IBM i
Version 1

Web Services Client for ILE Programming Guide

Note

Before using this information and the product it supports, read the information in “Notices” on page 219.

Sixth Edition (June 2018)

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.

Preface

Web Services Client for ILE provides a set of libraries and Java™ tools that enable you to build web service client applications. 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. It is not intended that the reader be a guru of either web technology or web services in order to find this document of value.

Conventions used in this book

Italics 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:

```
http://www.ibm.com/developerworks/ibmi/techupdates/iws
```

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 2018**
  - New information has been added to document new option (AXISC_PROPERTY_HTTP_HEADERS_RESPONSE) in the "axiscTransportGetProperty()" on page 174 transport API.
  - APIs relating to set SSL information have been updated to ensure the a NULL pointer is passed as the last parameter. The APIs affected include the "axiscStubSetSecure()" on page 161, "axiscTransportSetProperty()" on page 171, and "Stub::SetSecure()" on page 109 APIs.
- **February 2018**
  - New information has been added to document the enhancements relating to SSL connections, which includes the ability to specify application ID instead of certificate keystore path and Server Name Indication (SNI) in the "axiscStubSetSecure()" on page 161, "axiscTransportSetProperty()" on page 171, and "Stub::SetSecure()" on page 109 APIs.
  - New information has been added to document the ability of RPG stub applications to retrieve SOAP fault information as part as of the "SOAP fault C APIs" on page 168.
  - New information has been added to document new conversion options in the "axiscTransportSetProperty()" on page 171 transport API.
- **February 2017**
  - New information has been added to document enhancements to allow using SSL when connecting to a proxy server. See documentation updates for the C axiscStubSetProxySSL(), C axiscTransportSetProperty(), and C++ setTransportProperty() APIs.
  - New information has been added to document the ability to allow the establishment of SSL connections even if the SSL certificate is expired or not in the certificate store in the C axiscStubSetSecure(), C axiscTransportSetProperty(), and Axis C++ core APIs APIs.
- **August 2016**
  - New sections have been added to document new support for sending user-defined (e.g. REST and SOAP) requests in the Axis C core APIs chapter.
  - New information has been added to document enhancements to the SSL support for TLS v1.1 and TLS v1.2 in the Axis C core APIs and Axis C++ core APIs chapters.
- **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.

Contents

Part 1. Web service fundamentals

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

Chapter 1. What is a web service?

A *web service* enables the sharing of logic, data, and processes across networks using a programming interface.

Some of the key features of web services are the following:

- Web services are self-contained.

  On the client side, no additional software is required. A programming language with XML (Extensible Markup Language) and HTTP client support, for example, is enough to get you started. On the server side, merely a web server or application sever is required. It is possible to web service enable an existing application without writing a single line of code.

- Web services are self-describing.

  Neither the client nor the server knows or cares about anything besides the format and content of request and response messages (loosely coupled application integration). The definition of the message format travels with the message. No external metadata repositories is required.

- Web services are modular.

  Web services are a technology for deploying and providing access to business functions over the Web; J2EE (Java 2 Enterprise Edition), CORBA (Common Object Request Broker Architecture), and other standards are technologies for implementing these web services.

- Web services can be published (externalized), located, and invoked across the Web.

  All you need to access the web service from a client perspective is a URI (Uniform Resource Identifier).

- Web services are language independent and interoperable.

  The interaction between a service provider and a service requester is designed to be completely platform and language independent. This interaction requires a document to define the interface and describe the service. Because the service provider and the service requester have no idea what platforms or languages the other is using, interoperability is a given.

- Web services are inherently open and standards based.

  XML, JSON (JavaScript Object Notation) and HTTP are the technical foundation for web services. Using open standards provides broad interoperability among different vendor solutions. These principles mean that companies can implement web services without having any knowledge of the service requesters, and service requesters do not need to know the implementation specifics of service provider applications. This use of open standards facilitates just-in-time integration and allows businesses to establish new partnerships easily and dynamically.

- Web services are composable.

  Simple web services can be aggregated to more complex ones, either using workflow techniques or by calling lower-layer web services from a web service implementation.

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. 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.

Why web services?

Why should you care about web services? One reason is that web services is well suited to implementing a *Service-Oriented Architecture* (SOA). SOA is a business-centric information technology (IT) architectural approach that supports integrating your business as linked, repeatable business tasks, or services. Within this type of architecture, you can orchestrate the business services in business processes. Adopting the concept of services—a higher-level abstraction that's independent of application or infrastructure IT platform and of context or other services—SOA takes IT to another level, one that's more suited for interoperability and heterogeneous environments.

Because an SOA is built on standards acknowledged and supported by the major IT providers, such as web services, you can quickly build and interconnect its services. You can interconnect between enterprises regardless of their supported infrastructure, which opens doors to delegation, sharing, reuse, and maximizing the benefits of your existing assets.

With an SOA established, you bring your internal IT infrastructure to a higher, more visible, and manageable level. With reusable services and high-level processes, change is easier than ever and is more like disassembling and reassembling parts (services) into new, business-aligned processes. This not only promotes efficiency and reuse, it provides a strong ability to change and align IT with business. Figure 1 on page 4 shows web services in action. The operational systems layer shows the data and applications that contain the information to be delivered as a service. The services layer shows the services that enable the operational layer to be delivered as a service. The business process layer shows how web services can be linked together to create highly flexible and automated business processes. The people and application layer shows how web services are used to create web applications and dashboards. It is all about efficiency in creation, reuse for execution, and flexibility for change and growth.

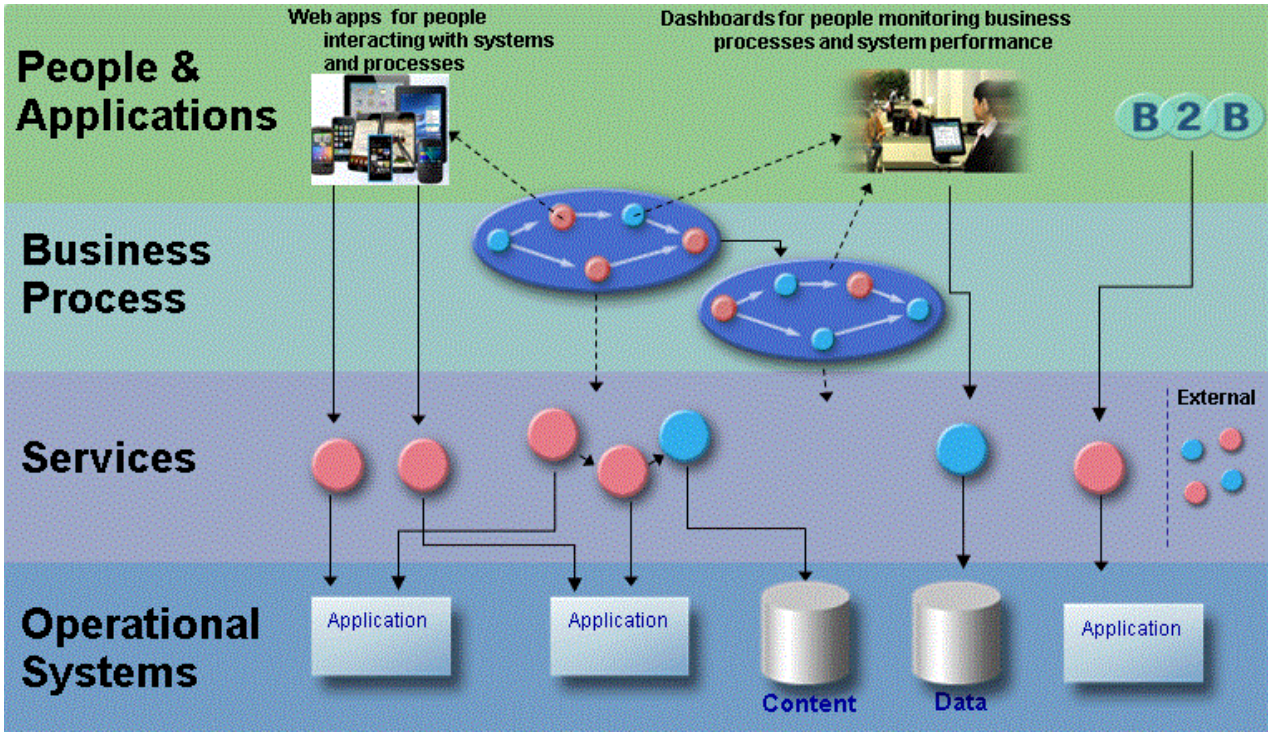


Figure 1: Web services in action

Another reason web services are important is due to web services that is commonly known as web Application Program Interfaces (APIs). An API is a public persona for a company, exposing defined assets, data, or services for public consumption.

In the 1990's when the World Wide Web (WWW) was relatively new many companies focused their business toward creating a web presence. As Internet access became more readily available, speed limitations lifted, and technology improved, many companies migrated from a relatively flat and static web presence to a more dynamic, content rich and interactive approach. Today we live in a data centric

world of connected devices where we expect data to be readily available at our fingertips. These devices include, but are not limited to, smart phones, tablets, games consoles, and even cars and refrigerators. As the number of devices has increased, so too has the complexity to manage and maintain the code for each of these devices and this is where an “API First” approach has really gained the most traction. Exposing the data via a common API allows a single point of maintenance, security, versioning and control. In this way data can be exposed consistently across multiple devices. APIs can help companies expose data that they wish to make available to the outside world or select business partners. These APIs can be used to create applications as well as act as a powerful means to market a company’s product and to help carve out new market opportunities. Once APIs are established they can be used to drive brand awareness and increase profit. Most importantly the APIs, which are a now a core part of the business also need to be treated as a product. Whether or not you or your company are considering exposing APIs it is very likely one of your competitors are. In the highly competitive world we live in today, this in itself is a significant reason to start considering an API strategy.

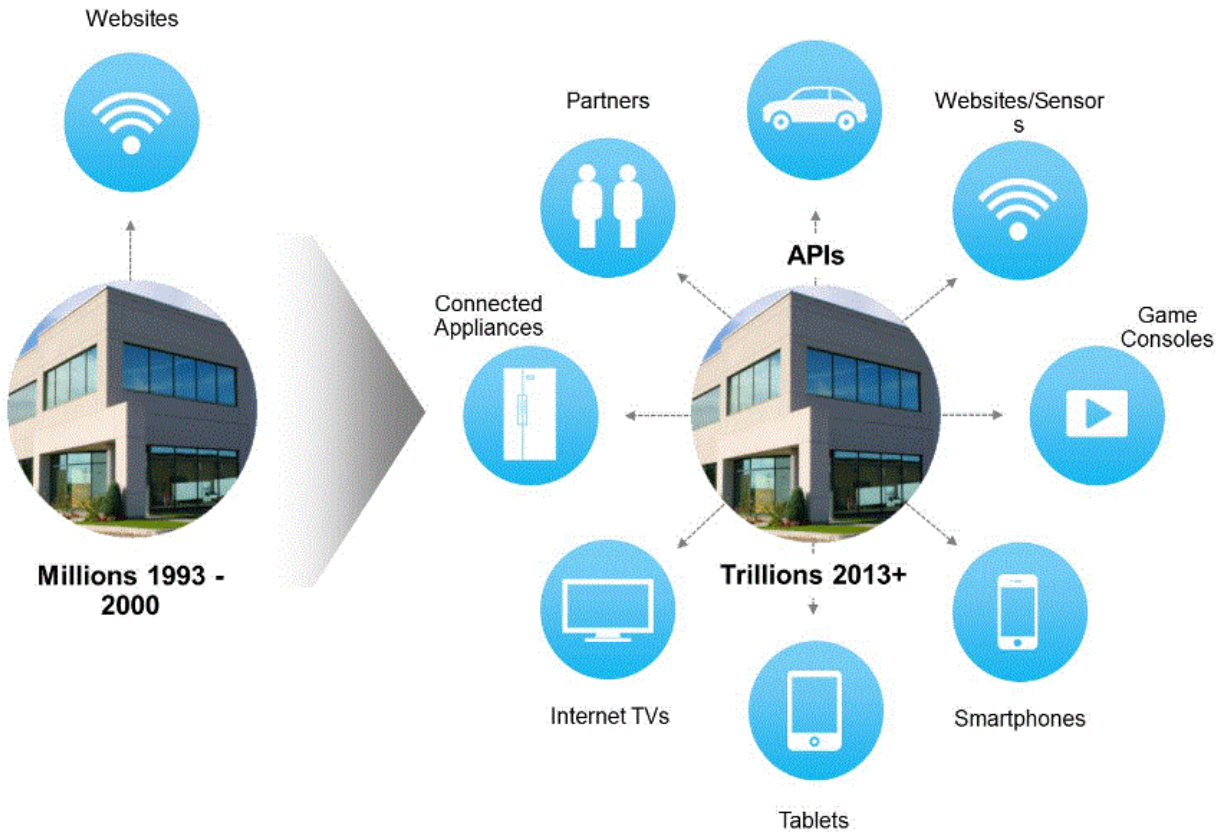


Figure 2: APIs in action

Figure 2 on page 5 shows a fundamental shift from websites as being the information technology access mechanism for the majority of businesses, to the rapidly growing ecosystem of interconnected devices that require APIs to consume business function. Today, we have applications in cars, appliances, smartphones, game consoles, and other devices, that communicate with back-end business functions through APIs. This “interconnected revolution” is here today: refrigerators can tell their manufacturer services systems when maintenance is required; cars can do the same with routine maintenance notification; and smart electric meters can provide usage and consumption information to the utility company.

All of this is possible through web APIs.

Chapter 2. Types of web services

A web service is composed of operations that are offered in one of two styles:

- A web service based on the Service Object Access Protocol (SOAP) protocol.
- A web service that follows the principles of Representational State Transfer (REST).

The following sections discusses each of the types of web services.

SOAP-based web services

A SOAP-based 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.

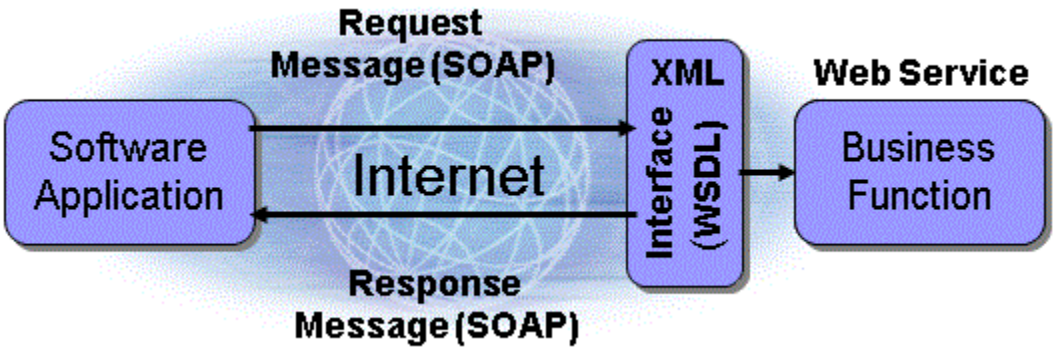


Figure 3: SOAP-based web services

Core technologies and standards

Several key technologies and standards exist within the SOAP-style web services community:

- XML, 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. For information about XML, see “XML primer” on page 8.
- SOAP, a standard protocol for exchanging XML messages. It also details the way applications should treat certain aspects of the message, such as elements in the "header", which enable you to create applications in which a message is passed between multiple intermediaries before reaching its final destination. For information about SOAP, see “SOAP primer” on page 17.
- WSDL, a specification that details a standard way to describe a SOAP-based web service, including the form the messages should take, and where they should be sent. It also details the response to such a message. For information about WSDL, see “WSDL primer” on page 24.

The big interoperability question: can web services continue to interoperate as the various standards they rely on 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 and SOAP Version 1.1 for their web services. Company Y has decided to use WSDL Version 1.1 and SOAP Version 1.2. Even though both companies are using web services, a client 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 enterprises begin to apply web services technologies to solve their integration and interoperability problems, they increasingly find that they require more advanced features such as security, reliable messaging, management and transactional capabilities. Some of these quality-of-service capabilities demand interoperable infrastructure services for such things as metadata, trust, resource management, event notification, and coordination services. The majority of today's deployed SOAP-style web services are limited to use of only the foundation technologies of SOAP, WSDL, and XML. However, SOAP-style web services provides a broad range of capabilities that compose with the foundation to provide more advanced qualities of service, infrastructure services and service composition. The quality of service extensions to the base SOAP-style web services standards include:

- WS-Addressing, which defines a standardized endpoint reference schema type and a set of message addressing properties that can be used in conjunction with the SOAP process model to effect a broad range of message exchange patterns beyond the simple request/response.
- WS-PolicyFramework, which provides a framework for articulating policy constraints of a service endpoint and a framework for attaching such policy constraints to WSDL and other web services artifacts.
- WS-Security, which provides a framework for an entire family of security specifications including WS-Secure Conversation providing session-based security capabilities and WS-Trust providing a standardized interface to a trust service.
- WS-ReliableMessaging, providing for the reliable exchange of messages between web services endpoints.
- WS-AtomicTransactions, which handles short-lived transactional activities.

For more information on the web services standards, consult an online reference of web services standards, such as is hosted on the IBMdeveloperWorks® web site, available at:

```
http://www.ibm.com/developerworks/webservices/standards/
```

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:

```
https://www.w3.org/TR/xml/
```

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 version="1.1"?> 1
<!-- Complete address tag --> 2
<Address>
   <Name> 3
     <Title>Mrs.</Title>
     <First-Name>Ashley</First-Name>
     <Middle-Name/> 4
     <Last-Name>Adams</Last-Name>
     <Phone>777-444-2222</Phone>
   </Name> 5
   <Street>123 Corporation Avenue</Street>
   <City state="NC">Hometown</City> 6
   <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:

```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope">
<Header>
  <n:AlertControl xmlns:n="http://ibm.com/alertcontrol">
   <n:Priority>1</n:Priority>
  </n:AlertControl>
 </Header>
 <Body>
  <m:Alert xmlns:m="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

```
<Books xmlns:BookInfo="http://www.ibm.com/BookInformation"
       xmlns:BookContent="http://www.ibm.com/BookContent"
       xmlns ="http://www.ibm.com/BookDefault" >

   <Book>
      <BookInfo:Name>Understanding Namespaces</BookInfo:Name>
      <Author>Whizlabs</Author>
      <BookInfo:ISBN>s677-898-765-098</BookInfo:ISBN>
      <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 1: Mapping of element names to namespaces

Element	**Namespace**
<Book>	http://www.ibm.com/BookDefault
<Name>	http://www.ibm.com/BookInformation
<Author>	http://www.ibm.com/BookDefault
<ISBN>	http://www.ibm.com/BookInformation
<Price>	http://www.ibm.com/BookContent
<Publisher>	http://www.ibm.com/Publishers

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” on page 12 and “Schema Elements - Complex Types” on page 13

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:

```
<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:

```
<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 Recommendation[1] is shown below:

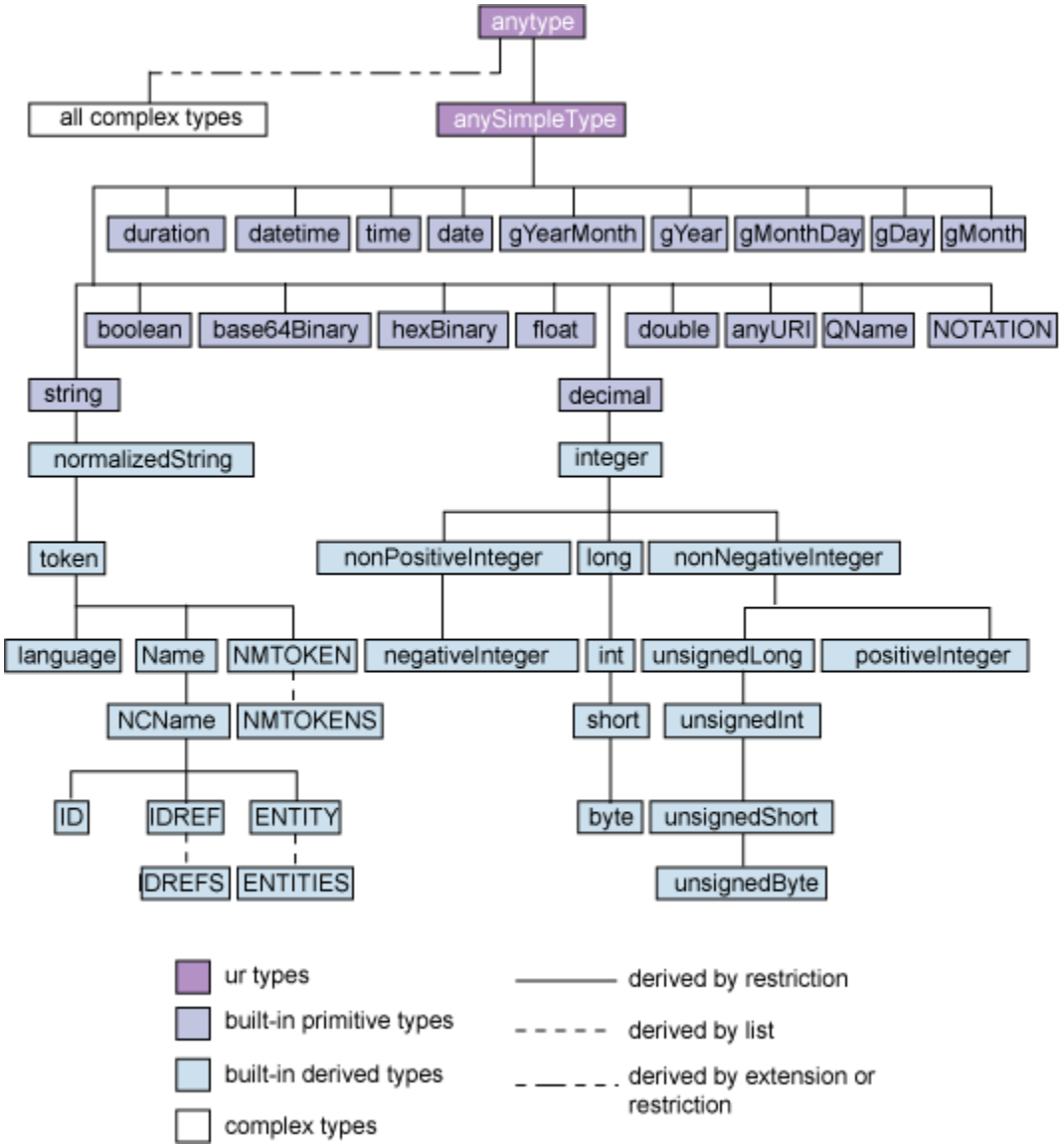


Figure 4: XML schema datatypes

Schema Elements - Complex Types

Complex types are elements that either:

- Contain other elements
- Contain attributes
- Are empty (empty elements)
- Contain text

[1] Copyright 2003 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231

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" on page 14 for more on these indicators.) For example:

```
<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:

- **`All`**: If `All` 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 `All` 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:

```
<xs:element name="Book">
   <xs:complexType>
      <xs:all>
         <xs:element name="Name" type="xs:string" />
         <xs:element name="ID" type="xs:string"/>
         <xs:element name="Authors" type="authorType"/>
         <xs:element name="Price" type="priceType"/>
      </xs:all>
   </xs:complexType>
</xs:element>

<xs:complexType name="authorType">
      <xs:sequence>
         <xs:element name="Author" type="xs:string" maxOccurs="4"/>
      </xs:sequence>
</xs:complexType >

<xs:complexType name="priceType">
   <xs:choice>
      <xs:element name="dollars" type="xs:double" />
      <xs:element name="pounds" type="xs:double" />
   </xs:choice>
</xs:complexType >
```

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:

```
<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.

  `minExclusive` Numeric value of the data type is greater than the value specified. For example:

```
<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:

```
<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:

```
<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:

```
<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:

```
<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:

```
<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 "/>
```

The syntax for `include` is:

```
<xs:include id="ID_DATATYPE" schemaLocation="anyURI_DATATYPE"/>
```

SOAP primer

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.

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.

The following sections will cover the SOAP 1.1 specification and the SOAP architecture in detail. For more information on SOAP (including SOAP 1.2), go to the following URL:

```
https://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 DOCTYPE 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.

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 18 shows the namespaces of SOAP and WS-I Basic Profile 1.0.

Table 2: SOAP namespaces

Prefix	Namespace URI	Explanation
SOAP-ENV	http://schemas.xmlsoap.org/soap/envelope/	SOAP 1.1 envelope namespace
SOAP-ENC	http://schemas.xmlsoap.org/soap/encoding/	SOAP 1.1 encoding namespace
	http://www.w3.org/2001/XMLSchema-instance	Schema instance namespace
	http://www.w3.org/2001/XMLSchema	XML Schema namespace
	http://schemas.xmlsoap.org/wsdl	WSDL namespace for WSDL framework
	http://schemas.xmlsoap.org/wsdl/soap	WSDL namespace for WSDL SOAP binding

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 5 on page 18). This is an XML document that includes all of the information necessary to process the message.

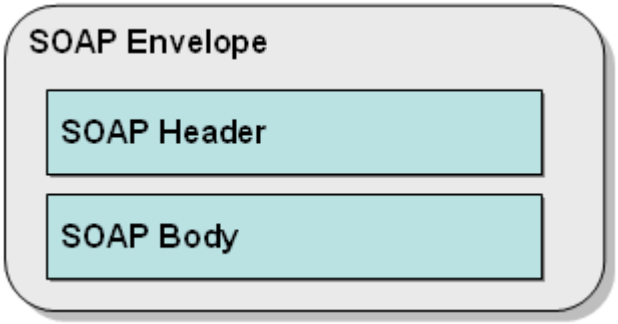


Figure 5: SOAP envelope

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 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 version='1.0' ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
 <env:Header>
   <wsa:ReplyTo xmlns:wsa=
      "http://schemas.xmlSOAP.org/ws/2004/08/addressing">
     <wsa:Address>
       http://schemas.xmlSOAP.org/ws/2004/08/addressing/role/anonymous
     </wsa:Address>
   </wsa:ReplyTo>
   <wsa:From>
     <wsa:Address>
       http://localhost:8080/axis/services/MyService</wsa:Address>
     </wsa:From>
    <wsa:MessageID>ECE5B3F187F29D28BC11433905662036</wsa: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):

```
<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:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
<env:Header>
...
</env:Header>
<env:Body>
  <cms:addArticle xmlns:cms="http://www.ibm.com/cms">
    <cms:category>classifieds</category>
    <cms:subcategory>forsale</cms:subcategory>
    <cms:articleHeadline></cms:articleHeadline>
    <cms:articleText>Vintage 1963 T-Bird.</cms:articleText>
  </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 XML elements inside the SOAP fault element are different in SOAP 1.1 and SOAP 1.2. In SOAP 1.1, the `<Fault>` element contains the following elements:

- `<faultcode>` is a mandatory element in the `<Fault>` element. It provides information about the fault in a form that can be processed by software. 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 mandatory element in the <Fault> element. It provides information about the fault in a form intended for a human reader.
- <faultactor> contains the URI of the SOAP node that generated the fault. A SOAP node that is not the ultimate SOAP receiver must include the <faultactor> element when it creates a fault. An ultimate SOAP receiver is not obliged to include this element, but may do so.
- <detail> carries application-specific error information related to the <Body> element. It must be present if the contents of the <Body> element were not successfully processed. It must not be used to carry information about error information belonging to header entries. Detailed error information belonging to header entries must be carried in header entries.

In SOAP 1.2, the <Fault> element contains the following elements:

- <Code> is a mandatory element in the <Fault> element. It provides information about the fault in a form that can be processed by software. It contains a <Value> element and an optional <Subcode> element.
- <Reason> is a mandatory element in the <Fault> element. It contains one or more <Text> elements, each of which contains information about the fault in a different native language.
- <Node> contains the URI of the SOAP node that generated the fault. A SOAP node that is not the ultimate SOAP receiver must include the <Node> element when it creates a fault. An ultimate SOAP receiver is not obliged to include this element, but may do so.
- <Role> contains a URI that identifies the role in which the node was operating at the point the fault occurred.
- <Detail> is an optional element, which contains application-specific error information related to the SOAP fault codes describing the fault. The presence of the <Detail> element has no significance regarding which parts of the faulty SOAP message were processed.

Here is an example of a SOAP 1.1 fault response message:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<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-referrence">
          <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 of a compound datatype named `Mobile`.

```
<? 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"> 1
   <xsd:complexType> 2
       <xsd:element name="modelNumber" type="xsd:int"> 3
       <xsd:element name="modelName"  type="xsd:string"> 4
        <xsd:element name="modelColor"> 5
           <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.

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

SOAP binding and encoding styles

You'll get deeper into this subject in “WSDL primer” on page 24, but as you create your application, you will need to decide[2] 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 integrated web services support, the decision has been made for you! But for completeness we discuss what is available. Integrated web services for i 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 and example of an RPC/encoded message.

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

A document/literal style of message simply involves adding the message:

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

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 “WSDL primer” on page 24.

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 rest of the discussion in this chapter will be from the perspective of the WSDL 1.1 specification. Information on WSDL 1.1 and WSDL 2.0 can be found at the following URLs:

```
https://www.w3.org/TR/wsdl

https://www.w3.org/TR/wsdl20-primer/
```

WSDL 1.1 document structure

WSDL conventionally divides the basic service description into two parts (see Figure 6 on page 25): the service interface and the service implementation. This enables each part to be defined separately and independently, and reused by other parts.

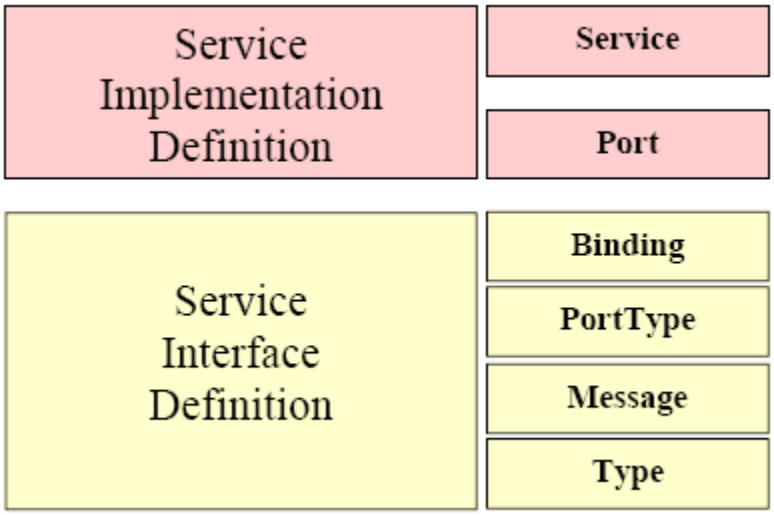


Figure 6: Basic service description

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. Now lets dive into the details.

Figure 7 on page 26 shows the elements comprising a WSDL document and the various relationships between them.

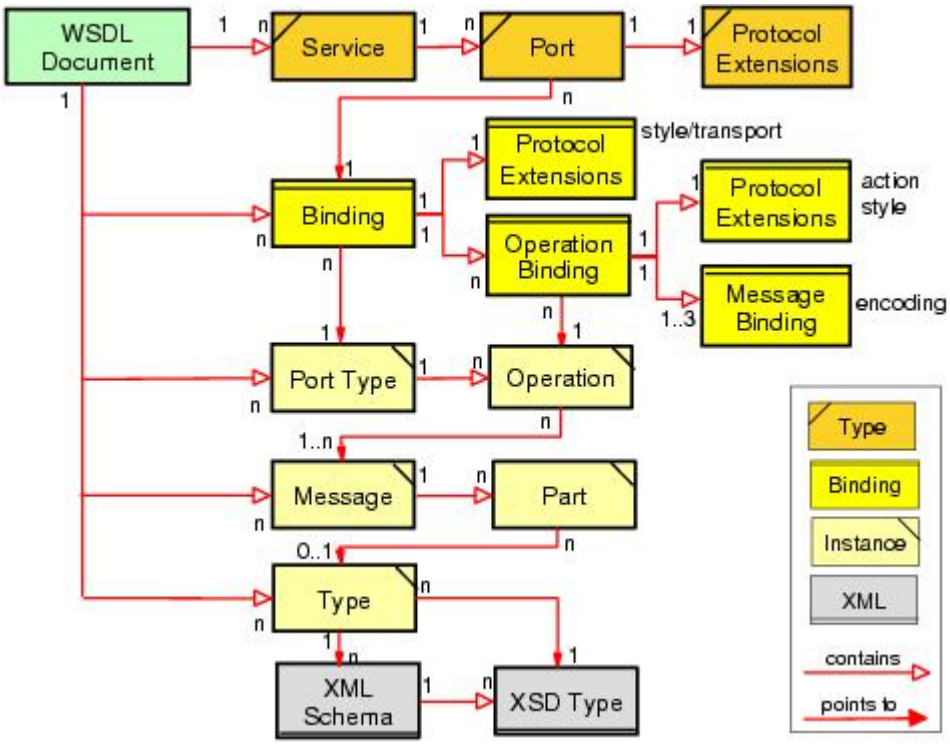


Figure 7: WSDL 1.1 elements and relationships

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.

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

<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=""/>
      <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>

 <wsdl:service name="AddressServiceService">
      <wsdl:port binding="intf:AddressSoapBinding" name="Address">
         <wsdlsoap:address
            location="http://localhost:8080/axis/services/Address"/>
      </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

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" on page 28):

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

<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 uses the XML namespaces listed in Table 3 on page 28.

Table 3: WSDL namespaces

Prefix	Namespace URI	Explanation
wsdl	http://schemas.xmlsoap.org/wsdl/	Namespace for WSDL framework.
soap	http://schemas.xmlsoap.org/wsdl/soap/	SOAP binding.
http	http://schemas.xmlsoap.org/wsdl/http/	HTTP binding.
mime	http://schemas.xmlsoap.org/wsdl/mime/	MIME binding.
soapenc	http://schemas.xmlsoap.org/soap/encoding/	Encoding namespace as defined by SOAP 1.1.
soapenv	http://schemas.xmlsoap.org/soap/envelope/	Envelope namespace as defined by SOAP 1.1.
xsi	http://www.w3.org/2000/10/XMLSchema-instance	Instance namespace as defined by XSD.
xsd	http://www.w3.org/2000/10/XMLSchema	Schema namespace as defined by XSD.
tns	(URL to WSDL file)	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.

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:

```
<definitions .... >
    <types>
        <xsd:schema .... />(0 or more)
    </types>
</definitions>
```

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>

     <element name="AddressBean" type="impl:AddressBean"/>
   </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.

```
...
 <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>
...
```

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"> (0 or more)
                <-- 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:

```
<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:

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

```
...
 <wsdl:service name="AddressServiceService">
      <wsdl:port binding="intf:AddressSoapBinding" name="Address">
         <wsdlsoap:address
            location="http://localhost:8080/axis/services/Address"/>
      </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

REST-based web services

REST defines a set of architectural principles by which you can design web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages. REST does not define the technical building blocks of the Web, such as URIs and HTTP, but rather provides guidelines for the development and use of such technologies in a manner designed to provide the necessary scalability and flexibility for a distributed system of global proportions, such as the World Wide Web.

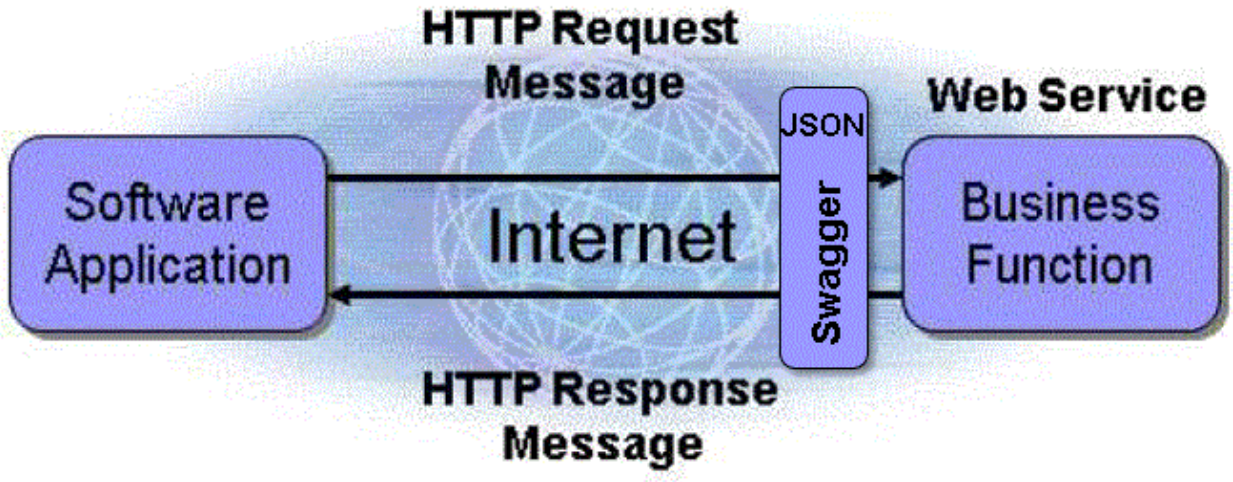


Figure 8: REST-based web services

Core technologies

Several key technologies and standards exist within the web services community:

- HTTP, a communications protocol for the transfer of information on intranets and the World Wide Web. For information on HTTP, see “HTTP protocol” on page 33.
- Uniform Resource Identifier (URI) provide a simple, consistent and persistent means of identifying and locating resources wherever they may exist online. For information about URIs, see “Uniform Resource Identifiers (URIs)” on page 33
- Architectural principles defined by REST. For information on REST, see “REST primer” on page 36.

- XML and/or JSON. For information about XML, see "XML primer" on page 8. For information about JSON, see "JSON primer" on page 34.
- Swagger, a specification for describing RESTful APIs, has become the defacto standard for describing RESTful APIs. For information about Swagger, see "Swagger primer" on page 42.

HTTP protocol

Hypertext Transfer Protocol (HTTP) is a communications protocol for the transfer of information on intranets and the World Wide Web. Its original purpose was to provide a way to publish and retrieve hypertext pages over the Internet.

HTTP development was coordinated by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF), culminating in the publication of a series of Request for Comments (RFCs), most notably RFC 2616 (June 1999), which defines HTTP/1.1, the version of HTTP in common use.

HTTP is a request/response standard between a client and a server. A client is the user and the server is the Web site. The client making an HTTP request using a Web browser, spider, or other user tool is referred to as the user agent. The responding server, which stores or creates resources such as HTML files and images, is called the origin server. In between the user agent and the origin server may be several intermediaries, such as proxies, gateways, and tunnels. HTTP is not constrained to using TCP/IP and its supporting layers, although TCP/IP is the most popular transport mechanism on the Internet. Indeed, HTTP can be implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport. Any protocol that provides such guarantees can be used.

Typically, an HTTP client initiates a request. It establishes a Transmission Control Protocol (TCP) connection to a particular port on a host (port 80 by default). An HTTP server listening on that port waits for the client to send a request message. Upon receiving the request, the server sends back a status line, such as HTTP/1.1 200 OK, and a message of its own, the body of which is perhaps the requested file, an error message, or some other information.

Resources to be accessed by HTTP are identified using Uniform Resource Identifiers (URIs) (or, more specifically, Uniform Resource Locators (URLs)) using the http or https URI schemes.

For more information about the HTTP standard, go to the following URL:

```
http://www.ietf.org/rfc/rfc2616.txt
```

Uniform Resource Identifiers (URIs)

Universal Resource Identifiers (URIs) are, without question, one of the single most important characteristics of web-based applications. URIs provide a simple, consistent and persistent means of identifying and locating resources wherever they may exist online.

An example of an URI is as follows:

```
http://www.ibm.com/systems/power/software/i/iws/index.html
```

According to the URI standard, the example is a URI and has several component parts:

- A scheme name (`http`)
- A domain name (`www.ibm.com`)
- A path (`/systems/power/software/i/iws/index.html`)

For more information about the URI standard, go to the following URL:

```
http://www.ietf.org/rfc/rfc3986.txt
```

JSON primer

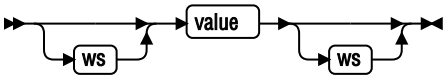
JavaScript Object Notation (JSON) is an open standard format for data interchange. Although originally used in the JavaScript scripting language, JSON is now language-independent, with many parsers available in many languages.

Compared to XML, JSON has many advantages. Most predominantly, JSON is more suited to data interchange. XML is an extremely verbose language: every element in the tree has a name, and the element must be enclosed in a matching pair of tags. Alternatively, JSON expresses trees in a nested array format similar to JavaScript. This enables the same data to be transferred in a far smaller data package with JSON than with XML. This lightweight data package lends itself to better performance when parsing.

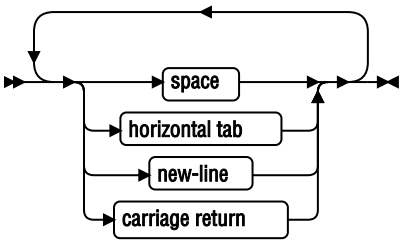
JSON can be seen as both human and machine-readable. JSON is an easy language for humans to read, and for machines to parse.

According to the standard, the JSON syntax is made up of a sequence of tokens. The tokens consist of six structural characters, strings, numbers, and three literal names. The tokens are logically organized into data, objects and arrays. Figure 1 on page 34 shows the syntax diagram for JSON text.

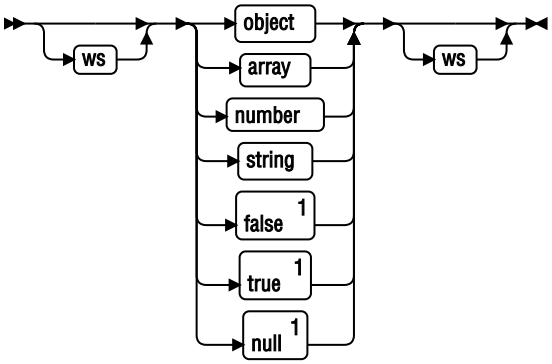
JSON text



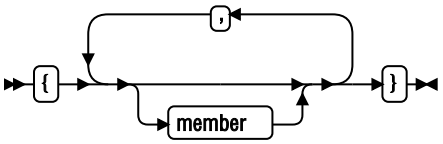
ws



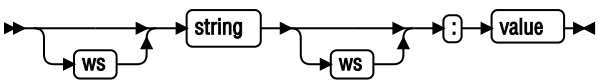
value



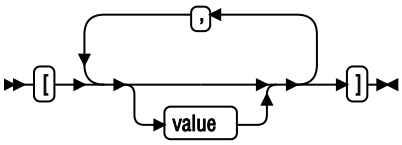
object



member



array



Notes:

[1] The actual literal name: `false`, `true`, or `null`. These values must be lowercase.

Figure 9: JSON text

The following sections provides basic information on JSON. More information about JSON may be found at:

```
http://www.rfc-editor.org/rfc/rfc7159.txt
```

JSON objects

The primary concept in JSON is the object, which is an unordered collection of name/value pairs, where the value can be any JSON value. JSON objects can be nested, but are not commonly deeply nested.

JSON objects begin with a left brace ({) and ends with a right brace (}). Name/value pairs in the object are separated by a comma (,). The name and value for a pair is separated by colon (:). The name is a string (see "JSON strings" on page 36 for more details).

The value may be a JSON object, a JSON array (see "JSON arrays" on page 35 for more details), or one of the four atomic types shown in Table 4 on page 35:

Table 4: JSON data types

Data type	**Example**
string	`"someStringValue"`

number	`3`
	`6.2`
	`-122.026020`
	`9.3e5`

boolean	`true` `false`
the special "null" value	`null`

The following example shows a simple JSON object:

```
{
 "isbn": "123-456-222",
 "title": "The Ultimate Database Study Guide",
 "abstract": "What you always wanted to know about databases",
 "price": 28.00
}
```

JSON arrays

A JSON array is an ordered collections of values. Arrays begin with a left bracket ([) and ends with a right bracket (]). Values in the array are separated by a comma (,).

The following is a simple example of a JSON object that contains arrays:

```
{
 "category": ["Non-Fiction", "Technology"],
 "ratings": [10, 5, 32, 78, 112]
}
```

JSON strings

Strings begins and ends with a quotation mark ("). Within the quotation marks any character may be used except for characters that must be escaped: quotation mark, backslash (\), and the control characters. Any character may be escaped. In addition, characters between Unicode hexadecimal values 0000 through FFFF may be represented by a six character sequence: backslash, followed by lowercase letter u, followed by four hexadecimal digits that encode the character's code point.

The following shows examples of JSON strings:

```
 "category"

 "15\u00f8C"
```

JSON numbers

JSON numbers are represented in base 10 using decimal digits. It contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part. Leading zeros are not allowed. A fraction part is a decimal point followed by one or more digits. An exponent part begins with the letter E in upper or lower case, which may be followed by a plus or minus sign. The E and optional sign are followed by one or more digits.

The following shows examples of JSON numbers:

```
 123

 -122.026020

 9.3e5
```

REST primer

REST was first introduced in 2000 by Roy Fielding at the University of California, Irvine, in his academic dissertation, "*Architectural Styles and the Design of Network-based Software Architectures*[3]," which analyzes a set of software architecture principles that use the Web as a platform for distributed computing.

The dissertation suggests that in its purest form, a concrete implementation of a REST web service follows four basic design principles:

- Expose directory structure-like URIs.
- Use HTTP methods explicitly.
- Be stateless.
- Transfer XML, JavaScript Object Notation (JSON), or both.

The following sections expand on these four principles. For more information about the REST, read Chapter 5 of Roy Fielding's dissertation, located at the following URL:

```
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
```

[3] The dissertation can be found at http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Expose directory structure-like URIs

The notion of using URIs to identify resources is central to REST style web services; by virtue of having a URI, resources are part of the Web. From the standpoint of client applications addressing resources, the URIs determine how intuitive the REST web service is going to be and whether the service is going to be used in ways that the designers can anticipate.

REST web service URIs should be intuitive to the point where they are easy to guess. Think of a URI as a kind of self-documenting interface that requires little, if any, explanation or reference for a developer to understand what it points to and to derive related resources. To this end, the structure of a URI should be straightforward, predictable, and easily understood.

One way to achieve this level of usability is to define directory structure-like URIs. This type of URI is hierarchical, rooted at a single path, and branching from it are subpaths that expose the service's main areas. According to this definition, a URI is not merely a slash-delimited string, but rather a tree with subordinate and superordinate branches connected at nodes. For example, in a discussion threading service that gathers topics ranging from RPG to paper, you might define a structured set of URIs like this:

```
http://www.myservice.org/discussion/topics/{topic}
```

The root, `/discussion`, has a `/topics` node beneath it. Underneath that there are a series of topic names, such as gossip, technology, and so on, each of which points to a discussion thread. Within this structure, it's easy to pull up discussion threads just by typing something after `/topics/`.

In some cases, the path to a resource lends itself especially well to a directory-like structure. Take resources organized by date, for instance, which are a very good match for using a hierarchical syntax. This example is intuitive because it is based on rules:

```
http://www.myservice.org/discussion/2008/12/10/{topic}
```

The first path fragment is a four-digit year, the second path fragment is a two-digit day, and the third fragment is a two-digit month. It may seem a little silly to explain it that way, but this is the level of simplicity we're after. Humans and machines can easily generate structured URIs like this because they are based on rules. Filling in the path parts in the slots of a syntax makes them good because there is a definite pattern from which to compose them:

```
http://www.myservice.org/discussion/{year}/{day}/{month}/{topic}
```

Some additional guidelines to make note of while thinking about URI structure for a RESTful web service are:

- Hide the server-side scripting technology file extensions (.jsp, .php, .asp), if any, so you can port to something else without changing the URIs.
- Keep everything lowercase.
- Substitute spaces with hyphens or underscores (one or the other).
- Avoid query strings as much as you can.
- Learn from popular APIs (Google, Facebook, Twitter, and so on.)

URIs should also be static so that when the resource changes or the implementation of the service changes, the link stays the same. This allows bookmarking. It's also important that the relationship between resources that's encoded in the URIs remains independent of the way the relationships are represented where they are stored.

Designing the URIs for a REST style web service requires special care, as they may be referenced by large numbers of applications, documents, or bookmarks for many years and thus have to be designed so that they are stable.

Use HTTP methods explicitly

One of the key characteristics of a RESTful web service is the explicit use of HTTP methods in a way that follows the protocol as defined by RFC 2616. HTTP GET, for instance, is defined as a data-producing method that's intended to be used by a client application to retrieve a resource, to fetch data from a web server, or to execute a query with the expectation that the web server will look for and respond with a set of matching resources.

REST asks developers to use HTTP methods explicitly and in a way that's consistent with the protocol definition. This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods. According to this mapping:

- To create a resource on the server, use POST.
- To retrieve a resource, use GET.
- To change the state of a resource or to update it, use PUT.
- To remove or delete a resource, use DELETE.

An unfortunate design flaw inherent in many web APIs is in the use of HTTP methods for unintended purposes. The request URI in an HTTP GET request, for example, usually identifies one specific resource. Or the query string in a request URI includes a set of parameters that defines the search criteria used by the server to find a set of matching resources. At least this is how the HTTP/1.1 RFC describes GET. But there are many cases of unattractive web APIs that use HTTP GET to trigger something transactional on the server - for instance, to add records to a database. In these cases the GET request URI is not used properly or at least not used RESTfully. If the web API uses GET to invoke remote procedures, it looks like this:

```
GET /adduser?name=Robert HTTP/1.1
```

It's not a very attractive design because the web method above supports a state-changing operation over HTTP GET. Put another way, the HTTP GET request above has side effects. If successfully processed, the result of the request is to add a new user - in this example, Robert -to the underlying data store. The problem here is mainly semantic. web servers are designed to respond to HTTP GET requests by retrieving resources that match the path (or the query criteria) in the request URI and return these or a representation in a response, not to add a record to a database. From the standpoint of the intended use of the protocol method then, and from the standpoint of HTTP/1.1-compliant web servers, using GET in this way is inconsistent.

Beyond the semantics, the other problem with GET is that to trigger the deletion, modification, or addition of a record in a database, or to change server-side state in some way, it invites web caching tools (crawlers) and search engines to make server-side changes unintentionally simply by crawling a link. A simple way to overcome this common problem is to move the parameter names and values on the request URI into the HTTP request payload (e.g. XML). The resulting tags, an XML representation of the entity to create, may be sent in the body of an HTTP POST whose request URI is the intended parent of the entity as follows:

```
POST /users HTTP/1.1
Host: myserver
Content-Type: application/xml
?xml version="1.0"?>
<user>
  <name>Robert</name>
</user>
```

The method above is exemplary of a RESTful request: proper use of HTTP POST and inclusion of the payload in the body of the request. On the receiving end, the request may be processed by adding the resource contained in the body as a subordinate of the resource identified in the request URI; in this case the new resource should be added as a child of `/users`. This containment relationship between the new entity and its parent, as specified in the POST request, is analogous to the way a file is subordinate to its parent directory. The client sets up the relationship between the entity and its parent and defines the new entity's URI in the POST request.

A client application may then get a representation of the resource using the new URI, noting that at least logically the resource is located under `/users` as follows:

```
GET /users/Robert HTTP/1.1
Host: myserver
Accept: application/xml
```

Using GET in this way is explicit because GET is for data retrieval only. GET is an operation that should be free of side effects, a property also known as idempotence.

A similar refactoring of a web method also needs to be applied in cases where an update operation is supported over HTTP GET, as shown below.

```
GET /updateuser?name=Robert&newname=Bob HTTP/1.1
```

This changes the name attribute (or property) of the resource. While the query string can be used for such an operation, and Listing 4 is a simple one, this query-string-as-method-signature pattern tends to break down when used for more complex operations. Because your goal is to make explicit use of HTTP methods, a more RESTful approach is to send an HTTP PUT request to update the resource, instead of HTTP GET, for the same reasons stated previously:

```
PUT /users/Robert HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user>
  <name>Bob</name>
</user>
```

Using PUT to replace the original resource provides a much cleaner interface that's consistent with REST's principles and with the definition of HTTP methods. The PUT request is explicit in the sense that it points at the resource to be updated by identifying it in the request URI and in the sense that it transfers a new representation of the resource from client to server in the body of a PUT request instead of transferring the resource attributes as a loose set of parameter names and values on the request URI. The PUT request in the example also has the effect of renaming the resource from Robert to Bob, and in doing so changes its URI to `/users/Bob`. In a REST web service, subsequent requests for the resource using the old URI would generate a standard 404 Not Found error.

As a general design principle, it helps to follow REST guidelines for using HTTP methods explicitly by using nouns in URIs instead of verbs. In a RESTful web service, the verbs - POST, GET, PUT, and DELETE - are already defined by the protocol. And ideally, to keep the interface generalized and to allow clients to be explicit about the operations they invoke, the web service should not define more verbs or remote procedures, such as `/adduser` or `/updateuser`. This general design principle also applies to the body of an HTTP request, which is intended to be used to transfer resource state, not to carry the name of a remote method or remote procedure to be invoked.

Stateless

REST web services need to scale to meet increasingly high performance demands. Clusters of servers with load-balancing and failover capabilities, proxies, and gateways are typically arranged in a way that forms a service topology, which allows requests to be forwarded from one server to the other as needed to decrease the overall response time of a web service call. Using intermediary servers to improve scale requires REST web service clients to send complete, independent requests; that is, to send requests that include all data needed to be fulfilled so that the components in the intermediary servers may forward, route, and load-balance without any state being held locally in between requests.

A complete, independent request doesn't require the server, while processing the request, to retrieve any kind of application context or state. A REST web service application (or client) includes within the HTTP headers and body of a request all of the parameters, context, and data needed by the server-side component to generate a response. Statelessness in this sense improves web service performance and simplifies the design and implementation of server-side components because the absence of state on the server removes the need to synchronize session data with an external application.

Figure 10 on page 40 illustrates a stateful service from which an application may request the next page in a multipage result set, assuming that the service keeps track of where the application leaves off while navigating the set. In this stateful design, the service increments and stores a `previousPage` variable somewhere to be able to respond to requests for next.

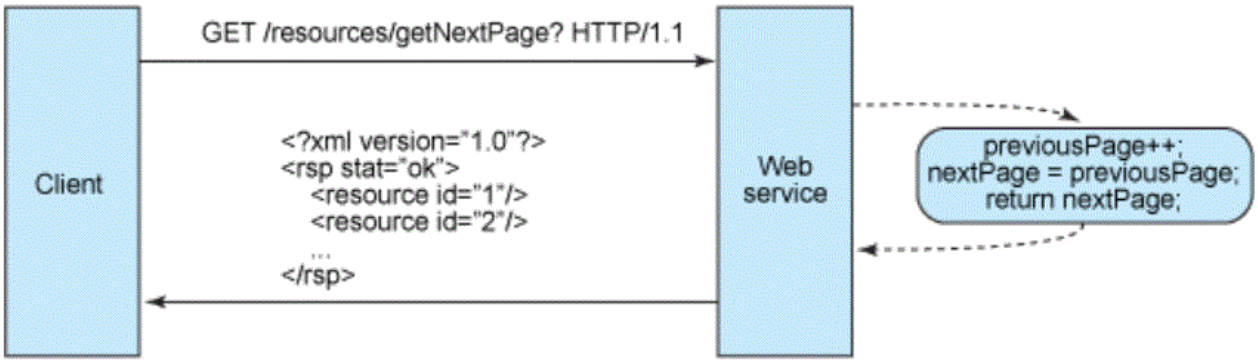


Figure 10: Stateful design

Stateful services like this get complicated. Stateful services may require a lot of up-front consideration to efficiently store and enable the synchronization of session data. Session synchronization adds overhead, which may impact server performance.

Stateless server-side components, on the other hand, are less complicated to design, write, and distribute across load-balanced servers. A stateless service not only performs better, it shifts most of the responsibility of maintaining state to the client application. In a RESTful web service, the server is responsible for generating responses and for providing an interface that enables the client to maintain application state on its own. For example, in the request for a multipage result set, the client should include the actual page number to retrieve instead of simply asking for next (see Figure 11 on page 40).

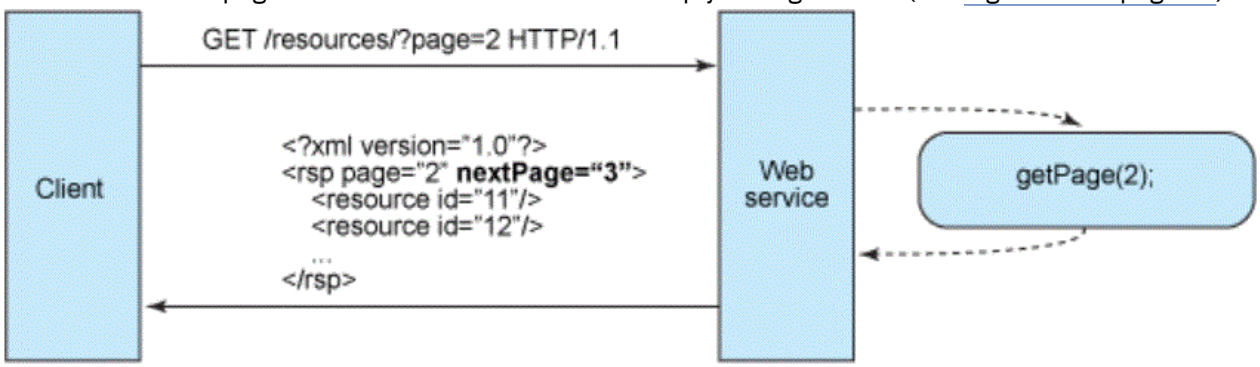


Figure 11: Stateless design

A stateless web service generates a response that links to the next page number in the set and lets the client do what it needs to in order to keep this value around. This aspect of RESTful web service design can be broken down into two sets of responsibilities as a high-level separation that clarifies just how a stateless service can be maintained:

- **Server**
  - Generates responses that include links to other resources to allow applications to navigate between related resources. This type of response embeds links. Similarly, if the request is for a parent or container resource, then a typical RESTful response might also include links to the parent's children or subordinate resources so that these remain connected.
  - Generates responses that indicate whether they are cacheable or not to improve performance by reducing the number of requests for duplicate resources and by eliminating some requests entirely. The server does this by including a `Cache-Control` and `Last-Modified` (a date value) HTTP response header.
- **Client application**

- Uses the `Cache-Control` response header to determine whether to cache the resource (make a local copy of it) or not. The client also reads the `Last-Modified` response header and sends back the date value in an `If-Modified-Since` header to ask the server if the resource has changed. This is called Conditional GET, and the two headers go hand in hand in that the server's response is a standard 304 code (Not Modified) and omits the actual resource requested if it has not changed since that time. A 304 HTTP response code means the client can safely use a cached, local copy of the resource representation as the most up-to-date, in effect bypassing subsequent GET requests until the resource changes.
- Sends complete requests that can be serviced independently of other requests. This requires the client to make full use of HTTP headers as specified by the web service interface and to send complete representations of resources in the request body. The client sends requests that make very few assumptions about prior requests, the existence of a session on the server, the server's ability to add context to a request, or about application state that is kept in between requests.

This collaboration between client application and service is essential to being stateless in a RESTful web service. It improves performance by saving bandwidth and minimizing server-side application state.

REST style web service payloads

A resource representation typically reflects the current state of a resource, and its attributes, at the time a client application requests it. Resource representations in this sense are mere snapshots in time. This could be a thing as simple as a representation of a record in a database that consists of a mapping between column names and XML tags, where the element values in the XML contain the row values. Or, if the system has a data model, then according to this definition a resource representation is a snapshot of the attributes of one of the things in your system's data model. These are the things you want your REST Web service to serve up.

The last set of constraints that goes into a RESTful Web service design has to do with the format of the data that the application and service exchange in the request/response payload or in the HTTP body. This is where it really pays to keep things simple, human-readable, and connected.

The objects in your data model are usually related in some way, and the relationships between data model objects (resources) should be reflected in the way they are represented for transfer to a client application. In the discussion threading service, an example of connected resource representations might include a root discussion topic and its attributes, and embed links to the responses given to that topic.

```
<?xml version="1.0"?>
<discussion date="{date}" topic="{topic}">
  <comment>{comment}</comment>
  <replies>
    <reply from="joe@mail.com" href="/discussion/topics/{topic}/joe"/>
    <reply from="bob@mail.com" href="/discussion/topics/{topic}/bob"/>
  </replies>
</discussion>
```

And last, to give client applications the ability to request a specific content type that's best suited for them, construct your service so that it makes use of the built-in HTTP Accept header, where the value of the header is a MIME type. Some common MIME types used by RESTful services are shown in Table 5 on page 41.

Table 5: Common MIME types used by RESTful services

MIME-type	**Content-type**
JSON	application/json
XML	application/xml

This allows the service to be used by a variety of clients written in different languages running on different platforms and devices. Using MIME types and the HTTP Accept header is a mechanism known as content negotiation, which lets clients choose which data format is right for them and minimizes data coupling between the service and the applications that use it.

Swagger primer

Swagger is an open specification for defining REST APIs. A Swagger document is the REST API equivalent of a WSDL document for a SOAP-based web service. The Swagger document specifies the list of resources that are available in the REST API and the operations that can be called on those resources. The Swagger document also specifies the list of parameters to an operation, including the name and type of the parameters, whether the parameters are required or optional, and information about acceptable values for those parameters. Additionally, the Swagger document can include JSON Schema that describes the structure of the request body that is sent to an operation in a REST API, and the JSON schema describes the structure of any response bodies that are returned from an operation.

The integrated web services server supports version 2.0 of the Swagger specification. Information on Swagger and the version 2.0 of the Swagger specification may be found at the following URLs:

```
http://swagger.io/

https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md
```

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 and commands.

Chapter 3. Web services client overview

The Web Services Client for ILE is based on Apache Extensible Interaction System (Axis) version 1.5. Axis is basically a SOAP engine that represents a framework for constructing and consuming SOAP messages.

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.
- **User-defined payloads**: IBM has made enhancements that enables users to send user-defined messages so that payloads other than SOAP messages can be sent and received.

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 stub-generation portion of 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)

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"`.

Client architecture

Figure 12 on page 46 illustrates the underlying architecture of Apache Axis. One of Axis' key features is the incredible pluggabilty 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.

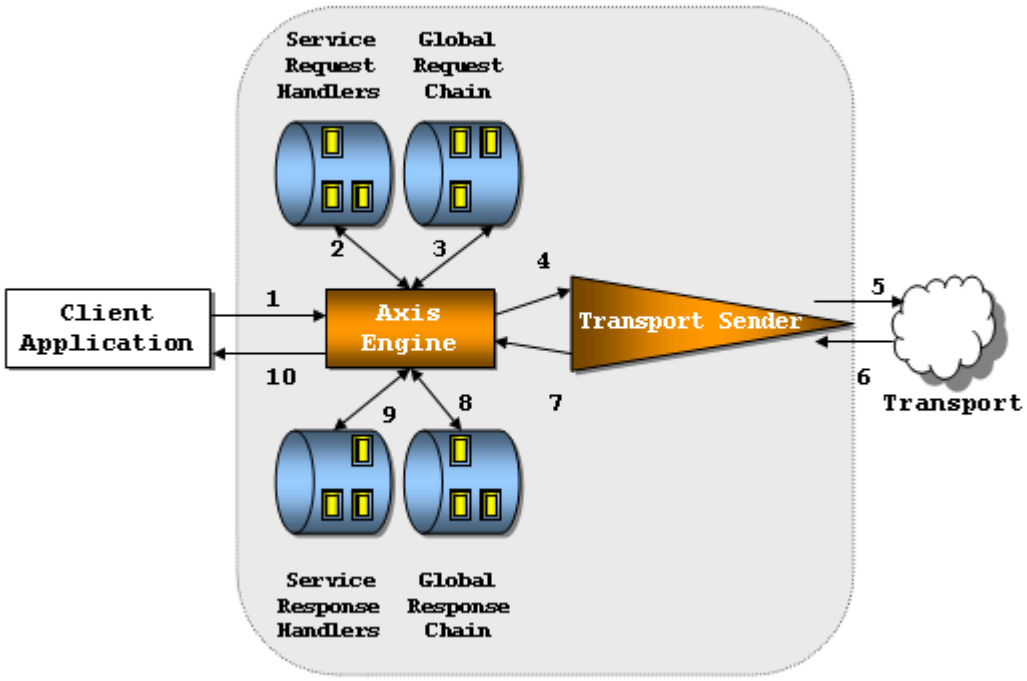


Figure 12: Axis client-side architecture

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 on page 46 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 Web Services Client for ILE gives client applications the 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 `wsdl2ws.sh` tool (more on this later) to generate stubs (proxy) that you can use to communicate with the web service server. The stub-based invocation only supports Web services using the SOAP protocol.
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. However, it is much easier to generate stubs to perform Web services requests simply for the fact that any changes in the WSDL would require a user to manually make changes to the code, which is more error prone than having the `wsdl2ws.sh` tool generate the stubs for you.

Stub-based invocation

The Web Services Client for ILE package provides a Java program that is invoked by the `wsdl2ws.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 on page 47.

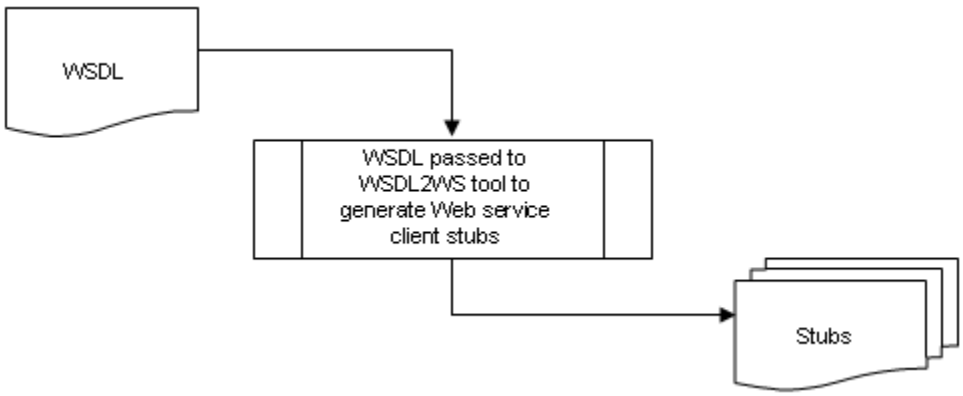


Figure 13: Process flow between WSDL source file and stubs

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 on page 48.

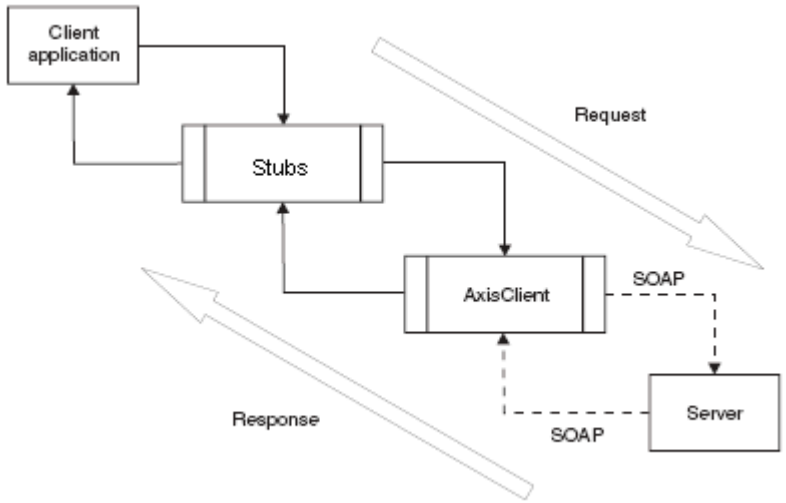


Figure 14: Process flow between client and server applications using stubs generated by WSDL2Ws

API-based invocation

The Web Services Client for ILE package provides allows users to use the same APIs used by the generated stubs to perform SOAP requests. In addition, there are APIs, called the transport APIs, that allows users to control the data that is sent. The payload is completed generated by the user, and the response is returned to the client for processing.

The C prototypes for the transport API functions are in the file `/QIBM/ProdData/OS/WebServices/V1/client/include/axis/ITransport.h`. The RPG prototypes for the functions are in the file `/QIBM/ProdData/OS/WebServices/V1/client/include/Axis.rpgleinc`. The typical flow of events when using the transport APIs is as follows:

1. Use the `axiscTransportCreate()` API to create a transport object. The URL to web service is specified on the call to the function.
2. Set any transport properties (e.g. connect timeout, HTTP method, HTTP headers, whether payload needs to be converted to UTF-8, etc.) using the `axiscTransportSetProperty()` function.
3. Send data (if any) using the `axiscTransportSend()` function. Data is buffered until the axiscTransportFlush() is called. The data is automatically converted to UTF-8 unless the AXISC_PROPERTY_CONVERT_PAYLOAD transport property is set to "false", in which case the data is sent as-is.
4. Send the request to the client by invoking the `axiscTransportFlush()` function.
5. Receive the response to the request by calling the `axiscTransportReceive()` function. This API must be called even if no data is returned in order to consume the HTTP response to the request, which includes the HTTP response headers and status code. The data is automatically converted from UTF-8 to the job CCSID unless the AXISC_PROPERTY_CONVERT_PAYLOAD transport property is set to "false", in which case the data is returned as-is.
6. If no errors were detected on the call to `axiscTransportReceive()`, retrieve the HTTP status code to determine what to do with received data using the axiscTransportGetProperty() API with property to be retrieved set to AXISC_PROPERTY_HTTP_STATUS_CODE.
7. Destroy the transport object by calling the `axiscTransportDestroy()` function.

For more information on the transport APIs, see “Transport C APIs” on page 169.

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 on page 49 is an amended version of Figure 14 on page 48 and illustrates the placement of handlers in the request and response flows.

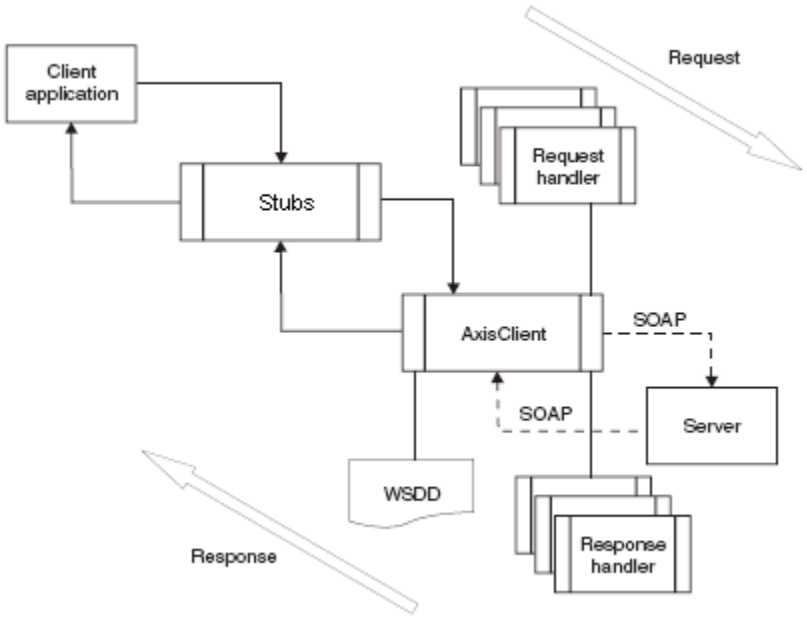


Figure 15: Process flow between client and server applications using generated stubs, and request and response handlers

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 on page 50:

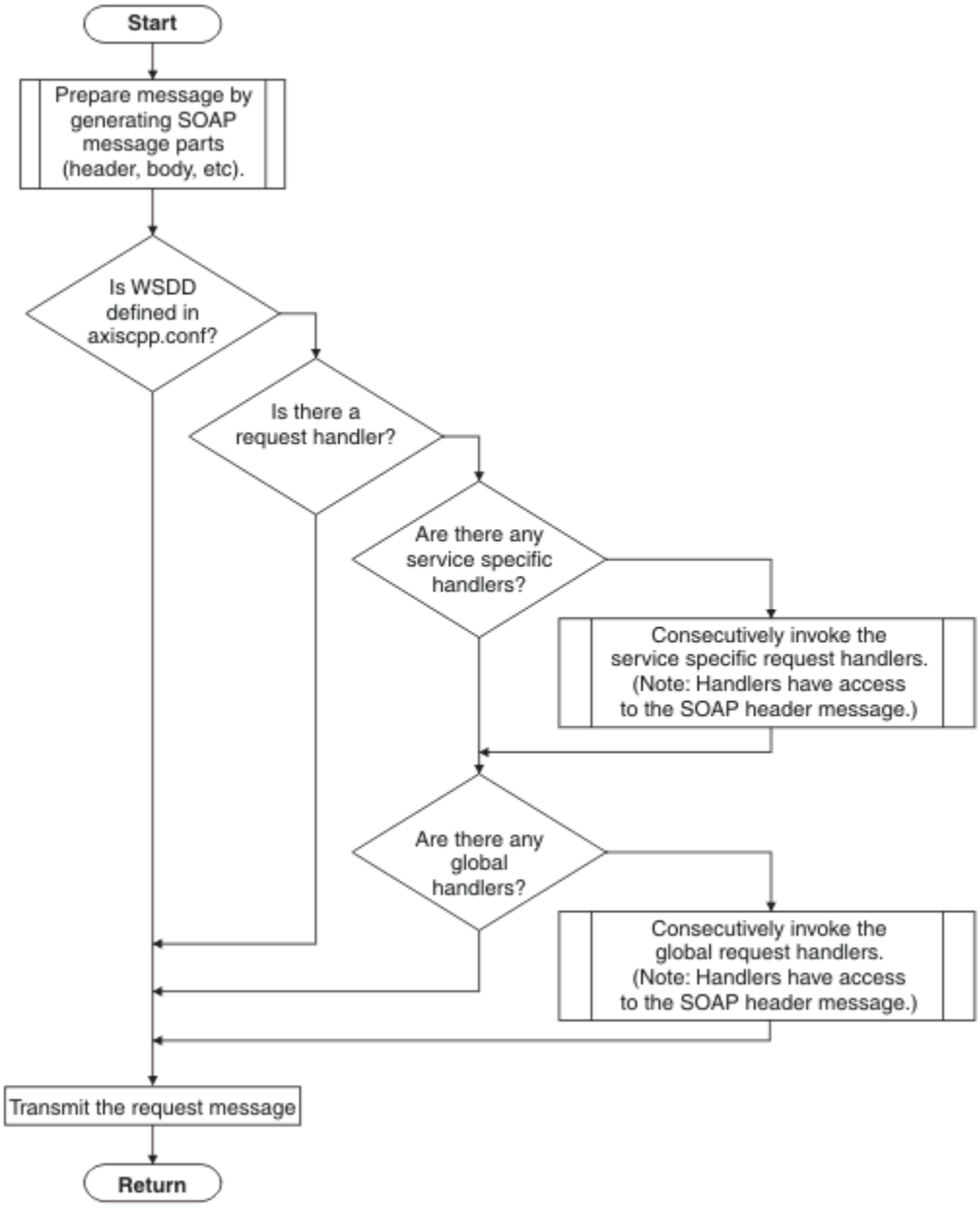


Figure 16: Process flow for pre-pivot handlers

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 51:

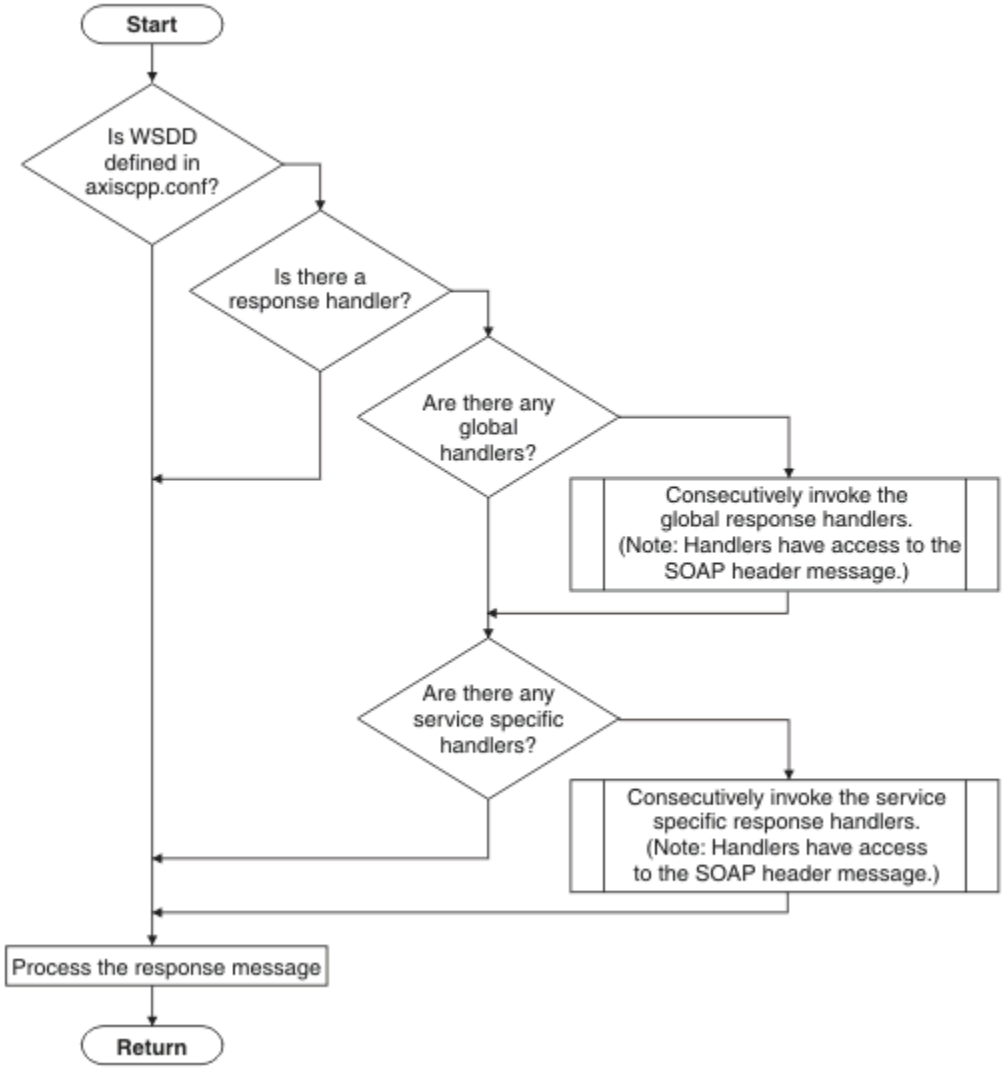


Figure 17: Process flow for post-pivot handlers

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 6 on page 52 shows the data types that are supported by Web Services Client for ILE:

Table 6: Supported schema types	
Category	**Schema type(s)**
Numeric	• byte • decimal • double • float • int • integer • long • negativeInteger • nonPositiveInteger • nonNegativeInteger • positiveInteger • unsignedByte • unsignedInt • unsignedLong • unsignedShort • short
Date/Time/Duration	• date • dateTime • duration • gDay • gMonth • gMonthDay • gYear • gYearMonth • time

Table 6: Supported schema types (continued)	
Category	**Schema type(s)**
String	• anyURI • ENTITY • ENTITIES • ID • IDREFS • language • Name • NCName • NMTOKEN • NMTOKENS • normalizedString • notation • QName • string • token
Various	• base64Binary • boolean • hexBinary

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 4. 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. 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:

- System Openness Includes - base option 13 of operating system
- 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 SE 7 32 bit
- 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.

Note: After installing the various license product options, you should load the latest HTTP Group PTF since all fixes and enhancements are packaged as part of the HTTP Group PTF. It would also be wise to load the latest Java group PTF. The various group PTFs for an IBM i release may be found at the IBM Support Portal.

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:

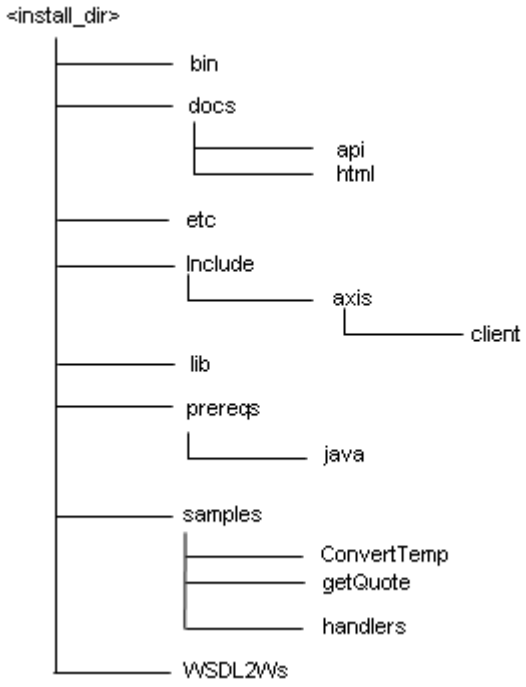


Figure 18: Install directory structure

The following table gives an overview of the contents of each directory:

Table 7: Contents of installed directories	
Installed directory	**Contents**
`<install_dir>/bin`	Contains the `wsdl2ws.sh` and `wsdl2rpg.sh` QShell scripts that are used to invoke the Java tool in order to generate Web service stubs.
`<install_dir>/docs`	Contains the Web Services Client for ILE API documentation in HTML format.
`<install_dir>/etc`	This directory level contains an example `axiscpp.conf` configuration file.
`<install_dir>/include`	Web Services Client for ILE header files, which are required for building web service specific generated stubs.
`<install_dir>/lib`	Contains all of the built libraries needed for building web service specific generated stubs.
`<install_dir>/prereqs/java`	The prerequisite Java jar files that are required for the `wsdl2ws.sh` and `wsdl2rpg.sh` QShell scripts.
`<install_dir>/samples`	The location of the samples that accompany Web Services Client for ILE.
`<install_dir>/WSDL2Ws`	The Java tool that is used to generate Web service stubs.

Chapter 5. 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')
```

The following sections gives more details of the supported commands.

wsdl2ws.sh command

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.

Synopsis

wsdl2ws.sh *[arguments] WSDL-URI*

Arguments

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*<c|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 65000[4]. Maximum value for IBM i 6.1 and subsequent releases is 8000000[4]. Default is 128. This option is valid only if **-lrpg** is specified.

-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 65000[4]. Maximum value for IBM i 6.1 and subsequent releases is 16000000[4]. This option is valid only if **-lrpg** is specified.

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

-ma*<max-array-size>*
Maximum size to use when defining RPG arrays. Minimum value is 1. Maximum value for IBM i 5.4 is 32000[4]. Maximum value for IBM i 6.1 and subsequent releases is 8000000[4]. 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.

-L
Generate spooled file compiler listing. Default is to not generate a compiler listing.

-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 `wsdl2ws.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:

   ```
   wsdl2ws.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:

   ```
   wsdl2ws.sh -lc -o/Stub/c -s/qsys.lib/ws.lib/wsc.srvpgm
               /Stub/ConvertTemp.wsdl
   ```

wsdl2rpg.sh command

The **wsdl2rpg.sh** command has the same options as **wsdl2ws.sh** with the exception of the **-l** (target language) option. Specifying **wsdl2ws.sh** with **-lrpg** is the same as using the **wsdl2rpg.sh** command.

Chapter 6. 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 8: List of axiscpp.conf configuration file properties

Property	Description
`ClientWSDDFilePath`	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` See "The Web services deployment descriptor (WSDD) file" on page 63 for further details on the WSDD file.

Table 8: List of axiscpp.conf configuration file properties (continued)

Property	Description

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 (information should all be on one line):
	`SecureInfo:keyRingFile,keyRingPasswordOrStash,keyRingLabel, v2CipherSpec,v3CipherSpec,tlsCipherSpec, tlsV11CipherSpec,tlsV12CipherSpec`
	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.
	tlsV11CipherSpec
	Whether to enable or disable TLS Version 1.1 ciphers. A value of NONE will disable the ciphers; any other value will enable the ciphers. By default, the TLS Version 1.1 ciphers are enabled.
	tlsV12CipherSpec
	Whether to enable or disable TLS Version 1.2 ciphers. A value of NONE will disable the ciphers; any other value will enable the ciphers. By default, the TLS Version 1.2 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:

   ```
   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 version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:C="http://xml.apache.org/axis/wsdd/providers/c"> 1

<!--Service Handler Definitions-->
<service name="Handler" provider="CPP:DOCUMENT" description="Handler"> 2
  <requestFlow> 3
     <handler name="myClientHandlerReq" type="/qsys.lib/samples.lib/handler.srvpgm"/> 4
  </requestFlow>

  <responseFlow> 5
     <handler name="myClientHandlerRes" type="/qsys.lib/samples.lib/handler.srvpgm"/> 6
  </responseFlow>
</service>

<!--Global Handler Definitions-->
<globalConfiguration name="GlobalHandler" provider="CPP:DOCUMENT" description="Global
Handler"> 7
  <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:

Line Number	Description
1	The deployment tag is the root element.

Line Number	Description
2	The `service` tag defines the Axis service for which the handlers specified within the `service` tag are invoked whenever the web service is used. 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`. 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.
3	The `requestFlow` 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.
4	The `handler` tag within a `requestFlow` 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.
5	The `responseFlow` 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.
6	The `handler` tag within a `responseFlow` 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.
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.

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 7. 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++ identifer 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 9 on page 67 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 9: XML to C++ type mapping

Schema Type	Axis Type	Actual C++ Type
	Numeric	
xsd:byte	`xsd__byte`	`signed char`
xsd:decimal	`xsd__decimal`	`double`
xsd:double	`xsd__double`	`double`
xsd:float	`xsd__float`	`float`
xsd:int	`xsd__int`	`int`
xsd:integer	`xsd__integer`	`long long`
xsd:long	`xsd__long`	`long long`
xsd:negativeInteger	`xsd__negativeInteger`	`long long`
xsd:nonPositiveInteger	`xsd__nonPositiveInteger`	`long long`
xsd:nonNegativeInteger	`xsd__nonNegativeInteger`	`unsigned long long`
xsd:positiveInteger	`xsd__positiveInteger`	`unsigned long long`
xsd:unsignedByte	`xsd__unsignedByte`	`unsigned char`
xsd:unsignedInt	`xsd__unsignedInt`	`unsigned int`
xsd:unsignedLong	`xsd__unsignedLong`	`unsigned long long`
xsd:unsignedShort	`xsd__unsignedShort`	`unsigned short`

Table 9: XML to C++ type mapping (continued)		
Schema Type	**Axis Type**	**Actual C++ Type**
xsd:short	xsd__short	short
Date/Time/Duration		
xsd:date	xsd__date	struct tm
xsd:dateTime	xsd__dateTime	struct tm
xsd:duration	xsd__duration	long
xsd:gDay	xsd__gDay	struct tm
xsd:gMonth	xsd__gMonth	struct tm
xsd:gMonthDay	xsd__gMonthDay	struct tm
xsd:gYear	xsd__gYear	struct tm
xsd:gYearMonth	xsd__gYearMonth	struct tm
xsd:time	xsd__time	struct tm
String		
xsd:anyURI	xsd__anyURI	char *
xsd:anyType	xsd__anyType	char *
xsd:ENTITY	xsd__ENTITY	char *
xsd:ENTITIES	xsd__ENTITIES	char *
xsd:ID	xsd__ID	char *
xsd:IDREFS	xsd__IDREFS	char *
xsd:language	xsd__language	char *
xsd:Name	xsd__Name	char *
xsd:NCName	xsd__NCName	char *
xsd:NMTOKEN	xsd__NMTOKEN	char *
xsd:NMTOKENS	xsd__NMTOKENS	char *
xsd:normalizedString	xsd__normalizedString	char *
xsd:notation	xsd__notation	char *
xsd:QName	xsd__QName	char *
xsd:string	xsd__string	char *
xsd:token	xsd__token	char *
Other		
xsd:base64Binary	xsd__base64Binary	Implemented as C++ class
xsd:boolean	xsd__boolean	enum
xsd:hexBinary	xsd__hexBinary	Implemented as C++ class

The Axis types are defined in the header file <install_dir>/include/axis/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 9 on page 67 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:

```
<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:

```
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 use 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:

```
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 205 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 1.1 document structure" on page 24. Now let us see how various WSDL definitions are mapped. The following table summarizes the WSDL and XML to C++ mappings:

Table 10: WSDL and XML to C++ mapping summary

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

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 67. 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.

```
...
<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:element>
        </ati:sequence>
      </ati:complexType>
    </ati:element>

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

```
...
<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 73 for further explanation of the mapping of wsdl:operation). The above WSDL port type definition maps to the following C++ service interface:

```
class StockQuote : public Stub
{
public:
    StockQuote(const char* pchEndpointUri, AXIS_PROTOCOL_TYPE eProtocol=APTHTTP1_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 74 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:

```
...
<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 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 when you construct a `StockQuote` object using the `StockQuote()` constructor.

```
...
<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. 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 20). 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:

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

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

```
...
<wsdl:message name="getQuoteFault">
  <wsdl:part element="impl:getQuoteFault" name="fault"/>
</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:fault message="impl:getQuoteFault" name="getQuoteFault"/>
  </wsdl:operation>
</wsdl:portType>

...
<wsdl:binding name="StockQuoteSoapBinding" type="impl:StockQuote">
  <wsdlsoap:binding style="document"
       transport="http://schemas.xmlsoap.org/soap/http"/>

  <wsdl:operation name="getQuote">
    <wsdlsoap:operation soapAction=""/>

    <wsdl:input name="getQuoteRequest">
      <wsdlsoap:body use="literal"/>
    </wsdl:input>

    <wsdl:output name="getQuoteResponse">
```

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

    <wsdl:fault name="getQuoteFault">
      <wsdlsoap:fault name="getQuoteFault" use="literal"/>
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>
...
```

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 85.

Chapter 8. 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 11: Files in the samples directory

File name	Description
`GetQuote.wsdl`	`GetQuote` WSDL file.
`myGetQuote.cpp`	Client implementation code written in C++.

Source listings for the client application code can be found at Appendix A, “Code Listings for myGetQuote Client Application,” on page 205.

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/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 `wsdl2ws.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 `wsdl2ws.sh` tool is shown below:

```
StockQuote.cpp  StockQuote.hpp  ws.cl
```

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:

```
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" on page 78.

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 77. However, before we continue, you should note the following points[5]:

- `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:

   ```
   cd '/myGetQuote/CPP'
   ```

[5] If you have not read Chapter 7, "WSDL and XML to C++ mappings," on page 67 then it would be a good time to do so prior to reading this section.

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:

```
COPY OBJ('<install_dir>/samples/getQuote/myGetQuote.cpp') TODIR('/myGetQuote/CPP')
```

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” on page 79.

Note: If you want to use one or more handlers with your application, see Chapter 9, “ Creating client-side handlers,” on page 81.

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 61.

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 9. Creating client-side handlers

This chapter describes how to develop client side handlers[6] 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[7], 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 `axiscpp.conf` files to include the appropriate details, as described in "The Web services deployment descriptor (WSDD) file" on page 63 and "The axiscpp.conf file" on page 61.

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.
3. Create `axiscpp.conf` file and update file so that it points to WSDD file.

[6] An overview of client-side handler concepts can be found at "Client-side handlers" on page 48.

[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.

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 12: Files in the samples/handlers directory

File name	Description
client.wsdd	The WSDD file that defines the client handler.
myClientHandler.cpp	Client handler implementation.
myClientHandler.hpp	Client handler implementation header file.
myClientHandlerFactory.cpp	Client handler factory implementation.

Source listings for the handler code can be found at Appendix B, "Code Listings for Client Handler," on page 213.

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" on page 82.

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 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>
    <handler name="myClientHandlerReq" type="/qsys.lib/HANDLERS.lib/MYCLH.srvpgm"/>
  </requestFlow>

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

</deployment>
```

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.srvpgm` 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 63.

Finally, we need to point to the WSDD file in the `axiscpp.conf` file.

Updating axiscpp.conf file to point to WSDD file

If you are using client side handlers, you must add an additional line to the `axiscpp.conf` configuration file, defining the path to the WSDD file. The following example shows the `axiscpp.conf` 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 `axiscpp.conf` 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 10. 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 74, 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:

```
class DivByZeroStruct : public SoapFaultException
{
  public: xsd__string varString;
  xsd__int varInt;
  xsd__float varFloat;
  DivByZeroStruct();
  ~DivByZeroStruct() throw();
};
```

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:

```
// 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;
}
catch(...)
{
  // Catch any other exception
  cout << "Unspecified Exception: " << 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.

```
<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:

```
public:
STORAGE_CLASS_INFO xsd__int add( …);
```

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

```
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.

```
<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:

```
public:
STORAGE_CLASS_INFO xsd__int * add( …);
```

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

```
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” on page 88
- “Array types as output parameters” on page 89

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 “Array types as input parameters” on page 88, 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:

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

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

// 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.
// Note that 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.

```
// 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)
   - Nillable
   - 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__NMTOKENS, 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 nillable 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:

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

    must be split into two lines of code:

    ```
    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 61 configuration file, or by configuring the settings for secure service using the “Stub::SetSecure()” on page 109 Stub class method.

Using secure connections with a proxy server

The integrated web services client gives you the option to send requests to a proxy server. By default, the connection that is established is unsecure. If you want to establish a secure connection to the proxy server you will need to invoke the “Call::setTransportProperty()” on page 111 Axis C++ API with the ENABLE_SSL_OVER_PROXY option.

The integrated web services client also supports *SSL tunneling*. In SSL tunneling, the client establishes an insecure connection to the proxy server, and then attempts to tunnel through the proxy server to the content server over a secure connection where encrypted data is passed through the proxy server unaltered. The SSL tunneling process is as follows:

1. The client establishes an insecure connection to the proxy server.
2. The client makes a tunneling request. The proxy accepts the connection on its port, receives the request, and connects to the destination server on the port requested by the client. The proxy replies to the client that a connection is established.
3. The proxy relays SSL handshake messages in both directions: From client to destination server, and from destination server to client.
4. After the secure handshake is completed, the proxy sends and receives encrypted data to be decrypted at the client or at the destination server.

In order for SSL tunneling to occur, the proxy server needs to support SSL tunneling requests, and the web service endpoint must be a secure endpoint (i.e. https).

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 13: Behavior of Web services client for ILE with regard to cookie attributes

Attribute	Behavior
expires	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.
path	This attribute is ignored. Cookies are sent on all requests and not just on requests to a URI applicable to the path.
domain	This attribute is ignored. Cookies have affinity to a stub and are domain neutral.
secure	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.

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.

```
// 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://calculatorserver/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.
currentCookieKey=0;
while(currentCookieKey< 3)
{
   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);

// If we continue to use this instance of the calculator stub then the cookies
// will be continue to be sent.
result = calculator.add(1,2);

// If we use a new instance of the calculator then it will fail because we have
// not set the cookies
Calculator newCalculatorInstance("http://calculatorserver/calculatorservice);
// This will fail with an exception because we have not set the authentication
// cookies
result = newCalculatorInstance.add(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:

```
service.deleteTransportProperty(cookiename);
```

  For example:

```
calculator.deleteTransportProperty("loginCookie");
```

- To delete all cookies from a C++ stub instance:

```
service.deleteTransportProperty("Cookie");
```

  Note the capital 'C' in "Cookie".

  For example:

```
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:

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

you must use the following formatting command:

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

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

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

you must use the following formatting command:

```
printf( "%.10g", myDouble);
```

Chapter 11. 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 45.

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 100 Axis C++ class.

Chapter 12. 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, copy the file `api.zip` from `/QIBM/ProdData/OS/WebServices/V1/client/docs/api.zip`, unzip it, and view the following file in a Web browser: `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 14: Axis class methods

Class methods	Description
Axis::initialize()	Initializes the Axis runtime.
Axis::terminate()	Terminates the Axis runtime.
Axis::AxisDelete()	Deletes storage allocated by the Axis engine.
Axis::startTrace()	Starts Axis logging.
Axis::stopTrace()	Stops Axis logging.
Axis::writeTrace()	Writes trace data to Axis log.

Axis::initialize()

```
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.

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

Axis::terminate()

```
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):

```
[13/11/2008 15:55:55:509] 00007860 transport > HTTPTransport::processHTTPHeader():
[13/11/2008 15:55:55:510] 00007860 transport < HTTPTransport::processHTTPHeader():
```

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:

```
[13/11/2008 15:55:55:510] 00007860 transport X HTTPTransport::readHTTPHeader():
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 a example of a transport trace:

```
.
.
.
[13/11/2008 15:55:55:280] 00007860 transport D HTTPChannel::writeBytes():
POST /axis HTTP/1.1
Host: 127.0.0.1:13260
Content-Type: text/xml; charset=UTF-8
SOAPAction: ""
Content-Length: 393


[13/11/2008 15:55:55:280] 00007860 transport < HTTPChannel::writeBytes(): Exit with integer
value of 122
[13/11/2008 15:55:55:281] 00007860 transport > HTTPChannel::writeBytes():
[13/11/2008 15:55:55:282] 00007860 transport D HTTPChannel::writeBytes():
<?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>

.
.
.
[13/11/2008 15:55:55:508] 00007860 transport D HTTPChannel::readBytes():
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

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 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
```

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 102 method.

Axis::stopTrace()

```
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()” on page 102 method.

Axis::writeTrace()

```
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.

```
#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.
        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 `wsdl2ws.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 15: Stub class methods

Class methods	Description
Stub::setTransportProperty()	Sets transport properties (e.g. HTTP headers).
Stub::getTransportProperty()	Gets transport properties (e.g. HTTP headers).
Stub::setTransportTimeout()	Sets the transport timeout.
Stub::createSOAPHeaderBlock()	Creates and adds a SOAP header block to the stub.
Stub::setMaintainSession()	Sets whether to maintain session with service or not.
Stub::setPassword()	Sets the password to be used for basic authentication.
Stub::setUsername()	Sets the user name to be used for basic authentication.
Stub::setProxy()	Sets the proxy server and port for transport.

Table 15: Stub class methods (continued)	
Class methods	**Description**
Stub::setProxyPassword()	Sets the password to be used for proxy authentication.
Stub::setProxyUsername()	Sets the user name to be used for proxy authentication.
Stub::SetSecure()	Sets SSL configuration properties.

Stub::setTransportProperty()

```
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.

```
stub.setTransportProperty("Cookie", "sessiontoken=123345456");
```

Stub::getTransportProperty()

```
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.

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

Stub::setTransportTimeout()

```
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:

```
<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>
</wsse:Security>
```

Here is the example:

```
#include "axis/Axis.hpp"
#include "axis/IHeaderBlock.hpp"
#include "axis/BasicNode.hpp"
#include "StockQuote.hpp"

#include <stdio.h>

int main()
{
.
.
.
  StockQuote *stub = new StockQuote("http://9.10.109.164:8088/StockQuote");


  // generate node wsse:Security element, declaring namespaces for wsse and wsu
  IHeaderBlock *phb = stub->createSOAPHeaderBlock(
    "Security",
    "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd",
    "wsse");

  phb->createNamespaceDecl(
    "wsu",
    "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd");

  phb->createStdAttribute(MUST_UNDERSTAND_TRUE, SOAP_VER_1_1);

  // Generate node wsse:UsernameToken as child node of wsse:Security
  BasicNode *Bnode1=phb->createChild(
    ELEMENT_NODE, "UsernameToken", "wsse", NULL, NULL);

  Bnode1->createAttribute(
    "Id","wsu",NULL,"UsernameToken-12345678");
  phb->addChild(Bnode1);

  // Generate node wsse:Username as child node of wsse:UsernameToken
  // and the associated character node
  BasicNode *Bnode2=phb->createChild(
    ELEMENT_NODE,"UserName","wsse",NULL,NULL);
  Bnode1->AddChild(Bnode2);

  BasicNode *Bnode3=phb->createChild(
    CHARACTER_NODE,NULL,NULL,NULL,"admin");
  Bnode2->addChild(Bnode3);

  // Generate node wsse:Password as child node of wsse:UsernameToken
  // and the associated character node
  Bnode2=phb->createChild(
    ELEMENT_NODE,"Password","wsse",NULL,NULL);
  Bnode2->createAttribute(
    "Type",NULL,NULL,
    "http://docs.oasis-open.org/wss/2004/01/"
    "oasis-200401-wss-username-token-profile-1.0#PasswordText");
  Bnode1->addChild(Bnode2);

  Bnode3=phb->createChild(
    CHARACTER_NODE,NULL,NULL,NULL,"admin");
  Bnode2->addChild(Bnode3);

  // Perform Web service operation
  .
  .
  .
  // Delete the web service.
  delete stub;

  // Exit.
  return 0;
}
```

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.

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

Stub::setPassword()

```
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.

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

Stub::setUsername()

```
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.

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

Stub::setProxy()

```
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.

Example

Following example sets proxy host and port information.

```
stub.setProxy("proxyserver", 40001);
```

Stub::setProxyPassword()

```
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()

```
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()

```
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. This parameter is ignored if the application ID parameter is set to a value.
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 as long as the user has authority to the certificate store file and the password has been used one time one the system. To specify any of the subsequent optional parameters, you must pass a value for this parameter. This parameter is ignored if the application ID parameter is set to a value.
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. This parameter is ignored if the application ID parameter is set to a value.
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.
pTLSv11Cipher	(optional) Pointer to null-terminated character string indicating whether to enable or disable the TLS Version 1.1 ciphers. A value of "NONE" will disable the ciphers; any other value will enable the ciphers. By default, the TLS Version 1.1 ciphers are enabled.
pTLSv12Cipher	(optional) Pointer to null-terminated character string indicating whether to enable or disable the TLS Version 1.2 ciphers. A value of "NONE" will disable the ciphers; any other value will enable the ciphers. By default, the TLS Version 1.2 ciphers are enabled.
pTolerate	(optional) Pointer to null-terminated character string indicating whether to tolerate soft validation errors (expired certificate or certificate not in certificate store). Specify a value of `true` to tolerate soft validation errors, or `false` to not tolerate soft validation errors. The default is `false`.
pAppid	(optional) Pointer to null-terminated character string indicating the application ID to use for the SSL environment.

pFQDN (optional) Pointer to null-terminated character string indicating the fully qualified domain name that will be used as Server Name Indication (SNI) as defined by RFC 6066.

Usage notes

1. The last parameter must be the NULL pointer.
2. If you indicate that soft validation errors should be tolerated, the application is responsible for the authentication of the server. It is highly recommended that this option only be used if an alternate authentication method is used.
3. If SSL communications is to be done by using a path to a keystore file, the user profile the application is running under must have authority to the file.
4. Digital Certificate Manager (DCM) manages an application database that contains application definitions. Each application definition encapsulates certificate processing information for a specific application. As of the IBM i 7.1 release, the application definition also encapsulates some System SSL attributes for the application. System SSL users know this application definition as an “Application ID.” Instead of specifying a path to a keystore, you can indicate what application ID to use. You would use this support to ensure consistency on what SSL attributes to use and if you do not want to give a user profile access to the system keystore file.
5. Server Name Indication (SNI) when establishing SSL connections, as defined by RFC 6066, allows TLS clients to provide to the TLS server the name of the server they are contacting. This function is used to facilitate secure connections to servers that host multiple 'virtual' servers at a single underlying network address. If the client requested FQDN does not match or no server SNI acknowledgment is sent, the secure connection will fail.
6. 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/.
7. 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)
```

8. 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.

```
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", NULL);

  // 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 16: Call class methods

Class methods	Description
Stub::setTransportProperty()	Sets transport properties.

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.

ENABLE_SSL_OVER_PROXY
: Sets whether or not a secure (SSL) connection should be used when connecting to the proxy server.

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.
- If the `type` is ENABLE_SSL_OVER_PROXY, then the possible values are "true" to use a secure connection when connecting to a proxy server, or "false" to not use a secure connection when connecting to a proxy server. The default is "false".

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 17: IHeaderBlock class methods	
Class methods	**Description**
IHeaderBlock::createNamespaceDecl()	Creates an attribute and adds it to the header block as a namespace.
IHeaderBlock::createStdAttribute()	Creates a standard header block attribute.
IHeaderBlock::createChild()	Creates a child node depending on the given node type.
IHeaderBlock::addChild()	Adds a child node to the header block.

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.

Example

See example for "Stub::createSOAPHeaderBlock()" on page 105.

IHeaderBlock::createStdAttribute()

```
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 105.

IHeaderBlock::createChild()

```
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 105.

IHeaderBlock::addChild()

```
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 105.

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 18: BasicNode class methods

Class methods	Description
BasicNode::createAttribute()	Creates an attribute and adds it to this basic node.
BasicNode::addChild()	Adds a child node to the basic node.

BasicNode::createAttribute()

```
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 105.

BasicNode::addChild()

```
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 105.

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 13. 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 + identifer 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 19 on page 119 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 19: XML to C type mapping

Schema Type	Axis Type	Actual C Type
	Numeric	
xsd:byte	`xsdc__byte`	`signed char`
xsd:decimal	`xsdc__decimal`	`double`
xsd:double	`xsdc__double`	`double`
xsd:float	`xsdc__float`	`float`
xsd:int	`xsdc__int`	`int`
xsd:integer	`xsdc__integer`	`long long`
xsd:long	`xsdc__long`	`long long`
xsd:negativeInteger	`xsdc__negativeInteger`	`long long`
xsd:nonPositiveInteger	`xsdc__nonPositiveInteger`	`long long`
xsd:nonNegativeInteger	`xsdc__nonNegativeInteger`	`unsigned long long`
xsd:positiveInteger	`xsdc__positiveInteger`	`unsigned long long`
xsd:unsignedByte	`xsdc__unsignedByte`	`unsigned char`
xsd:unsignedInt	`xsdc__unsignedInt`	`unsigned int`

Table 19: XML to C type mapping (continued)		
Schema Type	**Axis Type**	**Actual C Type**
xsd:unsignedLong	xsdc__unsignedLong	unsigned long long
xsd:unsignedShort	xsdc__unsignedShort	unsigned short
xsd:short	xsdc__short	short
Date/Time/Duration		
xsd:date	xsdc__date	struct tm
xsd:dateTime	xsdc__dateTime	struct tm
xsd:duration	xsdc__duration	long
xsd:gDay	xsdc__gDay	struct tm
xsd:gMonth	xsdc__gMonth	struct tm
xsd:gMonthDay	xsdc__gMonthDay	struct tm
xsd:gYear	xsdc__gYear	struct tm
xsd:gYearMonth	xsdc__gYearMonth	struct tm
xsd:time	xsdc__time	struct tm
String		
xsd:anyURI	xsdc__anyURI	char *
xsd:anyType	xsdc__anyType	char *
xsd:ENTITY	xsdc__ENTITY	char *
xsd:ENTITIES	xsdc__ENTITIES	char *
xsd:ID	xsdc__ID	char *
xsd:IDREFS	xsdc__IDREFS	char *
xsd:language	xsdc__language	char *
xsd:Name	xsdc__Name	char *
xsd:NCName	xsdc__NCName	char *
xsd:NMTOKEN	xsdc__NMTOKEN	char *
xsd:NMTOKENS	xsdc__NMTOKENS	char *
xsd:normalizedString	xsdc__normalizedString	char *
xsd:notation	xsdc__notation	char *
xsd:QName	xsdc__QName	char *
xsd:string	xsdc__string	char *
xsd:token	xsdc__token	char *
Other		

Table 19: XML to C type mapping (continued)		
Schema Type	**Axis Type**	**Actual C Type**

xsd:base64Binary	`xsdc__base64Binary`	Implemented as C structure:
		`typedef struct`
		`{`
		`  xsdc__unsignedByte * __ptr;`
		`  xsdc__int __size;`
		`} xsdc__base64Binary;`

xsd:boolean	`xsdc__boolean`	`enum`

xsd:hexBinary	`xsdc__hexBinary`	Implemented as C structure:
		`typedef struct`
		`{`
		`  xsdc__unsignedByte * __ptr;`
		`  xsdc__int __size;`
		`} xsdc__hexBinary;`

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`.

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 19 on page 119 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 *`):

```
<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:

```
<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:

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

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

```
int Axis_Serialize_Book(Book* param, AXISCHANDLE pSZ, AxiscBool bArray);
int Axis_DeSerialize_Book(Book* param, AXISCHANDLE 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:

```
<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 205 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 1.1 document structure" on page 24. 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 20: WSDL and XML to C mapping summary

WSDL and XML	C
`xsd:complexType` (structure) **Note:** The `xsd:complexType` can also represent an exception if referenced by a `wsdl:message` for a `wsdl:fault`.	C structure.
Nested `xsd:element` or `xsd:attribute`	C structure field.
`xsd:complexType` (array)	C Axis array structure.
`wsdl:message`	Service interface function signature.
`wsdl:portType`	Service interface function.
`wsdl:operation`	Service interface function.
`wsdl:binding`	No direct mapping, affects SOAP communications style and transport.
`wsdl:service`	No direct mapping.
`wsdl:port`	Used as default Web service location.

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 119. 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:element>
        </ati:sequence>
      </ati:complexType>
    </ati:element>
```

```
    <ati:element name="getQuoteResponse">
      <ati:complexType>
       <ati:sequence>
        <ati:element name="getQuoteReturn" type="xsd:float"></ati:element>
       </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 21: Web service proxy C functions

Function name	Description
`get_<portType-name>_stub`	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 `<install_dir>/include/axis/client/Stub.h` header file.
`destroy_<portType-name>_stub`	Function used to destroy service interface objects that are obtained by invoking `get_<portType-name>_stub`.
`get_<portType-name>_Status`	Function used to get the status of last Web service operation.
`set_<portType-name>_ExceptionHandler`	Function used to set the exception (i.e. SOAP fault) handler for the service interface.

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

```
...
<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:

```
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,
                                            AXIS_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" on page 125 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 126 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:

```
...
<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:

```
    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)).

```
...
<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? 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 wsdl: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:

```
...
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:

```
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

[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.

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 133.

[9] The `axiscAxisRegisterExceptionHandler()` function is defined in `<install_dir>/include/axis/Axis.h`.

Chapter 14. 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 22: Files in the samples directory

File name	Description
`GetQuote.wsdl`	`GetQuote` WSDL file.
`myGetQuote.c`	Client implementation code written in C.

Source listings for the client application code can be found at Appendix A, “Code Listings for myGetQuote Client Application,” on page 205.

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:

```
StockQuote.c  StockQuote.h  ws.cl
```

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:

```
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" on page 130.

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 129. However, before we continue, you should note the following points[10]:

- `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:

   ```
   cd '/myGetQuote/C'
   ```

[10] If you have not read Chapter 13, "WSDL and XML to C mappings," on page 119 then it would be a good time to do so prior to reading this section.

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:

```
COPY OBJ('<install_dir>/samples/getQuote/myGetQuote.c') TODIR('/myGetQuote/C')
```

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:

```
CRTCMOD MODULE(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 79.

Note: If you want to use one or more handlers with your application, see Chapter 9, “ Creating client-side handlers,” on page 81.

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 61.

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 15. 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`:

```
...
<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>
```

When you generate C stub code, the prototype function for setting the stub exception handler would be:

```
extern void set_MathOps_ExceptionHandler(AXISCHANDLE pStub,
                                         AXIS_EXCEPTION_HANDLER_FUNCT fp);
```

where AXIS_EXCEPTION_HANDLER_FUNCT is a typedef defined in Axis.h as:

```
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:

```
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.

```
// 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\"\n", sds->varString);
       }
       else if (0 == strcmp("OutOfBoundStruct", pcCmplxFaultName))
       {
        OutOfBoundStruct *oobs = (OutOfBoundStruct *)faultDetail;
        printf("OutOfBoundStruct Fault: \"%s\", %d, \"%s\"\n",
               oobs->varString, oobs->varInt, oobs->specialDetail->varString);
       }
       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.

```
<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:

```
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:

```
xsdc__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);
xsdc_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> </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:

```
xsdc__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 136
- "Array types as output parameters" on page 137

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.

```
// 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” on page 136, 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.

```
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...
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:

```
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:

```
extern int Axis_DeSerialize_ComplexType(ComplexType* param, AXISCHANDLE pDZ);
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 `wsdl2ws.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)
   - Nillable
   - 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 61 configuration file, or by configuring the settings for secure service using the "axiscStubSetSecure()" on page 161 Axis C API.

Using secure connections with a proxy server

The integrated web services client gives you the option to send requests to a proxy server. By default, the connection that is established is insecure. If you want to establish a secure connection to the proxy server you will need to invoke the "axiscStubSetProxySSL()" on page 160 Axis C API.

The integrated web services client also supports *SSL tunneling*. In SSL tunneling, the client establishes an insecure connection to the proxy server, and then attempts to tunnel through the proxy server to the content server over a secure connection where encrypted data is passed through the proxy server unaltered. The SSL tunneling process is as follows:

1. The client establishes an insecure connection to the proxy server.
2. The client makes a tunneling request. The proxy accepts the connection on its port, receives the request, and connects to the destination server on the port requested by the client. The proxy replies to the client that a connection is established.
3. The proxy relays SSL handshake messages in both directions: From client to destination server, and from destination server to client.
4. After the secure handshake is completed, the proxy sends and receives encrypted data to be decrypted at the client or at the destination server.

In order for SSL tunneling to occur, the proxy server needs to support SSL tunneling requests, and the web service endpoint must be a secure endpoint (i.e. https).

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 23: Behavior of Web services client for ILE with regard to cookie attributes	
Attribute	**Behavior**
expires	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.
path	This attribute is ignored. Cookies are sent on all requests and not just on requests to a URI applicable to the path.
domain	This attribute is ignored. Cookies have affinity to a stub and are domain neutral.
secure	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.

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.

```
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);
axiscStubSetMaintainSession(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:

  ```
  axiscStubDeleteTransportProperty(service, cookiename);
  ```

  For example:

  ```
  axiscStubDeleteTransportProperty( calculator, "loginCookie");
  ```

- To delete all cookies from a C stub instance:

  ```
  axiscStubDeleteTransportProperty(service, "Cookie");
  ```

  For example:

  ```
  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:

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

you must use the following formatting command:

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

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

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

you must use the following formatting command:

```
printf( "%.10g", myDouble);
```

Chapter 16. 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 45.

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 146 Axis C API.

Chapter 17. 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, copy the file `api.zip` from `/QIBM/ProdData/OS/WebServices/V1/client/docs/api.zip`, unzip it, and view the following file in a Web browser: `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 24: Axis APIs

Function	Description
axiscAxisInitialize()	Initializes the Axis runtime.
axiscAxisTerminate()	Terminates the Axis runtime.
axiscAxisDelete()	Deletes storage allocated by the Axis engine.
axiscAxisStartTrace()	Starts Axis logging.
axiscAxisStopTrace()	Stops Axis logging.
axiscAxisWriteTrace()	Writes trace data to Axis log.

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.

```
axiscAxisInitialize(0);
```

axiscAxisTerminate()

```
void axiscAxisTerminate()
```

Terminates the Axis runtime.

Example

The following example terminates the Axis client engine.

```
axiscAxisTerminate();
```

axiscAxisDelete()

```
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 `<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`.

```
axiscAxisDelete(ptr, XSDC_INT);
```

axiscAxisStartTrace()

```
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):

```
[13/11/2008 15:55:55:509] 00007860 transport > HTTPTransport::processHTTPHeader():
```

```
[13/11/2008 15:55:55:510] 00007860 transport < HTTPTransport::processHTTPHeader():
```

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:

```
[13/11/2008 15:55:55:510] 00007860 transport X HTTPTransport::readHTTPHeader():
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 a example of a transport trace:

```
.
.
.
[13/11/2008 15:55:55:280] 00007860 transport D HTTPChannel::writeBytes():
POST /axis HTTP/1.1
Host: 127.0.0.1:13260
Content-Type: text/xml; charset=UTF-8
SOAPAction: ""
Content-Length: 393


[13/11/2008 15:55:55:280] 00007860 transport < HTTPChannel::writeBytes(): Exit with integer
value of 122
[13/11/2008 15:55:55:281] 00007860 transport > HTTPChannel::writeBytes():
[13/11/2008 15:55:55:282] 00007860 transport D HTTPChannel::writeBytes():
<?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>

.
.
.
[13/11/2008 15:55:55:508] 00007860 transport D HTTPChannel::readBytes():
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

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 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
```

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()" on page 148 method.

axiscAxisStopTrace()

```
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()" on page 148 method.

axiscAxisWriteTrace()

```
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 a application-defined trace record to the Axis log.

```
#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";

  axiscAxisStartTrace("/tmp/axis.log");
  axiscAxisWriteTrace(AXISC_TRACE_TYPE_DEBUG,
                  "main-stockQuote", "start %d\n", 1);


    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 25: Stub C functions

Function	Description
axiscStubSetTransportProperty()	Sets transport properties (e.g. HTTP headers).
axiscStubGetTransportProperty()	Gets transport properties (e.g. HTTP headers).
axiscStubSetTransportConnectTimeout()	Sets the transport connect timeout.
axiscStubSetTransportTimeout()	Sets the transport timeout.
axiscStubSetTransportAutoRedirect()	Sets whether transport is to automatically handle HTTP redirects.

Table 25: Stub C functions (continued)	
Function	**Description**
axiscStubCreateSOAPHeaderBlock()	Creates and adds a SOAP header block to the stub.
axiscStubAddNamespaceToSOAPHeader()	Adds a namespace to the SOAP `Header` element.
axiscStubClearSOAPHeaderNamespaces()	Clears all namespaces from the SOAP `Header` element.
axiscStubAddAttributeToSOAPHeader()	Adds an attribute to the SOAP `Header` element.
axiscStubClearSOAPHeaderAttributes()	Clears all attributes from the SOAP `Header` element.
axiscStubAddNamespaceToSOAPBody()	Adds a namespace to SOAP Body element.
axiscStubClearSOAPBodyNamespaces()	Clears all namespaces from the SOAP Body element.
axiscStubAddAttributeToSOAPBody()	Adds an attribute to SOAP Body element.
axiscStubClearSOAPBodyAttributes()	Clears all attributes from the SOAP Body element.
axiscStubSetMaintainSession()	Sets whether to maintain session with service or not.
axiscStubSetPassword()	Sets the password to be used for basic authentication.
axiscStubSetUsername()	Sets the user name to be used for basic authentication.
axiscStubSetProxy()	Sets the proxy server and port for transport.
axiscStubSetProxySSL()	Sets the proxy server and port for transport.
axiscStubSetProxyPassword()	Sets the password to be used for proxy authentication.
axiscStubSetProxyUsername()	Sets the user name to be used for proxy authentication.
axiscStubSetSecure()	Sets SSL configuration properties.
axiscStubGetSOAPFault()	Retrieve the SOAP fault associated with the last request.

axiscStubSetTransportProperty ()

```
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.

```
axiscStubSetTransportProperty(stub, "Cookie", "sessiontoken=123345456");
```

axiscStubGetTransportProperty()

```
const char * axiscStubGetTransportProperty(AXISCHANDLE stub,
                                           const char * pcKey,
                                           AxiscBool response)
```

Searches for the transport property with the specified key. The function 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.

```
const char *cookie = axiscStubGetTransportProperty(stub, "Cookie", 1);
```

axiscStubSetTransportConnectTimeout()

```
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.

```
axiscStubSetTransportConnectTimeout(stub, 10);
```

axiscStubSetTransportTimeout()

```
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.

```
axiscStubSetTransportTimeout(stub, 10);
```

axiscStubSetTransportAutoRedirect()

```
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()

```
AXISCHANDLE axiscStubCreateSOAPHeaderBlock(AXISCHANDLE stub,
                                           AxisChar * pElemName,
                                           AxisChar * pNamespace,
                                           AxisChar * pPrefix)
```

Creates and adds a SOAP header block (i.e. SOAP header). The returned AXISCHANDLE 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;

  stub = get_StockQuote_stub("http://9.10.109.164:8088/StockQuote");


  // 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-username-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>
</wsse:Security>
```

axiscStubAddNamespaceToSOAPHeader()

```
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:

```
#include "axis/Axis.h"
#include "StockQuote.h"

#include <stdio.h>

int main()
{
.
.
.
  AXISHANDLE stub;

  stub = get_StockQuote_stub("http://9.10.109.164:8088/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:

```
<SOAP-ENV:Header xmlns:myNS="http://www.myns.com/MyNS" myNS:myAttr="myAttrValue">
```

axiscStubClearSOAPHeaderNamespaces()

```
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.

```
axiscStubClearSOAPHeaderNamespaces(stub);
```

axiscStubAddAttributeToSOAPHeader()

```
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 154.

axiscStubClearSOAPHeaderAttributes()

```
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.

```
axiscStubClearSOAPHeaderAttributes(stub);
```

axiscStubAddNamespaceToSOAPBody()

```
void axiscStubAddNamespaceToSOAPBody(AXISCHANDLE stub,
                                     const AxiscChar * pUri,
                                     const AxiscChar * 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:

```
#include "axis/Axis.h"
#include "StockQuote.h"

#include <stdio.h>

int main()
{
.
.
.
  AXISHANDLE stub;

  stub = get_StockQuote_stub("http://9.10.109.164:8088/StockQuote");


  // 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:

```
<SOAP-ENV:Body xmlns:myNS="http://www.myns.com/MyNS" myNS:myAttr="myAttrValue">
```

axiscStubClearSOAPBodyNamespaces()

```
void axiscStubClearSOAPBodyNamespaces(AXISCHANDLE stub)
```

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.

```
axiscStubClearSOAPBodyNamespaces(stub);
```

axiscStubAddAttributeToSOAPBody()

```
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 156.

axiscStubClearSOAPBodyAttributes()

```
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.

```
axiscStubClearSOAPBodyAttributes(stub);
```

axiscStubSetMaintainSession()

```
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 * pcProxyHost,
                       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);
```

axiscStubSetProxySSL()

```
void axiscStubSetProxySSL(AXISCHANDLE stub,
                          AxiscBool useSSL)
```

Sets whether or not a secure (SSL) connection should be used when connecting to the proxy server.

Parameters

stub	Pointer to stub object.
useSSL	Integer boolean flag. When set to zero (the default), an unsecure connection will be established when connecting to a proxy server. When set to a non-zero value, a secure connection will be established when connecting to a proxy server.

Usage notes

1. In order for a secure connection to be established, the API `axiscStubSetSecure()` must have been called.

Example

Following example tells the client library to use a secure connection when connecting to a proxy server.

```
axiscStubSetProxySSL(stub, 1);
```

axiscStubSetProxyPassword()

```
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()

```
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()

```
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. This parameter is ignored if the application ID parameter is set to a value.
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 as long as the user has authority to the certificate store file and the password has been used one time one the system. To specify any of the subsequent optional parameters, you must pass a value for this parameter. This parameter is ignored if the application ID parameter is set to a value.
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. This parameter is ignored if the application ID parameter is set to a value.
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.

pTLSv11Cipher (optional) Pointer to null-terminated character string indicating whether to enable or disable the TLS Version 1.1 ciphers. A value of "NONE" will disable the ciphers; any other value will enable the ciphers. By default, the TLS Version 1.1 ciphers are enabled.

pTLSv12Cipher (optional) Pointer to null-terminated character string indicating whether to enable or disable the TLS Version 1.2 ciphers. A value of "NONE" will disable the ciphers; any other value will enable the ciphers. By default, the TLS Version 1.2 ciphers are enabled.

pTolerate (optional) Pointer to null-terminated character string indicating whether to tolerate soft validation errors (expired certificate or certificate not in certificate store). Specify a value of `true` to tolerate soft validation errors, or `false` to not tolerate soft validation errors. The default is `false`.

pAppid (optional) Pointer to null-terminated character string indicating the application ID to use for the SSL environment.

pFQDN (optional) Pointer to null-terminated character string indicating the fully qualified domain name that will be used as Server Name Indication (SNI) as defined by RFC 6066.

Usage notes

1. The last parameter must be the NULL pointer.
2. If you indicate that soft validation errors should be tolerated, the application is responsible for the authentication of the server. It is highly recommended that this option only be used if an alternate authentication method is used.
3. If SSL communications is to be done by using a path to a keystore file, the user profile the application is running under must have authority to the file.
4. Digital Certificate Manager (DCM) manages an application database that contains application definitions. Each application definition encapsulates certificate processing information for a specific application. As of the IBM i 7.1 release, the application definition also encapsulates some System SSL attributes for the application. System SSL users know this application definition as an “Application ID.” Instead of specifying a path to a keystore, you can indicate what application ID to use. You would use this support to ensure consistency on what SSL attributes to use and if you do not want to give a user profile access to the system keystore file.
5. Server Name Indication (SNI) when establishing SSL connections, as defined by RFC 6066, allows TLS clients to provide to the TLS server the name of the server they are contacting. This function is used to facilitate secure connections to servers that host multiple 'virtual' servers at a single underlying network address. If the client requested FQDN does not match or no server SNI acknowledgment is sent, the secure connection will fail.
6. 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/.
7. 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)
```

8. 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 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:

```
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", NULL);

// Remainder of application
.
.
.
// End of application
destroy_ITestService_stub(serviceSecure);
return 0;
}
```

axiscStubGetSOAPFault()

```
AXISHANDLE axiscStubGetSOAPFault(AXISCHANDLE stub)
```

Returns the SOAP fault object associated with the last operation issued against the stub handle. The function returns NULL if there is no SOAP fault.

Parameters

stub	Pointer to stub object.

Returns

The SOAP fault object or NULL if it was not found.

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 26: Header block C functions

Function	Description
axiscHeaderBlockCreateNamespaceDeclINamespace()	Creates an attribute and adds it to the header block as a namespace.
axiscHeaderBlockCreateStdAttribute()	Creates a standard header block attribute.
axiscHeaderBlockCreateChildBasicNode()	Creates a child node depending on the given node type.
axiscHeaderBlockAddChild()	Adds a child node to the header block.

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 152.

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.

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 152.

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 152.

axiscHeaderBlockAddChild()

```
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 152.

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 27: Basic node C functions

Function	Description
axiscBasicNodeCreateAttribute()	Creates an attribute and adds it to this basic node.
axiscBasicNodeAddChild()	Adds a child node to the basic node.

axiscBasicNodeCreateAttribute()

```
AXISCHANDLE axiscBasicNodeCreateAttribute(AXISCHANDLE 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 152.

axiscBasicNodeAddChild()

```
int axiscBasicNodeAddChild(AXISCHANDLE basicNode,
                           AXISCHANDLE 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 152.

SOAP fault C APIs

The SOAP fault object is the base object that is used to construct objects containing SOAP fault information. The SOAP fault C APIs are defined in include file `<install_dir>/include/axis/ISoapFault.h`.

The following table lists the most commonly used methods.

Table 28: SOAP fault C functions	
Function	**Description**
axiscSoapFaultGetCmplxFaultObjectName	Get the SOAP fault name.
axiscSoapFaultGetFaultactor	Get the SOAP fault actor.
axiscSoapFaultGetFaultcode	Get the SOAP fault code.
axiscSoapFaultGetFaultstring	Get the SOAP fault string.
axiscSoapFaultGetSimpleFaultDetail	Get the SOAP fault detail.

axiscSoapFaultGetCmplxFaultObjectName()

```
const AxiscChar * axiscSoapFaultGetCmplxFaultObjectName(AXISCHANDLE soapFault)
```

Retrieve the identifier associated with the SOAP fault object.

Parameters

soapFault Pointer to SOAP fault object.

Returns

Pointer to null-terminated string representing the identifier of the SOAP fault. The ownership of the memory allocated for the object is owned by the stub.

axiscSoapFaultGetFaultactor()

```
const AxiscChar * axiscSoapFaultGetFaultactor(AXISCHANDLE soapFault)
```

Retrieve the SOAP fault actor.

Parameters

soapFault Pointer to SOAP fault object.

Returns

Pointer to null-terminated string representing the SOAP fault actor. The ownership of the memory allocated for the object is owned by the stub.

axiscSoapFaultGetFaultcode()

```
const AxiscChar * axiscSoapFaultGetFaultcode(AXISCHANDLE soapFault)
```

Retrieve the SOAP fault code.

Parameters

soapFault Pointer to SOAP fault object.

Returns

Pointer to null-terminated string representing the SOAP fault code. The ownership of the memory allocated for the object is owned by the stub.

axiscSoapFaultGetFaultstring()

```
const AxiscChar * axiscSoapFaultGetFaultstring(AXISCHANDLE soapFault)
```

Retrieve the SOAP fault string.

Parameters

soapFault Pointer to SOAP fault object.

Returns

Pointer to null-terminated string representing the identifier of the SOAP fault string. The ownership of the memory allocated for the object is owned by the stub.

axiscSoapFaultGetSimpleFaultDetail()

```
const AxiscChar * axiscSoapFaultGetSimpleFaultDetail(AXISCHANDLE soapFault)
```

Retrieve the simple fault detail string.

Parameters

soapFault Pointer to SOAP fault object.

Returns

Pointer to null-terminated string representing the fault data as a string.

Transport C APIs

The transport APIs may be used by client applications that want to control what is sent to a server and what is received from the server. Only textual data may be sent or received. Data sent by a client application is converted to UTF-8. Data received in response to a client request is assumed to be in UTF-8. The transport C APIs are defined in include file `<install_dir>/include/axis/ITransport.h`.

The following table lists the most commonly used functions.

Table 29: Transport C functions	
Function	**Description**
axiscTransportCreate()	Create a transport object.
axiscTransportDestroy()	Destroy a transport object.
axiscTransportReset()	Resets the transport object to its initial state.
axiscTransportSetProperty()	Sets a transport property.
axiscTransportGetProperty()	Gets a transport property.
axiscTransportSend()	Send bytes over transport.
axiscTransportFlush()	Flush the transport of any buffered data.
axiscTransportReceive()	Receive data from the transport.
axiscTransportGetLastErrorCode()	Get transport error code from last unsuccessful transport operation.
axiscTransportGetLastError()	Get transport error string from last unsuccessful transport operation.

axiscTransportCreate()

```
AXISCHANDLE axiscTransportCreate(const char * uri,
                                 int protocol)
```

Creates a transport object.

Parameters

uri
Pointer to null-terminated character string representing the URI that will be used to connect to the server.

protocol
Transport protocol to use. The protocols supported follows:

AXISC_PROTOCOL_HTTP11
The transport object should be created that uses the Hypertext Transfer Protocol (HTTP) 1.1.

Returns

Pointer to transport object if function call is successful; NULL if transport object cannot be created.

Example

The following example creates a HTTP 1.1 transport object.

```
AXISHANDLE h = axiscTransportCreate("http://hostname:10035/web/services/ECHOPATH",
                                    AXISC_PROTOCOL_HTTP11);
```

axiscTransportDestroy()

```
int axiscTransportDestroy(AXISCHANDLE tHandle)
```

Destroys the object created by `axiscTransportCreate()`.

Parameters

tHandle
Pointer to transport object.

Returns

Zero if call to function is successful; -1 on failure. If the function call fails, the transport object is unusable.

Example

```
axiscTransportDestroy(tHandle);
```

axiscTransportReset()

```
int axiscTransportReset(AXISCHANDLE tHandle,
                        const char * uri)
```

Resets the transport object to its initial state.

Parameters

tHandle
Pointer to transport object.

uri
Pointer to null-terminated character string representing the URI that will be used on the reset of the transport object. A NULL value will result in the URI that was used on the `axiscTransportCreate()` function call to be used.

Returns

Zero if call to function is successful; -1 on failure.

Example

The following example resets the transport object.

```
axiscTransportReset(tHandle, NULL);
```

axiscTransportSetProperty()

```
int axiscTransportSetProperty(AXISCHANDLE tHandle,
                              int type,
                              ...)
```

Sets a transport property.

Parameters

tHandle
Pointer to transport object.

type
An integer that specifies the property to be set. Possible values:

AXISC_PROPERTY_HTTP_BASICAUTH
Sets the user ID and password that will be used for HTTP basic authentication. The next two parameters must be pointers to character strings. The first parameter is the user ID, and the second parameter is the password.

AXISC_PROPERTY_HTTP_HEADER
Sets an HTTP header in the HTTP request. The next two parameters must be pointers to character strings. The first parameter is the header name, and the second parameter is the header value.

Note: If the content-type HTTP header is not set, the default content type that will be used is:

```
Content-Type: text/xml; charset=UTF-8
```

AXISC_PROPERTY_HTTP_METHOD
Sets the HTTP method in the HTTP request. The next parameter must be a pointer to a character string. The most common HTTP methods include GET, POST, PUT, and DELETE. The default HTTP method is GET.

AXISC_PROPERTY_HTTP_PROXY
Sets the HTTP proxy information. The next two parameters must be pointers to character strings. The first parameter is the HTTP proxy host, and the second parameter is the HTTP proxy port.

AXISC_PROPERTY_HTTP_PROXYAUTH
Sets the HTTP proxy authentication information. The next two parameters must be pointers to character strings. The first parameter is the user ID, and the second parameter is the password.

AXISC_PROPERTY_HTTP_PROXYSSL
Sets whether or not a secure (SSL) connection should be used when connecting to the proxy server. The next parameter must be a pointer to a character string that is set to either "true" or "false". A value of "true" indicates that the transport will connect to the proxy server using a secure channel. A value of "false" indicates that the transport will connect to the proxy server using an unsecure channel (note that to use a secure channel, the property AXISC_PROPERTY_HTTP_SSL must be set.)

AXISC_PROPERTY_HTTP_REDIRECT
Sets whether the transport object should follow HTTP redirects. The next parameter must be a pointer to an integer. If the value is greater than zero, the transport object will follow HTTP redirects up to the number specified. If the value is less than one, HTTP redirects will not be followed.

AXISC_PROPERTY_HTTP_SSL
Sets SSL information. Up to ten parameters that are pointers to character strings may be passed:

- Pointer to null-terminated character string representing the certificate store file to be used for the secure session or SSL environment. This parameter is ignored if the application ID parameter is set to a value. If SSL communications is to be done by using a path to a keystore file, the user profile the application is running under must have authority to the file.
- (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 as long as the user has authority to the certificate store file and the password has been used one time one the system. To specify any of the subsequent optional parameters, you must pass a value for this parameter. This parameter is ignored if the application ID parameter is set to a value.
- (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. This parameter is ignored if the application ID parameter is set to a value.

- (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.
- (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.
- (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.
- (optional) Pointer to null-terminated character string indicating whether to enable or disable the TLS Version 1.1 ciphers. A value of "NONE" will disable the ciphers; any other value will enable the ciphers. By default, the TLS Version 1.1 ciphers are enabled.
- (optional) Pointer to null-terminated character string indicating whether to enable or disable the TLS Version 1.2 ciphers. A value of "NONE" will disable the ciphers; any other value will enable the ciphers. By default, the TLS Version 1.2 ciphers are enabled.
- (optional) Pointer to null-terminated character string indicating whether to tolerate soft validation errors (expired certificate or certificate not in certificate store). Specify a value of `true` to tolerate soft validation errors, or `false` to not tolerate soft validation errors. The default is `false`.
- (optional) Pointer to null-terminated character string indicating the application ID to use for the SSL environment.
- (optional) Pointer to null-terminated character string indicating the fully qualified domain name that will be used as Server Name Indication (SNI) as defined by RFC 6066.

Note: The last parameter must be the NULL pointer.

AXISC_PROPERTY_CONNECT_TIMEOUT
Sets the connect timeout value. The next parameter must be a pointer to an integer. If the value is greater than zero, the value will be used as the maximum time, in seconds, to wait for a connection to complete. The default value is dependent on TCP/IP system settings.

AXISC_PROPERTY_CONVERT_PAYLOAD_REQUEST
Sets whether data conversion is to occur for the *request* payload. By default, data that is sent is converted from the job's coded character set identifier (CCSID) to UTF-8 before it is sent. The next parameter must be a pointer to a character string that is set to either `true` or `false`. If the value is `true`, data conversion will occur. If the value is `false`, data conversion will not occur. The default value is `true`.

AXISC_PROPERTY_CONVERT_PAYLOAD_RESPONSE
Sets whether data conversion is to occur for the *response* payload. By default, data that is received is converted from UTF-8 to the job's CCSID. The next parameter must be a pointer to a character string that is set to either `true` or `false`. If the value is `true`, data conversion will occur. If the value is `false`, data conversion will not occur. The default value is `true`.

AXISC_PROPERTY_CONVERT_PAYLOAD
Sets whether data conversion is to occur. By default, data that is sent is converted from the job's coded character set identifier (CCSID) to UTF-8 before it is sent, and data that is received is converted from UTF-8 to the job's CCSID. The next parameter must be a pointer to a character string that is set to either `true` or `false`. If the value is `true`, data conversion will occur. If the value is `false`, data conversion will not occur. The default value is `true`.

AXISC_PROPERTY_IO_TIMEOUT
Sets how long to wait on a read request to complete. The next parameter must be a pointer to an integer. If the value is greater than zero, the value will be used as the maximum time, in seconds, to wait for a read request to complete. The default value is dependent on TCP/IP system settings.

Returns

Zero if call to function is successful; -1 on failure.

Example

The following example set the transport connect timeout to 10 seconds.

```
int timeout=10;
int rc = axiscTransportSetProperty(tHandle,
                                   AXISC_PROPERTY_CONNECT_TIMEOUT, &timeout);
```

The following example sets SSL information, ensuring that only TLS 1.2 is enabled.

```
int rc = axiscTransportSetProperty(tHandle, AXISC_PROPERTY_HTTP_SSL,
                                   "/QIBM/USERDATA/ICSS/CERT/SERVER/DEFAULT.KDB",
                                   "", "",
                                   "NONE","NONE", "NONE", "NONE", NULL);
```

axiscTransportGetProperty()

```
int axiscTransportGetProperty(AXISCHANDLE tHandle,
                              int type,
                              ...)
```

Gets a transport property.

Parameters

tHandle
Pointer to transport object.

type
An integer that specifies the property to be retrieved. Possible values:

AXISC_PROPERTY_HTTP_HEADER
Retrieve an HTTP header from the server response. The next two parameters must be pointers. The first parameter is a pointer to a character string representing the HTTP header to retrieve. The second parameter must be a pointer to a pointer. If the header to be retrieved is found, the pointer will be set to the character string pointer representing the header value. The pointer storage is still owned by the transport object and should not be modified. If the header is not found, the pointer field will be set to NULL.

Note: In order to retrieve HTTP headers in a response, a function call to `axiscTransportReceive()` must have previously been done.

AXISC_PROPERTY_HTTP_HEADERS_RESPONSE
Retrieve list of HTTP header names in the response. The next parameter must be a pointer to a pointer. The pointer will be set to the character string pointer containing a colon-delimited list of HTTP header names. The pointer storage is still owned by the transport object and should not be modified.

Note: In order to retrieve HTTP headers in a response, a function call to `axiscTransportReceive()` must have previously been done.

AXISC_PROPERTY_HTTP_STATUS_CODE
Retrieve the HTTP status code that is returned by the server in the response. The next parameter must be a pointer to a pointer. The pointer will be set to the character string pointer representing

the HTTP status code. The pointer storage is still owned by the transport object and should not be modified.

Note: In order to retrieve the HTTP status code, a function call to `axiscTransportReceive()` must have previously been done.

Returns

Zero if call to function is successful; -1 on failure.

Example

The following example retrieves the HTTP status code.

```
char *statusCode;
int rc = axiscTransportGetProperty(tHandle,
                                   AXISC_PROPERTY_HTTP_STATUS_CODE, &statusCode);
```

axiscTransportSend()

```
int axiscTransportSend(AXISCHANDLE tHandle,
                       const char *buffer,
                       int bytesTosend,
                       int flags)
```

Send data to a server.

Parameters

tHandle
Pointer to transport object.

buffer
The pointer to the buffer in which the data that is to be sent is stored.

bytesTosend
Number of bytes to send.

flags
Reserved. Must be set to zero.

Returns

Number of bytes that will be sent if call to function is successful; -1 on failure.

Usage notes

1. By default the `AXISC_PROPERTY_CONVERT_PAYLOAD` transport property is set to true, which means the data that is to be sent must be in the coded character set identifier (CCSID) of the job. The data is converted to UTF-8 before it is sent. If no conversion is to be done you will need to set the `AXISC_PROPERTY_CONVERT_PAYLOAD` transport property to false.
2. Every time the function `axiscTransportSend()` is called, the data that is to be sent is buffered in the transport object. The data is not actually sent until the `axiscTransportFlush()` function is invoked.
3. In the case where there is no data to be sent, `axiscTransportFlush()` must still be invoked to initiate request and send any transport specific data (e.g. HTTP headers).
4. If the content-type HTTP header has not been set, the default content type that will be used is:

   ```
   Content-Type: text/xml; charset=UTF-8
   ```

Example

The following example send XML data.

```
char *buffer = "<TEMPIN>2337</TEMPIN>";
int rc = axiscTransportSend(tHandle, buffer, strlen(buffer), 0);
```

axiscTransportFlush()

```
int axiscTransportFlush(AXISCHANDLE tHandle)
```

Flushes the transport of any buffered data. When called, a connection to server is established, the request is built and sent.

Parameters

tHandle
Pointer to transport object.

Returns

Zero if call to function is successful; -1 on failure.

Usage notes

1. The next function that should be called after a successful invocation of `axiscTransportFlush()` is `axiscTransportReceive()`. This should be done in order to receive any transport-specific protocol data (e.g. HTTP headers) and to receive the payload, if any. Alternatively, you can also invoke `axiscTransportReset()` or `axiscTransportDestroy()`.

axiscTransportReceive()

```
int axiscTransportReceive(AXISCHANDLE tHandle,
                          char *buffer,
                          int bufferLen,
                          int flags)
```

Receive data through a transport object.

Parameters

tHandle
Pointer to transport object.

buffer
The pointer to the buffer in which the data that is to be read is stored.

bufferLen
The length of the buffer that will be used to store the data that is read.

flags
Reserved. Must be set to zero.

Returns

Number of bytes received if call to function is successful; -1 on failure.

Usage notes

1. On the first call to `axiscTransportReceive()`, the transport object will attempt to read all the data and store the data in the transport object.
2. By default the `AXISC_PROPERTY_CONVERT_PAYLOAD` transport property is set to true, which means the data that is received is converted from UTF-8 to the coded character set identifier (CCSID) of the job. If no conversion is to be done you will need to set the `AXISC_PROPERTY_CONVERT_PAYLOAD` transport property to false.

Example

The following example receives data.

```
char buffer[5001];

// Receive data
rc = axiscTransportReceive(tHandle, buffer, 5000, 0);
if (rc == 0)
   printf("No data to read\n");

while (rc > 0)
{
   bytesRead += rc;
   rc = axiscTransportReceive(t, buffer+bytesRead, 5000-bytesRead, 0);
}

// Dump data to stdout
if (rc == -1)
   printf("Error on read\n");
else if (bytesRead > 0)
{
   buffer[bytesRead] = 0x00;
   printf("Data: \n%s\n\n",  buffer);
}
```

axiscTransportGetLastErrorCode()

```
int axiscTransportGetLastErrorCode(AXISCHANDLE tHandle)
```

Retrieves the transport error code from last unsuccessful transport operation. The error codes are listed in the file `<install_dir>/include/axis/AxisException.h`.

Parameters

tHandle
Pointer to transport object.

Returns

Last error code from last unsuccessful transport operation.

Example

The following example retrieves error code from transport object.

```
int rc = axiscTransportGetLastErrorCode(tHandle);
if (rc == SERVER_TRANSPORT_HTTP_EXCEPTION)
{
  rc = axiscTransportGetProperty(t, AXISC_PROPERTY_HTTP_STATUS_CODE, &statusCode);
  if (rc != -1)
    printf("HTTP Status code:%s\n", statusCode);
}
```

axiscTransportGetLastError()

```
const char * axiscTransportGetLastError(AXISCHANDLE tHandle)
```

Retrieves the transport error string from last unsuccessful transport operation.

Parameters

tHandle
Pointer to transport object.

Returns

Pointer to character string. Storage is owned by transport and must not be modified.

Example

The following example retrieves error code string from transport object.

```
const char * errorString = axiscTransportGetLastError(h);
printf(errorString);
```

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 18. WSDL and XML to RPG mappings

The `wsdl2ws.sh`[11] 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 119.

XML schema to RPG type mapping

Table 30 on page 181 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 30: XML to RPG type mapping

Schema Type	Actual RPG type
	Numeric

xsd:byte	Implemented as RPG data structure:
	`D xsd_byte...`
	`D DS qualified based(Template)`
	`D isNil 1n`
	`D value 1a`

xsd:decimal	Implemented as RPG data structure:
	`D xsd_decimal...`
	`D DS qualified based(Template)`
	`D isNil 1n`
	`D value 8f`

xsd:double	Implemented as RPG data structure:
	`D xsd_double...`
	`D DS qualified based(Template)`
	`D isNil 1n`
	`D value 8f`

xsd:float	Implemented as RPG data structure:
	`D xsd_float...`
	`D DS qualified based(Template)`
	`D isNil 1n`
	`D value 4f`

[11] In this chapter, anything we mention about the `wsdl2ws.sh` tool also holds true for the `wsdl2rpg.sh` tool.

Table 30: XML to RPG type mapping (continued)

Schema Type	Actual RPG type

xsd:int	Implemented as RPG data structure:
	`D xsd_int...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                        10i 0`

xsd:integer	Implemented as RPG data structure:
	`D xsd_integer...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                        20i 0`

xsd:long	Implemented as RPG data structure:
	`D xsd_long...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                        20i 0`

xsd: negativeInteger	Implemented as RPG data structure:
	`D xsd_negativeInteger...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                        20i 0`

xsd: nonPositiveInteger	Implemented as RPG data structure:
	`D xsd_nonPositiveInteger...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                        20i 0`

xsd: nonNegativeInteger	Implemented as RPG data structure:
	`D xsd_nonNegativeInteger...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                        20u 0`

xsd: positiveInteger	Implemented as RPG data structure:
	`D xsd_positiveInteger...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                        20u 0`

xsd: unsignedByte	Implemented as RPG data structure:
	`D xsd_unsignedByte...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                         1a`

Table 30: XML to RPG type mapping (continued)

Schema Type	Actual RPG type

xsd: unsignedInt	Implemented as RPG data structure:
	`D xsd_unsignedInt...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                        10u 0`

xsd: unsignedLong	Implemented as RPG data structure:
	`D xsd_unsignedLong...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                        20u 0`

xsd: unsignedShort	Implemented as RPG data structure:
	`D xsd_unsignedShort...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                         5u 0`

xsd:short	Implemented as RPG data structure:
	`D xsd_short...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                         5i 0`

Date/Time/Duration	

xsd:date	Implemented as RPG data structure named xsd_date. The structure is defined as follows:
	`D xsd_date...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                          1n`
	`D  value                               likeds(xsd_tm)`
	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`

Table 30: XML to RPG type mapping (continued)

Schema Type	Actual RPG type
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	

xsd:anyURI	Implemented as RPG data structure named xsd_anyURI. For IBM i 6.1 and later releases the structure is defined as follows:
	`D xsd_anyURI...`
	`D DS qualified based(Template)`
	`D isNil 1n`
	`D value a varying(4) len(nnnnn)`
	`D reserved 1a`
	where *nnnnn* is the size of the character field.
	For i 5.4, the RPG data structure would be defined as follows:
	`D xsd_anyURI...`
	`D DS qualified based(Template)`
	`D isNil 1n`
	`D value nnnnna varying`
	`D reserved 1a`
	Note: The length that is used when defining the character field is directly related to the `-ms` argument on the `wsdl2ws.sh` tool. See "wsdl2ws.sh command" on page 57 for further details.

xsd:anyType	Implemented as RPG data structure named xsd_anyType. The data structure is defined in the same way as xsd:anyURI.
xsd:ENTITY	Implemented as RPG data structure named xsd_ENTITY. The data structure is defined in the same way as xsd:anyURI.
xsd:ENTITIES	Implemented as RPG data structure named xsd_ENTITIES. The data structure is defined in the same way as xsd:anyURI.

Table 30: XML to RPG type mapping (continued)

Schema Type	Actual RPG type
xsd:ID	Implemented as RPG data structure named xsd_ID. The data structure is defined in the same way as xsd:anyURI.
xsd:IDREFS	Implemented as RPG data structure named xsd_IDREFS. The data structure is defined in the same way as xsd:anyURI.
xsd: language	Implemented as RPG data structure named xsd_language. The data structure is defined in the same way as xsd:anyURI.
xsd:Name	Implemented as RPG data structure named xsd_Name. The data structure is defined in the same way as xsd:anyURI.
xsd:NCName	Implemented as RPG data structure named xsd_NCName. The data structure is defined in the same way as xsd:anyURI.
xsd:NMTOKEN	Implemented as RPG data structure named xsd_NMTOKEN. The data structure is defined in the same way as xsd:anyURI.
xsd:NMTOKENS	Implemented as RPG data structure named xsd_NMTOKENS. The data structure is defined in the same way as xsd:anyURI.
xsd: normalizedString	Implemented as RPG data structure named xsd_normalizedString. The data structure is defined in the same way as xsd:anyURI.
xsd:notation	Implemented as RPG data structure named xsd_notation. The data structure is defined in the same way as xsd:anyURI.
xsd:QName	Implemented as RPG data structure named xsd_QName. The data structure is defined in the same way as xsd:anyURI.
xsd:string	Implemented as RPG data structure named xsd_string. The data structure is defined in the same way as xsd:anyURI.
xsd:token	Implemented as RPG data structure named xsd_token. The data structure is defined in the same way as xsd:anyURI.
Other	

xsd: base64Binary	Implemented as RPG data structure named xsd_base64Binary. For i 6.1 and later releases the structure is defined as follows:
	`D xsd_base64Binary...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                        1n`
	`D  value                         a   varying(4) len(nnnnn)`
	where *nnnnn* is the size of the character field.
	For i 5.4, the RPG data structure would be defined as follows:
	`D xsd_base64Binary...`
	`D                 DS                  qualified based(Template)`
	`D  isNil                        1n`
	`D  value                    nnnnna   varying`
	Note: The length that is used when defining the character field is directly related to the `-mb` argument on the `wsdl2ws.sh` tool. See “wsdl2ws.sh command” on page 57 for further details.

Table 30: XML to RPG type mapping (continued)	
Schema Type	**Actual RPG type**

xsd:boolean	Implemented as RPG data structure:
	`D xsd_boolean...`
	`D DS qualified based(Template)`
	`D isNil 1n`
	`D value 10i 0`

xsd:hexBinary	Implemented as RPG data structure named xsd_hexBinary. For i 6.1 and later releases the structure is defined as follows:
	`D xsd_hexBinary...`
	`D DS qualified based(Template)`
	`D isNil 1n`
	`D value a varying(4) len(1024)`
	where *nnnnn* is the size of the character field.
	For i 5.4, the RPG data structure would be defined as follows:
	`D xsd_hexBinary...`
	`D DS qualified based(Template)`
	`D isNil 1n`
	`D value` *nnnnn*`a varying`
	Note: The length that is used when defining the character field is directly related to the `-mb` argument on the `wsdl2ws.sh` tool. See "wsdl2ws.sh command" on page 57 for further details.

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.rpgleinc`, where *<portType>* is the `wsdl:portType` defined in the WSDL and discussed in "WSDL to RPG mapping" on page 188. The reason that they are defined dynamically is because the string and array length values are obtained from what you specify on the `wsdl2ws.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:

```
<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:

```
D Book_t...
D                 DS                  qualified based(Template)
D isNil_Book_t                   1n
D reviewer                           likeds(xsd_string)
D author                             likeds(xsd_string)
D price                              likeds(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:

```
D <array_name>...
D                 DS                  qualified based(Template)
D isNil_<array_name>...
D                                 1n
D array                              likeds(<data-structure-name>)
D                                    dim(<nnnnn>)
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 `wsdl2ws.sh` tool. See “wsdl2ws.sh command” on page 57 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                           1n
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 wsdl2ws.sh tool. See "wsdl2ws.sh command" on page 57 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                     1n
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 205 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 1.1 document structure" on page 24. 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 31: WSDL and XML to RPG mapping summary	
WSDL and XML	**RPG**
`xsd:complexType` (structure)	RPG DS structure.
Nested `xsd:element` or `xsd:attribute`	RPG DS structure field.
`xsd:complexType` (array)	RPG DS Axis array structure.
`wsdl:message`	Service interface function signature.
`wsdl:portType`	Service interface function.
`wsdl:operation`	Service interface function.
`wsdl:binding`	No direct mapping, affects SOAP communications style and transport.
`wsdl:service`	No direct mapping.
`wsdl:port`	Used as default Web service location.

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 181. 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:element>
        </ati:sequence>
      </ati:complexType>
    </ati:element>

    <ati:element name="getQuoteResponse">
      <ati:complexType>
       <ati:sequence>
        <ati:element name="getQuoteReturn" type="xsd:float"></ati:element>
       </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 `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 in the names of the Web service proxy (termed service interface) functions. A port type is mapped to 2 functions:

Table 32: Web service proxy RPG functions

Function name	Description
`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).
`stub_destroy_<portType-name>`	Function used to destroy service interface objects that are obtained by invoking `stub_create_<portType-name>`.

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

```
...
<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 RPG functions:

```
D stub_create_StockQuote...
D                 PR             1N   extproc('stub_create_StockQuote@')
D this                                likeds(This_t)

D stub_destroy_StockQuote...
D                 PR             1N   extproc('stub_destroy_StockQuote@')
D this                                likeds(This_t)

D stub_op_getQuote...
D                 PR             1N   extproc('getQuote@')
D this                                likeds(This_t)
D Value0                              likeds(xsd_string)
D out                                 likeds(xsd_float)
```

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 uses 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 indicate 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 197.
- 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 191 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 190.

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 references wsdl:message elements describing the fault (refer to "Mapping of wsdl:fault" on page 192 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:

```
...
<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:

```
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.

```
...
<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 190.

More information on exception handling in RPG can be found in “RPG exception handling” on page 197.

Chapter 19. 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 `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 33: Files in the samples directory

File name	Description
`GetQuote.wsdl`	`GetQuote` WSDL file.
`myGetQuote.rpgle`	Client implementation code written in RPG.

Source listings for the client application code can be found at Appendix A, “Code Listings for myGetQuote Client Application,” on page 205.

Generating the RPG stub code

Before you can create a web service client application, you must first generate the RPG 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 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 `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 RPG stub as shown in following example:

```
<install_dir>/bin/wsdl2ws.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 `wsdl2ws.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 `wsdl2ws.sh` tool is shown below:

```
StockQuote_util.rpgle           StockQuote.cl
StockQuote_util.rpgleinc        StockQuote.h
StockQuote_xsdtypes.rpgleinc    StockQuote.rpgle
StockQuote.c                    StockQuote.rpgleinc
```

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:

- StockQuote_util.rpgle – RPG utility routines.
- StockQuote_util.rpgleinc – RPG utility routines include.
- StockQuote_xsdtypes.rpgleinc – standard data types include.
- StockQuote.rpgle – RPG Web service implementation code.
- StockQuote.rpgleinc – RPG Web service include.

From an RPG programmer perspective, the only files you would need to look at are the `StockQuote.rpgleinc` and `StockQuote_xsdtypes.rpgleinc` files.

Finally, there is also the file `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:

```
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
```

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” on page 195.

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 `wsdl2ws.sh` tool as described in "Generating the RPG stub code" on page 193. However, before we continue, you should note the following points[12]:

- `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" on page 195.

Note: If you want to use one or more handlers with your application, see Chapter 9, " Creating client-side handlers," on page 81.

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 18, "WSDL and XML to RPG mappings," on page 181 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 `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 61.

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 20. 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. In addition, if an exception has occurred you may determine if a SOAP fault is available by using the axiscStubGetSOAPFault() API and then using the various SOAP fault APIs to get data from the SOAP fault object.

The following shows how a client application may process exceptions.

```
.
.
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;

     soapFault = axiscStubGetSOAPFault(WsStub.handle);
     if (soapFault <> *NULL);
        FaultCode = axiscSoapFaultGetFaultcode(soapFault);
        FaultString = axiscSoapFaultGetFaultstring(soapFault);
        FaultActor = axiscSoapFaultGetFaultactor(soapFault);
        FaultDetail = axiscSoapFaultGetSimpleFaultDetail(soapFault);
     endif;
  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 61 configuration file, or by configuring the settings for secure service using the “axiscStubSetSecure()” on page 161 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.

```
h 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)
d NULLSTR         s              1

 /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.
      // Disable SSLv2, SSLv2, TLSv1
          NULLSTR = X'00';
      axiscStubSetSecure(WsStub.handle:
                         '/QIBM/USERDATA/ICSS/CERT/SERVER/DEFAULT.KDB':
                         NULLSTR: NULLSTR:
                         'NONE': 'NONE': 'NONE' : *NULL);

      // Remainder of application
      .
      .
      .
      // Destroy Web service stub.
      stub_destroy_StockQuote(WsStub);
    endif;
```

```
    *INLR=*ON;
 /end-free
```

For further information on the SSL parameters, see the “axiscStubSetSecure()” on page 161 Axis C API.

Using secure connections with a proxy server

The integrated web services client gives you the option to send requests to a proxy server. By default, the connection that is established is unsecure. If you want to establish a secure connection to the proxy server you will need to invoke the “axiscStubSetProxySSL()” on page 160 Axis C API.

The integrated web services client also supports *SSL tunneling*. In SSL tunneling, the client establishes an unsecure connection to the proxy server, and then attempts to tunnel through the proxy server to the content server over a secure connection where encrypted data is passed through the proxy server unaltered. The SSL tunneling process is as follows:

1. The client establishes an unsecure connection to the proxy server.
2. The client makes a tunneling request. The proxy accepts the connection on its port, receives the request, and connects to the destination server on the port requested by the client. The proxy replies to the client that a connection is established.
3. The proxy relays SSL handshake messages in both directions: From client to destination server, and from destination server to client.
4. After the secure handshake is completed, the proxy sends and receives encrypted data to be decrypted at the client or at the destination server.

In order for SSL tunneling to occur, the proxy server needs to support SSL tunneling requests, and the web service endpoint must be a secure endpoint (i.e. https).

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:

```
<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>
</wsse:Security>
```

The following RPG example uses various Axis C APIs to set SOAP header information.

```
h 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)
D phb             S               *
D BNode1          S               *
D BNode2          S               *
D BNode3          S               *
D uriWSSE         C                   'http://docs.oasis-open.org/wss-
D                                     /2004/01/oasis-200401-wss-wssec-
D                                     urity-secext-1.0.xsd'
```

```
D uriWSU          C                   'http://docs.oasis-open.org/wss-
D                                     /2004/01/oasis-200401-wss-wssec-
D                                     urity-utility-1.0.xsd'
D uriToken        C                   'http://docs.oasis-open.org/wss-
D                                     /2004/01/oasis-200401-wss-usern-
D                                     ame-token-profile-1.0#PasswordT-
D                                     ext'

 /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.
         // 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 17, “Axis C core APIs,” on page 145.

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 21. 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`[13] 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 45.

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                  1n
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                                1n
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.

```
h 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)

 /free

    // 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 146 Axis C API.

Appendix A. Code Listings for myGetQuote Client Application

The myGetQuote sample is a simple stock quote example that is referenced throughout the document. Table 34 on page 205 shows a list of the files.

Table 34: Client files in the samples directory

File name	Description
`GetQuote.wsdl`	GetQuote WSDL file.
`myGetQuote.cpp`	Client implementation code written in CPP.
`myGetQuote.c`	Client implementation code written in C.
`myGetQuote.rpgle`	Client implementation code written in RPG.

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 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:wsdl="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:element>
        </ati:sequence>
      </ati:complexType>
    </ati:element>

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

<wsdl:binding name="StockQuoteSoapBinding" type="impl:StockQuote">
  <wsdlsoap:binding style="document"
       transport="http://schemas.xmlsoap.org/soap/http"/>

  <wsdl:operation name="getQuote">
    <wsdlsoap:operation soapAction=""/>

    <wsdl:input name="getQuoteRequest">
      <wsdlsoap:body use="literal"/>
    </wsdl:input>

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

```
/*********************************************************************/
/*                                                                   */
/*                  IBM Web Services Client for ILE                  */
/*                                                                   */
/*  FILE NAME:      myGetQuote.cpp                                   */
/*                                                                   */
/*  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-INFRINGMENT 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 the WSDL2Ws generated StockQuote.hpp
#include "StockQuote.hpp"

// Include the C++ header file that defines the function cout
#include <iostream>

int main()
{
    try
    {
        // 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";
        StockQuote * pwsStockQuote = new StockQuote( pszEndpoint);

        // If your network requires the use of a proxy, then add the following line of
        // code to configure AxisClient.
        /*
        char *  pszProxyURL = "<ProxyHost>";
        int     iProxyPortNumber = <ProxyPort>;

        pwsStockQuote->setProxy( pszProxyURL, iProxyPortNumber);
        */

        // If you are using handlers, if the WSDL does not identify the SOAP action
        // then you will need to add your SOAP action before calling the web service.
        /*
        char *  pszHandlerName = "Handler";

        pwsStockQuote->setTransportProperty( SOAPACTIONHEADER , pszHandlerName);
        */

        // 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.
        char *  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.
        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
        // is returned.

        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-INFRINGMENT 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
    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-INFRINGMENT 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                             *
     *                                                                   *
     *  5722-SS1, 5761-SS1, 5770-SS1                                     *
     *                                                                   *
     *  (c) Copyright IBM Corp. 2010, 2010                               *
     *  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>                                                  *
     *                                                                   *
     ****************************************************************
     *
     /copy StockQuote.rpgleinc

    d OutputText      s             50
    d WsStub          ds                   likeds(This_t)
    d Input           ds                   likeds(xsd_string)
    d Result          ds                   likeds(xsd_float)

     *-------------------------------------------------------------------
     * Web service logic.
     *-------------------------------------------------------------------

     /free
        // Get a Web service stub. The host and port for the endpoint may need
        // to be changed to match host and port of Web service. Or you can pass
        // blanks and endpoint in the WSDL file will be used.
        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.
          dsply  OutputText;

          // 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 35 on page 213 shows a list of the files.

Table 35: Handler files in the samples directory

File name	Description
`client.wsdd`	The WSDD file that defines the client handler.
`myClientHandler.hpp`	Client handler implementation header file.
`myClientHandler.cpp`	Client handler implementation file.
`myClientHandlerFactory.cpp`	Client handler factory implementation.

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 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="myClientHandlereq" 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:

```
 /*****************************************************************/
 /*                                                               */
 /*                  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     */
 /* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND             */
 /* NON-INFRINGMENT 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>                                                  */
 /*                                                                   */
 /*********************************************************************/

#if !defined( _HANDLER_HPP__INCLUDED_)
#define _HANDLER_HPP__INCLUDED_

#include <axis>

AXIS_CPP_NAMESPACE_USE

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();
};

#endif // !defined(_HANDLER_HPP__INCLUDED_)
```

The myClientHandler.cpp File

The following listing is for the myClientHandler.cpp source file:

```
 /*********************************************************************/
 /*                                                                   */
 /*                  IBM Web Services Client for C/C++                */
 /*                                                                   */
 /*  FILE NAME:      myClientHandler.cpp                              */
 /*                                                                   */
 /*  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-INFRINGMENT 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 <axis>
#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-INFRINGMENT 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 hander 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:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

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

Ajax
Asynchronous JavaScript And XML. Ajax provides the ability for client-side code to send data to and retrieve from a server in the background without interfering with the display behavior of the existing page.

ANSI
American National Standard for Information Systems

API
Application Programming Interface

attachment
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.

AXISCPP_DEPLOY
Environment variable that points to the installation directory, referred to as <install_dir> in this documentation.

certificate
A credential used as an identity of proof between the server and client. It consists of a public key and some identifying information that a certificate authority (CA), an entity to sign certificates, has digitally signed. Each public key has an associated private key and the server must prove that it has access to the private key associated with the public key contained within the digital certificate. A self-signed certificate means it is signed by the server itself. If a self-signed certificate is specified to a server, clients might not trust the connection. To obtain a signed certificate from a public CA, you need to generate a Certificate Signing Request (CSR) and send it to the CA. After a certificate is returned, it is imported to your keystore.

client engine
A set of libraries that are made available at runtime to the client stubs.

DLL
Dynamic Link Library

global handler
A handler that is called regardless of the web service or message name.

GSKit
Global Security Kit, IBM's SSL component

handler
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.

HTTP
HyperText Transfer Protocol.

IBM Toolbox for Java
A library of Java classes supporting client/server and Internet programming model to an IBM i system.

IEEE
Institute of Electrical and Electronic Engineers.

JSON
JavaScript Object Notation. Lightweight data-interchange format that is built on a collection of name/value pairs alongside ordered lists of values.

keystore
A storage facility for cryptographic keys and certificates. A private key entry in a keystore file holds a cryptographic private key and a certificate chain for the corresponding public key. A private key entry can be specified to a server when configuring SSL. A trusted certificate entry contains a public key for

a trusted party, normally a CA. A trusted certificate is used to authenticate the signer of certificates provided by a server or client. The keystore types that the Web Administrator for i GUI supports are: JKS, JCEKS, PKCS12, and CMS. Additionally, the Digital Certificate Manager (DCM) *SYSTEM is also supported.

pivot point
The point where a message is either written on to or read from the wire.

post-pivot handler
A handler that works on a response message after it has been received.

pre-pivot handler
A handler that works on a request message that is to be transmitted.

RPC
Remote Procedure Call

secure endpoint URL
Endpoint beginning with https

service handler
A handler that is specific to the web service with which it is associated.

SOAP
Simple Object Access Protocol

SSL
Secure Sockets Layer

SSL tunneling
In SSL tunneling, the client establishes an unsecure connection to the proxy server, and then attempts to tunnel through the proxy server to the content server over a secure connection where encrypted data is passed through the proxy server unaltered.

TCPIP
Transmission Control Protocol/Internet Protocol

WAR file
A file used to distribute a collection of JavaServer Pages, Java Servlets, Java classes, XML files, static web pages, and other resources that together constitute a web application.

wire
All the underlying components that are responsible for physically sending or receiving a message on the web.

WSDD
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.

WSDL
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.

WSDL2Ws
Java tool that converts a WSDL into a set of client stubs that can be called by the client application.

XML
eXtensible Mark-up Language

XML4C
eXtensible Mark-up Language for C/C++

XSD
XML Schema Definition

Index

T

U

W

X