MICRO FOCUS

Orbix Mainframe 6.3.1

Artix Transport User's Guide

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
https://www.microfocus.com

2021-03-18

Contents

List of Figures

List of Tables

Preface

The Artix Transport component of Orbix Mainframe enables existing or new CORBA servers on the mainframe to be exposed as Web services to the network. Specifically, it allows distributed client applications across the Internet to integrate with Orbix COBOL or Orbix PL/I servers running in batch, CICS, or IMS on z/OS, using SOAP over HTTP(S) as the communications protocol. It enables Orbix servers on the mainframe to be exposed as Web services, without the need for any code changes to those applications. It therefore provides a powerful mechanism for the rapid integration of distributed network components, allowing Orbix Mainframe servers to participate fully in the business flow.

Standards compliance

The Artix Transport component complies with the following W3C specifications:

- SOAP 1.1
- HTTP 1.0 and 1.1
- WSDL 1.1

The Artix Transport component complies with the Web Services Interoperability Organization specification, *Basic Profile Version 1.0*.

The Artix Transport component complies with the OASIS *Web Services Security UsernameToken Profile 1.0* specification for credentials checking in SOAP headers.

Interoperability

The Artix Transport has been tested and certified for use with Web consumers developed using the following Web services products:

- Micro Focus Artix 3.x/4.x/5.x
- Microsoft .NET 2.0
- BEA WebLogic 9.2 MP1

Any Web services product that complies with the W3C and WS-I standards mentioned in "Standards compliance" on page ix can interoperate with applications deployed using the Artix Transport.

Audience

This guide is intended for z/OS systems programmers who want to use the Artix Transport component to configure and expose Orbix Mainframe applications on z/OS as Web services to the network. It is assumed that the reader is familiar with CICS and IMS administration and with Orbix Mainframe application development.

Related reading

The Orbix Mainframe library provides details of the following related topics:

- Orbix Mainframe application development in COBOL and PL/I.
- Non-SOAP related configuration details.
- CICS and IMS server adapter usage.
- Security-related issues.

The Orbix Mainframe library is available from:

https://www.microfocus.com/documentation/orbix/

Prerequisites

See the *Orbix Mainframe Installation Guide* for a full list of supported platforms, supported compilers, and other prerequisites to using Orbix Mainframe and the Artix Transport component.

Organization of this guide

This guide includes the following:

Chapter 1, "Introduction"

This chapter provides an introductory overview of the Artix Transport component, Web services in general, HTTP, and SOAP.

Chapter 2, "Getting Started"

This chapter is provided as a means to getting started with the Artix Transport component. It walks you through a simple demonstration that shows how to expose an existing Orbix server on z/OS as a Web service that can be contacted by various different types of clients.

Chapter 3, "Configuration"

This chapter provides the information needed to deploy and configure an existing Orbix Mainframe server so that it can be exposed as a Web service that accepts SOAP/HTTP(S) requests from distributed clients across the Internet. First it provides an overview of the steps involved in deploying an Orbix Mainframe server as a Web service. Then it provides details of the configuration items involved in enabling an Orbix server to accept SOAP/HTTP(S) requests.

Chapter 4, "SOAP Security Considerations"

This chapter provides details of the different security mechanisms supported by the Artix Transport component in terms of how they can be configured and what they involve.

Chapter 5, "Common Tasks"

This chapter provides details of topics that might be of interest to more advanced users of the Artix Transport component. These include a discussion of the different WSDL encoding styles supported and made available by the Artix Transport, and an explanation of how to perform various tasks relating to topics such as event logging and username and password checking in SOAP servers on the mainframe.

Appendix A, "Default Type Mappings"

This appendix provides a listing of the default type mappings that the Artix Transport component supports.

Additional resources

The Knowledge Base contains helpful articles, written by experts, about Orbix Mainframe, and other products:

https://community.microfocus.com/t5/Orbix/ct-p/Orbix

If you need help with Orbix Mainframe or any other products, contact technical support:

https://www.microfocus.com/en-us/support/

Typographical conventions

This guide uses the following typographical conventions:

`Constant width`

Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class.

Constant width paragraphs represent code examples or information a system displays on the screen. For example:

```
#include <stdio.h>
```

Italic

Italic words in normal text represent *emphasis* and *new terms*.

Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:

```
% cd /users/your_name
```

Note: Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS or Windows command prompt.
. . . . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

CHAPTER 1

Introduction

The Artix Transport component of Orbix Mainframe enables existing or new CORBA servers on the mainframe to be exposed as Web services to the network. Specifically, it allows distributed client applications across the Internet to integrate with Orbix COBOL or Orbix PL/I servers running in batch, CICS, or IMS on z/OS, using SOAP over HTTP(S) as the communication protocol. It enables Orbix servers on the mainframe to be exposed as Web services, without the need for any code changes to these applications. It therefore provides a powerful mechanism for the rapid integration of distributed network components, allowing Orbix Mainframe servers to participate fully in the business flow. This chapter provides an introductory overview of the Artix Transport component, Web services in general, HTTP, and SOAP.

In this chapter

This chapter discusses the following topics:

Artix Transport Overview

Overview

This section provides an introductory overview of the Artix Transport component in terms of its purpose and how it works. It discusses the following topics:

What is the Artix Transport component?

The Artix Transport component enables integration of existing and new Orbix Mainframe server applications with other applications in a heterogeneous environment, using SOAP over HTTP(S) as the communications protocol. (See "HTTP Overview" on page 13 and "SOAP Overview" on page 20 for more details of HTTP(S) and SOAP respectively.) It enables Orbix Mainframe servers to be exposed as Web services to Windows and UNIX clients across the Internet or Intranet.

The Artix Transport component is a powerful tool in that it allows you to combine the dynamic features offered by Web services technology with the reliability and scalability offered by Orbix. Combining standards such as SOAP, HTTP, and CORBA, it provides a fast and robust solution to your enterprise computing needs. Additionally, because it involves a simple matter of configuring your Orbix servers to determine whether they accept SOAP or IIOP requests, it provides the added advantage of requiring no modifications to your existing Orbix Mainframe server implementations.

Note: The Artix Transport can not be used to consume Web services. Orbix Mainframe applications can not make client invocations on SOAP endpoints. Orbix Mainframe only supports CORBA/GIOP client invocations using the client adapter for Orbix applications running in IMS and CICS. The same client invocation restriction applies to Orbix COBOL and PL/I applications running in batch.

Graphical overview of batch server invocation

Figure 1 provides a graphical overview of how the Artix Transport component allows an Orbix COBOL or Orbix PL/I server running in batch to be contacted by a Web services client using SOAP over HTTP(S).

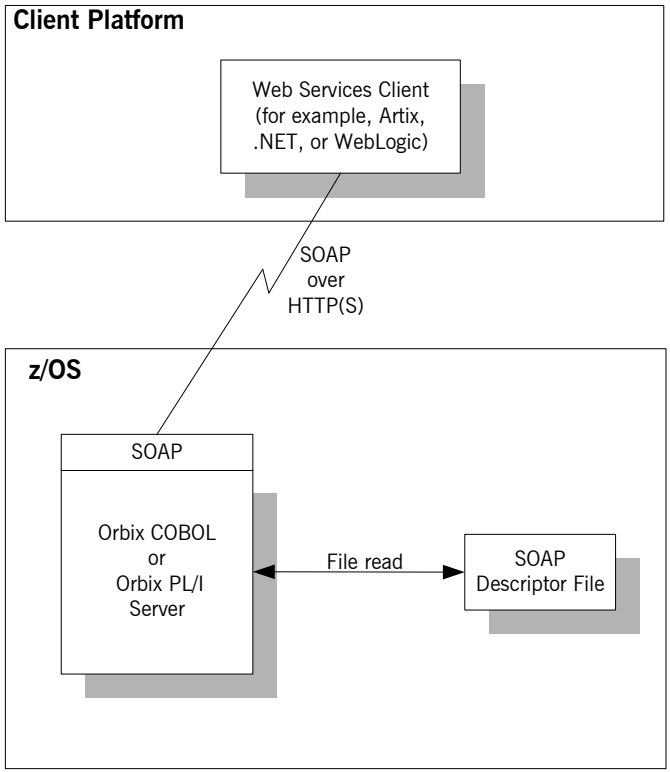

Figure 1: *Exposing Batch Server as Web Service*

Note: For the purposes of illustration, Figure 1 shows a direct persistent batch Orbix Mainframe server being exposed as a Web service.

Graphical overview of CICS or IMS server invocation

Figure 2 provides a graphical overview of how the Artix Transport component allows an Orbix COBOL or Orbix PL/I server running in CICS or IMS to be contacted (using the CICS or IMS server adapter) by a Web services client using SOAP over HTTP(S).

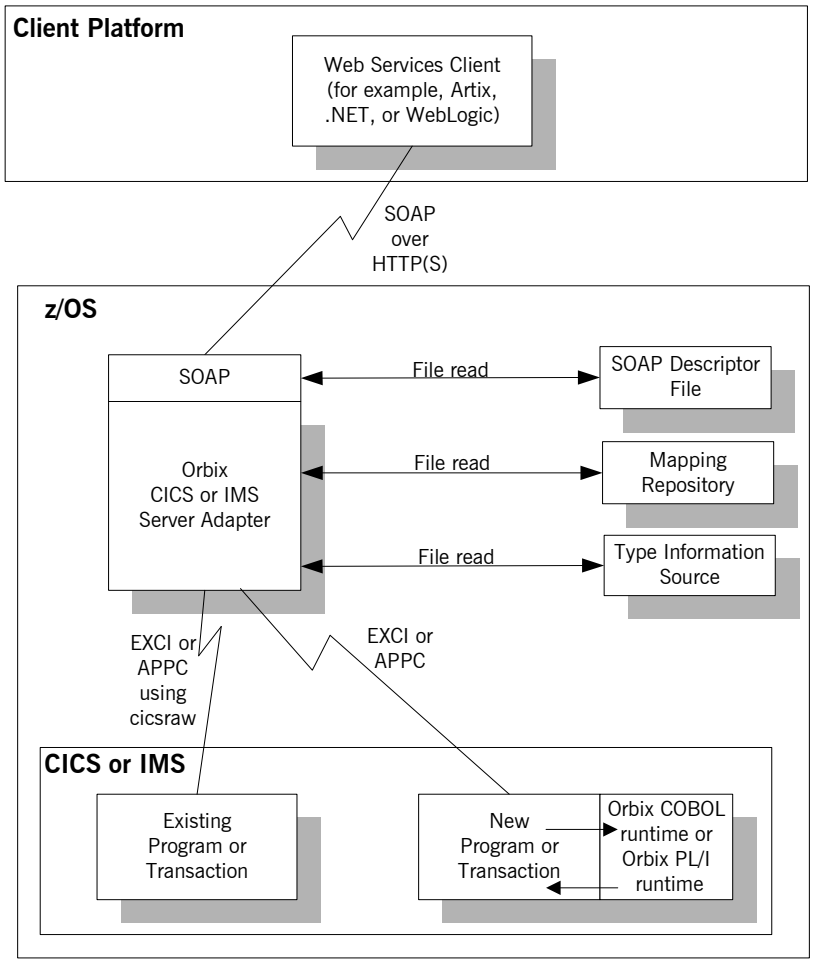

Figure 2: *Exposing CICS or IMS Server as Web Service*

Explanation of graphical overviews

As shown in Figure 1 on page 3 and Figure 2 on page 4, the Web services client uses SOAP over HTTP(S) to contact the server. The server-side SOAP plug-in in turn exposes a SOAP interface to the client. Server-side processing is performed in exactly the same way as per the processing of IIOP client requests, except for one extra feature required by SOAP—a SOAP descriptor file is required by the SOAP plug-in for marshalling purposes.

Note: References to *server* in this case relate to a batch server, a CICS server adapter, or an IMS server adapter.

Data encoding mechanisms

Because the Artix Transport component can enable an Orbix Mainframe application to support Web service client calls over SOAP, it supports various Web service encoding mechanisms for the transfer of data across the Internet. These Web service encoding mechanisms include:

- *Document literal encodings*—This is a document-oriented service that uses literal or schema-defined encoding rules to format request/response messages. Document literal is a reader-makes-right encoding. This means that the receiver is expected to use the schema definitions defined in the WSDL contract to drive encoding/decoding of parameters in request/response messages.

  There are two contract variants for this encoding: *wrapped* and *unwrapped* (or bare). The wrapped convention is generally the accepted best practice approach. The Artix Transport only supports the wrapped variant.
- *RPC literal encodings*—This is an RPC-based service that uses literal or schema-defined encoding rules to format request/response messages. RPC literal (like document literal) is a reader-makes-right encoding. This means that the receiver is expected to use the schema definitions defined in the WSDL contract to drive encoding/decoding of parameters in request/response messages.
- *RPC SOAP encodings*—This is an RPC-based service that uses the SOAP encoding rules to format request and response messages. To work around problems in various client-side SOAP stacks, there are two contract variants available for RPC SOAP encodings. The contract variant for use by .NET clients explicitly encodes sequences of

elements as `soapenc:Array` elements whereas the variant for use by other client SOAP stacks encodes sequences of elements as a schema defined sequence of those elements. See "Web Services Overview" on page 9 for more details of SOAP encoding rules.

Exposing a server as both a CORBA object and Web service

Figure 3 provides a graphical overview of how Orbix Mainframe allows you to expose the same Orbix Mainframe batch server as both a CORBA object and a Web service.

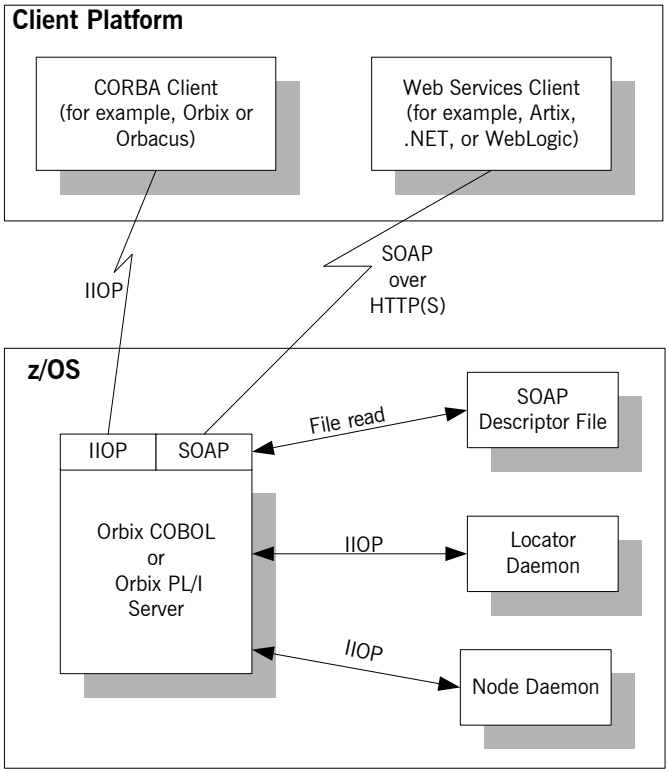

Figure 3: *Exposing Batch Server as Web Service and CORBA Object*

Note: For the purposes of illustration, both Figure 3 and Figure 4 on page 8 show a locator and node daemon being contacted. If a batch server or a CICS or IMS server adapter is configured to run in direct persistence mode, however, the locator and node daemon are not required. See "Configuration" on page 55 for more details about configuration.

Figure 4 provides a graphical overview of how Orbix Mainframe allows an Orbix COBOL or Orbix PL/I server running in CICS or IMS to be exposed (using the CICS or IMS server adapter) as both a CORBA object and a Web service.

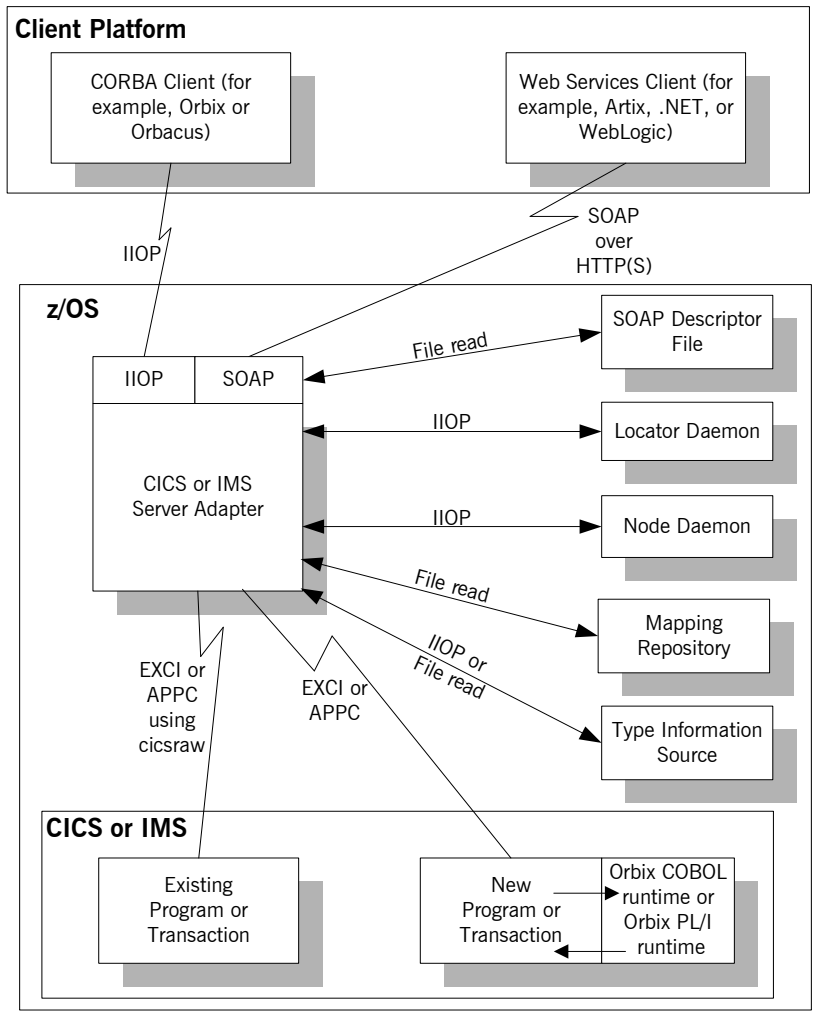

Figure 4: *Exposing CICS or IMS Server as Web Service and CORBA Object*

Web Services Overview

Overview

The Artix Transport component allows you to expose existing and new Orbix Mainframe servers as Web services across the Internet. This section provides an introductory overview of Web services technology in general and the existing standards on which it is based.

In this section

This section discusses the following topics:

Purpose and Advantages

Overview

This subsection provides an introductory overview of the purpose and advantages of Web services technology. It discusses the following topics:

Purpose

Web services technology is a means by which an organisation can expose its data and services across the Internet to other distributed web users and web-connected programs. Typical examples of services range from large-scale storage management or customer relationship management to small-scale furnishing of stock quotes or checking of auction bids.

Web services technology is not tied to the more traditional client-server model of computing, where a web browser client communicates with a web server using some graphical user interface. Instead, Web services allow for the interchange of data and services between organisations through the use of programmatic interfaces (APIs), and at a peer-to-peer level where both parties could equally be supplying services to each other. For this reason, Web services are sometimes referred to as application services, and providers of Web services are known as application service providers.

Advantages

The advantages of Web services technology include:

- It combines existing standards such as XML, SOAP, WSDL, UDDI, and HTML, to offer a standardized, standards-based way of integrating web-based applications.
- It defines data in XML, which is an easy-to-read format for human consumption.
- It offers an integration solution that does not require peers to have extensive knowledge of each other's IT infrastructure behind the security firewall.
- It offers an integration solution that does not care how components are implemented, because all services are described in a standard manner using WSDL, and all communication is standards-based using XML, SOAP and HTTP.

Background Standards

Overview

Web services technology offers a standardized way of integrating web-based applications. It is a popular, standards-based solution that combines various existing standards. This subsection provides an introductory overview of each of these standards in turn. It discusses the following topics:

XML

The *Extensible Markup Language* (XML) is used to define the data being made available by a Web service. See "Background to SOAP" on page 21 for more details about XML.

HTTP

The H*ypertext Transfer Protocol* (HTTP) is used as the transport protocol between distributed peers across the Internet. See "HTTP Overview" on page 13 for more details about HTTP.

SOAP

The *Simple Object Access Protocol* (SOAP) is used for data transfer between distributed peers across the Internet. See "SOAP Overview" on page 20 for more details about SOAP.

WSDL

The *Web Services Description Language* (WSDL) is used to describe the services being made available by a Web service. WSDL is an XML document format that describes a Web service as a collection of communication endpoints that are able to exchange messages. Each endpoint is defined by binding an abstract operation description to a concrete data format and specifying a network protocol and address for this binding. Because the abstract definitions of operations and messages are separated from the concrete data format and network protocol details, the abstract definitions can be reused and recombined to define various endpoints.

UDDI

The *Universal Discovery, Description and Integration* (UDDI) directory is used to list available services. UDDI is an XML-based, distributed registry (or directory) on the World Wide Web that helps to streamline online transactions by enabling companies to list themselves on the Internet, find each other, and make their systems interoperable for e-commerce for the purposes of conducting business. A business can list itself by name, product, location, or the Web services it offers. Comparable to a phone book's yellow and white pages, UDDI therefore acts as the service discovery protocol for Web services.

HTTP Overview

Overview

This section provides an introductory overview of the Hypertext Transfer Protocol (HTTP). It discusses the following topics:

Note: A complete introduction to HTTP is outside the scope of this guide. For more details see the W3C HTTP 1.1 specification at http://www.w3.org/Protocols/rfc2616/rfc2616.html. The Artix Transport component supports both version 1.0 and 1.1 of the W3C HTTP specification.

What is HTTP?

HTTP is used as the transport protocol between distributed peers across the Internet. It is the standard TCP/IP-based transport used for client-server communications on the World Wide Web. Its main function is to establish connections between distributed web browsers (clients) and web servers for the purposes of exchanging files and possibly other information across the Internet.

HTTP is termed an *application protocol*. It defines how messages between web browsers and web servers should be formatted and transmitted. It also defines how web browsers and web servers should behave in response to various commands. HTTP is available on all platforms, and HTTP requests are usually allowed through security firewalls.

Resources and URLs

The files and other information that can be transmitted are collectively known as *resources*. A resource is basically a block of information. Files are the most common example of resources and they can be in various multimedia formats, such as text, graphics, sound, and video. Other examples of resources are server-side script output or dynamically generated query results.

A resource is identifiable by a uniform resource locator (URL). As its name suggests, a URL is the address or location of a resource. A URL typically consists of protocol information followed by host (and optionally port) information followed by the full path to the resource. HTTP is not the only protocol or mechanism for data transfer; other examples include TELNET or the file transfer protocol (FTP). Each of the following is an example of a URL:

- `https://www.microfocus.com/documentation/orbix/orbixmf631/index.html`
- `ftp://ftp.omg.org/pub/docs/formal/01-12-35.pdf`
- `telnet://xyz.com`

In the first of the preceding examples, `https:` denotes that the protocol for data transfer is HTTPS, `//www.microfocus.com` denotes the hostname where the resource resides, and `/documentation/orbix/orbixmf631/index.html` is the full path to the resource (in this case, an HTML text file). The other URLs follow similar patterns.

HTTP transaction processing

When a web user on the client-side requests a resource, either by typing a URL or by clicking on a hypertext link, the client browser builds an HTTP request and opens a TCP/IP socket connection to send the request to the internet protocol (IP) address for the host denoted by the URL for the requested resource. The web server host contains an HTTP daemon that waits for client browser requests and handles them when they arrive. When the HTTP daemon receives a request, the requested resource is then returned to the client browser. The server's response can take the form of HTML pages and possibly other programs in the form of ActiveX controls or Java applets.

Format of HTTP client requests

The following is an example of the typical format of an HTTP client request:

```
GET REQUEST-URI HTTP/1.1
header field: value
header field: value

HTTP request body (if applicable)
```

The preceding code can be explained as follows:

`GET` This is an HTTP method that instructs the server to return the requested resource.

Other HTTP methods might be used here instead. These include:

- `HEAD`—this instructs the server to just return information about the resource (in headers) but not the actual resource itself.
- `POST`—this can be used if you want to send data in the body of the request for subsequent processing by the server.
- `PUT`—this can be used to replace the contents of the target resource with data from the client.

Note: `GET` and `POST` are the most commonly used methods in HTTP client requests.

`REQUEST-URI` This represents the URL of the resource that the client is requesting. The typical format of a URL is:

`http://`*`hostname/path-to-resource`*

For example:

`https://www.microfocus.com/documentation/orbix/orbixmf631/index.html`

`HTTP/1.1` This indicates that the client is using HTTP to transmit the request, and the version of HTTP that the client is using (in this example, `1.1`).

header field	Header information can be included to provide information about the request. In HTTP 1.1, the only mandatory header field is `Host:`, to identify the host where the requested resource resides. In Artix, a number of HTTP client request headers can be configured and sent as part of a client request to a server. See the Artix documentation set for more details.
`HTTP request body`	This can contain user-entered data or files that are being sent to the server for processing. **Note:** This is typically blank in an HTTP request unless the `PUT` or `POST` method is specified.

Format of HTTP server responses

The following is an example of the typical format of an HTTP server response:

```
HTTP/1.1 200 OK
header field: value
header field: value

HTTP response body
```

The preceding code can be explained as follows:

`HTTP/1.1`	This indicates that the server is using HTTP to transmit the response, and the version of HTTP that the server is using (in this example, `1.1`).

`200 OK`

This is status information that indicates whether the request was processed successfully. The 3-digit code is meant to be machine-readable, and the accompanying descriptive text is for human consumption.

Status codes can be broadly described as follows:

- `2`*`xx`*—A status code starting with `2` means the request was processed successfully.
- `3`*`xx`*—A status code starting with `3` means the resource is now located elsewhere and the client should redirect the request to that new location.
- `4`*`xx`*—A status code starting with `4` means that the request has failed because the client has either sent a request in the wrong syntax, or it might have requested a resource that is invalid or that it is not authorized to access.
- `5`*`xx`*—A status code starting with `5` means that the request has failed because the server has experienced internal problems or it does not support the request method specified.

`header field`

Header information can be included to provide information about the response itself or about the information contained in the body of the response.

`HTTP response body`

This is where the requested resource is returned to the client, if the request has been processed successfully. Otherwise, it might contain some explanatory text as to why the request was not processed successfully.

The data in the body of the response can be in a variety of formats, such as HTML or XML text, GIF or JPEG image, and so on.

HTTP properties

The basic properties of HTTP can be summarized as follows:

- Comprehensive addressing—The target resource on which a client request is to be invoked is indicated by means of a universal resource identifier (URI), either as a location (URL) or name (URN). As explained in "Resources and URLs" on page 14, a URL consists of protocol information followed, typically, by host (and optionally port) information followed by the full path to the resource. For example:

  ```
  https://www.microfocus.com/documentation/orbix/orbixmf631/
  index.html
  ```

  See "Resources and URLs" on page 14 for more details.
- Request/response paradigm—A client (web browser) can establish an HTTP connection with a web server by means of a URI, to send a request to that server. See "Format of HTTP client requests" on page 15 for details of the format of a client request message. See "Format of HTTP server responses" on page 16 for details of the format of a server response message.
- Connectionless protocol—HTTP is termed a connectionless protocol because an HTTP connection is typically closed after a single request/response operation. While it is possible for a client to request the server to keep a connection open for subsequent request/response operations, the server is not obliged to keep the connection open. The advantage of closing connections is that it does not incur any overhead in terms of session housekeeping; however, the disadvantage is that it makes it difficult to track user behavior.

  Note: A potential workaround to tracking user behavior is through the use of cookies. A cookie is a string sent by a web server to a web browser and which is then sent back to the web server again each time the browser subsequently contacts that server.

- Stateless protocol—Because HTTP connections are typically closed after each request/response operation, there is no memory or footprint between connections. A workaround to this, in CGI applications, is to encode state information in hidden fields, in the path information, or in URLs in the form returned to the client browser. State can also be

saved in a file, rather than being encoded, as in the typical example of a visitor counter program, where state is identified by means of a unique identifier in the form of a sequential integer.

- Multimedia support—HTTP supports the transfer of various types of data, such as text (for example, HTML or XML files), graphics (for example, GIF or JPEG files), sound, and video. These types are commonly referred to as multipart internet mail extension (MIME) types. A server response can include header information that informs the client of the MIME type of the information being sent by the server.
- Proxies and caches—The communication chain between a client and server might include intermediary programs known as proxies. A proxy can receive client requests, possibly modify the request in some way, and then forward the request along the chain possibly to another proxy or to the target server. Such intermediaries can employ caches to store responses that might be appropriate for subsequent requests. Caches can be shared (public) or private. Specific directives can be established in relation to cache behavior and not all responses might be cacheable.
- Security—Secure HTTP connections that run over the secure sockets layer (SSL) or transport layer security (TLS) protocol can also be established. A secure HTTP connection is referred to as HTTPS and uses port 443 by default. (A non-secure HTTP connection uses port 80 by default.)

SOAP Overview

Overview

This section provides an introductory overview of the simple object access protocol (SOAP) in terms of its purpose, how it evolved, the elements of a SOAP message, and how it handles (encodes) application data types.

In this section

This section discusses the following topics:

Note: A complete introduction to SOAP is outside the scope of this guide. For more details see the W3C SOAP 1.1 specification at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/. The Artix Transport component supports only version 1.1 of the W3C SOAP specification.

Background to SOAP

Overview

This subsection discusses the purpose of SOAP and how it evolved. It discusses the following topics:

What is SOAP?

SOAP is a lightweight, XML-based protocol that is used for client-server communications on the World Wide Web. The primary function of SOAP is to enable access to distributed services and to facilitate the exchange of structured and typed information between peers across the Web.

With the evolution of the Web, and the ever-increasing need to do business more quickly and more proactively across it, there arose a need to have a dynamic, flexible, extensible, but standards-based system of communication between applications across the Internet. SOAP evolved as a solution to this need, by combining existing standards such as extensible markup language (XML) and the hypertext transfer protocol (HTTP).

SOAP is termed a *messaging* protocol. It is a framework for transporting client request and server response messages in the form of XML documents over (usually) the HTTP transport.

XML

XML is a simple form of standard generalized markup language (SGML). The purpose of a markup language is to facilitate preparation of electronic documents, by allowing information to be added to the document text that indicates the logical components of the document or how they are to be formatted. SGML describes the relationship between a document's content and its structure.

XML uses user-defined tags to describe the actual data elements contained within a web page or file. (This is unlike the hypertext markup language (HTML), which can only use a limited set of predefined tags to describe how the contents of a web page or file are to be formatted.) XML tags are

unlimited, because they can be defined at the user's discretion, depending on the data elements that need to be defined. This is why XML is termed *extensible*. XML processors now exist for any common platform or language.

XML and Unicode

XML works on the assumption that all character data belongs to the universal character set (UCS). UCS is more commonly known as *Unicode*. This is a mechanism for setting up binary codes for text or script characters that relate to the principal written languages of the world. Unicode therefore provides a standard means of interchanging, processing, and displaying written texts in diverse languages. See http://www.unicode.org for details.

Because Unicode uses 16 bits to represent a particular character, it can represent more than 65,000 different international text characters. This makes Unicode much more powerful than other text representation formats, such as ASCII (American standard code for information interchange), which only uses 7 bits to represent a particular character and can only represent 128 characters. Unicode uses a conversion method called UTF (universal transformation format) that can convert text to 8–bit or 16–bit Unicode characters. To this effect, there are UTF–8 and UTF–16 encoding formats. All XML processors, regardless of the platform or programming language for which they are implemented, must accept character data encoded using UTF–8 or UTF–16 encoding formats.

HTTP

HTTP is used as the transport protocol between distributed peers across the Internet. It is the standard TCP/IP-based transport used for client-server communications on the World Wide Web. Its main function is to establish connections between distributed web browsers (clients) and web servers for exchanging files and possibly other information across the Internet. HTTP is available on all platforms, and HTTP requests are usually allowed through security firewalls. See "HTTP Overview" on page 13 for more details.

Given the dynamic features of XML and HTTP, SOAP has therefore become regarded as the optimum tool for enabling communication between distributed, heterogeneous applications over the Internet.

Note: Although most implementations of SOAP are HTTP-based, SOAP can be used with any transport that supports transmission of XML data. Depending on the particular transport in use, SOAP can also be implemented to support different types of message-exchange patterns, such as one-way or request-response.

SOAP specification

SOAP is a framework for transporting client request and server response messages in the form of XML documents over HTTP or some other transport. The W3C SOAP specification at http://www.w3.org/TR/SOAP/ defines the standards for SOAP in relation to:

- Format and components of SOAP messages.
- SOAP usage with HTTP.
- SOAP encoding rules for application-defined data types.
- SOAP standards for representing remote procedure calls (RPCs) and responses.

"SOAP Messages" on page 24 briefly discusses the format and components of SOAP messages, and their use with HTTP. "SOAP Encoding of Data Types" on page 30 briefly discusses how data types are handled in SOAP. Again, a complete introduction to these topics is outside the scope of this guide, and you should see the W3C SOAP 1.1 specification at http://www.w3.org/TR/SOAP/ for full details.

SOAP Messages

Overview

This subsection uses an example of a simple client-server application to outline the typical format of a SOAP request and response message. It discusses the following topics:

Example overview

The distributed application in this example involves a client that invokes a `GetStudentGrade` method on a target server. The client passes a student code and subject name, both of type `string`, as input parameters to the method request. On processing the request, the server returns the grade achieved by that student for that subject—the grade is of type `int`. The following example shows the logical definition of this application in a WSDL contract:

Example 1: *Example of logical definition in WSDL*

```
…
<message name="GetStudentGrade">
    <part name="StudentCode" type="xsd:string"/>
    <part name="Subject" type="xsd:string"/>
</message>
<message name="GetStudentGradeResponse">
    <part name="Grade" type="xsd:int"/>
</message>
<portType name="StudentPortType">
    <operation name="GetStudentGrade">
        <input message="tns:GetStudentGrade" name="GetStudentGrade"/>
        <output message="tns:GetStudentGradeResponse" name="GetStudentGradeResponse"/>
    </operation>
</portType>
…
```

Example of SOAP request message

Example 2 shows an example of the format of a typical SOAP request message, based on Example 1 on page 24 (in this case, the client has passed student code `815637` and subject `History` as input parameters):

Example 2: *Example of a SOAP Request Message*

```
1  POST /StockQuote HTTP/1.1
   Host: www.stockquoteserver.com
   Content-Type: text/xml; charset="utf-8"
   Content-Length: nnnn
   SOAPAction: "Some-URI"

   <?xml version="1.0" encoding='UTF-8'?>
2  <SOAP-ENV:Envelope
       xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
       SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
         encoding/"/>
3        <SOAP-ENV:Body>
             <m:GetStudentGrade xmlns:m="Some-URI">
                 <StudentCode>815637</StudentCode>
                 <Subject>History</Subject>
             </m:GetStudentGrade>
         </SOAP-ENV:Body>
   </SOAP-ENV:Envelope>
```

Explanation of SOAP request message

Example 2 on page 25 can be explained as follows:

1. The first five lines represent HTTP header information (in this example, the SOAP request is running over HTTP). When a SOAP request is running over HTTP, the HTTP method must be set to `POST`, the HTTP `Content-Type` header must be set to `text/xml`, and a `SOAPAction` HTTP header should also be included that specifies a URI indicating what is being requested. (However, the `SOAPAction` field can be left blank, in which case the URI specified in the first couple of lines is taken to indicate the intent of the request instead.)

   Note: See "Using the HTTP Plug-in" chapter for more details of the format of HTTP request headers.

2. The SOAP Envelope is the top-level element and is mandatory in every SOAP message. It defines a framework for describing what is in the message and how to process it.
3. The SOAP Body element is mandatory in every SOAP message. It holds the actual message data in sub-elements called body entries. Each body entry relates to a particular data type and must be encoded as an independent element. Body entries can contain attributes called `encodingStyle`, `id`, and `href` (see "SOAP Encoding of Data Types" on page 30 for more details of these).

   In Example 2 on page 25, the SOAP Body contains two body entries, `StudentCode` and `Subject`, within a wrapper element that corresponds to the `GetStudentGrade` operation. The two body entries in this case correspond to the two input parameters for the `GetStudentGrade` operation.

Example of SOAP response message

Example 3 shows an example of the format of a typical SOAP response message, based on Example 1 on page 24 (in this case, the server has returned grade `A`):

Example 3: *Example of a SOAP Response Message*

```
1  HTTP/1.1 200 OK
   Content-Type: text/xml; charset="utf-8"
   Content-Length: nnnn

   <?xml version="1.0" encoding='UTF-8'?>
2  <SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
       encoding/"/>
3          <SOAP-ENV:Body>
               <m:GetStudentGradeResponse xmlns:m="Some-URI">
                       <Grade>A</Grade>
               </m:GetStudentGradeResponse>
           </SOAP-ENV:Body>
   </SOAP-ENV:Envelope>
```

Explanation of SOAP response message

Example 3 can be explained as follows:

1. The first three lines represent HTTP header information (in this example, the SOAP response is running over HTTP). See "Using the HTTP Plug-in" chapter for more details of the format of HTTP response headers.
2. The explanation of the SOAP Envelope element is the same as in "Explanation of SOAP request message" on page 25.
3. The explanation of the SOAP Body element is the same as in "Explanation of SOAP request message" on page 25, except in this case the SOAP Body contains one body entry, `Grade`, within a wrapper element that corresponds to the server response part of the `GetStudentGrade` operation. The body entry in this case corresponds to the output parameter returned by the server in response to the client request (that is, the grade for the student and subject combination specified by the client).

Example of SOAP response with fault

If an error occurs during the processing of a SOAP request, the server can handle and report the error within the SOAP Body of the response. Example 4 shows an example of the format of a typical SOAP response message indicating an error.

Example 4: *Example of SOAP Response with Error Information*

```
1  HTTP/1.1 500 Internal Server Error
   Content-Type: text/xml; charset="utf-8"
   Content-Length: nnnn

   <SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
      <SOAP-ENV:Body>
2        <SOAP-ENV:Fault>
             <faultcode>SOAP-ENV:Server</faultcode>
             <faultstring>Server Error</faultstring>
             <detail>
                 <e:myfaultdetails xmlns:e="Some-URI">
                     <message>
                         Application did not work
                     </message>
                     <errorcode>
```

Example 4: *Example of SOAP Response with Error Information*

```
                    1001
                  </errorcode>
               </e:myfaultdetails>
            </detail>
         </SOAP-ENV:Fault>
      </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Explanation of SOAP response with fault

Example 4 on page 27 can be explained as follows:

1. The first three lines represent HTTP header information (in this example, the SOAP response is running over HTTP). See "Using the HTTP Plug-in" chapter for more details of the format of HTTP response headers.
2. Errors are reported within a SOAP Fault element within the SOAP Body. In this case, the SOAP Body must not contain any other elements. Only one SOAP Fault element can be defined in any SOAP message. SOAP Fault in turn defines the following four sub-elements:

`faultcode` This describes the error. The default faultcode values defined by the W3C SOAP specification are:

- `VersionMismatch`—This means the SOAP Envelope was associated with an invalid namespace (that is, a namespace other than `http://schemas.xmlsoap.org/soap/envelope/`).
- `MustUnderstand`—This means a header element that needed to be processed was not processed correctly.
- `Client`—This means the message was not properly formed or did not contain appopriate information to be successfully processed.
- `Server`—This means the message could not be processed, but not due to message contents.

`faultstring` This provides a human-readable explanation of the fault.

`faultactor` This indicates where the fault originated along the message path. This element is mandatory for an intermediary proxy application along the message path, but it is optional for the ultimate target server.

Example 4 on page 27 is an example of an error being reported by the ultimate target server, and it omits a `faultactor` attribute.

`detail` This in turn contains sub-elements, called *detail elements*, that hold application-specific error information when the fault is due to unsuccessful processing of the SOAP Body.

SOAP Encoding of Data Types

Overview

This subsection provides an overview of the concepts of SOAP encoding. It discusses the following topics:

What is encoding?

Encoding is the process of converting application-defined data to binary form for transfer across a network. *Decoding* is the process of converting binary data back to an application-defined format. XML encoding and decoding rules, such as UTF-8 or UTF-16, define how data is to be converted between application-defined and binary form.

SOAP encoding rules define how application data types are to be structured in an XML document before being converted to binary. The overall process of encoding, data transfer, and subsequent decoding is termed *serialization*.

Role of SOAP encoding

XML uses the UTF-8 and UTF-16 encoding formats to convert data to binary form. As explained in "Background to SOAP" on page 21, all XML processors (regardless of platform or programming language) must accept character data encoded using UTF-8 or UTF-16 formats.

Problems can arise, however, when converting data to and from binary, if the data is represented differently by different applications. For example, some systems might have an integer as a 32-bit value, while others might have it as a 16-bit value. Such disparities can lead to data corruption during the data conversion process.

To avoid potential data corruption due to differences between source and target systems, SOAP encoding and decoding rules are used as a stepping stone between the expression of data types in a particular programming language and the XML UTF-8 or UTF-16 encoding or decoding rules used to convert those data types to and from binary. (See Figure 5 on page 31 for

more details.) SOAP encoding rules, therefore, define the elements and data types that are designed to support serialization of data between disparate systems.

As shown in Figure 5, all data transferred as part of a SOAP payload is marshalled across the network as UTF-encoded binary strings.

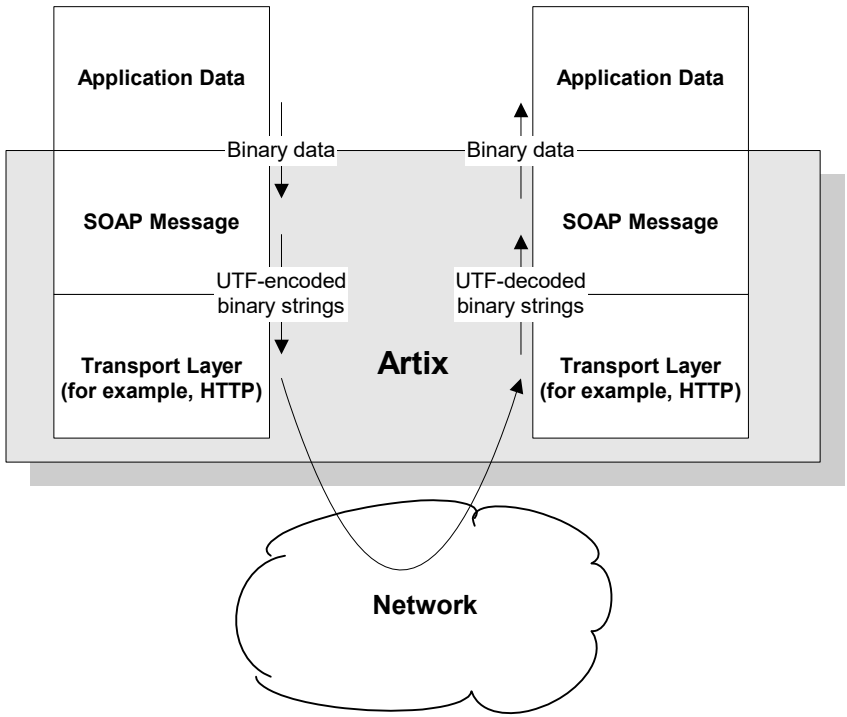

Figure 5: *Overview of Role of SOAP Encoding and Decoding*

SOAP encoding styles

A standard XML schema for SOAP encoding has been developed by the W3C and is located at http://schemas/xmlsoap/org/soap/encoding/. This W3C SOAP encoding schema uses the following namespace declaration:

```
xmlns:SOAP-ENC="http://schemas.xmlsoap/org/soap/encoding/"
```

It is recommended, but not mandatory, that a SOAP implementation adheres to the encoding style based on the W3C SOAP encoding schema. The W3C SOAP specification states that a company can use alternative encoding styles if it wants. To this effect, an `encodingStyle` attribute can be specified for any element within a SOAP message, to indicate the encoding rules that apply to that particular element.

An `encodingStyle` attribute can take one or more URIs as its value, with each URI denoting the location of a particular set of encoding rules. If specifying a list of URIs, each URI should be separated by a space. A list should also be ordered so that the URI relating to the most restrictive set of encoding rules is specified first, and the URI relating to the least restrictive set of encoding rules is specified last.

Encoding simple types

The W3C SOAP specification states that SOAP encodings can support all the simple types that are specified in the W3C *XML Schema Part 2: Datatypes* specification at `http://www.w3.org/TR/SOAP/#XMLS2`. In other words, a SOAP encoding should support any simple type that can be used in XML schema definition language.

The W3C SOAP encoding schema defines elements whose names correspond to each of the simple types defined in the W3C *XML Schema Part 2: Datatypes* specification. Among the simple types supported are integers, floats, doubles, booleans, and so on. Other types considered simple for the purposes of a SOAP encoding are strings, enumerations, and arrays of bytes.

In a SOAP encoding, each data value must be specified within an element. The type of a particular value can be denoted by the element name that encompasses it, provided that element name has been defined in the

encoding schema as a derived type. The following is an example of a schema fragment that defines a series of elements (for example, an element called `age` of type `int`, an element called `height` of type `float`, and so on):

```
<element name="age" type="int"/>
<element name="height" type="float"/>
<element name="displacement" type="negativeInteger"/>
<element name="color">
   <simpleType base="xsd:string">
      <enumeration value="Blue"/>
      <enumeration value="Brown"/>
   </simpleType>
</element>
```

The following is an example of how the elements defined in the preceding sample schema might then be used in a SOAP encoding:

```
<age>34</age>
<height>6.0</height>
<displacement>-350</displacement>
<color>Brown</color>
```

If an element name in a SOAP encoding has not been defined as a derived type in an encoding schema (for example, the element name relating to a member of an array), that element must include an `xsi:type` attribute in the SOAP encoding to indicate the data type. See "Encoding complex array types" on page 36 for an example of this.

Encoding complex struct types

The W3C SOAP specification defines two complex data types—structs and arrays. A struct is a compound value whose members are each distinguished by a unique name (also known as that member's *accessor*).

The following is an example of a schema fragment that defines elements called `Book`, `Author`, and `Address` respectively, each of which is a structure containing a series of types:

```
<element name="Book">
<complexType>
    <sequence>
    <element name="title" type="xsd:string"/>
    <element name="author" type="tns:Author"/>
    </sequence>
</complexType>
</e:Book>
<element name="Author">
<complexType>
    <sequence>
    <element name="name" type="xsd:string"/>
    <element name="address" type="tns:Address"/>
    </sequence>
</complexType>
</e:Author>
<element name="Address">
<complexType>
    <sequence>
    <element name="street" type="xsd:string"/>
    <element name="city" type="xsd:string"/>
    <element name="country" type="xsd:string"/>
    </sequence>
</complexType>
</e:Address>
```

The following is an example of how the preceding schema definition could be subsequently used in a SOAP encoding (the following example shows embedded single-reference values for the author and address):

```
<e:Book>
    <title>Great Expectations</title>
    <author>
        <name>Charles Dickens</name>
        <address>
            <street>Whitechurch Road</street>
            <city>London</city>
            <country>England</country>
        </address>
     </author>
</e:Book>
```

In some cases an element might potentially contain more than one possible value. For example, if there was another book also called Great Expectations, written by some other author, there could be potentially more than one possible value for the author and address in the preceding example. When an element can contain more than one possible value it is termed *multireference*. In this case, an `id` attribute must be used to identify a multireference element, and a `href` attribute can be used to reference that element. For example, the `href` attribute of the `<author>` element in the following example refers to the `id` attribute of the multireference `<Person>` element. Similarly, the `href` attribute of the `<address>` element refers to the `id` attribute of the multireference `<Home>` element (this is assuming the author in question has more than one home).

```
<e:Book>
    <title>Great Expectations</title>
    <author href="#Person-1"/>
</e:Book>
<e:Person id="Person-1">
    <name>Charles Dickens</name>
    <address> href="Home-1"/>
</e:Person>
<e:Home id="Home-1"/>
    <street>Whitechurch Road</street>
    <city>London</city>
    <country>England</country>
</e:Home>
```

Encoding complex array types

The W3C SOAP specification defines two complex data types—structs and arrays. An array is a compound value whose member values are distinguished by means of ordinal position within the array. An array in SOAP is of type `SOAP-ENC:Array` or a type derived from that.

The following is an example (taken from the W3C SOAP specification) of a schema fragment that defines an element called `myFavoriteNumbers` that is of type `SOAP-ENC:Array`:

```
<element name="myFavoriteNumbers"
    type="SOAP-ENC:Array"/>
```

The following is an example (taken from the W3C SOAP specification) of how the array defined in the preceding sample schema could be subsequently used in a SOAP encoding:

```
<myFavoriteNumbers SOAP-ENC:arrayType="xsd:int[2]">
    <number>3</number>
    <number>4</number>
</myFavoriteNumbers>>
```

The preceding example shows an array of two integers, with both members of the array called `number` (this is unlike the members of a struct which must all have unique names). The members of a SOAP array do not have to be all of the same type. The following is an example of the SOAP encoding for an array where an `xsi:type` attribute is used to specify the type of each member of the array:

Note: As explained in "Encoding simple types" on page 32, if the type of a value is not identifiable from the element name (or accessor) corresponding to that value, an `xsi:type` attribute must be used in the SOAP encoding.

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:ur-type[4]">
  <thing xsi:type="xsd:int">98765</thing>
  <thing xsi:type="xsd:decimal">3.857</thing>
  <thing xsi:type="xsd:string">The cat sat on the mat</thing>
  <thing
   xsi:type="xsd:uriReference">http://www.microfocus.com</thing>
</SOAP-ENC:Array>
```

SOAP encoding rules also support:

- Arrays of complex structs or other arrays.
- Multi-dimensional arrays.
- Partially transmitted arrays.
- Sparse arrays.

See the W3C SOAP specification for more details of the encoding guidelines for arrays.

CHAPTER 2

Getting Started

This chapter is provided as a means to getting started with the Artix Transport component. It walks through a simple demonstration showing how to expose an existing Orbix server on z/OS as a Web service that can be consumed using the Web services client platform of your choice.

In this chapter

This chapter discusses the following topics:

Note: The instructions provided in this chapter describe how to deploy the non-secure version of the `simple` demonstration.

Overview of Steps

Overview

This section provides an overview of the steps involved in this demonstration walkthrough.

Summary

The steps to run the demonstration can be summarized as follows:

- Generate the SOAP descriptor file
- Build and run the server
- Develop a client
- Build and run the client

The rest of this chapter describes each of these steps.

Demonstration versions

Your installation of the Artix Transport component includes a `simple` demonstration that illustrates how to expose an existing Orbix Mainframe server as a Web service. The Orbix Mainframe `simple` demonstration server is used for this purpose. Several variants of the `simple` demonstration are supplied with Orbix Mainframe, corresponding to the different languages and environments supported. For the purposes of demonstrating the Artix Transport component, any of the following COBOL or PL/I variants of the `simple` demonstration can be used:

- COBOL batch server
- PL/I batch server
- COBOL CICS server (using the CICS server adapter)
- PL/I CICS server (using the CICS server adapter)
- COBOL IMS server (using the IMS server adapter)
- PL/I IMS server (using the IMS server adapter)

Sample code for the Web consumer of this service is not provided. As mentioned in the preface, any Web services development product that supports the same WS-I standards as the Artix Transport can be used.

For sample purposes such as this, a Web services testing tool such as SOAPScope is also recommended.

Generating the SOAP Descriptor File

Overview

To expose an Orbix Mainframe resource as a Web service, a SOAP descriptor file is required. As shown in Figure 1 on page 3 and Figure 2 on page 4, a SOAP descriptor file is used by the SOAP plug-in on the server side for the purposes of marshalling SOAP requests. This section describes how to generate the SOAP descriptor file for the demonstration(s).

Sample JCL

Run the following supplied JCL to generate the SOAP descriptor file (where *orbixhlq* represents the high-level qualifier for your installation):

```
orbixhlq.DEMO.ARTIX.BLD.JCLLIB(SIMPLESI)
```

For the purposes of this demonstration, the generated file is saved by default to *orbixhlq*.DEMO.TYPEINFO(SIMPLEB).

Note: The location that the SOAP plug-in uses to retrieve SOAP descriptor files is specified using the `plugins:soap:type_info:source` configuration item.

Building and Running the Server

Overview

This section describes how to build and run the various types of simple demonstration server that are supplied. It discusses the following topics:

- "Steps for simple batch COBOL server" on page 42.
- "Steps for simple batch PL/I server" on page 43.
- "Steps for simple CICS COBOL server" on page 44.
- "Steps for simple CICS PL/I server" on page 46.
- "Steps for simple IMS COBOL server" on page 48.
- "Steps for simple IMS PL/I server" on page 50.

Follow the instructions that are relevant to the type of demonstration server you want to use.

Steps for simple batch COBOL server

The steps to build and run the simple batch COBOL server are:

Note: The source code for the demonstration is already supplied in the *orbixhlq*.DEMO.CBL.SRC PDS, so the options to generate it are disabled in the SIMPLIDL JCL, to avoid overwriting the shipped code.

1. Run the Orbix IDL compiler by submitting *orbixhlq*.DEMO.CBL.BLD.JCLLIB(SIMPLIDL). This takes as input the sample IDL in *orbixhlq*.DEMO.IDL(SIMPLE), and subsequently generates the relevant COBOL copybooks, which are stored in the *orbixhlq*.DEMO.CBL.COPYLIB PDS.
2. Build the server executable by submitting *orbixhlq*.DEMO.CBL.BLD.JCLLIB(SIMPLESB). This creates the server load module, which is automatically stored in the *orbixhlq*.DEMO.CBL.LOADLIB PDS.
3. Run the server by submitting *orbixhlq*.DEMO.CBL.RUN.JCLLIB(SIMPLESV).

4. To ensure the server is running successfully, open a web browser and enter the following URL (where *remotehost* represents the z/OS TCP/IP hostname and 5105 is the port that the server uses to listen for incoming requests):

```
http://remotehost:5105/ionasoap/Simple_SimpleObject?wsdl=doc_literal
```

If the server returns the WSDL for the Web service, this means it is successfully up and running.

Steps for simple batch PL/I server

The steps to build and run the simple batch PL/I server are:

Note: The source code for the demonstration is already supplied in the *orbixhlq*.DEMO.PLI.SRC PDS, so the options to generate it are disabled in the SIMPLIDL JCL, to avoid overwriting the shipped code.

1. Run the Orbix IDL compiler by submitting *orbixhlq*.DEMO.PLI.BLD.JCLLIB(SIMPLIDL). This takes as input the sample IDL in *orbixhlq*.DEMO.IDL(SIMPLE), and subsequently generates the relevant PL/I include members, which are stored in the *orbixhlq*.DEMO.PLI.PLINCL PDS.
2. Build the server executable by submitting *orbixhlq*.DEMO.PLI.BLD.JCLLIB(SIMPLESB). This creates the server load module, which is automatically stored in the *orbixhlq*.DEMO.PLI.LOADLIB PDS.
3. Run the server by submitting *orbixhlq*.DEMO.PLI.RUN.JCLLIB(SIMPLESV).
4. To ensure the server is running successfully, open a web browser and enter the following URL (where *remotehost* represents the z/OS TCP/IP hostname and 5105 is the port that the server uses to listen for incoming requests):

```
http://remotehost:5105/ionasoap/Simple_SimpleObject?wsdl=doc_literal
```

If the server returns the WSDL for the Web service, this means it is successfully up and running.

Steps for simple CICS COBOL server

The steps to build and run the simple CICS COBOL server are:

Note: The server implementation code is already supplied in *orbixhlq*.DEMO.CICS.CBL.SRC(SIMPLES), so the option to generate it is disabled in the SIMPLIDL JCL, to avoid overwriting the shipped code.

1. Run the Orbix IDL compiler by submitting *orbixhlq*.DEMO.CICS.CBL.BLD.JCLLIB(SIMPLIDL). This takes as input the sample IDL in *orbixhlq*.DEMO.IDL(SIMPLE), and subsequently generates:
   - The relevant COBOL copybooks for the CICS server, which are stored in the *orbixhlq*.DEMO.CICS.CBL.COPYLIB PDS.
   - The source code for the CICS server mainline program, which is stored in *orbixhlq*.DEMO.CICS.CBL.SRC(SIMPLESV).
   - The CICS server adapter mapping file, which is stored in the *orbixhlq*.DEMO.CICS.MFAMAP PDS.

   Note: If you are using the CICS APPC plug-in, first edit the SIMPLIDL JCL to ensure that the following two lines appear as follows (that is, ensure the first line is commented out and the second one is not):

   ```
   //*   IDLPARM='-cobol:-S:-TCICS -mfa:-tSIMPLESV'
   //    IDLPARM='-cobol:-S:-TCICS -mfa:-tSMSV'
   ```

2. Build the server executable by submitting *orbixhlq*.DEMO.CICS.CBL.BLD.JCLLIB(SIMPLESB). This creates the CICS server load module, which is stored in the *orbixhlq*.DEMO.CICS.CBL.LOADLIB PDS.
3. Define a transaction definition for the server, to allow it to run in CICS. See *orbixhlq*.JCLLIB(ORBIXCSD) for an example of the transaction definition for the supplied demonstration.
4. Provide the server load module to the CICS region that is to run the transaction, by adding *orbixhlq*.DEMO.CICS.CBL.LOADLIB and *orbixhlq*.MFA.LOADLIB to the DFHRPL for that CICS region.

5. In *orbixhlq*.DOMAINS(FILEDOMA), ensure that the plugins:cicsa:mapping_file configuration item (within the iona_services.cicsa scope) specifies the full path to the mapping file that contains the relevant mapping entries. As explained in point 1, the sample mapping entries for the demonstration are generated in *orbixhlq*.DEMO.CICS.MFAMAP(SIMPLEA) by default.
6. If the CICS server adapter is already running, it must be refreshed to pick up the mapping file updates. To do this, follow the instructions in *orbixhlq*.JCLLIB(ORXADMIN) for performing an itadmin mfa reload.

   If the CICS server adapter is not already running, start it as described in the *CICS Adapters Administrator's Guide*.

   Note: You can only use the itadmin mfa reload and itadmin mfa refresh commands if you are licensed to use Orbix Mainframe (that is, if you have a valid IIOP license). Otherwise, you must manually stop and restart the adapter to perform the equivalent of a reload and refresh.

7. To ensure the server is running successfully, open a web browser and enter the following URL (where *remotehost* represents the z/OS TCP/IP hostname and 5105 is the port that the server uses to listen for incoming requests):

```
http://remotehost:5051/ionasoap/Simple_SimpleObject?wsdl=doc_literal
```

   If the server returns the WSDL for the Web service, this means it is successfully up and running.

Steps for simple CICS PL/I server

The steps to build and run the simple CICS PL/I server are:

Note: The server implementation code is already supplied in *orbixhlq*.DEMO.CICS.PLI.SRC(SIMPLEI), so the option to generate it is disabled in the SIMPLIDL JCL, to avoid overwriting the shipped code.

1. Run the Orbix IDL compiler by submitting *orbixhlq*.DEMO.CICS.PLI.BLD.JCLLIB(SIMPLIDL). This takes as input the sample IDL in *orbixhlq*.DEMO.IDL(SIMPLE), and subsequently generates:
   - The relevant PL/I include files for the CICS server, which are stored in the *orbixhlq*.DEMO.CICS.PLI.PLINCL PDS.
   - The source code for the CICS server mainline program, which is stored in *orbixhlq*.DEMO.CICS.PLI.SRC(SIMPLEV).
   - The CICS server adapter mapping file, which is stored in the *orbixhlq*.DEMO.CICS.MFAMAP PDS.
2. Build the server executable by submitting *orbixhlq*.DEMO.CICS.PLI.BLD.JCLLIB(SIMPLESB). This creates the CICS server load module, which is stored in the *orbixhlq*.DEMO.CICS.PLI.LOADLIB PDS.
3. Define a transaction definition for the server, to allow it to run in CICS. See *orbixhlq*.JCLLIB(ORBIXCSD) for an example of the transaction definition for the supplied demonstration.
4. Provide the server load module to the CICS region that is to run the transaction, by adding *orbixhlq*.DEMO.CICS.PLI.LOADLIB and *orbixhlq*.MFA.LOADLIB to the DFHRPL for that CICS region.
5. In *orbixhlq*.DOMAINS(FILEDOMA), ensure that the plugins:cicsa:mapping_file configuration item (within the iona_services.cicsa scope) specifies the full path to the mapping file that contains the relevant mapping entries. As explained in point 1, the sample mapping entries for the demonstration are generated in *orbixhlq*.DEMO.CICS.MFAMAP(SIMPLEA) by default.

6. If the CICS server adapter is already running, it must be refreshed to pick up the mapping file updates. To do this, follow the instructions in *orbixhlq*.JCLLIB(ORXADMIN) for performing an `itadmin mfa reload`.

   If the CICS server adapter is not already running, start it as described in the *CICS Adapters Administrator's Guide*.

   Note: You can only use the `itadmin mfa reload` and `itadmin mfa refresh` commands if you are licensed to use Orbix Mainframe (that is, if you have a valid IIOP license). Otherwise, you must manually stop and restart the adapter to perform the equivalent of a reload and refresh.

7. To ensure the server is running successfully, open a web browser and enter the following URL (where *remotehost* represents the z/OS TCP/IP hostname and 5105 is the port that the server uses to listen for incoming requests):

```
http://remotehost:5051/ionasoap/Simple_SimpleObject?wsdl=doc_literal
```

   If the server returns the WSDL for the Web service, this means it is successfully up and running.

Steps for simple IMS COBOL server

The steps to build and run the simple IMS COBOL server are:

Note: The IMS server implementation code is already supplied in *orbixhlq*.DEMO.IMS.CBL.SRC(SIMPLES), so the option to generate it is disabled in the SIMPLIDL JCL, to avoid overwriting the shipped code.

1. Run the Orbix IDL compiler by submitting *orbixhlq*.DEMO.IMS.CBL.BLD.JCLLIB(SIMPLIDL). This takes as input the sample IDL in *orbixhlq*.DEMO.IDL(SIMPLE), and subsequently generates:
   - The relevant COBOL copybooks for the IMS server, which are stored in the *orbixhlq*.DEMO.IMS.CBL.COPYLIB PDS.
   - The source code for the IMS server mainline program, which is stored in *orbixhlq*.DEMO.IMS.CBL.SRC(SIMPLESV).
   - The IMS server adapter mapping file, which is stored in the *orbixhlq*.DEMO.IMS.MFAMAP PDS.
2. Build the server executable by submitting *orbixhlq*.DEMO.IMS.CBL.BLD.JCLLIB(SIMPLESB). This creates the IMS server load module, which is stored in the *orbixhlq*.DEMO.IMS.CBL.LOADLIB PDS.
3. If not already done, define a transaction definition for the server in IMS, using an IMS GEN, to allow it to run in IMS. For example, the following transaction definition is already defined for the supplied demonstration:

```
APPLCTN GPSB=SIMPLESV,                          x
      PGMTYPE=(TP,,2),                          x
      SCHDTYP=PARALLEL,LANG=COBOL
TRANSACT CODE=SIMPLESV,
      EDIT=(ULC)                                x
```

4. Provide the server load module to the IMS region that is to run the transaction, by adding *orbixhlq*.DEMO.IMS.CBL.LOADLIB and *orbixhlq*.MFA.LOADLIB to the STEPLIB for that IMS region.
5. In *orbixhlq*.DOMAINS(FILEDOMA), ensure that the plugins:imsa:mapping_file configuration item (within the iona_services.imsa scope) specifies the full path to the mapping file that contains the relevant mapping entries. As explained in point 1, the sample mapping entries for the demonstration are generated in *orbixhlq*.DEMO.IMS.MFAMAP(SIMPLEA) by default.
6. If the IMS server adapter is already running, it must be refreshed to pick up the mapping file updates. To do this, follow the instructions in *orbixhlq*.JCLLIB(ORXADMIN) for performing an itadmin mfa reload.

   If the IMS server adapter is not already running, start it as described in the *IMS Adapters Administrator's Guide*.

   Note: You can only use the itadmin mfa reload and itadmin mfa refresh commands if you are licensed to use Orbix Mainframe (that is, if you have a valid IIOP license). Otherwise, you must manually stop and restart the adapter to perform the equivalent of a reload and refresh.

7. To ensure the server is running successfully, open a web browser and enter the following URL (where *remotehost* represents the z/OS hostname and 5105 is the server port):

```
http://remotehost:5050/ionasoap/Simple_SimpleObject?wsdl=doc_literal
```

   If the server returns the WSDL for the Web service, this means it is successfully up and running.

Steps for simple IMS PL/I server

The steps to build and run the simple IMS PL/I server are:

> **Note:** The IMS server implementation code is already supplied in *orbixhlq*.DEMO.IMS.PLI.SRC(SIMPLEI), so the option to generate it is disabled in the SIMPLIDL JCL, to avoid overwriting the shipped code.

1. Run the Orbix IDL compiler by submitting *orbixhlq*.DEMO.IMS.PLI.BLD.JCLLIB(SIMPLIDL). This takes as input the sample IDL in *orbixhlq*.DEMO.IDL(SIMPLE), and subsequently generates:
   - The relevant PL/I include members for the IMS server, which are stored in the *orbixhlq*.DEMO.IMS.PLI.PLINCL PDS.
   - The source code for the IMS server mainline program, which is stored in *orbixhlq*.DEMO.IMS.PLI.SRC(SIMPLEV).
   - The IMS server adapter mapping file, which is stored in the *orbixhlq*.DEMO.IMS.MFAMAP PDS.
2. Build the server executable by submitting *orbixhlq*.DEMO.IMS.PLI.BLD.JCLLIB(SIMPLESB). This creates the IMS server load module, which is stored in the *orbixhlq*.DEMO.IMS.PLI.LOADLIB PDS.
3. If not already done, define a transaction definition for the server in IMS, using an IMS GEN, to allow it to run in IMS. For example, the following transaction definition is already defined for the supplied demonstration:

```
APPLCTN GPSB=SIMPLESV,                          x
      PGMTYPE=(TP,,2),                          x
      SCHDTYP=PARALLEL,LANG=PL/I
TRANSACT CODE=SIMPLESV,
      EDIT=(ULC)                                x
```

4. Provide the server load module to the IMS region that is to run the transaction, by adding *orbixhlq*.DEMO.IMS.PLI.LOADLIB and *orbixhlq*.MFA.LOADLIB to the STEPLIB for that IMS region.

5. In *orbixhlq*.DOMAINS(FILEDOMA), ensure that the plugins:imsa:mapping_file configuration item (within the iona_services.imsa scope) specifies the full path to the mapping file that contains the relevant mapping entries. As explained in point 1, the sample mapping entries for the demonstration are generated in *orbixhlq*.DEMO.IMS.MFAMAP(SIMPLEA) by default.
6. If the IMS server adapter is already running, it must be refreshed to pick up the mapping file updates. To do this, follow the instructions in *orbixhlq*.JCLLIB(ORXADMIN) for performing an itadmin mfa reload.

   If the IMS server adapter is not already running, start it as described in the *IMS Adapters Administrator's Guide*.

   Note: You can only use the itadmin mfa reload and itadmin mfa refresh commands if you are licensed to use Orbix Mainframe (that is, if you have a valid IIOP license). Otherwise, you must manually stop and restart the adapter to perform the equivalent of a reload and refresh.

7. To ensure the server is running successfully, open a web browser and enter the following URL (where *remotehost* represents the z/OS hostname and 5105 is the server port):

```
http://remotehost:5050/ionasoap/Simple_SimpleObject?wsdl=doc_literal
```

   If the server returns the WSDL for the Web service, this means it is successfully up and running.

Note: If no parameter is specified in the wsdl HTTP query part of the URL, the wrapped document literal (doc_literal) encoding style is assumed by default. For example:
http://remotehost:5050/ionasoap/Simple_SimpleObject?wsdl

Building and Running a Web Consumer

Overview

The previous section outlined how to obtain the WSDL from the deployed Web service running on the mainframe.

A simple SOAP client can now be developed using your Web consumer authoring tool of choice. You can use the HTTP URL directly in your client application, or save the WSDL to a local file on the client-side platform and use a file URL or path name as appropriate.

Sample input

The `call_me()` operation takes no data input arguments. For wrapped document literal, which is the recommended encoding convention, this means that an empty XML element is sent in the SOAP input message body. For example:

```
<soapenv:Body>
   <sim:call_me xmlns:sim="http://artixmf.iona.com/Simple_SimpleObject" />
</soapenv:Body>
```

Expected Output

The `call_me()` operation returns no output data. For wrapped document literal, this means that an empty XML element is sent in the SOAP output message body. For example:

```
<env:Body>
  <call_meResponse xmlns="http://artixmf.iona.com/Simple_SimpleObject">
  </call_meResponse>
</env:Body>
```

Further Information Sources

Overview

This section provides a pointer to further sources of information that you might need to perform various tasks. It discusses the following topics:

- "Configuration".
- "Adapter usage".
- "Server development".
- "Security".

Note: All user manuals referred to in this section are in the Orbix Mainframe 6.3 library unless otherwise specified.

Configuration

For details about configuring an Orbix Mainframe server to use the Artix Transport component, and the steps to deploy an Orbix Mainframe server as a Web service, see "Configuration" on page 55 of this guide.

For other general configuration details relating to batch servers, see the *Orbix Configuration Reference*.

For other general configuration details relating to CICS and IMS servers, see the *CICS Adapters Administrator's Guide* and *IMS Adapters Administrator's Guide*.

Adapter usage

For details about general CICS or IMS server adapter usage, see the *CICS Adapters Administrator's Guide* and *IMS Adapters Administrator's Guide*.

Server development

For details about developing Orbix Mainframe servers in COBOL or PL/I, see the *COBOL Programmer's Guide and Reference* and *PL/I Programmer's Guide and Reference*.

Security

For details about general security-related issues, see the *Mainframe Security Guide*.

For details about security issues relating specifically to CICS and IMS, see the *CICS Adapters Administrator's Guide* and *IMS Adapters Administrator's Guide*.

CHAPTER 3

Configuration

This chapter provides the information needed to configure Orbix Mainframe to use the Artix Transport component.

Note: The information provided in this chapter relates specifically to configuring the Artix Transport component of Orbix Mainframe in either secure or insecure mode. For details of configuration items general to Orbix Mainframe as a whole see the *CICS Adapters Administrator's Guide*, the *IMS Adapters Administrator's Guide*, and the *Mainframe Security Guide*.

In this chapter

This chapter discusses the following topics:

Introduction to Orbix Mainframe Configuration

Overview

This section provides an introductory overview of the files used for Orbix Mainframe configuration.

Configuration files summary

Orbix Mainframe employs a rich and flexible file-based configuration. By default, the Orbix Mainframe configuration domain is defined to *orbixhlq*.CONFIG(DEFAULT@) and it references several configuration fragments:

- *orbixhlq*.DOMAIN(FILEDOMA) or *orbixhlq*.DOMAIN(TLSBASE)

  This is the main configuration domain for an Orbix Mainframe deployment. It contains settings that are common to both secure and insecure deployments (for example, CICS and IMS adapter transport plug-in settings).
- *orbixhlq*.DOMAINS(TLSDOMA)

  For secure Orbix Mainframe deployments, this contains additional security-related settings, such as ciphersuite lists, `iiop_tls` ports, and so on.
- *orbixhlq*.CONFIG(ARTIX)

  This contains all the settings necessary for use of the Artix Transport plug-in (that is, for use of the SOAP plug-in).
- *orbixhlq*.CONFIG(ORXINTRL)

  This contains internal configuration settings that should not generally need to be modified.

You may choose to use a different location for your configuration domain. There are several mechanisms by which this can be accomplished. See the *Mainframe Installation Guide* for more details.

Configuring the Artix Transport component

All the configuration items necessary for use of the Artix Transport component are contained in the `ARTIX` configuration fragment.

The `include` statement for the `ARTIX` configuration file is commented out by default in the main configuration file. The comment character should be removed from this `include` statement as part of the Artix Transport customization steps that are described in the *Mainframe Installation Guide*.

Note: This chapter focuses specifically on how to configure the Artix Transport component. For details of configuration topics general to Orbix Mainframe as a whole, see the *CICS Adapters Administrator's Guide*, the *IMS Adapters Administrator's Guide*, and the *Mainframe Security Guide*.

Note: Where this chapter makes reference to the location of configuration items, these relate to default locations.

Common Configuration Items

Overview

This subsection describes the common configuration items for the Artix Transport component that are relevant regardless of whether used in conjunction with a batch, CICS-based, or IMS-based Orbix Mainframe server.

Note: All the items discussed in this section are within the global scope in the `ARTIX` configuration domain.

In this section

This section discusses the following topics:

Sample Configuration Overview

Overview

This subsection provides an overview of the common configuration items (that is, relevant to batch, CICS and IMS) for using the Artix Transport component.

Sample configuration

The following is a sample configuration extract for the common configuration items in the `ARTIX` configuration domain.

Note: The example provided here shows only the contents of the configuration file that are relevant to common Artix Transport configuration. (Ellipses are used to denote text that has been omitted for brevity.)

```
# Artix Configuration Template. The variables in this
# file are required when communicating over SOAP.
#
# Set the following policy to determine the context name
# to use in the URL for your web service.
#
plugins:soap:insecure:root_context_name = "/ionasoap";
plugins:soap:secure:root_context_name = "/secureionasoap";

# Set the following policy to "true" in order for HTTP or HTTPS
# messages to be sent to the log stream as INFO messages.
#
policies:http:trace_requests:enabled = "false";
policies:https:trace_requests:enabled = "false";

# The following settings allow navigation of available HTTP/SOAP
# endpoints in a server. It is recommended that these be disabled
# for a production system.
#
policies:http:browser_navigation:enabled = "true";

plugins:soap:signature_provider = "type_info";
plugins:soap:type_info:source = "DD:TYPEINFO";

…
```

Configuration Details

Overview

This subsection provides details of the common configuration items that are relevant regardless of whether used in conjunction with a batch, CICS-based, or IMS-based Orbix Mainframe server. It discusses the following topics:

- "Root context name" on page 60.
- "Sending INFO messages to log stream" on page 60.
- "Navigating available HTTP endpoints" on page 60.
- "Operation signatures provider" on page 61.
- "Location of type information file" on page 61.

Root context name

The related configuration items are `plugins:soap:insecure:root_context_name` and `plugins:soap:secure:root_context_name`, and they should be added to the global scope. They respectively specify the root context name that is to be used in URLs for insecure and secure invocations on target Web services. These items can be set to any value, but for demonstration purposes the insecure item is set to `"/ionasoap"`. and the secure item is set to `"/secureionasoap"`.

Sending INFO messages to log stream

The related configuration items are `policies:http:trace_requests:enabled` and `policies:https:trace_requests:enabled`, and are within the global scope. They respectively indicate whether HTTP messages should be sent to the log stream as INFO messages when running in insecure or secure mode. If no value is specified for these items, they default to `"false"`, to not send INFO messages to the log stream.

Navigating available HTTP endpoints

The related configuration item is `policies:http:browser_navigation:enabled`. It indicates whether navigation of available HTTP endpoints is enabled for the server.

The recommended setting for a development environment is `"true"`, to enable the easy location of supported WSDL interfaces. The recommended setting for a production environment is `"false"`, to not enable endpoint navigation.

Operation signatures provider

The related configuration item is `plugins:soap:signature_provider`. It determines the mechanism to be used for communication between SOAP and the server side. For the purposes of using the Artix Transport component, this is set to `"type_info"`.

Location of type information file

The related configuration item is `plugins:soap:type_info:source`. It specifies the location of the XML type information store used by the server.

The value for this variable can be set to the one of the following:

- An HFS directory (UNIX System Services)—Specifies that a directory is to be used as the source of the type information file. All entries in this directory are accessed as type information files.
- A PDS (native z/OS)—Specifies that the entire data set is to be used as the source of the type information file. All entries in this PDS are accessed as type information files.
- A DD statement (native z/OS)—Specifies the DD name in the JCL that specifies the PDS to be used for the type information file.

CICS-Specific Configuration Items

Overview

This subsection describes the configuration items for the Artix Transport component that are specific to using the CICS server adapter.

Note: All the items discussed in this section are within the `iona_services.cicsa` scope in the `ARTIX` configuration domain.

In this section

This section discusses the following topics:

Sample Configuration Overview

Overview

This subsection provides an overview of the CICS-specific configuration items for using the Artix Transport component.

Sample configuration

The following is a sample configuration extract for the CICS-specific configuration items in the `ARTIX` configuration domain.

Note: The example provided here shows only the contents of the configuration file that are relevant to CICS-specific Artix Transport configuration. (Ellipses are used to denote text that has been omitted for brevity.)

```
…
iona_services
{
    …
    cicsa
    {
          orb_plugins = ["local_log_stream", "http", "soap"];

          event_log:filters = ["*=WARN+ERROR+FATAL",
            "IT_MFA=INFO_HI+WARN+ERROR+FATAL"];

          plugins:cicsa:direct_persistence = "yes";

          plugins:cicsa:use_soap = "yes";

          binding:server_binding_list = ["SOAP", ""];

          policies:well_known_addressing_policy:http:addr_list =
                    ["%{LOCAL_HOSTNAME}:5051"];

          plugins:cicsa:repository_id    = "type_info";

          plugins:cicsa:type_info:source = "DD:TYPEINFO";
          plugins:soap:type_info:source = "DD:TYPEINFO";

          # set the following to true to enable user/password
          # authentication check
          plugins:soap:check_header_password = "false";
```

```
secure
{
     orb_plugins = ["local_log_stream", "https", "soap"];

     binding:server_binding_list = ["SOAP"];
     policies:well_known_addressing_policy:https:addr_list =
        ["%{LOCAL_HOSTNAME}:5061"];
};
```

Configuration Details

Overview

This subsection provides details of the Artix Transport configuration items that are specific to using the CICS server adapter. It discusses the following topics:

ORB plug-ins list

The related configuration item is `orb_plugins`. It specifies the ORB-level plug-ins that the CICS server adapter should load at `ORB_init()` time when using the SOAP plug-in. If you want the CICS server adapter to accept insecure HTTP connections, ensure that you include `"http"` as a setting for this item under the `iona_services.cicsa` scope. If you want the CICS server adapter to accept secure HTTP connections, ensure that you include `"https"` as a setting for this item under the `iona_services.cicsa.secure` scope. For example:

```
orb_plugins = ["local_log_stream", "https", "soap"];
```

Event logging

The related configuration item is `event_log:filters`. It is used to specify the types of events that the CICS server adapter logs and the extent to which it logs them. To obtain events specific to the CICS server adapter, the `IT_MFA` event logging subsystem can be added to this list item. For example:

```
event_log:filters = ["*=WARN+ERROR+FATAL",
    "IT_MFA=INFO_HI+INFO_MED+WARN+ERROR+FATAL",
    "IT_ATLI2_IP=*"];
```

Based on the preceding example, all IT_MFA events (except for `INFO_LOW`—low priority informational events), and any warning, error, and fatal events will be logged. The level of detail that is provided for IT_MFA events can therefore be controlled by setting the relevant logging levels.

The following is a categorization of the informational events associated with the IT_MFA subsystem.

`INFO_HI`	Configuration settings and server adapter startup and shutdown messages.
`INFO_MED`	Mapping gateway actions and CICS EXCI, IMS APPC or IMS OTMA calls, including return codes.
`INFO_LOW`	CICS or IMS segment data streams.

Persistence mode

The related configuration item is `plugins:cicsa:direct_persistence`. It specifies the persistence mode policy adopted by the CICS server adapter. To allow the CICS server adapter to run as a standalone service, set this to `"yes"`. Otherwise set it to `"no"`, in which case the server adapter contacts and registers with the locator service.

Register supported interfaces with SOAP plug-in

The related configuration item is `plugins:cicsa:use_soap`. It instructs the CICS server adapter to register the CORBA interfaces it supports as services with the SOAP plug-in. If no value is specified for this item, it defaults to `"no"`, to not register supported interfaces as services with the SOAP plug-in. Thus, this variable must be set to `"yes"` to expose these interfaces as SOAP endpoints.

Server binding list

The related configuration item is `binding:server_binding_list`. It specifies the server-side transport bindings that can be used by the CICS server adapter. By default, `"SOAP"` is the only option specified. If you want to run the CICS server adapter in IIOP mode, add `""` to the list.

If you want the CICS server adapter to run in insecure mode, set this item under the `iona_services.cicsa` scope. If you want the CICS server adapter to run in secure mode, set this item under the `iona_services.cicsa.secure` scope.

Port on which server adapter listens

The related configuration items are `policies:well_known_addressing_policy:http:addr_list` and `policies:well_known_addressing_policy:https:addr_list`. They respectively specify the port on which the CICS server adapter listens for client requests when running in insecure or secure mode. The insecure version of this item is under the `iona_services.cicsa` scope. The secure version of this item is under the `iona_services.cicsa.secure` scope.

Type information repository used

The related configuration item is `plugins:cicsa:repository_id`. It specifies the type information repository used by the CICS server adapter to store operation signatures for the processing of SOAP requests. This source supplies the CICS server adapter with operation signatures as required. Valid values are `"ifr"`, `"type_info"`, and `"none"`.

If no value is specified for this item, it defaults to `"ifr"`. However, the recommended setting for SOAP is `"type_info"`. See the *CICS Adapters Administrator's Guide* for more details about using type_info as a source of type information.

Location of adapter type_info store

The related configuration item is `plugins:cicsa:type_info:source`. It specifies the location of the `type_info` store that the adapter plug-in should use if it has been configured to use this particular type repository mechanism.

If no value is specified, the current working directory is used by default. The sample `ARTIX` configuration dataset uses a DD name for this setting.

Location of SOAP type_info store

The related configuration item is `plugins:soap:type_info:source`. It specifies the location of the `type_info` store that the SOAP plug-in should use to process SOAP payloads.

If no value is specified, the current working directory is used by default. The sample `ARTIX` configuration dataset uses a DD name for this setting.

It is recommended that the same location be used (for example, the same DD name) as that which is used by the adapter (see "Location of adapter type_info store" on page 67).

Username and password checking

The related configuration item is `plugins:soap:check_header_password`. It indicates whether the user name and password passed in the header of a SOAP request are to be checked for authentication purposes. If no value is specified for this item, it defaults to `"false"`, to not check the user name and password.

IMS-Specific Configuration Items

Overview

This subsection describes the configuration items for the Artix Transport component that are specific to using the IMS server adapter.

Note: All the items discussed in this section are within the `iona_services.imsa` scope in the `ARTIX` configuration domain.

In this section

This section discusses the following topics:

Sample Configuration Overview

Overview

This subsection provides an overview of the IMS-specific configuration items for using the Artix Transport component.

Sample configuration

The following is a sample configuration extract for the IMS-specific configuration items in the `ARTIX` configuration domain.

Note: The example provided here shows only the contents of the configuration file that are relevant to IMS-specific Artix Transport configuration. (Ellipses are used to denote text that has been omitted for brevity.)

```
...
iona_services
{
   imsa
   {
     event_log:filters = ["*=WARN+ERROR+FATAL","IT_MFA=INFO_HI+WARN+ERROR+FATAL"];

     plugins:imsa:direct_persistence = "yes";

     plugins:imsa:use_soap = "yes";

     orb_plugins = ["local_log_stream", "http", "soap"];

     binding:server_binding_list = ["SOAP", ""];

     policies:well_known_addressing_policy:http:addr_list = ["%{LOCAL_HOSTNAME}:5050"];

     plugins:imsa:repository_id    = "type_info";

     plugins:soap:type_info:source = "DD:TYPEINFO";
     plugins:imsa:type_info:source = "DD:TYPEINFO";
```

```
     # Set the following to true to enable user/password authentication check
     plugins:soap:check_header_password = "false";

     # Included to improve output of imsraw demo
     plugins:ims_otma:output_segment_num = "5";

     secure
        {
         orb_plugins = ["local_log_stream", "https", "soap"];

         binding:server_binding_list = ["SOAP"];

          policies:well_known_addressing_policy:https:addr_list = ["%{LOCAL_HOSTNAME}:5060"];
        };
...
```

Configuration Details

Overview

This subsection provides details of the Artix Transport configuration items that are specific to using the IMS server adapter. It discusses the following topics:

ORB plug-ins list

The related configuration item is `orb_plugins`. It specifies the ORB-level plug-ins that the IMS server adapter should load at `ORB_init()` time when using the SOAP plug-in. If you want the IMS server adapter to accept insecure HTTP connections, ensure that you include `"http"` as a setting for this item under the `iona_services.imsa` scope. If you want the IMS server adapter to accept secure HTTP connections, ensure that you include `"https"` as a setting for this item under the `iona_services.imsa.secure` scope. For example:

```
orb_plugins = ["local_log_stream", "https", "soap"];
```

Event logging

The related configuration item is `event_log:filters`. It is used to specify the types of events that the IMS server adapter logs and the extent to which it logs them. To obtain events specific to the IMS server adapter, the `IT_MFA` event logging subsystem can be added to this list item. For example:

```
event_log:filters = ["*=WARN+ERROR+FATAL",
    "IT_MFA=INFO_HI+INFO_MED+WARN+ERROR+FATAL",
   "IT_ATLI2_IP=*"];
```

Based on the preceding example, all IT_MFA events (except for `INFO_LOW`—low priority informational events), and any warning, error, and fatal events will be logged. The level of detail that is provided for IT_MFA events can therefore be controlled by setting the relevant logging levels.

The following is a categorization of the informational events associated with the IT_MFA subsystem.

`INFO_HI`	Configuration settings and server adapter startup and shutdown messages.
`INFO_MED`	Mapping gateway actions and CICS EXCI, IMS APPC or IMS OTMA calls, including return codes.
`INFO_LOW`	CICS or IMS segment data streams.

Persistence mode

The related configuration item is `plugins:imsa:direct_persistence`. It specifies the persistence mode policy adopted by the IMS server adapter. To allow the IMS server adapter to run as a standalone service, set this to `"yes"`. Otherwise set it to `"no"`, in which case the server adapter contacts and registers with the locator service.

Register supported interfaces with SOAP plug-in

The related configuration item is `plugins:imsa:use_soap`. It instructs the IMS server adapter to register the CORBA interfaces it supports as services with the SOAP plug-in. If no value is specified for this item, it defaults to `"no"`, to not register supported interfaces as services with the SOAP plug-in. Thus, this variable must be set to `"yes"` to expose these interfaces as SOAP endpoints.

Server binding list

The related configuration item is `binding:server_binding_list`. It specifies the server-side transport bindings that can be used by the IMS server adapter. By default, `"SOAP"` is the only option specified. If you want to run the IMS server adapter in IIOP mode, add `""` to the list.

If you want the IMS server adapter to run in insecure mode, set this item under the `iona_services.imsa` scope. If you want the IMS server adapter to run in secure mode, set this item under the `iona_services.imsa.secure` scope.

Port on which server adapter listens

The related configuration items are `policies:well_known_addressing_policy:http:addr_list` and `policies:well_known_addressing_policy:https:addr_list`. They respectively specify the port on which the IMS server adapter listens for client requests when running in insecure or secure mode. The insecure version of this item is under the `iona_services.imsa` scope. The secure version of this item is under the `iona_services.imsa.secure` scope.

Type information repository used

The related configuration item is `plugins:imsa:repository_id`. It specifies the type information repository used by the IMS server adapter to store operation signatures for the processing of SOAP requests. This source supplies the IMS server adapter with operation signatures as required. Valid values are `"ifr"`, `"type_info"`, and `"none"`.

If no value is specified for this item, it defaults to `"ifr"`. However, the recommended setting for SOAP is `"type_info"`. See the *IMS Adapters Administrator's Guide* for more details about using type_info as a source of type information.

Location of adapter type_info store

The related configuration item is `plugins:imsa:type_info:source`. It specifies the location of the `type_info` store that the adapter plug-in should use if it has been configured to use this particular type repository mechanism.

If no value is specified, the current working directory is used by default. The sample `ARTIX` configuration dataset uses a DD name for this setting.

Location of SOAP type_info store

The related configuration item is `plugins:soap:type_info:source`. It specifies the location of the `type_info` store that the SOAP plug-in should use to process SOAP payloads.

If no value is specified, the current working directory is used by default. The sample `ARTIX` configuration dataset uses a DD name for this setting.

It is recommended that the same location be used (for example, the same DD name) as that which is used by the adapter (see "Location of adapter type_info store" on page 74).

Username and password checking

The related configuration item is `plugins:soap:check_header_password`. It indicates whether the user name and password passed in the header of a SOAP request are to be checked for authentication purposes. If no value is specified for this item, it defaults to `"false"`, to not check the user name and password.

Number of initial output segments allocated

The related configuration item is `plugins:ims_otma:output_segment_num`. It specifies the number of initial output segments to be allocated by the OTMA-based IMS server adapter for the processing of SOAP requests. If no value is specified for this item, it defaults to `"5"`.

Note: This setting only applies if you have configured your IMS server adapter to use the `ims_otma` plug-in.

CHAPTER 4

SOAP Security Considerations

This chapter provides details of the different security mechanisms supported by the Artix Transport component in terms of how they can be configured and what they involve.

In this chapter

This chapter discusses the following topics:

Note: This chapter provides details of Orbix Mainframe security considerations when running in SOAP mode. For details of Orbix Mainframe security considerations when running in IIOP mode, see the *Mainframe Security Guide*. Additionally, for details of security considerations that are specific to CICS or IMS, see the *CICS Adapters Administrator's Guide* or *IMS Adapters Administrator's Guide*.

Security Architecture Overview for SOAP Mode

Overview

This subsection provides an overview of the architecture of the Orbix Mainframe security framework when running in SOAP mode.

Graphical overview

Figure 6 provides a graphical overview of the architecture of the Orbix Mainframe security framework when running in SOAP mode.

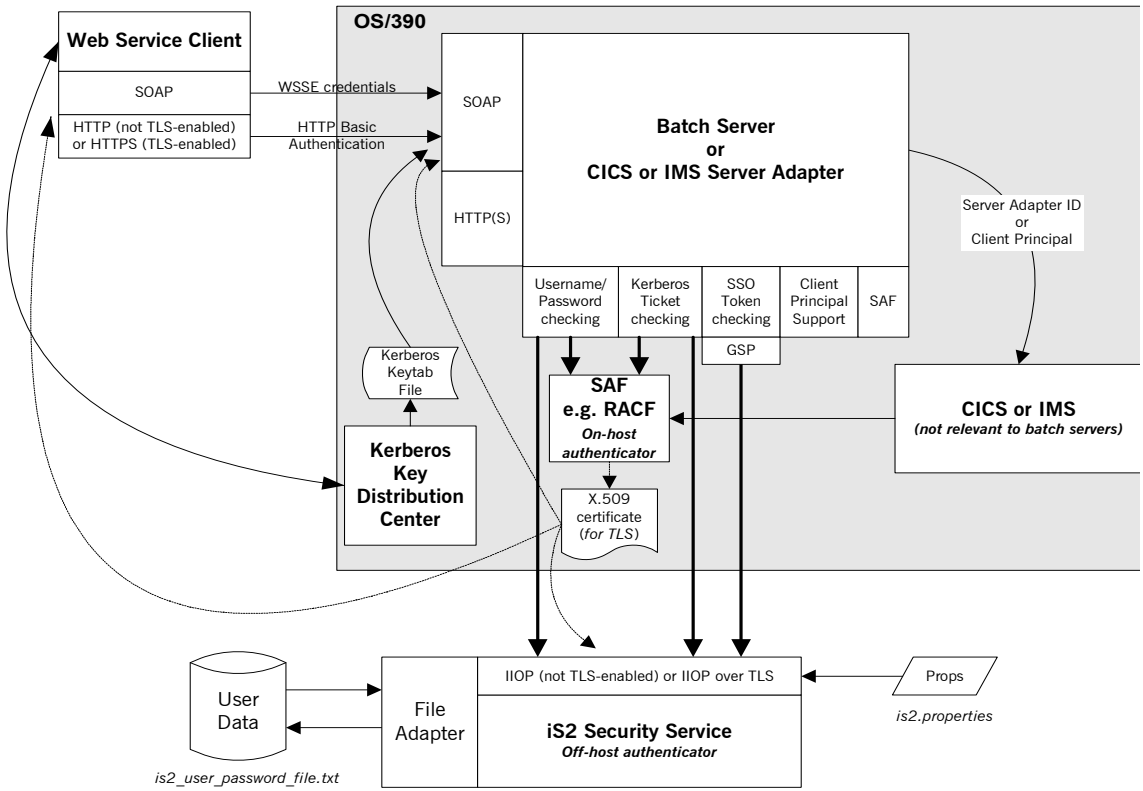

Figure 6: *Orbix Mainframe SOAP Mode Security Architecture*

Summary of Security Features and Credentials

Overview

This subsection provides an overview of the various types of security features and credentials supported by the Artix Transport component. It also outlines the order of precedence used for checking security credentials with a Web service request, and the different mechanisms that can be used to perform such checks.

Summary of security features

The Artix Transport component supports the following security features:

- User name and password checking—This involves checking a username/password combination sent in a SOAP header for the purposes of Web service client authentication.
- Kerberos ticket checking—This involves checking a Kerberos security ticket sent in a SOAP header for the purposes of Web service client authentication.
- Single Sign-On (SSO) token checking—This involves checking an SSO token that a client may use instead of its username/password to communicate with other applications, for the purposes of Web service client authentication.
- HTTP Basic Authentication—This involves checking a username/password combination sent in an HTTP header for the purposes of Web service client authentication.
- SSL/TLS security—This involves encrypting data on-the-wire and performing client authentication checks using trusted certificates.
- Client Principal support—This involves enabling the CICS or IMS server adapter to run CICS and IMS applications under the client's user ID instead of under its own user ID.
- SAF checking—This involves checking a Principal value sent in a SOAP header for the purposes of controlling client access to specific operations and servers.

Each of these features is described in turn later in this chapter.

Summary of security credentials

The various types of security credentials supported by the Artix Transport component include:

- *WSSE username token*—This relates to user name and password checking. The Web service security extension (WSSE) UsernameToken is a username/password combination that can be sent in a SOAP header. The specification of WSSE UsernameToken is contained in the *WSS UsernameToken Profile 1.0* document from OASIS at `www.oasis-open.org`.
- *WSSE Kerberos ticket*—The WSSE Kerberos specification is used to send a Kerberos security ticket in a SOAP header. If you use Kerberos, you must also configure the off-host Security Service (iSF) to use the Kerberos adapter.
- *SSO token*—An SSO token is propagated in the context of a system that uses single sign-on. This means that after initially signing on, a client communicates with other applications by passing an SSO token in place of the original username and password. This helps to minimize the exposure of user names and passwords to snooping.
- *HTTP Basic Authentication*—HTTP Basic Authentication is used to propagate username/password credentials in an HTTP header.
- *X.509 certificate*—This relates to SSL/TLS security, which ensures that data is encrypted on-the-wire and that client authentication is performed using trusted certificates.

Order of precedence for Web service credentials checking

The following is the order of precedence for checking the various security credentials that may be sent as part of a Web service client request (assuming Orbix Mainframe is configured to support all such checks):

1. Perform user name and password checking. That is, check the WSSE username token in the SOAP header.
2. If a username/password is not present in the SOAP header, and if the HTTP layer has received username/password credentials using HTTP Basic Authentication, check these HTTP credentials.
3. If a username/password is not present in either the SOAP or HTTP header, perform Kerberos ticket checking.

4. If the check in point 3 fails, or a Kerberos ticket is not present, perform SSO token checking.

Checking credentials on-host or off-host

For the purposes of credentials checking, Orbix Mainframe gives you the option of using either on-host SAF security facilities (such as RACF) or the off-host Security Service (iSF).

The `plugins:soap:use_security_service` configuration item allows you to determine whether credentials checking is to be performed on-host or off-host. The default value for this configuration item is `"false"`, which indicates that all credentials checking is to be performed by the on-host SAF security facilities. If you specify a value of `"true"`, this indicates that all credentials checking is to be performed off-host by iSF.

All credentials checking for the same client request must be performed either on-host or off-host. There is no option to mix and match between on-host and off-host credentials checking in the same request.

Note: SSO token checking can only be performed off-host by iSF. It is not supported by on-host SAF security facilities. For this reason, if you want to enable SSO token checking for a particular request, all other credentials checking for that request must also be performed by off-host iSF.

User Name and Password Checking

Overview

Orbix Mainframe can be configured to perform client authentication checks against a WSSE username token (that is, a username/password combination sent in a SOAP header). These authentication checks can be performed on-host by SAF security facilities (such as, RACF, CA-ACF/2 and CA-TopSecret) or off-host by the Security Service (iSF).

Graphical overview

Figure 7 provides a graphical overview of how user name and password checking is handled in the Orbix Mainframe security architecture.

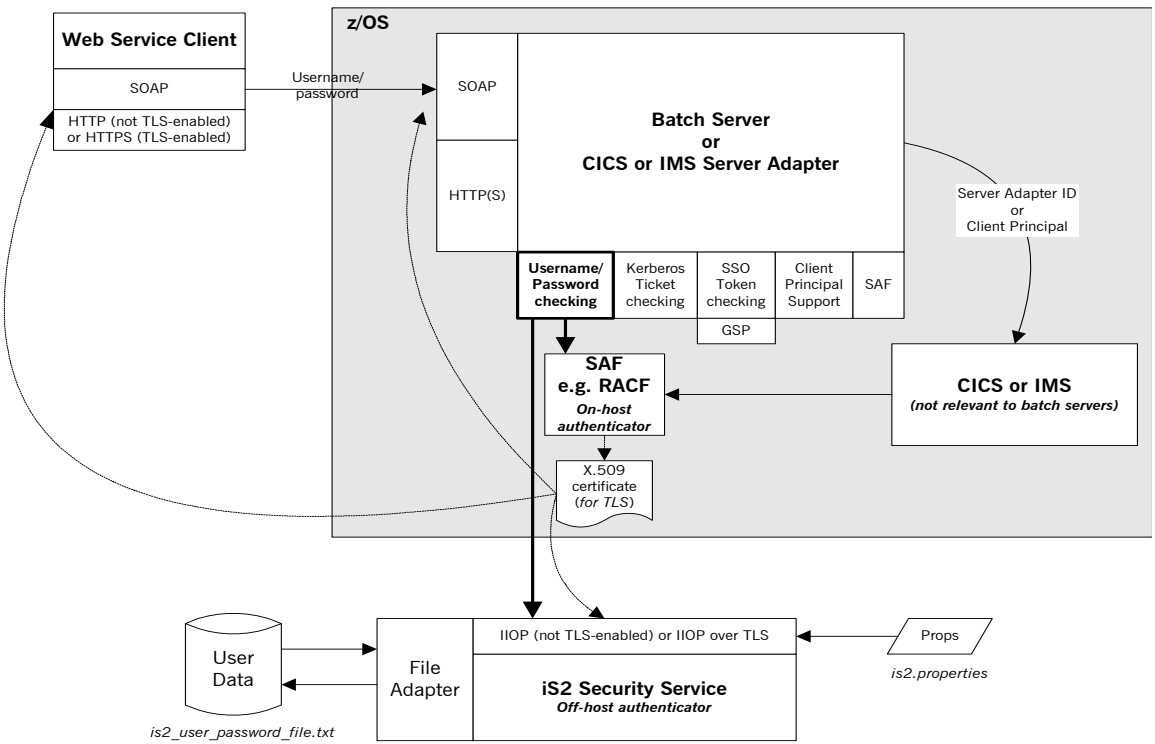

Figure 7: *Username/Password Checking in Security Architecture*

Purpose of password checking

The purpose of password checking is to enable the server to authenticate the client that is invoking a Web service request. If password checking is enabled in the server configuration, the client application must send a user name and password in the header of a SOAP request. If the user name and password are valid, the server can accept the client request (and, in the case of the CICS or IMS server adapter, forward it to the relevant CICS program or IMS transaction). However, if a SOAP request contains an invalid user name or password, or a non-existent header, the request fails if the server has no other way of authenticating the client.

Enabling user name and password checking

Password checking is enabled by setting the following configuration item in the configuration scope of the server:

```
plugins:soap:check_header_password = "true";
```

This configuration item is set to `"false"` by default, which means password checking is not enabled by default.

If you want on-host RACF facilities to check the user name and password, set the `plugins:soap:use_security_service` configuration item to `"false"`. Alternatively, if you want the off-host Security Service (iSF) to check the user name and password, set the `plugins:soap:use_security_service` configuration item to `"true"` (the default value).

Format of the user ID and password in a request

The server requires that the user name and password are transmitted in the format prescribed by the OASIS Web Services Security (WSS) UsernameToken Profile specification. The server first reads the user name and password from the SOAP header. Depending on where credentials checks are being performed, the server then makes a call to either on-host

SAF security facilities or off-host iSF, to check if the password is correct for the specified user name. The header passed from the client to the server should take the following format, as described by the WSS specification:

```
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
    xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext">
    <S:Header>
        …
        <wsse:Security>
            <wsse:UsernameToken>
                <wsse:Username>JSMITH</wsse:Username>
                <wsse:Password>PASS</wsse:Password>
            </wsse:UsernameToken>
        </wsse:Security>
        …
    </S:Header>
    …
</S:Envelope>
```

A number of more recent Web services client development products provide the ability to set this profile in a simple manner. Others enable the setting of SOAP Headers, which can be used to achieve the same results. Refer to the documentation of your Web services client development tool for details on how to create this security profile.

Setting up required privileges for on-host checks

If you want the server to perform password checks against on-host SAF security facilities, additional security settings might be required. The requirements vary, as follows, depending on whether the FACILITY RACF class profile BPX.DAEMON is defined on your system:

- If BPX.DAEMON is defined, the caller's user ID must have READ access to the BPX.DAEMON profile. Additionally, all programs running in the address space must be loaded from a library controlled by a security product. A library identified to RACF Program Control is one example.
- If BPX.DAEMON is not defined, this user ID must have a UID of 0 assigned to it in the OMVS segment of its RACF user profile.

Kerberos Ticket Checking

Overview

Orbix Mainframe can be configured to perform client authentication checks against a Kerberos ticket sent in a SOAP header. These authentication checks can be performed on-host by SAF security facilities (such as, RACF, CA-ACF/2 and CA-TopSecret) or off-host by the Security Service (iSF).

Graphical overview

Figure 8 provides a graphical overview of how Kerberos ticket checking is handled in the Orbix Mainframe security architecture.

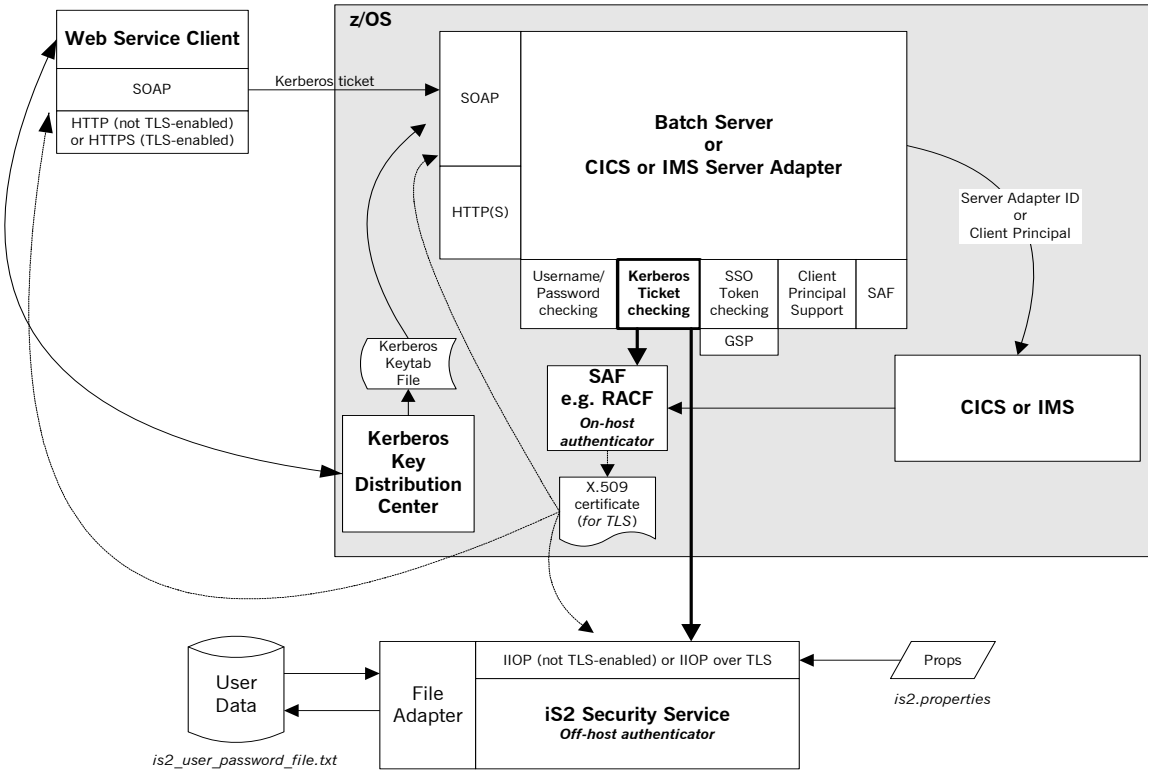

Figure 8: *Kerberos Ticket Checking in Security Architecture*

What is Kerberos?

Kerberos is an authentication system used for authenticating service requests in a computer network. It allows two parties to exchange private information across an otherwise public network. It provides a means by which servers can authenticate the clients that are requesting their services, without the need for the client's password to pass through the open network.

Note: Kerberos deals only with authentication checking and does not in any way facilitate authorization checks. In other words, while it can be used to identify a particular client, it cannot be used to determine whether that client has the authority to request a particular service.

How Kerberos works

The following is an overview of the Kerberos authentication process:

1. A client requests a session key—a ticket-granting ticket—from a central authority called the Key Distribution Center (KDC).
2. The KDC authenticates the client and generates the ticket-granting ticket based on the user's password and a randomly generated key representing the requested service. The KDC then sends the ticket-granting ticket to the client and places the randomly generated key in a keytab file for subsequent use by the server.
3. When the client wants to request a particular service, it requests a service ticket from the KDC, and sends its ticket-granting ticket as part of this request.

   By sending the ticket-granting ticket along with its service ticket request, the client can identify itself to the KDC without needing to be prompted again for its password.
4. The KDC generates a service ticket based on the requested server's key and encodes the ticket with the client's password. The service ticket is also time-stamped, so that the client can use it to make additional service requests within a certain time period without needing to be reauthenticated or request another service ticket. The KDC then sends the service ticket to the client.
5. The client decodes the reply from the KDC, using the appropriate password, and embeds the service ticket in the request message it sends to the server.

6. The server decodes the ticket, knowing that the client has already been authenticated by the KDC.

Note: At this point, the server can choose to accept or reject the client request. It is beyond the scope of Kerberos to determine whether a client has the authority to request a particular service. However, the security service (iSF) can do so, using a "roles" XML file for authorization purposes.

Kerberos concepts

The Key Distribution Center (KDC) is a central component in Kerberos security. As described in "How Kerberos works" on page 87, the KDC is responsible for initial client authentication, and for issuing ticket-granting tickets for particular client sessions and service tickets for particular service requests.

A *realm* is the scope or domain of the KDC. A realm name is usually based on the name of the TCP/IP subdomain it covers (for example, `microfocus.com`). Each client and server exists in a single realm, but there can be cross-realm authentication between clients and servers in different realms. By convention, realm names are written in uppercase (for example, `MICROFOCUS.COM`).

A *principal* is the name or token by which clients and servers are known to the KDC and each other. For client users, the token takes the form `name@realm` (for example, `John Smith@MICROFOCUS.COM`). For servers, the token takes the form *`name/instance@realm`*, where instance is usually the hostname on which the server is running (for example, `foo/mymachine.dublin.emea.microfocus.com@MICROFOCUS.COM`).

A *credentials cache* is a list of recently used tickets, including the ticket-granting ticket, for a particular user. It is usually stored in memory or in a temporary file with no shared permissions. The use of a credentials cache reduces the number of calls a client needs to make on the KDC. Kerberos provides the following command-line utilities to manage a credentials cache:

`kinit`	This lets the user initialize the credentials cache with their client principal name. The `kinit` command also automatically stores the ticket-granting ticket for the user in the credentials cache.
`klist`	This displays the contents of the credentials cache.
`kdestroy`	This deletes the contents of the credentials cache.

Kerberos on z/OS

The Network Authentication Service is IBM's implementation of Kerberos on z/OS. All Kerberos information on z/OS is stored in RACF. RACF user profiles have an extra KERB segment that contains the user's Kerberos principal and Kerberos password. (The Kerberos password is generated automatically by RACF whenever the user changes their z/OS password). KERBLINK profiles are used to map Kerberos principals to z/OS user IDs. When a RACF user profile is allocated a new KERB segment, z/OS automatically creates a KERBLINK profile that maps the Kerberos principal to the corresponding user ID.

The KDC component of Kerberos on z/OS runs as a started task. However, it reads its configuration details from the hierarchical file system. Generated keytab files on z/OS are always HFS files, and are never stored in data sets.

Enabling Kerberos ticket checking

Kerberos ticket checking is enabled by setting the following configuration items in the configuration scope of the server:

`plugins:soap:check_header_token`	This indicates whether to check the Kerberos ticket in incoming SOAP headers. This must be set to `"true"` to enable Kerberos ticket checking. The default is `"false"`.
`kerberos:server_principal`	This specifies the name to be used for validation of incoming Kerberos tickets. If this is not specified, the WSDL service name is used by default.

If you want on-host RACF facilities to check the Kerberos ticket, set the `plugins:soap:use_security_service` configuration item to `"false"`. Alternatively, if you want the off-host Security Service (iSF) to check the Kerberos ticket, set the `plugins:soap:use_security_service` configuration item to `"true"` (the default value).

On-host RACF checking

If `plugins:soap:use_security_service` is set to `"false"`, Kerberos ticket checking is performed using on-host RACF facilities. In this case, the token check uses the standard GSS-API. For this reason, Orbix Mainframe requires that a Kerberos token is a base64-encoded GSS-API token. The token can be checked using RACF in either of two ways:

- The key can be looked up in the user's RACF profile provided all of the following apply:
  - The `KRB5_SERVER_KEYTAB` environment variable is set to `1`.
  - The user ID that the server is running under has read access to `IRR.RUSERMAP.FACILITY`.
  - The server principal matches the default Kerberos principal for the user ID.

  Note: In this case, no keytab file is necessary.

- The key is looked up in the keytab file, which contains an entry for the server principal.

Off-host iSF checking

If `plugins:soap:use_security_service` is set to `"true"`, Kerberos ticket checking is performed off-host using an instance of the Security Service (iSF). In this case, all the following apply:

- Ensure that the following configuration items are included in the configuration scope of the server:
  - `initial_references:IT_SecurityService:reference`—This must specify the object reference to be used for contacting the iSF server.
  - `initial_references:IT_CSIAuthenticationObject:plugin`—This must specify a value of `"gsp"`, so that the GSP plug-in can be loaded, as required.

- Ensure that iSF is configured to use the Kerberos adapter. For example, consider the following example `is2.properties` file:

```
### Sample is2.properties file
com.iona.isp.adapters=krb5

com.iona.isp.adapter.krb5.class=com.microfocus.security.is2adapter.krb5.IS2KerberosAdapter
com.iona.isp.adapter.krb5.param.java.security.krb5.realm=DUBLIN.EMEA.MICROFOCUS.COM
com.iona.isp.adapter.krb5.param.java.security.krb5.kdc=
   mymachine.dublin.emea.microfocus.com
com.iona.isp.adapter.krb5.param.java.security.auth.login.config=/opt/jaas.conf
com.iona.isp.adapter.krb5.param.javax.security.auth.useSubjectCredsOnly=false
com.iona.isp.adapter.krb5.param.sun.security.krb5.debug=true
```

- Ensure that iSF specifies the correct Kerberos server principal. For example, consider the following example `jaas.conf` file:

```
/**
 * Sample jaas.conf file
 */
com.sun.security.jgss.accept {
  com.sun.security.auth.module.Krb5LoginModule required debug=true storeKey=true doNotPrompt=true
  principal="test_server/mymachine.dublin.emea.microfocus.com@DUBLIN.EMEA.MICROFOCUS.COM"
  useKeyTab=true keyTab="/home/bnorth/kerberos/keytab" ;
};
```

- Because iSF can use only a single server principal, ensure that each server has its own instance of iSF with which it can communicate, or alternatively ensure that all servers share the same Kerberos principal.

Extracting a valid client principal from an authenticated Kerberos ticket

You can use the `plugins:soap:extract_token_principal` configuration item to enable the server to set the GIOP requesting principal for an incoming request to an MVS user ID that matches the Kerberos principal found in the authenticated Kerberos ticket.

This option uses the RACF callable service, `R_usermap`. The `R_usermap` callable service requires that:

- The user making the call (that is, the server ID) has `READ` access to the `FACILITY` class `IRR.RUSERMAP`.
- A KERBLINK profile exists that maps the Kerberos principal being checked to the MVS user ID.

`R_usermap` does not require that the Kerberos KDC is running on z/OS, so `plugins:soap:extract_token_principal` can be set to `"true"` regardless of whether client authentication is to be performed on-host by RACF or off-host by iSF (that is, regardless of whether `plugins:soap:use_security_service` is set to `"true"` or `"false"`).

SSO Token Checking

Overview

Orbix Mainframe can be configured to perform client authentication checks against a Single Sign-On (SSO) token sent in a SOAP header. These authentication checks can be performed off-host by the Security Service (iSF).

Graphical overview

Figure 9 provides a graphical overview of how SSO token checking is handled in the Orbix Mainframe security architecture.

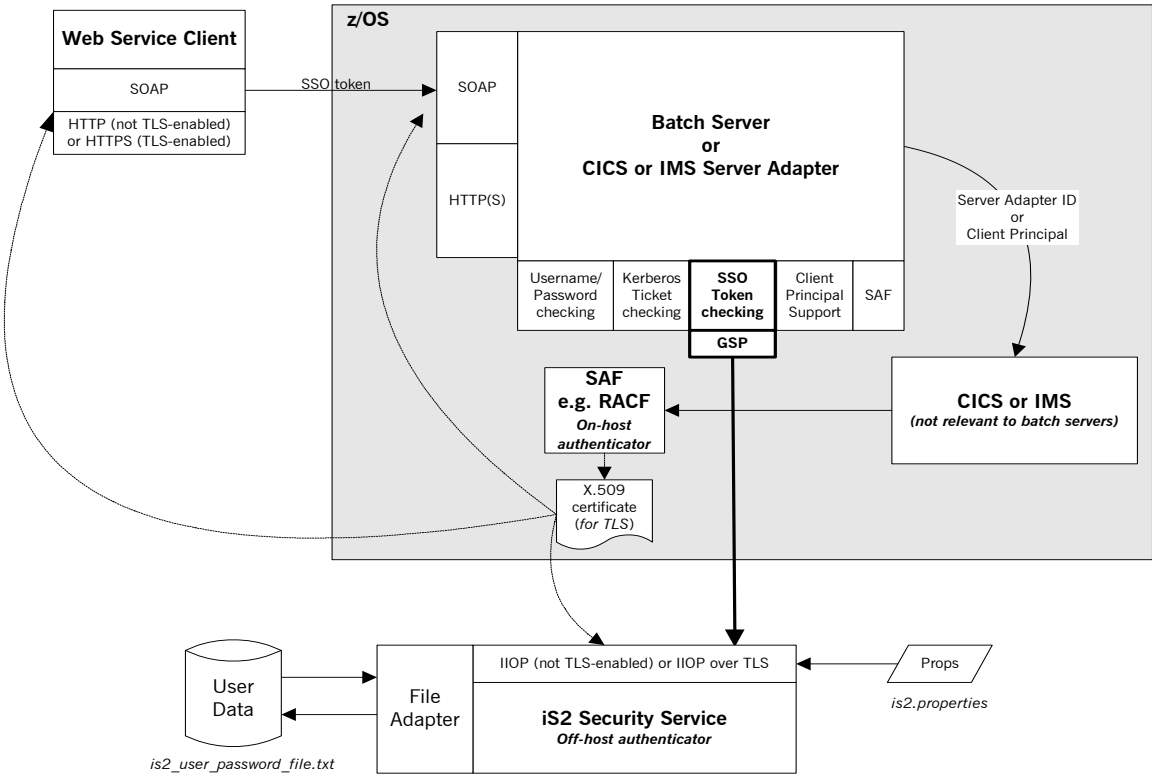

Figure 9: *SSO Token Checking in Security Architecture*

What is SSO?

Single sign-on (SSO) is an Orbix Security Framework feature that is used to minimize the exposure of user names and passwords to snooping. After initially signing on, a client communicates with other applications by passing an SSO token in place of the original username and password.

Graphical overview of how SSO works

Figure 10 provides a graphical overview of how a client first requests an SSO token and then propagates it as part of a Web service request.

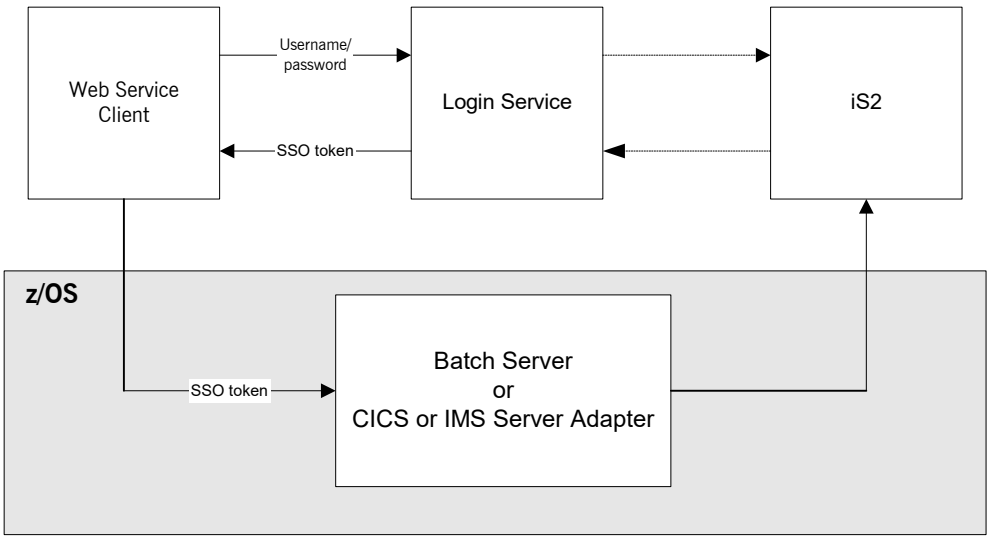

Figure 10: *Request for and Propagation of SSO tokens*

How SSO works

The following is an overview of how SSO token checking works (see Figure 10):

1. The Web service client first invokes the `login()` method, to communicate with the Login Service, which might be running separate to or within the Security Service (iSF).
2. The client passes a user name and password, which the Login Service authenticates against iSF.

3. If the client credentials are valid, the Login Service returns an SSO token to the client.
4. The client sends the SSO token in the SOAP header of its Web service request. In this case, the username field is populated with `_SSO_TOKEN_`, and the password field is populated with the actual SSO token.
5. The server invokes the `authenticate()` method, to request off-host iSF to authenticate the SSO token.

Enabling SSO token checking

Because SSO token checking is part of the family of WSSE credentials (along with username/password and Kerberos ticket checking), it is enabled by setting the following configuration item in the configuration scope of the server:

```
plugins:soap:check_header_token = "true";
```

This configuration item is set to `"false"` by default, which means SSO token checking is not enabled by default.

SSO token checking can only be performed off-host by the iSF security service. For this reason, the `plugins:soap:use_security_service` configuration item must be set to `"true"`.

Note: If you wish to pass an SSO token in the SOAP header, this means that any other credentials checking must also be performed off-host.

HTTP Basic Authentication

Overview

Orbix Mainframe can be configured to perform client authentication checks against a username/password combination sent in an HTTP header. These authentication checks can be performed on-host by SAF security facilities (such as, RACF, CA-ACF/2 and CA-TopSecret) or off-host by the Security Service (iSF).

Graphical overview

Figure 5 provides a graphical overview of how HTTP Basic Authentication is handled in the Orbix Mainframe security architecture.

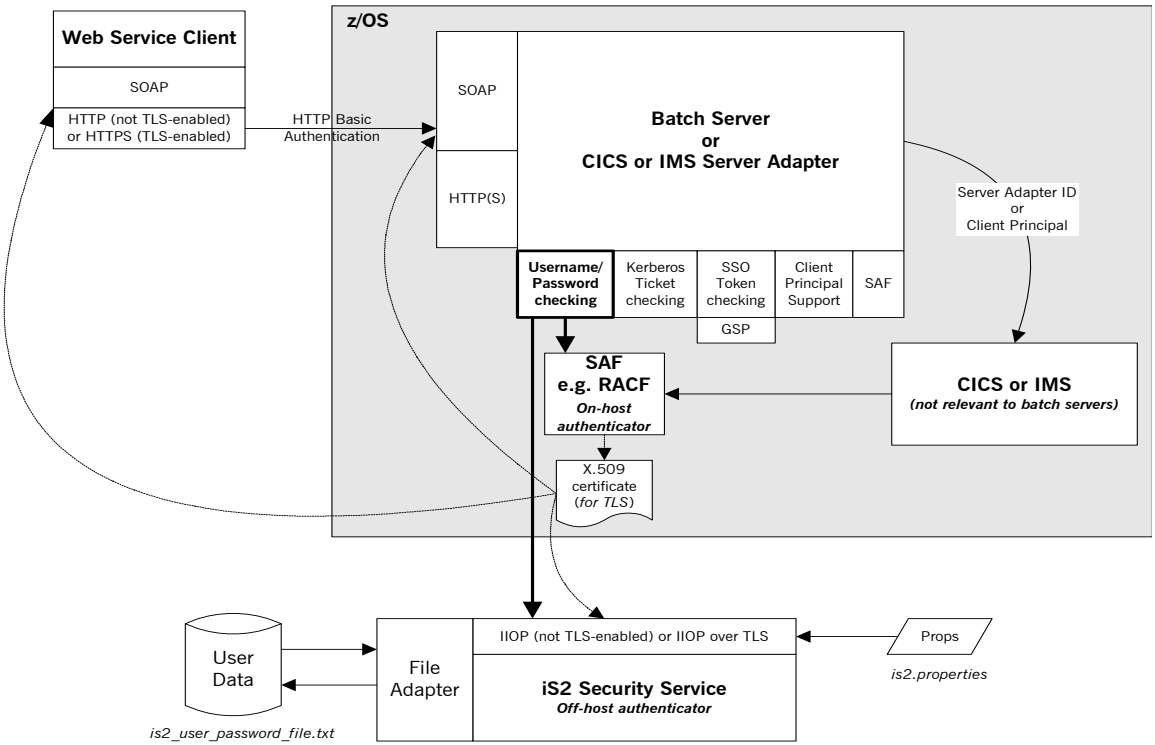

Figure 11: *HTTP Basic Authentication in Security Architecture*

Enabling HTTP Basic Authentication

HTTP Basic Authentication is enabled in the same way that you can enable checking of the username/password in the SOAP header; that is, by setting the following configuration item in the configuration scope of the server:

```
plugins:soap:check_header_password = "true";
```

This configuration item is set to `"false"` by default, which means HTTP Basic Authentication is not enabled by default.

If you want on-host RACF facilities to perform the authentication check on the credentials received in the HTTP header, set the `plugins:soap:use_security_service` configuration item to `"false"`. Alternatively, if you want the off-host Security Service (iSF) to perform the check, set the `plugins:soap:use_security_service` configuration item to `"true"` (the default value).

Client Principal Support

Overview

The CICS or IMS server adapter can be configured to run a CICS program or IMS transaction under the client user ID (that is, the client principal) rather than under the server adapter's own user ID. This subsection discusses the following topics:

Note: User name and password checking must be activated, as described in "User Name and Password Checking" on page 83", to allow client principal support to be activated.

Using a principal

The client principal (that is, client user ID) must be passed in the SOAP request as part of the SOAP header. It must be similar to the following:

```
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
    xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext">
    <S:Header>
        …
        <wsse:Security>
            <wsse:UsernameToken>
                <wsse:Username>JSMITH</wsse:Username>
            </wsse:UsernameToken>
        </wsse:Security>
        …
    </S:Header>
    …
</S:Envelope>
```

In the preceding example, `JSMITH` is the client principal.

Enabling client principal support

Client principal support is enabled by setting the `plugins:cicsa:use_client_principal` or `plugins:imsa:use_client_principal` configuration item to `"yes"` in the configuration domain for the server adapter.

Note: This configuration item is not included by default in the configuration file, so you must add it to the server adapter configuration if you want to activate client principal support. You can add it within the global scope, or alternatively within the `iona_services.cicsa` or `iona_services.imsa` scope.

When this item is set to `"yes"`, the principal is to be obtained from the SOAP request, truncated to eight characters, and converted to uppercase. The server adapter assumes the identity of the client and then runs the CICS program or IMS transaction, using the client principal rather than under the server adapter's own user ID. If no principal is available or it is invalid, the transaction fails. This results in CICS or IMS, and their associated plug-ins, making their security checks against the client principal.

If client principal support is not enabled, the CICS program or IMS transaction runs under the server adapter's own user ID, and security checks are made against this user ID. This requires that the server adapter has access to all the resources it needs.

Mapping client principal values to z/OS user IDs

For the purposes of checking access to CICS or IMS resources, the only translation that the server adapter performs between the client Principal value and the z/OS user ID is to convert lowercase letters to uppercase and to restrict the ID to no more than eight characters. Long Principal values from other platforms have their Principals truncated to eight characters. This means that Principals longer than eight characters must have their first eight characters match a valid z/OS user ID. Principals with eight characters or less in length must entirely match a valid z/OS user ID. This means that users must have the same name on the client platform and z/OS.

Switching threads

When client principal support is enabled, the server adapter uses the `pthread_security_np()` call on the thread that is processing the client request, to switch that thread to run the under the requested user ID (client

principal). It then makes the request, using the client principal. For this to work, the server adapter must be program-controlled. See "Setting up required privileges" next for details of how to do this.

Setting up required privileges

If client principal support is enabled, the user ID under which the server adapter runs (that is, the client principal) might need to be granted special privileges to enable thread-level security environments. The requirements vary, as follows, depending on whether the FACILITY RACF class profile BPX.SERVER is defined on your system:

- If BPX.SERVER is defined, the user ID must have READ access to the BPX.SERVER profile. Otherwise, EPERM errors result when the server adapter is trying to switch identities on the thread. Additionally, the server adapter executable must reside in a z/OS load library that is PADS-defined. (PADS is the acronym for Program Access to Data Sets.)
- If BPX.SERVER is not defined, this user ID must have a UID of 0 assigned to it in the OMVS segment of its RACF user profile.

Additionally, because the server adapter is processing requests for users without having their passwords, you must activate the SURROGAT RACF class and define profiles in it that allow the server adapter's user ID to impersonate particular users. You can do this by establishing a profile for each potential client user. For example:

```
RDEFINE SURROGAT BPX.SRV.client1 UACC(NONE)
PERMIT BPX.SRV.client1 CLASS(SURROGAT) ID(Adapter_ID)
   ACCESS(READ)
RDEFINE SURROGAT BPX.SRV.client2 UACC(NONE)
PERMIT BPX.SRV.client2 CLASS(SURROGAT) ID(Adapter_ID)
   ACCESS(READ)
```

Note: In the preceding example, *Adapter_ID* represents the RACF user ID for the server adapter.

Alternatively, you might want to use a generic profile that allows the server adapter to *impersonate* any client user. For example:

```
RDEFINE SURROGAT BPX.SRV.* UACC(NONE)
PERMIT BPX.SRV.* CLASS(SURROGAT) ID(Adapter_ID) ACCESS(READ)
```

Access to such profiles should be very tightly controlled.

Note: If, at this point, the address space is still not program-controlled, the server adapter throws an exception back to the client and logs an error message, to indicate that it could not switch to that user ID and is therefore not going to attempt to start the CICS program or IMS transaction.

Further reading

See the IBM publication *Planning:OpenEdition MVS, SC23-3015* for more information on enabling thread-level security for servers.

SAF Checking

Overview

The SAF plug-in provides optional Principal-based access control. A server might accept or reject incoming requests, based upon a Principal value in the SOAP request header. The value is treated as a z/OS user ID and access is checked against an operation-specific SAF profile name. Access can therefore be controlled on a per-operation basis, or (using generic profiles) on a per-server basis.

Enabling SAF checking

The steps to enable SAF checking are:

1. Add `"saf"` to the list of values for the `orb_plugins` configuration item in the configuration scope of the server, as follows:

```
…
orb_plugins=["local_log_stream", "saf", "http", "soap"];
…
```

2. Add `"SAF"` to the list of values for the `binding:server_binding_list` configuration item in the configuration scope of the server, as follows:

```
…
binding:server_binding_list = ["SAF+SOAP", "SAF"];
…
```

3. Ensure that the `plugins:soap:check_header_password` configuration item is set to `"true"`.
4. If you want to specify a profile class other than `"IONA"`, add the following configuration item to the global scope, to ensure that the SAF plug-in uses the correct SAF resource class (where *`MYCLASS`* represents the name of your SAF resource class):

```
plugins:saf:profile_class = "MYCLASS";
```

Note: If your chosen resource class does not exist, ask your systems programmer to add an entry for it in both the RACF Class Descriptor Table and the RACF Router Table. The IBM publication z/OS *Security Server (RACF) System Programmer's Guide* provides details of how to do this.

5. If you want Orbix Mainframe to use an ORB name other than "`iona_services.imsa`" or "`iona_services.cicsa`", to build up the resource name, add the following configuration item to the global scope:

```
plugins:saf:service_name = "MYSERVICE";
```

6. If the SAF FACILITY class profile, BPX.SERVER, is defined, ensure that the user ID under which the server runs has READ access to this profile.

   Alternatively, if the SAF FACILITY class profile, BPX.SERVER, is not defined, ensure that the user ID under which the server runs has a UID of zero.

HTTPS Security

Overview

Depending on the security requirements at your site, the activation of HTTPS (an SSL/TLS-based security system) should be considered. The use of SSL/TLS security ensures that data is encrypted on-the-wire and that client authentication is performed using trusted SSL certificates. Additionally, if user name and password checking is enabled (see "User Name and Password Checking" on page 83), SSL/TLS ensures that the password in the header of a SOAP request cannot be intercepted while it is travelling with the request.

Generating SSL certificates

SSL/TLS security uses a system of trusted X.509 certificates for client authentication. Orbix Mainframe allows you to generate these certificates as follows:

1. Customize the *orbixhlq*.JCLLIB(GENCERT) JCL supplied with your Orbix Mainframe installation.

   Note: Details of how to do this are provided in the GENCERT JCL itself.

2. If you have existing SSL certificates under your user ID, either delete them using *orbixhlq*.JCLLIB(DELCERT), or ensure that the certificates you are about to generate have a different name (DN) and label.

   Note: This step is optional but recommended.

3. Submit the *orbixhlq*.JCLLIB(GENCERT) JCL, to generate the following certificates that the server is to use for identification purposes when running in secure mode:

   - *orbixhlq*.CERT.USERID.CA
   - *orbixhlq*.CERT.USERID.CLNT

4. By default, the Artix Transport server demonstrations use the credentials defined to the PKCS#12 file *orbixhlq*.CERT.USERID.CLNT. To allow clients such as Artix, .NET, or WebLogic Web service clients to perform server authentication, the CA certificate that was used to generate this identity should be copied to the client system. Use FTP to copy the CA certificate *orbixhlq*.CERT.USERID.CA in ASCII format to the client system. Ensure that Orbix security policies have specified that server authentication is required.

   Note: See the *Mainframe Security Guide* for more details on security policies.

   Similarly, if the server is to perform client authentication, trusted CA certificates must be made available to the server.

Enabling the server to support SSL/TLS security

To enable the server to support SSL/TLS security:

1. Orbix Mainframe services (such as the CICS or IMS server adapters) require access to some IBM System SSL modules. You must therefore include the System SSL load library in the STEPLIB of *orbixhlq*.PROCLIB(ORXG). You can do this by removing the comment character (*) from the following line in the ORXG procedure:

   ```
   //*        DD  DISP=SHR,DSN=&SSLLOAD
   ```

   Refer to the ORXVARS include member for details on the setting of the SSLLOAD variable. By default, this is set to SYS1.SIEALNKE.

2. A TLS-enabled configuration domain can be set up, using the JCL supplied in *orbixhlq*.JCLLIB(DEPLOYT). This JCL copies the secure template configuration that is provided with your installation in *orbixhlq*.CONFIG(TLSTMPL) to *orbixhlq*.DOMAINS(TLSDOMA). It also customizes it, allowing services and applications to run securely.

   By default, a secure installation includes this configuration file. This means that *orbixhlq*.CONFIG(DEFAULT@) contains the following line:

   ```
   …
   include "//orbixhlq.CONFIG(TLSDOMA)";
   …
   ```

Note: If you have already previously deployed a secure CORBA domain, the `TLSTMPL` member should have already been copied to the *`orbixhlq`*`.DOMAINS` PDS. Therefore, you might choose to reuse the `TLSDOMA` member from your CORBA configuration domain.

Note: If you have changed the member names for the relevant configuration files, substitute the names in the preceding instruction, as appropriate.

3. Open the secure configuration domain (this is `TLSDOMA` by default) and customize the following configuration items:

`policies:well_known_addressing_policy:https:addr_list`	This specifies the hostname on which the server is running in SOAP mode, and the port number that it uses to listen for incoming secure Web service client requests.
`orb_plugins`	Ensure that this includes `"https"` (rather than `"http"`) among its settings.

Note: See the *Mainframe Security Guide* for more information on Orbix Mainframe security and for full details of all the configuration items that need to be set to enable TLS security.

CHAPTER 5

Common Tasks

This chapter provides details of topics that might be of interest to more advanced users of the Artix Transport component. These include a discussion of the different WSDL encoding styles supported and made available by the Artix Transport, and an explanation of how to perform various tasks relating to topics such as event logging and username and password checking in SOAP servers on the mainframe.

In this chapter

This chapter discusses the following topics:

Accessing WSDL Contracts

Overview

This section discusses how Orbix Mainframe servers that are being exposed as Web services publish their WSDL. It also describes how a different contract is made available for each WSDL encoding style and provides instructions on how to access these various contracts.

This section discusses the following topics:

- "WSDL contracts in Artix Transport".
- "Calling up available services".
- "Selecting a service".
- "Data encoding mechanisms".
- "Accessing a particular WSDL contract directly".

WSDL contracts in Artix Transport

The Artix Transport component publishes WSDL contracts for mainframe-based Web services. These contracts are made available from an HTTP or HTTPS endpoint directly on the Web service that is running on the mainframe.

Calling up available services

To view the WSDL contracts that are available on a server, you can call up the entire list of services available from an Internet browser, using a URL of the form `http://hostname:port/ionasoap`. (For secure domains, use a URL of the form `https://hostname:port/secureionasoap`).

Figure 12 shows an example of the output from a batch simple demonstration server.

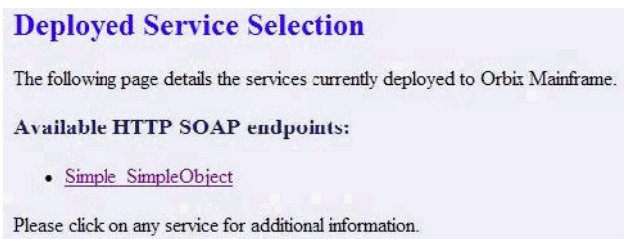

Figure 12: *Calling up available services*

Note: For IMS and CICS, hostname and port refer to the host and port on which the IMS or CICS server adapter is running. Typically, a server adapter exposes many servers.

Selecting a service

From the resulting service navigation page, you can select a service by clicking on it in the list. This in turn opens a WSDL contract selection page, as shown in Figure 13.

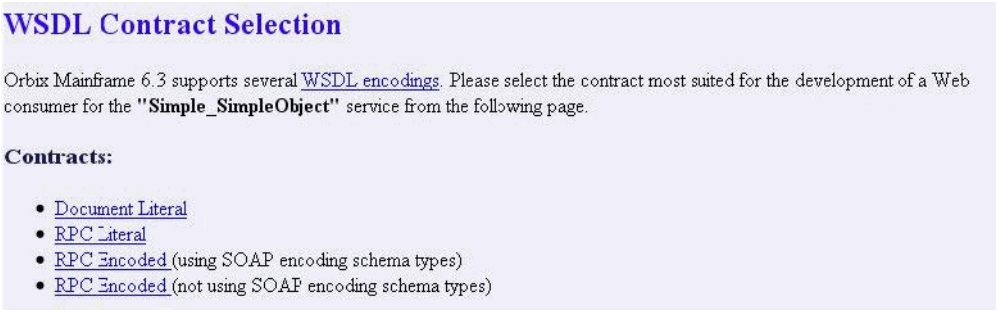

Figure 13: *Selecting a service*

From this WSDL contract selection page, you can view several variants of the WSDL contract for the selected service. These variants use different styles of WSDL encoding, and allow different client development environments to be supported, as follows:

- Wrapped document-literal encoding
- RPC-literal encoding
- RPC-Soap encoding (for .NET clients)
- RPC-Soap encoding (for Artix and WebLogic clients)

Note: See "Data encoding mechanisms" next for an overview of the different encoding styles.

Data encoding mechanisms

Because the Artix Transport component can enable an Orbix Mainframe application to support Web service client calls over SOAP, it supports various Web service encoding mechanisms for the transfer of data across the Internet. These Web service encoding mechanisms include:

- *Document literal encodings*—This is a document-oriented service that uses literal or schema-defined encoding rules to format request/response messages. Document literal is a reader-makes-right encoding. This means that the receiver is expected to use the schema definitions defined in the WSDL contract to drive encoding/decoding of parameters in request/response messages.

  There are two contract variants for this encoding: *wrapped* and *unwrapped* (or bare). The wrapped convention is generally the accepted best practice approach. The Artix Transport only supports the wrapped variant.
- *RPC literal encodings*—This is an RPC-based service that uses literal or schema-defined encoding rules to format request/response messages. RPC literal (like document literal) is a reader-makes-right encoding. This means that the receiver is expected to use the schema definitions defined in the WSDL contract to drive encoding/decoding of parameters in request/response messages.
- *RPC SOAP encodings*—This is an RPC-based service that uses the SOAP encoding rules to format request and response messages. To work around problems in various client-side SOAP stacks, there are two contract variants available for RPC SOAP encodings:
  - .NET
  - Artix/WebLogic

  It is not possible to provide a single WSDL mapping for arrays that is acceptable to all SOAP packages, using RPC-SOAP as the encoding style. Therefore, the Artix Transport exposes both .NET and Artix/WebLogic variants of RPC-SOAP-encoded WSDL.

This is necessary because a WSDL mapping for array types that works with Microsoft .NET does not necessarily work for other products, such as BEA Weblogic. See "SOAP Overview" on page 20 for more details.

Note: This encoding style is now deprecated by the WS-I, but is included here for completeness. Literal encodings are the preferred convention.

Accessing a particular WSDL contract directly

Wsdl contracts for the different encoding styles are available directly using the following URLS:

- Wrapped document-literal

  `http://host:port/ionasoap/ServiceName?wsdl=doc_literal`

  or

  `http://host:port/ionasoap/ServiceName?wsdl`
- Document-literal

  `http://host:port/ionasoap/ServiceName?wsdl=doc_literal`
- RPC-literal

  `http://host:port/ionasoap/ServiceName?wsdl=rpc_literal`
- RPC-SOAP (.NET)

  `http://host:port/ionasoap/ServiceName?wsdl=soap_encoded_arrays`
- RPC-SOAP (Artix and WebLogic)

  `http://host:port/ionasoap/ServiceName?wsdl=soap_encoded_arrays`

Enabling Logging on the Server

Overview

This section provides instructions on how to enable more detailed logging for the server and how to write this output to the server's JES2 output file and to the operator console (if required). This allows server events to be monitored.

Steps

The steps to complete this task are:

1. Open your Orbix Mainframe configuration file (by default, this is `TLSDOMA` for secure deployments or `FILEDOMA` for insecure deployments).
2. To enable logging to the server's JES2 output file, ensure that the `orb_plugins` configuration item includes `"local_log_stream"` among its settings in the configuration scope of the server, as follows:

```
…
orb_plugins = ["local_log_stream", "https", "soap"];
…
```

Note: The `"local_log_stream"` setting is included by default.

3. To enable logging to the operator console, ensure that the `orb_plugins` configuration item includes `"wto_log_stream"` among its settings in the configuration scope of the server, as follows:

```
…
orb_plugins = ["local_log_stream", "wto_log_stream",
   "https", "soap"];
…
```

Note: The "wto_log_stream" setting is not included by default, so you need to add it. Removal of the "local_log_stream" variable in this case is at the user's discretion. If both "local_log_stream" and "wto_log_stream" are specified, both logging mechanisms are used simultaneously.

If you want to modify the extent of what is logged, see "Modifying the Extent and Range of Logging" on page 114 for more details.

Modifying the Extent and Range of Logging

Overview

This section provides instructions on how to customize both the extent to which logging is to be performed (for example, only log fatal errors) and the range of system components that are to be logged (for example, only log SAF-related messages).

Steps

The steps to complete this task are:

1. Open your Orbix Mainframe configuration file (by default, this is `TLSDOMA` for secure deployments or `FILEDOMA` for insecure deployments).
2. Ensure that the `event_log:filters` configuration item in the configuration scope of the server specifies the level of logging desired. (See "Format" for details of how to set values for `event_log:filters`.)

Format

The format of the configuration settings for `event_log:filters` is as follows:

```
…
event_log:filters = ["component_to_log=events_to_log",
   "component_to_log=events_to_log",
    …]
…
```

Valid values

This is a sample range of valid values that can pertain to *`component_to_log`*:

`IT_MFA`	IMS/CICS Mainframe adapter events (for example, IMS/CICS transport configuration settings).
`IT_SAF`	SAF events (for example, SAF authorization checks)
`IT_HTTP`	HTTP events (for example, HTTP requests/responses)
`IT_HTTPS`	HTTPS events (for example, HTTPS requests/responses)
`IT_ATLI2_SOAP`	SOAP events (for example, registration of service endpoint)
`IT_ATLI2_IP`	IP events (for example, received inbound connection)
`IT_CORE`	Core events (for example, plug-in loaded)

The full range of valid values that can pertain to *events_to_log* are:

"NONE"	No events
"INFO_ALL"	All informational (non-warning, non-error) messages
"INFO_LO"	Low-priority informational (non-warning, non-error) messages
"INFO"	Low-priority informational (non-warning, non-error) messages
"INFO_HI"	High-priority informational (non-warning, non-error) messages
"INFO_MED"	Medium-priority informational (non-warning, non-error) messages
"WARN"	Warning (non-error) messages
"ERR"	Error messages
"FATAL"	Fatal error messages

Examples

The following are examples of possible settings for `event_log:filters`, based on the preceding format:

- The following indicates that all warnings and errors for all system components should be logged, and that information messages specific to the server adapter should also be logged:

```
event_log:filters = ["*=WARN+ERROR+FATAL",
   "IT_MFA=INFO_HI+WARN+ERROR+FATAL"];
```

- The following indicates that all events should be logged for all system components:

```
event_log:filters = ["*=*"];
```

- The following indicates that only errors should be logged for all system components:

```
event_log:filters = ["*=ERROR+FATAL"];
```

- The following indicates that only errors for HTTP messages should be logged:

```
event_log:filters = ["IT_HTTP=ERROR+FATAL"];
```

Enabling Logging of HTTP Requests and Responses

Overview

This section provides instructions on how to enable logging at the insecure HTTP request/response level.

Steps

The steps to complete this task are:

1. Open your Orbix Mainframe configuration file (by default, this is `TLSDOMA` for secure deployments or `FILEDOMA` for insecure deployments).
2. Ensure that the `orb_plugins` configuration item includes `"local_log_stream"` among its settings in the configuration scope of the server, as follows:

```
…
orb_plugins = ["local_log_stream", "http", "soap"];
…
```

Note: The `"local_log_stream"` setting is included by default.

3. Ensure that the `event_log:filters` configuration item (in the global configuration scope) includes `"IT_HTTP=*"` among its settings. For example:

```
…
event_log:filters = ["IT_HTTP=*"];
…
```

4. Ensure that the `policies:http:trace_requests:enabled` configuration item (in the global scope) is set to `"true"`.

Enabling Logging of HTTPS Requests and Responses

Overview

This section provides instructions on how to enable logging at the secure HTTPS request/response level.

Steps

The steps to complete this task are:

1. Open your Orbix Mainframe configuration file (by default, this is `TLSDOMA` for secure deployments or `FILEDOMA` for insecure deployments).
2. Ensure that the `orb_plugins` configuration item includes `"local_log_stream"` among its settings in the configuration scope of the server, as follows:

```
…
orb_plugins = ["local_log_stream", "https", "soap"];
…
```

Note: The `"local_log_stream"` setting is included by default.

3. Ensure that the `event_log:filters` configuration item (in the global configuration scope) includes `"IT_HTTPS=*"` among its settings. For example:

```
…
event_log:filters = ["IT_HTTPS=*"];
…
```

4. Ensure that the `policies:https:trace_requests:enabled` configuration item (in the global scope) is set to `"true"`.

Enabling User Name and Password Checking by RACF

Overview

This section provides instructions on how to configure the server to require clients to send an identifying user name and password in the SOAP header of a client request, and use on-host RACF facilities to authenticate the client's credentials.

User name and password checking can be used in combination with:

- SAF—to enforce access control.
- Client principal activation—to allow the CICS or IMS server adapter to start transactions under the client user ID rather than under its own user ID.

Steps

The steps to complete this task are:

1. If the `FACILITY` RACF class profile, `BPX.DAEMON`, is defined on your system, ensure that the caller's user ID has `READ` access to the `BPX.DAEMON` profile. Alternatively, if `BPX.DAEMON` is not defined, ensure that the caller's user ID has a UID of `0` assigned to it in the OMVS segment of its RACF user profile.
2. Open your Orbix Mainframe configuration file (by default, this is `TLSDOMA` for secure deployments or `FILEDOMA` for insecure deployments).
3. Ensure that the `plugins:soap:use_security_service` configuration item is set to `"false"` in the configuration scope of the server.
4. Ensure that the `plugins:soap:check_header_password` configuration item is set to `"true"` in the configuration scope of the server.

Enabling User Name and Password Checking by iSF

Overview

This section provides instructions on how to configure the server to require clients to send an identifying user name and password in the SOAP header of a client request, and use the off-host Security Service (iSF) to authenticate the client's credentials.

User name and password checking can be used in combination with:

- SAF—to enforce access control.
- Client principal activation—to allow the CICS or IMS server adapter to start transactions under the client user ID rather than under its own user ID.

Steps

The steps to complete this task are:

1. Open your Orbix Mainframe configuration file (by default, this is `TLSDOMA` for secure deployments or `FILEDOMA` for insecure deployments).
2. Ensure that the `plugins:soap:use_security_service` configuration item is set to `"true"` in the configuration scope of the server.
3. Ensure that the `plugins:soap:check_header_password` configuration item is set to `"true"` in the configuration scope of the server.

Enabling Client Principal Support

Overview

This section provides instructions on how to configure the CICS or IMS server adapter to be able to start CICS programs or IMS transactions under the client user ID rather than under the server adapter's own user ID.

Note: This requires that user name and password checking is also enabled, as described in "Enabling User Name and Password Checking by RACF" on page 119.

Steps

The steps to complete this task are:

1. If the `FACILITY` RACF class profile, `BPX.SERVER`, is defined on your system, ensure that the caller's user ID has `READ` access to the `BPX.SERVER` profile. Alternatively, if `BPX.SERVER` is not defined, ensure that the caller's user ID has a UID of `0` assigned to it in the OMVS segment of its RACF user profile.
2. Activate the `SURROGAT` RACF class and define profiles in it that allow the server adapter's user ID to impersonate particular users. You can do this by establishing a profile for each potential client user. For example:

```
RDEFINE SURROGAT BPX.SRV.client1 UACC(NONE)
PERMIT BPX.SRV.client1 CLASS(SURROGAT) ID(Transformer_ID)
   ACCESS(READ)
RDEFINE SURROGAT BPX.SRV.client2 UACC(NONE)
PERMIT BPX.SRV.client2 CLASS(SURROGAT) ID(Transformer_ID)
   ACCESS(READ)
```

3. Open your Orbix Mainframe configuration file (by default, this is `TLSDOMA` for secure deployments or `FILEDOMA` for insecure deployments).
4. Locate the `iona_services.imsa` scope (if you are using the IMS server adapter) or the `iona_services.cicsa` scope (if you are using the CICS server adapter).

5. If you are using the IMS server adapter, add the following configuration item:

```
plugins:imsa:use_client_principal = "yes";
```

Alternatively, if you are using the CICS server adapter, add the following configuration item:

```
plugins:cicsa:use_client_principal = "yes";
```

Note: The configuration item is not included by default in the configuration file, so you must add it if you want to activate client principal support. As an alternative to adding it in the scopes mentioned in point 4, you could add it within the global scope,

Default Type Mappings

This appendix provides a listing of the default type mappings that the Artix Transport component supports.

Note: In addition to the following table, see also the *COBOL Programmer's Guide and Reference* or *PL/I Programmer's Guide and Reference* for details of programming language limitations and supported types for the language you are using.

List of mappings

Table 1 provides a listing of the default OMG IDL-to-WSDL type mappings that the Artix Transport component supports.

Table 1: *Default IDL-to-WSDL Type Mappings (Sheet 1 of 2)*

OMG IDL Type	WSDL Type	Details
`CORBA::Long`	`xsd:Int`	-2147483648 -> 2147483647
`CORBA::ULong`	`xsd:unsignedInt`	0 -> 4294967295
`CORBA::Short`	`xsd:short`	-32768 -> 32768
`CORBA::UShort`	`xsd:unsignedShort`	0 -> 65535
`CORBA::Octet`	`xsd:unsignedByte`	0 -> 255
`CORBA::Char`	`xsd:short`	-32768 -> 32768

Table 1: *Default IDL-to-WSDL Type Mappings (Sheet 2 of 2)*

OMG IDL Type	WSDL Type	Details
`CORBA::String`	`xsd:string`	Contents restricted by UTF-8 and UTF-16 encoding
`CORBA::Wchar`	`xsd:short`	-32768 -> 32768
`CORBA::Wstring`	`xsd:string`	Contents restricted by UTF-8 and UTF-16 encoding
`CORBA::Float`	`xsd:float`	
`CORBA::Double`	`xsd:double`	
`CORBA::Boolean`	`xsd:boolean`	Valid values are: `0` = False `1` = True
`CORBA::LongLong`	`xsd:long`	-9223372036854775808 -> 9223372036854775807
`CORBA::UlongLong`	`xsd:unsignedLong`	0 -> 18446744073709551615
`CORBA::LongDouble`	`xsd:double`	
`CORBA::Fixed`	`xsd:decimal`	Constrained digits and scale
`CORBA::StringSeq`	`xsd:base64Binary`	Base-64-encoded buffer
`CORBA::OctetSeq`	`xsd:base64Binary`	Base-64-encoded buffer
CORBA arrays	`minOccurs="dim" maxOccurs="dim"`	
CORBA sequences	`minOccurs="0" maxOccurs="dim/unbounded"`	
`CORBA::Object`	Not supported	
`CORBA::TypeCode`	Not supported	
`CORBA::ValueBase`	Not supported	
CORBA unions	`xsd:choice`	

Index

T

U

W

X