOrdersByNumber RETURN INTEGER
    (Threshold AS DECIMAL):
```

- Java `CustomerOrderObjStub` method:

```java
public int getTotalOrdersByNumber
    (java.math.BigDecimal threshold)
```

Internal procedure using a temp-table

- ABL:

```abl
/* CustomerOrder.p */
PROCEDURE GetOrderDetails :
    DEFINE OUTPUT PARAMETER TABLE FOR OrderDetails.
```

- Java `CustomerOrderObjStub` method:

```java
public void getOrderDetails
    (OrderSvc.OrderInfo.CustomerOrder.
        holders.ArrayOfOrderDetailsRowHolder orderDetails)
```

**Note:** The output parameter holder class shown in bold is generated by the Axis `WSDL2Java` command.
Temp-table definition

- **ABL:**

```abl
/* CustomerOrder.p */
DEFINE TEMP-TABLE OrderDetails
   FIELD OrderNum LIKE Order.OrderNum
   FIELD SalesRep LIKE Order.SalesRep
   FIELD OrderDate LIKE Order.OrderDate
   FIELD ShipDate LIKE Order.ShipDate
   FIELD TotalDollars AS DECIMAL
   FIELD OrderStatus LIKE Order.OrderStatus.
```

- **Java OrderDetailsRow object:**

```java
public class OrderDetailsRow {
   private java.lang.Integer orderNum;
   private java.lang.String salesRep;
   private java.util.Date orderDate;
   private java.util.Date shipDate;
   private java.math.BigDecimal totalDollars;
   private java.lang.String orderStatus;
   ...
}
```

**External procedure passing a TABLE-HANDLE**

- **ABL:**

```abl
/* DynTT.p */
DEFINE INPUT-OUTPUT PARAMETER TABLE-HANDLE ttHandle.
```

- **Java DynTTObjStub method:**

```java
public void DynTT
    (DynTTSrvc.DynTT.holders.TableHandleParamHolder ttHandle)
```
Developing the Axis client application

Once you have generated the client interface classes for the Web service, some typical tasks for developing a Java Axis client include:

- Setting up the Web service objects
- Using the sample PscObjectIDHandler and HandlerControlBlock classes
- Setting up the PscObjectIDHandler objects
- Connecting to a Web service
- Running a non-persistent (external) procedure
- Creating server-side context for a ProcObject (running a persistent procedure)
- Running an internal procedure or user-defined function
- Running an internal procedure with a TABLE parameter
- Processing the data from a TABLE parameter
- Running a procedure with a TABLE-HANDLE parameter
- Using sample helper classes to manage TABLE-HANDLE parameters
- Preparing schema and data for input TABLE-HANDLE parameters
- Processing schema and data for output TABLE-HANDLE parameters
- Releasing an object
- Handling errors on the client

Notes: For more information on the concepts and procedures underlying these tasks, see Part II, “Creating OpenEdge Web Services.”

OpenEdge comes installed with complete sample client applications that access OpenEdge Web services. For more information, see the “Sample Web service applications” section on page 1–17.

Setting up the Web service objects

To set up the session-managed Web service objects for your client:

1. Create the Web service locator. For the sample, you generate the OrderInfo client's service object that manages the connection with the WSA:

   ```java
   OrderInfoService service = new OrderInfoServiceLocator();
   ```
2. Instantiate the client objects that do the marshalling and unmarshalling of SOAP and HTTP messages for methods on the `OrderInfo` object. For example, you might execute the following method for the sample:

```java
OrderInfoObjStub orderInfo = (OrderInfoObjStub)service.getOrderInfoObj(connectURL);
```

3. Instantiate the client objects that do the marshalling and unmarshalling of SOAP and HTTP messages for methods on the `CustomerOrder` object. For example you might execute the following method for the sample:

```java
CustomerOrderObjStub custOrder = (CustomerOrderObjStub)service.getCustomerOrderObj(connectURL);
```

### Using the sample PscObjectIDHandler and HandlerControlBlock classes

Axis is a JAX-RPC-based Web service toolkit. JAX-RPC-based Web Service toolkits provide the ability to examine, modify, and insert information into SOAP requests using handler objects that conform to a defined interface (`javax.xml.rpc.handler.GenericHandler`). Axis client interface classes do not give programmatic access to SOAP request and response headers; therefore a client application must create and use one of these handlers to access these SOAP message headers.

A client application defines one or more of these handlers and binds them to a `<portType>` (Open Client object) defined in the Web service's WSDL document. Each time a SOAP request or response message is handled by an object, the appropriate handlers bound to the object are called and allowed access to the request or response message. For more information, see the source code modules for the Java samples described in the “Sample Web service applications” section on page 1–17.

OpenEdge provides a sample handler class (`PscObjectIDHandler`) to manage object IDs. This `PscObjectIDHandler` class is one possible implementation of a JAX-RPC handler. It provides two essential pieces of functionality:

1. It is invoked for each incoming SOAP response for the `<portType>` element (Open Client object type) where it is bound. When invoked, it scans the SOAP response header for an object ID that corresponds to an Open Client object name. If it finds one of these Open Client object IDs, it extracts the object ID value and stores it along with the object name. The client application or the handler can then access this stored object ID to insert it into a subsequent SOAP request header. If a handler does not already have an object ID value reserved to insert in a given SOAP request header, it automatically uses the first object ID that it stores.

2. It is invoked for each outgoing SOAP request for the `<portType>` element (Open Client object type) where it is bound. It then inserts a SOAP header containing the object ID for the Open Client object name that corresponds to the `<portType>` element where it is bound. If an object ID value does not exist for the Open Client object name, the handler does not insert a SOAP header into the request.
Along with the PscObjectIDHandler class provided with the OpenEdge Web services samples is a companion class, HandlerControlBlock. This class allows the Web service client application to access object ID values stored in a PscObjectIDHandler object. It also gives the application the ability to control the Object ID value the PscObjectIDHandler inserts into SOAP request headers. For each PscObjectIDHandler object instance, there is a companion HandlerControlBlock created and associated with it.

**Setting up the PscObjectIDHandler objects**

After you have completed the Web service object setup (see the “Setting up the Web service objects” section on page B–12) that creates the OrderInfoObjStub and CustomerOrderObjStub objects, you can create and register the PscObjectIDHandler and HandlerControlBlock objects for both the OrderInfo and CustomerOrder Open Client objects.

The following steps create and register a unique handler for each Open Client object so it can automatically insert its corresponding object ID into outgoing SOAP requests without intervention by the client application.

To create and register a unique handler for each Open Client object:

1. Set the <portType> element name specified in the WSDL for the OrderInfoObjStub so a PscObjectIDHandler can be bound to it. For example:

   ```java
   orderInfo.setPortName("OrderInfoPort");
   ```

   **Note:** Apache Axis does not automatically obtain the <portType> element name from the WSDL document when it generates the client interface classes.

2. Set the <portType> element name specified in the WSDL for the CustomerOrderObjStub so a PscObjectIDHandler can be bound to it. For example:

   ```java
   custOrder.setPortName("CustomerOrderPort");
   ```

   **Note:** Apache Axis does not automatically obtain the <portType> element name from the WSDL document when it generates the client interface classes.

3. Declare the Open Client object names that the PscObjectIDHandler objects will use to scan for object IDs in SOAP response headers returned from the WSA. For example:

   ```java
   String[] orderInfoObjectNames =
   new String[] { "OrderInfo", "CustomerOrder" };
   ```

   **Note:** Both the "OrderInfo" and "CustomerOrder" object names are included because the object IDs for both are returned from factory methods contained in the OrderInfo AppObject.
4. Call the PscObjectIDHandler built-in factory method to create and register a PscObjectIDHandler object for the "OrderInfoPort" <portType> element. The handler is set to scan for and record the object ID values for the OrderInfo and CustomerOrder Open Client objects. This automatically creates and binds a HandlerControlBlock object to the PscObjectIDHandler object, returning a reference for it to the client application, as shown:

```java
HandlerControlBlock orderInfoControlBlock = null;
orderInfoControlBlock =
    PscObjectIDHandler.registerObjectIDHandler(
        service,
        orderInfoObjectNames,
        ",",
        "OrderInfoPort",
        debugHandler);
```

5. Call the PscObjectIDHandler built-in factory method to create and register a PscObjectIDHandler object for the "CustomerOrderPort" <portType> element. The handler is set to scan for and record the object ID values for the OrderInfo and CustomerOrder Open Client objects. This automatically creates and binds a HandlerControlBlock object to the PscObjectIDHandler object, returning a reference for it to the client application, as shown:

```java
HandlerControlBlock custOrderControlBlock = null;
custOrderControlBlock =
    PscObjectIDHandler.registerObjectIDHandler(
        service,
        orderInfoObjectNames,
        ",",
        "CustomerOrderPort",
        debugHandler);
```

**Connecting to a Web service**

For session-managed Web services, make a logical connection to the AppObject and save its object ID for later use. For example:

```java
orderInfo.connect_OrderInfo("", ",", "");
if (null == orderInfoControlBlock.getObjectNameID("OrderInfo"))
{
    throw new Exception("No header returned from a connect operation to OrderInfo.");
}
```

**Note:** The PscObjectIDHandler registered for the OrderInfoPort type object (OrderInfoObjStub) automatically looks for and stores the object ID value returned for the OrderInfoObjStub object if the connect_OrderInfo( ) method is successful. Because this is the first object ID that the handler stores, it is automatically set to be inserted as the object ID value in the SOAP header for each SOAP request sent to the WSA for the OrderInfoObjStub object.
Caution: Now that the Web service is connected, remember to release the AppObject when it is no longer needed.

Running a non-persistent (external) procedure

The following example shows how you might invoke the sample `FindCustomerByNum()` method:

```java
// Send a request to the AppServer to get the Customer Name
StringHolder custNameHolder = new StringHolder();
orderInfo.findCustomerByNum(3, custNameHolder);
System.out.println("Customer #3's name is " + custNameHolder.value);
```

The object ID handler automatically inserts the `orderInfo` object ID into the SOAP request header.

`StringHolder` is one of several holder classes defined by Axis.

Creating server-side context for a ProcObject (running a persistent procedure)

You create the server-side context for the ProcObject as shown below.

To create the context for a ProcObject on the AppServer:

1. Run the persistent procedure, creating the ProcObject:

```java
orderInfo.createPO_CustomerOrder(3);
```

Caution: Now that the ProcObject is created, remember to release the object later when it is no longer needed.

2. Check to see if an object ID was returned:

```java
if (null == orderInfoControlBlock.getObjectNameID("CustomerOrder"))
{
    throw new Exception("No header returned from a create operation for CustomerOrder.");
}
```
3. The CustomerOrderObjStub object is created from a factory method included in the OrderInfoObjStub object. Therefore, the PscObjectIDHandler bound to the OrderInfoObjStub locates and stores the object ID value for the CustomerOrderObjStub object in the OrderInfoObjStub object. Thus the client application must obtain (export) the object ID value for the CustomerOrderObjStub from the OrderInfoObjStub handler and insert (import) it into the CustomerOrderObjStub handler. For example:

```java
custOrderControlBlock.importObjectID("CustomerOrder",
    orderInfoControlBlock.exportObjectID("CustomerOrder"));
```

This enables the CustomerOrderObjStub handler to automatically insert the CustomerOrder object ID into the headers for all SOAP requests made on the CustomerOrderObjStub object.

4. The client application must then remove (release) the object ID value for the CustomerOrderObjStub object from the OrderInfoObjStub handler because it is no longer needed by that handler. For example:

```java
orderInfoControlBlock.releaseObjectNameID("CustomerOrder");
```

**Running an internal procedure or user-defined function**

The following sample shows how you might invoke the GetTotalOrdersByNumber( ) method:

```java
try {
    int orderCount = custOrder.getTotalOrdersByNumber(new BigDecimal(5.0));
    System.out.println("The total Customer Orders are: "+ orderCount);
} catch (NullPointer Exception e) {
    ...
}
```

**Note:** The object ID handler automatically inserts the custOrder object ID into the SOAP request header.
Running an internal procedure with a TABLE parameter

This example shows how you might invoke a method that takes a TABLE parameter, by calling the sample method, `GetOrderDetails( )`:

```java
// Create the holder for the TEMP-TABLE OrderDetails
ArrayOfOrderDetailsRowHolder orderDetailsHolder = new ArrayOfOrderDetailsRowHolder(null);

// Let's now ask the WebService for the order details of the current customer
custOrder.getOrderDetails(orderDetailsHolder);
```

**Note:** The object ID handler automatically inserts the `custOrder` object ID into the SOAP request header.

Processing the data from a TABLE parameter

This example shows how you might process data from a TABLE parameter:

```java
// Loop through the rows and print some of the column values
OrderDetailsRow[] rows = (OrderDetailsRow[]) orderDetailsHolder.value;

// Print out some labels
System.out.println("Customer's Order Details listed below:");
System.out.println("Order# " + "Order Status " + "Total Dollars");

// Now print out all the rows
for (int j = 0; j < rows.length; j++)
{
    OrderDetailsRow thisRow = rows[j];
    System.out.println(thisRow.getOrderNum( ) +
                        thisRow.getOrderStatus( ) +
                        thisRow.getTotalDollars( ));
}
```
Developing the Axis client application

Running a procedure with a TABLE-HANDLE parameter

This example shows how you might invoke a method that takes a TABLE-HANDLE parameter, by calling the sample method, `dynTT()`:

```java
import DynTTSrvC.*;
import DynTTSrvC.DynTT.*;
import DynTTSrvC.DynTT.holders.*;
import org.apache.axis.message.MessageElement;
import org.w3c.dom.Element;
...

// Prepare the Schema and Data for the Input TABLE-HANDLE Parameter
// Create the XML Schema and data for the dynamic TEMP-TABLE.
// Then create the TableHandleParam object, ttHandle
...
// Create the holder for the INPUT-OUTPUT TableHandleParam parameter, ttHandle
TableHandleParamHolder ttHolder = new TableHandleParamHolder();
ttHolder.value = ttHandle;

// Call DynTT.p, passing the Input-Output TABLE-HANDLE
dynTTObj.dynTT(ttHolder);

// Process the schema and data for the Output TABLE-HANDLE
...
```

Using sample helper classes to manage TABLE-HANDLE parameters

OpenEdge Web services sample applications come installed with a set of helper classes that you can use to manipulate TABLE-HANDLE parameters. For example:

- **ColumnMetaData.java** — Represents the schema of a column, as shown:

  ```java
  public class ColumnMetaData {
      protected String name;
      protected int type;
      protected int extent;
      ...
  }
  ``

- **RowSet.java** — Represents the XML Schema and data for a TABLE-HANDLE parameter, as shown:

  ```java
  public class RowSet {
      private Vector m_rows;
      private ColumnMetaData[] m_schema;
      ...
  }
  ``

In the RowSet class, `m_rows` is a Vector of rows, each of which is a Vector of column data. Each column datum is a Java object appropriate for the column data type. The `m_schema` variable is the `ColumnMetaData` array representing the schema for the temp-table.
• **SchemaParser.java** — Parses the `<schema>` element from a SOAP response message and creates a `ColumnMetaData` array for the `RowSet` class, as shown:

```java
public class SchemaParser
{
    protected ColumnMetaData[] m_schema;
    ...
}
```

### Preparing schema and data for input TABLE-HANDLE parameters

This example shows how you might prepare the schema and data for an input TABLE-HANDLE parameter using the helper classes described in the previous section:

```java
// Build up a simple RowSet object representing the input dynamic TEMP-TABLE.
RowSet rsIn = new RowSet();

// Create the MetaData, consisting of String and Integer columns
ColumnMetaData[] schema = new ColumnMetaData[2];
ColumnMetaData firstCol = new ColumnMetaData("Name",
    ColumnMetaData.CHAR_TYPE, 0);
ColumnMetaData secondCol = new ColumnMetaData("Number",
    ColumnMetaData.INT_TYPE, 0);
schema[0] = firstCol;
schema[1] = secondCol;
rsIn.setSchema(schema);

// Create a data row and add to the RowSet
Vector row;
row = new Vector();
row.addElement(new String("Sally Jones"));
row.addElement(new Integer(1));
rsIn.addRow(row);

// Convert the RowSet into a TableHandleParam Object, the input parameter for DynTT.p

// Get the RowSet as a DOM Element. This element has two
// children, a <schema> element and a <Data> element
Element dataSetIn = rsIn.getDataSetAsDom();

// Create a MessageElement containing the DataSet
MessageElement dataSetMsgEl = new MessageElement(dataSetIn);

// Place the DataSet MessageElement into the message element array
// and create the TableHandleParam object
MessageElement msgArrayIn[] = new MessageElement[1];
msgArrayIn[0] = dataSetMsgEl;

// Create the TableHandleParam object, ttHandle, representing the
// dynamic TABLE-HANDLE parameter
TableHandleParam ttHandle = new TableHandleParam();
ttHandle.set_any(msgArrayIn);
...
```
Processing schema and data for output TABLE-HANDLE parameters

This example shows how you might process the data from an output TABLE-HANDLE parameter by using the sample TABLE-HANDLE helper classes:

```java
// Process the Output TABLE-HANDLE parameter
ttHandle = ttHolder.value;

MessageElement[] msgArrayOut = ttHandle.get_any();

// msgArrayOut has one entry, the DataSet
// Get the DOM representation of the MessageElement
Element dataSetOut = msgArrayOut[0].getAsDOM();

// Create a new RowSet Object based on the DataSet
RowSet rsOut = new RowSet();

/*
Call the buildRowSet method on the RowSet object.
builtRowSet creates a SchemaParser object to parse the <schema> element of the
dataSetOut and creates the m_schema ColumnMetaData[ ] array representing the
schema. builtRowSet then parses the <Data> element and creates the m_rows
Vector representing the data.
*/
rsOut.buildRowSet(dataSetOut);

// print out the data for the dynamic temp-table to the console
rsOut.printData();
```

Releasing an object

You can manage the object IDs of the two objects about to be released by using the object ID handler (PscObjectIDHandler), as shown here.

To release the CustomerOrder ProcObject and the OrderInfo AppObject:

1. Release the logical connection. For example, to release the CustomerOrder ProcObject, call this method:
   ```java
custOrder.release_CustomerOrder( );
```

2. Remove (release) the ProcObject ID from the object ID handler, so this ProcObject ID cannot be used again, as shown:
   ```java
custOrderControlBlock.releaseObjectNameID("CustomerOrder");
```

3. Release the logical connection for the OrderInfo AppObject, as shown:
   ```java
orderInfo.release_OrderInfo( );
```
4. Remove (release) the AppObject ID from the object ID handler, so this AppObject ID cannot be used again. For example:

```java
orderInfoControlBlock.releaseObjectNameID("OrderInfo");
```

**Handling errors on the client**

To understand how to handle SOAP fault messages returned by the WSA, first note that the `<FaultDetail>` element is described in the interface Java class, servicehost/FaultDetail.java. For example:

```java
public class FaultDetail {
    java.lang.String errorMessage;
    java.lang.String requestID;
    ...
}
```

Place a try...catch block around any code that accesses the Web server and catch the AxisFault exception.

This is an example of Axis client code to catch a SOAP fault (an AxisFault exception):

```java
try {
    // code to Access the Web Service
} catch (AxisFault e) {
    // Get the (single) child element of the SOAP Fault <detail> element, // the <FaultDetail> element.
    Element[] faultDetails = e.getFaultDetails( );
    if (faultDetails.length > 0) {
        // Now get the contents of the <FaultDetail> element, // <errorMessage> and <requestID>.
        Node faultDetailElem = faultDetails[0].getFirstChild( );
        String detailErrorMsg = faultDetailElem.getFirstChild( ).getNodeValue( );
        System.err.println("Exception errorMessage: " + detailErrorMsg);
        Node requestIDElem = faultDetailElem.getNextSibling( );
        String requestID = requestIDElem.getFirstChild( ).getNodeValue( );
        System.err.println("Exception requestID : " + requestID);
    }
}
```

**Caution:** In any catch block where you exit the program, you must release all Web service objects you created in the program.
Compiling and running the client application

To compile and run the application:

1. Set up your Classpath as specified in the Axis User’s Guide. Make sure you put the path to your client interface on the Classpath.

2. Compile both the interface and your client application classes using the Java compiler.

3. Run the application, as usual, in the JVM.
ProDataSet parameters in Java Web service clients

This section provides basic information for handling ProDataSet parameters in Java.

**Note:** This section does not continue the sample application in the rest of this appendix.

Table B–6 describes the specifications for a sample Web service and interface used in this section.

**Table B–6: Java sample Web service specifications**

<table>
<thead>
<tr>
<th>Property or component</th>
<th>Value or name</th>
<th>Object type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web service</td>
<td>CustOrdersService</td>
<td>–</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://servicehost:80/wsa/wsa1">http://servicehost:80/wsa/wsa1</a></td>
<td>–</td>
</tr>
<tr>
<td>Session model</td>
<td>Session-Free</td>
<td>–</td>
</tr>
<tr>
<td>TargetNamespace</td>
<td>urn:CustOrders</td>
<td>–</td>
</tr>
<tr>
<td>WSDL objects</td>
<td>CustOrdersObj</td>
<td>AppObject</td>
</tr>
<tr>
<td></td>
<td>dsCustOrd</td>
<td>ProDataSet dsCustOrd</td>
</tr>
</tbody>
</table>

In general, the other information presented in the previous example for creating a client interface and developing a client application applies here as well. Except where the differences in the session model affect things, you would complete the same tasks to build this sample.
The code samples in this section rely on classes generated by Apache Axis from a Doc/Lit WSDL for the following ABL code snippet:

```abl
/* 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 */
...
```

**Proxy classes**

Table B–7 shows the classes created for the dsCustOrd ProDataSet by Apache Axis. Axis also creates classes similar to those described for the previous example in Table B–6.

<table>
<thead>
<tr>
<th>Web service components</th>
<th>Class</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProDataSet output parameter</td>
<td>DsCustOrd</td>
<td>.../CustOrders/DsCustOrd.java</td>
</tr>
<tr>
<td>Constituent temp-tables</td>
<td>DsCustOrdTtCust</td>
<td>.../CustOrders/DsCustOrdTtCust.java</td>
</tr>
<tr>
<td></td>
<td>DsCustOrdTtOrder</td>
<td>.../CustOrders/DsCustOrdTtOrder.java</td>
</tr>
<tr>
<td></td>
<td>DsCustOrdTtOrderTtOrderLine</td>
<td>.../CustOrders/DsCustOrdTtOrderLine.java</td>
</tr>
<tr>
<td>Holder class for output ProDataSet</td>
<td>DsCustOrdHolder</td>
<td>.../CustOrders/holders/DsCustOrdHolder.java</td>
</tr>
</tbody>
</table>
DATASET (static ProDataSet) parameters

In Java, the client interface for a DATASET parameter:

- Represents a DATASET object as arrays of the constituent temp-tables’ row objects. For example:

  ```java
  public class DsCustOrd implements java.io.Serializable {
    private CustOrders.CustOrders.DsCustOrdTtCust[] ttCust;
    private CustOrders.CustOrders.DsCustOrdTtOrder[] ttOrder;
    ...
  }
  ```

- Represents an output DATASET parameter using an array holder class. For example:

  ```java
  public final class DsCustOrdHolder implements javax.xml.rpc.holders.Holder {
    public CustOrders.CustOrders.DsCustOrd value;
    public DsCustOrdHolder() {
    }
    public DsCustOrdHolder(CustOrders.CustOrders.DsCustOrd value) {
      this.value = value;
    }
  }
  ```

- Represents a NESTED DATA-RELATION by embedding an array of the child temp-table’s row objects in the parent temp-table definition. For example:

  ```java
  public class DsCustOrdTtOrder implements java.io.Serializable {
    private java.lang.Integer orderNum;
    private java.lang.Integer custNum;
    private CustOrders.CustOrders.DsCustOrdTtOrderTtOrderLine[] ttOrderLine;
    ...
  }
  ```

The following Java application is an example of handling the DsCustOrd parameter in a Java client:

Sample Java application for getCustOrders (1 of 2)

```java
import java.net.URL;
import javax.xml.rpc.holders.*;
import CustOrders.*;
import CustOrders.CustOrders.*;
import CustOrders.CustOrders.holders.*;

public class CustOrders {
  public static void main(String[] args) {
```
Before-image data

The WSDL represents static ProDataSet parameters with before-image data as arbitrary complex data in an XML Schema <any> element. Apache Axis represents the <any> element as a Java class with a MessageElement[] member that contains the DOM tree for the ProDataSet. The client developer has to parse the XML to build a data set.
DatASET-HANDLE (dynamic ProDataSet) parameters

In Java, the client interface for a DATASET-HANDLE parameter is based on the following:

- For every Web service object containing a method that passes a DATASET-HANDLE parameter, the WSDL contains a <DatasetHandleParam> element defined as a <complexType> definition. Therefore, a DatasetHandleParam class is created for every client object which contains a method that passes a DATASET-HANDLE parameter.

- For input DATASET-HANDLE parameters, the client must create a DatasetHandleParam object, consisting of a MessageElement array containing the XML Schema and data for the DATASET-HANDLE. For output DATASET-HANDLE parameters, the client must parse the XML Schema and data in the MessageElement array.

The following code snippet reads an OUTPUT DATASET-HANDLE from a Web service, outDynDS:

Reading OUTPUT DATASET-HANDLE in Java

```java
StringHolder retval = new StringHolder();
DataSetHandleParamHolder dsParamH = new DataSetHandleParamHolder();
// call outDyn.p on appserver
appobj.outDyn(retval, dsParamH);
DataSetHandleParam dsParam = dsParamH.value;
MessageElement domElem[] = new MessageElement[2];
domElem = dsParam.get_any();
    // Extract the schema, followed by the data.
    MessageElement schemaElem = domElem[0];
    // parse the XML Schema for the Dataset
    MessageElement dataElem = domElem[1];
    // parse the XML Data for the Dataset
```

Before-image data

The WSDL represents dynamic ProDataSet parameters with before-image data as arbitrary complex data in an XML Schema <any> element. Apache Axis represents the <any> element as a Java class with a MessageElement[] member that contains the DOM tree for the ProDataSet. The client developer has to parse the XML to build a data set.
Many elements of ABL support consumption of 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 for consuming a Web service
- Statements for consuming Web services
- Attributes, methods, and events for consuming Web services
### Handles for consuming a Web service

Table C–1 lists the ABL handles that are either valid only for consuming a Web service or have special application in Web service client programming.

**Table C–1: Handles to consume Web services**  

<table>
<thead>
<tr>
<th>ABL handle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous request object handle.</td>
<td>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.</td>
</tr>
<tr>
<td>ERROR-STATUS system handle.</td>
<td>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.</td>
</tr>
<tr>
<td>Procedure object handle.</td>
<td>A handle to a type of procedure object (<em>Web service procedure object</em>) 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.</td>
</tr>
<tr>
<td>SELF system handle.</td>
<td>In the context of an asynchronous event procedure, returns the asynchronous request object handle of the completed request for which the event procedure is executing.</td>
</tr>
<tr>
<td>Server object handle.</td>
<td>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.</td>
</tr>
<tr>
<td>SESSION system handle.</td>
<td>Maintains ABL session status, including a list of all server objects created by the session to bind Web services.</td>
</tr>
<tr>
<td>SOAP fault object handle.</td>
<td>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.</td>
</tr>
<tr>
<td>SOAP fault-detail object handle.</td>
<td>A handle to an object that references any <code>&lt;detail&gt;</code> 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.</td>
</tr>
<tr>
<td>ABL handle</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SOAP header object handle.</td>
<td>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.</td>
</tr>
<tr>
<td>SOAP header-entryref object handle.</td>
<td>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.</td>
</tr>
<tr>
<td>X-document object handle.</td>
<td>A handle to an XML document object created for any SOAP message content that you must access as XML.</td>
</tr>
<tr>
<td>X-noderef object handle.</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Statements for consuming Web services

Table C–2 lists the ABL statements that are either valid only for consuming a Web service or have special application in Web service client programming.

**Table C–2: Statements to consume Web services** (1 of 2)

<table>
<thead>
<tr>
<th>ABL statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE SERVER <code>server-handle</code></td>
<td>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.</td>
</tr>
<tr>
<td>CREATE SOAP-HEADER <code>header-obj-handle</code></td>
<td>Creates a SOAP header object used to add a SOAP header to a pending SOAP request message generated by an OpenEdge Web service invocation.</td>
</tr>
<tr>
<td>CREATE SOAP-HEADER-ENTRYREF <code>soap-obj-handle</code></td>
<td>Creates a SOAP header-entryref object used to add SOAP header entries to an existing SOAP header.</td>
</tr>
<tr>
<td>DELETE OBJECT <code>handle</code></td>
<td>Deletes certain objects, including server objects, procedure objects, SOAP header objects, SOAP header-entryref objects, and asynchronous request objects.</td>
</tr>
<tr>
<td>DELETE PROCEDURE <code>procedure-handle</code></td>
<td>Deletes the procedure object associated with a Web service.</td>
</tr>
<tr>
<td>FUNCTION <code>operationName</code> [ RETURNS <code>dataType</code> ] (parameter [ , parameter ] ... ) IN <code>hPortType</code> .</td>
<td>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.</td>
</tr>
<tr>
<td><code>return = operationName</code> [ ( parameter [ , parameter ] ... ) ] .</td>
<td>Invokes a Web service operation defined as a user-defined function, where <code>return</code> is a variable to receive the value of the operation <code>&lt;return&gt;</code> parameter element, <code>operationName</code> is the name of the operation as specified in the WSDL file and whose ABL prototype is defined using the FUNCTION statement, and <code>parameter</code> 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.</td>
</tr>
<tr>
<td>PROCESS EVENTS</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Statements to consume Web services

<table>
<thead>
<tr>
<th>ABL statement</th>
<th>Description</th>
</tr>
</thead>
</table>
| **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` ] ]  
[ ( `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. |
| **WAIT-FOR** ... | Handles any pending PROCEDURE-COMPLETE events for asynchronous requests by forcing the event procedures 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 for consuming Web services

Table C–3 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>
<thead>
<tr>
<th>ABL attribute, method, or event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ACTOR</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>ADD-HEADER-ENTRY( )</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
| **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. |
<p>| <strong>CLIENT-CONNECTION-ID</strong>        | 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. |</p>
<table>
<thead>
<tr>
<th>ABL attribute, method, or event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLETE</td>
<td>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.</td>
</tr>
<tr>
<td>CONNECT(</td>
<td>A LOGICAL method on the server object handle that logically connects (binds) and associates a Web service to the server handle. All method parameters except the <code>connection-parameters</code> parameter are ignored for a Web service.</td>
</tr>
<tr>
<td>[ connection-parameters ]</td>
<td></td>
</tr>
<tr>
<td>[ , userid ]</td>
<td></td>
</tr>
<tr>
<td>[ , password ]</td>
<td></td>
</tr>
<tr>
<td>[ , appserver-info ]</td>
<td></td>
</tr>
<tr>
<td>CONNECTED( )</td>
<td>A LOGICAL method on the server object handle that returns TRUE if the handle is currently bound to a Web service.</td>
</tr>
<tr>
<td>DELETE-HEADER-ENTRY( )</td>
<td>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.</td>
</tr>
<tr>
<td>DISCONNECT( )</td>
<td>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.</td>
</tr>
<tr>
<td>ERROR</td>
<td>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.</td>
</tr>
<tr>
<td>ERROR-OBJECT-DETAIL</td>
<td>A HANDLE attribute on 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.</td>
</tr>
<tr>
<td>EVENT-PROCEDURE</td>
<td>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.</td>
</tr>
<tr>
<td>EVENT-PROCEDURE-CONTEXT</td>
<td>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.</td>
</tr>
<tr>
<td>FIRST-ASYNC-REQUEST( )</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table C–3: Attributes, methods, and events to consume Web services (3 of 7)

<table>
<thead>
<tr>
<th>ABL attribute, method, or event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST-PROCEDURE</td>
<td>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.</td>
</tr>
<tr>
<td>FIRST-SERVER</td>
<td>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.</td>
</tr>
<tr>
<td>GET-HEADER-ENTRY( )</td>
<td>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.</td>
</tr>
<tr>
<td>GET-NODE( )</td>
<td>A LOGICAL method that:</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td>GET-SERIALIZED( )</td>
<td>A LONGCHAR method that:</td>
</tr>
<tr>
<td></td>
<td>• On the SOAP header-entryref object handle, returns the XML for the underlying SOAP header entry in serialized form.</td>
</tr>
<tr>
<td></td>
<td>• On the SOAP fault-detail object handle, returns the XML for the underlying SOAP fault detail information in serialized form.</td>
</tr>
<tr>
<td>LAST-ASYNC-REQUEST( )</td>
<td>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.</td>
</tr>
<tr>
<td>LAST-PROCEDURE</td>
<td>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.</td>
</tr>
<tr>
<td>LAST-SERVER</td>
<td>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.</td>
</tr>
<tr>
<td>LOCAL-NAME</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table C–3: Attributes, methods, and events to consume Web services (4 of 7)

<table>
<thead>
<tr>
<th>ABL attribute, method, or event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUST-UNDERSTAND</td>
<td>A LOGICAL attribute on the SOAP header-entryref object handle that returns the value of the mustUnderstand attribute specified in the associated SOAP header.</td>
</tr>
<tr>
<td>NAME</td>
<td>A CHARACTER attribute that:</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• On a SOAP header object handle, returns the qualified name of the SOAP header (&quot;namespacePrefix:HEADER&quot;, where namespacePrefix is usually &quot;SOAP&quot; or &quot;SOAP-ENV&quot;).</td>
</tr>
<tr>
<td></td>
<td>• On a SOAP header-entryref object handle, returns the qualified name of the SOAP header entry (&quot;namespacePrefix:localName&quot;).</td>
</tr>
<tr>
<td>NAMESPACE-URI</td>
<td>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.</td>
</tr>
<tr>
<td>NEXT-SIBLING</td>
<td>A HANDLE attribute that:</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• 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).</td>
</tr>
<tr>
<td>NUM-HEADER-ENTRIES</td>
<td>An INTEGER attribute on the SOAP header object handle that returns the number of entries attached to the SOAP header.</td>
</tr>
<tr>
<td>ABL attribute, method, or event</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>PERSISTENT</strong></td>
<td>A LOGICAL attribute on procedure handles that returns TRUE for a procedure object associated with a Web service.</td>
</tr>
<tr>
<td><strong>PERSISTENT–PROCEDURE</strong></td>
<td>A HANDLE attribute on the asynchronous request object handle that returns the handle to the procedure object for the specified asynchronous Web service request.</td>
</tr>
</tbody>
</table>
| **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–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. |
| **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. |
<table>
<thead>
<tr>
<th>ABL attribute, method, or event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET-ACTOR( )</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 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. |
| 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-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. |
### Table C–3: Attributes, methods, and events to consume Web services

<table>
<thead>
<tr>
<th>ABL attribute, method, or event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAP-FAULT-ACTOR</td>
<td>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.</td>
</tr>
<tr>
<td>SOAP-FAULT-DETAIL</td>
<td>A HANDLE attribute on the SOAP fault object handle that references the SOAP fault-detail object, which contains application-specific error information.</td>
</tr>
<tr>
<td>STOP</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 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  
  - "APPSERVER" — For a server object bound to an AppServer  
  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" |
This appendix describes how OpenEdge converts between the following ABL data types and XML Schema data types in parameters passed to 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:

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

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

Table D–1 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 D–1: Supported casts between data types

<table>
<thead>
<tr>
<th>XML Schema data type</th>
<th>ABL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>anyURI</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>base64Binary</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td></td>
<td>MEMPTR</td>
</tr>
<tr>
<td></td>
<td>RAW</td>
</tr>
<tr>
<td>boolean</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>byte</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>date</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>DATETIME</td>
</tr>
<tr>
<td></td>
<td>DATETIME-TZ</td>
</tr>
<tr>
<td>dateTime</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>DATETIME</td>
</tr>
<tr>
<td></td>
<td>DATETIME-TZ</td>
</tr>
<tr>
<td>decimal</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
</tbody>
</table>
Table D–1:  Supported casts between data types

<table>
<thead>
<tr>
<th>XML Schema data type</th>
<th>ABL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>duration</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>ENTITIES</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>ENTITY</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>float</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>gDay</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>gMonth</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>gMonthDay</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>gYear</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>hexBinary</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td></td>
<td>MEMPTR</td>
</tr>
<tr>
<td></td>
<td>RAW</td>
</tr>
<tr>
<td>ID</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>IDREF</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>IDREFS</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>int</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
</tbody>
</table>
### Table D–1: Supported casts between data types

<table>
<thead>
<tr>
<th>XML Schema data type</th>
<th>ABL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>Language</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>long</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>Name</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>NCName</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>negativeInteger</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>NMTOKEN</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>NMTOKENS</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>nonNegativeInteger</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>nonPositiveInteger</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>normalizedString</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>DATETIME</td>
</tr>
<tr>
<td></td>
<td>DATETIME-TZ</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>NOTATION</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
</tbody>
</table>
Table D–1: Supported casts between data types

<table>
<thead>
<tr>
<th>XML Schema data type</th>
<th>ABL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>positiveInteger</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>qName</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>short</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>string</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>DATETIME</td>
</tr>
<tr>
<td></td>
<td>DATETIME-TZ</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>time</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>token</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>unsignedByte</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>unsignedInt</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>unsignedLong</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>unsignedShort</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INT64</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
</tbody>
</table>
### CHARACTER or LONGCHAR

Table D–2 describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL CHARACTER or LONGCHAR type and XML Schema.

<table>
<thead>
<tr>
<th>For this XML Schema type . . .</th>
<th>In this ABL parameter mode . . .</th>
<th>OpenEdge . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>All XML Schema types where CHARACTER is the suggested ABL mapping (see the “Mapping XML Schema data types to ABL data types” section on page 6–17).</td>
<td>INPUT¹</td>
<td>Attempts to format the ABL string value to match the specified XML Schema type for inclusion in the SOAP request message.²</td>
</tr>
<tr>
<td>All XML Schema types where CHARACTER or LONGCHAR is not the suggested ABL mapping (see the “Mapping XML Schema data types to ABL data types” section on page 6–17).</td>
<td>OUTPUT¹</td>
<td>Converts the value from the formatting of the specified XML Schema type before copying it into the ABL parameter as a string value.²</td>
</tr>
<tr>
<td>All XML Schema types where CHARACTER or LONGCHAR is not the suggested ABL mapping (see the “Mapping XML Schema data types to ABL data types” section on page 6–17).</td>
<td>OUTPUT¹</td>
<td>Copies the XML Schema value directly from the SOAP response message to the ABL parameter unchanged from its XML Schema format.</td>
</tr>
</tbody>
</table>

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

2. The process of serializing and de-serialized CHARACTER data inserts and removes XML character references, for example, replacing ”<” with ”&lt;”.

---

---
Table D–3 describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL DATE type and XML Schema.

**Table D–3: ABL data type cast—DATE**

<table>
<thead>
<tr>
<th>For this XML Schema type . . .</th>
<th>In this ABL parameter mode . . .</th>
<th>OpenEdge . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>string normalizedString</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>dateTime</td>
<td>INPUT</td>
<td>Serializes the ABL parameter according to the XML Schema serialization rules, with the time assumed to be 12:00 midnight local time.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>date</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules with no time zone.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
</tbody>
</table>
DATETIME

Table D–4 describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL DATETIME type and XML Schema.

Table D–4: ABL data type cast—DATETIME

<table>
<thead>
<tr>
<th>For this XML Schema type . . .</th>
<th>In this ABL parameter mode . . .</th>
<th>OpenEdge . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>normalizedString</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dateTime</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules without a time zone.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>date</td>
<td>INPUT</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
</tbody>
</table>
DATETIME-TZ

Table D–5 describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL DATETIME-TZ type and XML Schema.

### Table D–5: ABL data type cast—DATETIME-TZ

<table>
<thead>
<tr>
<th>For this XML Schema type . . .</th>
<th>In this ABL parameter mode . . .</th>
<th>OpenEdge . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>string normalizedString</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>dateTime</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules including the time zone.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>De-serializes the XML Schema value from the SOAP response message into the ABL parameter according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td>date</td>
<td>INPUT</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table D-6 describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL DECIMAL type and XML Schema.

Table D-6: ABL data type cast—DECIMAL

<table>
<thead>
<tr>
<th>For this XML Schema type . . .</th>
<th>In this ABL parameter mode . . .</th>
<th>OpenEdge . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>string normalizedString</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>boolean</td>
<td>INPUT</td>
<td>Serializes an ABL DECIMAL value of 0 to false and all other values for the ABL parameter to true.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>decimal</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>float double</td>
<td>INPUT</td>
<td>Serializes the ABL DECIMAL value according to XML Schema serialization rules. Some of the least significant digits might be lost.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>integer</td>
<td>INPUT</td>
<td>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.</td>
</tr>
<tr>
<td>nonPositiveInteger long int short byte nonNegativeInteger unsignedLong unsignedInt unsignedShort unsignedByte positiveInteger</td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table D–8 describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL INT64 type and XML Schema.

<table>
<thead>
<tr>
<th>For this XML Schema type . . .</th>
<th>In this ABL parameter mode . . .</th>
<th>OpenEdge . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>string normalizedString</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the XML Schema value as if the ABL STRING( ) function was called on the INT64 value.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>De-serializes the XML Schema value as if the INT64( ) function was called on string value.</td>
</tr>
<tr>
<td>boolean</td>
<td>INPUT</td>
<td>Serializes an INT64 value of 0 to false and all other values to true.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>decimal</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>float</td>
<td>INPUT</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>double</td>
<td>INPUT</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table D-7: ABL data type cast—INT64

<table>
<thead>
<tr>
<th>For this XML Schema type . . .</th>
<th>In this ABL parameter mode . . .</th>
<th>OpenEdge . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the serialization rules for the XML Schema data type.</td>
</tr>
<tr>
<td>nonPositiveInteger</td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>negativeInteger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td>nonNegativeInteger</td>
<td>OUTPUT</td>
<td>De-serializes the XML Schema value as if the ABL INT64( ) function was called on the value.</td>
</tr>
<tr>
<td>unsignedLong</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsignedInt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>positiveInteger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>INPUT</td>
<td>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.</td>
</tr>
<tr>
<td>short</td>
<td>OUTPUT</td>
<td>De-serializes the XML Schema value as if the ABL INT64( ) function was called on the value.</td>
</tr>
<tr>
<td>byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsignedShort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsignedByte</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(2 of 2)
Table D–8 describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL INTEGER type and XML Schema.

### Table D–8: ABL data type cast—INTEGER

<table>
<thead>
<tr>
<th>For this XML Schema type . . .</th>
<th>In this ABL parameter mode . . .</th>
<th>OpenEdge . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>string normalizedString</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the XML Schema value as if the ABL STRING( ) function was called on the INTEGER value.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>De-serializes the XML Schema value as if the INTEGER( ) function was called on string value.</td>
</tr>
<tr>
<td>boolean</td>
<td>INPUT</td>
<td>Serializes an INTEGER value of 0 to false and all other values to true.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>decimal</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>float</td>
<td>INPUT</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>double</td>
<td>INPUT</td>
<td>Serializes the ABL INTEGER value to a double according to the XML Schema rules.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table D–8: ABL data type cast—INTEGER

<table>
<thead>
<tr>
<th>For this XML Schema type . . .</th>
<th>In this ABL parameter mode . . .</th>
<th>OpenEdge . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the serialization rules for the XML Schema data type.</td>
</tr>
<tr>
<td>nonPositiveInteger</td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>negativeInteger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonNegativeInteger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsignedLong</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsignedInt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>positiveInteger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td>short</td>
<td>OUTPUT</td>
<td>De-serializes the XML Schema value as if the ABL INTEGER( ) function was called on the value.</td>
</tr>
<tr>
<td>byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsignedShort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsignedByte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>INPUT</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>De-serializes the XML Schema value as if the ABL INTEGER( ) function was called on the value.</td>
</tr>
</tbody>
</table>

For this XML Schema type . . .

In this ABL parameter mode . . .

OpenEdge . . .
Table D–9 describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL LOGICAL type and XML Schema.

### Table D–9: ABL data type cast—LOGICAL

<table>
<thead>
<tr>
<th>For this XML Schema type . . .</th>
<th>In this ABL parameter mode . . .</th>
<th>OpenEdge . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>string normalizedString</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>De-serializes the XML Schema string type as if it was a serialized boolean value.</td>
</tr>
<tr>
<td>boolean</td>
<td>INPUT</td>
<td>Serializes the ABL parameter into the SOAP request message according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>De-serializes the XML Schema value into the ABL parameter according to the XML Schema serialization rules.</td>
</tr>
<tr>
<td>decimal float double</td>
<td>INPUT</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>integer long int short byte</td>
<td>INPUT</td>
<td>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.</td>
</tr>
<tr>
<td>nonPositiveInteger unsignedLong unsignedIn unsignedShort unsignedByte</td>
<td>OUTPUT</td>
<td>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.</td>
</tr>
</tbody>
</table>
MEMPTR or RAW

Table D–10 describes the supported castings in each ABL parameter mode (INPUT and OUTPUT) between the ABL MEMPTR or RAW type and XML Schema.

### Table D–10: ABL data type cast—RAW or MEMPTR

<table>
<thead>
<tr>
<th>For this XML Schema type . . .</th>
<th>In this ABL parameter mode . . .</th>
<th>OpenEdge . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64Binary</td>
<td><strong>INPUT</strong></td>
<td>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.</td>
</tr>
<tr>
<td>hexBinary</td>
<td><strong>OUTPUT</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
Understanding WSDL Details

This appendix includes expanded information on WSDL files for OpenEdge Web services. Only certain users need this level of detail. This appendix covers the following topics:

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

```ABL
/* 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

```xml
<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

```xml
<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**

```xml
<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"/>
    </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**

```xml
<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 WSA instance is specified here for the location of the Web service.
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 CustomerAO AppObject with the following signature:

arraySample procedure signature

```abl
/* 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

```xml
<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

```xml
<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>
  </schema>
</wsdl:types>
```
Array mapping in WSDL documents

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

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
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

```xml
<complexType name="ArrayOfString">
  <sequence>
    <element name="item" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</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 (1 of 2)

```xml
<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>
    </import>
    <complexType name="ArrayOfXMLType">
      <complexContent>
        <restriction base="soapenc:Array">
          <attribute ref="soapenc:arrayType" wsdl:arrayType="xmlType[]"/>
        </restriction>
      </complexContent>
    </complexType>
  </schema>
</wsdl:types>
```
Array mapping in WSDL documents

RPC/Encoded WSDL schema example (2 of 2)

```
<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>

<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>
```
Defining 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**

```xml
<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**

```xml
<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**

```xml
<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 Table E–1.

### Table E–1: XML data types for TABLE parameter columns

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>XML Schema data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB(^1)</td>
<td><code>xsd:base64Binary</code></td>
</tr>
<tr>
<td>CHARACTER</td>
<td><code>xsd:string</code></td>
</tr>
<tr>
<td>CLOB(^1)</td>
<td><code>xsd:string</code></td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td><code>xsd:long</code></td>
</tr>
<tr>
<td>DATE</td>
<td><code>xsd:date</code></td>
</tr>
<tr>
<td>DATETIME</td>
<td><code>xsd:dateTime</code></td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td><code>xsd:dateTime</code></td>
</tr>
<tr>
<td>DECIMAL</td>
<td><code>xsd:decimal</code></td>
</tr>
<tr>
<td>INT64</td>
<td><code>xsd:long</code></td>
</tr>
<tr>
<td>INTEGER (32 bit)</td>
<td><code>xsd:int</code></td>
</tr>
<tr>
<td>LOGICAL</td>
<td><code>xsd:boolean</code></td>
</tr>
<tr>
<td>RAW</td>
<td><code>xsd:base64Binary</code></td>
</tr>
<tr>
<td>RECID (32 or 64 bit)</td>
<td><code>xsd:long</code></td>
</tr>
<tr>
<td>ROWID</td>
<td><code>xsd:base64Binary</code></td>
</tr>
<tr>
<td>WIDGET-HANDLE</td>
<td><code>xsd:long</code></td>
</tr>
</tbody>
</table>

\(^1\) 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.
Defining 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:

```
<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 Table E–2.

Table E–2: XML data types for TABLE-HANDLE parameter columns

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>XML Schema data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>xsd:string</td>
</tr>
<tr>
<td>DATE</td>
<td>xsd:date</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>xsd:dateTime</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>xsd:decimal</td>
</tr>
<tr>
<td>INT64</td>
<td>xsd:long</td>
</tr>
<tr>
<td>INTEGER (32 bit)</td>
<td>xsd:int</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>xsd:boolean</td>
</tr>
<tr>
<td>RAW</td>
<td>xsd:base64Binary</td>
</tr>
</tbody>
</table>
Defining 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**

```xml
<!-- 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>
</element>
```
Defining DATASET (static ProDataSet) parameters

**Note:** \(S\)n 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, \(PD\)Sname\(Param\), where \(PD\)Sname 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**

```xml
<!-- 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"/>
        </sequence>
      </complexType>
    </element>
    <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>
```

E-13
Including 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:

```sql
/* 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 (1 of 2)**

```xml
<!-- 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="BlttCust">
        <complexType>
          <sequence>
            <element name="CustNum" nillable="true" type="xsd:int"/>
            <element name="Name" nillable="true" type="xsd:string"/>
          </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.

**Using 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.
Defining 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

```xml
<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>`:

```xml
<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.
This appendix describes the command syntax for running the WSDL Analyzer (bprowsd1doc).
bprowsdldoc

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 programmer can access the Web service and its operations in ABL.

Syntax

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
<td>bprowsdldoc [-h ]</td>
</tr>
<tr>
<td>Windows</td>
<td>bprowsdldoc [-h ]</td>
</tr>
<tr>
<td></td>
<td>{ [-b</td>
</tr>
<tr>
<td></td>
<td>-WSDL_userid username</td>
</tr>
<tr>
<td></td>
<td>-proxy host</td>
</tr>
<tr>
<td></td>
<td>-proxy_userid username</td>
</tr>
<tr>
<td></td>
<td>-show100style</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>wsdl-url-or-filename [ target-directory ]</td>
</tr>
</tbody>
</table>

-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. For more information on connecting to Web services, see Chapter 7, “Connecting to Web Services from ABL.”

-nohostverify

Turns off host verification for a Secure Socket Layer (SSL) connection using HTTPS.

-nosessionreuse

Prevents any reuse of the SSL session ID for an HTTPS Web server connection when reconnecting the same Web server using HTTPS.

-WSDL_userid username

Specifies a username for accessing the WSDL file.
-WSDLPassword password

Specifies a password for accessing the WSDL file.

-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 the “Analyzing wrapped document literal” section on page 6–13.

-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.

For more information on the documentation output from the WSDL Analyzer, see the “Using the WSDL Analyzer” section on page 6–3.
Index

A

ABL applications
accessing Web service port types 7–8
alternate mappings to XML Schema D–2
analyzing WSDL for 6–3
AppServer compared to Web service access 1–13
asynchronous Web service requests 8–12
attributes and methods for Web service consumption C–6
binding to Web services 7–3
connecting to Web services 7–3
debugging Web service access 10–12
handling SOAP faults 10–2
handling SOAP headers 9–6
handling Web service errors 10–1
HTTPS 7–6
invoking Web service operations 8–1
managing SOAP headers 9–1
managing Web service bindings 7–8
object handles for Web service consumption C–2
sample of consuming Web services 1–15
SOAP header object model 9–4
statements for Web service consumption C–4
Web service client architecture 1–11
Web service connection parameters 7–3
Web service procedure object 8–2

ABL callbacks 9–6

ABL for consuming Web services
attributes and methods C–6
object handles C–2
statements C–4

AppServer applications
preparing for Web services 2–7
testing for Web services 5–2

Architecture
ABL client of a Web service 1–11
OpenEdge Web services 1–5

Asynchronous requests
ABL clients of Web services 8–12

B

Binding to a Web service
ABL clients 7–3

Binding Web services
managing 7–8

Bindings section
WSDL for OpenEdge Web services 3–4

bprowsddoc command
syntax F–2
using 6–3

C

Callback procedures 9–6

CHARACTER ABL client conversion rules D–6
Client interface methods
  session-free AppObjects 3–10
  session-managed AppObjects 3–8
  SubAppObjects and ProcObjects 3–11

Client interfaces
  affects from SOAP formats 3–5
  documentation for ABL clients 6–5

Client programming
  session-free Web services 3–7
  session-managed Web services 3–6
  Web service objects 3–8

Client programming for SOAP faults 3–28

Client requirements
  OpenEdge Web services 1–9

Complex data
  analysis for ABL clients 6–14
  examples of managing in ABL 8–8
  managing in ABL clients 8–7
  managing operation parameters 8–7

Complex types
  ABL client and XML Schema 6–22

Configuring the WSA 2–8

CONNECT() method
  parameters 7–3

Connecting to a Web service
  ABL clients 7–3

Connection parameters
  ABL client to Web service 7–3

Connections
  ABL client to Web service 7–1
  client to OpenEdge Web service
    session-free 3–10
    session-managed 3–8
  managing ABL client to Web service 7–8

conversion XML Schema and ABL types
  ABL client listings D–2
  detailed ABL client rules D–1

D

Data types
  handling complex types 6–22
  handling simple types 6–17
  parameter mapping 6–17
  parameter mapping in OpenEdge Web services 3–15

DATE ABL client conversion rules D–7

DATETIME ABL client conversion rules D–8

DATETIME-TZ ABL client conversion rules D–9

DECIMAL ABL client conversion rules D–10

Defining client requirements 1–6

Defining the Web service 1–7, 2–8

Definitions section
  WSDL for OpenEdge Web services 3–3

Deploying
  OpenEdge Web services 1–10

Developing OpenEdge Web services 1–5

Document/Literal (Doc/Lit) format
  analyzing wrapped form for ABL clients 6–13
  coding for complex data types
    unwrapped form 8–11
    wrapped form 8–10
  defined 1–4
  OpenEdge Web services 2–6
  unwrapped form 6–13
  wrapped form 6–13

Dynamic ProDataSet parameters
  OpenEdge Web services 3–22, E–17

Dynamic temp-table parameters
  OpenEdge Web services 3–19, E–10

E

ERROR-STATUS system handle
  SOAP fault detection 10–3

F

FUNCTION statement
  Web service operations 8–5

H

Header handlers 9–2
  defining in the ABL 9–8
  request messages 9–10
  response messages 9–9
  invoking in the ABL 9–11
  requirements for in the ABL 9–3

HTTPS in ABL applications 7–6
I

INTEGER ABL client conversion rules
  D–11, D–13

L

Log files
  OpenEdge Web service testing 5–7

LOGICAL ABL client conversion rules
  D–15

LONGCHAR ABL client conversion rules
  D–6

M

Memory management
  SOAP headers in ABL clients 9–26

MEMPTR ABL client conversion rules
  D–16

Messages section
  WSDL for OpenEdge Web services 3–4

N

Namespaces
  Web service 2–3

O

Object IDs 2–2
  client examples 3–14
  defining in WSDL 3–13
  passing in SOAP headers 3–13

Open Client
  object model for OpenEdge Web services
    2–2

OpenEdge Web services 3–22
  architectural overview 1–5
  client programming requirements 1–9
  creating and using ProcObjects 4–9
  deployment 1–10
  deployment information 2–3
  handling errors and SOAP faults 3–28
  handling SOAP headers 3–13
  internal procedures/user-defined functions 4–12
  mapping ABL procedures to SOAP 3–25
  method parameters 3–15
  non-persistent procedures 4–7
  Open Client object model 2–2
  ProDataSet parameters 4–21
  receiving SOAP faults sample 4–20
  releasing objects 4–14
  sample applications 1–17
  sample client development
    .NET A–1
    Java B–1
  session-free access 4–6
  session-free clients 3–7
  session-managed access 4–4
  session-managed clients 3–6
  SOAP formats supported 2–6
  steps for building and deploying 1–5
  temp-table handle parameters 4–17
  temp-table parameters 4–15
  testing and debugging 5–1
  Web service objects 3–8
  WSDL file overview 2–2

P

Performance tuning
  WSA instances 1–10

Port type procedure object 8–2

Port types
  ABL client access 7–8

Port types section
  WSDL for OpenEdge Web services 3–4

ProcObjects in OpenEdge Web services
  3–11

ProDataSet parameters 3–22
  OpenEdge Web services 3–21
  WSDL for OpenEdge Web services
    E–11, E–17

ProDataSets
  complex data in ABL clients 6–22
  SOAP fault detail in ABL clients 10–7
  SOAP headers in ABL clients 9–6

ProSOAPView
  using 10–14

R

RAW ABL client conversion rules D–16

RPC/Encoded format
  defined 1–4
  OpenEdge Web services 2–6

RPC/Literal format
  defined 1–4
  OpenEdge Web services 2–6

RUN statement
  Web service operations 8–4
  Web service procedure object 8–2
Sample applications
OpenEdge Web services 1–17

Services section
WSDL for OpenEdge Web services 3–4

Session models
OpenEdge Web services 2–4
session-free clients 3–7
session-managed clients 3–6

SET-CALLBACK-PROCEDURE()
method 9–6

SOAP
ABL header objects 9–4
ABL header-entryref objects 9–4
action 2–3
format 2–3
formats supported by OpenEdge 1–4, 2–6
message structure overview 1–3
messages for OpenEdge Web services 3–25
standards supported 1–2

SOAP fault object
attributes 10–6

SOAP fault-detail object
attributes and methods 10–6

SOAP faults
ABL clients handling 10–2
client handling 3–28
detecting 10–3
example ABL handling 10–8

SOAP formats
client interface affects 3–5
Document/Literal 1–4
RPC/Encoded 1–4
RPC/Literal 1–4

SOAP header objects
attributes and methods 9–28
model 9–4

SOAP header-entryref objects
attributes and methods 9–29

SOAP headers
ABL client overview 9–2
accessing in the ABL 9–3
client handling
ABL clients 9–1
OpenEdge Web services 3–13
defining ABL handlers 9–8
request messages 9–10
response messages 9–9
invoking ABL handlers 9–11
object model 9–4

setting access for ABL clients 9–6
structure 9–2

SOAP headers in the ABL
attributes and methods 9–28
creating and managing 9–13
creating for initial request 9–22
memory management 9–26
reusing 9–14
reusing with modifications 9–17

SOAP message samples 4–5 to 4–20

SOAP message viewers
ABL clients of Web services 10–12
OpenEdge Web services 5–10

SOAP messages
client method calls 3–27
creating and using ProcObjects sample 4–9
internal procedures/user-defined
functions sample 4–12
non-persistent procedures sample 4–7
sample of receiving SOAP faults 4–20
session-free access sample 4–6
session-managed access sample 4–4
temp-table parameters sample 4–15

Static ProDataSet parameters
OpenEdge Web services 3–21, E–11

Static temp-table parameters
OpenEdge Web services 3–18, E–8

SubAppObjects in OpenEdge Web services 3–11

Temp-table parameters
OpenEdge Web services 3–18, 3–19
WSDL for OpenEdge Web services E–8, E–10

Temp-tables
complex data in ABL clients 6–22
SOAP fault detail in ABL clients 10–7
SOAP headers in ABL clients 9–6

Testing
AppServer applications for Web services 5–2
error identification 5–12
log files 5–7
sample error scenarios 5–15
setting error levels 5–4
SOAP faults 5–8
SOAP message viewers 5–10

Types section
WSDL for OpenEdge Web services 3–3
U
Unit testing OpenEdge Web services 5–2

W
Web service objects
calling in OpenEdge Web services 3–8
session models 2–5
Web service operation
parameters 3–15
Web service operations
ABL procedure invocation 8–4
ABL user-defined function invocation 8–5
invoking from the ABL 8–3
managing complex data in ABL clients 8–7
managing parameters 8–6
Web service procedure object 8–2
Web service server objects 7–3
Web services
calling from ABL 1–13
debugging access from the ABL 10–12
session models in OpenEdge 2–4
Web Services Adapter (WSA) 2–8
configuring 2–8
URL 2–3
WSA, See Web Services Adapter (WSA) 2–8
WSAViewer
configuring for the WSA 5–10
using 10–13

WSDL Analyzer
command syntax F–2
complex data analysis 6–14
operation parameter documentation 6–15
sample documentation 6–15
SOAP fault documentation 6–16
SOAP header documentation 6–16
how to use 6–3
HTML documentation 6–5
data types document 6–12
index document 6–7
operation index document 6–9
port type document 6–9
service document 6–8
running 6–3
source for ABL client interface 1–13
Wrapped Doc/Lit convention 6–13

WSDL files
client calls to ABL procedures 3–26
complex data types 6–22
creating ABL clients from 6–1
creating client interfaces from 3–3
defining Object IDs for OpenEdge Web services 3–13
element overview for OpenEdge Web services 3–3
mapping operations to ABL procedures 3–26, E–2
mapping to ABL data types 6–17
OpenEdge Web services 2–2
overview 1–3
simple data types 6–17
sources for obtaining 2–10
standards supported 1–2

X
XML Schema
alternate mappings to ABL client types D–2
mapping client data types to ABL 6–17
mapping parameters to ABL 3–15
standards supported 1–2