**Note**

Before using this information and the product it supports, read the information in “Notices” on page 201.

---

**Second Edition (June 2012)**

This edition applies to version 1 of Web Services Client for ILE and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 2011, 2012.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
Preface

Web Services Client for ILE provides a set of libraries and Java tools that enable you to build Web service client applications from existing Web Service Description Language (WSDL) files. This book describes how to use the IBM® Web Services Client for ILE to create Web service client applications.

This book is structured into four parts:
- Part 1 presents the underlying concepts, architectures, and specifications for the use of Web services, including discussions on the Web services core technologies of XML, SOAP, and WSDL.
- Part 2 presents the underlying concepts and architecture of Web Services Client for ILE.
- Part 3 presents information on generating and using C++ stubs.
- Part 4 presents information on generating and using C stubs.
- Part 5 presents information on generating and using RPG stubs.

Who should read this book?

This book is primarily for application programmers who develop Web service client applications. Some of the information might also be useful to system administrators who manage systems on which Web service applications are developed and deployed.

What you need to know to understand this book

The only thing you need to have is application programming skills in one of the programming languages from the list below:
- RPG
- Cobol
- C
- C++

Conventions used in this book

*Italic* is used for new terms where they are defined.

*Constant width* is used for:
- Program language code listings
- WSDL file listings
- XML listings
- Command lines and options

*Constant width italic* is used for replaceable items in code or commands.

In addition, in order to simplify paths when referring to files or commands in the Web Services Client for ILE install directory, /QIBM/ProdData/OS/Webservices/V1/client, we will use <install_dir> as the initial path in path names to represent the install directory.

About examples in this book

Examples used in this book are kept simple to illustrate specific concepts. Some examples are fragments that require additional code to work.
What has changed in this document

As new features and enhancements are made, the information in this document will get updated. To use any new features or enhancements you should load the latest HTTP Group PTF for your IBM i release. To see what HTTP Group PTF a feature or enhancement is in, go to the IBM Integrated Web Services for i Technology Updates wiki, at URL:


Notes:
1. Sometimes new features or enhancements are not yet part of a group PTF, in which case the wiki will list the PTF number(s) containing the feature or enhancement.
2. To help you see where technical changes have been made since the previous edition, the character | is used to mark new and changed information.

The following lists the changes that have been made to the book since the previous edition:

- June 2012
  - New sections have been added to document new support for setting connect timeout in the `Axis C core APIs` and `Axis C++ core APIs` chapters.
  - The usage notes for the `wsdl2ws.sh` command line tool has been updated to show how one would generate web service client stubs when the transport protocol being used for the URI of the WSDL file is HTTP SSL.
  - New information on C-only APIs that allow the setting of attributes in the SOAP `Header` and `Body` elements have been added to the `Axis C core APIs` chapter.
Part 1. Web services fundamentals

This part of the document introduces Web services concepts and architecture, including a discussion on the core technologies that form the basis of Web services.

- **Chapter 1, “Introduction to Web services,” on page 3** gives a high-level overview of Web services.
- **Chapter 2, “XML primer,” on page 11** is an introduction to XML, an extensible tag language that can describe complicated structures in ways that are easy for programs to understand.
- **Chapter 3, “SOAP primer,” on page 21** is an introduction to SOAP, an XML protocol that is used to describe how to build and process a message that is sent between parties that participate in Web services.
- **Chapter 4, “WSDL primer,” on page 31** is an introduction to WSDL, a set of XML tags that are used to describe the interface to a Web service.

Note that technically you do not need to know the information presented in the chapters in this part of the document to use Web Services Client for ILE. However, references to various Web service technologies and terms are made throughout the book, so at the very least you should skim through the chapters.
Chapter 1. Introduction to Web services

A Web service is a self-contained software component with a well-defined interface that describes a set of operations that are accessible over the Internet. Extensible Markup Language (XML) technology provides a platform—and programming language-independent means by which a Web service's interface can be defined. Web services can be implemented using any programming language, and can be run on any platform, as long as two components are provided to indicate how the Web service can be accessed: a standardized XML interface description, called WSDL (Web Services Description Language), and a standardized XML-based protocol, called SOAP (Simple Object Access Protocol). Applications can access a Web service by issuing requests formatted according to the XML interface.

Web services allow applications to be integrated more rapidly, easily and less expensively than ever before. Integration occurs at a higher level in the protocol stack, based on messages centered more on service semantics and less on network protocol semantics, thus enabling loose integration of business functions. These characteristics are ideal for connecting business functions across the Web - both between enterprises and within enterprises. They provide a unifying programming model so that application integration inside and outside the enterprise can be done with a common approach, leveraging a common infrastructure. The integration and application of Web services can be done in an incremental manner, using existing languages and platforms and by adopting existing legacy applications.

Key technologies and standards for Web services

Several key technologies and standards exist within the Web services community:
- SOAP over HyperText Transfer Protocol (HTTP) is the common protocol for Web services.
- Web Services Description Language (WSDL) is the interface contract language between Web service providers and requestors, and can be used to drive assembly tools, code generators, and other tools to speed integration.
- Web services can be discovered using Universal Description, Discovery, and Integration (UDDI).
- The Web Services - Interoperability organization (WS-I) has developed specifications that have become the foundation for even the most basic Web services. WS-I is an industry consortium chartered to promote standards for interoperability of Web services.
Other current and emerging Web services specifications addressing things like message-level security, addressing, asynchronicity, reliable messaging, transaction management, resource management, and so forth. Because Web services standards are evolving at a rapid pace, it is useful to consult an online reference of Web services standards, such as is hosted on the IBM developerWorks® Web site, available at:


**Interoperability and WS-I Profiles**

The big interoperability question: can Web services continue to interoperate as the various standards they rely on to change over time? From a user perspective, the use of arbitrary collections of Web services technology should not stand in the way of interoperability between Web services.

The WS-I was formed with the intent of promoting standardized interoperability in the Web services marketplace. Without a controlled combination of the various technologies that make up Web services, interoperability would be almost impossible.

Consider the following: Company X has decided to use WSDL Version 1.2, SOAP Version 1.1, and UDDI Version 1 for their Web services. Company Y has decided to use WSDL Version 1.1, SOAP Version 1.2, and UDDI Version 2. Even though both companies are using Web services, a client or registry would need to know about the two different combinations of protocols in order to interact with both. The protocols by themselves are not enough to achieve interoperability. A standardized grouping of the protocols would make it possible for Company X, Company Y, and their clients and registries to adopt a common set of protocols and versions. Without a standardized grouping, the companies and clients can only pick what protocols and versions they think are appropriate according to their unique set of constraints or requirements, and hope that they will be able to communicate with each other.

The WS-I Profile initiative addresses the problem that Companies X and Y are facing. A profile is a grouping of Web services protocols and their versions under a title. By having such a grouping, organizations can negotiate their protocol requirements at more granular levels. Profiles also limit the number of official protocol sets from inestimable to whatever degree of finiteness the WS-I chooses.

As of this writing, WS-I has finalized the Basic Profile, Attachments Profile and Simple SOAP Binding Profile. Work on a Basic Security Profile is currently underway. A summary of the profiles:

- **Basic Profile 1.0**
  The Basic Profile outlines what you must and must not do when using the following common Web services specifications: SOAP 1.1, WSDL 1.1, UDDI 2.0, XML 1.0, XML Schema and HTTP 1.1. The Basic Profile 1.0 provides constraints and clarifications for the base specifications listed above. The Basic Profile addresses the following areas:
  - Protocol elements that are exchanged, usually over a network, to affect a Web service (e.g., SOAP/HTTP messages)
  - Descriptions of types, messages, interfaces, and their concrete protocol and data format bindings, as well as the network access points associated with Web services (e.g., WSDL descriptions)
  - Registry elements that are involved in the registration and discovery of Web services (e.g., UDDI)

- **Basic Profile 1.1 and Simple SOAP Binding Profile**
  The Basic Profile 1.0 does not allow for extensibility in bindings to accommodate SOAP with Attachments (facsimile images of legal documents, engineering drawings, and so forth) and other future extensions. To solve these problems, two new profiles were created: Basic Profile 1.1, which contains everything from the Basic Profile 1.0 except the information about SOAP/HTTP WSDL bindings; and Simple SOAP Binding Profile 1.0, which contains those parts of the Basic Profile 1.0 that relate to the SOAP/HTTP WSDL bindings.
  The Simple SOAP Binding Profile addresses the following areas:
  - Serialization of the soap:Envelope element and its content
- Protocol elements that transport the envelope (e.g., SOAP/HTTP messages)
- Descriptions of types, messages, interfaces, and their concrete protocol and data format bindings, as well as the network access points associated with Web services (e.g., WSDL descriptions)
- Software that implements a `wsdl:port` or a `uddi:bindingTemplate`
- Software that consumes a message according to the protocols associated with it (e.g., SOAP processors)

**Attachments Profile 1.0**

The Attachments Profile 1.0 complements the Basic Profile by adding support for conveying interoperable SOAP with Attachments (SwA) with SOAP messages. The Attachments Profile acts in the following ways:
- Deals with some of the most poorly specified parts of the SOAP and WSDL specifications
- Clarifies MIME bindings in WSDL
- Does not specify ordering of attachments
- Clarifies the SOAP message format for unreferenced attachments
- Addresses attachment references
- Creates a new schema type: `wsi:swaRef`

**Basic Security Profile 1.0**

The Basic Security Profile 1.0 is an interoperability profile that addresses transport security, SOAP messaging security, and other security considerations for the Basic Profile 1.0, as well as the Basic Profile 1.1, Simple SOAP Binding Profile 1.0, and Attachments Profile 1.0. The Basic Security Profile is intended to be used with other WS-I profiles and references existing specifications used to provide security, including the OASIS Web Services Security (WS-Security) 1.0 specification.

The Basic Security Profile 1.0 outlines what you must and must not do to incorporate security into your Web services. The following specifications are addressed in this profile: HTTP, SSL (Secure Socket Layer)/TLS (Transport Layer Security), and WS-Security. The areas addressed by the Basic Security Profile 1.0 are the following:
- Peer identification and authentication
- Data origin and authentication
- Data integrity
- Data confidentiality
- Message uniqueness

---

**Web services model**

The Web services architecture is based upon the interactions between three roles: service provider, service registry and service requestor. The interactions involve the publish, find and bind operations. Together, these roles and operations act upon the Web services artifacts: the Web service software module and its description. In a typical scenario, a service provider hosts a network-accessible software module (an implementation of a Web service). The service provider defines a service description for the Web service and publishes it to a service requestor or service registry. The service requestor uses a find operation to retrieve the service description locally or from the service registry and uses the service description to bind with the service provider and invoke or interact with the Web service implementation. Service provider and service requestor roles are logical constructs and a service can exhibit characteristics of both. Figure 2 on page 6 illustrates these operations, the components providing them and their interactions.
Let us look at the roles in a Web services architecture:

- **Service provider.** From a business perspective, this is the owner of the service. From an architectural perspective, this is the platform that hosts access to the service.

- **Service requestor.** From a business perspective, this is the business that requires certain functions to be satisfied. From an architectural perspective, this is the application that is looking for and invoking or initiating an interaction with a service. The service requestor role can be played by a browser driven by a person or a program without a user interface, for example another Web service.

- **Service registry.** This is a searchable registry of service descriptions where service providers publish their service descriptions. Service requestors find services and obtain binding information (in the service descriptions) for services during development for static binding or during execution for dynamic binding. For statically bound service requestors, the service registry is an optional role in the architecture, because a service provider can send the description directly to service requestors. Likewise, service requestors can obtain a service description from other sources besides a service registry, such as a local file, FTP site or Web site.

For an application to take advantage of Web services, three behaviors must take place: publication of service descriptions, lookup or finding of service descriptions, and binding or invoking of services based on the service description. These behaviors can occur singly or iteratively. In detail, these operations are:

- **Publish.** To be accessible, a service description needs to be published so that the service requestor can find it. Where it is published can vary depending upon the requirements of the application.

- **Find.** In the find operation, the service requestor retrieves a service description directly or queries the service registry for the type of service required. The find operation can be involved in two different lifecycle phases for the service requestor: at design time to retrieve the service's interface description for program development, and at runtime to retrieve the service's binding and location description for invocation.

- **Bind.** Eventually, a service needs to be invoked. In the bind operation the service requestor invokes or initiates an interaction with the service at runtime using the binding details in the service description to locate, contact and invoke the service.
Web services architecture

To perform the three operations of publish, find and bind in an interoperable manner, there must be a Web services stack that embraces standards at each level. Figure 3 shows a conceptual Web services stack. The upper layers build upon the capabilities provided by the lower layers. The vertical towers represent requirements that must be addressed at every level of the stack. The text on the left represents standard technologies that apply at that layer of the stack.

The foundation of the Web services stack is the transport. Web services must be network-accessible to be invoked by a service requestor. Web services that are publicly available on the Internet use commonly deployed network protocols. Because of its ubiquity, HTTP is the defacto standard network protocol for Internet-available Web services. Other Internet protocols can be supported, including SMTP and FTP.

The next layer, messaging, represents the use of XML as the basis for the messaging protocol. SOAP is the chosen XML messaging protocol for many reasons:

- It is the standardized enveloping mechanism for communicating document-centric messages and remote procedure calls using XML.
- It is simple; it is basically an HTTP POST with an XML envelope as payload.
- It is preferred over simple HTTP POST of XML because it defines a standard mechanism to incorporate orthogonal extensions to the message using SOAP headers and a standard encoding of operation or function.
- SOAP messages support the publish, find and bind operations in the Web services architecture.

The service description layer is actually a stack of description documents. First, WSDL is the defacto standard for XML-based service description. This is the minimum standard service description necessary to support interoperable Web services. WSDL defines the interface and mechanics of service interaction. Additional description is necessary to specify the business context, qualities of service and service-to-service relationships.

Figure 3. Conceptual Web services stack
Since a Web service is defined as being network-accessible via SOAP and represented by a service description, the first three layers of the stack are required to provide or use any Web service. The simplest stack would consist of HTTP for the network layer, the SOAP protocol for the XML messaging layer and WSDL for the service description layer. This is the interoperable base stack that all inter-enterprise, or public, Web services should support. Figure 4 depicts the interoperable base stack.

![Figure 4. Interoperable base Web services stack](image)

The stack depicted in Figure 4 provides for interoperability and enables Web services to leverage the existing Internet infrastructure. This creates a low cost of entry to a ubiquitous environment. Flexibility is not compromised by the interoperability requirement, because additional support can be provided for alternative and value-add technologies. For example, SOAP over HTTP must be supported, but SOAP over MQ can be supported as well.

While the bottom three layers of the stack identify technologies for compliance and interoperability, the next two layers - service publication and service discovery - can be implemented with a range of solutions.

Any action that makes a WSDL document available to a service requestor, at any stage of the service requestor’s lifecycle, qualifies as service publication. The simplest, most static example at this layer is the service provider sending a WSDL document directly to a service requestor. This is called direct publication. E-mail is one vehicle for direct publication. Direct publication is useful for statically bound applications. Alternatively, the service provider can publish the WSDL document describing the service to a host local WSDL registry, private UDDI registry or the UDDI operator node.

A Web service cannot be discovered if it has not been published, so service discovery depends upon service publication. The variety of discovery mechanisms at this layer parallels the set of publication mechanisms. Any mechanism that allows the service requestor to gain access to the service description and make it available to the application at runtime qualifies as service discovery. The simplest, most static example of discovery is static discovery wherein the service requestor retrieves a WSDL document from a local file. This is usually the WSDL document obtained through a direct publish or the results of a previous find operation. Alternatively, the service can be discovered at design time or runtime using a local WSDL registry, a private UDDI registry or the UDDI operator node.

Because a Web service’s implementation is a software module, it is natural to produce Web services by composing Web services. A composition of Web services could play one of several roles. Intra-enterprise Web services might collaborate to present a single Web service interface to the public, or the Web services from different enterprises might collaborate to perform machine-to-machine, business-to-business transactions. Alternatively, a workflow manager might call each Web service as it participates in a business process. The topmost layer, service flow, describes how service-to-service communications, collaborations, and flows are performed. Business Process Execution Language (BPEL) is used to describe these interactions.
For a Web services application to meet the stringent demands of today's e-businesses, enterprise-class infrastructure must be supplied, including security, management and quality of service. These vertical towers must be addressed at each layer of the stack. The solutions at each layer can be independent of each other.

**Message exchange patterns**

Web services is a very flexible message-oriented concept—there are no restrictions and very little documented in the standards regarding message exchange patterns (sometimes referred to as interaction patterns).

When it comes to sending messages, you have a wide variety of choices, from sending a request and waiting for a response to sending request and forgetting about it, to sending a request and having it passed through multiple intermediaries before it gets to its final destination. In the end, though, you really have only two choices:

- One-way
- Request-response

The following sections will give conceptual information about the exchange patterns listed above. To see how one would specify a message exchange pattern, see "Bindings" on page 37.

**One-way**

In this very simple message exchange pattern (see Figure 5), sometimes known as the "fire-and-forget" method, messages are pushed in one direction only. The source does not care whether the destination accepts the message (with or without error conditions). The service provider implements a Web service to which the requestor can send messages.

![Figure 5. One-way message exchange pattern](image)

An example of a one-way message exchange pattern is a resource monitoring component. Whenever a resource changes in an application (the source), the new value is sent to a monitoring application (the destination).

**Request-response**

Probably the most common message exchange pattern, a remote procedure call (RPC) or request-response pattern, involves a request message and a synchronous response message (see Figure 6 on page 10). In this message exchange pattern, the underlying transport protocol provides an implicit association between the request message and the response message.

In situations where the message exchange pattern is truly synchronous, such as when an end user is waiting for a response, there is little point in decoupling the consumer and producer. In this situation, the use of SOAP/HTTP as a transport provides the highest level of interoperability.
There are numerous examples of this message exchange pattern, for example, requesting an account balance on a bank account.
Chapter 2. XML primer

XML stands for Extensible Markup Language and it has become one of the most important standard of modern times. XML is a specification developed by the World Wide Web Consortium (W3C) for defining markup languages. XML allows the definition, transmission, validation, and interpretation of data between applications. It is a meta-language: a language for defining other markup languages, interchange formats and message sets. XML is the standard upon which many Web services standards are based and thus we will briefly touch upon some of the more important parts of the specifications as a very quick primer. The entire specification can be studied at the web site of the World Wide Web Consortium at: http://www.w3.org/

Basic rules for creating XML documents

Below is an example of an XML document. XML documents are created with three main XML components: elements, attributes and "text" contents of the elements. XML documents should be defined by a corresponding XML definitional document (for example, an XSD) - not shown here - which will be discussed later.

```xml
<?xml version="1.1"?>
<!-- Complete address tag -->
<Address>
    <Name>
        <Title>Mrs.</Title>
        <First-Name>Ashley</First-Name>
        <Middle-Name/>
        <Last-Name>Adams</Last-Name>
        <Phone>777-444-2222</Phone>
    </Name>
    <Street>123 Corporation Avenue</Street>
    <City state="NC">Hometown</City>
    <Postal-Cde>27709</Postal-Cde>
    <Department>Industrial Design</Department>
</Address>
```

- **XML declaration**: In the above example, line 1 (`<?xml version="1.1"?>`) is the XML declaration that provides basic information about the document to the parser.
- **Tag**: A tag is the text between the left angle bracket (<) and the right angle bracket (>). There are starting tags (such as `<Name>` on line 3) and ending tags (such as `</Name>` on line 5).
- **Element**: An element is the starting tag, the ending tag and everything in between. The `<Name>` element on line 3, contains four child elements: `<Title>`, `<First-Name>`, `<Middle-Name/>` and `<Last-Name>`. Element rules include:
  - There's only one root element in an XML document.
  - The first element is considered the root element. It is also the outermost element, so its end tag is last.
  - Elements must be properly nested and follow well-formed XML code structure.
  - Opening and closing tags cannot cross each other. At any given depth of open tags, it is only valid to close the innermost element (the last one to have been opened at that point).
  - An element does not directly contain characters: consecutive characters are grouped into a “Text” node and the "Text" node is the child of Element. Although "Text" is the official term, schemas can require that a text node actually contain a number, date or other type of data. Schemas can impose similar requirements on attribute values.
• **Attribute**: An attribute is a name-value pair inside the starting tag of an element. On line 6 (<City state="NC">Hometown</City>), state is an attribute of the <City> element. The "NC" is the value of the attribute.

  Attribute rules include:
  - Attributes must have values. However, an attribute can have a value that is an empty string (for example, <House color=""/>).
  - Those values must be enclosed with single or double quotation marks.

• **Comment tag**: Line 2 contains a comment tag. Comments can appear anywhere in the document; they can even appear before or after the root element. A comment begins with <!-- and ends with -->. A comment can not contain a double hyphen (--) except at the end; with that exception, a comment can contain anything.

• **Empty element**: An empty element contains no content. Line 4 contains the markup <Middle-Name/>. There is no middle name so it is empty. The markup could also be written as <Middle-Name/></Middle-Name>. The shorter version still has an ending tag of "/>". An XML parser would treat them in the same way. If your XML document was referencing an XML schema and the XML schema was checking for that element, you would make sure that you included that element in your XML document, but leave it empty if you don't have data.

---

### Naming rules for elements and attribute tags

The following are examples of the naming rules for XML (for a complete list of naming rules, see the W3C XML recommendations):

- A name must consist of at least one letter and can be either upper or lower case.
- XML code is case sensitive. <c> and <C> are considered two different tags.
- You can use an underscore (_) as the first character of a name, if the name consists of more than one character.
- Digits can be used in a name after the first character.
- Colons are used to set off the namespace prefix and should not otherwise be used in a name.

---

### Nesting tags

By nesting tags, XML provides you with the ability to describe hierarchical structures as well as sequence. Nesting requirements mean that a well-formed XML document can be treated as a tree structure of elements. Many XML specs will casually refer to the term XML tree when referring to the structure of elements.

---

### Understanding XML namespace

Namespace is a method of qualifying the element and attribute names used in XML documents by associating them with a Universal Resource Identifier (URI). A URI is a string of characters that identifies an Internet Resource (IR). The most common URI is the Uniform Resource Locator (URL), which identifies an Internet domain address along with other system identifiers. Another, not so common, type of URI is the Universal Resource Name (URN).

An XML namespace is a collection of names identified by a URI reference, which are used in XML documents and defines the scope of the element and attribute names. Element and attribute names defined in the same namespace must be unique.

An XML document can have a default namespace (using `<xmlns=`) and any element can belong to the default, or another specified namespace. The collection of defined elements and attributes within the same namespace are said to be in the same “XML vocabulary.” The example below shows some examples
of the use of namespace:

```xml
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope">
  <Header>
    <n:AlertControl xmlns="http://ibm.com/alertcontrol">
      <n:Priority>1</n:Priority>
    </n:AlertControl>
  </Header>
  <Body>
    <m:Alert xmlns="http://ibm.com/alert">
      <m:Msg>Pick up Mary at school at 2pm</m:Msg>
    </m:Alert>
  </Body>
</Envelope>
```

### Default Namespaces and Scope

For a namespace definition, a prefix is optional. All elements that are defined without a prefix and appear within the element containing the namespace declaration belong to that default namespace.

A namespace declaration applies to the element that contains the definition as well as its child elements, unless it is overridden by another namespace declaration within the element definition. If we look at the example below, we see that

```xml
       xmlns:BookContent="http://www.ibm.com/BookContent"
       xmlns="http://www.ibm.com/BookDefault">
  <Book>
    <BookInfo:Name>Understanding Namespaces</BookInfo:Name>
    <Author>Whizlabs</Author>
    <BookContent:Price>53.50</BookContent:Price>
    <Publisher xmlns="http://www.ibm.com/Publishers">Whizlabs</Publisher>
  </Book>
</Books>
```

the following are the element names and the namespaces they belong to:

<table>
<thead>
<tr>
<th>Element</th>
<th>Namespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Book&gt;</td>
<td><a href="http://www.ibm.com/BookDefault">http://www.ibm.com/BookDefault</a></td>
</tr>
<tr>
<td>&lt;Name&gt;</td>
<td><a href="http://www.ibm.com/BookInformation">http://www.ibm.com/BookInformation</a></td>
</tr>
<tr>
<td>&lt;Author&gt;</td>
<td><a href="http://www.ibm.com/BookDefault">http://www.ibm.com/BookDefault</a></td>
</tr>
<tr>
<td>&lt;Price&gt;</td>
<td><a href="http://www.ibm.com/BookContent">http://www.ibm.com/BookContent</a></td>
</tr>
<tr>
<td>&lt;Publisher&gt;</td>
<td><a href="http://www.ibm.com/Publishers">http://www.ibm.com/Publishers</a></td>
</tr>
</tbody>
</table>

### Attributes

As with elements, you can also qualify attributes by assigning them a prefix that’s mapped to a namespace declaration. But attributes behave differently from elements when it comes to the application of namespaces. If an attribute is not qualified with a prefix, it does not belong to any namespace, so default namespace declarations do not apply to attributes.
Definition of XML documents

An XML schema is a document that defines constraints for the structure and content of an XML document. This is in addition to the rules imposed by XML itself and should be looked at as a higher level of organizational restriction.

One of the first XML schema definition languages has been the Document Type Definition (DTD) language. Because of its complexity it has been largely replaced by the XML Schema Definition (XSD) specification. XSD allows us to define what elements and attributes may appear in a document, which ones are optional or required and their relationship to each other. It also defines the type of data that can occur in elements and helps define complex data types.

Having an XSD document also allows us to verify an XML document for validity. In addition, you will notice that WSDL documents usually reference the XSD namespace in their <types> section and utilize the XSD specification therein to define the input and output messages of the Web Service.

Schema definition

A schema is defined in a separate file and generally stored with the .xsd extension. Every schema definition has a schema root element that belongs to the http://www.w3.org/2001/XMLSchema namespace. The schema element can also contain optional attributes. For example:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
```

This indicates that the elements used in the schema come from the http://www.w3.org/2001/XMLSchema namespace.

Schema linking

An XML file links to its corresponding schema using the schemaLocation attribute of the schema namespace. You have to define the schema namespace in order to use the schemaLocation attribute. All of these definitions appear in the root element of the XML document. The syntax is:

```
<ROOT_ELEMENT
    SCHEMA_NAMESPACE_DEFINITION
    SCHEMA_LOCATION_DEFINITION >
```

And here’s an example of it in use:

```
<Books
    xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
    xs:schemaLocation="http://www.booksforsale.com Books.xsd">
```

Schema elements

A schema file contains definitions for element and attributes, as well as data types for elements and attributes. It is also used to define the structure or the content model of an XML document. Elements in a schema file can be classified as either simple or complex -- defined in “Schema elements - simple types” and “Schema Elements - Complex Types” on page 16.

Schema elements - simple types

A simple type element is an element that cannot contain any attributes or child elements; it can only contain the data type specified in its declaration. The syntax for defining a simple element is:

```
<xs:element name="ELEMENT_NAME" type="DATA_TYPE" default/fixed="VALUE" />
```
Where DATA_TYPE is one of the built-in schema data types (see below).

You can also specify default or fixed values for an element. You do this with either the `default` or `fixed` attribute and specify a value for the attribute. The `default` and `fixed` attributes are optional.

An example of a simple type element is:

```xml
<xs:element name="Author" type="xs:string" default="Whizlabs"/>
```

All attributes are simple types, so they are defined in the same way that simple elements are defined. For example:

```xml
<xs:attribute name="title" type="xs:string"/>
```

**Schema data types.** All data types in schema inherit from `anyType`. This includes both simple and complex data types. You can further classify simple types into built-in-primitive types and built-in-derived types. A complete hierarchical diagram from the XML Schema Datatypes Recommendation1 is shown below:

---

Schema Elements - Complex Types

Complex types are elements that either:
- Contain other elements
- Contain attributes
- Are empty (empty elements)
- Contain text

Figure 7. XML schema datatypes
To define a complex type in a schema, use a complexType element. You can specify the order of occurrence and the number of times an element can occur (cardinality) by using the order and occurrence indicators, respectively. (See “Occurrence and Order Indicators” for more on these indicators.) For example:

```xml
<xs:element name="Book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="xs:string" />
      <xs:element name="Author" type="xs:string" maxOccurs="4"/>
      <xs:element name="ID" type="xs:string"/>
      <xs:element name="Price" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

In this example, the order indicator is xs:sequence, and the occurrence indicator is maxOccurs in the Author element name.

**Occurrence and Order Indicators**

*Occurrence indicators* specify the number of times an element can occur in an XML document. You specify them with the minOccurs and maxOccurs attributes of the element in the element definition.

As the names suggest, minOccurs specifies the minimum number of times an element can occur in an XML document while maxOccurs specifies the maximum number of times the element can occur. It is possible to specify that an element might occur any number of times in an XML document. This is determined by setting the maxOccurs value to unbounded. The default values for both minOccurs and maxOccurs is 1, which means that by default an element or attribute can appear exactly one time.

*Order indicators* define the order or sequence in which elements can occur in an XML document. Three types of order Indicators are:

- **A11**: If A11 is the order indicator, then the defined elements can appear in any order and must occur only once. Remember that both the maxOccurs and minOccurs values for A11 are always 1.
- **Sequence**: If Sequence is the order indicator, then the elements must appear in the order specified.
- **Choice**: If Choice is the order indicator, then any one of the elements specified must appear in the XML document.

Take a look at the following example:
In the above example, the `xs:all` indicator specifies that the `Book` element, if present, must contain only one instance of each of the following four elements: `Name`, `ID`, `Authors`, `Price`. The `xs:sequence` indicator in the `authorType` declaration specifies that elements of this particular type (`Authors` element) contain at least one `Author` element and can contain up to four `Author` elements. The `xs:choice` indicator in the `priceType` declaration specifies that elements of this particular type (`Price` element) can contain either a `dollars` element or a `pounds` element, but not both.

**Restriction**

A main advantage of schema is that you have the ability to control the value of XML attributes and elements. A restriction, which applies to all of the simple data elements in a schema, allows you to define your own data type according to the requirements by modifying the facets (restrictions on XML elements) for a particular simple type. To achieve this, use the `restriction` element defined in the schema namespace.

W3C XML Schema defines 12 facets for simple data types. The following list includes each facet, along with its effect on the data type value and an example.

- **enumeration**: Value of the data type is constrained to a specific set of values. For example:

  ```xml
  <xs:simpleType name="Subjects">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Biology"/>
      <xs:enumeration value="History"/>
      <xs:enumeration value="Geology"/>
    </xs:restriction>
  </xs:simpleType>
  ```

- **maxExclusive**: Numeric value of the data type is less than the value specified.

  ```xml
  <xs:simpleType name="id">
    <xs:restriction base="xs:integer">
      <xs:maxExclusive value="101"/>
      <xs:minExclusive value="1"/>
    </xs:restriction>
  </xs:simpleType>
  ```
• **maxInclusive** - Numeric value of the data type is less than or equal to the value specified.
  
  **minInclusive** - Numeric value of the data type is greater than or equal to the value specified. For example:

  ```xml
  <xs:simpleType name="id">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>
  ```

  • **maxLength** - Specifies the maximum number of characters or list items allowed in the value.
  
  **minLength** - Specifies the minimum number of characters or list items allowed in the value.
  
  **pattern** - Value of the data type is constrained to a specific sequence of characters that are expressed using regular expressions. For example:

  ```xml
  <xs:simpleType name="nameFormat">
    <xs:restriction base="xs:string">
      <xs:minLength value="3"/>
      <xs:maxLength value="10"/>
      <xs:pattern value="[a-z][A-Z]*"/>
    </xs:restriction>
  </xs:simpleType>
  ```

  • **length** - Specifies the exact number of characters or list items allowed in the value. For example:

  ```xml
  <xs:simpleType name="secretCode">
    <xs:restriction base="xs:string">
      <xs:length value="5"/>
    </xs:restriction>
  </xs:simpleType>
  ```

  • **whiteSpace** - Specifies the method for handling white space. Allowed values for the value attribute are preserve, replace, and collapse. For example:

  ```xml
  <xs:simpleType name="FirstName">
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="preserve"/>
    </xs:restriction>
  </xs:simpleType>
  ```

  • **fractionDigits** - Constrains the maximum number of decimal places allowed in the value.
  
  **totalDigits** - The number of digits allowed in the value. For example:

  ```xml
  <xs:simpleType name="reducedPrice">
    <xs:restriction base="xs:float">
      <xs:totalDigits value="4"/>
      <xs:fractionDigits value="2"/>
    </xs:restriction>
  </xs:simpleType>
  ```

**Extension**

The `extension` element defines complex types that might derive from other complex or simple types. If the base type is a simple type, then the complex type can only add attributes. If the base type is a complex type, then it is possible to add attributes and elements. To derive from a complex type, you have to use the `complexContent` element in conjunction with the `base` attribute of the `extension` element.

Extensions are particularly useful when you need to reuse complex element definitions in other complex element definitions. For example, it is possible to define a `Name` element that contains two child elements...
(First and Last) and then reuse it in other complex element definitions. Here is an example:

<!--Base element definition -->
<xs:complexType name="Name">
   <xs:sequence>
      <xs:element name="First"/>
      <xs:element name="Last"/>
   </xs:sequence>
</xs:complexType>

<!-- Customer element that reuses it -->
<xs:complexType name="Customer">
   <xs:complexContent>
      <xs:extension base="Name">
         <xs:sequence>
            <xs:element name="phone" type="xs:string"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>

<!-- Student element that reuses it -->
<xs:complexType name="Student">
   <xs:complexContent>
      <xs:extension base="Name">
         <xs:sequence>
            <xs:element name="school" type="xs:string"/>
            <xs:element name="year" type="xs:string"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>

Import and Include

The import and include elements help to construct a schema from multiple documents and namespaces. The import element brings in a schema from a different namespace, while the include element brings in a schema from the same namespace.

When you use include, the target namespace of the included schema must be the same as the target namespace of the including schema. In the case of import, the target namespace of the included schema must be different from the target namespace of the including schema.

The syntax for import is:

<xs:import id="ID_DATATYPE" namespace="anyURI_DATATYPE"
   schemaLocation="anyURI_DATATYPE ">
</xs:import>

The syntax for include is:

<xs:include id="ID_DATATYPE" schemaLocation="anyURI_DATATYPE"/>
Chapter 3. SOAP primer

The most fundamental underpinnings of the Web services architecture is XML messaging. The current industry standard for XML messaging is SOAP, a lightweight XML-based messaging protocol defined by the World Wide Web Consortium (W3C) that is used to encode the information in Web service request and response messages before sending them over the Internet.

SOAP is defined independently of any operating system or protocol and provides a way to communicate between applications running on different computers, using different operating systems, and with different technologies and programming languages as long as the SOAP request and response messages match the message formats that are defined in the WSDL document.

SOAP consists of three parts: An envelope that defines a framework for describing message content and process instructions, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses.

SOAP is, in principle, transport protocol-independent and can, therefore, potentially be used in combination with a variety of protocols such as HTTP, JMS, SMTP, or FTP. Right now, the most common way of exchanging SOAP messages is through HTTP, which is also the only protocol supported by WS-I Basic Profile 1.0.

The way SOAP applications communicate when exchanging messages is often referred to as the message exchange pattern (MEP). Message exchange patterns are discussed in “Message exchange patterns” on page 9.

There are two versions of SOAP: SOAP 1.1 and SOAP 1.2. Both SOAP 1.1 and SOAP 1.2 are W3C standards. Web services can be deployed that support not only SOAP 1.1 but also support SOAP 1.2. SOAP 1.2 provides a more specific definition of the SOAP processing model, which removes many of the ambiguities that sometimes led to interoperability problems in the absence of the Web Services-Interoperability (WS-I) profiles. Some of the more significant changes in the SOAP 1.2 specification include:

- The ability to now officially define other transport protocols other than the HTTP protocol as long as vendors conform to the binding framework that is defined in SOAP 1.2.
- The fact that SOAP 1.2 is based on the XML Information Set (XML Infoset). The information set provides a way to describe the XML document using the XSD schema but does not necessarily serialize the document by using XML 1.0 serialization. SOAP 1.1 is based upon XML 1.0 serialization. The information set will make it easier to use other serialization formats such as a binary protocol format. You can use a binary protocol format shrink the message into a much more compact format where some of the verbose tagging information might not be required.

Since the Web services client for ILE only supports the SOAP 1.1 specification, the following sections will cover the SOAP 1.1 specification and the SOAP architecture in detail. For more information on SOAP, go to the following URL:

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

SOAP message structure

A SOAP message, which is an XML document based on the SOAP protocol, consists of four parts:

1. The SOAP <Envelope> element, the root element of a SOAP message, contains an optional SOAP header and mandatory SOAP body elements. The SOAP protocol namespace prefix (http://schemas.xmlsoap.org/soap/envelope/) is usually declared in the envelope open tag.
2. The optional and extensible <header> element describes metadata, such as security, transaction, and conversational-state information.

3. The mandatory <body> element contains the XML document of the sender. The sender’s XML document must not contain an XML declaration or DTD declaration. There are two main paradigms which the sender’s document can adhere to: document-style or RPC-style (more about these later). The serialization rules for the contents of the body can be specified by setting the encodingStyle attribute. The standard SOAP encoding namespace is http://schemas.xmlsoap.org/soap/encoding/.

4. Elements called <faults> can be used by a processing node (SOAP intermediary or ultimate SOAP destination) to describe any exceptional situations it could encounter that might occur while reading the SOAP message.

A W3C note also specifies a way to embed and describe attachments to a SOAP message. An attachment can be any type of file, whether binary- or character-based. Instead of creating a new encoding scheme, the Attachments specification employs the MIME specification rules, as shown in Figure 8 below.

![Figure 8. SOAP message structure with attachments](image)

The following sections discusses the major elements of a SOAP message.

**Namespaces**

The use of namespaces plays an important role in SOAP message, because a message can include several different XML elements that must be identified by a unique namespace to avoid name collision. Especially, the WS-I Basic Profile 1.0 requires that all application-specific elements in the body must be namespace qualified to avoid name collision. Table 2 on page 23 shows the namespaces of SOAP and WS-I Basic Profile 1.0.
Table 2. SOAP namespaces

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Namespace URI</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAP-ENV</td>
<td><a href="http://schemas.xmlsoap.org/soap/envelope/">http://schemas.xmlsoap.org/soap/envelope/</a></td>
<td>SOAP 1.1 envelope namespace</td>
</tr>
<tr>
<td>SOAP-ENC</td>
<td><a href="http://schemas.xmlsoap.org/soap/encoding/">http://schemas.xmlsoap.org/soap/encoding/</a></td>
<td>SOAP 1.1 encoding namespace</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a></td>
<td>Schema instance namespace</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
<td>XML Schema namespace</td>
</tr>
<tr>
<td></td>
<td><a href="http://schemas.xmlsoap.org/wsdl">http://schemas.xmlsoap.org/wsdl</a></td>
<td>WSDL namespace for WSDL framework</td>
</tr>
<tr>
<td></td>
<td><a href="http://schemas.xmlsoap.org/wsdl/soap">http://schemas.xmlsoap.org/wsdl/soap</a></td>
<td>WSDL namespace for WSDL SOAP binding</td>
</tr>
</tbody>
</table>

**URN**

A *unified resource name* (URN) uniquely identifies the service to clients. It must be unique among all services deployed in a single SOAP server, which is identified by a certain network address. A URN is encoded as a *universal resource identifier* (URI).

All other addressing information is transport dependent. For example, when using HTTP as the transport, the URL of the HTTP request points to the SOAP server instance on the destination host.

**The SOAP envelope**

The basic unit of a Web service message is the actual SOAP envelope (see Figure 9). This is an XML document that includes all of the information necessary to process the message.

![SOAP Envelope](image)

*Figure 9. SOAP envelope*

A SOAP message is a (possibly empty) set of headers plus one body. The Envelope element is the root element of any SOAP message. Generally, it contains the definition for the required envelope namespace. For example:

```xml
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
<env:Header>
</env:Header>
<env:Body>
</env:Body>
</env:Envelope>
```

In the example above, you have a simple Envelope, with the namespace specified as SOAP version 1.1. It includes two sub elements, a Header and a Body.
Let's look at what each of those elements do.

**The SOAP header**

The Header in a SOAP message is meant to provide information about the message itself, as opposed to information meant for the application. For example, the Header might include routing information, as it does in this example shown below:

```xml
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
  <env:Header>
      http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous
    </wse:ReplyTo>
    <wse:From>
      http://localhost:8080/axis/services/MyService
    </wse:From>
    <wse:MessageID>ECE5B3F187F29D2B11433905662036</wse:MessageID>
  </env:Header>
  <env:Body>
  </env:Body>
</env:Envelope>
```

In this case you see a WS-Addressing element, which includes information on where the message is going and to where replies should go.

Headers are optional elements in the envelope. If present, the element must be the first immediate child element of a SOAP envelope element. All immediate child elements of the header element are called header entries.

As has been previously stated, headers can include all kinds of information about the message itself. In fact, the SOAP specification spends a great deal of time on elements that can go in the Header, and how they should be treated by SOAP intermediaries (applications that are capable of both receiving and forwarding SOAP messages on their way to the final destination). In other words, the SOAP specification makes no assumption that the message is going straight from one point to another, from client to server. It allows for the idea that a SOAP message might actually be processed by several intermediaries, on its way to its final destination, and the specification is very clear on how those intermediaries should treat information they find in the Header. That discussion is beyond the scope of this document. However, there are two predefined header attributes that you should be aware of: SOAP-ENV:mustUnderstand and SOAP-ENV:actor.

The header attribute SOAP-ENV:mustUnderstand is used to indicate to the service provider that the semantics defined by the element must be implemented. The value of the mustUnderstand attribute is either 1 or 0 (the absence of the attribute is semantically equivalent to the value 0):

```xml
<thens:qos xmlns:thens="someURI" SOAP-ENV:mustUnderstand="1">3</thens:qos>
```

In the example above, the header element specifies that a service invocation must fail if the service provider does not support the quality of service (qos) 3 (whatever qos=3 stands for in the actual invocation and servicing context).

The header attribute SOAP-ENV:actor is used to identify the recipient of the header information. The value of the SOAP actor attribute is the URI of the mediator, which is also the final destination of the particular header element (the mediator does not forward the header). If the actor is omitted or set to the predefined default value, the header is for the actual recipient and the actual recipient is also the final
destination of the message (body). The predefine value is: `http://schemas.xmlsoap.org/soap/actor/next`. If a node on the message path does not recognize a mustUnderstand header and the node plays the role specified by the actor attribute, the node must generate a SOAP mustUnderstand fault (more on faults later). Whether the fault is sent back to the sender depends on the message exchange pattern (e.g. request/response) in use.

Now let’s look at the actual payload.

**The SOAP body**

When you’re sending a SOAP message, you’re doing it with a reason in mind. You are trying to tell the receiver to do something, or you’re trying to impart information to the server. This information is called the ‘payload’. The payload goes in the Body of the Envelope. It also has its own namespace, in this case corresponding to the content management system. The choice of namespace, in this case, is completely arbitrary. It just needs to be different from the SOAP namespace. For example:

```xml
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header>
    ...
  </env:Header>
  <env:Body>
      <cms:category>classifieds</category>
      <cms:subcategory>forsale</cms:subcategory>
      <cms:articleHeadline>Vintage 1963 T-Bird.</cms:articleHeadline>
    </cms:addArticle>
  </env:Body>
</env:Envelope>
```

In this case, you have a simple payload that includes instructions for adding an article to the content management system.

The body element is encoded as an immediate child element of the SOAP envelope element. If a header element is present, then the body element must immediately follow the header element. Otherwise it must be the first immediate child element of the envelope element. All immediate child elements of the body element are called body entries, and each body entry is encoded as an independent element within the SOAP body element. In the most simple case, the body of a basic SOAP message consists of:

- A message name.
- A reference to a service instance.
- One or more parameters carrying values and optional type references.

Typical uses of the body element include invoking RPC calls with appropriate parameters, returning results, and error reporting. Fault elements are used in communicating error situations.

The choice of how to structure the payload involves the style and encoding.

**Error handling (SOAP faults)**

SOAP itself predefines one body element, which is the fault element used for reporting errors. If present, the fault element must appear as a body entry and must not appear more than once within a body element. The fields of the fault element are defined as follows:

- `faultcode` is a code that indicates the type of the fault. SOAP defines a small set of SOAP fault codes covering basic SOAP faults:
  - `soapenv:Client`, indicating incorrectly formatted messages
  - `soapenv:Server`, for delivery problems
– soapenv:VersionMismatch, which can report any invalid namespaces for envelope
element
– soapenv:MustUnderstand, for errors regarding the processing of header content

• faultstring is a human-readable description of the fault. It must be present in a fault element.
• faultactor is an optional field that indicates the URI of the source of the fault. It is similar to the
  SOAP actor attribute, but instead of indicating the destination of the header entry, it indicates the
  source of the fault. The value of the faultactor attribute is a URI identifying the source that caused
  the error. Applications that do not act as the ultimate destination of the SOAP message must include
  the faultactor element in a SOAP fault element.

• detail is an application-specific field that contains detailed information about the fault. It must not be
  used to carry information about errors belonging to header entries. Therefore, the absence of the detail
  element in the fault element indicates that the fault is not related to the processing of the body element
  (the actual message).

Here is an example of a fault response message:

```xml
<SOAP-ENV:Header>
<m:Order xmlns:m="some URI" SOAP-ENV:mustUnderstand="1">
</m:Order>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:Server</faultcode>
<faultstring>Not necessary information</faultstring>
<detail>
  <d:faultdetail xmlns:d = "uri-reference">
    <msg>application is not responding properly. </msg>
    <errorcode>12</errorcode>
  </d:faultdetail>
</detail>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

**Data model**

One of the promises of SOAP is interoperability between different programming languages. That is the
purpose of the SOAP data model, which provides a language-independent abstraction for common
programming language types. It consists of:

• **Simple XSD types**: Basic data types found in most programming languages such as int, float, and
  null-terminated character data (i.e. strings).

• **Compound types**: There are two kinds of compound types, *structs* and *arrays*:

  – Structs are named aggregated types. Each element has a unique name, its accessor. An accessor is an
    XML tag. Structs are conceptually similar to records in languages, such as RPG, or method-less
    classes with public data members in object-based programming languages.

  – Elements in an array are identified by position, not by name. Array values can be structs or other
    compound values. Also, nested arrays (which means arrays of arrays) are allowed.

Let us take a look at an example. Below is a XML schema (if you want a refresher on schema’s, see
"Definition of XML documents” on page 14) a compound datatype named Mobile.
In the listing above, line 1 shows the name (Mobile) of our type while line 2 acknowledges that it is a complex datatype that contains sub-elements named modelNumber, modelName and modelColor. The sub-element defined in line 3, modelNumber, has a type of int (that is, modelNumber can take only integer values). The sub-element defined in line 4 is named modelName and is of type string. The sub-element defined in line 5 requires a bit more understanding since it has a sub element named simpleType. Here you are defining a simple type inside the complex type, Mobile. The name of your simpleType is modelColor and it is an enumeration. It has an attribute, base, carrying the value xsd:string, which indicates that the simple type modelColor has the functionality of the string type defined in the SOAP schema. Each <enumeration> tag carries an attribute, value (blue and black). The enumerated types enable us to select one value from multiple options. Now let us look at how this translates into a SOAP message.

The listing below is demonstrates the use of compound types in a SOAP message. It shows an envelope carrying a request in the Body element, in which you are calling the addModel method of an m namespace. The listing uses the data type Mobile that was defined above. The AddModel method takes an argument of type Mobile. We're referring Mobile structure with msd namespace reference (see the xmlns:msd declaration in <SOAP-ENV:Envelope> element). This is an example of employing user defined data types in SOAP requests.

`<? xml version="1.0" ?>`  
`<xsd:schema xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"`  
`xmlns:xsd="http://www.w3.org/1999/XMLSchema"`  
`targetNameSpace= "www.mobilephoneservice.com/phonequote">`  
`<xsd:element name ="Mobile">`  
```xml
  <xsd:complexType>
    <xsd:element name="modelNumber" type="xsd:int"> </xsd:element>
    <xsd:element name="modelName" type="xsd:string"> </xsd:element>
    <xsd:element name="modelColor">
      <simpleType base="xsd:string">
        <enumeration value="blue" />
        <enumeration value="black" />
      </simpleType>
    </xsd:element>
  </complexType>
</xsd:element>
</xsd:schema>`

In the listing above, line 1 shows the name (Mobile) of our type while line 2 acknowledges that it is a complex datatype that contains sub-elements named modelNumber, modelName and modelColor. The sub-element defined in line 3, modelNumber, has a type of int (that is, modelNumber can take only integer values). The sub-element defined in line 4 is named modelName and is of type string. The sub-element defined in line 5 requires a bit more understanding since it has a sub element named simpleType. Here you are defining a simple type inside the complex type, Mobile. The name of your simpleType is modelColor and it is an enumeration. It has an attribute, base, carrying the value xsd:string, which indicates that the simple type modelColor has the functionality of the string type defined in the SOAP schema. Each <enumeration> tag carries an attribute, value (blue and black). The enumerated types enable us to select one value from multiple options. Now let us look at how this translates into a SOAP message.

The listing below is demonstrates the use of compound types in a SOAP message. It shows an envelope carrying a request in the Body element, in which you are calling the addModel method of an m namespace. The listing uses the data type Mobile that was defined above. The AddModel method takes an argument of type Mobile. We're referring Mobile structure with msd namespace reference (see the xmlns:msd declaration in <SOAP-ENV:Envelope> element). This is an example of employing user defined data types in SOAP requests.

```xml
 xmlns:xsd="http://www.w3.org/1999/XMLSchema"
 xmlns:msd="www.ibm.com/phonequote">
 <SOAP-ENV:Body>
  <msd:Mobile xmlns:msd="www.ibm.com">
   <modelNumber>1</modelNumber>
   <modelName>mlr97</modelName>
   <modelColor>blue</modelColor>
  </msd:Mobile>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>`

**SOAP binding and encoding styles**

You'll get deeper into this subject in Chapter 4, “WSDL primer,” on page 31, which talks about Web Services Description Language (WSDL), but as you create your application, you will need to decide on the structure of the actual payload you're sending back and forth. To that end, let's take this opportunity to discuss SOAP binding (also referred as programming or communication binding) and encoding styles.

---

2. Well, in the case of Web Services Client for ILE it is made for you! But for completeness we discuss what is available. Web Services Client for ILE only supports Document/Literal. To understand what that means, read on.
To simplify the discussion, the following XML message payload is used as an example:

```
<article>
  <category>classifieds</category>
  <subcategory>forsale</subcategory>
  <articleText>Vintage 1963 T-Bird.</articleText>
</article>
```

This piece of XML payload can be presented in a SOAP message in two different styles: Remote Procedure Calls (RPC) and document. RPC style SOAP describes the semantics of a procedure call and its return value. In this style, the idea is that you're sending a command to the server, such as "add an article", and you're including the parameters command, such as the article to add and the category to which it should as child elements of the overall method. This programming style thus adds extra elements to the SOAP XML to simulate a method call (i.e. the XML payload is wrapped inside an operation element in a SOAP body). A document style message, on the other hand, has the XML payload directly placed in a SOAP body. Document style SOAP is described as being one-way or asynchronous, as there is not a concept of a call and return as in the RPC model. Basically, a document-style message lets you describe an arbitrary XML document using SOAP.

Both the RPC and document message can be either a literal or encoded message. A literal message implies that a schema is utilized to provide a description and constraint for an XML payload in SOAP. An Encoded message implies that the message includes type information. Let us look at some examples.

The example below is a typical RPC/literal example.

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header></env:Header>
  <env:Body>
    <addArticle>
      <article>
        <category>classifieds</category>
        <subcategory>forsale</subcategory>
        <articleText>Vintage 1963 T-Bird.</articleText>
      </article>
    </addArticle>
  </env:Body>
</env:Envelope>
```

The addArticle element is the operation to be invoked. The element article (which contains sub-elements category, subcategory, and articleText) is the input parameters to the operation.

If we include type information in the message as in the example below, we have an example of an RPC/encoded message.

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header></env:Header>
  <env:Body>
    <addArticle>
      <article xsi:type="xsd:string">classifieds</category>
      <subcategory xsi:type="xsd:string">forsale</subcategory>
      <articleText xsi:type="xsd:string">Vintage 1963 T-Bird.</articleText>
    </addArticle>
  </env:Body>
</env:Envelope>
```

A document/literal style of message simply involves adding the message:
In this case, the message itself doesn't include information on the process to which the data is to be submitted; that is handled by the routing software. For example, all calls to a particular URL or endpoint might point to a particular operation.

Finally, you could technically use the document/encoded style, but nobody does, so for now, ignore it.

Different trade-offs are involved with each of these styles. However, the Encoded style has been a source of interoperability problems and is not WS-I compliant, so should be avoided. Although RPC/literal has its usefulness, the most popular form of binding and encoding styles has become document/literal. The document/literal style goes a long way in eliminating interoperability problems, and also has proven to be a good performer while generating the least complex SOAP message.

**SOAP response messages**

In the previous section the discussion has been about request messages. But what about response messages? What do they look like? By now it should be clear to you what the response message looks like for a document/literal message. The contents of the *soap:body* are fully defined by a schema, so all you have to do is look at the schema to know what the response message looks like.

But what is the child of the *soap:body* for the RPC style responses? The WSDL 1.1 specification is not clear. But WS-I comes to the rescue. WS-I's Basic Profile dictates that in the RPC/literal response message, the name of the child of *soap:body* is “... the corresponding *wsdl:operation* name suffixed with the string 'Response'.” For more information on *wsdl:operation*, see Chapter 4, “WSDL primer,” on page 31.
Chapter 4. WSDL primer

WSDL (Web Services Description Language) is an XML document for describing Web Services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented (RPC) messages. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints or services. WSDL is extensible to allow description of endpoints and their messages, regardless of what message formats or network protocols are used to communicate. Some of the currently described bindings are for SOAP 1.1, HTTP POST, and Multipurpose Internet Mail Extensions (MIME).

There are two versions of the WSDL: WSDL 1.1 and WSDL 2.0. The changes in WSDL 2.0 are generally made for the purposes of interoperability - constructs that are not legal under WS-I's Basic Profile are generally forbidden - or to make it easier to use WSDL with extended SOAP specifications.

The Web services client for ILE supports WSDL 1.1 and the rest of the discussion in this chapter will be from the perspective of the WSDL 1.1 specification.

WSDL conventionally divides the basic service description into two parts (see Figure 10): the service interface and the service implementation. This enables each part to be defined separately and independently, and reused by other parts.

A service interface definition is an abstract or reusable service definition that can be instantiated and referenced by multiple service implementation definitions. Think of a service interface definition as an Interface Definition Language (IDL), Java interface or Web service type. This allows common industry-standard service types to be defined and implemented by multiple service implementers. This is analogous to defining an abstract interface in a programming language and having multiple concrete implementations. The service interface contains WSDL elements that comprise the reusable portion of the service description:

- **binding**: Describes the protocol, data format, security and other attributes for a particular service interface (i.e. portType).
- **portType**: Defines Web service operations. The operations define what XML messages can appear in the input and output data flows. Think of an operation as a method signature in a programming language.
- **message**: Specifies which XML data types constitute various parts of a message and is used to define the input and output parameters of an operation.
- **type**: Describes the use complex data types within the message.

The *service implementation definition* describes how a particular service interface is implemented by a given service provider. A Web service is modeled as a *service* element. A *service* element contains a collection (usually one) of *port* elements. A port associates an endpoint (for example, a network address location or URL) with a *binding* element from a service interface definition.

The service interface definition together with the service implementation definition makes up a complete WSDL definition of the service. This pair contains sufficient information to describe to the service requestor how to invoke and interact with the Web service.

In the following sections, we cover WSDL 1.1. Information on WSDL 1.1 can be found at:

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

Now let's dive into the details.

**WSDL document structure**

*Figure 11 on page 33* shows the elements comprising a WSDL document and the various relationships between them.
The diagram should be read in the following way:

- One WSDL document contains zero or more services. A service contains zero or more port definitions (service endpoints), and a port definition contains a specific protocol extension.
- The same WSDL document contains zero or more bindings. A binding is referenced by zero or more ports. The binding contains one protocol extension, where the style and transport are defined, and zero or more operations bindings. Each of these operation bindings is composed of one protocol extension, where the action and style are defined, and one to three messages bindings, where the encoding is defined.
- The same WSDL document contains zero or more port types. A port type is referenced by zero or more bindings. This port type contains zero or more operations, which are referenced by zero or more operations bindings.
- The same WSDL document contains zero or more messages. An operation usually points to an input and an output message, and optionally to some faults. A message is composed of zero or more parts.
- The same WSDL document contains zero or more types. A type can be referenced by zero or more parts.
- The same WSDL document points to zero or more XML Schemas. An XML Schema contains zero or more XSD types that define the different data types.

The containment relationships shown in the diagram directly map to the XML Schema for WSDL.

Below is an example of a simple, complete, and valid WSDL file. As we will see, even a simple WSDL document contains quite a few elements with various relationships to each other.
<wsdl:definitions targetNamespace="http://address.samples"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://address.samples"
xmlns:intf="http://address.samples"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<wsdl:types>
  <schema elementFormDefault="qualified"
targetNamespace="http://address.samples"
xmlns="http://www.w3.org/2001/XMLSchema">
    <complexType name="AddressBean">
      <sequence>
        <element name="street" type="xsd:string"/>
        <element name="zipcode" type="xsd:int"/>
      </sequence>
    </complexType>
    <element name="AddressBean" type="impl:AddressBean"/>
  </schema>
</wsdl:types>

<wsdl:message name="updateAddressRequest">
  <wsdl:part name="in0" type="intf:AddressBean"/>
  <wsdl:part name="in1" type="xsd:int"/>
</wsdl:message>

<wsdl:message name="updateAddressResponse">
  <wsdl:part name="return" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="updateAddressFaultInfo">
  <wsdl:part name="fault" type="xsd:string"/>
</wsdl:message>

<wsdl:portType name="AddressService">
  <wsdl:operation name="updateAddress">
    <wsdl:input message="intf:updateAddressRequest"
      name="updateAddressRequest"/>
    <wsdl:output message="intf:updateAddressResponse"
      name="updateAddressResponse"/>
    <wsdl:fault message="intf:updateAddressFaultInfo"
      name="updateAddressFaultInfo"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="AddressSoapBinding" type="intf:AddressService">
  <wsdlsoap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>

  <wsdl:operation name="updateAddress">
    <wsdlsoap:operation soapAction=""/>
    <wsdlsoap:input name="updateAddressRequest"/>
    <wsdl:input/>
  </wsdl:operation>

  <wsdl:output name="updateAddressResponse"/>
  <wsdlsoap:body use="literal"/>
  <wsdl:output/>

  <wsdl:fault name="updateAddressFaultInfo"/>
  <wsdlsoap:fault name="updateAddressFaultInfo" use="literal"/>
</wsdl:binding>
So let us begin discussing the various components that make up a WSDL document.

**Namespaces**

WSDL documents begin with a declarative section that lays out two key components. The first declarative component consists of the various namespace declarations, declared as attributes of the root element (the second is the types element discussed in "Types"):

```xml
<wsdl:definitions targetNamespace="http://address.samples"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://address.samples"
xmlns:intf="http://address.samples"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
</wsdl:definitions>
```

WSDL uses the XML namespaces listed in Table 3:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Namespace URI</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>soap</td>
<td><a href="http://schemas.xmlsoap.org/wsdl/soap/">http://schemas.xmlsoap.org/wsdl/soap/</a></td>
<td>SOAP binding.</td>
</tr>
<tr>
<td>soapenc</td>
<td><a href="http://schemas.xmlsoap.org/soap/encoding/">http://schemas.xmlsoap.org/soap/encoding/</a></td>
<td>Encoding namespace as defined by SOAP 1.1.</td>
</tr>
<tr>
<td>soapenv</td>
<td><a href="http://schemas.xmlsoap.org/soap/envelope/">http://schemas.xmlsoap.org/soap/envelope/</a></td>
<td>Envelope namespace as defined by SOAP 1.1.</td>
</tr>
<tr>
<td>xsi</td>
<td><a href="http://www.w3.org/2000/10/XMLSchema-instance">http://www.w3.org/2000/10/XMLSchema-instance</a></td>
<td>Instance namespace as defined by XSD.</td>
</tr>
<tr>
<td>xsd</td>
<td><a href="http://www.w3.org/2000/10/XMLSchema">http://www.w3.org/2000/10/XMLSchema</a></td>
<td>Schema namespace as defined by XSD.</td>
</tr>
<tr>
<td>tns</td>
<td>(URL to WSDL file)</td>
<td>The this namespace (tns) prefix is used as a convention to refer to the current document. Do not confuse it with the XSD target namespace, which is a different concept.</td>
</tr>
</tbody>
</table>

The first four namespaces are defined by the WSDL specification itself; the next four definitions reference namespaces that are defined in the SOAP and XSD standards. The last one is local to each specification. Note that in our example, we do not use real namespaces; the URIs contain localhost.

**Types**

The types element encloses data type definitions used by the exchanged messages. WSDL uses XML Schema Definitions (XSDs) as its canonical and built-in type system:
The XSD type system can be used to define the types in a message regardless of whether or not the resulting wire format is XML. There is an extensibility element (placeholder for additional XML elements, that is) that can be used to provide an XML container element to define additional type information in case the XSD type system does not provide sufficient modeling capabilities. In our example, the type definition, shown below, is where we specify that there is a complex type called AddressBean, which is composed of two elements, street and zipcode. We also specify that the type of the street element is a string and the type of the zipcode element is a number (int).

```
<wsdl:types>
  <schema targetNamespace="http://address.samples"
       xmlns="http://www.w3.org/2001/XMLSchema">
    <complexType name="AddressBean">
      <sequence>
        <element name="street" type="xsd:string"/>
        <element name="zipcode" type="xsd:int"/>
      </sequence>
    </complexType>
  </schema>
</wsdl:types>
```

### Messages

Messages consist of one or more logical parts. A message represents one interaction between a service requestor and service provider. If an operation is bidirectional (a call returning a result, for example), at least two message definitions are used in order to specify the transmission on the way to and from the service provider:

```
<definitions ..... >
  <message name="nmtoken">(0 or more)
    <part name="nmtoken" element="qname">(0 or 1) type="qname" (0 or 1)/
    (0 or more)
  </message>
</definitions>
```

The abstract message definitions are used by the operation element. Multiple operations can refer to the same message definition. Operations and messages are modeled separately in order to support flexibility and simplify reuse of existing specifications. For example, two operations with the same parameters can share one abstract message definition. In our example, the messages definition, shown below, is where we specify the different parts that compose each message. The request message updateAddressRequest is composed of an AddressBean part and an int part. The response message updateAddressResponse is composed of a string part. The fault message updateAddressFaultInfo is composed of a string part.
Port types

A port type is a named set of abstract operations and the abstract messages involved:

```
<wsdl:definitions .... >
  <wsdl:portType name="nmtoken">  
    <wsdl:input name="nmtoken"(0 or 1) message="qname"/> (0 or 1)  
    <wsdl:output name="nmtoken"(0 or 1) message="qname"/> (0 or 1)  
    <wsdl:fault name="nmtoken" message="qname"/> (0 or more)  
  </wsdl:portType>
</wsdl:definitions>
```

Presence and order of the input, output, and fault messages determine the type of message. For example, for one-way messages the `wsdl:FAULT` and `wsdl:OUTPUT` operations would be removed. For a request/response messages, one would include both `wsdl:INPUT` and `wsdl:OUTPUT` operations. It should be noted that a request-response operation is an abstract notion. A particular binding must be consulted to determine how the messages are actually sent. For example, the HTTP protocol is a request/response protocol; however, it does not preclude you from sending one-way messages. It simply means that the Web service must send an HTTP response back to the client. The response will be consumed by the transport and nothing is propagated back to the client since the response is purely an HTTP response - that is, no SOAP data is associated with the response.

In our example, the port type and operation definition, shown below, are where we specify the port type, called `AddressService`, and a set of operations. In this case, there is only one operation, called `updateAddress`. We also specify the interface that the Web service provides to its possible clients, with the input message `updateAddressRequest`, the output message `updateAddressResponse`, and the `updateAddressFaultInfo` that are used in the transaction.

```
...<wsdl:portType name="AddressService">
  <wsdl:operation name="updateAddress">
    <wsdl:input message="intf:updateAddressRequest" name="updateAddressRequest"/>
    <wsdl:output message="intf:updateAddressResponse" name="updateAddressResponse"/>
    <wsdl:fault message="intf:updateAddressFaultInfo" name="updateAddressFaultInfo"/>
  </wsdl:operation>
</wsdl:portType>
...
```

**Bindings**

A binding contains:
- Protocol-specific general binding data, such as the underlying transport protocol and the communication style for SOAP.
- Protocol extensions for operations and their messages, such as the URN and encoding information for SOAP.
Each binding references one port type; one port type can be used in multiple bindings. All operations defined within the port type must be bound in the binding. The pseudo XSD for the binding looks like this:

```
<wsdl:definitions .... >
<wsdl:binding name="nmtoken" type="qname"> (0 or more)
  <-- extensibility element (1) --> (0 or more)
  <wsdl:operation name="nmtoken"> (0 or more)
    <-- extensibility element (2) --> (0 or more)
    <wsdl:input name="nmtoken"(0 or 1) > (0 or 1)
      <-- extensibility element (3) -->
      </wsdl:input>
    <wsdl:output name="nmtoken"(0 or 1) > (0 or 1)
      <-- extensibility element (4) --> (0 or more)
      </wsdl:output>
    <wsdl:fault name="nmtoken">
      <-- extensibility element (5) --> (0 or more)
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>
</wsdl:definitions>
```

As we have already seen, a port references a binding. The port and binding are modeled as separate entities in order to support flexibility and location transparency. Two ports that merely differ in their network address can share the same protocol binding.

The extensibility elements `<-- extensibility element (x) -->` use XML namespaces in order to incorporate protocol-specific information into the language- and protocol-independent WSDL specification.

In our example, the binding definition, shown below, is where we specify our binding name, AddressSoapBinding. The connection is SOAP HTTP, and the style is document. We provide a reference to our operation, updateAddress; define the input message updateAddressRequest and the output message updateAddressResponse; and the fault message, updateAddressFaultInfo. Additionally, the input and output messages of the operation are defined as literal XML in compliance with the WS-I Basic Profile.

```
...<wsdl:binding name="AddressSoapBinding" type="intf:AddressService">
  <wsdlsoap:binding style="document">
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="updateAddress">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="updateAddressRequest">
      <wsdlsoap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="updateAddressResponse">
      <wsdlsoap:body use="literal"/>
    </wsdl:output>
    <wsdl:fault name="updateAddressFaultInfo">
      <wsdlsoap:fault name="updateAddressFaultInfo" use="literal"/>
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>
...```

In the above example, both input and output messages are specified. Thus, the operation is governed by the request-response message exchange pattern. If the output message (wsdl:output element) was removed, you would have one-way message exchange pattern.
Service definition

A service definition merely bundles a set of ports together under a name, as the following pseudo XSD definition of the service element shows. This pseudo XSD notation is introduced by the WSDL specification:

```xml
<wsdl:definitions .... >
  <wsdl:service name="nmtoken"> (0 or more)
    <wsdl:port .... /> (0 or more)
  </wsdl:service>
</wsdl:definitions>
```

Multiple service definitions can appear in a single WSDL document.

Port definition

A port definition describes an individual endpoint by specifying a single address for a binding:

```xml
<wsdl:definitions .... >
  <wsdl:service ..... > (0 or more)
    <wsdl:port name="nmtoken" binding="qname"> (0 or more)
      <!-- extensibility element (1) -->
      <wsdl:port>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

The binding attribute is of type QName, which is a qualified name (equivalent to the one used in SOAP). It refers to a binding. A port contains exactly one network address; all other protocol-specific information is contained in the binding.

Any port in the implementation part must reference exactly one binding in the interface part.

The `<-- extensibility element (1) -->` is a placeholder for additional XML elements that can hold protocol-specific information. This mechanism is required, because WSDL is designed to support multiple runtime protocols. For SOAP, the URL of the service is specified as the SOAP address here.

In our example, the service and port definition, shown below, is where we specify our service, called AddressServiceService, that contains a collection of our ports. In this case, there is only one that uses the AddressSoapBinding and is called Address. In this port, we specify our connection point.

```xml
...<wsdl:service name="AddressServiceService">
  <wsdl:port binding="intf:AddressSoapBinding" name="Address">
    <wsdl:soap:address
      location="http://localhost:8080/axis/services/Address"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```
Part 2. Web services client for ILE concepts

This part of the book introduces Web Services Client for ILE concepts and architecture, including installation details.

- **Chapter 5, “Web services client overview,” on page 43** introduces Web services client for ILE architecture and programming module.
- **Chapter 6, “Command line tools,” on page 53** lists the commands that are used to generate Web service client stubs.
- **Chapter 7, “Configuration files,” on page 57** describes the configuration files that affect the Web services client engine.
- **Chapter 8, “The Web services client for ILE installation details,” on page 61** describes what you need to install to get Web services client for ILE functionality and describes where things are located in the package directories.
Chapter 5. Web services client overview

The Web Services Client for ILE is based on Apache Extensible Interaction System (Axis) version 1.53. Axis is basically a SOAP engine that represents a framework for constructing SOAP processors such as clients, servers, or gateways. The only piece that this document will focus on is the client support. However, since Axis is an Apache open-source project, you can find more information at:

http://ws.apache.org/axis/cpp/

The following are the key features of this AXIS framework:

- **Flexible messaging framework**: It provides a flexible messaging framework that includes handlers, chain, serializers, and deserializers. A handler is an object processing request, response, and fault flow. A handler can be grouped together into chains and the order of these handlers can be configured using a flexible deployment descriptor.

- **Data encoding support**: Axis provides automatic serialization of a wide variety of data types as per the XML Schema specifications.

- **Client stub generation**: Axis includes a tool to generate C or C++ Web service client stubs from a WSDL file. This tool has been enhanced by IBM so that it also generates RPG Web service client stubs.

This chapter will give an overview of Web Services Client for ILE, including what specifications and standards are currently supported, the client architecture, and the programming model.

### Supported specifications and standards

The Web Services Client for ILE product has the following capabilities:

- Support for WSDL 1.1 (document literal only)
- SOAP 1.1 is the only supported over-the-wire protocol (as compliant with WS-I 1.1 basic profile)
- Support for SSL
- JAX-RPC style implementation

The following are known limitations and restrictions:

- Dates sent and received must be after midnight 1st January 1970.
- Attachments are not supported.
- WSDL's used against the Integrated Web service client for ILE tooling (wsdl2ws.sh) must be encoded throughout using UTF-8.
- Web service responses must be in UTF-8 format.
- The following schema-related types and constructs are not supported:
  - The use of xsd:list.
  - The use of xsd:union.
  - Complex content extensions is not supported. There is limited support for simple content extensions.
  - The namespace and processContents attributes on xsd:any are not supported. This gives support equivalent to setting namespace="##any" and processContents="skip".

For the most current information regarding product restrictions and limitations, you should read the Limitations Web page at the Integrated Web Services for i Web site (URL to Web site is http://www.ibm.com/systems/i/software/iws/).

© Copyright IBM Corp. 2011, 2012
Client architecture

Figure 12 illustrates the underlying architecture of Apache Axis. One of Axis' key features is the incredible pluggability it offers its users. Almost everything in the Axis engine can be replaced with a customized component, and the single most important component is a handler. More information about handlers will be covered in the following sections but for now let's look, at a high level, the core components of the Axis architecture and what happens on the client.

The core components of the AXIS Architecture include:

• **Axis Engine**: This acts like a central controller for other components.

• **Handlers and Chains**: Handlers are the basic building blocks in the AXIS system. Think of handlers as pluggable components through which the message will pass, allowing each handler a chance to perform some action based on the message or even to modify the message. A chain is a special handler that represents a sequence of other handlers.

• **Transport**: Provides transport for messages to get into the Axis engine and the return of response messages to the client. Axis has support for HTTP and HTTPS transports.

Ignoring the cylinders in Figure 12 for a moment, the concept here is pretty simple; the client code will invoke the Axis client, which will construct an outgoing message and send it to a SOAP server (for example, using HTTP transport). Then a response (or incoming) message is received back, processed by the Axis client and any results are returned back to the client code.
Now getting back to the cylinders, you’ll notice that there are two types of Request (sometimes referred to as Pre-Processing) Handlers or Chains, and two types of Response (sometimes referred to as Post-Processing) Handlers or Chains invoked. Axis allows you to place a handler into two different stages of a message’s processing flow:

1. **Service-Specific**: The handler will be called just when a specific service is invoked.
2. **Global**: The handler will be invoked for all services invoked.

The triangle in the figure, the Transport Sender, is known as the pivot point. A *pivot point* is just another handler, but it indicates the point at which the request becomes a response.

Axis handlers can only modify the SOAP header part of the message.

So what are handlers good for? While it is true that anything you could do in a handler you could also do in your client-side code, componentizing the logic into handlers gives you several benefits. For one, it allows you to cleanly separate the business logic from your SOAP processing logic. A good example of this is adding security to your SOAP requests. With the definition of new security specifications, such as WS-Security, you’ll want to be able to add these new features to your services (or your SOAP environment) with minimal impact on your code and configuration. By keeping a clean separation between your service and these add-on features, you can add and remove them as needed. Also, as third-party vendors develop handlers, you will be able to plug them into your configuration without any changes to your service. And in fact your service will be totally unaware of their existence.

### Client programming model

The goal of the Web Services Client for ILE is to allow client applications that ability to invoke Web services based on the SOAP 1.1 standard through the HTTP 1.1 or HTTPS SOAP bindings. The client invokes Web service methods without distinguishing whether those are performed locally or in a remote runtime environment. Further, the client has no control over the life cycle of the Web service.

There are two distinct ways of using Web Services Client for ILE:

1. **Stub-Based Invocation**: The first and most commonly used approach is to create a WSDL source file that describes the communication between the client and server, and then use the `wsl2ws.sh` tool (more on this later) to generate stubs (proxy) that you can use to communicate with the web service server.

2. **API-Based Invocation**: The second, more specialized approach, is to use the Web Services Client for ILE APIs to manage messaging between the client and server directly.

Both of these approaches work well, but the second approach is not recommended because the client application writer has to be more aware of the API and of the communication protocols between the client and server. Also, if changes are required, the writer has to make the changes to the code manually, which is more prone to error. The recommended implementation is to use WSDL to describe messages and any associated XSD, and then use the `wsl2ws.sh` tool to generate the stubs.

In the following sections and chapters we will mainly focus on how to call Web services using stub-based invocations. However, if you have an interest in seeing how to use the API-based invocation method to call Web services, simply generate the static stubs and peek at the generated code.

### Stub-based invocation

The Web Services Client for ILE package provides a Java program that is invoked by the `wsl2ws.sh` QShell script. This tool enables you to turn a WSDL into a C, C++, or RPG stub and data objects that you can call and pass information to, and that request information from the server and then wait for the corresponding reply before passing the response data objects back to the client. The stub hides the
internet communication from the application writer. All you need to know is the name of the service, the method it contains and the structure of any data objects that are passed.

The first step in the process is to generate the stub from the WSDL file, as shown in Figure 13.

Figure 13. Process flow between WSDL source file and stubs

Once the stub has been generated, you then create a client application that uses C++ stub method calls (a client application using a C or RPG stub would call a function). This method (C++ stub) or function (C or RPG stub) calls a number of underlying methods in the Axis client library, which generates the SOAP message that communicates with the server. The flow is depicted by Figure 14.

Figure 14. Process flow between client and server applications using stubs generated by WSDL2Ws
Client-side handlers

As has been indicated previously, you can add web service handlers to the Axis client library to allow further processing of the SOAP message, either before it is transmitted to the server or after the corresponding reply has been received from the server.

Web Services Client for ILE supports two basic types of handler:
- The service handler, which is specific to the Web service with which it is associated.
- The global handler, which is called regardless of the Web service port or message name.

A service handler is associated with a particular service/port combination and is only invoked when a SOAP message with the appropriate destination has been called. A global handler is always invoked, regardless of the message destination.

Figure 15 is an amended version of Figure 14 on page 46 and illustrates the placement of handlers in the request and response flows.

Figure 15. Process flow between client and server applications using generated stubs, and request and response handlers

The Web Service Deployment Descriptor (WSDD) file controls what handler is invoked and when it is invoked.

The pivot point is the name given to the point where a message is either written on to or read from the wire. The term wire refers to all the underlying components that are responsible for physically sending or receiving a message on the web. Any handler that works on the request message to be transmitted is a pre-pivot handler and conversely, any handler that works on the response message after it has been received is a post-pivot handler.
For pre-pivot handlers, when a request message is being prepared, the handlers are the last link in the message construction chain, and are invoked just before the message is transmitted, as shown in the flow diagram depicted in Figure 16:

**Figure 16. Process flow for pre-pivot handlers**

For post-pivot handlers, when a response message is being prepared, the handlers are the first link in the message deconstruction chain, and are invoked just after the message is received, as shown in the flow diagram depicted in Figure 17 on page 49.
Web Services Client for ILE supports the following:

- SOAP over HTTP document/literal (WS-I Basic Profile - SOAP 1.1)
- SOAP over HTTPS

**Figure 17. Process flow for post-pivot handlers**

**Binding**

Web Services Client for ILE supports the following:

- SOAP over HTTP document/literal (WS-I Basic Profile - SOAP 1.1)
- SOAP over HTTPS
Data types

Table 4 shows the data types that are supported by Web Services Client for ILE:

<table>
<thead>
<tr>
<th>Category</th>
<th>Schema type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>• byte&lt;br&gt;• decimal&lt;br&gt;• double&lt;br&gt;• float&lt;br&gt;• int&lt;br&gt;• integer&lt;br&gt;• long&lt;br&gt;</td>
</tr>
<tr>
<td></td>
<td>• negativeInteger&lt;br&gt;• nonPositiveInteger&lt;br&gt;• nonNegativeInteger&lt;br&gt;• positiveInteger&lt;br&gt;• unsignedByte&lt;br&gt;• unsignedInt&lt;br&gt;• unsignedLong&lt;br&gt;• unsignedShort&lt;br&gt;</td>
</tr>
<tr>
<td>Date/Time/Duration</td>
<td>• date&lt;br&gt;• dateTime&lt;br&gt;• duration&lt;br&gt;• gDay&lt;br&gt;• gMonth&lt;br&gt;• gMonthDay&lt;br&gt;• gYear&lt;br&gt;• gYearMonth&lt;br&gt;• time</td>
</tr>
<tr>
<td>String</td>
<td>• anyURI&lt;br&gt;• ENTITY&lt;br&gt;• ENTITIES&lt;br&gt;• ID&lt;br&gt;• IDREFS&lt;br&gt;• language&lt;br&gt;• Name&lt;br&gt;• NCName&lt;br&gt;• NMTOKEN&lt;br&gt;• NMTOKENS&lt;br&gt;• normalizedString&lt;br&gt;• notation&lt;br&gt;• QName&lt;br&gt;• string&lt;br&gt;• token</td>
</tr>
</tbody>
</table>
Table 4. Supported schema types (continued)

<table>
<thead>
<tr>
<th>Category</th>
<th>Schema type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Various</td>
<td>• base64Binary</td>
</tr>
<tr>
<td></td>
<td>• boolean</td>
</tr>
<tr>
<td></td>
<td>• hexBinary</td>
</tr>
</tbody>
</table>

In addition to the types in the table above, complex types and arrays of complex types and primitive types are supported. More information on WSDL/XML to programming language mapping will be discussed when we talk about the supported stub target languages later on in this document.

**SOAP faults**

Web Services Client for ILE will map a SOAP fault to a service specific exception. If the SOAP fault does not map to a service specific exception (i.e. not defined in the WSDL file for the service operation), the SOAP fault will map to a generic exception. More information on SOAP Fault mapping will be discussed when we talk about the supported stub target languages later on in this document.

**Message exchange patterns**

Web Services Client for ILE supports the request-response and one-way message exchange patterns.

For the one-way message exchange pattern, the Web Services Client for ILE expects an HTTP response from the invoked Web service, since the HTTP protocol is based on a response being returned to an HTTP request. If the invoked Web service returns a SOAP fault, the Web Services Client for ILE will process the fault and return it to the client.
Chapter 6. Command line tools

Web service client for ILE provides the following commands: wsdl2ws.sh and wsdl2rpg.sh.

The commands can be found in the <install_dir>/bin directory.

The commands must be run from within QShell. There are several ways to run QShell commands:

- Invoke the fully qualified path name of the command from within QShell (to enter the interactive shell session you would issue STRQSH CL command). For example,
  `<install_dir>/bin/wsdl2ws.sh [arguments] WSDL-URI`
- Invoke the script from the IBM i command line or from an IBM i CL program. To use this method, run the STRQSH CL command and specify the fully qualified path name of the script. For example:
  `STRQSH CMD('`<install_dir>/bin/wsdl2ws.sh [arguments] WSDL-URI')`

This chapter will list the supported commands.

The wsdl2ws.sh command line tool

Synopsis

wsdl2ws.sh [arguments] WSDL-URI

Description

The `wsdl2ws.sh` command enables you to generate Web service client stubs (sometimes referred to service interface stubs or Web service client proxies) from a WSDL file. The `wsdl2ws.sh` command uses the WSDL that is passed to it, and any associated XSD files, to create the client stub code.

Required arguments

WSDL-URI

  Specifies the location of the input WSDL file using a Universal Resource Identifier (URI). You can also use a regular file path if the WSDL file is on the local file system.

Optional arguments

- `-h, -help`
  
  Displays a help message and exits.

- `-l<cl|c++|rpg>`
  
  Target language - default is c++. C or RPG stubs can also be generated.

- `-o<directory>`
  
  Sets the root directory for emitted files. Default is the current working directory. If a directory in the specified path `<directory>` does not exist, the directory will be created.

- `-ms<max-string-size>`
  
  Maximum size to use when defining RPG character fields to hold string data. Minimum value is 16. Maximum value for IBM i 5.4 is 65004. Maximum value for IBM i 6.1 and subsequent releases is 80000004. Default is 128. This option is valid only if `-lrgp` is specified.

---

4. Theoretical limit. Actual limit is dependent on system resources and programming language limitations.

© Copyright IBM Corp. 2011, 2012
-mb<max-binary-size>
  Maximum size to use when defining RPG character fields to hold binary data (XSD hexBinary and base64Binary types). Minimum value is 48. Maximum value for IBM i 5.4 is 65004. Maximum value for IBM i 6.1 and subsequent releases is 16000004. This option is valid only if -lrpg is specified.

-ma<max-array-size>
  Maximum size to use when defining RPG arrays. Minimum value is 1. Maximum value for IBM i 5.4 is 320004. Maximum value for IBM i 6.1 and subsequent releases is 80000004. Default is 20. This option is valid only if -lrpg is specified.

-s<service-program>
  Path to service program to be built using the generated code. For example, -s/QSYS.LIB/ MYLIB.LIB/MYWS.SRVPGM. The path can also point to a library, in which case the name of the service program will be the prefix (specified by the -p option) appended with 'WS.SRVPGM'. If the option is not specified, a service program is not built.

-d Generate service program with debug views. If specified, lowest level of optimization is used. Default is to generate fully optimized code with no debug views.

-p<prefix>
  1-3 character prefix. The prefix will be used in the names of module objects (*MODULE). It will also be used in the service program name if a name is not passed on the -s option.

-v Be verbose - will show exception stack trace when exceptions occur.

-t<timeout>
  Specifies how long the command waits, in seconds, for the WSDL-URI to respond before giving up. The default is 0 (no timeout).

-b<binding-name>
  Binding name that will be used to generate stub.

-w<wrapped | unwrapped>
  Generate wrapper style or not - default is wrapped. This affects the definition of the Web service operation that is generated - whether the Web service operation will have one input structure as a parameter (unwrapped) or whether the individual fields in the structure can be passed as individual parameters (wrapper-style). In order for an operation to be eligible for wrapper-style, the following criteria must be met:
  • There is at most one single part in input and output messages.
  • Each part definition must reference an element.
  • The input element must have the same name as the operation.
  • The input and output elements have no attributes.

Usage notes

If you are getting exceptions when you specify a URI that uses HTTP with the SSL protocol (HTTPS), you may need to import the security certificate into the Java runtime environment (JRE) keystore. You will first have to obtain the certificate and save it to a file in the integrated file system. This typically is done by using the WSDL URI (e.g., https://lp02ut18:9080/web/services/ConvertTemp?wsdl) in your web browser and using the browser to view and save the certificate information. The general steps to get the certificate are as follows:
  1. Bring up a browser and use the WSDL URI as the URL and press enter.
  2. You should get a security alert. At this point view the certificate.
  3. Go to tab or click on link that will allow you to view the certificate details.
  4. Export the file (for Internet Explorer ensure format is DER encoded binary X.509 (.CER)) to your system.
Once the certificate is stored on your system, you will now need to import the certificate using the
keytool command into the keystore for the JRE that is being used by the wsd12ws.sh tool. For IBM i 6.1
and previous releases, use the following command from within the QShell shell interpreter:

```
/qopensys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/bin/keytool
-import -trustcacerts -storepass changeit -file <certificate_file>
-keystore /qopensys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/security/cacerts
```

For IBM i 7.1, use the following command from within the QShell shell interpreter:

```
/qopensys/QIBM/ProdData/JavaVM/jdk60/32bit/jre/bin/keytool
-import -trustcacerts -storepass changeit -file <certificate_file>
-keystore /qopensys/QIBM/ProdData/JavaVM/jdk60/32bit/jre/lib/security/cacerts
```

**Note:** The password used for the -storepass option, changeit, is the default password for the keystore. It
may have been changed on your system.

**Examples**

1. The following generates RPG stub code in directory /Stub/rpg using URI to WSDL file and compiles
   the stub code into a service program:
   ```
   wsd12ws.sh -lrpg -o/Stub/rpg -s/qsys.lib/ws.lib/wsrpg.srvpgm
   http://lp02ut18:10021/web/services/ConvertTemp?wsdl
   ```

2. The following generates C stub code in directory /Stub/c using path to WSDL file and compiles the
   stub code into a service program:
   ```
   wsd12ws.sh -lc -o/Stub/c -s/qsys.lib/ws.lib/wsc.srvpgm
   /Stub/ConvertTemp.wsdl
   ```

---

**The wsd12rpg.sh command line tool**

The `wsd12rpg.sh` command has the same options as `wsd12ws.sh` with the exception of the `-l` (target
language) option. Specifying `wsd12ws.sh` with `-lrpg` is the same as using the `wsd12rpg.sh` command.
Chapter 7. Configuration files

The Axis engine will process the following configuration files:

- The axiscpp.conf file
- The Web Service Deployment Descriptor (WSDD) file

Each will be described in the following sections.

The axiscpp.conf file

The Axis configuration file axiscpp.conf affects the processing of the Axis engine if certain configuration properties are inserted in the file. The default axiscpp.conf file is located in `<install_dir>/etc` and is shown below - a file with no properties defined:

```
# The comment character is '

# Available directives are as follows
#
# ClientWSDDFilePath: The path to the client WSDD
# SecureInfo: The GSKit security information
#
```

The axiscpp.conf file supports the following properties:

*Table 5. List of axiscpp.conf configuration file properties*

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientWSDDFilePath</td>
<td>Used to define the path to the Web Services Deployment Descriptor (WSDD) file. The WSDD file contains information on handlers. For example: ClientWSDDFilePath:/conf/clientHandlers.wsdd</td>
</tr>
</tbody>
</table>

See “The Web services deployment descriptor (WSDD) file” on page 59 for further details on the WSDD file.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| SecureInfo   | Used to define SSL information that is to be used by all Web service clients (i.e. you are not setting the SSL information programmatically). The property value contains comma-delimited strings as follows (should be all one line): SecureInfo:keyRingFile,keyRingPasswordOrStash,keyRingLabel,v2CipherSpec,v3CipherSpec,tlsCipherSpec where:  
keyRingFile  
  Full path and filename to the certificate store file to be used for the secure session or SSL environment.  
keyRingPassword  
The password for the certificate store file to be used for the secure session or SSL environment.  
keyRingLabel  
The certificate label associated with the certificate in the certificate store to be used for the secure session or SSL environment.  
v2CipherSpec  
The list of SSL Version 2 ciphers to be used for the secure session or the SSL environment. Specifying NONE for this field will disable SSL Version 2 ciphers. Valid values: 01, 02, 03, 04, 06 or 07.  
v3CipherSpec  
The list of SSL Version 3/TLS Version 1 ciphers to be used for the secure session or the SSL environment. Specifying NONE for this field will disable SSL Version 3 ciphers. Valid values: 00, 01, 02, 03, 04, 05, 06, 09, 35, 0A, 2F, or 35.  
tlsCipherSpec  
Whether to enable or disable TLS Version 1 ciphers. A value of NONE will disable the ciphers; any other value will enable the ciphers. By default, the TLS Version 1 ciphers are enabled. For example:  
SecureInfo:/sslkeys/myKeyRing.kdb,axis4all,AXIS,NONE,05,NONE  
To set the security information programmatically, see the programming considerations chapter for the programming language you are interested in. |

You only need to change this file if you are using handlers or securing your Web service communications using SSL and you do not want to set SSL information programmatically. If you do need to add properties to the file, then the following steps must be taken:  
1. Create a directory.  
2. Copy the \(<install_dir>/etc \) directory and directory contents into the newly created directory.  

**Note:** It is important that you copy the directory and not update the configuration file that is shipped with the product since any updates to the file will be lost when product PTFs are installed.  
3. Reveal to the Axis engine the location of updated Axis configuration file by defining the AXISCPP_DEPLOY environment and by using the CL command ADDENVVAR as follows:  

\( \text{ADDENVVAR ENVVAR(AXISCPP_DEPLOY) VALUE(’<MYINSTALL_DIR>’)} \)  

where \(<MYINSTALL_DIR>\) is the path to the directory created in step 1. The environment variable must be set in the job where the Web service application is to be run.
The Web services deployment descriptor (WSDD) file

The WSDD file contains the rules governing when the Axis engine must invoke a handler library (i.e. service program). Service handlers and global handlers are defined in separate sections of the WSDD file.

The WSDD file is an XML style file containing information that the Axis engine uses as it builds request messages and decodes response messages. A WSDD file has two main sections, one for service handlers and one for global handlers. For service handlers, each service definition that requires a handler must be defined with the appropriate handler list given for pre-pivot and post-pivot invocation. For global handlers, the WSDD file only needs to list those handlers that are to be invoked pre-pivot and post-pivot.

Below is a sample WSDD file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/">
    <service name="Handler" provider="CPP:DOCUMENT" description="Handler">
        <requestFlow>
            <handler name="myClientHandlerReq" type="/qsys.lib/samples.lib/handler.srvpgm"/>
        </requestFlow>

        <responseFlow>
            <handler name="myClientHandlerRes" type="/qsys.lib/samples.lib/handler.srvpgm"/>
        </responseFlow>
    </service>

    <globalConfiguration name="GlobalHandler" provider="CPP:DOCUMENT" description="Global Handler">
        <requestFlow>
            <handler name="myGlobalHandlerReq" type="/qsys.lib/samples.lib/glbhandler.srvpgm"/>
        </requestFlow>

        <responseFlow>
            <handler name="myGlobalHandlerRes" type="/qsys.lib/samples.lib/glbhandler.srvpgm"/>
        </responseFlow>
    </globalConfiguration>
</deployment>
```

Here is an explanation of the XML element tags in the WSDD file:

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The deployment tag is the root element.</td>
</tr>
<tr>
<td>2</td>
<td>The service tag defines the Axis service for which the handlers specified within the service tag are invoked whenever the web service is used.</td>
</tr>
<tr>
<td></td>
<td>The name attribute defines the name of the service and is used by the Axis engine to determine when the handler is invoked. It does this by comparing the value specified in the name attribute with the SOAP action (soapAction in WSDL) specified for a service. Thus, the SOAP action must either appear in the WSDL or set in the client application if the handler is to be called by the Axis engine. For the example above, the SOAP action must be set to #handler.</td>
</tr>
<tr>
<td></td>
<td>The provider attribute defines the name of the provider for the service and must be set to CPP:DOCUMENT.</td>
</tr>
<tr>
<td></td>
<td>The description attribute defines a comment for this line and is not used by the Axis engine.</td>
</tr>
<tr>
<td>Line Number</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>3</td>
<td>The <code>requestFlow</code> tag defines the start of a list of one or more handlers that are invoked when a request message is about to be transmitted. The handlers are invoked in the order in which they appear in the WSDD file.</td>
</tr>
<tr>
<td>4</td>
<td>The <code>handler</code> tag within a <code>requestFlow</code> defines the unique name of a handler to be invoked, and the type, which is a fully qualified path to the location of the handler.</td>
</tr>
<tr>
<td>5</td>
<td>The <code>responseFlow</code> tag defines the start of a list of one or more handlers that are invoked when a response message has just been received. The handlers are invoked in the order in which they appear in the WSDD file.</td>
</tr>
<tr>
<td>6</td>
<td>The <code>handler</code> tag within a <code>responseFlow</code> defines the unique name of a handler to be invoked, and the type, which is a fully qualified path to the location of the handler.</td>
</tr>
</tbody>
</table>
| 7 | The `globalConfiguration` tag defines the handlers that are not specific to a web service and are called regardless of what web service is used.  
The `name` attribute defines the name of the global handler and is not used by the Axis engine.  
The `provider` attribute defines the name of the provider for the service and must be set to `CPP:DOCUMENT`.  
The `description` attribute defines a comment for this line and is not used by the Axis engine. |
Chapter 8. The Web services client for ILE installation details

This chapter describes the Web services client for ILE package, including what you need to do to install Web services client for ILE and a description of the various components that make up the Web services client for ILE package.

Installing Web services client for ILE

Web services client for ILE is included in option 3 (Extended Base Directory Support) of the base operating system (e.g. 5722SS1 for IBM i 5.4, 5761SS1 for i 6.1, 5770SS1 for i 7.1, etc.).

In addition to installing base option 3 of the operating system, the following prerequisite products will also need to be installed:

- Qshell - base option 30 of operating system
- PASE - base option 33 of operating system
- Digital Certificate Manager - base option 34 of operating system
- IBM Technology for Java Virtual Machine - J2SE 5.0 32 bit (option 8 of 57xxJV1) or Java SE 6 32 bit (57xxJV1 Option 11)
- One or more of the following ILE compilers:
  - ILE C++ - option 52 of 57xxWDS. The ILE C++ compiler does not need to be installed if you do not plan on generating C++ stubs.
  - ILE C - option 51 of 57xxWDS. The ILE C compiler does not need to be installed if you do not plan on generating C or RPG stubs.
  - ILE RPG - option 31 of 57xxWDS. The ILE RPG compiler does not need to be installed if you do not plan on generating RPG stubs.

The Web services client for ILE package

The installation directory for Web Services Client for ILE is /QIBM/ProdData/OS/WebServices/V1/client.

In this chapter, and throughout this documentation, the installation directory is shown as <install_dir>.

When the package has been installed, the installation directory (<install_dir>) contains the following directory structure:
The following table gives an overview of the contents of each directory:

<table>
<thead>
<tr>
<th>Installed directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;install_dir&gt;/bin</td>
<td>Contains the wsdl2ws.sh and wsdl2rpg.sh QShell scripts that are used to invoke the Java tool in order to generate Web service stubs.</td>
</tr>
<tr>
<td>&lt;install_dir&gt;/docs</td>
<td>Contains the Web Services Client for ILE API documentation in HTML format.</td>
</tr>
<tr>
<td>&lt;install_dir&gt;/etc</td>
<td>This directory level contains an example axiscpp.conf configuration file.</td>
</tr>
<tr>
<td>&lt;install_dir&gt;/include</td>
<td>Web Services Client for ILE header files, which are required for building web service specific generated stubs.</td>
</tr>
<tr>
<td>&lt;install_dir&gt;/lib</td>
<td>Contains all of the built libraries needed for building web service specific generated stubs.</td>
</tr>
<tr>
<td>&lt;install_dir&gt;/prereqs/java</td>
<td>The prerequisite Java jar files that are required for the wsdl2ws.sh and wsdl2rpg.sh QShell scripts.</td>
</tr>
<tr>
<td>&lt;install_dir&gt;/samples</td>
<td>The location of the samples that accompany Web Services Client for ILE.</td>
</tr>
<tr>
<td>&lt;install_dir&gt;/WSDL2Ws</td>
<td>The Java tool that is used to generate Web service stubs.</td>
</tr>
</tbody>
</table>
Part 3. Using C++ stubs

This part of the document provides details regarding all things related C++ stub programming. If you have no interest in C++ stub programming, you should skip this part of the document.

- Chapter 9, “WSDL and XML to C++ mappings,” on page 65 describes in detail how WSDL and XML constructs are mapped to C++ constructs.
- Chapter 10, “Developing a Web services client application using C++ stubs,” on page 75 is a step-by-step description on how to develop a client application using a C++ stub.
- Chapter 12, “C++ programming considerations,” on page 83 discusses various programming topics relating to using the C++ stubs, including how to set up an SSL connection.
- Chapter 13, “Troubleshooting C++ client stubs,” on page 95 describes how to troubleshoot command line tool and runtime problems.
- Chapter 14, “Axis C++ core APIs,” on page 97 Summarizes the core (i.e. most commonly used) Axis C++ classes and methods.
Chapter 9. WSDL and XML to C++ mappings

The `wsdl2ws.sh` command tool can generate C++ stub code. This chapter will describe the mappings from WSDL and XML Schema types to C++ language constructs.

Mapping XML names to C++ identifiers

XML names are much richer than C++ identifiers. They can include characters that are either reserved or not permitted in C++ identifiers. The `wsdl2ws.sh` command generates unique and valid names for C++ identifiers from the schema element names using the following rules:

1. Invalid characters are replaced by underscore (`_`). Invalid characters include the following characters: `/ ! * $ % & ' ( ) + , . : ; < = > ? @ \ ^ ` { } [ ]`
2. Names that conflict with C++ keywords will have an underscore inserted at the beginning of the name. For example, an XML element name of `register` will be generated as a C++ identifier of `_register`.
3. If a name that is used as a C++ identifier conflicts with a class with the same name, the identifier will have `_Ref` appended to the name.

XML schema to C++ type mapping

Table 7 specifies the C++ mapping for each built-in simple. The table shows the XML Schema type and the corresponding the Axis type (column 2), which generally is a typedef to a C++ language built-in type (column 3).

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Axis Type</th>
<th>Actual C++ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Numeric</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>xsd:byte</code></td>
<td><code>xsd__byte</code></td>
<td><code>signed char</code></td>
</tr>
<tr>
<td><code>xsd:decimal</code></td>
<td><code>xsd__decimal</code></td>
<td><code>double</code></td>
</tr>
<tr>
<td><code>xsd:double</code></td>
<td><code>xsd__double</code></td>
<td><code>double</code></td>
</tr>
<tr>
<td><code>xsd:float</code></td>
<td><code>xsd__float</code></td>
<td><code>float</code></td>
</tr>
<tr>
<td><code>xsd:int</code></td>
<td><code>xsd__int</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>xsd:integer</code></td>
<td><code>xsd__integer</code></td>
<td><code>long long</code></td>
</tr>
<tr>
<td><code>xsd:long</code></td>
<td><code>xsd__long</code></td>
<td><code>long long</code></td>
</tr>
<tr>
<td><code>xsd:negativeInteger</code></td>
<td><code>xsd__negativeInteger</code></td>
<td><code>long long</code></td>
</tr>
<tr>
<td><code>xsd:nonPositiveInteger</code></td>
<td><code>xsd__nonPositiveInteger</code></td>
<td><code>long long</code></td>
</tr>
<tr>
<td><code>xsd:nonNegativeInteger</code></td>
<td><code>xsd__nonNegativeInteger</code></td>
<td><code>unsigned long long</code></td>
</tr>
<tr>
<td><code>xsd:positiveInteger</code></td>
<td><code>xsd__positiveInteger</code></td>
<td><code>unsigned long long</code></td>
</tr>
<tr>
<td><code>xsd:unsignedByte</code></td>
<td><code>xsd__unsignedByte</code></td>
<td><code>unsigned char</code></td>
</tr>
<tr>
<td><code>xsd:unsignedInt</code></td>
<td><code>xsd__unsignedInt</code></td>
<td><code>unsigned int</code></td>
</tr>
<tr>
<td><code>xsd:unsignedLong</code></td>
<td><code>xsd__unsignedLong</code></td>
<td><code>unsigned long long</code></td>
</tr>
<tr>
<td><code>xsd:unsignedShort</code></td>
<td><code>xsd__unsignedShort</code></td>
<td><code>unsigned short</code></td>
</tr>
<tr>
<td><code>xsd:short</code></td>
<td><code>xsd__short</code></td>
<td><code>short</code></td>
</tr>
</tbody>
</table>

**Date/Time/Duration**

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Axis Type</th>
<th>Actual C++ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xsd:date</code></td>
<td><code>xsd__date</code></td>
<td><code>struct tm</code></td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 2011, 2012
<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Axis Type</th>
<th>Actual C++ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:dateTime</td>
<td>xsd__dateTime</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:duration</td>
<td>xsd__duration</td>
<td>long</td>
</tr>
<tr>
<td>xsd:gDay</td>
<td>xsd__gDay</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:gMonth</td>
<td>xsd__gMonth</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:gMonthDay</td>
<td>xsd__gMonthDay</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:gYear</td>
<td>xsd__gYear</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:gYearMonth</td>
<td>xsd__gYearMonth</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:time</td>
<td>xsd__time</td>
<td>struct tm</td>
</tr>
</tbody>
</table>

**String**

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Axis Type</th>
<th>Actual C++ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:anyURI</td>
<td>xsd__anyURI</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:anyType</td>
<td>xsd__anyType</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:ENTITY</td>
<td>xsd__ENTITY</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:ENTITIES</td>
<td>xsd__ENTITIES</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:ID</td>
<td>xsd__ID</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:IDREFS</td>
<td>xsd__IDREFS</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:language</td>
<td>xsd__language</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:Name</td>
<td>xsd__Name</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:NCName</td>
<td>xsd__NCName</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:NM_TOKEN</td>
<td>xsd__NM_TOKEN</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:NM_TOKENS</td>
<td>xsd__NM_TOKENS</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:normalizedString</td>
<td>xsd__normalizedString</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:notation</td>
<td>xsd__notation</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:QName</td>
<td>xsd__QName</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:string</td>
<td>xsd__string</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:token</td>
<td>xsd__token</td>
<td>char *</td>
</tr>
</tbody>
</table>

**Other**

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Axis Type</th>
<th>Actual C++ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:base64Binary</td>
<td>xsd__base64Binary</td>
<td>Implemented as C++ class</td>
</tr>
<tr>
<td>xsd:boolean</td>
<td>xsd__boolean</td>
<td>enum</td>
</tr>
<tr>
<td>xsd:hexBinary</td>
<td>xsd__hexBinary</td>
<td>Implemented as C++ class</td>
</tr>
</tbody>
</table>

The Axis types are defined in the header file `<install_dir>/include/AxisUserAPI.hpp`. The struct `tm` structure used for many of the time-related types can be found in header file `time.h`.

**Simple types**

Most of the simple XML data types defined by XML Schema and SOAP 1.1 encoding are mapped to their corresponding C++ types. You can see the details of the mapping in Table 7 on page 65 above.

One thing to keep in mind is how an element declaration with a `nillable` attribute set to true for a built-in simple XML data type is mapped. If the simple type is not already a pointer type (i.e. all the string types are pointer types), the simple type will be mapped to a pointer type. For example, the following schema fragment will get mapped to an integer pointer type (i.e. `xsd__int *`):

```
<xsd:element name="code" type="xsd:int" nillable="true"/>
```
In addition, a simple type that is optional (minoccurs attribute set to 0) will also be mapped to a pointer type if the type is not already a pointer type.

**Complex types**

XML Schema complex types are mapped to C++ classes with getters and setters to access each element in the complex type.

Let us look at the mapping that occurs for the following schema fragment:

```xml
<xsd:complexType name="Book">
  <sequence>
    <element name="author" type="xsd:string"/>
    <element name="price" type="xsd:float"/>
  </sequence>
  <xsd:attribute name="reviewer" type="xsd:string"/>
</xsd:complexType>
```

The above example is an example of a complex type that is named Book, and contains two elements, author and price, in addition to an attribute, reviewer. The complex type will get mapped to the following C++ class:

```cpp
class Book
{
  public:
    xsd__string reviewer;
    xsd__string author;
    xsd__float price;
    xsd__string getreviewer();
    void setreviewer(xsd__string InValue, bool deep = true);
    xsd__string getauthor();
    void setauthor(xsd__string InValue, bool deep = true);
    xsd__float getprice();
    void setprice(xsd__float InValue);
  .
  .
};
```

So let us discuss what is generated. The class name is the name of the complex type. There are setter and getter methods for elements as well as attributes. The setter methods have an additional parameter, deep, with a default value of true. This parameter will always be generated for pointer types (but only if type is simple), and is an indication whether the object should make a deep or shallow copy of the data. A deep copy means that memory will be allocated and an exact copy of the data will be created and stored in the object, and when the destructor for the object gets called the memory allocated to store that data will be deleted. A shallow copy means a copy of the pointer is stored in the object, but the caller still owns the data and any memory resources associated with the data, so when the destructor of the object is called the memory will not be deleted (and user must ensure not to delete resources until the object is reclaimed).

In addition to the Book C++ class, the following functions are generated:

```cpp
int Axis_Serialize_Book(Book* param, IWrapperSoapSerializer* pSZ, bool bArray=false);
int Axis_DeSerialize_Book(Book* param, IWrapperSoapDeSerializer* pDZ);
void* Axis_Create_Book(int nSize=0);
void Axis_Delete_Book(Book* param, int nSize=0);
```

The `Axis_Serialize_Book()` and `Axis_DeSerialize_Book()` functions are used by the Axis engine to serialize and deserialize elements of type Book. The Axis engine uses the `Axis_Create_Book()` function to create the C++ class that will hold the data during deserialization. The nSize parameter is used to
indicate whether a single (i.e. when nSize equals to zero) class is to be returned or an array (i.e. when nSize greater than zero) of classes is to be returned. The Axis_Delete_Book() is the function used by client applications to free up C++ objects of type Book that are returned by the Axis engine. In the case of Axis_Delete_Book(), the nSize parameter is used to indicate whether a single class is to be deleted or an array of classes is to be deleted.

Arrays

Axis defines the class Axis_Array as the parent class for all arrays. The class is defined in the header file <install_dir>/include/axis/AxisUserAPI.hpp. The class is depicted below:

class Axis_Array
{
public:
    Axis_Array();
    Axis_Array(const Axis_Array & original);
    virtual ~Axis_Array();
    void clone(const Axis_Array & original);
    virtual Axis_Array * clone() const;
    void set(void** array, int size, XSDTYPE type);
    void** get(int& size, XSDTYPE& type) const;
    void clear();
    void addElement(void* element);
protected:
    void** m_Array;
    int m_Size;
    XSDTYPE m_Type;
    bool m_belongsToAxisEngine;
};

To access elements of the array, one would use the get() method, which will return a C-style array. When calling get() method, two parameters are passed by-reference: the size and type (XSDTYPE is an enumerator defined in <install_dir>/include/axis/TypeMapping.hpp) parameters. Upon successful completion of the get() method, size will be set to the number of elements in the array and type will indicate the element type.

Axis provides array objects for each of the defined simple types. These are defined in <install_dir>/include/Axis/AxisUserAPIArrays.hpp. An example of a simple array type is xsd__int_Array.

Below is the same schema fragment we have used previously, but we have also increased the number of authors a book can have to 10 by adding maxOccurs="10" to the author element:

```
<xsd:complexType name="Book">
    <sequence>
        <element name="author" type="xsd:string" maxOccurs="10"/>
        <element name="price" type="xsd:float"/>
    </sequence>
    <xsd:attribute name="reviewer" type="xsd:string"/>
</xsd:complexType>
```

For the above XML Schema, the following class is generated:

class STORAGE_CLASS_INFO Book
{
    public:
    xsd__string reviewer;
    class xsd__string_Array* author;
    xsd__float price;

    xsd__string getreviewer();
    void setreviewer(xsd__string InValue, bool deep = true);

    xsd__string_Array* getauthor();
}
void setauthor(xsd__string_Array* InValue);

xsd__float getprice();
void setprice(xsd__float InValue);

As you can see, the string array class is now being used to store the values for the author element.

---

**WSDL to C++ mapping**

Now that we understand how the XML Schema types are mapped to Axis-defined language types, we can now review how a service described in a WSDL document gets mapped to the corresponding C++ representation. The following sections will refer to the GetQuote.wsdl WSDL document that is shipped as part of the product in directory `<install_dir>/samples/getQuote` and is listed in The GetQuote.wsdl File on page 187 to illustrate how various WSDL definitions get mapped to C++. You should note the following:

- GetQuote.wsdl has only one service called GetQuoteService.
- The service only has one port type called StockQuote.
- The StockQuote port type has only one operation called getQuote. The input to the getQuote operation is a string (the stock identifier) and the output from the operation is a float (the stock's price).

If you want to fully understand the WSDL document structure, see "WSDL document structure" on page 32. Now let us see how various WSDL definitions are mapped. The following table summarizes the WSDL and XML to C++ mappings:

<table>
<thead>
<tr>
<th>WSDL and XML</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:complexType (structure)</td>
<td>C++ class.</td>
</tr>
<tr>
<td>Note: The xsd:complexType can also represent a C++ exception if referenced by a wsdl:message for a wsdl:Fault.</td>
<td></td>
</tr>
<tr>
<td>Nested xsd:element or xsd:attribute</td>
<td>C++ class property (i.e. a field in the class with getter and setter methods)</td>
</tr>
<tr>
<td>xsd:complexType (array)</td>
<td>C++ Axis array class.</td>
</tr>
<tr>
<td>wsdl:message determines the method parameter signature.</td>
<td>Service interface method signature.</td>
</tr>
<tr>
<td>wsdl:portType</td>
<td>Service interface.</td>
</tr>
<tr>
<td>wsdl:operation</td>
<td>Service interface method.</td>
</tr>
<tr>
<td>wsdl:binding</td>
<td>No direct mapping, affects SOAP communications style and transport.</td>
</tr>
<tr>
<td>wsdl:service</td>
<td>No direct mapping.</td>
</tr>
<tr>
<td>wsdl:port</td>
<td>Used as default Web service location.</td>
</tr>
</tbody>
</table>

**Mapping XML defined in wsdl:types**

The wsdl2ws.sh command will either use an existing C++ type or generate a new C++ type (a C++ class) for the XML schema constructs defined in the wsdl:types section. The mappings that the wsdl2ws.sh command supports is discussed in "XML schema to C++ type mapping" on page 65. In general, the wsdl2ws.sh command either will ignore constructs that it does not support or issue an error message.
If we look at the wsdl:types part of the WSDL document we see that two elements are defined: getQuote, defined as a complex type with one element of type xsd:string; and getQuoteResponse, also defined as a complex type with one element of type xsd:float.

```xml
<wsdl:types>
  <ati:schema elementFormDefault="qualified"
    targetNamespace="http://stock.ibm.com"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:ati="http://www.w3.org/2001/XMLSchema">
    <ati:element name="getQuote">
      <ati:complexType>
        <ati:sequence>
          <ati:element name="arg_0_0" type="xsd:string"/>
        </ati:sequence>
      </ati:complexType>
    </ati:element>
    <ati:element name="getQuoteResponse">
      <ati:complexType>
        <ati:sequence>
          <ati:element name="getQuoteReturn" type="xsd:float"/>
        </ati:sequence>
      </ati:complexType>
    </ati:element>
  </ati:schema>
</wsdl:types>
```

For the WSDL document fragment above, the wsdl2ws.sh command does not generate any new classes since both elements are defined to be built-in simple types. The xsd:string type is mapped to xsd__string and the xsd:float type is mapped to xsd__float.

**Mapping of wsdl:portType**

A port type is a named set of abstract operations and the abstract messages involved. The name of the wsdl:portType will be used as the name of the Web service proxy (termed service interface) class. All service interface classes inherit from the Stub class, which is the interface between the service interface class and the Axis engine. The Stub class header file is located in <install_dir>/include/axis/client/Stub.hpp.

Now let us see how the wsdl:portType below gets mapped.

```xml
<wsdl:portType name="StockQuote">
  <wsdl:operation name="getQuote">
    <wsdl:input message="impl:getQuoteRequest" name="getQuoteRequest"/>
    <wsdl:output message="impl:getQuoteResponse" name="getQuoteResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

The wsdl2ws.sh command will generate a C++ class named StockQuote. The service interface class will contain methods mapped from the wsdl:operation elements defined in the wsdl:portType (refer to "Mapping of wsdl:operation" on page 71 for further explanation of the mapping of wsdl:operation). The above WSDL port type definition maps to the following C++ service interface:

```cpp
class StockQuote : public Stub
{
  public:
    StockQuote(const char* pchEndpointUri, AXIS_PROTOCOL_TYPE eProtocol=APTHHTTP1_1);
    StockQuote();
    public:
```
virtual ~StockQuote();
public:
xsd__float getQuote(xsd__string Value0);
};

One thing to notice about the service interface class is that there are two constructors. If the one without parameters is used, then the default URL to the Web service will be used, which is whatever is specified in wsdl:port. If the constructor with parameters is used, you can specify a URL to the Web service in addition to specifying a transport protocol. However, the only protocol that is supported by Web Services Client for ILE is HTTP.

**Mapping of wsdl:operation**

A wsdl:operation within a wsdl:portType is mapped to a method of the service interface. The name of the wsdl:operation is mapped to the name of the method.

The wsdl:operation contains wsdl:input and wsdl:output elements that reference the request and response wsdl:message constructs using the message attribute. Each method parameter is defined by a wsdl:message part referenced from the input and output elements:

- A wsdl:part in the request wsdl:message is mapped to an input parameter.
- A wsdl:part in the response wsdl:message is mapped to the return value.
- If there are multiple wsdl:parts in the response message, they are mapped to output parameters.
- A wsdl:part that is both the request and response wsdl:message is mapped to an inout parameter.

The wsdl:operation can contain wsdl:fault elements that references wsdl:message elements describing the fault (refer to [Mapping of wsdl:fault on page 72](#)) for more details on wsdl:fault mapping.

The Web Services Client for ILE supports the mapping of operations that use either a request/response or one-way (where wsdl:output is not specified in the wsdl:operation element) message exchange pattern. For the one-way message exchange pattern, the Axis engine expects an HTTP response to be returned from the Web service. Under normal conditions, the HTTP response would contain no SOAP data. However, if a SOAP fault is returned by the Web service, the Axis engine will process the fault and throw a C++ exception.

Below are the wsdl:message and wsdl:portType WSDL definitions in the GetQuote.wsdl document:

```xml
...<wsdl:message name="getQuoteRequest">
  <wsdl:part element="impl:getQuote" name="parameters"/>
</wsdl:message>

<wsdl:message name="getQuoteResponse">
  <wsdl:part element="impl:getQuoteResponse" name="parameters"/>
</wsdl:message>
...

<wsdl:portType name="StockQuote">
  <wsdl:operation name="getQuote">
    <wsdl:input message="impl:getQuoteRequest" name="getQuoteRequest"/>
    <wsdl:output message="impl:getQuoteResponse" name="getQuoteResponse"/>
  </wsdl:operation>
</wsdl:portType>
...
```

The above wsdl:operation definition gets mapped to the following service interface method:

```
xsd__float getQuote(xsd__string Value0);
```
Mapping of wsdl:binding

The wsdl:binding information is used to generate an implementation specific client side stubs. What code is generated is dependent on protocol-specific general binding data, such as the underlying transport protocol and the communication style of SOAP.

There is no C++ representation of the wsdl:binding element.

Mapping of wsdl:port

A wsdl:port definition describes an individual endpoint by specifying a single address for a binding. The specified endpoint will be used as the default location of the Web service. So in the case of our example, the URL specified in wsdl:port definition below will be the URL that is used when you construct a StockQuote object using the StockQuote() constructor.

```xml
<wsdl:service name="GetQuoteService">
  <wsdl:port name="StockQuote" binding="impl:StockQuoteSoapBinding">
    <wsdl:soap:address location="http://localhost:9080/StockQuote/services/GetQuoteService"/>
  </wsdl:port>
</wsdl:service>
```

Mapping of wsdl:fault

Within the wsdl:operation definition you can optionally specify the wsdl:fault element, which specifies the abstract message format for any error messages that may be returned as a result of invoking a Web service operation.

The wsdl:fault element must reference a wsdl:message that contains a single message part. As of this writing, Axis only supports message parts that are xsd:complexType types. The mapping that occurs is similar to the mapping that occurs when generating code for complex types. However, the C++ class that is generated will inherit from the SoapFaultException class in order to store standard SOAP fault related information such as the faultcode, faultstring, faultactor, etc (for more information on SOAP faults, see "Error handling (SOAP faults)" on page 25). The SoapFaultException class header file is located in <install_dir>/include/axis/SoapFaultException.hpp.

Let us look at an example. If we extend the GetQuote.wsdl WSDL document by adding the following element in the wsdl:types definitions:

```xml
<element name="getQuoteFault">
  <complexType>
    <sequence>
      <element name="errorInfo" type="xsd:string"/>
      <element name="errorReturnCode" type="xsd:int"/>
    </sequence>
  </complexType>
</element>
```

And also adding a new wsdl:message definition that is referenced in the wsdl:operation and wsdl:binding definitions:

```xml
<wsdl:message name="getQuoteFault">
  <wsdl:part element="impl:getQuoteFault" name="fault"/>
</wsdl:message>
```

...
Now with the changes made above, the type mapping will result in the following C++ SOAP fault exception class:

class getQuoteFault : public SoapFaultException
{
public:
    xsd_string errorInfo;
    xsd_int errorReturnCode;

    xsd_string geterrorInfo();
    void seterrorInfo(xsd_string InValue, bool deep = true);

    xsd_int geterrorReturnCode();
    void seterrorReturnCode(xsd_int InValue);

    getQuoteFault();
    getQuoteFault(const getQuoteFault & original);

    void reset();
    virtual ~getQuoteFault() throw();

private:
    bool __axis_deepcopy_errorInfo;
};

This exception class will be thrown by the getQuote() method if a Web service returns a SOAP fault that has an element name that matches getQuoteFault. So what happens to SOAP faults that do not match any specified in wsdl:fault? That is where the OtherFaultException C++ class comes into play. The OtherFaultException exception is used to throw SOAP faults that do not match any specified in wsdl:fault (or if wsdl:fault was not even specified). The class inherits from the SoapFaultException class and contains the getFaultDetail() method that simply returns the SOAP fault contents as a string.

More information on exception handling in C++ can be found in "C++ exception handling" on page 83.
Chapter 10. Developing a Web services client application using C++ stubs

This chapter will describe the steps one must go through to develop a Web service client application using a C++ stub code.

To develop a Web services client application, the following steps should be followed:
1. Generate the client Web service stubs using the wsd12ws.sh command.
2. Complete the client implementation.
3. (Optional) Create client-side handler.
4. Deploy the application.

The following sections will discuss each of these steps. For illustrative purposes we will be using the sample code that is shipped as part of the product in directories <install_dir>/samples/getQuote. We will be using the following files:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetQuote.wsdl</td>
<td>GetQuote WSDL file.</td>
</tr>
<tr>
<td>myGetQuote.cpp</td>
<td>Client implementation code written in C++.</td>
</tr>
</tbody>
</table>

Source listings for the client application code can be found at Appendix A, “Code Listings for myGetQuote Client Application,” on page 187.

Generating the C++ stub code

Before you can create a web service client application, you must first generate the C++ client stub using the wsd12ws.sh tool. The wsd12ws.sh tool uses the WSDL file that is passed to it, and any associated XSD files referenced in the WSDL file, to create client stubs.

We will be using the GetQuote.wsdl file located in directory <install_dir>/samples/getQuote. The WSDL file comes from the installation Web Services Samples provided with WebSphere® Application Server (Version 5.0 or later). This very simple sample provides a good introduction to using wsd12ws.sh.

To generate the client stub from the WSDL source file, complete the following steps.
1. Create a library called MYGETQUOTE in which the program objects will be stored by issuing the CL command CRTLIB from the CL command line as follows:
   CRTLIB MYGETQUOTE
2. Start a Qshell session by issuing the QSH CL command from the CL command line.
3. Run the wsd12ws.sh tool to generate the client C++ stub as shown in following example:
   ```bash
   <install_dir>/bin/wsd12ws.sh -o/myGetQuote/CPP
   -s/qsys.lib/mygetquote.lib/wscpp.srvpgm
   <install_dir>/samples/getQuote/GetQuote.wsdl
   ```

   If you examine the command, you see that we are indicating to the wsd12ws.sh tool that the generated stub code should be stored in directory /myGetQuote/CPP, and that a service program, /qsys.lib/mygetquote.lib/wscpp.srvpgm, should be created using the generated stub code.

   The files generated by the wsd12ws.sh tool is shown below:
Note that in addition to C++ code being generated, the file ws.cl is also generated. This file is a CL source file that has the CL commands needed to recreate the service program. You can copy this source file to a source physical file and create a CL program. Here is the contents of the file:

```cl
PGM
DCL VAR(&LIB) TYPE(*CHAR) LEN(10) VALUE(MYGETQUOTE)
DCL VAR(&SRVPGM) TYPE(*CHAR) LEN(10) VALUE(WSCPP)
QSYS/CRTCPPMOD MODULE(&LIB/wsc0) +
  OPTIMIZE(40) DBGVIEW(*NONE) +
  SRCSTMF('/myGetQuote/CPP/StockQuote.cpp') +
  INCDIR('/QIBM/PRODDATA/OS/WEBSERVICES/V1/CLIENT/INCLUDE') +
  REPLACE(*YES) ENUM(*INT) +
  TEXT('StockQuote.cpp')
QSYS/CRTSRVPGM SRVPGM(&LIB/&SRVPGM) +
  MODULE( +
    &LIB/wsc0 +
  ) +
  EXPORT(*ALL) ACTGRP(*CALLER) +
  BNDSRVPGM(QSYSDIR/QAXIS10C) +
  TEXT('ws Web service')
ENDPGM
```

Now that the C++ stub code has been created and a service program containing the C++ stub code created, you can go on to the next step, “Completing C++ client implementation.”

**Completing C++ client implementation**

After the client stubs have been generated, the stubs can be used to create a Web service client application.

We will illustrate what you need to do to create C++ applications using the example of the C++ stub code generated from GetQuote.wsdl by the wsd12ws.sh tool as described in “Generating the C++ stub code” on page 75. However, before we continue, you should note the following points:

- GetQuote.wsdl has only one service called getQuoteService.
- The service only has one port type called StockQuote.
- The StockQuote port type has only one operation called getQuote.
- The Web service is called StockQuote. The Web service is implemented as a class of the same name. You can create either a dynamic or static instance of the class and then call any available public method.

To build the myGetQuote client application, complete the following steps.

1. Change the current working directory to the location of the C++ stub code. Issue the following command from the CL command line:
   ```cl
cd '/myGetQuote/CPP'
```
2. Copy the sample C++ code that uses the generated stub code from the product samples directory to the current working directory by issuing the following command from the CL command line:
   ```cl
COPY OBJ('<install_dir>/samples/getQuote/myGetQuote.cpp') TODIR('/myGetQuote/CPP')
```

5. If you have not read Chapter 9, “WSDL and XML to C++ mappings,” on page 65 then it would be a good time to do so prior to reading this section.
3. Change the ServerName and PortNumber in the file copied in the previous step to match your server. If WebSphere Application Server is on your own machine and the default values have been used, ServerName is localhost and PortNumber is 9080.

4. Build the client application by using the following commands from the CL command line:

   CRTCPPMOD MODULE(MYGETQUOTE/mygetquote)
   
   SRCSTMF('/myGetQuote/CPP/myGetQuote.cpp')
   
   INCDIR('/qibm/proddata/os/webservices/v1/client/include')
   
   ENUM(*INT)

   CRTPGM PGM(MYGETQUOTE/MYGETQUOTE)
   
   MODULE(MYGETQUOTE/MYGETQUOTE)
   
   BNDSRVPGM(QSYSDIR/QAXIS10C MYGETQUOTE/WSCPP)

   When you have finished coding and building your web service client application, you are ready to deploy and test the application as described in “Deploying the client application.”

   **Note:** If you want to use one or more handlers with your application, see Chapter 11, “Creating client-side handlers,” on page 79.

### Deploying the client application

When you have finished coding and building your web service client application, you are ready to deploy and test the application.

In our example, we have not modified the Axis configuration file axiscpp.conf. However, if we had modified it (e.g., we were using client-side handlers), we would need to ensure that the AXISCPP_DEPLOY environment variable points to the directory containing the /etc directory (the axiscpp.conf file would be located in the /etc directory), as described in “The axiscpp.conf file” on page 57.

The steps below use the example myGetQuote client application, and assume that a GetQuote service is running. (This service is with the samples supplied with WebSphere Application Server Version 5.0.2 or later). If you do not have the appropriate service, you must create the service code from the WSDL in the samples directory.

Once you have confirmed the above prerequisites, run and test the client application by completing the following steps.

1. Run the myGetQuote application.

2. Check that the myGetQuote application has returned the price of IBM shares in dollars.

   The example screen shot below shows the myGetQuote application run from the command line in which client-side handlers are not being used.

   ```
   > call MYGETQUOTE/MYGETQUOTE
   The stock quote for IBM is $94.33
   ```

   If we were had implemented client-side handlers, then we would have seen the following results:

   ```
   > call MYGETQUOTE/MYGETQUOTE
   Before the pivot point Handler can see the request message.
   Past the pivot point Handler can see the response message.
   The stock quote for IBM is $94.33
   ```
Chapter 11. Creating client-side handlers

This chapter describes how to develop client side handlers for use with your Web service client applications.

Client side handlers are optional. You only need to use handlers if you need to alter the SOAP header of a request SOAP message before the message is transmitted, or the SOAP header of a response SOAP message before the body of the message is deserialized. The important point to remember is that only the SOAP header can be changed inside a request or response handler.

As has been previously discussed, Web Services Client for ILE supports two basic types of handler: service handlers, which is specific to the Web service with which it is associated; and global handlers, which is called regardless of the Web service port or message name. From a coding perspective, there are no differences between service and global handlers. A service handler is associated with a particular service/port combination and is only invoked when a SOAP message with the appropriate destination has been called. A global handler is always invoked, regardless of the message destination.

To allow a handler to be used, you must create or amend the WSDD and axisconf.conf files to include the appropriate details, as described in “The Web services deployment descriptor (WSDD) file” on page 59 and “The axisconf.conf file” on page 57.

Handlers must conform to the rules listed below:

- One or more handlers can be called together for outgoing or incoming requests. They are called in the order in which they appear in the WSDD file.
- Must be written in C++ language.
- Each handler must be created as an individual shared library (i.e. service program).
- The handler must implement the BasicHandler interface, defined in <install_dir>/include/axis/BasicHandler.hpp header file.
- Each handler library must have the following export functions:
  int GetClassInstance(BasicHandler ** ppClassInstance);
  int DestroyInstance(BasicHandler * pClassInstance);
  The GetClassInstance() function returns an instance of the handler, while DestroyInstance() is used to destroy the instance of the handler.
- Handler names must be unique.
- Handlers can only modify the SOAP header part of the message.
- If the WSDL file that you are using does not specify SOAP actions, then these need to be added to your client application before calling the web service method, if you want the service handler to be invoked. The method that you use for doing this is setTransportProperty() (C++ Stub class) or axiscStubSetTransportProperty() (C function).

If the same handler is to be used on the request and response sides, take care to ensure that the handler is aware of its invoked context (i.e. pre-pivot and post-pivot).

To create client-side handlers, perform the following steps:

1. Implement client-side handler.
2. Create the WSDD file.

---

6. An overview of client-side handler concepts can be found at “Client-side handlers” on page 47.
7. There are also APIs that allow you to add SOAP headers, so handlers are not necessarily required if you want to add headers to a request SOAP message.

© Copyright IBM Corp. 2011, 2012
3. Create axisconf file and update file so that it points to WSDD file.

The following sections will discuss each of these steps. For illustrative purposes we will be using the sample code that is shipped as part of the product in the directory <install_dir>/samples/handlers. We will be using the following files:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>client.wsdd</td>
<td>The WSDD file that defines the client handler.</td>
</tr>
<tr>
<td>myClientHandler.cpp</td>
<td>Client handler implementation.</td>
</tr>
<tr>
<td>myClientHandler.hpp</td>
<td>Client handler implementation header file.</td>
</tr>
<tr>
<td>myClientHandlerFactory.cpp</td>
<td>Client handler factory implementation.</td>
</tr>
</tbody>
</table>

Source listings for the handler code can be found at Appendix B, “Code Listings for Client Handler,” on page 195.

### Implementing a client-side handler

To create a handler, you first create a client handler header file, a client handler file and client handler factory file. You can then use these files to build your handler library in the same way as you would any other library. The example files supplied with Web Services Client for ILE provide templates that you can use for guidance when you are creating your own handlers.

Since we will not be modifying the sample handler files, we just need to build the service program containing the handler as follows:

1. Create a library called HANDLERS using the CL command CRTLIB from the CL command line as follows:
   
   CRTLIB HANDLERS

2. Change the current working directory to the location of the sample handler files. Issue the following command from the CL command line:
   
   cd '/qibm/proddata/os/webservices/v1/client/samples/handlers'

3. Build the handler service program using the following CL commands:
   
   CRTCPPMOD MODULE(HANDLERS/mychfact) SRCSTMF('myClientHandlerFactory.cpp') INCDIR('/qibm/proddata/os/webservices/v1/client/include') ENUM(*INT)
   
   CRTCPPMOD MODULE(HANDLERS/mych) SRCSTMF('myClientHandler.cpp') INCDIR('/qibm/proddata/os/webservices/v1/client/include') ENUM(*INT)
   
   CRTSRVPGM SRVPGM(HANDLERS/MYCLH)MODULE(HANDLERS/MYCH HANDLERS/MYCHFACT) EXPORT(*ALL) BNDSRVPGM(QSYSDIR/QAXIS10C)

Next step is create a WSDD file, see “Creating a WSDD File.”

### Creating a WSDD File

The WSDD is used by the Axis engine to determine what handler is to be invoked and when is the handler to be invoked as the SOAP request and response messages are being processed. So we need to create a WSDD. The code below is an example of a WSDD file that has a service handler that will be called during the pre-pivot and post-pivot phases.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/
 xmlns:C="http://xml.apache.org/axis/wsdd/providers/c">
<!--Service Handler Definitions-->
<service name="Handler" provider="CPP:DOCUMENT" description="Handler">
<requestFlow>
```

80 Web Services Client for ILE: Programming Guide
The interpretation of the above file is as follows. We are telling the Axis engine that the handler that is located in /qsys.lib/HANDLERS.lib/MYCLH.srvgm should be invoked for the service that sets the SOAP action to Handler during the pre-pivot and post-pivot phases.

For a detailed description of each tag and the parts that the WSDD file may contain, see “The Web services deployment descriptor (WSDD) file” on page 59.

Finally, we need to point to the WSDD file in the axisconf file.

**Updating axisconf file to point to WSDD file**

If you are using client side handlers, you must add an additional line to the axisconf configuration file, defining the path to the WSDD file. The following example shows the axisconf file with WSDD information added.

```
# The comment character is '
# Available directives are as follows
#
# ClientWSDDFilePath: The path to the client WSDD
# SecureInfo: The GSKit security information
#
ClientWSDDFilePath:/getQuote/client.wsdd
```

The important line of the above example is the first line after the comments, the ClientWSDDFilePath definition. When this line appears in the axisconf file, Web Services Client for ILE uses this reference to determine which handlers are to be included in the SOAP request/response message parser.
Chapter 12. C++ programming considerations

This chapter covers programming considerations when you begin writing your applications to take advantage of Web services client for ILE C++ stub code.

C++ exception handling

Web Services Client for ILE uses exceptions to report back any errors that have occurred during the transmission of a SOAP message. This includes errors that are detected by the Axis engine or SOAP faults that are returned by the Web service.

In C++ applications, Web service stub methods that are invoked should be within a try block. How many catch clauses you have depends on how much detail you want and whether there are SOAP faults defined for the Web service operation that you want to handle separately. So let us take a look at an example. Below is a wsdl:portType definition called MathOps that has a div operation and has three SOAP faults defined - DivByZeroStruct, SpecialDetailStruct and OutOfBoundStruct:

```
...<wsdl:portType name="MathOps">
   <wsdl:operation name="div">
     <wsdl:input message="impl:divRequest" name="divRequest"/>
     <wsdl:output message="impl:divResponse" name="divResponse"/>
     <wsdl:fault message="impl:DivByZeroStruct" name="DivByZeroStruct"/>
     <wsdl:fault message="impl:SpecialDetailStruct" name="SpecialDetailStruct"/>
     <wsdl:fault message="impl:OutOfBoundStruct" name="OutOfBoundStruct"/>
   </wsdl:operation>
</wsdl:portType>
...
```

The definition of the SOAP fault messages is as follows:

```
<complexType name="OutOfBoundStruct">
   <sequence>
     <element name="varString" nillable="true" type="xsd:string"/>
     <element name="varInt" type="xsd:int"/>
     <element name="specialDetail" nillable="true" type="impl:SpecialDetailStruct"/>
   </sequence>
</complexType>
<complexType name="SpecialDetailStruct">
   <sequence>
     <element name="varString" nillable="true" type="xsd:string"/>
   </sequence>
</complexType>
<complexType name="DivByZeroStruct">
   <sequence>
     <element name="varString" nillable="true" type="xsd:string"/>
     <element name="varInt" type="xsd:int"/>
     <element name="varFloat" type="xsd:float"/>
   </sequence>
</complexType>
```

As has been previously discussed in "Mapping of wsdl:fault" on page 72, each SOAP fault defined in the WSDL is represented as a generated C++ class. For example, the SOAP fault DivByZeroStruct is represented by the following C++ class:

```cpp
class DivByZeroStruct : public SoapFaultException {
   public: xsd_string varString;
   xsd_int varInt;
);
```
An instance of this exception class is thrown by the fault handler inside the MathOps stub, if such a fault is returned by the server. If a SOAP fault that is not defined in the WSDL is returned, the fault handler will throw an instance of the Axis-defined OtherFaultException exception. If you look at the generated fault above and the OtherFaultException class you will find that both extend the SoapFaultException class. So, the client application may catch a specific SOAP fault or any SoapFaultException. In addition, the SoapFaultException extends AxisException, which the Axis engine throws when it detects errors in the processing of a SOAP message, such as when the endpoint URL of the server is invalid.

To sum it all up, a Web service client application can catch the different types of faults that may be thrown by the stub and decode the contents appropriately. The following example shows how a client application may catch and process exceptions:

```cpp
// Attempt to divide by zero.
try
{
    // Create the Web Service with an endpoint URL.
    MathOps ws( pszEndpoint);

    // Call the div method with two parameters.
    // This will attempt to divide 1 by 0.
    int iResult = ws.div( 1, 0);

    // Output the result of the division.
    cout << "Result is " << iResult << endl;
} catch(DivByZeroStruct& dbzs)
{
    // Catch a divide by zero fault
    // This is a user soap fault defined in the WSDL
    cout << "DivByZeroStruct Fault: " << dbzs.varString << "", "
    << dbzs.varInt << ", " << dbzs.varFloat << endl;
} catch(SpecialDetailStruct& sds)
{
    // Catch a special detail fault
    // This is a user soap fault defined in the WSDL
    cout << "SpecialDetailStruct Fault: "" << sds.varString << endl;
} catch(OutOfBoundStruct& oobs)
{
    // Catch an out of bounds fault
    // This is a user soap fault defined in the WSDL
    cout << "OutOfBoundStruct Fault: "" << oobs.varString << "", "
    << oobs.varInt << ", " << oobs.specialDetail->varString << endl;
} catch(SoapFaultException& sfe)
{
    // Catch any other SOAP faults
    cout << "SoapFaultException: " << sfe.getFaultCode() << " " << sfe.what() << endl;
} catch(AxisException& e)
{
    // Catch an AXIS exception
    cout << "AxisException: " << e.getExceptionCode() << " " << e.what() << endl;
} catch(exception& e)
{
    // Catch a general exception
    cout << "Unknown Exception: " << e.what() << endl;
}
```
C++ memory management

The WSDL specification provides a framework for how information is to be represented and conveyed from place to place. Web services client for ILE maps this framework to program-language specific data object, such as classes or structures. The data objects that are dynamically allocated from the storage heap must be deleted in order to avoid memory leaks. Information is represented by four generic types: simple types, arrays of simple type, complex types, and arrays of complex type. This section describes what you need to be aware of in order to avoid memory leaks.

Built-in simple types

There are more than 45 built-in simple types, which are defined in `<install_dir>/include/Axis/AxisUserAPI.hpp`. When a type is nillable or optional (that is, minOccurs="0"), it is defined as a pointer to a simple type.

The example below shows a typical simple type in a WSDL. The simple type used in this example is xsd:int, which is mapped to C++ type xsd__int. The extract from the WSDL has an element called addReturn of type integer. This element is used by the add operation, which uses the addResponse element to define the type of response expected when the add operation is called.

```xml
<element name="addResponse">
  <complexType>
    <sequence>
      <element name="addReturn" type="xsd:int"/>
    </sequence>
  </complexType>
</element>
```

Later in the WSDL, the addResponse element is the response part for the add method. This produces the following Web Services Client for ILE web services method prototype from the simple type in the WSDL:

```cpp
public:
  STORAGE_CLASS_INFO xsd__int add( ...);
```

Thus, the user generated application code for this example is as follows:

```cpp
xsd__int xsd_iReturn = ws.add( ...);
```

When a type is nillable, (that is, nillable="true"), optional (that is, minOccurs="0"), or a text type (such as xsd:string), it is defined as a pointer.

```xml
<element name="addResponse">
  <complexType>
    <sequence>
      <element name="addReturn" nillable="true" type="xsd:int"/>
    </sequence>
  </complexType>
</element>
```

This produces the following Web Services Client for ILE web services method prototype:

```cpp
public:
  STORAGE_CLASS_INFO xsd__int * add( ...);
```

The user generated application code produced by the nillable simple type in the WSDL is as follows:

```cpp
xsd__int * xsd_piReturn = ws.add( ...);
```

// Later in the code...
// Delete this pointer and set to NULL.
delete xsd_piReturn;
xsd_piReturn = NULL;

Note: The example above shows the deletion of the return value. Any pointer that Web Services Client
for ILE returns becomes the responsibility of the client application and does not go out of scope if the
web service is deleted. The user application must delete the pointer to the object type once it is no longer
required.

Arrays of simple type
Web services client for ILE provides array objects for each of the defined simple types. These are defined
in <install_dir>/include/Axis/AxisUserAPIArrays.hpp. An example of a simple array type is
xsd__int_Array.

The following example shows an extract from a WSDL that has two elements called simpleArrayRequest
and simpleArrayResponse of array type integer. These elements are used by the simpleArray operation,
which uses the simpleArrayRequest element to define the type of request and simpleArrayResponse
element to define the type of response expected when the simpleArray operation is called.
<xsd:element name="simpleArrayRequest">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="simpleTypeRes" type="xsd:int" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="simpleArrayResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="simpleTypeReq" type="xsd:int" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

Note that the maxOccurs attribute is used in this example. Web services client for ILE creates an array
object for any type that is declared as having maxOccurs greater than one. Later in the WSDL, the
simpleArrayRequest and simpleArrayResponse become the input and output parameters for the
simpleArray method whose prototype is shown below:
public: xsd__int_Array * simpleArray( xsd__int_Array * pValue);

The prototype requires input and output arrays to be created. To avoid memory leaks, these must be
created and managed properly. For information about the generation management and deletion of a
typical input and output array, see the following two subsections:
• “Array types as input parameters”
• “Array types as output parameters” on page 87

Array types as input parameters
The prototype method requires an input array to be created. This array must be created and managed
properly.

If an array is to be used as an input parameter, then it has to be created and filled.

The following example shows the typical usage of a nillable simple array type required by a generated
stub. The array is an example of the input array to the method. The example assumes that the array
contains three elements whose values are 0, 1 and 2 respectively.
// Need an input array of 3 elements.
int iArraySize = 3;

// Final object type to be passed as an input parameter to the web service.
xsd__int_Array iInputArray;

// Preparatory array that contains the values to be passed. Note that the
// array is an array of pointers of the required type.
xsd__int ** ppiPrepArray = new xsd__int*[iArraySize];

// Loop used to populate the preparatory array.
for( int iCount=0; iCount < iArraySize ; iCount++)
{
    // Each element in the array of type pointers is filled with a pointer to an
    // object of the required type. In this example we have chosen the value of
    // each element to be the same as the current count and so have passed this
    // value to the new instance of the pointer.
    ppiPrepArray[iCount] = new xsd__int( iCount);
}

// Set the contents of the final object to contain the elements of the
// preparatory array.
iInputArray.set( ppiPrepArray, iArraySize);

// Call the web service(s) that use the input array ...

// No longer require iInputArray. Delete the preparatory array held within.
for( int iCount = 0; iCount < iArraySize ; iCount++)
{
    // Delete each pointer in the pointer array.
    delete ppiPrepArray[iCount];
    ppiPrepArray[iCount] = NULL;
}

// Delete the array of pointers and then set the value to NULL so that it
// cannot be reused.
delete [] ppiPrepArray;
ppiPrepArray = NULL;

When the method returns, iInputType can be destroyed. If iInputType was created as a pointer
(piInputType), then the client user code must remember to delete it otherwise the code will have created
a memory leak.

**Array types as output parameters**
The prototype method requires an output array to be created. This array must be created and managed
properly.

Following on from the example in the example in "Array types as input parameters" on page 86, the following example shows the client application calling the simpleArray method on the web service and using the returned array. The following example shows a typical usage of the method produced by the WSDL example of an
array of nillable simple type. The response integer array is not directly accessible. To get the embedded
integer array, the user has to call the get method on the piSimpleResponseArray object as follows:

```cpp
xsd__int_Array * piSimpleResponseArray = ws.simpleArray( &iInputArray);
nisd__int * piIntArray;
```

```cpp
int iSize = 0; // Size of the array.
```

```cpp
// Pointer to a pointer that will contain the array. Get the contents
// of the response. The return value will be a pointer to a pointer containing
// the array and iSize will contain the number of elements in the array.
// If it is a const pointer so cannot be manipulated.
const xsd__int ** ppiIntArray = piSimpleResponseArray->get( (int&) iSize);
```

// Check if the array size greater than zero before processing it.
if( iSize > 0)
{
// For each element of the array...
for( int iCount = 0 ; iCount < iSize ; iCount++)
{

// Check that that element is not null before use...
if( ppiIntArray[iCount] != NULL)
{
    cout <<"Element[" << iCount << "]=" << *ppiIntArray[iCount] <<endl;
}
}

// Later in the code...
delete piSimpleResponseArray;
piSimpleResponseArray = NULL;

Notes:
1. The returned pointer is not NULL.
2. The user only needs to delete the object returned by the call to the web service. The client must not
   delete any object that is extracted from within this object. For example, in the previous code sample,
   ppiIntArray must not be deleted by the user as it will be deleted by the parent object
   (piSimpleArrayResponse) when that is deleted.
3. If the pointer to the array of pointers to integer values (ppiIntArray) is NULL, then this indicates an
   empty array. If this is the case, iSize is equal to zero.

Complex types and arrays of complex type

When complex types are used in a web service, the same rules as for simple types apply.

Complex types

The following example shows classes produced from WSDL with a complex type. As shown in this
example, complex types only take shallow copies of the data when using the set and get methods.

class STORAGE_CLASS_INFO ComplexType
{
    public:
        class xsd__string Message;
        class xsd__int MessageSize;

        xsd__string getMessage();
        void setMessage( xsd__string InValue);
        xsd__int getMessageSize();
        void setMessageSize( xsd__int InValue);

    The client has to remember that when using pointers to objects, only the pointer is copied and it is not
    cloned. For example, if a complex type contains a string, the client can set the contents of the string by
    creating a local string and then using the set method on the complex object to copy that string into the
    object.

    The following example shows restrictions that can be applied when using a complex type:
    
    xsd__int iStringLength = strlen( "Hello World");
    xsd__string myNewString = new char[iStringLength + 1];
    strcpy( myNewString, "Hello World");
myComplexType.setMessage(myNewString);
delete myNewString; // Do this and myComplexType.Message will be left pointing to
// invalid memory.

Alternatively:
delete myComplexType; // Do this and myNewMessage will be pointing to invalid
// memory.

The same rules as for simple types apply to the parameters of a complex type when used on a method
call. These rules are as follows:

- The client is responsible for generating the input parameter information and for deleting any objects
  created during this process.
- The client is responsible for deleting the output object returned by the method.
- If you have complex objects of the same type and you use the copy constructor, for example:
  ComplexType * myNewComplexType = new ComplexType(myExistingComplexType);
  then this takes a deep copy of all the member variables from the original object to populate the new
  object.

Arrays of complex type

If a WSDL describes a complex type being used within an array, the wsdl2ws.sh tool generates a
 corresponding array object using the complex name type suffixed with "_Array".

Deep copying

Web services client for ILE supports deep copying. Deep copying is where, when setting a value on a
 complex type, the set method makes a private copy of the original data. Subsequent modification or
deletion of the original data does not affect the complex type, and the application must delete the original
data to prevent memory leaks.

// This is an example of deep copying.
ComplexType * complexType = new ComplexType();
xsd_string aStringType = new char[9];
strcpy(aStringType, "Welcome!");
complexType->setaStringType(aStringType);
// Note: By default deep copying will take place.
delete [] aStringType;
// This object is no longer required by the generated objects so can be deleted
// at the earliest opportunity.

Result result = ws.useComplexType(complexType);
delete complexType;

// This is an example of explicitly deep copying.
ComplexType * complexType = new ComplexType();
xsd_string aStringType = new char[9];
strcpy(aStringType, "Welcome!");
complexType->setaStringType(aStringType, true);
// Note: Use of additional parameter set to 'true' indicates deep copying is to
// take place.
delete [] aStringType;
// This object is no longer required by the generated objects so can be deleted
// at the earliest opportunity.

Result result = ws.useComplexType(complexType);
delete complexType;
Note: Web services client for ILE does not support shallow copying, which is where, when setting a value on a complex type, the set method maintains a reference (or pointer) to the original data. The original data should not be modified and must not be deleted during the lifecycle of the complex type (that is, until the complex type is deleted). The application must delete the original data to prevent memory leaks.

```c++
// This is an example of shallow copying
ComplexType * complexType = new ComplexType();
xsd__string aStringType = new char[9];
strcpy(aStringType, "Welcome!");
complexType->setaStringType(aStringType, false);
// Note: Use of additional parameter set to 'false' indicates shallow copying is to take place.
```

Result result = ws.useComplexType(complexType);

delete complexType;
delete [] aStringType;
// This object MUST NOT be deleted until generated object has been deleted.

**Summary of rules**

There are a number of rules relating to memory management that you must follow when using the C++ stub code generated by the `wsdl2ws.sh` tool.

1. Objects that are passed to or obtained from the web service method as pointers are the responsibility of the client application.
2. Objects that are defined as a class hide the objects that they contain and instead have get and set methods to manipulate the object contents.
3. For objects that are classes and used as inputs, the client application is responsible for the deletion of these objects when they are no longer required.
4. For objects that are classes and used as outputs, the client application is responsible for the deletion of these objects when they are no longer required. They must not delete any object that is returned from a call to the ‘get’ method as this is deleted by the parent when the parent object is deleted.
5. If a stub is "new"ed (rather than being a stack object), it must be deleted.
6. Return parameters must be deleted when they are one of:
   - Complex type
   - Array
   - String based type rule (see rule 7)
   - Nullable
   - Optional
7. When deleting string based types, use: `delete [] string;`. The string based types are: `xsd__string`, `xsd__normalizedString`, `xsd__token`, `xsd__language`, `xsd__Name`, `xsd__NCName`, `xsd__ID`, `xsd__IDREF`, `xsd__IDREFS`, `xsd__ENTITY`, `xsd__ENTITIES`, `xsd__NMTOKEN`, `xsd__NMTOKEN`, `xsd__anyURI`, `xsd__QName` and `xsd__NOTATION`.
8. The "`set(xsd__unsignedByte * data, xsd__int size)`" method on `xsd__hexBinary` and `xsd__base64Binary` take a copy of the data. Remember to delete the original data.
9. When using the "`xsd__unsignedByte * get(xsd__int & size) const`" method on `xsd__hexBinary` and `xsd__base64Binary` do NOT delete the returned pointer, as this pointer is deleted by the destructor on the `xsd__base64Binary` or `xsd__hexBinary` object.
10. When setting members of complex types, the corresponding set method takes a deep copy of the original data. Remember to delete the original data.
11. If a complex type contains an `xsd__hexBinary` or an `xsd__base64Binary` element, which is also neither nullable nor optional, you must take care when using the generated get method on the complex type with the get method on the `xsd__hexBinary` or `xsd__base64Binary` object. You cannot use both in a single line of code, for example:
must be split into two lines of code:

```c
xsd__base64Binary binaryObject = myComplexType.getElement();
xsd__unsignedByte * data = binaryObject.get(size);
```

12. Setting members of complex types directly (that is, not using the corresponding set method) is not supported and may produce unknown side-effects.

13. When initializing an Array (Axis_Array and its derivates - xsd__<built-in simple type>_Array or <generated type>_Array) using the set() method takes a deep copy of the data. Remember to delete the original array elements and the original c-style pointer array.

---

**Securing web service communications in C++ stub code**

This section explains how to use Secure Sockets Layer (SSL) to set up security when using C++ stub code.

You can secure your HTTP messages by using SSL, which encrypts the request and response messages before they are transmitted over the wire.

**Note:** Handlers are not affected by SSL as they receive the message either before encryption or after decryption.

Any web service that uses SSL adds the suffix ‘s’ for secure to the http name in the URL. For example: http://some.url.com becomes https://some.url.com

A secure endpoint URL is an endpoint beginning with ‘https’. To allow secure endpoint URLs to be used, you must pass security information to the C++ stub. You can do this either by adding the required information to the “The axiscpp.conf file” on page 57 configuration file, or by configuring the settings for secure service using the “Stub::SetSecure()” on page 106 Stub class method.

---

**Cookies**

This section describes the cookie support that Web services client for ILE provides, including getting cookies from services and adding cookies to other services, and removing cookies from C++ stub instances.

**Cookie attributes**

The following table summarizes how Web services client for ILE handles cookie attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>expires</td>
<td>This attribute is ignored. If a server sends a signal to the client asking it to expire a cookie, the client does not do so. Once set by a server, the client continues to send cookies on each request using that stub. If a new stub instance is created and used, then the cookies from the original stub instance are not sent on requests from the new stub instance.</td>
</tr>
<tr>
<td>path</td>
<td>This attribute is ignored. Cookies are sent on all requests and not just on requests to a URI applicable to the path.</td>
</tr>
<tr>
<td>domain</td>
<td>This attribute is ignored. Cookies have affinity to a stub and are domain neutral.</td>
</tr>
<tr>
<td>secure</td>
<td>This attribute is ignored. If secure is set on a cookie, this has no effect and the cookie is sent on all future requests regardless of whether the channel is secure or not.</td>
</tr>
</tbody>
</table>
Use of cookies across multiple stub instances

If cookies are required in a different instance of a stub such as when a login is done on one service and the login session cookies are required on other services, you can use the APIs in the following example. This C++ example uses two instances of the calculator service and a login service. The first instance uses the login service and receives some cookies back representing the session cookies. These cookies are required for interacting with the calculator service in order to authenticate to the server that hosts the calculator service.

```cpp
// Call the webservice
LoginService loginService("http://loginserver/loginservice");

// must tell the service to save cookies
loginService.setMaintainSession(true);

// login so that we can get the session cookies back
loginService.login("myusername", "mypassword");

// Store the cookies so they can be given to the Calculator web service as
// authentication.
int currentCookieKey=0;
string cookieKeys[2];
const char* key = loginService.getFirstTransportPropertyKey();
string keyString(key);
if(key)
{
  // Only get the "Set-Cookie" transport properties - as these are
  // what the server sends to the client to set cookies.
  if(keyString.compare("Set-Cookie") == 0)
  {
    string valueString(loginService.getCurrentTransportPropertyValue());
    cookieKeys[currentCookieKey++] = valueString;
  }
}

// then the rest of the cookies
while(key = loginService.getNextTransportPropertyKey())
{
  string nextKeyString(key);
  // Only get the "Set-Cookie" transport properties - as these
  // are what the server sends to the client to set cookies.
  if(nextKeyString.compare("Set-Cookie") == 0)
  {
    string valueString(loginService.getCurrentTransportPropertyValue());
    cookieKeys[currentCookieKey++] = valueString;
  }
}

// Now we've logged in and stored the cookies we can create the calculator service,
// set the cookies on that stub instance and use the calculator.
Calculator calculator("http://calculatorservice/calculatorservice");
calculator.setMaintainSession(true);

// OK, Now add the previously saved session cookies on to this new service
// as this service does not pick up the cookies from the other stub.
calculator.setTransportProperty("Cookie",
  cookieKeys[currentCookieKey++].c_str());
}

// Now, when we use the service it will send the session cookies to the server
// in the http message header
// which allows the server to authenticate this instance of the service.
int result = calculator.add(1, 2);```

92  Web Services Client for ILE: Programming Guide
Manipulation of cookies using C++ AXIS APIs

It is sometimes necessary to remove cookies from stub instances.

- To delete a single cookie from a C++ stub instance:
  ```c++
  service.deleteTransportProperty(cookiename);
  ```
  For example:
  ```c++
  calculator.deleteTransportProperty("loginCookie");
  ```
- To delete all cookies from a C++ stub instance:
  ```c++
  service.deleteTransportProperty("Cookie");
  ```
  Note the capital 'C' in "Cookie".
  ```c++
  calculator.deleteTransportProperty("Cookie");
  ```

Floating point numbers in C++ types

This section provides reference information about using floating point numbers with Web services client for ILE.

The XML specification refers to the IEEE specification for floating point numbers. The specification lists that float and double have the following precision:

- Float type numbers, 1 sign bit, 23 mantissa bits and 8 exponent bits.
- Double type numbers, 1 sign bit, 52 mantissa bits and 11 exponent bits.

For float, with a mantissa able to represent any number in the range $1 > x > 1/2^{23}$, this gives a minimum accuracy of 6 digits. Similarly, for double, with a mantissa able to represent any number in the range $1 > x > 1/2^{52}$, this gives a minimum accuracy of 10 digits.

When displaying floating point numbers, you must ensure that any potential inaccuracies due to rounding errors, and so on are not visible. Therefore, to ensure the correct level of precision, for float types, instead of using:

```c++
printf( "%f", myFloat);
```

you must use the following formatting command:

```c++
printf( "%f.6g", myFloat);
```

Similarly, to ensure the correct level of precision for double types, instead of using:

```c++
printf( "%f", myDouble);
```

you must use the following formatting command:

```c++
printf( "%f.10g", myDouble);
```
Chapter 13. Troubleshooting C++ client stubs

This chapter is intended to help you learn how to detect, debug, and resolve possible problems that you may encounter when generating or using C++ stub code.

**C++ stub code generation problems**

When you use the `wsdl2ws.sh` tool to generate C++ stub code, the tool will generate an exception for any error that is encountered. Typical errors include the inability for the tool to resolve to an XSD file used in the specified WSDL file or a syntactically incorrect WSDL file. You will need to correct the problem and try running the tool again.

**C++ stub code compile problems**

If there is a compile problem in C++ stub code, the most likely cause of the problem is the use of an unsupported construct. The `wsdl2ws.sh` tool will not always generate an exception when used against a WSDL file that contains an unsupported WSDL construct. The problem may manifest itself when compiling the generated stub code. To see what is supported by the tool, see “Supported specifications and standards” on page 43.

**C++ stub code runtime problems**

Invoking a Web service operation may result in the Web service returning a SOAP fault as a response. There can be many reasons for this, and the only sure way to determine where the problem lies is by examining the generated SOAP request and resulting response.

The Web services client for ILE client engine has a tracing capability that traces the request and response messages. To learn about the tracing support in Axis, see the `Axis::startTrace()` on page 98 Axis C++ class.

© Copyright IBM Corp. 2011, 2012
Chapter 14. Axis C++ core APIs

This chapter summarizes the core (i.e. most commonly used) Axis C++ classes and methods. For a complete list of the Axis classes and associated methods, view the following file in a Web browser: /QIBM/ProdData/OS/WebServices/V1/client/docs/api/index.html.

Axis class

Contains methods that affect the Axis client engine, such as methods to initialize and terminate the Axis runtime, and methods to free allocated memory resources. The Axis C++ class is defined in include file <install_dir>/include/axis/Axis.hpp.

The following table lists the Axis class methods.

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis::initialize()</td>
<td>Initializes the Axis runtime.</td>
</tr>
<tr>
<td>Axis::terminate()</td>
<td>Terminates the Axis runtime.</td>
</tr>
<tr>
<td>Axis::AxisDelete()</td>
<td>Deletes storage allocated by the Axis engine.</td>
</tr>
<tr>
<td>Axis::startTrace()</td>
<td>Starts Axis logging.</td>
</tr>
<tr>
<td>Axis::stopTrace()</td>
<td>Stops Axis logging.</td>
</tr>
<tr>
<td>Axis::writeTrace()</td>
<td>Writes trace data to Axis log.</td>
</tr>
</tbody>
</table>

Axis::initialize()

```cpp
static void initialize(bool bIsServer)
```

Initializes the Axis runtime. Creating a stub also initializes the Axis runtime and deleting the stub terminates it. So simple applications that only ever use one stub at a time do not need to call these methods. More complicated applications that initialize multiple stubs, use them and delete them later, should initialize Axis at the start of their application using Axis::initialize() and terminate Axis at the very end of their application with Axis::terminate(). Applications that use Axis in multiple threads should also call Axis::initialize() and Axis::terminate().

Parameters

- bIsServer: Boolean flag that must be set to false.

Example

The following example initializes the Axis client engine.

```cpp
Axis::initialize(false);
```

Axis::terminate()

```cpp
static void terminate()
```

Terminates the Axis runtime.
Example

The following example terminates the Axis client engine.

```
Axis::terminate();
```

**Axis::AxisDelete()**

```
static void AxisDelete(void* pValue,
                      XSDTYPE type)
```

Deletes storage allocated by the Axis engine.

**Parameters**

- **pValue** Pointer to storage that is to be deleted.
- **type** The type of storage to be deleted. The XSDTYPE type is an enumerator defined `<install_dir>/include/axis/TypeMapping.hpp`.

Example

The following example deletes a pointer that was dynamically allocated by the Axis engine and that is used to store data with a type of `xsd:int`.

```
Axis::AxisDelete(ptr, XSD_INT);
```

**Axis::startTrace()**

```
static int startTrace(const char* logFilePath,
                      const char *logFilter=NULL)
```

Starts Axis logging. This must be done prior to any activity in order to propagate logging attributes to parser and transport. If there are active transports and parsers, you will not get trace records other than those associated with the engine and newly instantiated transports and parsers.

A typical trace record will look like the following (following are entry/exit trace records):

```
```

A trace record includes a timestamp, a thread ID, the component that is doing the trace, a one character field indicating Trace type:

- > (entry)
- < (exit)
- X (exception)
- D (debug)

and the method/function name. After which there will be additional trace data. When tracing is enabled, you will know exactly where an exception is being thrown from. A typical trace record for when an exception is thrown is as follows:

```
Line=1851: File=/home/amra/axis/L1.1.0/src/ws-axis/c/src/transport/axis3/HTTPTransport.cpp:
HTTPTransportException - SERVER_TRANSPORT_HTTP_EXCEPTION:
Server sent HTTP error: 'Not Found'
```

Request and response messages can be traced by enabling transport trace. Here is an example of a transport trace:
POST /axis HTTP/1.1
Host: 127.0.0.1:13260
Content-Type: text/xml; charset=UTF-8
SOAPAction: ""
Content-Length: 393

<?xml version='1.0' encoding='utf-8' ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Body>

<ns1:div xmlns:ns1="http://soapinterop.org/wsdl">
<ns1:arg_0_0>10</ns1:arg_0_0>
<ns1:arg_1_0>5</ns1:arg_1_0>
</ns1:div>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

Parameters

logFilePath  Pointer to null-terminated character string representing the path to where trace records are written to.
logFilter    Pointer to null-terminated character string representing the trace filter. The string filter is a semicolon delimitied string of possible filters. Possible filters include:

stub        - show trace records generated by stubs
engine      - show trace records generated by engine
parser      - show trace records generated by parser
transport   - show trace records generated by transport
noEntryExit - do not show entry/exit trace records

The default filter is "stub;engine;parser;transport". Specifying a null pointer or a null string is equivalent to requesting the default filter.

Returns

Zero if the method call is successful; otherwise -1 is returned.

Example

See example for the "Axis::writeTrace()" on page 100 method.
**Axis::stopTrace()**

```c
static void stopTrace()
```

Stops Axis logging. This should be done as the last step when everything has been cleaned up. Otherwise, active transports and parsers will continue to trace.

**Example**

See example for the "Axis::writeTrace()" method.

**Axis::writeTrace()**

```c
static void writeTrace(Axis_TRACE_TYPE type,
                       const char* funcName,
                       const char * format,
                       ...
```

Writes specified data to the Axis log file.

**Parameters**

- **type**
  The trace type. AXIS_TRACE_TYPE is an enumerator that can be set to one of the following values:
  - AXIS_TRACE_TYPE_ENTRY=0
  - AXIS_TRACE_TYPE_EXIT=1
  - AXIS_TRACE_TYPE_EXCEPTION=2
  - AXIS_TRACE_TYPE_DEBUG=3

- **funcName**
  Pointer to null-terminated character string representing class method or function for which trace record is being written.

- **format**
  Pointer to null-terminated character string representing the format as defined for the `printf()` function.

- **...**
  Variable number of parameters, the number of which is dependent on the specified `format` parameter.

**Example**

The following example writes a application-defined trace record to the Axis log.

```c
#include "axis/Axis.hpp"
#include "StockQuote.hpp"
#include <iostream>

int main()
{
    Axis::startTrace("/tmp/axis.log");
    Axis::writeTrace(Axis::AXIS_TRACE_TYPE_DEBUG,
                     "main-stockQuote", "start %d\n", 1);
    try
    {
        // GetQuoteService web service.
        char * pszEndpoint = "http://localhost:40001/StockQuote/services/urn:xmltoday-delayed-quotes";
        StockQuote * pwsStockQuote = new StockQuote( pszEndpoint);
        // Call the 'getQuote' method to find the quoted stock price
        char * pszStockName = "XXX";
        xsd__float fQuoteDollars = pwsStockQuote->getQuote( pszStockName);
        // Output the quote.
    }
} // try
```

100  Web Services Client for ILE: Programming Guide
if( fQuoteDollars != -1)
{
    cout << "The stock quote for " << pszStockName << " is $" << fQuoteDollars << endl;
} else
{
    cout << "There is no stock quote for " << pszStockName << endl;
}

// Delete the web service.
delete pwsStockQuote;

} catch( SoapFaultException& sfe)
{
    // Catch any other SOAP faults
    cout << "SoapFaultException: " << sfe.getFaultCode() << " " << sfe.what() << endl;
} catch( AxisException& e)
{
    // Catch an AXIS exception
    cout << "AxisException: " << e.getExceptionCode() << " " << e.what() << endl;
}

Axis::stopTrace();

// Exit.
return 0;

---

**Stub class**

This is the client base class to be inherited by all stub classes generated by wsd12ws.sh tool. This class acts as the interface between the client application and the Axis engine. The Stub C++ class is defined in include file `<install_dir>/include/axis/client/Stub.hpp`.

The following table lists the most commonly used methods.

*Table 13. Stub class methods*

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stub::setTransportProperty()</td>
<td>Sets transport properties (e.g. HTTP headers).</td>
</tr>
<tr>
<td>Stub::getTransportProperty()</td>
<td>Gets transport properties (e.g. HTTP headers).</td>
</tr>
<tr>
<td>Stub::setTransportTimeout()</td>
<td>Sets the transport timeout.</td>
</tr>
<tr>
<td>Stub::createSOAPHeaderBlock()</td>
<td>Creates and adds a SOAP header block to the stub.</td>
</tr>
<tr>
<td>Stub::setMaintainSession()</td>
<td>Sets whether to maintain session with service or not.</td>
</tr>
<tr>
<td>Stub::setPassword()</td>
<td>Sets the password to be used for basic authentication.</td>
</tr>
<tr>
<td>Stub::setUsername()</td>
<td>Sets the user name to be used for basic authentication.</td>
</tr>
<tr>
<td>Stub::setProxy()</td>
<td>Sets the proxy server and port for transport.</td>
</tr>
<tr>
<td>Stub::setProxyPassword()</td>
<td>Sets the password to be used for proxy authentication.</td>
</tr>
<tr>
<td>Stub::setProxyUsername()</td>
<td>Sets the user name to be used for proxy authentication.</td>
</tr>
<tr>
<td>Stub::SetSecure()</td>
<td>Sets SSL configuration properties.</td>
</tr>
</tbody>
</table>
**Stub::setTransportProperty()**

```c
void setTransportProperty(const char * pcKey,
                         const char * pcValue)
```

Sets the specified transport property. Calling this function with the same key multiple times will result in the property being set to the last value.

**Parameters**

- **pcKey**: Pointer to null-terminated character string representing the transport property to set.
- **pcValue**: Pointer to null-terminated character string representing the value of the transport property corresponding to pcKey.

**Example**

The following example sets the cookie HTTP header.

```c
stub.setTransportProperty("Cookie", "sessiontoken=123456");
```

**Stub::getTransportProperty()**

```c
const char * getTransportProperty(const char * pcKey,
                                   bool response = true)
```

Searches for the transport property with the specified key. The method returns NULL if the property is not found.

**Parameters**

- **pcKey**: Pointer to null-terminated character string representing the transport property to retrieve.
- **response**: Boolean flag, when set to true, searches the response message for the property; and when set to false searches the request message.

**Returns**

The value of the property or NULL if it was not found.

**Example**

The following example retrieves the HTTP cookie header from the response message.

```c
const char *cookie = stub.getTransportProperty("Cookie", true);
```

**Stub::setTransportTimeout()**

```c
void setTransportTimeout(long iTimeout)
```

Sets a specified timeout value, in seconds, to be used when waiting for a response from the Web service. If the timeout expires before receiving a Web service response, an Axis exception is thrown. A timeout of zero, which is the default, is interpreted as an infinite timeout.

**Parameters**

- **iTimeout**: An integer that specifies the receive timeout value in seconds.
Example

The following example set the transport timeout to 10 seconds.
stub.setTransportTimeout(10);

**Stub::createSOAPHeaderBlock()**

| IHeaderBlock * createSOAPHeaderBlock(AxisChar * pElemName, |
| AxisChar * pNamespace, |
| AxisChar * pPrefix) |

Creates and adds a SOAP header block (i.e. SOAP header). The returned `IHeaderBlock` pointer must be used to add the elements and values of the SOAP header block.

**Parameters**

- `pElemName` Pointer to null-terminated character string representing the element tag name of the SOAP header.
- `pNamespace` Pointer to null-terminated character string representing the URI of namespace.
- `pPrefix` Pointer to null-terminated character string representing the prefix that will be associated with the specified namespace.

**Returns**

Pointer to SOAP header block object. The ownership of the memory allocated for the object is owned by the stub.

**Example**

The following example will generate the following SOAP header:

```xml
<wsse:Security
  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
  SOAP-ENV:mustUnderstand="1"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <wsse:UsernameToken wsu:Id="UsernameToken-12345678">
    <wsse:UserName>admin</wsse:UserName>
    <wsse:Password
      Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-password-token-profile-1.0#PasswordText">
      admin
    </wsse:Password>
  </wsse:UsernameToken>
</wsse:Security>
```

Here is the example:

```c++
#include "axis/Axis.hpp"
#include "axis/IHeaderBlock.hpp"
#include "axis/BasicNode.hpp"
#include "StockQuote.hpp"
#include <stdio.h>

int main()
{
    
```
Stub::setMaintainSession()

void setMaintainSession(bool bSession)

Sets whether to maintain session with service or not.

Parameters
bSession  Boolean flag, when set to true, indicates that session should be maintained with Web service. When set to false the session will not be maintained.

Example

Following example indicates to the Axis engine that session to Web service should be maintained.

```c++
stub.setMaintainSession(true);
```

**Stub::setPassword()**

```c++
void setPassword(const char * pPassword)
```

Sets the password to be used for HTTP basic authentication.

**Parameters**

pPassword  Pointer to null-terminated character string representing the password.

Example

Following example sets HTTP basic authentication password.

```c++
stub.setPassword("password1");
```

**Stub::setUsername()**

```c++
void setUsername(const char * pUsername)
```

Sets the username to be used for HTTP basic authentication.

**Parameters**

pUsername  Pointer to null-terminated character string representing the username.

Example

Following example sets HTTP basic authentication username.

```c++
stub.setUsername("user1");
```

**Stub::setProxy()**

```c++
void setProxy(const char * pcProxyHost, unsigned int uiProxyPort)
```

Sets the proxy server and port.

**Parameters**

pcProxyHost  Pointer to null-terminated character string representing the host name of proxy server.

uiProxyPort  The port the proxy server listening on.
Following example sets proxy host and port information.
stub.setProxy("proxyserver", 40001);

**Stub::setProxyPassword()**

```c
void setProxyPassword(const char * pPassword)
```

Sets password to be used for proxy authentication.

**Parameters**

- `pPassword` Pointer to null-terminated character string representing the password.

**Example**

Following example sets password for proxy authentication.
stub.setProxyPassword("proxypwd1");

**Stub::setProxyUsername()**

```c
void setProxyUsername(const char * pUsername)
```

Sets the username to be used for Proxy authentication.

**Parameters**

- `pUsername` Pointer to null-terminated character string representing the username.

**Example**

Following example sets username for proxy authentication.
stub.setProxyUsername("proxyusr1");

**Stub::SetSecure()**

```c
void SetSecure(char * pKeyRingFile, ...
```

Sets SSL configuration properties.

**Parameters**

- `pKeyRingFile` Pointer to null-terminated character string representing the certificate store file to be used for the secure session or SSL environment.
- `pKeyRingPS` (optional) Pointer to null-terminated character string representing the password for the certificate store file to be used for the secure session or SSL environment. If the parameter is not passed or is set to the null string, the internal stash file associated with the user profile that is being used to run the application is used. To specify any of the subsequent optional parameters, you must pass a value for this parameter.
pKeyRingLbl (optional) Pointer to null-terminated character string representing the certificate label associated with
the certificate in the certificate store to be used for the secure session or SSL environment. If the
parameter is not passed or is set to the null string, the default certificate label in the specified
certificate store file is used for the SSL environment. To specify any of the subsequent optional
parameters, you must pass a value for this parameter.

pV2Cipher (optional) Pointer to null-terminated character string representing the list of SSL Version 2 ciphers to
be used for the secure session or the SSL environment. Specifying "NONE" for this parameter will
disable SSL Version 2 ciphers. To specify any of the subsequent optional parameters, you must pass a
value for this parameter.

pV3Cipher (optional) Pointer to null-terminated character string representing the list of SSL Version 3/TLS
Version 1 ciphers to be used for the secure session or the SSL environment. Specifying "NONE" for this
parameter will disable SSL Version 3 ciphers. To specify any of the subsequent optional parameters,
you must pass a value for this parameter.

pTLSCipher (optional) Pointer to null-terminated character string indicating whether to enable or disable the TLS
Version 1 ciphers. A value of "NONE" will disable the ciphers; any other value will enable the ciphers.
By default, the TLS Version 1 ciphers are enabled.

Usage notes
1. The Web services client for ILE supports secure sessions by using the Global Secure ToolKit (GSKit)
APIs. For the latest information on ciphers, see the gsk_attribute_set_buffer() API usage notes
2. The following GSK_V3_CIPHER_SPECS values are the SSL Version 3 ciphers and the TLS Version 1
ciphers supported:
   01 = *RSA_NULL_MD5
   02 = *RSA_NULL_SHA
   03 = *RSA_EXPORT_RC4_40_MD5
   04 = *RSA_RC4_128_MD5
   05 = *RSA_RC4_128_SHA
   06 = *RSA_EXPORT_RC2_CBC_40_MD5
   09 = *RSA_DES_CBC_SHA
   0A = *RSA_3DES_EDE_CBC_SHA
   2F = *RSA_AES_128_CBC_SHA (TLS Version 1 only)
   35 = *RSA_AES_256_CBC_SHA (TLS Version 1 only)
3. SSL Version 2 support is disabled IBM i 6.1 and later releases when the operating system is installed
resulting in no SSL Version 2 ciphers being supported. If SSL Version 2 is enabled (not recommended),
the following GSK_V2_CIPHER_SPECS values are the SSL Version 2 ciphers that would be supported
if shipped supported cipher list has not been altered.
   1 = *RSA_RC4_128_MD5
   2 = *RSA_EXPORT_RC4_40_MD5
   4 = *RSA_EXPORT_RC2_CBC_40_MD5
The following GSK_V2_CIPHER_SPECS values are the SSL Version 2 ciphers potentially supported if
an administrator later enables SSL Version 2:
   1 = *RSA_RC4_128_MD5
   2 = *RSA_EXPORT_RC4_40_MD5
   3 = *RSA_RC2_CBC_128_MD5
   4 = *RSA_EXPORT_RC2_CBC_40_MD5
   6 = *RSA_DES_CBC_MD5
   7 = *RSA_3DES_EDE_CBC_MD5

Example

The following example shows a sample client application that configures security information before
calling a web service. To configure the secure setting within your own application, add the code shown in
bold in this example.

```c
int main()
{
    // URL for secure communication. The localhost may require
```
// a port number, i.e. localhost:80
char * pszSecureURL = "https://localhost/Test/services/TestPort;

// Load instances of the service with secure URL settings.
ITestService * serviceSecure = new ITestService( pszSecureURL);
// Initialise the secure settings for the secure service.
serviceSecure->SetSecure("<Path to KeyRing.kbd>",
            "<stash password or NULL string>",
            "<label>", "NONE", "05", "NONE");

// Remainder of application


// End of application
delete serviceSecure;
return 0;
}

Call class

All stubs generated by the wsdl2ws.sh tool inherits from the Stub C++ class. The Stub C++ class contains
a pointer to an object instantiated from the Call C++ class, which is the actual interface to the Axis
engine. The Call C++ class is defined in include file <install_dir>/include/axis/client/Call.hpp. In
general, the only time you need to access this class is to invoke the setTransportProperty() method in
order to set the socket connect timeout value or to enable HTTP redirection.

The following table lists the most commonly used methods.

Table 14. Call class methods

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stub::setTransportProperty()</td>
<td>Sets transport properties.</td>
</tr>
</tbody>
</table>

Call::setTransportProperty()

void setTransportProperty(AXIS_TRANSPORT_INFORMATION_TYPE type,
                          const char * value)

Sets the specified transport property.

Parameters

type  Enumerator indicating what transport property to set. The information types are defined in
       <install_dir>/include/axis/GDefine.hpp. The relevant values are as follows:

ENABLE_AUTOMATIC_REDIRECT
Sets whether the transport is to automatically handle HTTP redirects. By default, redirects
are not handled by the transport. If enabled, auto-redirect will only occur when going from
http to http or https to https.

MAX_AUTOMATIC_REDIRECT
Sets how many redirects to follow if automatic redirection is enabled.

CONNECT_TIMEOUT
Sets a specified timeout value, in seconds, to be used when attempting to connect to the
server hosting the Web service. If the timeout expires before establishing a connection to the
server, an Axis exception is thrown.
value Pointer to null-terminated character string representing the value of the transport information to be set. The possible value is dependent on what is specified for the type:

- If the type is ENABLE_AUTOMATIC_REDIRECT, then the possible values are "true" to enable automatic redirection, or "false" to disable automatic redirection. By default automatic redirection is disabled.
- If the type is MAX_AUTOMATIC_REDIRECT, then the value represents an integer that indicates how many redirects to follow. The default is "1". A value less than "1" is the same as setting value to "0".
- If the type is CONNECT_TIMEOUT, then the value represents an integer that indicates the connect timeout value in seconds. A value of "0", which is the default, is interpreted as an infinite timeout.

Examples

The following example enables redirection up to a maximum of 5 redirections.
Call *call = stub.getCall();
call->setTransportProperty(ENABLE_AUTOMATIC_REDIRECT, "true");
call->setTransportProperty(MAX_AUTOMATIC_REDIRECT, "5");

The following example sets the connect timeout value to 5 seconds.
Call *call = stub.getCall();
call->setTransportProperty(CONNECT_TIMEOUT, "5");

IHeaderBlock class

Interface class that is inherited by the SOAP header block object. The IHeaderBlock C++ class is defined in include file <install_dir>/include/axis/IHeaderBlock.hpp.

The following table lists the most commonly used methods.

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IHeaderBlock::createNamespaceDecl()</td>
<td>Creates an attribute and adds it to the header block as a namespace.</td>
</tr>
<tr>
<td>IHeaderBlock::createStdAttribute()</td>
<td>Creates a standard header block attribute.</td>
</tr>
<tr>
<td>IHeaderBlock::createChild()</td>
<td>Creates a child node depending on the given node type.</td>
</tr>
<tr>
<td>IHeaderBlock::addChild()</td>
<td>Adds a child node to the header block.</td>
</tr>
</tbody>
</table>

IHeaderBlock::createNamespaceDecl()

INamespace * createNamespaceDecl(const AxisChar *pPrefix, const AxisChar *pNamespace)

Creates an attribute and adds it to the header block as a namespace.

Parameters

pPrefix Pointer to null-terminated character string representing the prefix that will be associated with the specified namespace.
pNamespace Pointer to null-terminated character string representing the URI of namespace.

Returns

Pointer to namespace object. The ownership of the memory allocated for the object is owned by the stub.
**IHeaderBlock::createStdAttribute()**

```cpp
IAttribute* createStdAttribute(HEADER_BLOCK_STD_ATTR_TYPE eAttribute,
                               SOAP_VERSION eSOAPVers)
```

Creates and adds a standard SOAP header block attribute.

**Parameters**

- `eAttribute` Enumerator indicating which of the following attributes are to be set:
  - ACTOR: Creates actor attribute to point to next.
  - MUST_UNDERSTAND_TRUE: Creates the mustUnderstand attribute set to "1".
  - MUST_UNDERSTAND_FALSE: Creates the mustUnderstand attribute set to "0".

- `eSOAPVers` Enumerator indicating the SOAP version. This parameter must always be set to:
  - SOAP_VER_1_1: SOAP version 1.1.

The enumerator SOAP_VERSION is defined in `<install_dir>/include/axis/SoapEnvVersions.hpp`.

**Returns**

Pointer to attribute object. The ownership of the memory allocated for the object is owned by the stub.

**Example**

See example for "Stub::createSOAPHeaderBlock()" on page 103.

**IHeaderBlock::createChild()**

```cpp
BasicNode* createChild(NODE_TYPE eNodeType,
                       AxisChar *pElemName,
                       AxisChar *pPrefix,
                       AxisChar *pNamespace,
                       AxisChar* pachValue)
```

Creates an instance of a basic node of the specified type.

**Parameters**

- `eNodeType` Enumerator indicating one of the following node types:
  - ELEMENT_NODE=1
  - CHARACTER_NODE=2

  The enumerator NODE_TYPE is defined in `<install_dir>/include/axis/BasicNode.hpp`.

- `pElemName` Pointer to null-terminated character string representing the element tag name of the node. This parameter is ignored for CHARACTER_NODE node types.

- `pPrefix` Pointer to null-terminated character string representing the prefix that will be associated with the specified namespace. This parameter is ignored for CHARACTER_NODE node types.

- `pNamespace` Pointer to null-terminated character string representing the URI of namespace. This parameter is ignored for CHARACTER_NODE node types.

- `pachValue` Pointer to null-terminated character string representing the value of the node. This parameter is ignored for ELEMENT_NODE node types.
Returns

Pointer to a basic node object. The ownership of the memory allocated for the object is owned by the caller until the node is added to the header block.

Example

See example for "Stub::createSOAPHeaderBlock()" on page 103.

IHeaderBlock::addChild()

```c
int addChild(BasicNode* pBasicNode)
```

Adds a child node to the SOAP header block.

Parameters

pBasicNode Pointer to basic node object to be added to SOAP header block.

Returns

Zero if node was added successfully; otherwise -1 is returned.

Example

See example for "Stub::createSOAPHeaderBlock()" on page 103.

BasicNode class

Interface class that is inherited by a basic node object. The BasicNode C++ class is defined in include file <install_dir>/include/axis/BasicNode.hpp.

The following table lists the most commonly used methods.

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicNode::createAttribute()</td>
<td>Creates an attribute and adds it to this basic node.</td>
</tr>
<tr>
<td>BasicNode::addChild()</td>
<td>Adds a child node to the basic node.</td>
</tr>
</tbody>
</table>

BasicNode::createAttribute()

```c
IAttribute* createAttribute(const AxisChar* pAttrName,
                           const AxisChar* pPrefix,
                           const AxisChar* pNamespace,
                           const AxisChar* pValue)
```

Creates an attribute and adds it to the basic node.

Parameters

pAttrName Pointer to null-terminated character string representing the attribute name.
pPrefix Pointer to null-terminated character string representing the attribute prefix that will be associated with the specified namespace.
pNamespace  Pointer to null-terminated character string representing the URI of attribute namespace.
pValue  The value of the attribute.

Returns

Pointer to created attribute object. The ownership of the memory allocated for the object is owned by the stub.

Example

See example for “Stub::createSOAPHeaderBlock()” on page 103.

**BasicNode::addChild()**

```c
int addChild(BasicNode * pBasicNode)
```

Adds a basic node as a child node to another basic node.

Parameters

pBasicNode  Pointer to basic node to be added as a child node.

Returns

Zero if node was added successfully; otherwise, -1 is returned.

Example

See example for “Stub::createSOAPHeaderBlock()” on page 103.
Part 4. Using C stubs

This part of the document provides details regarding all things related C stub programming. If you have no interest in C stub programming, you should skip this part of the document.

- Chapter 15, “WSDL and XML to C mappings,” on page 115 describes in detail how WSDL and XML constructs are mapped to C constructs.
- Chapter 16, “Developing a Web services client application using C stubs,” on page 123 is a step-by-step description on how to develop a client application using a C stub.
- Chapter 17, “C stub programming considerations,” on page 127 discusses various programming topics relating to using the C stubs, including how to set up an SSL connection.
- Chapter 18, “Troubleshooting C client stubs,” on page 137 describes how to troubleshoot command line tool and runtime problems.
- Chapter 19, “Axis C core APIs,” on page 139 Summarizes the core (i.e. most commonly used) Axis C functions.
Chapter 15. WSDL and XML to C mappings

The \texttt{wsdl2ws.sh} command tool can generate C stub code. This chapter will describe the mappings from WSDL and XML Schema types to C language constructs.

Mapping XML names to C identifiers

XML names are much richer than C identifiers. They can include characters that are either reserved or not permitted in C identifiers. The \texttt{wsdl2ws.sh} command generates unique and valid names for C identifiers from the schema element names using the following rules:

1. Invalid characters are replaced by underscore (\_). Invalid characters include the following characters:
   
   \begin{verbatim}
   / ! * $ % & ' ( ) + , - . ; < = > ? \ ^ ` { | } ~ [ ]
   \end{verbatim}

2. Names that conflict with C keywords will have an underscore inserted at the beginning of the name. For example, an XML element name of register will be generated as a C identifier of \_register.

3. If a name that is used as a C identifier conflicts with a structure with the same name, the identifier will have _Ref appended to the name.

XML schema to C type mapping

Table 17 specifies the C mapping for each built-in simple. The table shows the XML Schema type and the corresponding the Axis type (column 2), which generally is a typedef to a C language built-in type (column 3).

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Axis Type</th>
<th>Actual C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xsd:byte</td>
<td>xsdc__byte</td>
<td>signed char</td>
</tr>
<tr>
<td>xsd:decimal</td>
<td>xsdc__decimal</td>
<td>double</td>
</tr>
<tr>
<td>xsd:double</td>
<td>xsdc__double</td>
<td>double</td>
</tr>
<tr>
<td>xsd:float</td>
<td>xsdc__float</td>
<td>float</td>
</tr>
<tr>
<td>xsd:int</td>
<td>xsdc__int</td>
<td>int</td>
</tr>
<tr>
<td>xsd:integer</td>
<td>xsdc__integer</td>
<td>long long</td>
</tr>
<tr>
<td>xsd:long</td>
<td>xsdc__long</td>
<td>long long</td>
</tr>
<tr>
<td>xsd:negativeInteger</td>
<td>xsdc__negativeInteger</td>
<td>long long</td>
</tr>
<tr>
<td>xsd:nonPositiveInteger</td>
<td>xsdc__nonPositiveInteger</td>
<td>long long</td>
</tr>
<tr>
<td>xsd:nonNegativeInteger</td>
<td>xsdc__nonNegativeInteger</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>xsd:positiveInteger</td>
<td>xsdc__positiveInteger</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>xsd:unsignedByte</td>
<td>xsdc__unsignedByte</td>
<td>unsigned char</td>
</tr>
<tr>
<td>xsd:unsignedInt</td>
<td>xsdc__unsignedInt</td>
<td>unsigned int</td>
</tr>
<tr>
<td>xsd:unsignedLong</td>
<td>xsdc__unsignedLong</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>xsd:unsignedShort</td>
<td>xsdc__unsignedShort</td>
<td>unsigned short</td>
</tr>
<tr>
<td>xsd:short</td>
<td>xsdc__short</td>
<td>short</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date/Time/Duration</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:date</td>
<td>xsdc__date</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:dateTime</td>
<td>xsdc__dateTime</td>
<td>struct tm</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 2011, 2012
Table 17. XML to C type mapping (continued)

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Axis Type</th>
<th>Actual C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:duration</td>
<td>xsdc__duration</td>
<td>long</td>
</tr>
<tr>
<td>xsd:gDay</td>
<td>xsdc__gDay</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:gMonth</td>
<td>xsdc__gMonth</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:gMonthDay</td>
<td>xsdc__gMonthDay</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:gYear</td>
<td>xsdc__gYear</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:gYearMonth</td>
<td>xsdc__gYearMonth</td>
<td>struct tm</td>
</tr>
<tr>
<td>xsd:time</td>
<td>xsdc__time</td>
<td>struct tm</td>
</tr>
</tbody>
</table>

**String**

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Axis Type</th>
<th>Actual C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:anyURI</td>
<td>xsdc__anyURI</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:anyType</td>
<td>xsdc__anyType</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:ENTITY</td>
<td>xsdc__ENTITY</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:ENTITIES</td>
<td>xsdc__ENTITIES</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:ID</td>
<td>xsdc__ID</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:IDREFS</td>
<td>xsdc__IDREFS</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:language</td>
<td>xsdc__language</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:Name</td>
<td>xsdc__Name</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:NCName</td>
<td>xsdc__NCName</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:NMToken</td>
<td>xsdc__NMToken</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:NMTokenS</td>
<td>xsdc__NMTokenS</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:normalizedString</td>
<td>xsdc__normalizedString</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:notation</td>
<td>xsdc__notation</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:QName</td>
<td>xsdc__QName</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:string</td>
<td>xsdc__string</td>
<td>char *</td>
</tr>
<tr>
<td>xsd:token</td>
<td>xsdc__token</td>
<td>char *</td>
</tr>
</tbody>
</table>

**Other**

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Axis Type</th>
<th>Actual C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:base64Binary</td>
<td>xsdc__base64Binary</td>
<td>Implemented as C structure:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>typedef struct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xsdc_unsignedByte * __ptr;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xsdc_int __size;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>} xsdc_base64Binary;</td>
</tr>
<tr>
<td>xsd:boolean</td>
<td>xsdc__boolean</td>
<td>enum</td>
</tr>
<tr>
<td>xsd:hexBinary</td>
<td>xsdc__hexBinary</td>
<td>Implemented as C structure:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>typedef struct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xsdc_unsignedByte * __ptr;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xsdc_int __size;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>} xsdc_hexBinary;</td>
</tr>
</tbody>
</table>

The Axis types are defined in the header file `<install_dir>/include/axis/AxisUserAPI.h`. The `struct tm` structure used for many of the time-related types can be found in header file `time.h`. 

116  Web Services Client for ILE: Programming Guide
Simple types

Most of the simple XML data types defined by XML Schema and SOAP 1.1 encoding are mapped to their corresponding C types. You can see the details of the mapping in Table 17 on page 115 above.

One thing to keep in mind is how an element declaration with a nillable attribute set to true for a built-in simple XML data type is mapped. If the simple type is not already a pointer type (i.e. all the string types are pointer types), the simple type will be mapped to a pointer type. For example, the following schema fragment will get mapped to an integer pointer type (i.e. xsdc__int *):

```xml
<xsd:element name="code" type="xsd:int" nillable="true"/>
```

In addition, a simple type that is optional (minoccurs attribute set to 0) will also be mapped to a pointer type if the type is not already a pointer type.

Complex types

XML Schema complex types are mapped to C structures.

Let us look at the mapping that occurs for the following schema fragment:

```xml
<xsd:complexType name="Book">
  <sequence>
    <element name="author" type="xsd:string"/>
    <element name="price" type="xsd:float"/>
  </sequence>
  <xsd:attribute name="reviewer" type="xsd:string"/>
</xsd:complexType>
```

The above example is an example of a complex type that is named Book, and contains two elements, author and price, in addition to an attribute, reviewer. The complex type will get mapped to the following C structure:

```c
typedef struct BookTag {
  xsdc__string reviewer;
  xsdc__string author;
  xsdc__float price;
} Book;
```

In addition to the Book structure, the following functions are generated:

```c
int Axis_Serialize_Book(Book* param, AXISHANDLE pSZ, AxiscBool bArray);
int Axis_DeSerialize_Book(Book* param, AXISHANDLE pDZ);
void* Axis_Create_Book(int nSize=0);
void Axis_Delete_Book(Book* param, int nSize=0);
```

The Axis_Serialize_Book() and Axis_DeSerialize_Book() functions are used by the Axis engine to serialize and deserialize elements of type Book. The Axis engine uses the Axis_Create_Book() function to create the C structure that will hold the data during deserialization. The nSize parameter is used to indicate whether a single (i.e. when nSize equals to zero) structure is to be returned or an array (i.e. when nSize greater than zero) of structures is to be returned. The Axis_Delete_Book() is the function used by client applications to free up C structures of type Book that are returned by the Axis engine. In the case of Axis_Delete_Book(), the nSize parameter is used to indicate whether a single structure is to be deleted or an array of structures is to be deleted.

Arrays

Arrays for the C language are patterned after the structure Axisc_Array. The structure is defined in the header file <install_dir>/include/axis/AxisUserAPI.h. The structure is depicted below:
typedef struct {
    void** m_Array;
    int m_Size;
    AXISC_XSDTYPE m_Type;
} Axisc_Array;

The fields in the structure include: m_Array, which contains the elements of the array; m_Size, which contains the size of the array; and m_Type, which is an enumerator that gives an indication of the type of element (AXISC_XSDTYPE type is an enumerator defined in <install_dir>/include/axis/TypeMapping.h).

Axis provides array structures for each of the defined simple types. These are defined in <install_dir>/include/axis/AxisUserAPI.h. An example of a simple array type is xsdc__int_Array.

Below is the same schema fragment we have used previously, but we have also increased the number of authors a book can have to 10 by adding maxOccurs="10" to the author element:

```xml
<xsd:complexType name="Book">
    <sequence>
        <element name="author" type="xsd:string" maxOccurs="10"/>
        <element name="price" type="xsd:float"/>
    </sequence>
    <xsd:attribute name="reviewer" type="xsd:string"/>
</xsd:complexType>
```

For the above XML Schema, the following structure is generated:

typedef struct BookTag {
    xsdc__string reviewer;
    xsdc__string_Array* author;
    xsdc__float price;
} Book;

As you can see, the string array structure is now being used to store the values for the author element.

**WSDL to C mapping**

Now that we understand how the XML Schema types are mapped to Axis-defined language types, we can now review how a service described in a WSDL document gets mapped to the corresponding C representation. The following sections will refer to the GetQuote.wsdl WSDL document that is shipped as part of the product in directory <install_dir>/samples/getQuote and is listed in "The GetQuote.wsdl File" on page 187 to illustrate how various WSDL definitions get mapped to C. You should note the following:

- GetQuote.wsdl has only one service called GetQuoteService.
- The service only has one port type called StockQuote.
- The StockQuote port type has only one operation called getQuote. The input to the getQuote operation is a string (the stock identifier) and the output from the operation is a float (the stock's price).

If you want to fully understand the WSDL document structure, see "WSDL document structure" on page 32. Now let us see how various WSDL definitions are mapped.

This section describes the mapping of a service described in a WSDL document to the corresponding C representation. The table below summarizes the WSDL and XML to C mappings:

<table>
<thead>
<tr>
<th>WSDL and XML</th>
<th>C structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:complexType (structure)</td>
<td>C structure.</td>
</tr>
</tbody>
</table>

Note: The xsd:complexType can also represent an exception if referenced by a wsdl:message for a wsdl:fault.
Table 18. WSDL and XML to C mapping summary (continued)

<table>
<thead>
<tr>
<th>WSDL and XML</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested xsd:element or xsd:attribute</td>
<td>C structure field.</td>
</tr>
<tr>
<td>xsd:complexType (array)</td>
<td>C Axis array structure.</td>
</tr>
<tr>
<td>wsdl:message determines the method parameter signature.</td>
<td>Service interface function signature.</td>
</tr>
<tr>
<td>wsdl:portType</td>
<td>Service interface function.</td>
</tr>
<tr>
<td>wsdl:operation</td>
<td>Service interface function.</td>
</tr>
<tr>
<td>wsdl:binding</td>
<td>No direct mapping, affects SOAP communications style and transport.</td>
</tr>
<tr>
<td>wsdl:service</td>
<td>No direct mapping.</td>
</tr>
<tr>
<td>wsdl:port</td>
<td>Used as default Web service location.</td>
</tr>
</tbody>
</table>

**Mapping XML defined in wsdl:types**

The wsdl2ws.sh command will either use an existing C type or generate a new C type (a C structure) for the XML schema constructs defined in the wsdl:types section. The mappings that the wsdl2ws.sh command supports is discussed in ["XML schema to C type mapping" on page 115](#). As previously stated, the wsdl2ws.sh command either will ignore constructs that it does not support or issue an error message.

If we look at the wsdl:types part of the WSDL document we see that two elements are defined: getQuote, defined as a complex type with one element of type xsd:string; and getQuoteResponse, also defined as a complex type with one element of type xsd:float.

```
<wsdl:types>
  <ati:schema elementFormDefault="qualified"
    targetNamespace="http://stock.ibm.com"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:ati="http://www.w3.org/2001/XMLSchema">

    <ati:element name="getQuote">
      <ati:complexType>
        <ati:sequence>
          <ati:element name="arg_0_0" type="xsd:string"/>
        </ati:sequence>
      </ati:complexType>
    </ati:element>

    <ati:element name="getQuoteResponse">
      <ati:complexType>
        <ati:sequence>
          <ati:element name="getQuoteReturn" type="xsd:float"/>
        </ati:sequence>
      </ati:complexType>
    </ati:element>

  </ati:schema>
</wsdl:types>
```

For the WSDL document fragment above, the wsdl2ws.sh command does not generate any new structures since both elements are defined to be built-in simple types. The xsd:string type is mapped to xsdc__string and the xsd:float type is mapped to xsdc__float.
**Mapping of wsdl:portType**

A port type is a named set of abstract operations and the abstract messages involved. The name of the wsdl:portType will be used in the names of the Web service proxy (termed service interface) functions. A port type is mapped to four functions:

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_&lt;portType-name&gt;_stub</td>
<td>Function that is used to get an object representing the service interface (i.e. the Web service proxy stub). The type of the object is AXISCHANDLE, which is defined as a void pointer (i.e. void*). The AXISCHANDLE is used by Axis to represent different objects so one must be sensitive to what the object represents. In this case the object is the interface to the Axis engine. The operations that can be performed on the object are listed in the &lt;install_dir&gt;/include/axis/client/Stub.h header file.</td>
</tr>
<tr>
<td>destroy_&lt;portType-name&gt;_stub</td>
<td>Function used to destroy service interface objects that are obtained by invoking get_&lt;portType-name&gt;_stub.</td>
</tr>
<tr>
<td>get_&lt;portType-name&gt;_Status</td>
<td>Function used to get the status of last Web service operation.</td>
</tr>
<tr>
<td>set_&lt;portType-name&gt;_ExceptionHandler</td>
<td>Function used to set the exception (i.e. SOAP fault) handler for the service interface.</td>
</tr>
</tbody>
</table>

Now let us look at a concrete example of how the wsdl:portType below gets mapped.

```xml
<wSDL:portType name="StockQuote">
  <wSDL:operation name="getQuote">
    <wSDL:input message="impl:getQuoteRequest" name="getQuoteRequest"/>
    <wSDL:output message="impl:getQuoteResponse" name="getQuoteResponse"/>
  </wSDL:operation>
</wSDL:portType>
```

The wsdl2ws.sh command will generate the following C functions:

```c
extern AXISCHANDLE get_StockQuote_stub(const char* pchEndPointUri);
extern void destroy_StockQuote_stub(AXISCHANDLE pStub);
extern int get_StockQuote_Status(AXISCHANDLE pStub);
extern void set_StockQuote_ExceptionHandler(AXISCHANDLE pStub,
                                           AX Exception_HANDLER_FUNCT fp);
extern xsdc__float getQuote(AXISCHANDLE pStub, xsdc__string Value0);
```

The first four C functions shown above are the functions that are generated in support of the service interface. You see how the wsdl:portType name StockQuote is used in the naming of the functions. The last C function shown, getQuote(), is mapped from the wsdl:operation element defined in the wsdl:portType (refer to [Mapping of wsdl:operation](#) for further explanation of the mapping of wsdl:operation). One thing to note about the get_StockQuote_stub() function, and that is the default URL to the Web service will be used if get_StockQuote_stub() is invoked with a NULL value for endpoint parameter. The default URL is whatever is specified in the wsdl:port WSDL definition.

**Mapping of wsdl:operation**

A wsdl:operation within a wsdl:portType is mapped to a C function. The name of the wsdl:operation is mapped to the name of the function. The first parameter is of type AXISCHANDLE that represents the service interface stub object.
The wsdl:operation contains wsdl:input and wsdl:output elements that reference the request and response wsdl:message constructs using the message attribute. Each function parameter (except the first) is defined by a wsdl:message part referenced from the input and output elements:

- A wsdl:part in the request wsdl:message is mapped to an input parameter.
- A wsdl:part in the response wsdl:message is mapped to the return value.
- If there are multiple wsdl:parts in the response message, they are mapped to output parameters.
- A wsdl:part that is both the request and response wsdl:message is mapped to an inout parameter.

The wsdl:operation can contain wsdl:fault elements that references wsdl:message elements describing the fault (refer to "Mapping of wsdl:fault" on page 122 for more details on wsdl:fault mapping).

The Web Services Client for ILE supports the mapping of operations that use either a request/response or one-way (where wsdl:output is not specified in the wsdl:operation element) message exchange pattern. For the one-way message exchange pattern, the Axis engine expects an HTTP response to be returned from the Web service. Under normal conditions, the HTTP response would contain no SOAP data. However, if a SOAP fault is returned by the Web service, the Axis engine will process the fault.

Below are the wsdl:message and wsdl:portType WSDL definitions in the GetQuote.wsdl document:

```xml
...<wsdl:message name="getQuoteRequest">
  <wsdl:part element="impl:getQuote" name="parameters"/>
</wsdl:message>
<wsdl:message name="getQuoteResponse">
  <wsdl:part element="impl:getQuoteResponse" name="parameters"/>
</wsdl:message>
...
<wsdl:portType name="StockQuote">
  <wsdl:operation name="getQuote">
    <wsdl:input message="impl:getQuoteRequest" name="getQuoteRequest"/>
    <wsdl:output message="impl:getQuoteResponse" name="getQuoteResponse"/>
  </wsdl:operation>
</wsdl:portType>
...
```

The above wsdl:operation definition gets mapped to the following service interface function:

```c
extern xsdc__float getQuote(AXISCHANDLE pStub, xsdc__string Value0);
```

**Mapping of wsdl:binding**

The wsdl:binding information is used to generate an implementation specific client side stubs. What code is generated is dependent on protocol-specific general binding data, such as the underlying transport protocol and the communication style of SOAP.

There is no C representation of the wsdl:binding element.

**Mapping of wsdl:port**

A wsdl:port definition describes an individual endpoint by specifying a single address for a binding.

The specified endpoint will be used in as the default location of the Web service. So in the case of our example, the URL specified in wsdl:port definition below will be the URL that is used if get_StockQuote_stub() is invoked with a NULL value for endpoint parameter (i.e. get_StockQuote_stub(NULL)).
Mapping of wsd1:fault

Within the wsd1:operation definition you can optionally specify the wsd1:fault element, which specifies the abstract message format for any error messages that may be returned as a result of invoking a Web service operation.

The wsd1:fault element must reference a wsd1:message that contains a single message part. As of this writing, Axis only supports message parts that are xsd:complexType types. The mapping that occurs is similar to the mapping that occurs when generating code for complex types.

So what happens when a SOAP fault is received? In order to obtain information about SOAP faults Axis provides a way where a C client application can register a function that will act as the exception handler.

Recall in the discussion about how the C mapping of wsd1:portType results in four functions being generated, including a function to set the exception handler for a service interface. For our example, the GetQuote.wsdl WSDL document, the following exception handler is generated:

```c
extern void set_StockQuote_ExceptionHandler(AXISCHANDLE pStub,
                                          AXIS_EXCEPTION_HANDLER_FUNCT fp);
```

where AXIS_EXCEPTION_HANDLER_FUNCT is a typedef defined in <install_dir>/include/axis/Axis.h as:
```c
typedef void (* AXIS_EXCEPTION_HANDLER_FUNCT)(int exceptionCode,
                                              const char *exceptionString,
                                              AXISCHANDLE pSoapFault,
                                              void *faultDetail);
```

When a SOAP fault is encountered (or a non-Fault exception for that matter), the Axis engine will throw an exception, and the C interfaces to the Axis engine catch the exception and attempts to produce a SoapFault object and associated fault detail. The C interfaces to the Axis engine then determines if there is a service interface exception handler and calls the function, passing it the generic exception code and exception string associated with the exception, in addition to the SoapFault\(^8\) object and fault detail (note that it is possible that there is no SoapFault or fault detail, in which case a NULL pointer is passed). If there is no service interface exception handler, then the generic exception handler is invoked. The generic exception handler by default will simply print out the exception string to stderr. However, a client application can override the default exception handler by invoking axiscAxisRegisterExceptionHandler()\(^9\).

More information on exception handling in C can be found in “C exception handling” on page 127.

---

8. Operations that can be done against a SoapFault object are listed in the <install_dir>/include/axis/ISoapFault.h. There are functions to retrieve information such as the faultcode, faultstring and faultactor.

9. The axiscAxisRegisterExceptionHandler() function is defined in <install_dir>/include/axis/Axis.h.
Chapter 16. Developing a Web services client application using C stubs

This chapter will describe the steps one must go through to develop a Web service client application using a C stub code.

To develop a Web services client application, the following steps should be followed:

1. Generate the client Web service stubs using the `wsdl2ws.sh` command.
2. Complete the client implementation.
3. (Optional) Create client-side handler.
4. Deploy the application.

The following sections will discuss each of these steps. For illustrative purposes we will be using the sample code that is shipped as part of the product in directories `<install_dir>/samples/getQuote`. We will be using the following files:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetQuote.wsdl</td>
<td>GetQuote WSDL file.</td>
</tr>
<tr>
<td>myGetQuote.c</td>
<td>Client implementation code written in C.</td>
</tr>
</tbody>
</table>

Source listings for the client application code can be found at Appendix A, “Code Listings for myGetQuote Client Application,” on page 187.

Generating the C stub code

Before you can create a web service client application, you must first generate the C client stub using the `wsdl2ws.sh` tool. The `wsdl2ws.sh` tool uses the WSDL file that is passed to it, and any associated XSD files referenced in the WSDL file, to create client stubs.

We will be using the GetQuote.wsdl file located in directory `<install_dir>/samples/getQuote`. The WSDL file comes from the installation Web Services Samples provided with WebSphere Application Server (Version 5.0 or later). This very simple sample provides a good introduction to using `wsdl2ws.sh`.

To generate the client stub from the WSDL source file, complete the following steps.

1. Create a library called `MYGETQUOTE` in which the program objects will be stored by issuing the CL command `CRTLIB` from the CL command line as follows:

```
CRTLIB MYGETQUOTE
```

2. Start a Qshell session by issuing the `QSH` CL command from the CL command line.

3. Run the `wsdl2ws.sh` tool to generate the client C stub as shown in following example:

```
<install_dir>/bin/wsdl2ws.sh -o/myGetQuote/C -lc -s/qsys.lib/mygetquote.lib/wsc.srvpgm <install_dir>/samples/getQuote/GetQuote.wsdl
```

If you examine the command, you see that we are indicating to the `wsdl2ws.sh` tool that C stub code should be generated and stored in directory `/myGetQuote/C`, and that a service program, `/qsys.lib/mygetquote.lib/wsc.srvpgm`, should be created using the generated stub code.

The files generated by the `wsdl2ws.sh` tool is shown below:
Note that in addition to C code being generated, the file ws.cl is also generated. This file is a CL source file that has the CL commands needed to recreate the service program. You can copy this source file to a source physical file and create a CL program. Here is the contents of the file:

```cl
PGM
DCL VAR(&LIB) TYPE(*CHAR) LEN(10) VALUE(MYGETQUOTE)
DCL VAR(&SRVPGM) TYPE(*CHAR) LEN(10) VALUE(WSC)
QSYS/CRTCMOD MODULE(&LIB/wsc0) +
    OPTIMIZE(40) DBGVIEW(*NONE) +
    SRCSTMF('/myGetQuote/C/StockQuote.c') +
    INCDIR('/QIBM/PRODDATA/OS/WEBSERVICES/V1/CLIENT/INCLUDE') +
    REPLACE(*YES) ENUM(*INT) +
    TEXT('StockQuote.c')
QSYS/CRTSRVPGM SRVPGM(&LIB/&SRVPGM) +
    MODULE( +
        &LIB/wsc0 +
    ) +
    EXPORT(*ALL) ACTGRP(*CALLER) +
    BNDSRVPGM(QSYSDIR/QAXIS10CC) +
    TEXT('ws Web service')
ENDPGM
```

Now that the C stub code has been created and a service program containing the C stub code created, you can go on to the next step, "Completing C client implementation."

**Completing C client implementation**

After the client stubs have been generated, the stubs can be used to create a Web service client application.

We will illustrate what you need to do to create C applications using the example of the C stub code generated from GetQuote.wsdl by the wsdl2ws.sh tool as described in "Generating the C stub code" on page 123. However, before we continue, you should note the following points:

- GetQuote.wsdl has only one service called getQuoteService.
- The service only has one port type called StockQuote.
- The StockQuote port type has only one operation called getQuote.
- The Web service is called StockQuote. So to get an instance of the Web service you would call the get_StockQuote_stub() function. The handle that is returned by the function should then be used when calling the Web service operation. To destroy the Web service instance, you would call the destroy_StockQuote_stub() function.

To build the myGetQuote client application, complete the following steps.

1. Change the current working directory to the location of the C stub code. Issue the following command from the CL command line:
   ```cl
   cd '/myGetQuote/C'
   ```
2. Copy the sample C code the uses the generated stub code from the product samples directory to the current working directory by issuing the following command from the CL command line:
   ```cl
   COPY OBJ('<install_dir>/samples/getQuote/myGetQuote.c') TODIR('/myGetQuote/C')
   ```

10. If you have not read Chapter 15, "WSDL and XML to C mappings," then it would be a good time to do so prior to reading this section.
3. Change the ServerName and PortNumber in the file copied in the previous step to match your server. If WebSphere Application Server is on your own machine and the default values have been used, ServerName is localhost and PortNumber is 9080.

4. Build the client application by using the following commands from the CL command line:

```
CRTCMD MOD(MYGETQUOTE/mygetquote)
   SRCSTMF('/myGetQuote/C/myGetQuote.c')
   INCDIR('/qibm/proddata/os/webservices/v1/client/include')
   ENUM(*INT)

CRTPGM PGM(MYGETQUOTE/MYGETQUOTE)
   MODULE(MYGETQUOTE/MYGETQUOTE)
   BNDSRVPGM(QSYSDIR/QAXIS10CC MYGETQUOTE/WSC)
```

When you have finished coding and building your web service client application, you are ready to deploy and test the application as described in “Deploying the client application” on page 77.

**Note:** If you want to use one or more handlers with your application, see Chapter 11, “Creating client-side handlers,” on page 79.

### Deploying the client application

When you have finished coding and building your web service client application, you are ready to deploy and test the application.

In our example, we have not modified the Axis configuration file axiscpp.conf. However, if we had modified it (e.g. we were using client-side handlers), we would need to ensure that the AXISCPP_DEPLOY environment variable points to the directory containing the /etc directory (the axiscpp.conf file would be located in the /etc directory), as described in “The axiscpp.conf file” on page 57.

The steps below use the example myGetQuote client application, and assume that a GetQuote service is running. (This service is with the samples supplied with WebSphere Application Server Version 5.0.2 or later). If you do not have the appropriate service, you must create the service code from the WSDL in the samples directory.

Once you have confirmed the above prerequisites, run and test the client application by completing the following steps.

1. Run the myGetQuote application.
2. Check that the myGetQuote application has returned the price of IBM shares in dollars.

The example screen shot below shows the myGetQuote application run from the command line in which client-side handlers are not being used.

```
> call MYGETQUOTE/MYGETQUOTE
The stock quote for IBM is $94.33
```

If we were had implemented client-side handlers, then we would have seen the following results:

```
> call MYGETQUOTE/MYGETQUOTE
Before the pivot point Handler can see the request message.
Past the pivot point Handler can see the response message.
The stock quote for IBM is $94.33
```
Chapter 17. C stub programming considerations

This chapter covers programming considerations when you begin writing your applications to take advantage of Web services client for ILE C stub code.

C exception handling

Web Services Client for ILE uses exceptions to report back any errors that have occurred during the transmission of a SOAP message. This includes errors that are detected by the Axis engine or SOAP faults that are returned by the Web service.

When using the C-stub interfaces, errors that occur are reported to the client application by calling an exception handler function. There are two locations where Web services client for ILE looks for the exception handler: the stub exception handler and then the generic exception handler.

When C stubs are generated, in addition to functions to get a stub and destroy a stub, there is also a function to register a stub exception handler.

So let us take a look at an example. Below is a wsdl:portType definition called MathOps that has a div operation and has three SOAP faults defined - DivByZeroStruct, SpecialDetailStruct and OutOfBoundStruct:

```xml
<wsdl:portType name="MathOps">
  <wsdl:operation name="div">
    <wsdl:input message="impl:divRequest" name="divRequest"/>
    <wsdl:output message="impl:divResponse" name="divResponse"/>
    <wsdl:fault message="impl:DivByZeroStruct" name="DivByZeroStruct"/>
    <wsdl:fault message="impl:SpecialDetailStruct" name="SpecialDetailStruct"/>
    <wsdl:fault message="impl:OutOfBoundStruct" name="OutOfBoundStruct"/>
  </wsdl:operation>
</wsdl:portType>
```

The definition of the SOAP fault messages is as follows:

```xml
<complexType name="OutOfBoundStruct">
  <sequence>
    <element name="varString" nillable="true" type="xsd:string"/>
    <element name="varInt" type="xsd:int"/>  
    <element name="specialDetail" nillable="true" type="impl:SpecialDetailStruct"/>
  </sequence>
</complexType>
<complexType name="SpecialDetailStruct">
  <sequence>
    <element name="varString" nillable="true" type="xsd:string"/>
  </sequence>
</complexType>
<complexType name="DivByZeroStruct">
  <sequence>
    <element name="varString" nillable="true" type="xsd:string"/>
    <element name="varInt" type="xsd:int"/>
    <element name="varFloat" type="xsd:float"/>
  </sequence>
</complexType>
```

When you generate C stub code, the prototype function for setting the stub exception handler would be:

```c
extern void set_MathOps_ExceptionHandler(AXISHANDLE pStub, 
                                       AXIS_EXCEPTION_HANDLER_FUNCT fp);
```
where AXIS_EXCEPTION_HANDLER_FUNCT is a typedef defined in Axis.h as:

```c
typedef void (* AXIS_EXCEPTION_HANDLER_FUNCT)(int exceptionCode,
    const char *exceptionString,
    AXISCHANDLE pSoapFault,
    void *faultDetail);
```

When Web services client for ILE throws an exception, the C-stub interfaces catch the exception and attempts to produce a SoapFault object and associated fault detail. The C-stub interfaces then determines if there is a stub exception handler and calls the function, passing it the generic exception code and exception string associated with the exception, in addition to the SoapFault object and fault detail (note that it is possible that there is no SoapFault or fault detail, in which case a NULL pointer is passed). If there is no stub exception handler, then the generic exception handler is invoked. The generic exception handler by default will simply print out the exception string to stderr. However, a client application can override the default exception handler by invoking the AXIS API axiscAxisRegisterExceptionHandler().

Each SOAP fault defined in the WSDL is represented as a structure. The generated DivByZeroStruct is as shown below:

```c
typedef struct DivByZeroStructTag {
    xsdc_string varString;
    xsdc_int varInt;
    xsdc_float varFloat;
} DivByZeroStruct
```

A pointer to this structure would be passed to an exception handler as the fault detail parameter.

An exception handler would need to obtain the fault object name to determine what the fault detail parameter represents since each SOAP fault that is defined will represent a different structure. This is done by calling the axiscSoapFaultGetCmplxFaultObjectName() API. For SOAP faults that are not defined in the WSDL, the fault detail, if one exists, is simply a character string.

The following example shows how a client application may process exceptions.

```c
// stub exception handler
int exceptionOccurred = 0;
void myExceptionHandler(int exceptionCode,
    const char *exceptionString,
    AXISCHANDLE pSoapFault,
    void *faultDetail)
{
    const char *pcCmplxFaultName;
    exceptionOccurred = 1;

    if (pSoapFault) {
        pcCmplxFaultName = axiscSoapFaultGetCmplxFaultObjectName(pSoapFault);
        if (0 == strcmp("DivByZeroStruct", pcCmplxFaultName))
            DivByZeroStruct *dbzs = (DivByZeroStruct *)faultDetail;
            printf("DivByZeroStruct Fault: \"%s\", \%d, %.6g\n",
                   dbzs->varString, dbzs->varInt, dbzs->varFloat);
        else if (0 == strcmp("SpecialDetailStruct", pcCmplxFaultName))
            SpecialDetailStruct *sds = (SpecialDetailStruct *)faultDetail;
            printf("SpecialDetailStruct Fault: \"%s\"

```
else
{
    printf("SoapFaultException: %s\n", faultDetail);
}

// Attempt to divide by zero.
main()
{
    AXISCHANDLE ws;
    int iResult;

    // Create the Web Service with default endpoint URL.
    ws = get_MathOps_stub( NULL );

    // register stub exception handler
    set_MathOps_ExceptionHandler(ws, myExceptionHandler);

    // Call the div method with two parameters. This will attempt to divide 1 by 0.
    iResult = div(ws, 1, 0);

    // Output the result of the division.
    if (!exceptionOccurred)
        printf("Result is %d\n", iResult);

    destroy_MathOps_stub(ws);
}

---

C memory management

The WSDL specification provides a framework for how information is to be represented and conveyed from place to place. Web services client for ILE maps this framework to program-language specific data object, such as structures. The data objects that are dynamically allocated from the storage heap must be deleted in order to avoid memory leaks. Information is represented by four generic types: simple types, arrays of simple type, complex types, and arrays of complex type. This section describes what you need to be aware of in order to avoid memory leaks.

Built-in simple types

There are more than 45 built-in simple types, which are defined in `<install_dir>/include/Axis/AxisUserAPI.h`. When a type is nillable or optional (that is, minOccurs="0"), it is defined as a pointer to a simple type.

The example below shows a typical simple type in a WSDL. The simple type used in this example is xsd:int, which is mapped to C type xsdc__int. The extract from the WSDL has an element called addReturn of type integer. This element is used by the add operation, which uses the addResponse element to define the type of response expected when the add operation is called.

```xml
<element name="addResponse">
    <complexType>
        <sequence>
            <element name="addReturn" type="xsd:int"/>
        </sequence>
    </complexType>
</element>
```

Later in the WSDL, the addResponse element is the response part for the add method. This produces the following Web services client for ILE web services function prototype from the simple type in the WSDL:

```c
extern xsdc__int add( ...);
```

Thus, the user generated application code for this example is as follows:
xsdc__int xsdc_iReturn = add(ws, ...);

When a type is nillable, (that is, nillable="true"), optional (that is, minOccurs="0"), or a text type (such as xsd:string), it is defined as a pointer.

<element name="addResponse">
    <complexType>
        <sequence>
            <element name="addReturn" nillable="true" type="xsd:int"/>
        </sequence>
    </complexType>
</element>

This produces the following Web services client for ILE web services function prototype:
extern xsdc__int * add( ...);

The user generated application code produced by the nillable simple type in the WSDL is as follows:
xsd__int * xsdc_piReturn = add(ws, ...);

// Later in the code...
// Delete this pointer and set it to NULL (as it is owned by the client application).
axiscAxisDelete(xsdc_piReturn, XSDC_INT);
xsd_piReturn = NULL;

Note: The example above shows the deletion of the return value. Any pointer that Web services client for ILE returns becomes the responsibility of the client application and does not go out of scope if the web service is deleted. The user application must delete the pointer to the object type once it is no longer required. For simple types, the pointer must be deleted by invoking axiscAxisDelete().

Arrays of simple type
Web services client for ILE provides array objects for each of the defined simple types. These are defined in <install_dir>/include/Axis/AxisUserAPI.h. An example of a simple array type is xsdc__int_Array.

The following example shows an extract from a WSDL that has two elements called simpleArrayRequest and simpleArrayResponse of array type integer. These elements are used by the simpleArray operation, which uses the simpleArrayRequest element to define the type of request and simpleArrayResponse element to define the type of response expected when the simpleArray operation is called.

<xsd:element name="simpleArrayRequest">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="simpleTypeRes" type="xsd:int" maxOccurs="unbounded"/>
        </xsd:sequence>
    </complexType>
</xsd:element>

<xsd:element name="simpleArrayResponse">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="simpleTypeReq" type="xsd:int" maxOccurs="unbounded"/>
        </xsd:sequence>
    </complexType>
</xsd:element>

Note that the maxOccurs attribute is used in this example. Web services client for ILE creates an array object for any type that is declared as having maxOccurs greater than one. Later in the WSDL, the simpleArrayRequest and simpleArrayResponse become the input and output parameters for the simpleArray method whose prototype is shown below:
xsd__int_Array * simpleArray(ws, xsdc__int_Array * pValue);

The prototype requires input and output arrays to be created. To avoid memory leaks, these must be created and managed properly. For information about the generation management and deletion of a typical input and output array, see the following two subsections:

• "Array types as input parameters" on page 131
• "Array types as output parameters" on page 131
**Array types as input parameters**

The prototype function requires an input array to be created. This array must be created and managed properly.

This array must be created and managed properly. If an array is to be used as an input parameter, then it has to be created and filled.

The following example shows the typical usage of a nillable simple array type required by a generated stub. The array is an example of the input array to the function. The example assumes that the array contains three elements whose values are 0, 1 and 2 respectively.

```c
// Need an input array of 3 elements.
int iArraySize = 3;

// Final object type to be passed as an input parameter to the web service.
xsdc__int_Array iInputArray;

// Preparatory array that contains the values to be passed. Note that the
// array is an array of pointers of the required type.
xsdc__int * ppiPrepArray[3];

// Loop used to populate the preparatory array.
for( iCount=0; iCount < iArraySize ; iCount++) {
    // Each element in the array of type pointers is filled with a pointer to an
    // object of the required type. In this example we have chosen the value of
    // each element to be the same as the current count and so have passed this
    // value to the new instance of the pointer.
    ppiPrepArray[iCount] = (xsdc__int *)axiscAxisNew(XSDC_INT, 0);
    *ppiPrepArray[iCount] = iCount;
}

// Set the contents of the final object to contain the elements of the
// preparatory array.
iInputArray.m_Array = ppiPrepArray;
iInputArray.m_Size = iArraySize;
iInputArray.m_Type = XSDC_INT;

... Call the web service(s) that use the input array ...

// No longer require iInputArray. Delete allocated memory.
for( int iCount=0; iCount < iArraySize ; iCount++) {
    axiscAxisDelete(ppiPrepArray[iCount], XSDC_INT);
    ppiPrepArray[iCount] = NULL;
}
```

When the method returns, iInputType can be destroyed. If iInputType was created as a pointer, then the client user code must remember to delete it otherwise the code will have created a memory leak.

**Array types as output parameters**

The prototype method requires an output array to be created. This array must be created and managed properly.

Following on from the example in “Array types as input parameters,” the following example shows the client application calling the simpleArray method on the web service and using the returned array. The following example shows a typical usage of the method produced by the WSDL example of an array of nillable simple type.

```c
xsdc__int_Array * piSimpleResponseArray = simpleArray(ws, &InputArray);

for( iCount = 0 ; iCount < piSimpleResponseArray->m_Size; iCount++) {
    if ( piSimpleResponseArray->m_Array[iCount] != NULL)
        printf("Element[%d]=%d\n", iCount, *piSimpleResponseArray->m_Array[iCount]);
```
Later in the code...

```c
axiscAxisDelete(piSimpleResponseArray, XSDC_ARRAY);
piSimpleResponseArray = NULL;
```

**Notes:**
1. The returned pointer is not NULL.
2. The user only needs to delete the object returned by the call to the web service. The client must not delete any object that is extracted from within this object. For example, in the previous code sample, `piSimpleResponseArray->m_Array` must not be deleted by the user as it will be deleted when the container structure is deleted on the call to `axiscAxisDelete()`.
3. If the pointer to the array of pointers to integer values (`m_Array`) is NULL, then this indicates an empty array. If this is the case, `m_Size` is equal to zero.
4. Arrays of simple types, if generated by the web service, must be deleted by a call to `axiscAxisDelete()`.

**Complex types and arrays of complex type**

When complex types are used in a web service, the same rules as for simple types apply.

**Complex types**

The following example shows classes produced from WSDL with a complex type. As shown in this example, complex types are represented as structures in C:

```c
typedef struct ComplexTypeTag
{
    xsdc_string Message;
    xsdc_int MessageSize;
} ComplexType;
```

For non-simple types, Web services client for ILE produces functions that are used by the serializer and deserializer when generating the SOAP message and when deserializing data obtained from the Web service. There is a function to create a complex type, serialize and deserialize a complex type, and to delete a complex type. For example, the following prototypes of the functions are typical of what is produced:

```c
extern int Axis_DeSerialize_ComplexType(ComplexType* param, AXISCHANDLE pSZ);
extern void* Axis_Create_ComplexType(int nSize);
extern void Axis_Delete_ComplexType(ComplexType* param, int nSize);
extern int Axis_Serialize_ComplexType(ComplexType* param, AXISCHANDLE pSZ, AxiscBool bArray);
```

From the client perspective, the important function is the delete function (in the above example, `Axis_Delete_ComplexType`) that is used to delete output generated by a call to a web service function.

The same rules as for simple types apply to a complex type when used on a call to a web service function:

- The client is responsible for generating the input parameter information and for deleting any objects created during this process.
- The client is responsible for deleting the output object returned by the call to the web service function. The deletion must be done by invoking the corresponding `Axis_Delete_xxxxx()` function (where `xxxxx` is the datatype of the complex object).

**Arrays of complex type**

If a WSDL describes a complex type being used within an array, the `wsdl2ws.sh` tool generates a corresponding array object using the complex name type suffixed with "_Array".
Summary of rules

There are a number of rules relating to memory management that you must follow when using the C stub code generated by the wsd12ws.sh tool.

1. Resources for objects that are passed to or obtained from the web service function call as pointers are the responsibility of the client application.
2. Stub objects must be destroyed when not needed by calling the associated destroy function.
3. Return parameters must be deleted when they are one of:
   - Complex type
   - Array
   - String based types (for example, xsdc__string, xsdc__Name, and so on)
   - Nullable
   - Optional
   - xsdc__base64Binary or xsdc__hexBinary (see rule 6)
4. Return parameters that are complex types and non-simple type arrays, must be deleted by calling the appropriate deletion function.
5. Return parameters that are pointer-based simple types must be deleted by calling the axiscAxisDelete() function.
6. When the return parameter is a simple type of xsdc__base64Binary or xsdc__hexBinary, which are represented by a structure that contains a pointer and length, the pointer in the structure must be deleted by calling axiscAxisDelete() function.

Securing web service communications in C stub code

This section explains how to use Secure Sockets Layer (SSL) to set up security when using C stub code.

You can secure your HTTP messages by using SSL, which encrypts the request and response messages before they are transmitted over the wire.

Note: Handlers are not affected by SSL as they receive the message either before encryption or after decryption.

Any web service that uses SSL adds the suffix ‘s’ for secure to the http name in the URL. For example: http://some.url.com becomes https://some.url.com

A secure endpoint URL is an endpoint beginning with ‘https’. To allow secure endpoint URLs to be used, you must pass security information to the C stub. You can do this either by adding the required information to the The axiscpp.conf file on page 57 configuration file, or by configuring the settings for secure service using the ”axiscStubSetSecure()” on page 153 Axis C API.

Cookies

This section describes the cookie support that Web services client for ILE provides, including getting cookies from services and adding cookies to other services, and removing cookies from C stub instances.
Cookie attributes

The following table summarizes how Web services client for ILE handles cookie attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>expires</td>
<td>This attribute is ignored. If a server sends a signal to the client asking it to expire a cookie, the client does not do so. Once set by a server, the client continues to send cookies on each request using that stub. If a new stub instance is created and used, then the cookies from the original stub instance are not sent on requests from the new stub instance.</td>
</tr>
<tr>
<td>path</td>
<td>This attribute is ignored. Cookies are sent on all requests and not just on requests to a URI applicable to the path.</td>
</tr>
<tr>
<td>domain</td>
<td>This attribute is ignored. Cookies have affinity to a stub and are domain neutral.</td>
</tr>
<tr>
<td>secure</td>
<td>This attribute is ignored. If secure is set on a cookie, this has no effect and the cookie is sent on all future requests regardless of whether the channel is secure or not.</td>
</tr>
</tbody>
</table>

Use of cookies across multiple stub instances

If cookies are required in a different instance of a stub such as when a login is done on one service and the login session cookies are required on other services, you can use the APIs in the following example. This C example uses two instances of the calculator service and a login service. The first instance uses the login service and receives some cookies back representing the session cookies. These cookies are required for interacting with the calculator service in order to authenticate to the server that hosts the calculator service.

```c
AXISCHANDLE calculator = NULL;
AXISHANDLE newCalculatorInstance = NULL;
int result;

// Get instance of login service
AXISCHANDLE loginService = get_LoginService_stub("http://loginserver/loginservice");

// must tell the service to save cookies
axiscStubSetMaintainSession(loginService, 1);

// login so that we can get the session cookies back
login(loginService, "myusername", "mypassword");

// Store the cookies so they can be given to the Calculator web service as
// authentication.
int currentCookieKey=0;
char * cookieKeys[2];
const char* key = axiscStubGetFirstTransportPropertyKey(loginService, 1);
char *keyString = (char *)key;
if (key)
{
    // Only get the "Set-Cookie" transport properties - as these are what the server
    // sends to the client to set cookies.
    if(strcmp(keyString, "Set-Cookie")==0)
    {
        const char* valueString = axiscStubGetCurrentTransportPropertyValue(loginService, 1));
        cookieKeys[currentCookieKey++] = (char *)valueString;
    }
}

// then the rest of the cookies
while(key = axiscStubGetNextTransportPropertyKey(loginService, 1))
{
    char *nextKeyString = (char *)key;
    // Only get the "Set-Cookie" transport properties - as these are what the server
// sends to the client to set cookies.
if(strcmp(nextKeyString, "Set-Cookie") == 0)
{
    char * valueString = axiscStubGetCurrentTransportPropertyValue(loginService, 1));
    cookieKeys[currentCookieKey++] = valueString;
}

// Now we've logged in and stored the cookies we can create the calculator service,
// set the cookies on that stub instance and use the calculator.
calculator = get_Calculator_stub("http://calculatorserver/calculatorservice);
taxiscStubSetMaintainSession(calculator, 1);

// OK, Now add the previously saved session cookies on to this new service
// as this service does not pick up the cookies from the other stub.
currentCookieKey = 0;
while(currentCookieKey < 3)
{
    axiscStubSetTransportProperty(calculator, "Cookie",
        cookieKeys[currentCookieKey++]);
}

// Now, when we use the service it will send the session cookies to the server
// in the http message header
// which allows the server to authenticate this instance of the service.
result = add(calculator, 1, 2);

// If we continue to use this instance of the calculator stub then the cookies
// will be continue to be sent.
result = add(calculator, 1, 2);

// If we use a new instance of the calculator then it will fail because we have
// not set the cookies
newCalculatorInstance = get_Calculator_stub("http://calculatorserver/calculatorservice);

// This will fail with an exception because we have not set the authentication
// cookies
result = add(newCalculatorInstance, 1, 2);

**Manipulation of cookies using C AXIS APIs**

It is sometimes necessary to remove cookies from stub instances.

- To delete a single cookie from a C stub instance:
  
  ```c
  axiscStubDeleteTransportProperty(service, cookiename);
  ```
  
  For example:
  ```c
  axiscStubDeleteTransportProperty(calculator, "loginCookie");
  ```

- To delete all cookies from a C stub instance:
  ```c
  axiscStubDeleteTransportProperty(service, "Cookie");
  ```
  
  For example:
  ```c
  axiscStubDeleteTransportProperty(calculator, "Cookie");
  ```

**Floating point numbers in C types**

This section provides reference information about using floating point numbers with Web services client for ILE.

The XML specification refers to the IEEE specification for floating point numbers. The specification lists that float and double have the following precision:

- **Float type numbers**, 1 sign bit, 23 mantissa bits and 8 exponent bits.
- **Double type numbers**, 1 sign bit, 52 mantissa bits and 11 exponent bits.
For float, with a mantissa able to represent any number in the range $1 > x > 1/2^{23}$, this gives a minimum accuracy of 6 digits. Similarly, for double, with a mantissa able to represent any number in the range $1 > x > 1/2^{52}$, this gives a minimum accuracy of 10 digits.

When displaying floating point numbers, you must ensure that any potential inaccuracies due to rounding errors, and so on are not visible. Therefore, to ensure the correct level of precision, for float types, instead of using:

```c
printf( "%f", myFloat);
```

you must use the following formatting command:

```c
printf( "%g", myFloat);
```

Similarly, to ensure the correct level of precision for double types, instead of using:

```c
printf( "%f", myDouble);
```

you must use the following formatting command:

```c
printf( "%g", myDouble);
```
Chapter 18. Troubleshooting C client stubs

This chapter is intended to help you learn how to detect, debug, and resolve possible problems that you may encounter when generating or using C stub code.

C stub code generation problems

When you use the wsdl2ws.sh tool to generate C stub code, the tool will generate an exception for any error that is encountered. Typical errors include the inability for the tool to resolve to an XSD file used in the specified WSDL file or a syntactically incorrect WSDL file. You will need to correct the problem and try running the tool again.

C stub code compile problems

If there is a compile problem in C stub code, the most likely cause of the problem is the use of an unsupported construct. The wsdl2ws.sh tool will not always generate an exception when used against a WSDL file that contains an unsupported WSDL construct. The problem may manifest itself when compiling the generated stub code. To see what is supported by the tool, see “Supported specifications and standards” on page 43.

C stub code runtime problems

Invoking a Web service operation may result in the Web service returning a SOAP fault as a response. There can be many reasons for this, and the only sure way to determine where the problem lies is by examining the generated SOAP request and resulting response.

The Web services client for ILE client engine has a tracing capability that traces the request and response messages. To learn about the tracing support in Axis, see the axiscAxisStartTrace() on page 140 Axis C API.
Chapter 19. Axis C core APIs

This chapter summarizes the core (i.e. most commonly used) Axis C functions. For a complete list of the Axis functions, view the following file in a Web browser: /QIBM/ProdData/OS/WebServices/V1/client/docs/api/index.html.

The majority of the Axis C APIs operate on pointer objects. All objects in AXIS are mapped to AXISCHANDLE, which is simply a type definition to a pointer type. Thus, as you code your application, you should keep in mind what the handle is so you know what APIs you can use against the handle.

For example, when generate a stub instance, you will receive a stub handle. This stub handle can then be used by Axis APIs that have a name starting with "axiscStub". Similarly, if you create a header block object, the header block pointer can be used by Axis APIs that have a name starting with "axiscHeaderBlock".

**Axis C APIs**

Contains methods that affect the Axis client engine, such as functions to initialize and terminate the Axis runtime, and functions to free allocated memory resources. The Axis APIs are defined in include file <install_dir>/include/axis/Axis.h.

The following table lists the Axis APIs.

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axiscAxisInitialize()</td>
<td>Initializes the Axis runtime.</td>
</tr>
<tr>
<td>axiscAxisTerminate()</td>
<td>Terminates the Axis runtime.</td>
</tr>
<tr>
<td>axiscAxisDelete()</td>
<td>Deletes storage allocated by the Axis engine.</td>
</tr>
<tr>
<td>axiscAxisStartTrace()</td>
<td>Starts Axis logging.</td>
</tr>
<tr>
<td>axiscAxisStopTrace()</td>
<td>Stops Axis logging.</td>
</tr>
<tr>
<td>axiscAxisWriteTrace()</td>
<td>Writes trace data to Axis log.</td>
</tr>
</tbody>
</table>

**axiscAxisInitialize()**

```
void axiscAxisInitialize(AxiscBool bIsServer)
```

Initializes the Axis runtime. Creating a stub also initializes the Axis runtime and deleting the stub terminates it. So simple applications that only ever use one stub at a time do not need to call these methods. More complicated applications that initialize multiple stubs, use them and delete them later, should initialize Axis at the start of their application using axiscAxisInitialize() and terminate Axis at the very end of their application with axiscAxisTerminate(). Applications that use Axis in multiple threads should also call axiscAxisInitialize() and axiscAxisTerminate().

**Parameters**

- **bIsServer**: Integer boolean flag that must be set to 0.

**Example**
The following example initializes the Axis client engine.

```c
axiscAxisInitialize();
```

**axiscAxisTerminate()**

```c
void axiscAxisTerminate()
```

Terminates the Axis runtime.

**Example**

The following example terminates the Axis client engine.

```c
axiscAxisTerminate();
```

**axiscAxisDelete()**

```c
void axiscAxisDelete(void* pValue,
                     AXISC_XSDTYPE type)
```

Deletes storage allocated by the Axis engine.

**Parameters**

- **pValue**  Pointer to storage that is to be deleted.
- **type**  The type of storage to be deleted. The AXISC_XSDTYPE type is an enumerator defined in `<install_dir>/include/axis/TypeMapping.h`.

**Example**

The following example deletes a pointer that was dynamically allocated by the Axis engine and that is used to store data with a type of xsd:int.

```c
axiscAxisDelete(ptr, XSDC_INT);
```

**axiscAxisStartTrace()**

```c
int axiscAxisStartTrace(const char* logFilePath,
                        const char *logFilter)
```

Starts Axis logging. This must be done prior to any activity in order to propagate logging attributes to parser and transport. If there are active transports and parsers, you will not get trace records other than those associated with the engine and newly instantiated transports and parsers.

A typical trace record will look like the following (following are entry/exit trace records):

```
```

A trace record includes a timestamp, a thread ID, the component that is doing the trace, a one character field indicating Trace type:

- `>` (entry)
- `<` (exit)
- `X` (exception)
- `D` (debug)
and the method/function name. After which there will be additional trace data. When tracing is enabled, you will know exactly where an exception is being thrown from. A typical trace record for when an exception is thrown is as follows:

```
Line=1851: File=/home/amra/axis/L1.1.0/src/ws-axis/c/src/transport/axis3/HTTPTransport.cpp:
HTTPTransportException - SERVER_TRANSPORT_HTTP_EXCEPTION:
Server sent HTTP error: 'Not Found'
```

Request and response messages can be traced by enabling transport trace. Here is an example of a transport trace:

```
POST /axis HTTP/1.1
Host: 127.0.0.1:13260
Content-Type: text/xml; charset=UTF-8
SOAPAction: ""
Content-Length: 393

<?xml version='1.0' encoding='utf-8' ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Body>
<ns1:div xmlns:ns1="http://soapinterop.org/wsdl">
<ns1:arg_0_0>10</ns1:arg_0_0>
<ns1:arg_1_0>5</ns1:arg_1_0>
</ns1:div>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
HTTP/1.1 404 Not Found
Server: WebSphere Application Server/5.1
Content-Type: text/html;charset=UTF-8
Content-Language: en-GB
Transfer-Encoding: chunked
21
Error 404: File not found: null 0
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logFilePath</td>
<td>Pointer to null-terminated character string representing the path to where trace records are written to.</td>
</tr>
<tr>
<td>logFilter</td>
<td>Pointer to null-terminated character string representing the trace filter. The string filter is a semicolon delimited string of possible filters. Possible filters include: stub - show trace records generated by stubs, engine - show trace records generated by engine, parser - show trace records generated by parser, transport - show trace records generated by transport, noEntryExit - do not show entry/exit trace records</td>
</tr>
</tbody>
</table>

The default filter is "stub;engine;parser;transport". Specifying a null pointer or a null string is equivalent to requesting the default filter.
Returns

Zero if the method call is successful; otherwise -1 is returned.

Example

See example for the “axiscAxisWriteTrace()” method.

axiscAxisStopTrace()

```c
void axiscAxisStopTrace()
```

Stops Axis logging. This should be done as the last step when everything has been cleaned up. Otherwise, active transports and parsers will continue to trace.

Example

See example for the “axiscAxisWriteTrace()” method.

axiscAxisWriteTrace()

```c
void axiscAxisWriteTrace(AXISC_TRACE_TYPE type,
                        const char* funcName,
                        const char* format,
                        ...)
```

Writes specified data to the Axis log file.

Parameters

type The trace type. AXISC_TRACE_TYPE is an enumerator that can be set to one of the following values:
- AXISC_TRACE_TYPE_ENTRY=0
- AXISC_TRACE_TYPE_EXIT=1
- AXISC_TRACE_TYPE_EXCEPTION=2
- AXISC_TRACE_TYPE_DEBUG=3

funcName Pointer to null-terminated character string representing class method or function for which trace record is being written.

format Pointer to null-terminated character string representing the format as defined for the printf() function.

... Variable number of parameters, the number of which is dependent on the specified format parameter.

Example

The following example writes an application-defined trace record to the Axis log.

```c
#include "axis/Axis.h"
#include "StockQuote.h"
#include <stdio.h>

int main()
{
    char * pszStockName;
    xsdc__float fQuoteDollars;
    AXISCHANDLE pwsStockQuote;
    char * pszEndpoint =
        "http://localhost:40001/StockQuote/services/urn:xmltoday-delayed-quotes";
```
pwsStockQuote = get_StockQuote_stub( pszEndpoint);
if (NULL == pwsStockQuote)
    return -1;

// Call the 'getQuote' method
pszStockName = "XXX";
fQuoteDollars = getQuote(pwsStockQuote, pszStockName);

// Output the quote. If the stock name is unknown, then getQuote() will
// return -1. If name was recognized by the server a value is returned.
if ( fQuoteDollars != -1)
    printf("The stock quote for %s is $%f\n", pszStockName, fQuoteDollars);
else
    printf("There is no stock quote for %s\n", pszStockName);

// Delete the web service.
destroy_StockQuote_stub(pwsStockQuote);

axiscAxisStopTrace();

// Exit.
return 0;

---

**Stub C APIs**

The stub object is returned by the code generated by `wsdl2ws.sh` tool when you use the service interface function to create a stub. This object acts as the interface between the client application and the Axis engine. The stub C APIs are defined in include file `<install_dir>/include/axis/client/Stub.h`.

The following table lists the most commonly used functions.

*Table 23. Stub C functions*

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>axiscStubSetTransportProperty()</code></td>
<td>Sets transport properties (e.g. HTTP headers).</td>
</tr>
<tr>
<td><code>axiscStubGetTransportProperty()</code></td>
<td>Gets transport properties (e.g. HTTP headers).</td>
</tr>
<tr>
<td><code>axiscStubSetTransportConnectTimeout()</code></td>
<td>Sets the transport connect timeout.</td>
</tr>
<tr>
<td><code>axiscStubSetTransportTimeout()</code></td>
<td>Sets the transport timeout.</td>
</tr>
<tr>
<td><code>axiscStubSetTransportAutoRedirect()</code></td>
<td>Sets whether transport is to automatically handle HTTP redirects.</td>
</tr>
<tr>
<td><code>axiscStubCreateSOAPHeaderBlock()</code></td>
<td>Creates and adds a SOAP header block to the stub.</td>
</tr>
<tr>
<td><code>axiscStubAddNamespaceToSOAPHeader()</code></td>
<td>Adds a namespace to the SOAP Header element.</td>
</tr>
<tr>
<td><code>axiscStubClearSOAPHeaderNamespaces()</code></td>
<td>Clears all namespaces from the SOAP Header element.</td>
</tr>
<tr>
<td><code>axiscStubAddAttributeToSOAPHeader()</code></td>
<td>Adds an attribute to the SOAP Header element.</td>
</tr>
<tr>
<td><code>axiscStubClearSOAPHeaderAttributes()</code></td>
<td>Clears all attributes from the SOAP Header element.</td>
</tr>
<tr>
<td><code>axiscStubAddNamespaceToSOAPBody()</code></td>
<td>Adds a namespace to SOAP Body element.</td>
</tr>
<tr>
<td><code>axiscStubClearSOAPBodyNamespaces()</code></td>
<td>Clears all namespaces from the SOAP Body element.</td>
</tr>
<tr>
<td><code>axiscStubAddAttributeToSOAPBody()</code></td>
<td>Adds an attribute to SOAP Body element.</td>
</tr>
<tr>
<td><code>axiscStubClearSOAPBodyAttributes()</code></td>
<td>Clears all attributes from the SOAP Body element.</td>
</tr>
<tr>
<td>Class methods</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>axiscStubSetMaintainSession()</td>
<td>Sets whether to maintain session with service or not.</td>
</tr>
<tr>
<td>axiscStubSetPassword()</td>
<td>Sets the password to be used for basic authentication.</td>
</tr>
<tr>
<td>axiscStubSetUsername()</td>
<td>Sets the user name to be used for basic authentication.</td>
</tr>
<tr>
<td>axiscStubSetProxy()</td>
<td>Sets the proxy server and port for transport.</td>
</tr>
<tr>
<td>axiscStubSetProxyPassword()</td>
<td>Sets the password to be used for proxy authentication.</td>
</tr>
<tr>
<td>axiscStubSetProxyUsername()</td>
<td>Sets the user name to be used for proxy authentication.</td>
</tr>
<tr>
<td>axiscStubSetSecure()</td>
<td>Sets SSL configuration properties.</td>
</tr>
</tbody>
</table>

**axiscStubSetTransportProperty()**

```c
void axiscStubSetTransportProperty(AXISCHANDLE stub,
 const char * pcKey,
 const char * pcValue)
```

Sets the specified transport property. Calling this function with the same key multiple times will result in the property being set to the last value.

**Parameters**

- `stub` Pointer to stub object.
- `pcKey` Pointer to null-terminated character string representing the transport property to set.
- `pcValue` Pointer to null-terminated character string representing the value of the transport property corresponding to `pcKey`.

**Example**

The following example sets the cookie HTTP header.

```c
axiscStubSetTransportProperty(stub, "Cookie", "sessiontoken=123345456");
```

**axiscStubGetTransportProperty()**

```c
const char * axiscStubGetTransportProperty(AXISCHANDLE stub,
 const char * pcKey,
 AxiscBool response)
```

Searches for the transport property with the specified key. The method returns NULL if the property is not found.

**Parameters**

- `stub` Pointer to stub object.
- `pcKey` Pointer to null-terminated character string representing the transport property to retrieve.
- `response` Integer flag, when set to 1, searches the response message for the property; and when set to 0 searches the request message.

**Returns**

The value of the property or NULL if it was not found.
Example

The following example retrieves the HTTP cookie header from the response message.

```c
const char *cookie = axiscStubGetTransportProperty(stub, "Cookie", 1);
```

### `axiscStubSetTransportConnectTimeout()`

```c
void axiscStubSetTransportConnectTimeout(AXISCHANDLE stub,
                                        long iTimeout)
```

Sets a specified timeout value, in seconds, to be used when attempting to connect to the server hosting
the Web service. If the timeout expires before establishing a connection to the server, an Axis exception is
thrown. A timeout of zero, which is the default, is interpreted as an infinite timeout.

**Parameters**

- **stub** Pointer to stub object.
- **iTimeout** An integer that specifies the connect timeout value in seconds.

### Example

The following example set the transport connect timeout to 10 seconds.

```c
axiscStubSetTransportConnectTimeout(stub, 10);
```

### `axiscStubSetTransportTimeout()`

```c
void axiscStubSetTransportTimeout(AXISCHANDLE stub,
                                  long iTimeout)
```

Sets a specified timeout value, in seconds, to be used when waiting for a response from the Web service.
If the timeout expires before receiving a Web service response, an Axis exception is thrown. A timeout of
zero, which is the default, is interpreted as an infinite timeout.

**Parameters**

- **stub** Pointer to stub object.
- **iTimeout** An integer that specifies the receive timeout value in seconds.

### Example

The following example set the transport timeout to 10 seconds.

```c
axiscStubSetTransportTimeout(stub, 10);
```

### `axiscStubSetTransportAutoRedirect()`

```c
void axiscStubSetTransportAutoRedirect(AXISCHANDLE stub,
                                       AxiscBool redirectFlag,
                                       int maxCount)
```

Sets whether the transport is to automatically handle HTTP redirects. By default, redirects are not
handled by the transport. If enabled, auto-redirect will only occur when going from http to http or https
to https.
Parameters

stub        Pointer to stub object.
redirectFlag  Integer boolean flag. When set to zero (the default), automatic redirects will not occur. When set to a
non-zero value, automatic redirects will occur.
maxCount   How many redirects to follow. Default is 1. A value less than 1 is the same as setting redirectFlag
to zero.

Example

The following example enables automatic redirects.
axiscStubSetTransportAutoRedirect(stub, 1, 10);

axiscStubCreateSOAPHeaderBlock()

AXISHANDLE axiscStubCreateSOAPHeaderBlock(AXISHANDLE stub,
AxisChar * pElemName,
AxisChar * pNamespace,
AxisChar * pPrefix)

Creates and adds a SOAP header block (i.e. SOAP header). The returned AXISHANDLE is a pointer that
represents a header block object and must be used to add the elements and values of the SOAP header
block.

Parameters

stub        Pointer to stub object.
pElemName   Pointer to null-terminated character string representing the element tag name of the SOAP header.
pNamespace Pointer to null-terminated character string representing the URI of namespace.
pPrefix     Pointer to null-terminated character string representing the prefix that will be associated with the
specified namespace.

Returns

Pointer to SOAP header block object. The ownership of the memory allocated for the object is owned by
the stub.

Example

The following example will generate a Security element that is inserted in the SOAP message as a SOAP
header:
#include "axis/Axis.h"
#include "axis/IHeaderBlock.h"
#include "axis/BasicNode.h"
#include "StockQuote.h"
#include <stdio.h>

int main()
{

AXISHANDLE phb;
AXISHANDLE Bnode1;
AXISHANDLE Bnode2;
AXISHANDLE Bnode3;
AXISHANDLE stub;

// generate node wsse:Security element, declaring namespaces for wsse and wsu
phb = axiscStubCreateSOAPHeaderBlock(stub,
   "Security",
   "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd",
   "wsse");
axiscHeaderBlockCreateNamespaceDeclINamespace(phb,
   "wsu",
   "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd");
axiscHeaderBlockCreateStdAttribute(phb, MUST_UNDERSTAND_TRUE, SOAP_VER_1_1);

// Generate node wsse:UsernameToken as child node of wsse:Security
Bnode1=axiscHeaderBlockCreateChildBasicNode(phb,
   ELEMENT_NODE, "UsernameToken", "wsse", NULL, NULL);
axiscBasicNodeCreateAttribute(Bnode1,
   "Id", "wsu", NULL, "UsernameToken-12345678");
axiscHeaderBlockAddChild(phb,Bnode1);

// Generate node wsse:Username as child node of wsse:UsernameToken
// and the associated character node
Bnode2=axiscHeaderBlockCreateChildBasicNode(phb,
   ELEMENT_NODE, "UserName", "wsse", NULL, NULL);
axiscBasicNodeAddChild(Bnode1,Bnode2);
Bnode3=axiscHeaderBlockCreateChildBasicNode(phb,
   CHARACTER_NODE, NULL, NULL, NULL, "admin");
axiscBasicNodeAddChild(Bnode2,Bnode3);

// Generate node wsse:Password as child node of wsse:UsernameToken
// and the associated character node
Bnode2=axiscHeaderBlockCreateChildBasicNode(phb,
   ELEMENT_NODE, "Password", "wsse", NULL, NULL);
axiscBasicNodeCreateAttribute(Bnode2,
   "Type", NULL, NULL,
   "http://docs.oasis-open.org/wss/2004/01/
   "oasis-200401-wss-password-token-profile-1.0#PasswordText");
axiscBasicNodeAddChild(Bnode1,Bnode2);
Bnode3=axiscHeaderBlockCreateChildBasicNode(phb,
   CHARACTER_NODE, NULL, NULL, NULL, "admin");
axiscBasicNodeAddChild(Bnode2,Bnode3);

// Perform Web service operation
.
.
.
// Delete the web service.
destroy_StockQuote_stub(stub);

// Exit.
return 0;
}

Here is what the Security element looks like when sent out as part of the SOAP request:

<wsse:Security
 xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
 SOAP-ENV:mustUnderstand="1"
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
<wsse:UsernameToken wsu:Id="UsernameToken-12345678"/>
<wsse:UserName>admin</wsse:UserName>
<wsse:Password
    Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">
    admin
</wsse:Password>
</wsse:UsernameToken>

axiscStubAddNamespaceToSOAPHeader()

```c
void axiscStubAddNamespaceToSOAPHeader AXISCHANDLE stub,
    const AxiscChar * pUri,
    const AxiscChar * pPrefix)
```

Adds a namespace to the SOAP Header element.

**Parameters**

- **stub** Pointer to stub object.
- **pUri** Pointer to null-terminated character string representing the namespace URI.
- **pPrefix** Pointer to null-terminated character string representing the prefix that will be associated with the specified namespace.

**Example**

The following example will generate a SOAP Header element that declares the namespace `myNS` with a value of `http://www.myns.com/MyNS` and adds an attribute to the SOAP Header element using the declared namespace:

```c
#include "axis/Axis.h"
#include "StockQuote.h"
#include <stdio.h>

int main()
{
    
    AXISHANDLE stub;
    stub = get_StockQuote_stub("http://9.10.164:8088/StockQuote");

    // Declare namespace for myNS in SOAP Header element
    axiscStubAddNamespaceToSOAPHeader(stub, "http://www.myns.com/MyNS", "myNS");

    // Add attribute to SOAP Header element using declared namespace myNS
    axiscStubAddAttributeToSOAPHeader(stub, "myAttr", "myNS", "myAttrValue");

    // Perform Web service operation
    
    // Delete the web service.
    destroy_StockQuote_stub(stub);

    // Exit.
    return 0;
}
```

Here is a partial SOAP request showing the added namespace declaration:
axiscStubClearSOAPHeaderNamespaces()

```c
void axiscStubClearSOAPHeaderNamespaces(AxisCHandle stub)
```

Clears all namespaces from the SOAP Header element.

**Parameters**

- `stub` Pointer to stub object.

**Example**

Following example clears out all namespaces from the SOAP Header element.

```c
axiscStubClearSOAPHeaderNamespaces(stub);
```

axiscStubAddAttributeToSOAPHeader()

```c
void axiscStubAddAttributeToSOAPHeader(AxisCHandle stub,
                                       const AxiscChar * pLocalname,
                                       const AxiscChar * pPrefix,
                                       const AxiscChar * pValue)
```

Adds an attribute to the SOAP Header element.

**Parameters**

- `stub` Pointer to stub object.
- `pLocalname` Pointer to null-terminated character string representing the attribute name.
- `pPrefix` Pointer to null-terminated character string representing the prefix that will be associated with the specified attribute.
- `pValue` Pointer to null-terminated character string representing the value of the attribute.

**Example**

See example for "axiscStubAddNamespaceToSOAPHeader()" on page 148.

axiscStubClearSOAPHeaderAttributes()

```c
void axiscStubClearSOAPHeaderAttributes(AxisCHandle stub)
```

Clears all attributes from the SOAP Header element.

**Parameters**

- `stub` Pointer to stub object.

**Example**

Following example clears out all attributes from the SOAP Header element.

```c
axiscStubClearSOAPHeaderAttributes(stub);
```
axiscStubAddNamespaceToSOAPBody()

```c
void axiscStubAddNamespaceToSOAPBody(AXISCHANDLE stub,
    const AxisChar * pUri,
    const AxisChar * pPrefix)
```

Adds a namespace to the SOAP Body element.

**Parameters**

- `stub` Pointer to stub object.
- `pUri` Pointer to null-terminated character string representing the namespace URI.
- `pPrefix` Pointer to null-terminated character string representing the prefix that will be associated with the specified namespace.

**Example**

The following example will generate a SOAP Body element that declares the namespace `myNS` with a value of `http://www.myns.com/MyNS` and adds an attribute to the SOAP Body element using the declared namespace:

```c
#include "axis/Axis.h"
#include "StockQuote.h"
#include <stdio.h>

int main()
{
    
    AXISHANDLE stub;

    // Declare namespace for myNS in SOAP Header element
    axiscStubAddNamespaceToSOAPBody(stub,"http://www.myns.com/MyNS", "myNS");

    // Add attribute to SOAP Header element using declared namespace myNS
    axiscStubAddAttributeToSOAPBody(stub, "myAttr", "myNS", "myAttrValue");

    // Perform Web service operation
    
    // Delete the web service.
    destroy_StockQuote_stub(stub);

    // Exit.
    return 0;
}
```

Here is a partial SOAP request showing the added namespace declaration:

```xml
<SOAP-ENV:Body xmlns:myNS="http://www.myns.com/MyNS" myNS:myAttr="myAttrValue">
```

axiscStubClearSOAPBodyNamespaces()
Clears all namespaces from the SOAP Body element.

**Parameters**

stub Pointer to stub object.

**Example**

Following example clears out all namespaces from the SOAP Body element.

```c
axiscStubClearSOAPBodyNamespaces(stub);
```

### axiscStubAddAttributeToSOAPBody()

```c
void axiscStubAddAttributeToSOAPBody(AXISCHANDLE stub,
const AxiscChar * pLocalname,
const AxiscChar * pPrefix,
const AxiscChar * pValue)
```

Adds an attribute to the SOAP Body element.

**Parameters**

stub Pointer to stub object.
pLocalname Pointer to null-terminated character string representing the attribute name.
pPrefix Pointer to null-terminated character string representing the prefix that will be associated with the specified attribute.
pValue Pointer to null-terminated character string representing the value of the attribute.

**Example**

See example for "axiscStubAddNamespaceToSOAPBody()" on page 150.

### axiscStubClearSOAPBodyAttributes()

```c
void axiscStubClearSOAPBodyAttributes(AXISCHANDLE stub)
```

Clears all attributes from the SOAP Body element.

**Parameters**

stub Pointer to stub object.

**Example**

Following example clears out all attributes from the SOAP Body element.

```c
axiscStubClearSOAPBodyAttributes(stub);
```

### axiscStubSetMaintainSession()

```c
void axiscStubSetMaintainSession(AXISCHANDLE stub,
AxiscBool bSession)
```

Sets whether to maintain session with service or not.
Parameters

stub Pointer to stub object.
bSession Integer flag, when set to 1, indicates that session should be maintained with Web service. When set to 0 the session will not be maintained.

Example

Following example indicates to the Axis engine that session to Web service should be maintained.
axiscStubSetMaintainSession(stub, 1);

axiscStubSetPassword()

void axiscStubSetPassword(AXISCHANDLE stub,
                          const char * pPassword)

Sets the password to be used for HTTP basic authentication.

Parameters

stub Pointer to stub object.
pPassword Pointer to null-terminated character string representing the password.

Example

Following example sets HTTP basic authentication password.
axiscStubSetPassword(stub, "password1");

axiscStubSetUsername()

void axiscStubSetUsername(AXISCHANDLE stub,
                          const char * pUsername)

Sets the username to be used for HTTP basic authentication.

Parameters

stub Pointer to stub object.
pUsername Pointer to null-terminated character string representing the username.

Example

Following example sets HTTP basic authentication username.
axiscStubSetUsername(stub, "user1");

axiscStubSetProxy()

void axiscStubSetProxy(AXISCHANDLE stub,
                       const char * pProxyHost,
                       unsigned int uiProxyPort)

Sets the proxy server and port.
Parameters

stub    Pointer to stub object.
pcProxyHost    Pointer to null-terminated character string representing the host name of proxy server.
uiProxyPort    The port the proxy server listening on.

Example

Following example sets proxy host and port information.
axiscStubSetProxy(stub, "proxyserver", 40001);

axiscStubSetProxyPassword()

```c
void axiscStubSetProxyPassword(AXISCHANDLE stub,
                          const char * pPassword)
```

Sets password to be used for proxy authentication.

Parameters

stub    Pointer to stub object.
pPassword    Pointer to null-terminated character string representing the password.

Example

Following example sets password for proxy authentication.
axiscStubSetProxyPassword(stub, "proxypwd1");

axiscStubSetProxyUsername()

```c
void axiscStubSetProxyUsername(AXISCHANDLE stub,
                          const char * pUsername)
```

Sets the username to be used for Proxy authentication.

Parameters

stub    Pointer to stub object.
pUsername    Pointer to null-terminated character string representing the username.

Example

Following example sets username for proxy authentication.
axiscStubSetProxyUsername(stub, "proxyusr1");

axiscStubSetSecure()

```c
void axiscStubSetSecure(AXISCHANDLE stub,
                     char * pKeyRingFile,
                     ...)
```

Sets SSL configuration properties.
Parameters

stub Pointer to stub object.
pKeyRingFile Pointer to null-terminated character string representing the certificate store file to be used for the secure session or SSL environment.
pKeyRingPS (optional) Pointer to null-terminated character string representing the password for the certificate store file to be used for the secure session or SSL environment. If the parameter is not passed or is set to the null string, the internal stash file associated with the user profile that is being used to run the application is used. To specify any of the subsequent optional parameters, you must pass a value for this parameter.
pKeyRingLbl (optional) Pointer to null-terminated character string representing the certificate label associated with the certificate in the certificate store to be used for the secure session or SSL environment. If the parameter is not passed or is set to the null string, the default certificate label in the specified certificate store file is used for the SSL environment. To specify any of the subsequent optional parameters, you must pass a value for this parameter.
pV2Cipher (optional) Pointer to null-terminated character string representing the list of SSL Version 2 ciphers to be used for the secure session or the SSL environment. Specifying "NONE" for this parameter will disable SSL Version 2 ciphers. To specify any of the subsequent optional parameters, you must pass a value for this parameter.
pV3Cipher (optional) Pointer to null-terminated character string representing the list of SSL Version 3/TLS Version 1 ciphers to be used for the secure session or the SSL environment. Specifying "NONE" for this parameter will disable SSL Version 3 ciphers. To specify any of the subsequent optional parameters, you must pass a value for this parameter.
pTLS_cipher (optional) Pointer to null-terminated character string indicating whether to enable or disable the TLS Version 1 ciphers. A value of "NONE" will disable the ciphers; any other value will enable the ciphers. By default, the TLS Version 1 ciphers are enabled.

Usage notes

1. The Web services client for ILE supports secure sessions by using the Global Secure ToolKit (GSKit) APIs. For the latest information on ciphers, see the gsk_attribute_set_buffer() API usage notes section at the IBM i Information Center Web site - [http://www.ibm.com/systems/i/infocenter/](http://www.ibm.com/systems/i/infocenter/)

2. The following GSK_V3_CIPHER_SPECS values are the SSL Version 3 ciphers and the TLS Version 1 ciphers supported:

   01 = *RSA_NULL_MD5
   02 = *RSA_NULL_SHA
   03 = *RSA_EXPORT_RC4_40_MD5
   04 = *RSA_RC4_128_MD5
   05 = *RSA_RC4_128_SHA
   06 = *RSA_EXPORT_RC2_CBC_40_MD5
   09 = *RSA_DES_CBC_SHA
   0A = *RSA_3DES_EDE_CBC_SHA
   2F = *RSA_AES_128_CBC_SHA (TLS Version 1 only)
   35 = *RSA_AES_256_CBC_SHA (TLS Version 1 only)

3. SSL Version 2 support is disabled IBM i 6.1 and later releases when the operating system is installed resulting in no SSL Version 2 ciphers being supported. If SSL Version 2 is enabled (not recommended), the following GSK_V2_CIPHER_SPECS values are the SSL Version 2 ciphers that would be supported if shipped supported cipher list has not been altered.

   1 = *RSA_RC4_128_MD5
   2 = *RSA_EXPORT_RC4_40_MD5
   4 = *RSA_EXPORT_RC2_CBC_40_MD5

The following GSK_V2_CIPHER_SPECS values are the SSL Version 2 ciphers potentially supported if an administrator later enables SSL Version 2:

   1 = *RSA_RC4_128_MD5
   2 = *RSA_EXPORT_RC4_40_MD5
   3 = *RSA_RC2_CBC_128_MD5
   4 = *RSA_EXPORT_RC2_CBC_40_MD5
   6 = *RSA_DES_CBC_MD5
   7 = *RSA_3DES_EDE_CBC_MD5
The following C example shows a sample client application that configures security information before calling a web service. To configure the secure setting within your own application, add the code shown in bold in this example.

Below is a C example of how to set security information:

```c
int main() {
    // URL for secure communication. The localhost may require
    // a port number, i.e. localhost:80
    char *pszSecureURL = "https://localhost/Test/services/TestPort";

    // Load instances of the service with secure URL settings.
    AXISCHANDLE serviceSecure = get_ITestService_stub(pszSecureURL);

    // Initialise the secure settings for the secure service.
    axiscStubSetSecure(serviceSecure, "<path to KeyRing.kbd>",
                        "<password to stash or NULL string>", 
                        "<label>", "NONE", "05", "NONE");

    // Remainder of application
    .
    .
    // End of application
    destroy_ITestService_stub(serviceSecure);
    return 0;
}
```

---

**Header block C APIs**

The header block object represents a SOAP header block object. The header block C APIs are defined in include file `<install_dir>/include/axis/IHeaderBlock.h`.

The following table lists the most commonly used methods.

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>axiscHeaderBlockCreateNamespaceDeclINamespace()</code></td>
<td>Creates an attribute and adds it to the header block as a namespace.</td>
</tr>
<tr>
<td><code>axiscHeaderBlockCreateStdAttribute()</code></td>
<td>Creates a standard header block attribute.</td>
</tr>
<tr>
<td><code>axiscHeaderBlockCreateChildBasicNode()</code></td>
<td>Creates a child node depending on the given node type.</td>
</tr>
<tr>
<td><code>axiscHeaderBlockAddChild()</code></td>
<td>Adds a child node to the header block.</td>
</tr>
</tbody>
</table>

### `axiscHeaderBlockCreateNamespaceDeclINamespace()`

`AXISCHANDLE axiscHeaderBlockCreateNamespaceDeclINamespace(AXISCHANDLE headerBlock, const AxisChar *pPrefix, const AxisChar *pNamespace)`

Creates an attribute and adds it to the header block as a namespace.

**Parameters**

- `headerBlock` Pointer to header block object.
- `pPrefix` Pointer to null-terminated character string representing the prefix that will be associated with the specified namespace.
pNamespace  Pointer to null-terminated character string representing the URI of namespace.

Returns

Pointer to namespace object. The ownership of the memory allocated for the object is owned by the stub.

Example

See example for "axiscStubCreateSOAPHeaderBlock()" on page 146.

axiscHeaderBlockCreateStdAttribute()

AXISCHANDLE axiscHeaderBlockCreateStdAttribute(AXISCHANDLE headerBlock,
AXISC_HEADER_BLOCK_STD_ATTR_TYPE eAttribute,
AXISC_SOAP_VERSION eSOAPVers)

Creates and adds a standard SOAP header block attribute.

Parameters

headerBlock  Pointer to header block object.

Parameters

eAttribute  Enumerator indicating which of the following attributes are to be set:

ACTOR : Creates actor attribute to point to next.
MUST_UNDERSTAND_TRUE : Creates the mustUnderstand attribute set to "1".
MUST_UNDERSTAND_FALSE: Creates the mustUnderstand attribute set to "0".

eSOAPVers  Enumerator indicating the SOAP version. This parameter must always be set to:

SOAP_VER_1_1 : SOAP version 1.1.

The enumerator AXISC_SOAP_VERSION is defined in <install_dir>/include/axis/SoapEnvVersions.h

Returns

Pointer to attribute object. The ownership of the memory allocated for the object is owned by the stub.

Example

See example for "axiscStubCreateSOAPHeaderBlock()" on page 146.

axiscHeaderBlockCreateChildBasicNode()

AXISCHANDLE axiscHeaderBlockCreateChildBasicNode(AXISCHANDLE headerBlock,
AXISC_NODE_TYPE eNodeType,
AxisChar *pElemName,
AxisChar *pPrefix,
AxisChar *pNamespace,
AxisChar* pachValue)

Creates an instance of a basic node of the specified type.

Parameters

headerBlock  Pointer to header block object.
eNodeType Enumerator indicating one of the following node types:

- ELEMENT_NODE=1
- CHARACTER_NODE=2

The enumerator AXISC_NODE_TYPE is defined in `<install_dir>/include/axis/BasicNode.h`

pElemName Pointer to null-terminated character string representing the element tag name of the node. This parameter is ignored for CHARACTER_NODE node types.

pPrefix Pointer to null-terminated character string representing the prefix that will be associated with the specified namespace. This parameter is ignored for CHARACTER_NODE node types.

pNamespace Pointer to null-terminated character string representing the URI of namespace. This parameter is ignored for CHARACTER_NODE node types.

pachValue Pointer to null-terminated character string representing the value of the node. This parameter is ignored for ELEMENT_NODE node types.

Returns

Pointer to a basic node object. The ownership of the memory allocated for the object is owned by the caller until the node is added to the header block.

Example

See example for "axiscStubCreateSOAPHeaderBlock()" on page 146.

axiscHeaderBlockAddChild()

```c
int axiscHeaderBlockAddChild(AxisCHandle headerBlock,
                             AXISCHANDLE pBasicNode)
```

Adds a child node to the SOAP header block.

Parameters

- headerBlock Pointer to header block object.
- pBasicNode Pointer to basic node object to be added to SOAP header block.

Returns

Zero if node was added successfully; otherwise -1 is returned.

Example

See example for "axiscStubCreateSOAPHeaderBlock()" on page 146.

Basic node C APIs

The basic node object is the base object that is used to construct the various nodes of a SOAP header. The basic node C APIs are defined in include file `<install_dir>/include/axis/BasicNode.h`.

The following table lists the most commonly used methods.

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axiscBasicNodeCreateAttribute()</td>
<td>Creates an attribute and adds it to this basic node.</td>
</tr>
</tbody>
</table>
Table 25. Basic node C functions (continued)

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axiscBasicNodeAddChild()</td>
<td>Adds a child node to the basic node.</td>
</tr>
</tbody>
</table>

**axiscBasicNodeCreateAttribute()**

```c
AXISHANDLE axiscBasicNodeCreateAttribute(AXISHANDLE basicNode,
    const AxisChar* pAttrName,
    const AxisChar* pPrefix,
    const AxisChar* pNamespace,
    const AxisChar* pValue)
```

Creates an attribute and adds it to the basic node.

**Parameters**

- `basicNode`: Pointer to basic node object.
- `pAttrName`: Pointer to null-terminated character string representing the attribute name.
- `pPrefix`: Pointer to null-terminated character string representing the attribute prefix that will be associated with the specified namespace.
- `pNamespace`: Pointer to null-terminated character string representing the URI of attribute namespace.
- `pValue`: The value of the attribute.

**Returns**

Pointer to created attribute object. The ownership of the memory allocated for the object is owned by the stub.

**Example**

See example for "axiscStubCreateSOAPHeaderBlock()" on page 146.

**axiscBasicNodeAddChild()**

```c
int axiscBasicNodeAddChild(AXISHANDLE basicNode,
    AXISHANDLE pBasicNode)
```

Adds a basic node as a child node to another basic node.

**Parameters**

- `basicNode`: Pointer to basic node object.
- `pBasicNode`: Pointer to basic node to be added as a child node.

**Returns**

Zero if node was added successfully; otherwise, -1 is returned.

**Example**

See example for "axiscStubCreateSOAPHeaderBlock()" on page 146.
Part 5. Using RPG stubs

This part of the document provides details regarding all things related RPG stub programming. If you have no interest in RPG stub programming, you should skip this part of the document.

- [Chapter 20, “WSDL and XML to RPG mappings,” on page 161](#) describes in detail how WSDL and XML constructs are mapped to RPG constructs.
- [Chapter 21, “Developing a Web services client application using RPG stubs,” on page 173](#) is a step-by-step description on how to develop a client application using a RPG stub.
- [Chapter 22, “RPG stub programming considerations,” on page 177](#) discusses various programming topics relating to using the RPG stubs, including how to set up an SSL connection.
- [Chapter 23, “Troubleshooting RPG client stubs,” on page 183](#) describes how to troubleshoot command line tool and runtime problems.
Chapter 20. WSDL and XML to RPG mappings

The `wsdl2ws.sh` command tool can generate RPG stub code.

This chapter will describe the mappings from WSDL and XML Schema types to RPG language constructs. But first, it should be noted that the RPG stub code is built on top of the C stub code, so anytime you generate RPG stub code, C stub code will also get generated. You should keep this in mind when reading about the RPG stub support in Web services client for ILE.

**XML names**

RPG identifiers are generated from the corresponding C identifiers. For information on how C identifiers are generated, see "Mapping XML names to C identifiers" on page 115.

**XML schema to RPG type mapping**

Table 26 specifies the RPG mapping for each built-in simple. The table shows the XML Schema type and the corresponding the Axis RPG type (column 2).

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Actual RPG type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:byte</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_byte...</td>
</tr>
<tr>
<td></td>
<td>D isNil</td>
</tr>
<tr>
<td></td>
<td>D value</td>
</tr>
<tr>
<td></td>
<td>DS qualified based(Template)</td>
</tr>
<tr>
<td>xsd:decimal</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_decimal...</td>
</tr>
<tr>
<td></td>
<td>D isNil</td>
</tr>
<tr>
<td></td>
<td>D value</td>
</tr>
<tr>
<td></td>
<td>DS qualified based(Template)</td>
</tr>
<tr>
<td>xsd:double</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_double...</td>
</tr>
<tr>
<td></td>
<td>D isNil</td>
</tr>
<tr>
<td></td>
<td>D value</td>
</tr>
<tr>
<td></td>
<td>DS qualified based(Template)</td>
</tr>
<tr>
<td>xsd:float</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_float...</td>
</tr>
<tr>
<td></td>
<td>D isNil</td>
</tr>
<tr>
<td></td>
<td>D value</td>
</tr>
<tr>
<td></td>
<td>DS qualified based(Template)</td>
</tr>
<tr>
<td>xsd:int</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_int...</td>
</tr>
<tr>
<td></td>
<td>D isNil</td>
</tr>
<tr>
<td></td>
<td>D value</td>
</tr>
<tr>
<td></td>
<td>DS qualified based(Template)</td>
</tr>
</tbody>
</table>

11. In this chapter, anything we mention about the `wsdl2ws.sh` tool also holds true for the `wsdl2rpg.sh` tool.

© Copyright IBM Corp. 2011, 2012
<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Actual RPG type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:integer</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_integer...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value 20i 0</td>
</tr>
<tr>
<td>xsd:long</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_long...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value 20i 0</td>
</tr>
<tr>
<td>xsd:negativeInteger</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_negativeInteger...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value 20i 0</td>
</tr>
<tr>
<td>xsd:nonPositiveInteger</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_nonPositiveInteger...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value 20i 0</td>
</tr>
<tr>
<td>xsd:nonNegativeInteger</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_nonNegativeInteger...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value 20u 0</td>
</tr>
<tr>
<td>xsd:positiveInteger</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_positiveInteger...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value 20u 0</td>
</tr>
<tr>
<td>xsd:unsignedByte</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_unsignedByte...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value 1a</td>
</tr>
<tr>
<td>xsd:unsignedInt</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_unsignedInt...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value 10u 0</td>
</tr>
<tr>
<td>xsd:unsignedLong</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_unsignedLong...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value 20u 0</td>
</tr>
<tr>
<td>xsd:unsignedShort</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_unsignedShort...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value 5u 0</td>
</tr>
</tbody>
</table>
Table 26. XML to RPG type mapping (continued)

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Actual RPG type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:short</td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_short...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value 5i 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date/Time/Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:date</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

where xsd_time is a data structure defined as:  
   D xsd_tm DS align qualified based(Template)  
   D sec 10i 0  
   D* seconds after the minute (0-61)  
   D min 10i 0  
   D* minutes after the hour (0-59)  
   D hour 10i 0  
   D* hours since midnight (0-23)  
   D mday 10i 0  
   D* day of the month (1-31)  
   D mon 10i 0  
   D* months since January (0-11)  
   D year 10i 0  
   D* years since 1900  
   D wday 10i 0  
   D* days since Sunday (0-6)  
   D yday 10i 0  
   D* days since January 1 (0-365)  
   D isdst 10i 0  
   D* Daylight Saving Time flag  

| xsd:dateTime | Implemented as RPG data structure named xsd_dateTime. The data structure is defined in the same way as xsd:date. |
| xsd:duration | Implemented as RPG data structure: |
|               |   D xsd_duration... |
|               |   D DS qualified based(Template) |
|               |   D isNil 1n |
|               |   D value 10i 0 |

| xsd:gDay      | Implemented as RPG data structure named xsd_gDay. The data structure is defined in the same way as xsd:date. |
| xsd:gMonth    | Implemented as RPG data structure named xsd_gMonth. The data structure is defined in the same way as xsd:date. |
| xsd:gMonthDay | Implemented as RPG data structure named xsd_gMonthDay. The data structure is defined in the same way as xsd:date. |
| xsd:gYear     | Implemented as RPG data structure named xsd_gYear. The data structure is defined in the same way as xsd:date. |
| xsd:gYearMonth| Implemented as RPG data structure named xsd_gYearMonth. The data structure is defined in the same way as xsd:date. |
| xsd:time      | Implemented as RPG data structure named xsd_time. The data structure is defined in the same way as xsd:date. |

String
<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Actual RPG type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:anyURI</td>
<td>Implemented as RPG data structure named xsd_anyURI. For IBM i 6.1 and later</td>
</tr>
<tr>
<td></td>
<td>releases the structure is defined as follows:</td>
</tr>
<tr>
<td></td>
<td>D xsd:anyURI...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value a varying(4) len(nnnnn)</td>
</tr>
<tr>
<td></td>
<td>D reserved 1a</td>
</tr>
<tr>
<td></td>
<td>where nnnnn is the size of the character field.</td>
</tr>
<tr>
<td></td>
<td>For i 5.4, the RPG data structure would be defined as follows:</td>
</tr>
<tr>
<td></td>
<td>D xsd:anyURI...</td>
</tr>
<tr>
<td></td>
<td>D DS qualified based(Template)</td>
</tr>
<tr>
<td></td>
<td>D isNil 1n</td>
</tr>
<tr>
<td></td>
<td>D value nnnn varying</td>
</tr>
<tr>
<td></td>
<td>D reserved 1a</td>
</tr>
<tr>
<td></td>
<td>Note: The length that is used when defining the character field is directly</td>
</tr>
<tr>
<td></td>
<td>related to the -ms argument on the wsdl2ws.sh tool. See “The wsdl2ws.sh command</td>
</tr>
<tr>
<td></td>
<td>line tool” on page 53 for further details.</td>
</tr>
<tr>
<td>xsd:anyType</td>
<td>Implemented as RPG data structure named xsd_anyType. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:ENTITY</td>
<td>Implemented as RPG data structure named xsd_ENTITY. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:ENTITIES</td>
<td>Implemented as RPG data structure named xsd_ENTITIES. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:ID</td>
<td>Implemented as RPG data structure named xsd_ID. The data structure is defined</td>
</tr>
<tr>
<td></td>
<td>in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:IDREFS</td>
<td>Implemented as RPG data structure named xsd_IDREFS. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:language</td>
<td>Implemented as RPG data structure named xsd_language. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:Name</td>
<td>Implemented as RPG data structure named xsd_Name. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:NCName</td>
<td>Implemented as RPG data structure named xsd_NCName. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:NMTOKEN</td>
<td>Implemented as RPG data structure named xsd_NMTOKEN. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:NMTOKENS</td>
<td>Implemented as RPG data structure named xsd_NMTOKENS. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:normalizedString</td>
<td>Implemented as RPG data structure named xsd_normalizedString. The data structure</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:notation</td>
<td>Implemented as RPG data structure named xsd_notation. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:QName</td>
<td>Implemented as RPG data structure named xsd_QName. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:string</td>
<td>Implemented as RPG data structure named xsd_string. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
<tr>
<td>xsd:token</td>
<td>Implemented as RPG data structure named xsd_token. The data structure is</td>
</tr>
<tr>
<td></td>
<td>defined in the same way as xsd:anyURI.</td>
</tr>
</tbody>
</table>
Table 26. XML to RPG type mapping (continued)

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Actual RPG type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>xsd:base64Binary</strong></td>
<td>Implemented as RPG data structure named xsd_base64Binary. For i 6.1 and later releases the structure is defined as follows:</td>
</tr>
<tr>
<td></td>
<td>D xsd_base64Binary... D qualified based(Template) D isNil 1n D value a varying(4) len(nnnnn)</td>
</tr>
<tr>
<td></td>
<td>where nnnnn is the size of the character field.</td>
</tr>
<tr>
<td></td>
<td>For i 5.4, the RPG data structure would be defined as follows:</td>
</tr>
<tr>
<td></td>
<td>D xsd_base64Binary... D qualified based(Template) D isNil 1n D value nnnnn varying</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> The length that is used when defining the character field is directly related to the -mb argument on the wsd12ws.sh tool. See “The wsd12ws.sh command line tool” on page 53 for further details.</td>
</tr>
<tr>
<td><strong>xsd:boolean</strong></td>
<td>Implemented as RPG data structure:</td>
</tr>
<tr>
<td></td>
<td>D xsd_boolean... D qualified based(Template) D isNil 1n D value 10i 0</td>
</tr>
<tr>
<td><strong>xsd:hexBinary</strong></td>
<td>Implemented as RPG data structure named xsd_hexBinary. For i 6.1 and later releases the structure is defined as follows:</td>
</tr>
<tr>
<td></td>
<td>D xsd_hexBinary... D qualified based(Template) D isNil 1n D value a varying(4) len(1024)</td>
</tr>
<tr>
<td></td>
<td>where nnnnn is the size of the character field.</td>
</tr>
<tr>
<td></td>
<td>For i 5.4, the RPG data structure would be defined as follows:</td>
</tr>
<tr>
<td></td>
<td>D xsd_hexBinary... D qualified based(Template) D isNil 1n D value nnnnn varying</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> The length that is used when defining the character field is directly related to the -mb argument on the wsd12ws.sh tool. See “The wsd12ws.sh command line tool” on page 53 for further details.</td>
</tr>
</tbody>
</table>

The Axis RPG types listed in the table above are defined dynamically as part of the RPG stub code generation and included in the file <portType>_xsdtypes.rpleinc, where <portType> is the wsd1:portType defined in the WSDL and discussed in “WSDL to RPG mapping” on page 168. The reason that they are defined dynamically is because the string and array length values are obtained from what you specify on the wsd12ws.sh command (if you do not specify values default values will be used).

In general, all RPG simple types contain two fields:

- An isNil field, which is defined as an indicator to indicate whether the variable of this type is nil or not, and
- A value field to store data of the defined type.
Simple types

Most of the simple XML data types defined by XML Schema and SOAP 1.1 encoding are mapped to the RPG types discussed in the previous section.

The element declaration with a nillable attribute set to true for a built-in simple XML data type is handled by the isNil field that is contained in all RPG simple types. The isNil field is also used for simple types that are optional (minoccurs attribute set to 0).

Note: If a WSDL element does not have the nillable attribute set to true or is not optional, the isNil field is ignored for non-string types.

Complex types

XML Schema complex types are mapped to RPG data structures.

Let us look at the mapping that occurs for the following schema fragment:

```xml
<xsd:complexType name="Book">
    <sequence>
        <element name="author" type="xsd:string"/>
        <element name="price" type="xsd:float"/>
    </sequence>
    <xsd:attribute name="reviewer" type="xsd:string"/>
</xsd:complexType>
```

The above example is an example of a complex type that is named Book, and contains two elements, author and price, in addition to an attribute, reviewer. The complex type will get mapped to the following RPG structure:

```rpg
D Book_t...
D isNil_Book_t 1n
D reviewer likes(xsd_string)
D author likes(xsd_string)
D price likes(xsd_float)
```

In the example above you see one additional field that is generated, isNil_Book_t. This indicator is a way to indicate to the Axis engine whether the variable of this type should be nil or not. When used as an input parameter, setting this field to *ON will result in the nillable attribute set to true for the element when the request is sent to the Web service. When the data structure is received as a response to a Web service request, the field should be checked to determine if the element was nil or omitted from the Web service response. If the field is set to *OFF, then you can be assured that element was returned in the Web service response.

Arrays

Arrays for the RPG language are defined as a data structure as follows:

```rpg
D <array_name>...
D D isNil_<array_name>...
D array likes(<data-structure-name>)
D dim(<nnnn>)
D size 10i 0
D type 10i 0
```

The fields in the structure include:

- isNil_<array_name> field, which is defined as an indicator to indicate to the Axis engine whether the variable of this type should be nil or not. When used as an input parameter, setting this field to *ON
will result in the nillable attribute set to true for the element when the request is sent to the Web service. When the data structure is received as a response to a Web service request, the field should be checked to determine if the element was nil or omitted from the Web service response. If the field is set to *OFF, then you can be assured that element was returned in the Web service response.

* array field, which contains the elements of the array of type <data-structure-name>. The length that is used when defining the dimension of the array is directly related to the -ma argument on the wsd12ws.sh tool. See "The wsd12ws.sh command line tool" on page 53 for further details.

* size field, which contains the number of valid elements in the array. For example, if an array with a dimension of 20 has 2 valid elements that should be sent in a Web service request, the size field should be set to 2.

* type field, which is an indication of the type of element (for example, array of integers, or an array of user-defined complex structures). Constants for the possible types are defined in the generated <portType>_xsdtypes.rpgleinc file. There are constants for all the simple types. For example, XSDC_STRING and XSDC_INT. For complex types, the field should be set to XSDC_USER_TYPE.

Web services client for ILE includes provides array data structures for each of the defined simple types, defined in the generated <portType>_xsdtypes.rpgleinc file. An example of a simple array type is xsd_int_Array shown below.

```
D xsd_int_array...
D   DS qualified based(Template)
D   isNil In
D   array likeds(xsd_int)
D   dim(MAX_ARRAY_LEN)
D   size 10i 0
D   type 10i 0
```

**Notes:**
1. The name isNil does not include the type in the name. This is true for all simple type arrays.
2. The MAX_ARRAY_LEN is a constant that is set to a value that is directly related to the -ma argument on the wsd12ws.sh tool. See "The wsd12ws.sh command line tool" on page 53 for further details.

Below is the same schema fragment we have used previously, but we have also increased the number of authors a book can have to 10 by adding maxOccurs="10" to the author element:

```
<xsd:complexType name="Book">
   <sequence>
      <element name="author" type="xsd:string" maxOccurs="10"/>
      <element name="price" type="xsd:float"/>
   </sequence>
   <xsd:attribute name="reviewer" type="xsd:string"/>
</xsd:complexType>
```

For the above XML Schema, the following data structure is generated:

```
D Book_t DS qualified based(Template)
D isNil_Book_t In
D reviewer likeds(xsd_string)
D author likeds(xsd_string_array)
D price likeds(xsd_float)
```

As you can see, the string array data structure is now being used to store the values for the author element.
**WSDL to RPG mapping**

Now that we understand how the XML Schema types are mapped to Axis-defined language types, we can now review how a service described in a WSDL document gets mapped to the corresponding C representation. The following sections will refer to the GetQuote.wsdl WSDL document that is shipped as part of the product in directory `<install_dir>/samples/getQuote` and is listed in "The GetQuote.wsdl File" on page 187 to illustrate how various WSDL definitions get mapped to RPG. You should note the following:

- GetQuote.wsdl has only one service called GetQuoteService.
- The service only has one port type called StockQuote.
- The StockQuote port type has only one operation called getQuote. The input to the getQuote operation is a string (the stock identifier) and the output from the operation is a float (the stock's price).

If you want to fully understand the WSDL document structure, see "WSDL document structure" on page 32. Now let us see how various WSDL definitions are mapped.

This section describes the mapping of a service described in a WSDL document to the corresponding RPG representation. The table below summarizes the WSDL and XML to RPG mappings:

<table>
<thead>
<tr>
<th>WSDL and XML</th>
<th>RPG</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:complexType (structure)</td>
<td>RPG DS structure.</td>
</tr>
<tr>
<td>Nested xsd:element or xs:attribute</td>
<td>RPG DS structure field.</td>
</tr>
<tr>
<td>xsd:complexType (array)</td>
<td>RPG DS Axis array structure.</td>
</tr>
<tr>
<td>wsdl:message determines the method parameter signature.</td>
<td>Service interface function signature.</td>
</tr>
<tr>
<td>wsdl:portType</td>
<td>Service interface function.</td>
</tr>
<tr>
<td>wsdl:operation</td>
<td>Service interface function.</td>
</tr>
<tr>
<td>wsdl:binding</td>
<td>No direct mapping, affects SOAP communications style and transport.</td>
</tr>
<tr>
<td>wsdl:service</td>
<td>No direct mapping.</td>
</tr>
<tr>
<td>wsdl:port</td>
<td>Used as default Web service location.</td>
</tr>
</tbody>
</table>

**Mapping XML defined in wsdl:types**

The `wsdl2ws.sh` command will either use an existing RPG simple type or generate a new RPG type (a DS structure) for the XML schema constructs defined in the `wsdl:types` section. The mappings that the `wsdl2ws.sh` command supports is discussed in "XML schema to RPG type mapping" on page 161. As previously stated, the `wsdl2ws.sh` command either will ignore constructs that it does not support or issue an error message.

If we look at the `wsdl:types` part of the WSDL document we see that two elements are defined: `getQuote`, defined as a complex type with one element of type `xsd:string`; and `getQuoteResponse`, also defined as a complex type with one element of type `xsd:float`.

```xml
<wxsd:types>
  <ati:schema elementFormDefault="qualified"
              targetNamespace="http://stock.ibm.com"
              xmlns="http://www.w3.org/2001/XMLSchema"
              xmlns:ati="http://www.w3.org/2001/XMLSchema">
    <ati:element name="getQuote">
      <ati:complexType>
```
For the WSDL document fragment above, the \texttt{wsdl2ws.sh} command does not generate any new structures since both elements are defined to be built-in simple types. The \texttt{xsd:string} type is mapped to \texttt{xsd_string} and the \texttt{xsd:float} type is mapped to \texttt{xsd_float}.

\textbf{Mapping of \texttt{wsdl:portType}}

A port type is a named set of abstract operations and the abstract messages involved. The name of the \texttt{wsdl:portType} will be used in the names of the Web service proxy (termed service interface) functions. A port type is mapped to 2 functions:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Function name} & \textbf{Description} \\
\hline
\texttt{stub_create_<portType-name>} & Function that is used to get an object (more on this later) representing the service interface (i.e. the Web service proxy stub). \\
\texttt{stub_destroy_<portType-name>} & Function used to destroy service interface objects that are obtained by invoking \texttt{stub_create_<portType-name>}. \\
\hline
\end{tabular}
\end{table}

Now let us look at a concrete example of how the \texttt{wsdl:portType} below gets mapped.

\texttt{...<wsdl:portType name="StockQuote">}
\texttt{<wsdl:operation name="getQuote">}
\texttt{<wsdl:input message="impl:getQuoteRequest" name="getQuoteRequest"/>}
\texttt{<wsdl:output message="impl:getQuoteResponse" name="getQuoteResponse"/>}
\texttt{</wsdl:operation>}
\texttt{</wsdl:portType>...}

The \texttt{wsdl2ws.sh} command will generate the following RPG functions:

\begin{verbatim}
D stub_create_StockQuote...
D this         PR  IN  extproc('stub_create_StockQuote@')
D extra
D Value0 likeds(xsd_string)
D out likeds(xsd_float)
\end{verbatim}
The `stub_create_StockQuote()` and `stub_destroy_StockQuote()` functions shown above are the functions that are generated in support of the service interface. You see how the `wsdl:portType` name `StockQuote` is used in the naming of the functions. To use these functions you need to pass in a variable that is a data structure of type `This_t`. This type is defined in the generated `<portType>_xsdtypes.rpgleinc` file. In this example, the file would be named `StockQuote_xsdtypes.rpgleinc`. This variable represents a stub instance once the `stub_create_StockQuote()` is called and the function completes successfully. The data structure `This_t` is defined as follows:

```
D This_t DS qualified based(Template)
D endpoint 2048a
D handle *
D excOccurred 1n
D excCode 10i 0
D excString 2048a
D reserved 1024a
```

where:

- the `endpoint` field represents the URL of the Web service. The endpoint must be set to a value or blanks before calling the function to get an instance of the RPG stub. If the endpoint is set to blanks, then the default URL to the Web service will be used. The default URL is whatever is specified in the `wsdl:port` WSDL definition.
- the `handle` field represents the C stub instance (remember that RPG stub code is built on top of the C stub code) and is the interface to the Axis engine. This handle would be the variable you pass in all Axis C APIs that begin “axiscStub”. For example, to set an HTTP header “MYHEADER” you would create an RPG stub instance and invoke the AXIS API `axiscStubSetTransportProperty()` as follows:

```
stub_create_StockQuote(WsStub);
axiscStubSetTransportProperty(WsStub.handle:'MYHEADER':'SOMEVALUE');
```

- the `excOccurred` field indicates whether a service interface function call was successful or not. The field will be set to *ON if the function call was not successful, and *OFF if the function call was successful. For more information on exception handling, see [“RPG exception handling” on page 177](#).
- the `excCode` field will contain the exception code if `excOccurred` is set to *ON. Exception codes are defined `<install_dir>/include/Axis.rpgleinc`.
- the `excString` field will contain the exception error message if `excOccurred` is set to *ON.

The last RPG function shown, `stub_op_getQuote()`, is mapped from the `wsdl:operation` element defined in the `wsdl:portType` (refer to [“Mapping of `wsdl:operation`” on page 169](#) for further explanation of the mapping of `wsdl:operation`).

### Mapping of `wsdl:operation`

A `wsdl:operation` within a `wsdl:portType` is mapped to an RPG function. The name of the `wsdl:operation` is used in the generation of the Web service operation function. All Web service operation functions will start with “stub_op_” followed by the name of the `wsdl:operation`. The first parameter is of type `This_t` that represents the service interface stub object and is discussed in [“Mapping of `wsdl:portType`” on page 169](#).

The `wsdl:operation` contains `wsdl:input` and `wsdl:output` elements that reference the request and response `wsdl:message` constructs using the message attribute. Each function parameter (except the first) is defined by a `wsdl:message` part referenced from the input and output elements:

- A `wsdl:part` in the request `wsdl:message` is mapped to an input parameter.
- A `wsdl:part` in the response `wsdl:message` is mapped to the return value.
The **wsdl:operation** can contain **wsdl:fault** elements that reference **wsdl:message** elements describing the fault (refer to “Mapping of **wsdl:fault**” on page 172 for more details on **wsdl:fault** mapping).

The Web Services Client for ILE supports the mapping of operations that use either a request/response or one-way (where **wsdl:output** is not specified in the **wsdl:operation** element) message exchange pattern. For the one-way message exchange pattern, the Axis engine expects an HTTP response to be returned from the Web service. Under normal conditions, the HTTP response would contain no SOAP data. However, if a SOAP fault is returned by the Web service, the Axis engine will process the fault.

Below are the **wsdl:message** and **wsdl:portType** WSDL definitions in the GetQuote.wsdl document:

```xml
...<wsdl:message name="getQuoteRequest">
    <wsdl:part element="impl:getQuote" name="parameters"/>
</wsdl:message>
<wsdl:message name="getQuoteResponse">
    <wsdl:part element="impl:getQuoteResponse" name="parameters"/>
</wsdl:message>
...
<wsdl:portType name="StockQuote">
    <wsdl:operation name="getQuote">
        <wsdl:input message="impl:getQuoteRequest" name="getQuoteRequest"/>
        <wsdl:output message="impl:getQuoteResponse" name="getQuoteResponse"/>
    </wsdl:operation>
</wsdl:portType>
...
```

The above **wsdl:operation** definition gets mapped to the following service interface function:

```rpg
D stub_op_getQuote...
D PR 1N extproc('getQuote@')
D this likeds(This_t)
D Value0 likeds(xsd_string)
D out likeds(xsd_float)
```

**Mapping of **wsdl:binding**

The **wsdl:binding** information is used to generate an implementation specific client side stubs. What code is generated is dependent on protocol-specific general binding data, such as the underlying transport protocol and the communication style of SOAP.

There is no RPG representation of the **wsdl:binding** element.

**Mapping of **wsdl:port**

A **wsdl:port** definition describes an individual endpoint by specifying a single address for a binding.

The specified endpoint will be used in as the default location of the Web service. So in the case of our example, the URL specified in **wsdl:port** definition below will be the URL that is used if **stub_create_StockQuote()** is invoked with the endpoint field in the **This_t** data structure is set to blanks.

```xml
...<wsdl:service name="GetQuoteService">
    <wsdl:port name="StockQuote" binding="impl:StockQuoteSoapBinding">
        <wsdlsoap:address
            location="http://localhost:9080/StockQuote/services/GetQuoteService"/>
    </wsdl:port>
</wsdl:service>
...
Mapping of wsdl:fault

Within the wsdl:operation definition you can optionally specify the wsdl:fault element, which specifies the abstract message format for any error messages that may be returned as a result of invoking a Web service operation.

The wsdl:fault element must reference a wsdl:message that contains a single message part. As of this writing, Axis only supports message parts that are xsd:complexType types. The mapping that occurs is similar to the mapping that occurs when generating code for complex types.

So what happens when a SOAP fault is received? When you call a service interface function and a SOAP fault is encountered (or a non-Fault exception for that matter), the Axis engine will throw an exception, and the C interfaces to the Axis engine catch the exception and invokes the RPG stub service interface exception handler, passing it the generic exception code and exception string associated with the exception. The RPG stub service interface exception handler stores the exception data in global fields. The service interface function then regains control and checks the fields to see if an exception had occurred, and if so, copies the exception data to the RPG stub instance that is represented by the This_t data structure. For more information on the This_t structure, see "Mapping of wsdl:portType" on page 169.

More information on exception handling in RPG can be found in "RPG exception handling" on page 177.
Chapter 21. Developing a Web services client application using RPG stubs

This chapter will describe the steps one must go through to develop a Web service client application using a RPG stub code.

To develop a Web services client application, the following steps should be followed:
1. Generate the client Web service stubs using the wsd12ws.sh command.
2. Complete the client implementation.
3. (Optional) Create client-side handler.
4. Deploy the application.

The following sections will discuss each of these steps. For illustrative purposes we will be using the sample code that is shipped as part of the product in directories <install_dir>/samples/getQuote. We will be using the following files:

Table 29. Files in the samples directory

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetQuote.wsdl</td>
<td>GetQuote WSDL file.</td>
</tr>
<tr>
<td>myGetQuote.rpgle</td>
<td>Client implementation code written in RPG.</td>
</tr>
</tbody>
</table>

Source listings for the client application code can be found at Appendix A, “Code Listings for myGetQuote Client Application,” on page 187.

Generating the RPG stub code

Before you can create a web service client application, you must first generate the RPG client stub using the wsd12ws.sh tool. The wsd12ws.sh tool uses the WSDL file that is passed to it, and any associated XSD files referenced in the WSDL file, to create the client stub code.

We will be using the GetQuote.wsdl file located in directory <install_dir>/samples/getQuote. The WSDL file comes from the installation Web Services Samples provided with WebSphere Application Server (Version 5.0 or later). This very simple sample provides a good introduction to using wsd12ws.sh.

To generate the client stub from the WSDL source file, complete the following steps.
1. Create a library called MYGETQUOTE in which the program objects will be stored by issuing the CL command CRTLIB from the CL command line as follows:
   CRTLIB MYGETQUOTE
2. Start a Qshell session by issuing the QSH CL command from the CL command line.
3. Run the wsd12ws.sh tool to generate the client RPG stub as shown in following example:
   `<install_dir>/bin/wsd12ws.sh -o/myGetQuote/RPG
   -lrpg -ms256 -ma5
   -s/qsys.lib/mygetquote.lib/wsrpg.srvpgm
   `<install_dir>/samples/getQuote/GetQuote.wsdl`

If you examine the command, you see that we are indicating to the wsd12ws.sh tool that RPG stub code should be generated and stored in directory /myGetQuote/RPG, and that a service program, /qsys.lib/mygetquote.lib/wsrpg.srvpgm, should be created using the generated stub code. In addition, we indicate that the maximum string size is 256 bytes and that the maximum array size should be 5.
The files generated by the \texttt{wsdl2ws.sh} tool is shown below:

\begin{verbatim}
 StockQuote_util.rpgle  StockQuote.cl
 StockQuote_util.rpgleinc StockQuote.h
 StockQuote-xsdtypes.rpgleinc StockQuote.rpgle
 StockQuote.c  StockQuote.rpgleinc
\end{verbatim}

Note that in addition to the RPG stub code being generated, C stub code is also generated since the RPG stub code is built on top of the C stub code.

Here is a description of each RPG file that is generated:

- \texttt{StockQuote_util.rpgle} – RPG utility routines.
- \texttt{StockQuote_util.rpgleinc} – RPG utility routines include.
- \texttt{StockQuote-xsdtypes.rpgleinc} – standard data types include.
- \texttt{StockQuote.rpgle} – RPG Web service implementation code.
- \texttt{StockQuote.rpgleinc} – RPG Web service include.

From an RPG programmer perspective, the only files you would need to look at are the \texttt{StockQuote.rpgleinc} and \texttt{StockQuote-xsdtypes.rpgleinc} files.

Finally, there is also the file \texttt{StockQuote.cl} that is also generated. This file is a CL source file that has the CL commands needed to recreate the service program. You can copy this source file to a source physical file and create a CL program. Here is the contents of the file:

\begin{verbatim}
PGM
 DCL VAR(&LIB) TYPE(*CHAR) LEN(10) VALUE(MYGETQUOTE)
 DCL VAR(&SRVPGM) TYPE(*CHAR) LEN(10) VALUE(WSRPG)

 QSYS/CRTCMOD MODULE(&LIB/wsc0) +
   OPTIMIZE(40) DBGVIEW(*NONE) +
   SRCSTMF('/myGetQuote/RPG/StockQuote.c') +
   INCDIR('/QIBM/PRODDATA/OS/WEBSERVICES/V1/CLIENT/INCLUDE') +
   REPLACE(*YES) ENUM(*INT) +
   TEXT('StockQuote.c')

 QSYS/CRTRPGMOD MODULE(&LIB/wsr1) +
   SRCSTMF('/myGetQuote/RPG/StockQuote.rpgle') +
   OPTIMIZE(*FULL) DBGVIEW(*NONE) +
   REPLACE(*YES) +
   TEXT('StockQuote.rpgle')

 QSYS/CRTRPGMOD MODULE(&LIB/wsr2) +
   SRCSTMF('/myGetQuote/RPG/StockQuote_util.rpgle') +
   OPTIMIZE(*FULL) DBGVIEW(*NONE) +
   REPLACE(*YES) +
   TEXT('StockQuote_util.rpgle')

 QSYS/CRTSRVPGM SRVPGM(&LIB/&SRVPGM) +
   MODULE( +
     &LIB/wsr1 +
     &LIB/wsr2 +
     &LIB/wsc0 +
   ) +
   EXPORT(*ALL) ACTGRP(*CALLER) +
   BNDSRVPGM(QSYSDIR/QAXIS10CC) +
   TEXT('StockQuote Web service')

 ENDPGM
\end{verbatim}
Now that the RPG and C stub code has been created and a service program containing the RPG and C stub code is created, you can go on to the next step, “Completing RPG client implementation.”

### Completing RPG client implementation

After the client stubs have been generated, the stubs can be used to create a Web service client application.

We will illustrate what you need to do to create RPG applications using the example of the RPG stub code generated from GetQuote.wsdl by the wsd12ws.sh tool as described in “Generating the RPG stub code” on page 173. However, before we continue, you should note the following points:

- GetQuote.wsdl has only one service called getQuoteService.
- The service only has one port type called StockQuote.
- The StockQuote port type has only one operation called getQuote. The corresponding RPG stub operation (defined in the generated StockQuote.rpgleinc include file) is stub_op_getQuote().
- The Web service is called StockQuote. So to get an instance of the Web service you would call the stub_create_StockQuote() function. The handle that is returned by the function should then be used when calling the Web service operation. To destroy the Web service instance, you would call the stub_destroy_StockQuote() function. (Both these functions are defined in the generated StockQuote.rpgleinc include file.)

To build the myGetQuote client application, complete the following steps.

1. Change the current working directory to the location of the RPG stub code. Issue the following command from the CL command line:
   ```
   cd '/myGetQuote/RPG'
   ```
2. Copy the sample RPG code the uses the generated stub code from the product samples directory to the current working directory by issuing the following command from the CL command line:
   ```
   COPY OBJ('<install_dir>/samples/getQuote/myGetQuote.rpgle') TODIR('/myGetQuote/RPG')
   ```
3. Change the ServerName and PortNumber in the file copied in the previous step to match your server. If WebSphere Application Server is on your own machine and the default values have been used, ServerName is localhost and PortNumber is 9080.
4. Build the client application by using the following commands from the CL command line:
   ```
   CRTRPGMOD MODULE(MYGETQUOTE/mygetquote)
   SRCSTMF('/myGetQuote/RPG/myGetQuote.rpgle')
   CRTPGM PGM(MYGETQUOTE/MYGETQUOTE)
   MODULE(MYGETQUOTE/MYGETQUOTE)
   BNDSRVPGM(QSYSDIR/QAXIS10CC MYGETQUOTE/WSRPG)
   ```

When you have finished coding and building your web service client application, you are ready to deploy and test the application as described in “Deploying the client application.”

**Note:** If you want to use one or more handlers with your application, see Chapter 11, “Creating client-side handlers,” on page 79.

### Deploying the client application

When you have finished coding and building your web service client application, you are ready to deploy and test the application.

12. If you have not read Chapter 20, “WSDL and XML to RPG mappings,” on page 161 then it would be a good time to do so prior to reading this section.
In our example, we have not modified the Axis configuration file axisconf.conf. However, if we had modified it (e.g. we were using client-side handlers), we would need to ensure that the AXISCPP_DEPLOY environment variable points to the directory containing the /etc directory (the axisconf.conf file would be located in the /etc directory), as described in "The axisconf.conf file" on page 57.

The steps below use the example myGetQuote client application, and assume that a GetQuote service is running. (This service is with the samples supplied with WebSphere Application Server Version 5.0.2 or later). If you do not have the appropriate service, you must create the service code from the WSDL in the samples directory.

Once you have confirmed the above prerequisites, run and test the client application by completing the following steps.
1. Run the myGetQuote application.
2. Check that the myGetQuote application has returned the price of IBM shares in dollars.

The example screen shot below shows the myGetQuote application run from the command line in which client-side handlers are not being used.

```
> call MYGETQUOTE/MYGETQUOTE
DSPLY The stock quote for IBM is $94.33
```
Chapter 22. RPG stub programming considerations

This chapter covers programming considerations when you begin writing your applications to take advantage of Web services client for ILE RPG stub code.

RPG exception handling

Web Services Client for ILE uses exceptions to report back any errors that have occurred during the transmission of a SOAP message. This includes errors that are detected by the Axis engine or SOAP faults that are returned by the Web service.

When using the RPG-stub interfaces, errors that occur are reported to the client application in two ways:

1. A return indicator on the stub function interfaces. If the return indicator is *ON, then the function call completed successfully. If the return indicator is *OFF, then the function call did not complete successfully.

2. By interrogating the stub instance handle that is of type This_t. The This_t data structure is as follows:

   ```
   D This_t DS qualified based(Template)
   D endpoint 2048a
   D handle *
   D excOccurred 1n
   D excCode 10i 0
   D excString 2048a
   D reserved 1024a
   ```

   After performing the stub interface function call, you can check the excOccurred field to determine if an exception occurred. If excOccurred is *ON, then an exception has occurred, and the exception details can be obtained in the exception code (excCode) and exception string (excString) fields.

The following shows how a client application may process exceptions.

```rpg
if (stub_create_StockQuote(WsStub) = *ON);
  // Invoke the StockQuote Web service operation.
  stub_op_getQuote(WsStub:Input:Result);
  if (WsStub.excOccurred = *OFF);
    OutputText = 'The stock quote for ' + Input.value + ' is ' + %CHAR(Result.value);
  else;
    OutputText = WsStub.excString;
  endif;
  // Display results.
  dsply OutputText;
  // Destroy Web service stubs.
  stub_destroy_StockQuote(WsStub);
endif;
```
RPG memory management

The WSDL specification provides a framework for how information is to be represented and conveyed from place to place. Web services client for ILE maps this framework to program-language specific data object, such as structures. The RPG stub code does not expose any dynamic storage via pointers, so there is no memory to manage when coding to the RPG stub code interfaces. The only thing one needs to ensure is that the instance of the RPG stub that is created is eventually destroyed by calling the destroy function for the RPG stub.

Securing web service communications in RPG stub code

This section explains how to use Secure Sockets Layer (SSL) to set up security when using RPG stub code.

You can secure your HTTP messages by using SSL, which encrypts the request and response messages before they are transmitted over the wire.

Note: Handlers are not affected by SSL as they receive the message either before encryption or after decryption.

Any web service that uses SSL adds the suffix ‘s’ for secure to the http name in the URL. For example: http://some.url.com becomes https://some.url.com

A secure endpoint URL is an endpoint beginning with ‘https’. To allow secure endpoint URLs to be used, you must pass security information to the RPG stub. You can do this either by adding the required information to the [The axiscpp.conf file on page 57] configuration file, or by configuring the settings for secure service using the [axiscStubSetSecure() on page 153] Axis C API.

The following RPG example shows a sample client application that configures security information before calling a web service. To configure the secure setting within your own application, add the code shown in bold in this example.

```rpg
h DFTNAME(MYGETQUOTE)
/copy StockQuote.rpgleinc
/copy /QIBM/ProdData/OS/WebServices/V1/client/include/Axis.rpgleinc

/outputText s 50
/d WsStub ds likeds(This_t)
/d Input ds likeds(xsd_string)
/d Result ds likeds(xsd_float)

/free
   clear WsStub;

   // URL for secure communication.
   WsStub.endpoint = 'https://localhost/Test/services/TestPort';

   // Get a Web service stub.
   if (stub_create_StockQuote(WsStub) = *ON);
   // Initialise the secure settings for the secure service.
   axiscStubSetSecure(WsStub.handle;
     '<path to KeyRing.kbd>':
     '<password to stash or NULL string>':
     '<label>': 'NONE'; '05': 'NONE');

   // Remainder of application
   .
   .
   .

   // Destroy Web service stub.
   stub_destroy_StockQuote(WsStub);
```

178 Web Services Client for ILE: Programming Guide
Note: The <password to stash or NULL string> parameter can be replaced with a NULL string, which indicates to SSL support that the internal stash file is to be used.

For further information on the SSL parameters, see the "axiscStubSetSecure()" on page 153 Axis C API.

### Setting SOAP headers

This section explains how to set SOAP headers when using RPG stub code.

You can set SOAP headers in the RPG stub by using various Axis C APIs.

Say we want to send the following SOAP header in the Web service request:

```xml
<wsse:Security
    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse-secext-1.0.xsd"
    SOAP-ENV:mustUnderstand="1"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse-utility-1.0.xsd">
    <wsse:UsernameToken wsu:Id="UsernameToken-12345678">
        <wsse:UserName>admin</wsse:UserName>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">
            admin
        </wsse:Password>
    </wsse:UsernameToken>
</wsse:Security>
```

The following RPG example uses various Axis C APIs to set SOAP header information.

```rpg
h DFTNAME(MYGETQUOTE)
/copy StockQuote.rpgleinc
/copy /QIBM/ProdData/OS/WebServices/V1/client/include/Axis.rpgleinc
```

```rpg
d OutputText s 50
   d WsStub ds likeds(This_t)
   d Input ds likeds(xsd_string)
   d Result ds likeds(xsd_float)
   D phb S *
   D BNode1 S *
   D BNode2 S *
   D BNode3 S *
   D uriWSSE C 'http://docs.oasis-open.org/wss-
     /2004/01/oasis-200401-wss-wsse-secext-1.0.xsd'
   D uriWSU C 'http://docs.oasis-open.org/wss-
     /2004/01/oasis-200401-wss-wsse-utility-1.0.xsd'
   D uriToken C 'http://docs.oasis-open.org/wss-
     /2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText'

   /free
   clear WsStub;
   // URL for secure communication.
   WsStub.endpoint = 'https://localhost/Test/services/TestPort';

   // Get a Web service stub.
   if (stub_create_StockQuote(WsStub) = *ON);
       // Set SOAP headers.
```

Chapter 22. RPG stub programming considerations  179
// generate node wsse:Security element, declaring namespaces for wsse and wsu
phb = axiscStubCreateSOAPHeaderBlock(
    WsStub.handle:'Security':uriWSSE:'wsse');
axiscHeaderBlockCreateNamespaceDeclINamespace(
    phb:'wsu':uriWSU);
axiscHeaderBlockCreateStdAttribute(phb:
    AXISC_ATTR_MUST_UNDERSTAND_TRUE:
    AXISC_SOAP_VER_1_1);

// Generate node wsse:UsernameToken as child node of wsse:Security
Bnode1=axiscHeaderBlockCreateChildBasicNode(phb:
    AXISC_ELEMENT_NODE:'UsernameToken':'wsse':*NULL:*NULL);
axiscBasicNodeCreateAttribute(Bnode1:
    'Id':'wsu':*NULL:'UsernameToken-12345678');
axiscHeaderBlockAddChild(phb:Bnode1);

// Generate node wsse:Username as child node of wsse:UsernameToken
// and the associated character node
Bnode2=axiscHeaderBlockCreateChildBasicNode(phb:
    AXISC_ELEMENT_NODE:'UserName':'wsse':*NULL:*NULL);
axiscBasicNodeAddChild(Bnode1:Bnode2);
Bnode3=axiscHeaderBlockCreateChildBasicNode(phb:
    AXISC_CHARACTER_NODE:*NULL:*NULL:*NULL:'admin');
axiscBasicNodeAddChild(Bnode2:Bnode3);

// Generate node wsse:Password as child node of wsse:UsernameToken
// and the associated character node
Bnode2=axiscHeaderBlockCreateChildBasicNode(phb:
    AXISC_ELEMENT_NODE:'Password':'wsse':*NULL:*NULL);
axiscBasicNodeCreateAttribute(Bnode2:
    'Type':*NULL:*NULL:uriToken);
axiscBasicNodeAddChild(Bnode1:Bnode2);
Bnode3=axiscHeaderBlockCreateChildBasicNode(phb:
    AXISC_CHARACTER_NODE:*NULL:*NULL:*NULL:'admin');
axiscBasicNodeAddChild(Bnode2:Bnode3);

// Remainder of application

// Destroy Web service stub.
stub_destroy_StockQuote(WsStub);
endif;

*INLR=*ON;
/end-free

For further information on the Axis C APIs used in this example, see Chapter 19, “Axis C core APIs,” on page 139.

Floating point numbers in RPG types

This section provides reference information about using floating point numbers with Web services client for ILE.

The XML specification refers to the IEEE specification for floating point numbers. The specification lists that float and double have the following precision:

- Float type numbers, 1 sign bit, 23 mantissa bits and 8 exponent bits.
- Double type numbers, 1 sign bit, 52 mantissa bits and 11 exponent bits.
For float, with a mantissa able to represent any number in the range $1 > x > 1/2^{23}$, this gives a minimum accuracy of 6 digits. Similarly, for double, with a mantissa able to represent any number in the range $1 > x > 1/2^{52}$, this gives a minimum accuracy of 10 digits.

When displaying floating point numbers, you must ensure that any potential inaccuracies due to rounding errors, and so on are not visible.
Chapter 23. Troubleshooting RPG client stubs

This chapter is intended to help you learn how to detect, debug, and resolve possible problems that you may encounter when generating or using RPG stub code.

RPG stub code generation problems

When you use the wsdl2ws.sh tool to generate RPG stub code, the tool will generate an exception for any error that is encountered. Typical errors include the inability for the tool to resolve to an XSD file used in the specified WSDL file or a syntactically incorrect WSDL file. You will need to correct the problem and try running the tool again.

RPG stub code compile problems

Recall that the RPG stub code is built on top of the C stub code. So you may get compile problems when compiling the C stub code or the RPG stub code.

If there is a compile problem in C stub code, the most likely cause of the problem is the use of an unsupported construct. The wsdl2ws.sh tool will not always generate an exception when used against a WSDL file that contains an unsupported WSDL construct. The problem may manifest itself when compiling the generated stub code. To see what is supported by the tool, see "Supported specifications and standards" on page 43.

If there is a compile problem in RPG stub code, the most likely cause is one of the following cases:

- The sizes of fields or data structures exceeding the language limits. For example, in IBM i 5.4 the size of a data structure cannot exceed 65535 bytes, while in i 6.1 the limit is 16773104 bytes. To resolve the problem, you need to experiment with the wsdl2ws.sh tool arguments that related to field and array sizes.
- The WSDL specifies two types that reference each other. Here is an example of the generated RPG code that will not compile:

  ```
  D Type1_t DS qualified based(Template)
  D isNil_Type1_t
  D att_kind likeds(xsd_string)
  D followings likeds(Type1_Array_t)
  D kind likeds(xsd_string)
  D index likeds(xsd_int)

  D Type1_Array_t DS qualified based(Template)
  D isNil_Type1_Array_t...
  D array likeds(Type1_t)
  D dim(20)
  D size 10i 0
  D type 10i 0
  ```

  The only way to resolve this kind of problem is by changing the WSDL file so that the cyclic reference is removed.

RPG stub code runtime problems

Invoking a Web service operation may result in the Web service returning a SOAP fault as a response. There can be many reasons for this, and the only sure way to determine where the problem lies is by examining the generated SOAP request and resulting response.

13. Any references to the wsdl2ws.sh is also applicable to the wsdl2rpg.sh tool.
The Web services client for ILE client engine has a tracing capability that traces the request and response messages. To enable tracing, the `axiscAxisStartTrace()` needs to be called. The following is an example of how tracing is enabled.

```
DFTNAME(MYGETQUOTE)
/copy StockQuote.rpgleinc
/copy /QIBM/ProdData/OS/WebServices/V1/client/include/Axis.rpgleinc

D OutputText s 50
D WsStub       ds likeds(This_t)
D Input        ds likeds(xsd_string)
D Result       ds likeds(xsd_float)

// Enable trace, specifying trace file
axiscAxisStartTrace('/tmp/axis.log':*NULL);

// Remainder of application
.
.
.
*INLR=*ON;
/end-free
```

To learn about the tracing support in Axis, see the “axiscAxisStartTrace()” on page 140 Axis C API.
Part 6. Appendixes
Appendix A. Code Listings for myGetQuote Client Application

The myGetQuote sample is a simple stock quote example that is referenced throughout the document. Table 30 shows a list of the files.

Table 30. Client files in the samples directory

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetQuote.wsdl</td>
<td>GetQuote WSDL file.</td>
</tr>
<tr>
<td>myGetQuote.cpp</td>
<td>Client implementation code written in CPP.</td>
</tr>
<tr>
<td>myGetQuote.c</td>
<td>Client implementation code written in C.</td>
</tr>
<tr>
<td>myGetQuote.rple</td>
<td>Client implementation code written in RPG.</td>
</tr>
</tbody>
</table>

All files can be found in `<install_dir>/samples/getQuote`.

The GetQuote.wsdl File

The following listing is for the GetQuote.wsdl WSDL document:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://stock.ibm.com"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:impl="http://stock.ibm.com"
 xmlns:wssdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<wsdl:types>
<ati:schema elementFormDefault="qualified"
 targetNamespace="http://stock.ibm.com"
 xmlns="http://www.w3.org/2001/XMLSchema"
 xmlns:ati="http://www.w3.org/2001/XMLSchema">
<ati:element name="getQuote">
  <ati:complexType>
    <ati:sequence>
      <ati:element name="arg_0_0" type="xsd:string"/>
    </ati:sequence>
  </ati:complexType>
</ati:element>

<ati:element name="getQuoteResponse">
  <ati:complexType>
    <ati:sequence>
      <ati:element name="getQuoteReturn" type="xsd:float"/>
    </ati:sequence>
  </ati:complexType>
</ati:element>
</ati:schema>
</wsdl:types>
<wsdl:message name="getQuoteRequest">
  <wsdl:part element="impl:getQuote" name="parameters"/>
</wsdl:message>
<wsdl:message name="getQuoteResponse">
  <wsdl:part element="impl:getQuoteResponse" name="parameters"/>
</wsdl:message>
<wsdl:portType name="StockQuote">
```
<wsdl:operation name="getQuote">
  <wsdl:input message="impl:getQuoteRequest" name="getQuoteRequest"/>
  <wsdl:output message="impl:getQuoteResponse" name="getQuoteResponse"/>
</wsdl:operation>

<wsdl:binding name="StockQuoteSoapBinding" type="impl:StockQuote">
  <wsdl:operation name="getQuote">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="getQuoteRequest">
      <wsdlsoap:body use="literal"/>
    </wsdl:input>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="GetQuoteService">
  <wsdl:port name="StockQuote" binding="impl:StockQuoteSoapBinding">
    <wsdlsoap:address location="http://localhost:9080/StockQuote/services/GetQuoteService"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>

The myGetQuote.cpp File

The following listing is for the C++ myGetQuote.cpp source file listing:

```cpp
// Web Services Client for ILE

/* LICENSE AND DISCLAIMER */
/* This material contains IBM copyrighted sample programming source code (Sample Code). */
/* IBM grants you a nonexclusive license to compile, link, execute, display, reproduce, distribute and prepare derivative works of this Sample Code. The Sample Code has not been thoroughly tested under all conditions. IBM, therefore, does not guarantee or imply its reliability, serviceability, or function. IBM provides no program services for the Sample Code. */
/* All Sample Code contained herein is provided to you "AS IS" without any warranties of any kind. THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED. */
/* SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSIONS MAY NOT APPLY TO YOU. IN NO EVENT WILL IBM BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES FOR ANY USE OF THE SAMPLE CODE INCLUDING, WITHOUT LIMITATION, ANY LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF PROGRAMS OR OTHER DATA ON YOUR INFORMATION HANDLING SYSTEM OR OTHERWISE, EVEN IF WE ARE EXPRESSLY ADVISED OF */
```
#include "StockQuote.hpp"
#include <iostream>

int main()
{
  try
  {
    char * pszEndpoint = "http://<Host>:<PortNumber>/StockQuote/services/urn:xmltoday-delayed-quotes";
    StockQuote * pwsStockQuote = new StockQuote( pszEndpoint);

    char * pszProxyURL = "<ProxyHost>";
    int iProxyPortNumber = <ProxyPort>;
    pwsStockQuote->setProxy( pszProxyURL, iProxyPortNumber);

    char * pszHandlerName = "Handler";
    pwsStockQuote->setTransportProperty( SOAPACTIONHEADER, pszHandlerName);

    char * pszStockName = "XXX";
    xsd__float fQuoteDollars = pwsStockQuote->getQuote( pszStockName);

    // Output the quote. If the stock name is unknown, then getQuote() will return -1. This name was recognized by the server and a constant value
  }
}
if( fQuoteDollars != -1) {
    cout << "The stock quote for " << pszStockName << " is $" << fQuoteDollars << endl;
} else {
    cout << "There is no stock quote for " << pszStockName << endl;
}

// Delete the web service.
delete pwsStockQuote;
}
catch( SoapFaultException& sfe) {
    // Catch any other SOAP faults
    cout << "SoapFaultException: " << sfe.getFaultCode() << " " << sfe.what() << endl;
} catch( AxisException& e) {
    // Catch an AXIS exception
    cout << "AxisException: " << e.getExceptionCode() << " " << e.what() << endl;
} catch( exception& e) {
    // Catch a general exception
    cout << "Unknown Exception: " << e.what() << endl;
} catch( ...) {
    // Catch any other exception
    cout << "Unspecified Exception: " << endl;
}

// Exit.
return 0;

The myGetQuote.c File

The following listing is for the C myGetQuote.c source file listing:

/***************************************************************************/
/* */
/* IBM Web Services Client for ILE */
/* */
/* FILE NAME: myGetQuote.c */
/* */
/* DESCRIPTION: main program to call the generated */
/* StockQuote stub */
/* */
/***************************************************************************/
/* LICENSE AND DISCLAIMER */
/* */
/* This material contains IBM copyrighted sample programming source */
/* code ( Sample Code ). */
/* IBM grants you a nonexclusive license to compile, link, execute, */
/* display, reproduce, distribute and prepare derivative works of */
/* this Sample Code. The Sample Code has not been thoroughly */
/* tested under all conditions. IBM, therefore, does not guarantee */
/* or imply its reliability, serviceability, or function. IBM */
/* provides no program services for the Sample Code. */
/* */
/* All Sample Code contained herein is provided to you "AS IS" */
/* without any warranties of any kind. THE IMPLIED WARRANTIES OF */
/* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND */
/* */
/* NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED. */
/* SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED */
/* WARRANTIES, SO THE ABOVE EXCLUSIONS MAY NOT APPLY TO YOU. IN NO */
/* EVENT WILL IBM BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, */
/* SPECIAL OR OTHER CONSEQUENTIAL DAMAGES FOR ANY USE OF THE SAMPLE */
/* CODE INCLUDING, WITHOUT LIMITATION, ANY LOST PROFITS, BUSINESS */
/* INTERRUPTION, LOSS OF PROGRAMS OR OTHER DATA ON YOUR INFORMATION */
/* HANDLING SYSTEM OR OTHERWISE, EVEN IF WE ARE EXPRESSLY ADVISED OF */
/* THE POSSIBILITY OF SUCH DAMAGES. */
/* */
/* <START_COPYRIGHT> */
/* */
/* Licensed Materials - Property of IBM */
/* */
/* 5724-M08 */
/* */
/* (c) Copyright IBM Corp. 2006, 2006 */
/* All Rights Reserved */
/* */
/* U.S. Government Users Restricted Rights - use, duplication or disclosure restricted by GSA */
/* */
/* ADP Schedule Contract with IBM Corp. */
/* */
/* Status: Version 1 Release 0 */
/* */
/* <END_COPYRIGHT> */
/* */
/*****************************************************************************************************/

#include <stdio.h>
#include <axis>
// Include the WSDL2Ws generated StockQuote.h
#include "StockQuote.h"

// Following function is used as stub exception handler
int globalExceptionOccurred = 0;
void StockQuoteExceptionHandler(int errorCode, char * errorString,
                                AXISCHANDLE soapFault, void *faultdetail)
{
    if (NULL != soapFault)
        printf("SoapFaultException: %d %s\n", axiscSoapFaultGetFaultcode(soapFault),
               axiscSoapFaultGetFaultstring(soapFault));
    else
        printf("AxisException: %d %s\n", errorCode, errorString);
    globalExceptionOccurred = 1;
}

int main()
{
    char * pszStockName;
    xsdc_float fQuoteDollars;

    // Create a character string that contains the server endpoint URI for the
    // GetQuoteService web service. Then pass the endpoint to the instantiator
    // for the GetQuote class that was generated by the WSDL2Ws tool. The
    // endpoint will pointing to the location of service on Websphere Application
    // Server.
    char * pszEndpoint = "http://<Host>:<PortNumber>/StockQuote/services/urn:xmltoday-delayed-quotes";
    AXISCHANDLE pwsStockQuote = get_StockQuote_stub(pszEndpoint);

    if (NULL == pwsStockQuote)
        return -1;

    // Set the stub exception handler function

Appendix A. Code Listings for myGetQuote Client Application 191
set_StockQuote_ExceptionHandler( pwsStockQuote, StockQuoteExceptionHandler );

// If your network requires the use of a proxy, then add the following line of
// code to configure AxisClient.
/*
axiscStubSetProxy(pwsStockQuote, "$ProxyHost", $ProxyPort);
*/

// Set the stock name to be quoted by the web service. To test just the
// web service, XXX is being used. This should return a stock quote of 55.25.
pszStockName = "XXX";

// Call the 'getQuote' method that is part of the StockQuote web service to
// find the quoted stock price for the given company whose name is in
// pszStockName. The result of the quote search will be returned by this
// method as a xsd__float type.
fQuoteDollars = getQuote(pwsStockQuote, pszStockName);

// Output the quote. If the stock name is unknown, then getQuote() will
// return -1. If name was recognized by the server a value is returned.
if (!globalExceptionOccurred)
{
    if ( fQuoteDollars != -1)
        printf("The stock quote for %s is $%f\n", pszStockName, fQuoteDollars);
    else
        printf("There is no stock quote for %s\n", pszStockName);
}

// Delete the web service.
destroy_StockQuote_stub(pwsStockQuote);

// Exit.
return 0;

The myGetQuote.rpgle File

The following listing is for the RPG myGetQuote.rpgle source file listing:

h DFTNAME(MYGETQUOTE)
*********************************************************************
**
* IBM Web Services Client for ILE
*
* FILE NAME: myGetQuote.rpgle
*
* DESCRIPTION: Source for GetQuote Web service client
*
*********************************************************************

* LICENSE AND DISCLAIMER
* ----------------------
* This material contains IBM copyrighted sample programming source
* code (Sample Code).
* IBM grants you a nonexclusive license to compile, link, execute,
* display, reproduce, distribute and prepare derivative works of
* this Sample Code. The Sample Code has not been thoroughly
* tested under all conditions. IBM, therefore, does not guarantee
* or imply its reliability, serviceability, or function. IBM
* provides no program services for the Sample Code.
*
* All Sample Code contained herein is provided to you "AS IS"
* without any warranties of any kind. THE IMPLIED WARRANTIES OF
* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
* NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED.
* SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED
/copy StockQuote.rpgleinc

**--------------------------------------------------------------------**

** Web service logic.**

**--------------------------------------------------------------------**

/clear WsStub;
WsStub.endpoint =
  'http://<ServerName>:<PortNumber>/StockQuote/services/+ urn:xmltoday-delayed-quotes';

// Set the stock name to be quoted by the web service. To test just the
// web service, XXX is being used. This should return a stock quote.
clear input;
input.value = 'XXX';

if (stub_create_StockQuote(WsStub) = *ON);
  // Invoke the StockQuote Web service operation.
  if (stub_op_getQuote(WsStub:Input:Result) = *ON);
    OutputText = 'The stock quote for ' + Input.value + ' is ' + %CHAR(Result.value);
  else;
    OutputText = WsStub.excString;
  endif;
  // Display results.
display OutputText;
else;
  // Destroy Web service stubs.
  stub_destroy_StockQuote(WsStub);
**--------------------------------------------------------------------**
endif;

*INLR*=ON;
/end-free
Appendix B. Code Listings for Client Handler

These code samples provide templates that demonstrate how you can create handlers for a client application. Table 31 shows a list of the files.

Table 31. Handler files in the samples directory

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>client.wsdd</td>
<td>The WSDD file that defines the client handler.</td>
</tr>
<tr>
<td>myClientHandler.hpp</td>
<td>Client handler implementation header file.</td>
</tr>
<tr>
<td>myClientHandler.cpp</td>
<td>Client handler implementation file.</td>
</tr>
<tr>
<td>myClientHandlerFactory.cpp</td>
<td>Client handler factory implementation.</td>
</tr>
</tbody>
</table>

All files can be found in <install_dir>/samples/handlers.

The client.wsdd File

The following listing is for the client.wsdd source file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:C="http://xml.apache.org/axis/wsdd/providers/c">
    <service name="Handler" provider="CPP:DOCUMENT" description="Handler">
        <requestFlow>
            <handler name="myClientHandlerreq" type="/qsys.lib/sample.lib/handler.srvpgm"/>
        </requestFlow>
        <responseFlow>
            <handler name="myClientHandlers" type="/qsys.lib/sample.lib/handler.srvpgm"/>
        </responseFlow>
    </service>
</deployment>
```

The myClientHandler.hpp File

The following listing is for the myClientHandler.hpp source file:

```c
/****************************************************************************
/*                                                                            */
/* IBM Web Services Client for C/C++                                       */
/*                                                                            */
/* FILE NAME: myClientHandler.hpp                                        */
/*                                                                            */
/* DESCRIPTION: Example Client handler header file                       */
/* for the Stock Quote sample                                              */
/*                                                                            */
/****************************************************************************
/* LICENSE AND DISCLAIMER */
/* ---------------------- */
/* This material contains IBM copyrighted sample programming source */
/* code (Sample Code). */
/* IBM grants you a nonexclusive license to compile, link, execute, */
/* display, reproduce, distribute and prepare derivative works of */
/* this Sample Code. The Sample Code has not been thoroughly */
/* tested under all conditions. IBM, therefore, does not guarantee */
/* or imply its reliability, serviceability, or function. IBM */
/* provides no program services for the Sample Code. */
/* */
/* All Sample Code contained herein is provided to you "AS IS" */
/* without any warranties of any kind. THE IMPLIED WARRANTIES OF */
```

© Copyright IBM Corp. 2011, 2012
The myClientHandler.cpp File

The following listing is for the myClientHandler.cpp source file:

```cpp
#include <axis>

class myClientHandler : public Handler
{
 public:
  myClientHandler();
  virtual ~myClientHandler();

  // init is called when the Handler is loaded.
  int AXISCALL init();

  // invoke is called when AxisClient is about to send the request SOAP message
  // or when a response message has just been received.
  int AXISCALL invoke( void * pvIMsg);

  // onFault is called if there is a fault with message processing.
  void AXISCALL onFault( void * pvIMsg);

  // fini is called when the Handler is about to unloaded.
  int AXISCALL fini();
};
```

The following listing is for the myClientHandler.cpp source file:
/**
/* DESCRIPTION: Example Client Handler
/* for the stock quote sample
/*
/*****************************************************************
/* LICENSE AND DISCLAIMER
/* ----------------------
/* This material contains IBM copyrighted sample programming source
/* code (Sample Code).
/* IBM grants you a nonexclusive license to compile, link, execute,
/* display, reproduce, distribute and prepare derivative works of
/* this Sample Code. The Sample Code has not been thoroughly
/* tested under all conditions. IBM, therefore, does not guarantee
/* or imply its reliability, serviceability, or function. IBM
/* provides no program services for the Sample Code.
/*
/* All Sample Code contained herein is provided to you "AS IS"
/* without any warranties of any kind. THE IMPLIED WARRANTIES OF
/* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
/* NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED.
/* SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED
/* WARRANTIES, SO THE ABOVE EXCLUSIONS MAY NOT APPLY TO YOU. IN NO
/* EVENT WILL IBM BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT,
/* SPECIAL OR OTHER CONSEQUENTIAL DAMAGES FOR ANY USE OF THE SAMPLE
/* CODE INCLUDING, WITHOUT LIMITATION, ANY LOST PROFITS, BUSINESS
/* INTERRUPTION, LOSS OF PROGRAMS OR OTHER DATA ON YOUR INFORMATION
/* HANDLING SYSTEM OR OTHERWISE, EVEN IF WE ARE EXPRESSLY ADVISED OF
/* THE POSSIBILITY OF SUCH DAMAGES.
/*
/* <START_COPYRIGHT>
/* Licensed Materials - Property of IBM
/* 5724-M08
/* (c) Copyright IBM Corp. 2004, 2005
/* All Rights Reserved
/* U.S. Government Users Restricted Rights - use,
/* duplication or disclosure restricted by GSA
/* ADP Schedule Contract with IBM Corp.
/* Status: Version 1 Release 0
/* <END_COPYRIGHT>
/*****************************************************************

// Include myClientHandler header file to obtain the class definition, etc.
#include "myClientHandler.hpp"

// Include the header file to obtain the BasicHandler object, etc.
#include <axis>
#include <iostream>

// myHandler is called when the object is created.
myClientHandler::myClientHandler()
{
}

// "myClientHandler is called when the object is destroyed.
myClientHandler::~myClientHandler()
{
int myClientHandler::invoke( void * pvHandlerMessage)
{
    // Cast the current message into the IMessageData type. This will allow the
    // user to change the SOAP message as appropriate.
    IMessageData * pIMsgData = (IMessageData *) pvHandlerMessage;

    // Check if the SOAP message is just about to be transmitted or has just been
    // received.
    if( pIMsgData->isPastPivot())
    {
        // Yes - the available SOAP message is a response
        cout << "Past the pivot point - Handler can see the response message." << endl;
    }
    else
    {
        // No - the available SOAP message is a request
        cout << "Before the pivot point - Handler can see the request message\n" << endl;
    }

    return AXIS_SUCCESS;
}

void myClientHandler::onFault( void * pvFaultMessage)
{
    // Please leave empty.
}

int myClientHandler::init()
{
    return AXIS_SUCCESS;
}

int myClientHandler::fini()
{
    return AXIS_SUCCESS;
}

The myClientHandlerFactory.cpp File

The following listing is for the myClientHandlerFactory.cpp source file:

/*********************************************************************/
/* */
/* IBM Web Services Client for C/C++ */
/* */
/* FILE NAME: myClientHandlerFactory.cpp */
/* */
/* DESCRIPTION: Example client handler factory */
/* for the stock quote sample */
/* */
/*********************************************************************/
/* LICENSE AND DISCLAIMER */
/* *************************/
/* This material contains IBM copyrighted sample programming source */
/* code ( Sample Code ). */
/* IBM grants you a nonexclusive license to compile, link, execute, */
/* display, reproduce, distribute and prepare derivative works of */
/* this Sample Code. The Sample Code has not been thoroughly */
/* tested under all conditions. IBM, therefore, does not guarantee */
/* or imply its reliability, serviceability, or function. IBM */
/* provides no program services for the Sample Code. */
/* */
/* All Sample Code contained herein is provided to you "AS IS" */
/* without any warranties of any kind. THE IMPLIED WARRANTIES OF */
/* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND */
/* NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED. */
/* SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED */

/* WARRANTIES, SO THE ABOVE EXCLUSIONS MAY NOT APPLY TO YOU. IN NO */
/* EVENT WILL IBM BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, */
/* SPECIAL OR OTHER CONSEQUENTIAL DAMAGES FOR ANY USE OF THE SAMPLE */
/* CODE INCLUDING, WITHOUT LIMITATION, ANY LOST PROFITS, BUSINESS */
/* INTERRUPTION, LOSS OF PROGRAMS OR OTHER DATA ON YOUR INFORMATION */
/* HANDLING SYSTEM OR OTHERWISE, EVEN IF WE ARE EXPRESSLY ADVISED OF */
/* THE POSSIBILITY OF SUCH DAMAGES. */
/* */
/* <START_COPYRIGHT> */
/* */
/* Licensed Materials - Property of IBM */
/* */
/* 5724-M08 */
/* */
/* (c) Copyright IBM Corp. 2004, 2005 */
/* */
/* All Rights Reserved */
/* */
/* U.S. Government Users Restricted Rights - use, */
/* duplication or disclosure restricted by GSA */
/* */
/* ADP Schedule Contract with IBM Corp. */
/* */
/* Status: Version 1 Release 0 */
/* <END_COPYRIGHT> */
/* */
/* ************************************************************************/ */

// Include myClientHandler header file to obtain the class definition, etc.
#include "myClientHandler.hpp"

// Include the header file to obtain the BasicHandler object, etc.
#include <axis>

// External methods available to the loader of this handler library.
extern "C"
{
    // GetClassInstance is passed a pointer to a pointer that will contain the
    // handler object to be created by this factory. Before the handler object is
    // returned, it is wrapped in a BasicHandler object and the handler's
    // initialise method is called.
    STORAGE_CLASS_INFO int GetClassInstance( BasicHandler ** ppClassInstance)
    {
        *ppClassInstance = new BasicHandler();

        myClientHandler * pmyClientHandler = new myClientHandler();

        // Setting functions to zero indicates that the handler is a C++ type
        (*ppClassInstance)->_functions = 0;

        // If the handler was loaded successfully, save the handler object and
        // initialise it.
        if( pmyClientHandler)
        {
            (*ppClassInstance)->object = pmyClientHandler;

            return pmyClientHandler->init();
        }

        // If the handler was not loaded successfully, then return an error.
        return AXIS_FAIL;
    }

    // DestroyInstance is passed a pointer to a generic BasicHandler object that
    // contains an instance of this type of handler object. The handler is
    // unwrapped from the BasicHandler object whereupon, the handler's finish
    // method is called before deleting the handler and then the BasicHandler
    // wrapper.
    STORAGE_CLASS_INFO int DestroyInstance( BasicHandler * pClassInstance)
    {
if (pClassInstance)
{
    //Cast the generic handler object to the specific class.
    myClientHandler * pmyClientHandler = static_cast<myClientHandler> (pClassInstance->_object);

    // Call the finish method on the handler. This will allow the handler to
    // 'tidy' before it is deleted.
    pmyClientHandler->fini();

    // Delete the handler objects.
    delete pmyClientHandler;
    delete pClassInstance;

    // Return success.
    return AXIS_SUCCESS;
}

// Return error if there was no handler to close down and delete.

return AXIS_FAIL;
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing 2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM’s suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.
If you are viewing this information softcopy, the photographs and color illustrations may not appear.

**Trademarks**

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
AIX 5L

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.
### Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI</td>
<td>American National Standard for Information Systems</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>attachment</td>
<td>Data that is attached to a message on the wire, separately from the SOAP envelope. Attachments are often used for sending large files or images.</td>
</tr>
<tr>
<td>AXISCPP_DEPLOY</td>
<td>Environment variable that points to the installation directory, referred to as &lt;install_dir&gt; in this documentation</td>
</tr>
<tr>
<td>client engine</td>
<td>A set of libraries that are made available at runtime to the client stubs</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Link Library</td>
</tr>
<tr>
<td>handler</td>
<td>A library component that has the ability to manipulate a SOAP message, thus allowing the user to customize or extend any message components. Handlers are invoked either just before a request message is transmitted or just after a response message has been received</td>
</tr>
<tr>
<td>global handler</td>
<td>A handler that is called regardless of the web service or message name</td>
</tr>
<tr>
<td>GSKit</td>
<td>Global Security Kit, IBM's SSL component</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronic Engineers</td>
</tr>
<tr>
<td>pivot point</td>
<td>The point where a message is either written on to or read from the wire</td>
</tr>
<tr>
<td>post-pivot handler</td>
<td>A handler that works on a response message after it has been received</td>
</tr>
<tr>
<td>pre-pivot handler</td>
<td>A handler that works on a request message that is to be transmitted</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>service handler</td>
<td>A handler that is specific to the web service with which it is associated</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Sockets Layer</td>
</tr>
<tr>
<td>TCPIP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>wire</td>
<td>All the underlying components that are responsible for physically sending or receiving a message on the web</td>
</tr>
<tr>
<td>WSDD</td>
<td>Web Service Deployment Descriptor. An XML style file containing information that Web Services Client for C/C++ uses as it builds request messages and decodes response messages</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Description Language. WSDLs are XML files containing all the information relating to services that are available at a particular location on the internet</td>
</tr>
<tr>
<td>WSDL2Ws</td>
<td>Java tool that converts a WSDL into a set of client stubs that can be called by the client application</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Mark-up Language</td>
</tr>
<tr>
<td>XML4C</td>
<td>eXtensible Mark-up Language for C/C++</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 2011, 2012
wsdl2ws.sh tool (continued)
  C example  123
  C++ example  75
  RPG example  173
  troubleshooting  95, 137, 183

X
XML  3
  attribute  11
  definition  11
  document  11
  element  11
  namespace  12
  Naming rules  12
XML schema  14
  complex types  16
  elements  14
  simple types  14