

Service Architecture Leveraging Tuxedo (SALT)

Programming Guide

12c Release 2 (12.1.3)

April 2014

Copyright © 2006, 2014 Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Introduction to SALT Programming

| | |
|---|-----|
| SALT Web Services Programming | 1-1 |
| SALT Proxy Service | 1-2 |
| SALT Message Conversion | 1-2 |
| SALT Programming Tasks Quick Index | 1-2 |
| REpresentational State Transfer (REST) Message Conversion | 1-3 |

Data Type Mapping and Message Conversion

| | |
|--|------|
| Overview of Data Type Mapping and Message Conversion | 2-1 |
| Understanding SALT Message Conversion | 2-2 |
| Inbound Message Conversion | 2-2 |
| Outbound Message Conversion | 2-2 |
| Tuxedo-to-XML Data Type Mapping for Oracle Tuxedo Services | 2-3 |
| Oracle Tuxedo STRING Typed Buffers | 2-15 |
| Oracle Tuxedo CARRAY Typed Buffers | 2-16 |
| Oracle Tuxedo MBSTRING Typed Buffers | 2-18 |
| Oracle Tuxedo XML Typed Buffers | 2-19 |
| Oracle Tuxedo VIEW/VIEW32 Typed Buffers | 2-22 |
| Oracle Tuxedo FML/FML32 Typed Buffers | 2-24 |
| Oracle Tuxedo X_C_TYPE Typed Buffers | 2-28 |
| Oracle Tuxedo X_COMMON Typed Buffers | 2-29 |
| Oracle Tuxedo X_OCTET Typed Buffers | 2-29 |
| Custom Typed Buffers | 2-29 |
| XML-to-Tuxedo Data Type Mapping for External Web Services | 2-29 |
| XML Schema Built-In Simple Data Type Mapping | 2-30 |
| XML Schema User Defined Data Type Mapping | 2-33 |
| WSDL Message Mapping | 2-40 |
| REST Data Mapping | 2-42 |

| | |
|--------------------------------------|------|
| Inbound Message Conversion | 2-43 |
| Outbound Message Conversion. | 2-55 |

Web Service Client Programming

| | |
|--|------|
| Overview. | 3-1 |
| REpresentational State Transfer (REST) Support | 3-2 |
| SALT Web Service Client Programming Tips | 3-5 |
| Web Service Client Programming References | 3-10 |
| Online References | 3-10 |

Oracle Tuxedo ATMI Programming for Web Services

| | |
|--|-----|
| Overview. | 4-1 |
| Converting WSDL Model Into Oracle Tuxedo Model. | 4-2 |
| WSDL-to-Tuxedo Object Mapping. | 4-2 |
| Invoking SALT Proxy Services | 4-3 |
| SALT Supported Communication Patterns | 4-3 |
| Oracle Tuxedo Outbound Call Programming: Main Steps | 4-4 |
| Managing Error Code Returned from GWWS | 4-5 |
| Handling Fault Messages in an Oracle Tuxedo Outbound Application | 4-6 |
| See Also | 4-8 |

Using SALT Plug-Ins

| | |
|--|------|
| Understanding SALT Plug-Ins | 5-1 |
| Plug-In Elements | 5-1 |
| Programming Message Conversion Plug-ins | 5-7 |
| How Message Conversion Plug-ins Work. | 5-7 |
| When Do We Need Message Conversion Plug-in. | 5-10 |
| Developing a Message Conversion Plug-in Instance | 5-12 |
| SALT 1.1 Custom Buffer Type Conversion Plug-in Compatibility | 5-16 |

| | |
|--|------|
| Programming Outbound Authentication Plug-Ins | 5-17 |
| How Outbound Authentication Plug-Ins Work | 5-17 |
| Implementing a Credential Mapping Interface Plug-In. | 5-18 |
| Mapping the Oracle Tuxedo UID and HTTP Username. | 5-19 |

Introduction to SALT Programming

This chapter includes the following topics:

- [SALT Web Services Programming](#)

SALT Web Services Programming

SALT provides bi-directional connectivity between Oracle Tuxedo applications and Web service applications. Existing Oracle Tuxedo services can be easily exposed as Web Services without requiring additional programming tasks. SALT generates a WSDL file that describes the Oracle Tuxedo Web service contract so that any standard Web service client toolkit can be used to access Oracle Tuxedo services.

Web service applications (described using a WSDL document), can be imported as if they are standard Oracle Tuxedo services and invoked using Oracle Tuxedo ATMIs from various Oracle Tuxedo applications (for example, Oracle Tuxedo ATMI clients, ATMI servers, Jolt clients, COBOL clients, and .NET wrapper clients).

- [SALT Proxy Service](#)
- [SALT Message Conversion](#)
- [SALT Programming Tasks Quick Index](#)
- [REpresentational State Transfer \(REST\) Message Conversion](#)

SALT Proxy Service

SALT proxy services are Oracle Tuxedo service entries advertised by the GWWS SALT Gateway. The proxy services are converted from the Web service application WSDL file. Each WSDL file `wsdl:operation` object is mapped as one SALT proxy service.

The SALT proxy service is defined using the Service Metadata Repository service definition syntax. These service definitions must be loaded into the Service Metadata Repository. To invoke proxy services from an Oracle Tuxedo application, you must refer to the Oracle Tuxedo Service Metadata Repository to get the service contract description.

For more information, see [“Oracle Tuxedo ATMI Programming for Web Services”](#).

SALT Message Conversion

To support Oracle Tuxedo application and Web service application integration, the SALT gateway converts SOAP messages into Oracle Tuxedo typed buffers, and Oracle Tuxedo typed buffers into SOAP messages. The message conversion between SOAP messages and Oracle Tuxedo typed buffers is subject to a set of SALT pre-defined basic data type mapping rules.

When exposing Oracle Tuxedo services as Web services, a set of Tuxedo-to-XML data type mapping rules are defined. The message conversion process that conforms to Tuxedo-to-XML data type mapping rules is called “Inbound Message Conversion”.

When importing external Web services as SALT proxy services, a set of XML-to-Tuxedo data type mapping rules are defined. The message conversion process that conforms to XML-to-Tuxedo data type mapping rules is called “Outbound Message Conversion”.

For more information, see [“Understanding SALT Message Conversion”](#).

SALT Programming Tasks Quick Index

[Table 1-1](#) lists a quick index of SALT programming tasks. You can locate programming tasks first, and then click on the corresponding link for detailed description.

Table 1-1 SALT Programming Tasks Quick Index

| | Tasks | Refer to ... |
|--|--|--|
| Invoking Oracle Tuxedo services (inbound) through SALT | Develop Web service client programs for Oracle Tuxedo services invocation. | SALT Web Service Client Programming Tips. |
| | Understand inbound message conversion and data type mapping rules. | Understanding SALT Message Conversion. Tuxedo-to-XML Data Type Mapping for Oracle Tuxedo Services. |
| | Develop inbound message conversion plug-in. | Programming Message Conversion Plug-ins. |
| Invoking external Web services (outbound) through SALT | Understand the general outbound service programming concepts. | Oracle Tuxedo ATMI Programming for Web Services. |
| | Understand outbound message conversion and data type mapping rules. | Understanding SALT Message Conversion. XML-to-Tuxedo Data Type Mapping for External Web Services. |
| | Develop outbound message conversion plug-in. | Programming Message Conversion Plug-ins. |
| | Develop your own plug-in to map Oracle Tuxedo user name with user name for outbound HTTP basic authentication. | Programming Outbound Authentication Plug-Ins. |

REpresentational State Transfer (REST) Message Conversion

The basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods.

The REST principles around are as follows:

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

For more information, see Data Type Mapping and Message conversion, and SALT Configuration Tool in the [SALT Configuration Guide](#).

Data Type Mapping and Message Conversion

This chapter contains the following sections:

- [Overview of Data Type Mapping and Message Conversion](#)
- [Understanding SALT Message Conversion](#)
- [Tuxedo-to-XML Data Type Mapping for Oracle Tuxedo Services](#)
- [XML-to-Tuxedo Data Type Mapping for External Web Services](#)
- [REST Data Mapping](#)

Overview of Data Type Mapping and Message Conversion

SALT supports bi-directional data type mapping between WSDL messages and Oracle Tuxedo typed buffers. For each service invocation, the GWWS server converts each message between Oracle Tuxedo typed buffers and SOAP message payloads. A SOAP message payload is the XML effective data encapsulated within the `<soap:body>` element. For more information, see [“Understanding SALT Message Conversion”](#).

For native Oracle Tuxedo services, each Oracle Tuxedo buffer type is described using an XML Schema in the SALT generated WSDL document. Oracle Tuxedo service request/response buffers are represented in regular XML format. For more information, see [“Tuxedo-to-XML Data Type Mapping for Oracle Tuxedo Services”](#).

For external Web services, each WSDL message is mapped as an Oracle Tuxedo `FML32` buffer structure. An Oracle Tuxedo application invokes SALT proxy service using `FML32` buffers as

input/output. For more information see, [“XML-to-Tuxedo Data Type Mapping for External Web Services”](#).

SALT also supports non-SOAP data type mapping (i.e., REST over HTTP in both XML and JSON format. This is initiated when services are exposed as HTTP/REST services. For more information, see [REST Data Mapping](#).

Understanding SALT Message Conversion

SALT message conversion is the message transformation process between SOAP XML data and Oracle Tuxedo typed buffers. SALT introduces two message conversion rules: Inbound Message Conversion, and Outbound Message Conversion.

Inbound Message Conversion

Inbound message conversion is the SOAP XML Payload and Oracle Tuxedo typed buffer conversion process that conforms to “Tuxedo-to-XML data type mapping rules”. Inbound message conversion happens in two phases:

- When GWWS accepts SOAP requests for legacy Oracle Tuxedo services;
- When GWWS accepts response typed buffers from legacy Oracle Tuxedo services.

SALT encloses Oracle Tuxedo buffer content using elements `<inbuf>`, `<outbuf>` and/or `<errbuf>` in the SOAP message, the content included within elements `<inbuf>`, `<outbuf>` and/or `<errbuf>` is called “Inbound XML Payload”.

Outbound Message Conversion

Outbound message conversion process is the SOAP XML Payload and Oracle Tuxedo typed buffer conversion process that conforms to the “Tuxedo-to-XML data type mapping rules”. Outbound message conversion happens in two phases:

- When GWWS accepts request typed buffers sent from an Oracle Tuxedo application;
- When GWWS accepts SOAP response messages from external Web services.

[Table 2-1](#) compares inbound message conversion and outbound message conversion .

Table 2-1 Inbound Message Conversion vs. Outbound Message Conversion

| Inbound Message Conversion | Outbound Message Conversion |
|---|--|
| SOAP message payload is encapsulated with <code><inbuf></code> , <code><outbuf></code> or <code><errbuf></code> . | SOAP message payload is the entire <code><soap:body></code> |
| Transformation according to “Tuxedo-to-XML data type mapping rules”. | Transformation according to “XML-to-Tuxedo data type mapping rules”. |
| All Oracle Tuxedo buffer types are involved. | Only Oracle Tuxedo <code>FML32</code> buffer type is involved. |

Tuxedo-to-XML Data Type Mapping for Oracle Tuxedo Services

SALT provides a set of rules for describing Oracle Tuxedo typed buffers in an XML document as shown in [Table 2-2](#). These rules are exported as XML Schema definitions in SALT WSDL documents. This simplifies buffer conversion and does not require previous Oracle Tuxedo buffer type knowledge.

Table 2-2 Oracle Tuxedo Buffer Mapping to XML Schema

| Oracle Tuxedo Buffer Type | Description | XML Schema Mapping for SOAP Message |
|---------------------------|---|---|
| STRING | Oracle Tuxedo <code>STRING</code> typed buffers are used to store character strings that terminate with a <code>NULL</code> character. Oracle Tuxedo <code>STRING</code> typed buffers are self-describing. | <p data-bbox="848 423 989 444"><code>xsd:string</code></p> <p data-bbox="848 461 1233 574">In the SOAP message, the XML element that encapsulates the actual string data, must be defined using <code>xsd:string</code> directly.</p> <p data-bbox="848 591 915 612">Notes:</p> <ul data-bbox="848 630 1233 1159" style="list-style-type: none"> <li data-bbox="848 630 1233 1034">• The <code>STRING</code> data type can be specified with a max data length in the Oracle Tuxedo Service Metadata Repository. If defined in Oracle Tuxedo, the corresponding SOAP message also enforces this maximum. The GWWS server validates the actual message byte length against the definition in Oracle Tuxedo Service Metadata Repository. A SOAP fault message is returned if the message byte length exceeds supported maximums. <li data-bbox="848 1052 1233 1159">• If GWWS server receives a SOAP message other than “UTF-8”, the corresponding string value is in the same encoding. |

Table 2-2 Oracle Tuxedo Buffer Mapping to XML Schema

| Oracle Tuxedo Buffer Type | Description | XML Schema Mapping for SOAP Message |
|--|---|--|
| CARRAY (Mapping with SOAP Message plus Attachments) | Oracle Tuxedo CARRAY typed buffers store character arrays, any of which can be NULL. CARRAY buffers are used to handle data opaquely and are not self-describing. | <p>The CARRAY buffer raw data is carried within a MIME multipart/related message, which is defined in the “SOAP Messages with Attachments” specification.</p> <p>The two data formats supported for MIME Content-Type attachments are:</p> <ul style="list-style-type: none"> • application/octet-stream <ul style="list-style-type: none"> – For Apache Axis • text/xml <ul style="list-style-type: none"> – For Oracle WebLogic Server <p>The format depends on which Web service client-side toolkit is used.</p> <p>Note: The SOAP with Attachment rule is only interoperable with Oracle WebLogic Server and Apache Axis.</p> <p>Note: CARRAY data types can be specified with a max byte length. If defined in Oracle Tuxedo, the corresponding SOAP message is enforced with this limitation. The GWWS server validates the actual message byte length against the definition in the Oracle Tuxedo Service Metadata Repository.</p> |

Table 2-2 Oracle Tuxedo Buffer Mapping to XML Schema

| Oracle Tuxedo Buffer Type | Description | XML Schema Mapping for SOAP Message |
|---------------------------------------|---|--|
| CARRAY (Mapping with base64Binary) | Oracle Tuxedo CARRAY typed buffers store character arrays, any of which can be NULL. CARRAY buffers are used to handle data opaquely and are not self-describing. | <p data-bbox="848 423 1072 449"><code>xsd:base64Binary</code></p> <p data-bbox="848 465 1233 661">The CARRAY data bytes must be encoded with <code>base64Binary</code> before it can be embedded in a SOAP message. Using <code>base64Binary</code> encoding with this opaque data stream saves the original data and makes the embedded data well-formed and readable.</p> <p data-bbox="848 682 1233 791">In the SOAP message, the XML element that encapsulates the actual CARRAY data, must be defined with <code>xsd:base64Binary</code> directly.</p> <p data-bbox="848 817 1233 1107">Note: CARRAY data types can be specified with a max byte length. If defined in Oracle Tuxedo, the corresponding SOAP message is enforced with this limitation. The GWWS server validates the actual message byte length against the definition in the Oracle Tuxedo Service Metadata Repository.</p> |

Table 2-2 Oracle Tuxedo Buffer Mapping to XML Schema

| Oracle Tuxedo Buffer Type | Description | XML Schema Mapping for SOAP Message |
|---------------------------|---|---|
| MBSTRING | <p>Oracle Tuxedo MBSTRING typed buffers are used for multibyte character arrays. Oracle Tuxedo MBSTRING buffers consist of the following three elements:</p> <ul style="list-style-type: none"> • Code-set character encoding • Data length • Character array of the encoding. | <p><code>xsd:string</code></p> <p>The XML Schema built-in type, <code>xsd:string</code>, represents the corresponding type for buffer data stored in a SOAP message.</p> <p>The GWWS server only accepts “UTF-8” encoded XML documents. If the Web service client wants to access Oracle Tuxedo services with MBSTRING buffer, the mbstring payload must be represented as “UTF-8” encoding in the SOAP request message.</p> <p>Note: The GWWS server transparently passes the “UTF-8” character set string to the Oracle Tuxedo service using MBSTRING Typed buffer format. The actual Oracle Tuxedo services handles the UTF-8 string.</p> <p>For any Oracle Tuxedo response MBSTRING typed buffer (with any encoding character set), the GWWS server automatically transforms the string into “UTF-8” encoding and sends it back to the Web service client.</p> |

Table 2-2 Oracle Tuxedo Buffer Mapping to XML Schema

| Oracle Tuxedo Buffer Type | Description | XML Schema Mapping for SOAP Message |
|---------------------------|-------------|---|
| MBSTRING (cont.) | | <p>Limitation:</p> <p>Oracle Tuxedo MBSTRING data type can be specified with a max byte length in the Oracle Tuxedo Service Metadata Repository. The GWWS server checks the byte length of the converted MBSTRING buffer value.</p> <p>Note: Max byte length value is not used to enforce the character number contained in the SOAP message.</p> |

Table 2-2 Oracle Tuxedo Buffer Mapping to XML Schema

| Oracle Tuxedo Buffer Type | Description | XML Schema Mapping for SOAP Message |
|---------------------------|--|---|
| XML | Oracle Tuxedo XML typed buffers store XML documents. | <p data-bbox="784 425 938 442"><code>xsd:anyType</code></p> <p data-bbox="784 465 1166 630">The XML Schema built-in type, <code>xsd:anyType</code>, is the corresponding type for XML documents stored in a SOAP message. It allows you to encapsulate any well-formed XML data within the SOAP message.</p> <p data-bbox="784 652 901 670">Limitation:</p> <p data-bbox="784 692 1166 800">The GWWS server validates that the actual XML data is well-formed. It will not do any other enforcement validation, such as Schema validation.</p> <p data-bbox="784 822 1166 899">Only a single root XML buffer is allowed to be stored in the SOAP body; the GWWS server checks for this.</p> <p data-bbox="784 921 1166 1055">The actual XML data must be encoded using the “UTF-8” character set. Any original XML document prolog information cannot be carried within the SOAP message.</p> <p data-bbox="784 1078 1166 1185">XML data type can specify a max byte data length. If defined in Oracle Tuxedo, the corresponding SOAP message must also enforce this limitation.</p> <p data-bbox="784 1208 1166 1442">Note: The SALT WSDL generator will not have <code>xsd:maxLength</code> restrictions in the generated WSDL document, but the GWWS server will validate the byte length according to the Oracle Tuxedo Service Metadata Repository definition.</p> |
| X_C_TYPE | X_C_TYPE buffer types are equivalent to VIEW buffer types. | See VIEW/VIEW32 |

Table 2-2 Oracle Tuxedo Buffer Mapping to XML Schema

| Oracle Tuxedo Buffer Type | Description | XML Schema Mapping for SOAP Message |
|---------------------------|--|-------------------------------------|
| X_COMMON | X_COMMON buffer types are equivalent to VIEW buffer types, but are used for compatibility between COBOL and C programs. Field types should be limited to short, long, and string | See VIEW/VIEW32 |
| X_OCTET | X_OCTET buffer types are equivalent to CARRAY buffer types | See CARRAY xsd:base64Binary |

Table 2-2 Oracle Tuxedo Buffer Mapping to XML Schema

| Oracle Tuxedo Buffer Type | Description | XML Schema Mapping for SOAP Message |
|---------------------------|---|---|
| VIEW/VIEW32 | <p>Oracle Tuxedo VIEW and VIEW32 typed buffers store C structures defined by Oracle Tuxedo applications.</p> <p>VIEW structures are defined by using VIEW definition files. A VIEW buffer type can define multiple fields.</p> <p>VIEW supports the following field types:</p> <ul style="list-style-type: none"> • short • int • long • float • double • char • string • carray • bool • unsigned char • signed char • wchar_t* or wchar_t • unsigned int • unsigned long • long long • unsigned long long • long double <p>VIEW32 supports all the VIEW field types, mbstring, and embedded VIEW32 type.</p> | <p>Each VIEW or VIEW32 data type is defined as an XML Schema complex type. Each VIEW field should be one or more sub-elements of the XML Schema complex type. The name of the sub-element is the VIEW field name. The occurrence of the sub-element depends on the count attribute of the VIEW field definition. The value of the sub-element should be in the VIEW field data type corresponding XML Schema type.</p> <p>The the field types and the corresponding XML Schema type are listed as follows:</p> <ul style="list-style-type: none"> • short maps to xsd:short • int maps to xsd:int • long maps to xsd:long • float maps to xsd:float • double maps to xsd:double • char (defined as byte in Oracle Tuxedo Service Metadata Repository definition), maps to xsd:byte • char (defined as char in Oracle Tuxedo Service Metadata Repository definition) maps to xsd:string (with restrictions maxlength=1). • string maps to xsd:string • carray maps to xsd:base64Binary • mbstring maps to xsd:string |

Table 2-2 Oracle Tuxedo Buffer Mapping to XML Schema

| Oracle Tuxedo Buffer Type | Description | XML Schema Mapping for SOAP Message |
|---------------------------|-------------|--|
| VIEW/VIEW32 (cont.) | | <ul style="list-style-type: none"> • bool maps to xsd:Boolean • unsigned char maps to xsd:unsignedByte • signed char maps to xsd:byte • wchar_t* or wchar_t array maps to xsd:string • unsigned int maps to xsd:unsignedInt • unsigned long maps to xsd:unsignedLong • long long maps to xsd:long • unsigned long long maps to xsd:unsignedLong • long double maps to xsd:double. Do not set the value of C importer option size of long double to 128 bit. This option does not import successfully; use the default 64 bit • VIEW32 maps to tuxtype:view <viewname> <p>For more information, see VIEW/VIEW32 Considerations.</p> |

Table 2-2 Oracle Tuxedo Buffer Mapping to XML Schema

| Oracle Tuxedo Buffer Type | Description | XML Schema Mapping for SOAP Message |
|---------------------------|---|---|
| FML/FML32 | <p>Oracle Tuxedo FML and FML32 type buffers are proprietary Oracle Oracle Tuxedo system self-describing buffers. Each data field carries its own identifier, an occurrence number, and possibly a length indicator.</p> <p>FML supports the following field types:</p> <ul style="list-style-type: none"> • FLD_CHAR • FLD_SHORT • FLD_LONG • FLD_FLOAT • FLD_DOUBLE • FLD_STRING • FLD_CARRAY <p>FML32 supports all the FML field types and FLD_PTR, FLD_MBSTRING, FLD_FML32, and FLD_VIEW32.</p> | <p>FML/FML32 buffers can only have basic data-dictionary-like definitions for each basic field data. A particular FML/FML32 buffer definition should be applied for each FML/FML32 buffer with a different type name.</p> <p>Each FML/FML32 field should be one or more sub-elements within the FML/FML32 buffer XML Schema type. The name of the sub-element is the FML field name. The occurrence of the sub-element depends on the count and required count attribute of the FML/FML32 field definition.</p> <p>The e field types and the corresponding XML Schema type are listed below:</p> <ul style="list-style-type: none"> • short maps to xsd:short • int maps to xsd:int • long maps to xsd:long • float maps to xsd:float • double maps to xsd:double • char (defined as byte in Oracle Tuxedo Service Metadata Repository definition) maps to xsd:byte • char (defined as char in Oracle Tuxedo Service Metadata Repository definition) maps to xsd:string • string maps to xsd:string • carray maps to xsd:base64Binary • mbstring maps to xsd:string |

Table 2-2 Oracle Tuxedo Buffer Mapping to XML Schema

| Oracle Tuxedo Buffer Type | Description | XML Schema Mapping for SOAP Message |
|---------------------------|-------------|---|
| FML/FML32 (cont.) | | <ul style="list-style-type: none"> • <code>view32</code> maps to <code>tuxtype:view<viewname></code> • <code>fm132</code> maps to <code>tuxtype:fm132<svcname>_p<SeqNum></code> <p>To avoid multiple embedded FML32 buffers in an FML32 buffer, a unique sequence number (<SeqNum>) is used to distinguish the embedded FML32 buffers.</p> <p>Note: <code>ptr</code> is not supported.</p> <p>For limitations and considerations regarding mapping FML/FML32 buffers, refer to FML/FML32 Considerations.</p> |

Oracle Tuxedo STRING Typed Buffers

Oracle Tuxedo `STRING` typed buffers are used to store character strings that end with a `NULL` character. Oracle Tuxedo `STRING` typed buffers are self-describing.

[Listing 2-1](#) shows a SOAP message for the `TOUPPER` Oracle Tuxedo service example that accepts a `STRING` typed buffer.

Listing 2-1 Soap Message for a String Typed Buffer in TOUPPER Service

```
<?xml ... encoding="UTF-8" ?>
.....
<SOAP:body>
  <m:TOUPPER xmlns:m="urn:.....">
    <inbuf>abcdefg</inbuf>
  </m:TOUPPER>
</SOAP:body>
```

The XML Schema for <inbuf> is:

```
<xsd:element name="inbuf" type="xsd:string" />
```

Oracle Tuxedo CARRAY Typed Buffers

Oracle Tuxedo CARRAY typed buffers are used to store character arrays, any of which can be NULL. They are used to handle data opaquely and are not self-describing. Oracle Tuxedo CARRAY typed buffers can map to `xsd:base64Binary` or MIME attachments. The default is `xsd:base64Binary`.

Mapping Example Using base64Binary

[Listing 2-2](#) shows the SOAP message for the `TOUPPER` Oracle Tuxedo service, which accepts a CARRAY typed buffer using `base64Binary` mapping.

Listing 2-2 Soap Message for a CARRAY Typed Buffer Using base64Binary Mapping

```
<SOAP:body>
  <m:TOUPPER xmlns:m="urn:.....">
    <inbuf>QWxhZGRpbjpvYVUuIHNlc2FtZQ==</inbuf>
  </m:TOUPPER>
</SOAP:body>
```

The XML Schema for <inbuf> is:

```
<xsd:element name="inbuf" type="xsd:base64Binary" />
```

Mapping Example Using MIME Attachment

[Listing 2-3](#) shows the SOAP message for the `TOUPPER` Oracle Tuxedo service, which accepts a CARRAY typed buffer as a MIME attachment.

Listing 2-3 Soap Message for a CARRAY Typed Buffer Using MIME Attachment

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
  start="<claim061400a.xml@example.com>"
```

Content-Description: This is the optional message description.

```
--MIME_boundary
```

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

```
Content-Transfer-Encoding: 8bit
```

```
Content-ID: <claim061400a.xml@ example.com>
```

```
<?xml version='1.0' ?>
```

```
<SOAP-ENV:Envelope
```

```
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
```

```
<SOAP-ENV:Body>
```

```
..
```

```
<m:TOUPPER xmlns:m="urn:...">
```

```
<inbuf href="cid:claim061400a.carray@example.com" />
```

```
</m:TOUPPER>
```

```
..
```

```
</SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
```

```
--MIME_boundary
```

```
Content-Type: text/xml
```

```
Content-Transfer-Encoding: binary
```

```
Content-ID: <claim061400a.carray @example.com>
```

```
...binary carray data...
```

```
--MIME_boundary--
```

The WSDL for carray typed buffer will look like the following:

```
<wsdl:definitions ...>
```

```
<wsdl:types ...>
```

```
  <xsd:schema ...>
```

```
    <xsd:element name="inbuf" type="xsd:base64Binary" />
```

```
  </xsd:schema>
```

```
</wsdl:types>
```

```
.....
```

```
<wsdl:binding ...>
```

```
  <wsdl:operation name="TOUPPER">
```

```

    <soap:operation ...>
    <input>
        <mime:multipartRelated>
            <mime:part>
                <soap:body parts="..." use="..." />
            </mime:part>
            <mime:part>
                <mime:content part="..." type="text/xml"/>
            </mime:part>
        </mime:multipartRelated>
    </input>
    .....
</wsdl:operation>
</wsdl:binding>

</wsdl:definitions>

```

Oracle Tuxedo MBSTRING Typed Buffers

Oracle Tuxedo MBSTRING typed buffers are used for multibyte character arrays. Oracle Tuxedo MBSTRING typed buffers consist of the following three elements:

- code-set character encoding
- data length
- character array encoding.

Note: You cannot embed multibyte characters with non “UTF-8” code sets in the SOAP message directly.

[Listing 2-4](#) shows the SOAP message for the MBSERVICE Oracle Tuxedo service, which accepts an MBSTRING typed buffer.

Listing 2-4 SOAP Message for an MBSIRING Buffer

```

<?xml encoding="UTF-8"?>
  <SOAP:body>

```

```
<m:MBSERVICE xmlns:m="http://.....">
  <inbuf> こんにちは </inbuf>
</m:MBSERVICE>
```

The XML Schema for `<inbuf>` is:

```
<xsd:element name="inbuf" type="xsd:string" />
```

WARNING: SALT converts the Japanese character "—" (EUC-JP 0xa1bd, Shift-JIS 0x815c) into UTF-16 0x2015.

If you use another character set conversion engine, the EUC-JP or Shift-JIS multibyte output for this character may be different. For example, the Java `il8n` character conversion engine, converts this symbol to UTF-16 0x2014. The result is the also same when converting to UTF-8, which is the SALT default.

If you use another character conversion engine and Japanese "—" is included in `MBSTRING`, Oracle Tuxedo server-side `MBSTRING` auto-conversion cannot convert it back into Shift-JIS or EUC-JP.

Oracle Tuxedo XML Typed Buffers

Oracle Tuxedo XML typed buffers store XML documents.

[Listing 2-5](#) shows the Stock Quote XML document.

[Listing 2-6](#) shows the SOAP message for the `STOCKINQ` Oracle Tuxedo service, which accepts an XML typed buffer.

Listing 2-5 Stock Quote XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- "Stock Quotes". -->
<stockquotes>
  <stock_quote>
    <symbol>BEAS</symbol>
    <when>
      <date>01/27/2001</date>
      <time>3:40PM</time>
```

```
</when>
<change>+2.1875</change>
<volume>7050200</volume>
</stock_quote>
</stockquotes>
```

Then part of the SOAP message will look like [Listing 2-6](#):

Listing 2-6 SOAP Message for an XML Buffer

```
<SOAP:body>
  <m: STOCKINQ xmlns:m="urn:.....">
    <inbuf>
      <stockquotes>
        <stock_quote>
          <symbol>BEAS</symbol>
          <when>
            <date>01/27/2001</date>
            <time>3:40PM</time>
          </when>
          <change>+2.1875</change>
          <volume>7050200</volume>
        </stock_quote>
      </stockquotes>
    </inbuf>
  </m: STOCKINQ >
</SOAP:body>
```

The XML Schema for <inbuf> is:

```
<xsd:element name="inbuf" type="xsd:anyType" />
```

Note: If a default namespace is contained in an Oracle Tuxedo XML typed buffer and returned to the GWWS server, the GWWS server converts the default namespace to a regular name. Each element is then prefixed with this name.

For example, if an Oracle Tuxedo service returns a buffer having a default namespace to the GWWS server as shown in [Listing 2-7](#), the GWWS server converts the default namespace to a regular name as shown in [Listing 2-8](#).

Listing 2-7 Default Namespace Before Sending to GWWS Server

```
<Configuration xmlns="http://www.bea.com/Tuxedo/Salt/200606">
  <Servicelist id="simpapp">
    <Service name="toupper"/>
  </Servicelist>
  <Policy/>
  <System/>
  <WSGateway>
    <GWInstance id="GWWS1">
      <HTTP address="//myhost:8080"/>
    </GWInstance>
  </WSGateway>
</Configuration>
```

Listing 2-8 GWWS Server Converts Default Namespace to Regular Name

```
<dom0:Configuration
xmlns:dom0="http://www.bea.com/Tuxedo/Salt/200606">
  <dom0:Servicelist dom0:id="simpapp">
    <dom0:Service dom0:name="toupper"/>
  </dom0:Servicelist>
  <dom0:Policy></dom0:Policy>
  <dom0:System></dom0:System>
  <dom0:WSGateway>
    <dom0:GWInstance dom0:id="GWWS1">
      <dom0:HTTP dom0:address="//myhost:8080"/>
    </dom0:GWInstance>
  </dom0:WSGateway>
</dom0:Configuration>
```

Oracle Tuxedo VIEW/VIEW32 Typed Buffers

Oracle Tuxedo VIEW and VIEW32 typed buffers are used to store C structures defined by Oracle Tuxedo applications. You must define the VIEW structure with the VIEW definition files. A VIEW buffer type can define multiple fields.

Listing 2-9 shows the MYVIEW VIEW definition file.

Listing 2-10 shows the SOAP message for the MYVIEW Oracle Tuxedo service, which accepts a VIEW typed buffer.

Listing 2-9 VIEW Definition File for MYVIEW Service

```
VIEW MYVIEW
#type      cname      fbname      count      flag      size      null
float      float1     -           1          -         -         0.0
double     double1    -           1          -         -         0.0
long       long1      -           3          -         -         0
string     string1    -           2          -         20        '\0'
END
```

Listing 2-10 SOAP Message for a VIEW Typed Buffer

```
<SOAP:body>
  <m: STOCKINQ xmlns:m="http://.....">
    <inbuf>
      <float1>12.5633</float1>
      <double1>1.3522E+5</double1>
      <long1>1000</long1>
      <long1>2000</long1>
      <long1>3000</long1>
      <string1>abcd</string1>
      <string1>ubook</string1>
    </inbuf>
  </m: STOCKINQ >
</SOAP:body>
```

The XML Schema for <inbuf> is shown in [Listing 2-11](#).

Listing 2-11 XML Schema for a VIEW Typed Buffer

```
<xsd:complexType name=" view_MYVIEW">
  <xsd:sequence>
    <xsd:element name="float1" type="xsd:float" />
    <xsd:xsd:element name="double1" type="xsd:double" />
    <xsd:element name="long1" type="xsd:long" minOccurs="3" />
    <xsd:element name="string1" type="xsd:string minOccurs="3" />
  </xsd:sequence>
</xsd: complexType >
<xsd:element name="inbuf" type="tuatype:view_MYVIEW" />
```

VIEW/VIEW32 Considerations

The following considerations apply when converting Oracle Tuxedo VIEW/VIEW32 buffers to and from XML.

- You must create an environment for converting XML to and from VIEW/VIEW32. This includes setting up a VIEW directory and system VIEW definition files. These definitions are automatically loaded by the GWWS server.
- The GWWS server provides strong consistency checking between the Oracle Tuxedo Service Metadata Repository VIEW/VIEW32 parameter definition and the VIEW/VIEW32 definition file at start up.

If an inconsistency is found, the GWWS server cannot start. Inconsistency messages are printed in the ULOG file.

- tmwsdlgen also provides strong consistency checking between the Oracle Tuxedo Service Metadata Repository VIEW/VIEW32 parameter definition and the VIEW/VIEW32 definition file at start up. If an inconsistency is found, the GWWS server will not start. Inconsistency messages are printed in the ULOG file.

If the VIEW definition file cannot be loaded, tmwsdlgen attempts to use the Oracle Tuxedo Service Metadata Repository definitions to compose the WSDL document.

- Because `dec_t` is not supported, if you define `VIEW` fields with type `dec_t`, the service cannot be exported as a Web service and an error message is generated when the SALT configuration file is loading.
- Although the Oracle Tuxedo Service Metadata Repository may define a size attribute for “string/ mbstring” typed parameters (which represents the maximum byte length that is allowed in the Oracle Tuxedo typed buffer), SALT does not expose such restriction in the generated WSDL document.
- When a `VIEW32` embedded `MBString` buffer is requested and returned to the GWWS server, the GWWS miscalculates the required `MBString` length and reports that the input string exceeds the `VIEW32` `maxlength`. This is because the header is included in the transfer encoding information. You must include the header size when defining the `VIEW32` field length.
- The Oracle Tuxedo primary data type “long” is indefinite between 32-bit and 64-bit scope, depending on the platform. However, the corresponding `xsd:long` schema type is used to describe 64-bit numeric values.

If the GWWS server runs in 32-bit mode, and the Web service client sends `xsd:long` typed data that exceeds the 32-bit value range, you may get a SOAP fault.

Oracle Tuxedo FML/FML32 Typed Buffers

Oracle Tuxedo `FML` and `FML32` typed buffer are proprietary Oracle Tuxedo system self-describing buffers. Each data field carries its own identifier, an occurrence number, and possibly a length indicator.

FML Data Mapping Example

[Listing 2-12](#) shows the SOAP message for the `TRANSFER` Tuxedo service, which accepts an `FML` typed buffer.

The request fields for service `LOGIN` are:

```
ACCOUNT_ID      1      long          /* 2 occurrences, The withdrawal
account is 1st, and the deposit account is 2nd */
AMOUNT          2      float         /* The amount to transfer */
```

Part of the SOAP message is shown in [Listing 2-12](#):

Listing 2-12 SOAP Message for an FML Typed Buffer

```

<SOAP:body>
  <m:TRANSFER xmlns:m="urn:.....">
    <inbuf>
      <ACCOUNT_ID>40069901</ACCOUNT_ID>
      <ACCOUNT_ID>40069901</ACCOUNT_ID>
      <AMOUNT>200.15</AMOUNT>
    </inbuf>
  </m:TRANSFER >
</SOAP:body>

```

The XML Schema for <inbuf> is shown in [Listing 2-13](#).

Listing 2-13 XML Schema for an FML Typed Buffer

```

<xsd:complexType name=" fml_TRANSFER_In">
  <xsd:sequence>
    <xsd:element name="ACCOUNT_ID" type="xsd:long" minOccurs="2"/>
    <xsd:element name=" AMOUNT" type="xsd:float" />
  </xsd:sequence>
</xsd: complexType >
<xsd:element name="inbuf" type="tuftype: fml_TRANSFER_In" />

```

FML32 Data Mapping Example

[Listing 2-14](#) shows the SOAP message for the TRANSFER Oracle Tuxedo service, which accepts an FML32 typed buffer.

The request fields for service LOGIN are:

```

CUST_INFO          1          fml32          /* 2 occurrences, The withdrawal
customer is 1st, and the deposit customer is 2nd */
ACCOUNT_INFO      2          fml32          /* 2 occurrences, The withdrawal
account is 1st, and the deposit account is 2nd */
AMOUNT            3          float          /* The amount to transfer */

```

Each embedded CUST_INFO includes the following fields:

| | | |
|--------------|----|--------|
| CUST_NAME | 10 | string |
| CUST_ADDRESS | 11 | carray |
| CUST_PHONE | 12 | long |

Each embedded ACCOUNT_INFO includes the following fields:

| | | |
|------------|----|--------|
| ACCOUNT_ID | 20 | long |
| ACCOUNT_PW | 21 | carray |

Part of the SOAP message is shown in [Listing 2-14](#):

Listing 2-14 SOAP Message for Service with FML32 Buffer

```
<SOAP:body>
  <m:STOCKING xmlns:m="urn:.....">
    <inbuf>
      <CUST_INFO>
        <CUST_NAME>John</CUST_NAME>
        <CUST_ADDRESS>Building 15</CUST_ADDRESS>
        <CUST_PHONE>1321</CUST_PHONE>
      </CUST_INFO>
      <CUST_INFO>
        <CUST_NAME>Tom</CUST_NAME>
        <CUST_ADDRESS>Building 11</CUST_ADDRESS>
        <CUST_PHONE>1521</CUST_PHONE>
      </CUST_INFO>
      <ACCOUNT_INFO>
        <ACCOUNT_ID>40069901</ACCOUNT_ID>
        <ACCOUNT_PW>abc</ACCOUNT_PW>
      </ACCOUNT_INFO>
      <ACCOUNT_INFO>
        <ACCOUNT_ID>40069901</ACCOUNT_ID>
        <ACCOUNT_PW>zyx</ACCOUNT_PW>
      </ACCOUNT_INFO>

      <AMOUNT>200.15</AMOUNT>
    </inbuf>
```

```

    </m: STOCKINQ >
</SOAP:body>

```

The XML Schema for <inbuf> is shown in [Listing 2-15](#).

Listing 2-15 XML Schema for an FML32 Buffer

```

<xsd:complexType name="fml32_TRANSFER_In">
  <xsd:sequence>
    <xsd:element name="CUST_INFO" type="tuatype:fml32_TRANSFER_p1"
minOccurs="2" />
    <xsd:element name="ACCOUNT_INFO" type="tuatype:fml32_TRANSFER_p2"
minOccurs="2" />
    <xsd:element name="AMOUNT" type="xsd:float" />
  /xsd:sequence>
</xsd:complexType >

<xsd:complexType name="fml32_TRANSFER_p1">
  <xsd:element name="CUST_NAME" type="xsd:string" />
  <xsd:element name="CUST_ADDRESS" type="xsd:base64Binary" />
  <xsd:element name="CUST_PHONE" type="xsd:long" />
</xsd:complexType>

<xsd:complexType name="fml32_TRANSFER_p2">
  <xsd:element name="ACCOUNT_ID" type="xsd:long" />
  <xsd:element name="ACCOUNT_PW" type="xsd:base64Binary" />
</xsd:complexType>

<xsd:element name="inbuf" type="tuatype: fml32_TRANSFER_In" />

```

FML/FML32 Considerations

The following considerations apply to converting Oracle Tuxedo FML/FML32 buffers to and from XML.

- You must create an environment for converting XML to and from FML/FML32. This includes an FML field table file directory and system FML field definition files. These definitions are automatically loaded by the GWWS. FML typed buffers can be handled only if the environment is set up correctly.
- FML32 field type FLD_PTR is not supported.
- The GWWS server provides strong consistency checking between the Oracle Tuxedo Service Metadata Repository FML/FML32 parameter definition and FML/FML32 definition file during start up.

If an FML/32 field is found that is not in accordance with the environment setting, or the field table field data type definition is different from the parameter data type definition in the Oracle Tuxedo Service Metadata Repository, the GWWS cannot start. Inconsistency messages are printed in the ULOG file.

- The `tmwsdlgen` command checks for consistency between the Oracle Tuxedo Service Metadata Repository FML/FML32 parameter definition and FML/FML32 definition file. If inconsistencies are found, it issues a warning and allows inconsistencies.

If an FML/32 field is found that is not in accordance with the environment setting, or the field table field data type definition is different from the parameter data type definition in the Oracle Tuxedo Service Metadata Repository, `tmwsdlgen` attempts to use Oracle Tuxedo Service Metadata Repository definitions to compose the WSDL document.

- Although the Oracle Tuxedo Service Metadata Repository may define a size attribute for “string/mbstring” typed parameters, which represents the maximum byte length that is allowed in the Oracle Tuxedo typed buffer, SALT does not expose such restriction in the generated WSDL document.
- Oracle Tuxedo primary data type “long” is indefinite between 32-bit and 64-bit scope according to different platforms. But the corresponding `xsd:long` schema type is used to describe 64-bit numeric value. The following scenario generates a SOAP fault:

The GWWS runs in 32-bit mode, and a Web service client sends a `xsd:long` typed data which exceeds the 32-bit value range.

Oracle Tuxedo X_C_TYPE Typed Buffers

Oracle Tuxedo `X_C_TYPE` typed buffers are equivalent (and have a similar WSDL format to), Oracle Tuxedo `VIEW` typed buffers. They are transparent for SOAP clients. However, even though usage is similar to the Oracle Tuxedo `VIEW` buffer type, SALT administrators must configure the Oracle Tuxedo Service Metadata Repository for any particular Oracle Tuxedo service that uses this buffer type.

Note: All `view` related considerations also take effect for `X_C_TYPE` typed buffer.

Oracle Tuxedo X_COMMON Typed Buffers

Oracle Tuxedo `X_COMMON` typed buffers are equivalent to Oracle Tuxedo `VIEW` typed buffers. However, they are used for compatibility between COBOL and C programs. Field types should be limited to `short`, `long`, and `string`.

Oracle Tuxedo X_OCTET Typed Buffers

Oracle Tuxedo `X_OCTET` typed buffers are equivalent to `CARRAY`.

Note: Oracle Tuxedo `X_OCTET` typed buffers can only map to `xsd:base64Binary` type. SALT 1.1 does not support `MIME` attachment binding for Oracle Tuxedo `X_OCTET` typed buffers.

Custom Typed Buffers

SALT provides a plug-in mechanism that supports custom typed buffers. You can validate the SOAP message against your own XML Schema definition, allocate custom typed buffers, and parse data into the buffers and other operations.

XML Schema built-in type `xsd:anyType` is the corresponding type for XML documents stored in a SOAP message. While using custom typed buffers, you should define and represent the actual data into an XML format and transfer between the Web service client and Oracle Tuxedo Web service stack. As with XML typed buffers, only a single root XML buffer can be stored in the SOAP body. The GWWS checks this for consistency.

For more plug-in information, see [Using SALT Plug-Ins](#).

XML-to-Tuxedo Data Type Mapping for External Web Services

SALT maps each `wsdl:message` as an Oracle Tuxedo `FML32` buffer structure. SALT defines a set of rules for representing the XML Schema definition using `FML32`. To invoke external Web Services, you need to understand the exact `FML32` structure that converted from the external Web Service XML Schema definition of the corresponding message.

The following sections describe detailed WSDL message to Oracle Tuxedo `FML32` buffer mapping rules:

- [XML Schema Built-In Simple Data Type Mapping](#)

- [XML Schema User Defined Data Type Mapping](#)
- [WSDL Message Mapping](#)

XML Schema Built-In Simple Data Type Mapping

Table 2-3 shows the supported XML Schema Built-In Simple Data Type and the corresponding Oracle Tuxedo FML32 Field Data Type.

Table 2-3 Supported XML Schema Built-In Simple Data Type

| XML Schema Built-In Simple Type | Oracle Tuxedo FML32 Field Data Type | C/C++ Primitive Type In Oracle Tuxedo Program | Note |
|---------------------------------|-------------------------------------|---|---|
| xsd:byte | FLD_CHAR | char | |
| xsd:unsignedByte | FLD_UCHAR | unsigned char | |
| xsd:boolean | FLD_BOOL | char/bool | Value Pattern ['T' 'F'] |
| xsd:short | FLD_SHORT | short | |
| xsd:unsignedShort | FLD_USHORT | unsigned short | |
| xsd:int | FLD_LONG | long | |
| xsd:unsignedInt | FLD_UINT | unsigned int | |
| xsd:long | FLD_LONG | long | In a 32-bit Oracle Tuxedo program, the C primitive type long <i>cannot</i> represent all xsd:long valid value. |
| xsd:long | FLD_LLONG | long long | In a 32-bit Oracle Tuxedo program, the C primitive type long long <i>can</i> represent all xsd:long valid values. |

Table 2-3 Supported XML Schema Built-In Simple Data Type

| XML Schema Built-In Simple Type | Oracle Tuxedo FML32 Field Data Type | C/C++ Primitive Type In Oracle Tuxedo Program | Note |
|--|-------------------------------------|---|---|
| xsd:unsignedLong | FLD_LONG | unsigned long | In a 32-bit Oracle Tuxedo program, the C primitive type unsigned long <i>cannot</i> represent all xsd:long valid value. |
| xsd:unsignedLong | FLD_ULONG | unsigned long long | In a 32-bit Oracle Tuxedo program, the C primitive type unsigned long long <i>can</i> represent all xsd:unsignedLong valid values. |
| xsd:float | FLD_FLOAT | float | |
| xsd:double | FLD_DOUBLE | double | |
| xsd:string (and all xsd:string derived built-in type, such as xsd:token, xsd:Name, etc.) | FLD_STRING FLD_MBSTRING | char [] wchar_t [] (Null-terminated string) | xsd:string can be optionally mapped as FLD_STRING or FLD_MBSTRING using wsdlcv . |
| xsd:base64Binary | FLD_CARRAY | char [] | |
| xsd:hexBinary | FLD_CARRAY | char [] | |
| All other built-in data types (Data / Time related, decimal / Integer related, any URL, QName, NOTATION) | FLD_STRING | char [] | You should comply with the value pattern of the corresponding XML built-in data type. Otherwise, server-side Web service will reject the request. |

The following samples demonstrate how to prepare data in a Oracle Tuxedo program for XML Schema Built-In Simple Types.

- [XML Schema Built-In Type Sample - xsd:string](#)
- [XML Schema Built-In Type Sample - xsd:hexBinary](#)
- [XML Schema Built-In Type Sample - xsd:date](#)

Table 2-4 XML Schema Built-In Type Sample - xsd:string

| XML Schema Definition | | | | |
|---|----------------|-----------------|------------|----------------|
| <code><xsd:element name="message" type="xsd:string" /></code> | | | | |
| Corresponding FML32 Field Definition (FLD_MBSTRING) | | | | |
| # | Field_name | Field_type | Field_flag | Field_comments |
| | message | mbstring | - | |
| C Pseudo Code | | | | |
| <pre> FBFR32 * request; FLDLEN32 len, mbsize = 1024; char * msg, * mbmsg; msg = calloc(...); mbmsg = malloc(mbsize); ... strncpy(msg, "...", len); /* The string is UTF-8 encoding */ Fmbpack32("utf-8", msg, len, mbmsg, &mbsize, 0); /* prepare mbstring*/ Fadd32(request, message, mbmsg, mbsize); </pre> | | | | |

Table 2-5 XML Schema Built-In Type Sample - xsd:hexBinary

| XML Schema Definition | | | | |
|---|---------------------|---------------|------------|----------------|
| <code><xsd:element name="mem_snapshot" type="xsd:hexBinary" /></code> | | | | |
| Corresponding FML32 Field Definition (FLD_MBSTRING) | | | | |
| # | Field_name | Field_type | Field_flag | Field_comments |
| | mem_snapshot | carray | - | |

Table 2-5 XML Schema Built-In Type Sample - xsd:hexBinary

| C Pseudo Code |
|---|
| <pre> FBFR32 * request; FLDLLEN32 len; char * buf; buf = calloc(...); ... memcpy(buf, "...", len); /* copy the original memory */ Fadd32(request, mem_snapshot, buf, len); </pre> |

Table 2-6 XML Schema Built-In Type Sample - xsd:date

| XML Schema Definition | | | | | | | | | | |
|--|------------------|---------------|------------|----------------|----------------|--|------------------|---------------|---|--|
| <pre> <xsd:element name="IssueDate" type="xsd:date" /> </pre> | | | | | | | | | | |
| Corresponding FML32 Field Definition (FLD_STRING) | | | | | | | | | | |
| <table border="1"> <thead> <tr> <th>#</th> <th>Field_name</th> <th>Field_type</th> <th>Field_flag</th> <th>Field_comments</th> </tr> </thead> <tbody> <tr> <td></td> <td>IssueDate</td> <td>string</td> <td>-</td> <td></td> </tr> </tbody> </table> | # | Field_name | Field_type | Field_flag | Field_comments | | IssueDate | string | - | |
| # | Field_name | Field_type | Field_flag | Field_comments | | | | | | |
| | IssueDate | string | - | | | | | | | |
| C Pseudo Code | | | | | | | | | | |
| <pre> FBFR32 * request; char date[32]; ... strcpy(date, "2007-06-04+8:00"); /* Set the date value correctly */ Fadd32(request, IssueDate, date, 0); </pre> | | | | | | | | | | |

XML Schema User Defined Data Type Mapping

Table 2-7 lists the supported XML Schema User Defined Simple Data Type and the corresponding Oracle Tuxedo FML32 Field Data Type.

Table 2-7 Supported XML Schema User Defined Data Type

| XML Schema User Defined Data Type | Oracle Tuxedo FML32 Field Data Type | C/C++ Primitive Type In Oracle Tuxedo Program | Note |
|--|--|---|---|
| <xsd:anyType> | FLD_MBSTRING | char [] | You should prepare entire XML document enclosing with the element tag. |
| <xsd:simpleType> derived from built-in primitive simple data types | Equivalent FML32 Field Type of the primitive simple type (see Table 2-3) | Equivalent C Primitive Data Type of the primitive simple type (see Table 2-3) | Facets defined with <xsd:restriction> are not enforced in Oracle Tuxedo. |
| <xsd:simpleType> defined with <xsd:list> | FLD_MBSTRING | char [] | Same as <xsd:anyType>. The Schema compliancy is not enforced in Oracle Tuxedo.. |
| <xsd:simpleType> defined with <xsd:union> | FLD_MBSTRING | char [] | Same as <xsd:anyType>. The Schema compliancy is not enforced in Oracle Tuxedo.. |
| <xsd:complexType> defined with <xsd:simpleContent> | FLD_MBSTRING | char [] | Same as <xsd:anyType>. The Schema compliancy is not enforced in Oracle Tuxedo.. |
| <xsd:complexType> defined with <xsd:complexContent> | FLD_MBSTRING | char [] | Same as <xsd:anyType>. The Schema compliancy is not enforced in Oracle Tuxedo.. |

Table 2-7 Supported XML Schema User Defined Data Type

| XML Schema User Defined Data Type | Oracle Tuxedo FML32 Field Data Type | C/C++ Primitive Type In Oracle Tuxedo Program | Note |
|---|-------------------------------------|---|---|
| <xsd:complexType> defined with shorthand <xsd:complexContent>, sub-elements composited with sequence or all | FLD_FML32 | FBFR32 * embedded fml32 buffer | Each sub-element of the complex type is defined as an embedded FML32 field. |
| <xsd:complexType> defined with shorthand <xsd:complexContent>, sub-elements composited with choice | FML_FML32 | FBFR32 * embedded fml32 buffer | Each sub-element of the complex type is defined as an embedded FML32 field. You should only add one sub field into the fml32 buffer. |
| <xsd:complexType> with sub-elements composited with sequence. The complexType can contain attribute and elements. | FLD_FML32 | FBFR32 * embedded fml32 buffer | Each sub-element of the complex type is defined as an embedded FML32 field. |

The following samples demonstrate how to prepare data in an Oracle Tuxedo program for XML Schema User Defined Data Types:

- [XML Schema User Defined Type Sample - xsd:simpleType Derived from Primitive Simple Type](#)
- [XML Schema User Defined Type Sample - xsd:simpleType Defined with xsd:list](#)
- [External Service Schema Attribute Use Example](#)

Table 2-8 XML Schema User Defined Type Sample - xsd:simpleType Derived from Primitive Simple Type

| XML Schema Definition | | | | |
|---|--------------|---------------|------------|----------------|
| <pre><xsd:element name="Grade" type="Alphabet" /> <xsd:simpleType name="Alphabet"> <xsd:restriction base="xsd:string"> <xsd:maxLength value="1" /> <xsd:pattern value="[A-Z]" /> </xsd:restriction> </xsd:simpleType></pre> | | | | |
| Corresponding FML32 Field Definition (FLD_STRING) | | | | |
| # | Field_name | Field_type | Field_flag | Field_comments |
| | Grade | string | - | |
| C Pseudo Code | | | | |
| <pre>char grade[2]; FBFR32 * request; ... grade[0] = 'A'; grade[1] = '\0'; Fadd32(request, Grade, (char *)grade, 0);</pre> | | | | |

Table 2-9 XML Schema User Defined Type Sample - xsd:simpleType Defined with xsd:list

| XML Schema Definition (Target Namespace "urn:sample.org") | | | | |
|--|--------------|-----------------|------------|----------------|
| <pre><xsd:element name="Users" type="namelist" /> <xsd:simpleType name="namelist"> <xsd:list itemType="xsd:NMTOKEN"> </xsd:simpleType></pre> | | | | |
| Corresponding FML32 Field Definition (FLD_MBSTRING) | | | | |
| # | Field_name | Field_type | Field_flag | Field_comments |
| | Users | mbstring | - | |

Table 2-9 XML Schema User Defined Type Sample - xsd:simpleType Defined with xsd:list

C Pseudo Code

```

char * user[5];
char users[...];
char * mbpacked;
FLDLEN32 mbsize = 1024;
FBFR32 * request;
...
sprintf(users, "<n1:Users xmlns:n1=\"urn:sample.org\">");
for ( i = 0 ; i < 5 ; i++ ) {
    strcat(users, user[i]);
    strcat(users, " ");
}
strcat(users, "</n1:Users>");
...
mbpacked = malloc(mbsize);
/* prepare mbstring*/
Fmbpack32("utf-8", users, strlen(users), mbpacked, &mbsize, 0);
Fadd32( request, Users, mbpacked, mbsize);

```

Note: In [Table 2-10](#), attributes are supported in External Web Services calls using the form "<xs:attribute name="[name]" type="[type]" />" only. Qualifiers such as "fixed=" are currently not supported."

Table 2-10 External Service Schema Attribute Use Example

| XML Schema Definition | | | | | |
|--|------------|--------|-------|-------|----------------------------------|
| <pre> <xs:element name="add"> <xs:complexType> <xs:sequence> <xs:element name="param0" nillable="true" type="xs:int"/> <xs:element name="param1" nillable="true" type="xs:int"/> </xs:sequence> <xs:attribute name="aType" type="xs:string"/> </xs:complexType> </xs:element> </pre> | | | | | |
| Corresponding FML32 Field Definition | | | | | |
| ... | | | | | |
| #name | rel-number | | type | flags | comment |
| #---- | ----- | ---- | ----- | ----- | |
| add | 1 | fml32 | - | | fullname=add, schema=axis2:add |
| aType | 3 | string | - | | fullname=aType, schema=xs:string |
| param0 | 4 | long | - | | fullname=param0, schema=xs:int |
| param1 | 5 | long | - | | fullname=param1, schema=xs:int |
| ... | | | | | |
| Corresponding SALT Metadata Repository Definition | | | | | |

Table 2-10 External Service Schema Attribute Use Example

```

...
servicemode=webservice
inbuf=FML32
outbuf=FML32
errbuf=FML32
    param=add
    access=in
    paramschema=XSD_E:add@http://calc.sample
    type=fml32
    (
        param=param0
        access=in
        paramschema=XSD_E:param0@http://calc.sample
        type=long
        primetype=int

        param=param1
        access=in
        paramschema=XSD_E:param1@http://calc.sample
        type=long
        primetype=int

        param=aType
        access=in
        paramschema=XSD_E:attribute:aType@http://calc.sample
        type=string
        primetype=string
    )

```

Table 2-10 External Service Schema Attribute Use Example

Corresponding Sample Pseudo code

```
...
    FBFR32 *f, *fin;
    long len;
    FLDLEN32 len2;
    long inputnum1, inputnum2;
    char ret_val[25];
    char ret_attr[25];
    char *programName;
    int counter;
...
    char addType[25];
    strcpy(addType, argv[1]);
    Fadd32(fin, aType, addType, 0);
    inputnum1 = atoi(argv[2]);
    Fadd32(fin, param0, (char *)&inputnum1, 0);
    inputnum1 = atoi(argv[2]);
    Fadd32(fin, param0, (char *)&inputnum1, 0);
    Fadd32(f, add, (char *)fin, 0)
    tpcall("add", (char *)f, 0, (char **)&f, &len, TPSIGRSTRT)
...

```

WSDL Message Mapping

Oracle Tuxedo FML32 buffer type is always used in mapping WSDL messages.

[Table 2-11](#) lists the WSDL message mapping rules defined by SALT.

Table 2-11 WSDL Message Mapping Rules

| WSDL Message Definition | Oracle Tuxedo Buffer/Field Definition | Note |
|--|---|--|
| <wsdl:input> message | Oracle Tuxedo Request Buffer (Input buffer) | |
| <wsdl:output> message | Oracle Tuxedo Response Buffer with TPSUCCESS (Output buffer) | |
| <wsdl:fault> message | Oracle Tuxedo Response Buffer with TPFFAIL (error buffer) | |
| Each message part defined in <wsdl:input> or <wsdl:output> | Mapped as top level field in the Oracle Tuxedo FML32 buffer. Field type is the equivalent FML32 field type of the message part XML data type. (See Table 2-3 and Table 2-7) | |
| <faultcode> in SOAP 1.1 fault message | Mapped as a fixed top level FLD_STRING field (faultcode) in the Oracle Tuxedo error buffer: faultcode string - - | This mapping rule applies for SOAP 1.1 only. |
| <faultstring> in SOAP 1.1 fault message | Mapped as a fixed top level FLD_STRING field (faultstring) in the Oracle Tuxedo error buffer: faultstring string - - | This mapping rule applies for SOAP 1.1 only. |
| <faultactor> in SOAP 1.1 fault message | Mapped as a fixed top level FLD_STRING field (faultactor) in the Oracle Tuxedo error buffer: faultactor string - - | This mapping rule applies for SOAP 1.1 only. |
| <Code> in SOAP 1.2 fault message | Mapped as a fixed top level FLD_FML32 field (Code) in the Oracle Tuxedo error buffer, which containing two fixed sub FLD_STRING fields (Value and Subcode): Code fml32 - - Value string - - Subcode string - - | This mapping rule applies for SOAP 1.2 only. |

Table 2-11 WSDL Message Mapping Rules

| WSDL Message Definition | Oracle Tuxedo Buffer/Field Definition | Note |
|--|--|---|
| <Reason> in SOAP 1.2 fault message | Mapped as a fixed top level FLD_FML32 field (Reason) in the Oracle Tuxedo error buffer, which containing zero or more fixed sub FLD_STRING field (Text): Reason fml32 - - Text string - - | This mapping rule applies for SOAP 1.2 only. |
| <Node> in SOAP 1.2 fault message | Mapped as a fixed top level FLD_STRING field (Node) in the Oracle Tuxedo error buffer: Node string - - | This mapping rule applies for SOAP 1.2 only. |
| <Role> in SOAP 1.2 fault message | Mapped as a fixed top level FLD_STRING field (Role) in the Oracle Tuxedo error buffer: Role string - - | This mapping rule applies for SOAP 1.2 only. |
| <detail> in SOAP fault message | Mapped as a fixed top level FLD_FML32 field in the Oracle Tuxedo error buffer: detail fml32 - - | This mapping rule applies for both SOAP 1.1 and SOAP 1.2. |
| Each message part defined in <wsdl: fault> | Mapped as a sub field of “detail” field in the Oracle Tuxedo FML32 buffer. Field type is the equivalent FML32 field type of the message part XML data type. (See Table 2-3 and Table 2-7) | This mapping rule applies for both SOAP 1.1 and SOAP 1.2. |

REST Data Mapping

This section contains the following topics:

- [Inbound Message Conversion](#)
- [Outbound Message Conversion](#)

Note: If a VIEW32 buffer is used as input of an Oracle Tuxedo service exposed as a RESTful service using GET or DELETE, and that VIEW32 contains an MBSTRING type, some content must be specified in the calling query string as MBSTRING type fields cannot be defaulted.

If not, the call results in an HTTP 500 error, with `TPEINVAL` being returned with the following `ULOG` message:

```
...
181356.hostname!server.5535.451673280.0: GP_CAT:1582: ERROR:
Input codeset encoding argument not defined
...
```

Inbound Message Conversion

This section contains the following topics:

- [Query String Mapping](#)
- [JSON Data Mapping](#)
- [XML Data Mapping](#)

Query String Mapping

For `GET` and `DELETE` methods, input data is passed as an HTTP query string.

Data passed as query string can be mapped within the limitations of query string representation:

- `keyword=value` model, when applicable. For simple buffer types the actual data may be passed directly, e.g.: `http://host:1234/myTOUPPER?inputstring`
- No nesting possibly of keyword/value pairs.
- Encoding must be performed for some characters (space for instance).
- Limited amount of data. While GWWS does not impose any limit, the browser or client toolkit may.

The mapping is described below for the different types of buffers supported by Oracle Tuxedo.

Table 2-12 Query String Mapping

| Tuxedo Buffer Type | Query String Mapping | Notes |
|--------------------|--|---|
| STRING | <code>http://host:port/service?data</code> | Data as is, possibly URL encoded, GWWS performs the decoding. |
| CARRAY | <code>http://host:port/service?data</code> | Data represented as base64 encoded string. |

Table 2-12 Query String Mapping

| Tuxedo Buffer Type | Query String Mapping | Notes |
|--------------------|-------------------------------|--|
| MBSTRING | http://host:port/service?data | Data represented as URL encoded of UTF-8 representation of the Oracle Tuxedo MBSTRING. |
| XML | http://host:port/service?data | XML fragment as is, URL encoded. |
| X_C_TYPE | Same as VIEW/VIEW32 | |
| X_COMMON | Same as VIEW/VIEW32 | |
| X_OCTET | Same as CARRAY | |

Table 2-12 Query String Mapping

| Tuxedo Buffer Type | Query String Mapping | Notes |
|--------------------|--|---|
| VIEW/VIEW32 | <p>http://host:port/service?value1&value2 or http://host:port/service?fieldname1=value1&fieldname2=value2</p> | <p>Actual values are converted from URL encoded string representations to their native types.</p> <p>GWWS attempts to convert values to the corresponding VIEW/VIEW32 member depending on the target type: number types from their string representation to their Oracle Tuxedo ones:</p> <ul style="list-style-type: none"> • float notation for float and double VIEW/VIEW32 types • integer notation for int, long and other integer based types <p>FLD_CHAR fields are translated from URL-encoded content(i.e., representable characters or their '%xx' representation string for all other types)</p> <p>The <code>fieldname=value</code> notation is used with:</p> <ul style="list-style-type: none"> • FBNAME field name when configured in the view description. • CNAME value when no FBNAME is present in the view description. <p>If neither FBNAME nor CNAME matches for this subtype, a mapping error is returned.</p> |

Table 2-12 Query String Mapping

| Tuxedo Buffer Type | Query String Mapping | Notes |
|--------------------|--|--|
| FML/FML32 | <p>http://host:port/service?fieldname1=value1&fieldname2=value2</p> <p>or, for multiple occurrences:</p> <p>http://host:port/service?fieldname1=value1&fieldname1=value2</p> | <p>Actual values are converted from URL encoded string representations to their native types.</p> <p>GWWS attempts to convert values to the corresponding VIEWFML/VIEWFML32 member depending on the target type: number types from their string representation to their Oracle Tuxedo ones:</p> <ul style="list-style-type: none"> float notation for float and double VIEWFML/VIEWFML32 types integer notation for int, long and other integer-based types FLD_CHAR fields are translated from URL-encoded content (i.e., representable characters or their '%xx' representation string for all other types |

JSON Data Mapping

The different Oracle Tuxedo buffer types are converted into/from JSON as shown in [Table 2-13](#).

Table 2-13 JSON Data Mapping

| Oracle Tuxedo Buffer Type | JSON equivalent/example | Notes |
|---------------------------|-------------------------|--|
| STRING | <buffer content> | |
| CARRAY | <binary buffer content> | |
| MBSTRING | <Multi-byte string> | In order to transmit encodings other than UTF-8, the "enableMultiEncoding" property must be set to "true" in the SALTDEPLOY configuration. |

Table 2-13 JSON Data Mapping

| Oracle Tuxedo Buffer Type | JSON equivalent/example | Notes |
|---------------------------|---|--|
| XML | <XML fragment as-is> | In order to transmit encodings other than UTF-8, the "enableMultiEncoding" property must be set to "true" in the SALTDEPLOY configuration. |
| X_C_TYPE | Same as VIEW/VIEW32 | |
| X_COMMON | Same as VIEW/VIEW32 | |
| X_OCTET | Same as CARRAY | |
| VIEW/VIEW32 | <pre>{'<fieldname>': '<fieldcontent>', '<fieldname>': '<fieldcontent>'}</pre> <p>possibly nested</p> <pre>{'<fieldname>': {'<fieldname>': '<fieldcontent>'}}</pre> <p>JSON has the following primitive types:</p> <ul style="list-style-type: none"> • boolean (true/false) • Number (int or double float) • String <p>VIEW/VIEW32 field types are mapped as follows (Oracle Tuxedo type: JSON type):</p> | <p>See VIEW/VIEW32 considerations and examples for fieldname mapping details.</p> <p>Some types may be truncated if represented in their primitive types (long long, long double), in that case they are rendered as JSON strings.</p> |

Table 2-13 JSON Data Mapping

| Oracle Tuxedo Buffer Type | JSON equivalent/example | Notes |
|------------------------------|---|-------|
| | <ul style="list-style-type: none"> • short: Number • int: Number • long: Number • float: Number • double: Number • char: String • string: String • carray: String (base64 encoded) • bool: boolean • unsigned char: String • signed char: String • wchar_t* or wchar_t: String • unsigned int: Number • unsigned long: Number • long long: String (See notes below table) • unsigned long long: String (See notes) • long double: String (See notes below table) • mbstring: String • view32: nested JSON record | |

Table 2-13 JSON Data Mapping

| Oracle Tuxedo Buffer Type | JSON equivalent/example | Notes |
|---------------------------|--|--|
| FML/FML32 | <pre data-bbox="404 418 753 531">'<fieldname>': '<fieldcontent>', '<fieldname>': '<fieldcontent>'}</pre> <p data-bbox="404 545 689 569">possibly nested, FML32 only:</p> <pre data-bbox="404 583 753 638">'<fieldname>': {'<fieldname>': '<fieldcontent>'}}</pre> <p data-bbox="404 656 753 739">FML/FML32 field types are mapped as follows (Oracle Tuxedo type: JSON type):</p> <ul data-bbox="404 753 753 1260" style="list-style-type: none"> • FLD_SHORT: Number • FLD_LONG: Number • FLD_FLOAT: Number • FLD_DOUBLE: Number • FLD_CHAR: String or character 'T' for JSON true or 'F' for JSON false • FLD_STRING: String • FLD_CARRRAY: String (base64 encoded) • FLD_MBSTRING: String • FLD_VIEW32: JSON nested record, see VIEW/VIEW32 mapping for individual types • FLD_FML32: JSON object | <p data-bbox="787 418 1214 501">Nested FLD_VIEW32: the name of the view subtype must be the name of the embedded VIEW32. For Example:</p> <p data-bbox="787 517 1120 541">VIEW32 example.v definition file:</p> <pre data-bbox="787 557 1214 765">VIEW v32example char flag1-1- - - string str-1100 - - ... JSON content (EVIEW32 is a FLD_VIEW32 fml32 type):</pre> <pre data-bbox="787 847 1214 1052">{"EVIEW32" : {"v32example": {"flag1": "x", "str": "somestring"} } }</pre> |

Notes: Non-structured buffer types (STRING, CARRAY, X_OCTET and MBSTRING) will not wrap data as JSON objects, the data is transmitted as is.

JSON internally handles all floating point types differently than XML. XML conversion floating point conversion may incur some precision loss over similar JSON conversions. This is currently a limitation.

VIEW/VIEW32 Considerations

The following considerations apply when converting Oracle Tuxedo VIEW/VIEW32 buffers to and from XML:

- You must create an environment for converting XML to and from VIEW/VIEW32. This includes setting up a VIEW directory and system VIEW definition files. These definitions are automatically loaded by the GWWS server.

FML/FML32 Considerations

The following considerations apply to converting Oracle Tuxedo FML/FML32 buffers to and from XML:

- You must create an environment for converting XML to and from FML/FML32. This includes an FML field table file directory and system FML field definition files. These definitions are automatically loaded by the GWWS. FML typed buffers can be handled only if the environment is set up correctly.

Note: FML32 Field type FLD_PTR is not supported.

XML Data Mapping

XML data mapping is performed using similar rules as the mapping used in SOAP mode.

The following differences are to be noted:

- Floating point numbers without decimal value get represented as integers, for example: 10.0 is printed as 10. This is currently a limitation.
- No namespaces are generated or processed, since REST mode does not use interfaces.
- Simple buffers (STRING, CARRAY, MBSTRING and XML) are sent and received as is, without any XML processing. The behavior is identical to JSON processing (i.e., no mapping is necessary).
- FML and FML32 requests are wrapped by a root element (which name is ignored, as long as the XML is formed properly), and replies are wrapped in an element with the same name as the subtype as specified in the REST/Service/Method/@inputbuffer attribute of the SALTDEPLOY configuration file, or <root> element, since there is not necessarily one if subtype is not configured. VIEW, VIEW32, X_COMMON and X_C_TYPE buffers are the subtype name as root element name.

The different Oracle Tuxedo buffer types are converted into/from XML as shown in [Table 2-14](#).

Table 2-14 XML Data Mapping

| Tuxedo Buffer Type | Description | REST XML Mapping Example |
|--------------------|--|--|
| STRING | Oracle Tuxedo <code>STRING</code> typed buffers are used to store character strings that terminate with a <code>NULL</code> character. Oracle Tuxedo <code>STRING</code> typed buffers are self-describing. | HELLO WORLD! |
| CARRAY | Oracle Tuxedo <code>CARRAY</code> typed buffers store character arrays, any of which can be <code>NULL</code> . <code>CARRAY</code> buffers are used to handle data opaquely and are not self-describing. | Binary content |
| MBSTRING | <p>Oracle Tuxedo <code>MBSTRING</code> typed buffers are used for multibyte character arrays. Oracle Tuxedo <code>MBSTRING</code> buffers consist of the following three elements:</p> <ul style="list-style-type: none"> • Code-set character encoding • Data length • Character array of the encoding. <p>In order to transmit encodings other than UTF-8, the <code>"enableMultiEncoding"</code> property must be set to <code>"true"</code> in the <code>SALTDEPLOY</code> configuration.</p> | Multi-byte string encoded according to Content-Type setting. |

Table 2-14 XML Data Mapping

| Tuxedo Buffer Type | Description | REST XML Mapping Example |
|--------------------|---|--------------------------|
| XML | <p>Oracle Tuxedo XML typed buffers store XML documents.</p> <p>The GWWS server validates that the actual XML data is well-formed. It will not do any other enforcement validation, such as Schema validation.</p> <p>Only a single root XML buffer is allowed to be stored in the payload; the GWWS server checks for this.</p> <p>Any original XML document prologue information cannot be carried within the payload.</p> <p>In order to transmit encodings other than UTF-8, the "enableMultiEncoding" property must be set to "true" in the SALTDEPLOY configuration.</p> | XML fragment as is |
| X_C_TYPE | Same as VIEW/VIEW32 | |
| X_COMMON | Same as VIEW/VIEW32 | |
| X_OCTET | Same as CARRAY | |

Table 2-14 XML Data Mapping

| Tuxedo Buffer Type | Description | REST XML Mapping Example |
|--------------------|--|---|
| VIEW/VIEW32 | <p>Oracle Tuxedo VIEW and VIEW32 typed buffers store C structures defined by Oracle Tuxedo applications.</p> <p>VIEW structures are defined by using VIEW definition files. A VIEW buffer type can define multiple fields.</p> <p>VIEW supports the following field types:</p> <ul style="list-style-type: none"> • short • int • long • float • double • char • string • carray (represented as base64 encoded content) • bool • unsigned char • signed char • wchar_t* or wchar_t • unsigned int • unsigned long • long long • unsigned long long • long double <p>VIEW32 supports all the VIEW field types, mbstring, and embedded VIEW32 type.</p> <p>The name of the sub-element is the VIEW field name. The occurrence of the sub-element depends on the count attribute of the VIEW field definition. The value of the sub-element should be in the VIEW field data type corresponding XML Schema type.</p> | <pre><VIEW> <viewfieldname> fieldcontent </viewfieldname> </VIEW></pre> |

Table 2-14 XML Data Mapping

| Tuxedo Buffer Type | Description | REST XML Mapping Example |
|--------------------|---|--|
| FML/FML32 | <p>Oracle Tuxedo FML and FML32 type buffers are proprietary Oracle Oracle Tuxedo system self-describing buffers. Each data field carries its own identifier, an occurrence number, and possibly a length indicator.</p> <p>FML supports the following field types:</p> <ul style="list-style-type: none"> • FLD_CHAR • FLD_SHORT • FLD_LONG • FLD_FLOAT • FLD_DOUBLE • FLD_STRING • FLD_CARRAY (as base64 encoded content) <p>FML32 supports all the FML field types and FLD_PTR, FLD_MBSTRING, FLD_FML32, and FLD_VIEW32.</p> | <p>Nested FLD_VIEW32: the name of the view subtype must be the name of the embedded VIEW32. For Example:</p> <pre>VIEW32 example.v definition file: VIEW v32example char flag1-1- - - string str-1100 - - ... XML content (EVIEW32 is a FLD_VIEW32 fml32 type): <EVIEW32> <v32example> <flag1>x</flag1> <str>somestring</str> </v32example> </EVIEW32></pre> |

Note: Non-structured buffer types (STRING, CARRAY, X_OCTET and MBSTRING) do not wrap data as XML objects, the data is transmitted as is.

VIEW/VIEW32 Considerations:

The following considerations apply when converting Oracle Tuxedo VIEW/VIEW32 buffers to and from XML:

- You must create an environment for converting XML to and from VIEW/VIEW32. This includes setting up a VIEW directory and system VIEW definition files. These definitions are automatically loaded by the GWWS server.

FML/FML32 Considerations

The following considerations apply to converting Oracle Tuxedo FML/FML32 buffers to and from XML:

- You must create an environment for converting XML to and from FML/FML32. This includes an FML field table file directory and system FML field definition files. These definitions are automatically loaded by the GWWS. FML typed buffers can be handled only if the environment is set up correctly.

Note: FML32 Field type `FLD_PTR` is not supported.

Outbound Message Conversion

This section contains the following topics:

- [Query String Mapping](#)
- [JSON Data Mapping](#)
- [XML Data Mapping](#)

Query String Mapping

Note: Attempting to use embedded FML32 and VIEW32 fields will result in a TPEPROTO error in this mode.

For GET and DELETE methods, requested data is passed as an HTTP query string. For example: `http://host:1234/banking?account=1234`

Data passed as query string can be mapped within the limitations of query string representation:

- `keyword=value` model, when applicable. For simple buffer types the actual data may be passed directly (e.g., `http://host:1234/svc?inputstring`).
- No nesting of keyword/value pairs.
- Encoding must be performed for some characters (“space” for instance).
- Limited amount of data. While GWWS does not impose any limit, the browser or client toolkit may.

The mapping is as described in [Table 2-15](#) for different types of buffers supported by OracleTuxedo.

Table 2-15 Query String Mapping

| Tuxedo Buffer Type | Query String Mapping | Notes |
|--------------------|----------------------------|---|
| STRING | http://host:port/path?data | Data as is possibly URL encoded, GWWS will perform the encoding. |
| CARRAY | http://host:port/path?data | Data represented as base64 encoded string. |
| MBSTRING | http://host:port/path?data | Data represented as URL encoded of UTF-8 representation of the Tuxedo MBSTRING. |
| XML | http://host:port/path?data | XML fragment as is, URL encoded. |
| X_C_TYPE | Same as VIEW/VIEW32 | |
| X_COMMON | Same as VIEW/VIEW32 | |
| X_OCTET | Same as CARRAY | |

Table 2-15 Query String Mapping

| Tuxedo Buffer Type | Query String Mapping | Notes |
|--------------------|--|---|
| VIEW/VIEW32 | <p>http://host:port/path?value1&value2 or http://host:port/service?fieldname1=value1&fieldname2=value2</p> | <p>GWWS attempts to convert values to the corresponding VIEW/VIEW32 member depending on the target type: number types from their string representation to their Oracle Tuxedo ones:</p> <ul style="list-style-type: none"> • float notation for float and double VIEW/VIEW32 types • integer notation for int, long and other integer based types • FLD_CHAR fields are translated from URL-encoded content, i.e. representable characters or their '%xx' representation • string for all other types <p>The <code>fieldname=value</code> notation is used with:</p> <ul style="list-style-type: none"> • FBNAME field name when one is configured in the view description. • CNAME value when no FBNAME is present in the view description. • If neither FBNAME nor CNAME matches for this subtype a mapping error is returned. |

Table 2-15 Query String Mapping

| Tuxedo Buffer Type | Query String Mapping | Notes |
|--------------------|---|--|
| FML/FML32 | <p>http://host:port/path?fieldname1=value1&fieldname2=value2</p> <p>or, for multiple occurrences:</p> <p>http://host:port/service?fieldname1=value1&fieldname1=value2</p> | <p>Actual values are converted from URL encoded string representations to their native types.</p> <p>GWWS attempts to convert values to the corresponding FML/FML32 member depending on the target type: number types from their string representation to their Tuxedo ones:</p> <ul style="list-style-type: none"> float notation for <code>float</code> and double FML/FML32 types integer notation for <code>int</code>, <code>long</code> and other integer-based types FLD_CHAR fields are translated from URL-encoded content (i.e., representable characters or their '%xx' representation) <code>string</code> for all other types |

JSON Data Mapping

The different Tuxedo buffer types are converted into/from JSON as shown in [Table 2-16](#).

Table 2-16 JSON Data Mapping

| Oracle Tuxedo Buffer Type | JSON equivalent/example | Notes |
|---------------------------|-------------------------|--|
| STRING | <buffer content> | |
| CARRAY | <binary buffer content> | |
| MBSTRING | <Multi-byte string> | In order to transmit encodings other than UTF-8, the "enableMultiEncoding" property must be set to "true" in the SALTDEPLOY configuration. |

Table 2-16 JSON Data Mapping

| Oracle Tuxedo Buffer Type | JSON equivalent/example | Notes |
|---------------------------|-------------------------|--|
| XML | <XML fragment as-is> | In order to transmit encodings other than UTF-8, the "enableMultiEncoding" property must be set to "true" in the SALTDEPLOY configuration. |
| X_C_TYPE | Same as VIEW/VIEW32 | |
| X_COMMON | Same as VIEW/VIEW32 | |
| X_OCTET | Same as CARRAY | |

Table 2-16 JSON Data Mapping

| Oracle Tuxedo Buffer Type | JSON equivalent/example | Notes |
|---------------------------|--|-------|
| VIEW/VIEW32 | <pre>{ '<fieldname>': '<fieldcont ent>', '<fieldname>': '<fieldcont ent>'}</pre> <p>possibly nested:</p> <pre>{ '<fieldname>': { '<fieldnam e>': '<fieldcontent>' }}</pre> <p>JSON has the following primitive types:</p> <p>boolean (true/false)</p> <p>Number (int or double float)</p> <p>String</p> <p>VIEW/VIEW32 field types will be mapped as follows (Tuxedo type: JSON type):</p> <ul style="list-style-type: none"> • short: Number • int: Number • long: Number • float: Number • double: Number • char: String • string: String • carray: String (base64 encoded) • bool: boolean • unsigned char: String • signed char: String • wchar_t* or wchar_t: String • unsigned int: Number • unsigned long: Number | |

Table 2-16 JSON Data Mapping

| Oracle Tuxedo Buffer Type | JSON equivalent/example | Notes |
|---------------------------|---|--|
| | <ul style="list-style-type: none"> long double: String (See notes) mbstring: String view32: nested JSON record | <p>See VIEW/VIEW32 considerations and examples for fieldname mapping details.</p> <p>Some types may be truncated if represented in their primitive types (long long, long double), in that case they will be rendered as JSON strings.</p> |
| FML/FML32 | <pre>{ '<fieldname>': '<fieldcontent>', '<fieldname>': '<fieldcontent>' }</pre> <p>possibly nested, FML32 only:</p> <pre>{ '<fieldname>': { '<fieldname>': '<fieldcontent>' } }</pre> <p>FML/FML32 field types are mapped as follows (Tuxedo type: JSON type):</p> <ul style="list-style-type: none"> FLD_SHORT: Number FLD_LONG: Number FLD_FLOAT: Number FLD_DOUBLE: Number FLD_CHAR: String or character 'T' for JSON true or 'F' for JSON false FLD_CARRRAY: String (base64 encoded) FLD_MBSTRING: String FLD_VIEW32: JSON nested record, see VIEW/VIEW32 mapping for individual types FLD_FML32: JSON bject | <p>Nested FLD_VIEW32: the name of the view subtype must be the name of the embedded VIEW32. For Example:</p> <p>VIEW32 example .v definition file:</p> <pre>VIEW v32 example charflag1 - 1 --- string str - 1 100 - -</pre> <p>JSON content (EVIEW32 is a FLD_VIEW32 fml32 type):</p> <pre>{ "EVIEW32" : { "v32example": { "flag1": "x", "str": "somestring" } } }</pre> |

Notes: Non-structured buffer types (`STRING`, `CARRAY`, `X_OCTET` and `MBSTRING`) will not wrap data as JSON objects, the data is transmitted as is. The content-type setting is ignored for those buffer types with respect to data mapping.

JSON internally handles all floating point types differently than XML. XML conversion floating point conversion may incur some precision loss over similar JSON conversions. This is currently a limitation.

VIEW/VIEW32 Considerations:

The following considerations apply when converting Oracle Tuxedo `VIEW/VIEW32` buffers to and from XML:

- You must create an environment for converting XML to and from `VIEW/VIEW32`. This includes setting up a `VIEW` directory and system `VIEW` definition files. These definitions are automatically loaded by the GWWS server.

FML/FML32 Considerations

The following considerations apply to converting Oracle Tuxedo `FML/FML32` buffers to and from XML:

- You must create an environment for converting XML to and from `FML/FML32`. This includes an FML field table file directory and system FML field definition files. These definitions are automatically loaded by the GWWS. `FML` typed buffers can be handled only if the environment is set up correctly.

Note: `FML32` Field type `FLD_PTR` is not supported.

Conversion Examples:

Listing 2-16 VIEW Description File

```
VIEW empname
#TYPE          CNAME  FBNAME    COUNT FLAG SIZE NULL
char           fname  EMP_FNAME  1    -   25   -
char           minit  EMP_MINIT  1    -   1    -
char           lname  EMP_LNAME  1    -   25   -
```

```
END
```

```
VIEW emp
```

```

struct      empname  ename      1  -  -  -
unsignedlong id      EMP_ID    1  -  -  -
long       ssn      EMP_SSN    1  -  -  -
double     salaryhist EMP_SAL  10 -  -  -

```

END

Corresponding header file after compilation

Listing 2-17 Compilation

```

struct empname {
    char fname[25];
    char minit;
    char lname[25];
};

struct emp {
    struct empname ename;
    unsigned long id;
    long ssn;
    double salaryhist[10];
}

```

JSON Content Example

Listing 2-18 JSON Content Example

```
{
  "ename" :
    {
      "EMP_FNAME" : "John",
      "EMP_MINIT" : "R",
      "EMP_LNAME" : "Smith"
    },
  "EMP_ID" : 1234,
  "EMP_SSN" : 123456789,
  "EMP_SAL" :
    [10000.0,
     11000.0,
     12000.0,
     13000.0,
     14000.0,
     15000.0,
     16000.0,
     17000.0,
     18000.0,
     19000.0]
    }
}
```

Without `FBNAME`(names specified in the view file), the content is represented using the `CNAME` values. Since nesting cannot be expressed without field names because the field name is also the subtype name for the nested view, only structures with 1 level are represented.

For example:

Listing 2-19 VIEW Description

```
VIEW empname
#TYPE          CNAME  FBNAME      COUNT FLAG SIZE NULL
char           fname  -           1    -  25   -
char           minit  -           1    -   1   -
char           lname  -           1    -  25   -
END
```

Corresponding header file after compilation

Listing 2-20 Compilation

```
struct empname {
    char fname[25];
    char minit;
    char lname[25];
};
```

Listing 2-21 JSON Content Example

```
{
    "fname": "John",
    "minit": "R",
    "lname": "Smith"
}
```

FML32

Listing 2-22 Field Table

| #name | rel-num | bertype | flags | comment |
|--------------|---------|---------|-------|---------|
| BIKES | 1 | fml32 | - | |
| COLOR | 2 | string | - | |
| CURSERIALNO3 | | string | - | |
| INSTOCK4 | | string | - | |
| NAME | 5 | string | - | |
| ORDERDATE6 | | string | - | |
| PRICE | 7 | float | - | |
| SERIALNO8 | | string | - | |
| SIZE | 9 | long | - | |
| SKU | 10 | string | - | |
| TYPE | 11 | string | - | |

Listing 2-23 JSON Content Example

```
"BIKES":  
  [  
    { "COLOR": "BLUE",  
      "CURSERIALNO": "AZ123",  
      "INSTOCK": "Y",
```

```

        "NAME" : "CUTTER" ,
        "ORDERDATE" : "11/03/2012" ,
        "PRICE" : 1234.55 ,
        "SERIALNO" : "123456" ,
        "SIZE" : 52 ,
        "SKU" : "CU521234" ,
        "TYPE" : "ROAD" } ,
    { "COLOR" : "RED" ,
      "CURSERIALNO" : "BZ123" ,
      "INSTOCK" : "Y" ,
      "NAME" : "ROCKGLIDER" ,
      "ORDERDATE" : "11/06/2012" ,
      "PRICE" : 1455.55 ,
      "SERIALNO" : "123457" ,
      "SIZE" : 16 ,
      "SKU" : "RG161234" ,
      "TYPE" : "MTB" } ,
    ]
  }
}

```

XML Data Mapping

XML data mapping is performed using similar rules as the mapping used in SOAP mode.

Note the following:

- Floating point numbers without decimal value get represented as integers, for example: 10.0 is printed as 10. This is currently a limitation.
- No namespaces is generated or processed, since HTTP mode does not use interfaces.

- Simple buffers (`STRING`, `CARRAY`, `MBSTRING`, and `XML`) are sent and received as is, without any XML processing. The behavior is identical to JSON processing (i.e., no mapping is necessary).
- `FML` and `FML32` requests must be wrapped by a root element (which name is ignored, as long as the XML is formed properly), and replies are wrapped in an element with the same name as the subtype as specified in the `HTTP/Service/@outputbuffer` attribute of the `SALTDEPLOY` configuration file, or `<root>` element if subtype is not configured. `VIEW`, `VIEW32`, `X_COMMON`, and `X_C_TYPE` buffers use the subtype name as the root element name.

The different Oracle Tuxedo buffer types are converted into/from XML in the following manner as shown in [Table 2-17](#)

Table 2-17 XML Data Mapping

| Oracle Tuxedo Buffer Type | Description | HTTP XML Mapping Example |
|---------------------------|---|--------------------------|
| <code>STRING</code> | Oracle Tuxedo <code>STRING</code> typed buffers are used to store character strings that terminate with a <code>NULL</code> character. Oracle Tuxedo <code>STRING</code> typed buffers are self-describing. | HELLO WORLD! |
| <code>CARRAY</code> | Oracle Tuxedo <code>CARRAY</code> typed buffers store character arrays, any of which can be <code>NULL</code> . <code>CARRAY</code> buffers are used to handle data opaquely and are not self-describing. | Binary content |

Table 2-17 XML Data Mapping

| Oracle Tuxedo Buffer Type | Description | HTTP XML Mapping Example |
|---------------------------|---|--|
| MBSTRING | <p>Oracle Tuxedo MBSTRING typed buffers are used for multibyte character arrays. Oracle Tuxedo MBSTRING buffers consist of the following three elements:</p> <ul style="list-style-type: none"> - Code-set character encoding - Data length - Character array of the encoding. <p>In order to transmit encodings other than UTF-8, the "enableMultiEncoding" property must be set to "true" in the SALTDEPLOY configuration.</p> | Multi-byte string encoded according to Content-Type setting. |
| XML | <p>Oracle Tuxedo XML typed buffers store XML documents.</p> <p>The GWWS server validates that the actual XML data is well-formed. It will not do any other enforcement validation, such as Schema validation.</p> <p>Only a single root XML buffer is allowed to be stored in the payload; the GWWS server checks for this.</p> <p>Any original XML document prologue information cannot be carried within the payload.</p> <p>In order to transmit encodings other than UTF-8, the "enableMultiEncoding" property must be set to "true" in the SALTDEPLOY configuration.</p> | XML fragment as-is |
| X_C_TYPE | Same as VIEW/VIEW32 | |

Table 2-17 XML Data Mapping

| Oracle Tuxedo Buffer Type | Description | HTTP XML Mapping Example |
|---------------------------|--|---|
| X_COMMON | Same as VIEW/VIEW32 | |
| X_OCTET | Same as CARRAY | |
| VIEW/VIEW32 | <p>Oracle Tuxedo VIEW and VIEW32 typed buffers store C structures defined by Oracle Tuxedo applications.</p> <p>VIEW structures are defined by using VIEW definition files. A VIEW buffer type can define multiple fields.</p> <p>VIEW supports the following field types:</p> <ul style="list-style-type: none"> • short • int • long • float • double • char • string • carray (represented as base64 encoded content) • bool • unsigned char • signed char • wchar_t* or wchar_t • unsigned int • unsigned long • long long • unsigned long long • long double | <pre><VIEW> <viewfieldname> fieldcontent </viewfieldname> </VIEW></pre> |

Table 2-17 XML Data Mapping

| Oracle Tuxedo Buffer Type | Description | HTTP XML Mapping Example |
|---------------------------|---|--|
| | <p>VIEW32 supports all the VIEW field types, mbstring, and embedded VIEW32 type.</p> <p>The name of the sub-element is the VIEW field name. The occurrence of the sub-element depends on the count attribute of the VIEW field definition. The value of the sub-element should be in the VIEW field data type corresponding XML Schema type.</p> | |
| FML/FML32 | <p>Oracle Tuxedo FML and FML32 type buffers are proprietary Oracle Oracle Tuxedo system self-describing buffers. Each data field carries its own identifier, an occurrence number, and possibly a length indicator.</p> <p>FML supports the following field types:</p> <ul style="list-style-type: none"> • FLD_CHAR • FLD_SHORT • FLD_LONG • FLD_FLOAT • FLD_DOUBLE • FLD_STRING • FLD_CARRAY (as base64 encoded content) <p>FML32 supports all the FML field types and FLD_PTR, FLD_MBSTRING, FLD_FML32, and FLD_VIEW32.</p> | <p>Nested FLD_VIEW32: the name of the view subtype must be the name of the embedded VIEW32. For Example:</p> <pre>VIEW32 example.v definition file: VIEW v32example char flag1 - 1 --- string str - 1 - 100 XML content (EVIEW32 is a FLD_VIEW32 fml32 type): <EVIEW32> <v32example> <flag1>x</flag1> <str>somestring</str> </v32example> </EVIEW32></pre> |

Note: Non-structured buffer types (STRING, CARRAY, X_OCTET and MBSTRING) will not wrap data as XML objects, the data is transmitted as is.

VIEW/VIEW32 Considerations:

The following considerations apply when converting Oracle Tuxedo VIEW/VIEW32 buffers to and from XML:

- You must create an environment for converting XML to and from VIEW/VIEW32. This includes setting up a VIEW directory and system VIEW definition files. These definitions are automatically loaded by the GWWS server.

FML/FML32 Considerations

The following considerations apply to converting Oracle Tuxedo FML/FML32 buffers to and from XML:

- You must create an environment for converting XML to and from FML/FML32. This includes an FML field table file directory and system FML field definition files. These definitions are automatically loaded by the GWWS. FML typed buffers can be handled only if the environment is set up correctly.

Note: FML32 Field type FLD_PTR is not supported

Web Service Client Programming

This chapter contains the following topics:

- [Overview](#)
- [SALT Web Service Client Programming Tips](#)
- [Web Service Client Programming References](#)

Overview

SALT is a configuration-driven product that publishes existing Oracle Tuxedo application services as industry-standard Web services. From a Web services client-side programming perspective, SALT (used in conjunction with the Oracle Tuxedo framework), is a standard Web service provider. You only need to use the SALT WSDL file to develop a Web service client program.

To develop a Web service client program, do the following steps:

1. Generate or download the SALT WSDL file. For more information, see [Configuring SALT](#).
2. Use a Web service client-side toolkit to parse the SALT WSDL document, and generate client stub code. For more information, see [SALT Web Service Client Programming Tips](#).
3. Write client-side application code to invoke a SALT Web service using the functions defined in the client-generated stub code.
4. Compile and run your client application.

REpresentational State Transfer (REST) Support

With REST enabled, requests received on a REST port are processed as follows by GWWS.

URIs must comply with the following pattern:

```
<REST service name>
```

Where the Oracle Tuxedo service name is the name of the REST service invoked (for example, TOUPPER).

Data format and input Oracle Tuxedo buffer types are specified using the following HTTP header:

- content-type:

Set to `application/json`: indicates that JSON is used to transfer data to/from HTTP client.

Set to `application/xml`: indicates that XML is used to transfer data to/from HTTP client.

Note: `application/json` and `application/xml` will only apply to structured buffer types (`VIEW`, `VIEW32`, `FML`, `FML32`, `X_C_TYPE` and `X_COMMON`). To use simple buffers and `POST` or `PUT`, you must set Content-type to appropriate values ("`text/plain`" for `STRING`, "`application/octet-stream`" for `CARRAY`, etc.).

Oneway (in and out)

If no data is input, the Oracle Tuxedo service is invoked with a `NULL` Oracle Tuxedo buffer. Similarly, if the Oracle Tuxedo service does not return any data, the response also contains no data (which is a valid use-case).

ATMI and SCA Support

There is no restriction in the type of Oracle Tuxedo service being exposed as REST (whether ATMI or SCA). To use SCA components, you must conform to SCA data mapping conventions as found in [SCA Data Type Mapping](#). Name mapping may apply, as outlined in [SCA and Oracle Tuxedo Interoperability](#).

Examples

Example 1: .h interface

Listing 3-1 .h interface

```
#include <string>
/**
 * Tuxedo service business interface
 */
class TuxService
{
public:
    virtual std::string TOUPPER(const std::string inputString) = 0;
};
```

Example 2: SCDL Descriptor

Listing 3-2 SCDL Descriptor

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="myComponent">
  <service name="TuxService">
    <interface.cpp header="TuxService.h"/>
    <binding.atmi/>
    <inputBufferType>STRING</inputBufferType>
    <outputBufferType>STRING</outputBufferType>
    <reference>MYComponent</reference>
  </service>
```

```
<component name="MYComponent">
  <implementation.cpp library="TuxService" header="TuxServiceImpl.h"/>
</component>
</composite>
```

Example 3: SALTDEPLOY REST Service Definition

Listing 3-3 SALTDEPLOY REST Service Definition

```
<REST>
  <Network http="myhost:1234"/>
  <Service name="testSCA">
    <Method name="GET"
      reposservice=" "
      service="TuxService/TOUPPER"
      inputbuffer="STRING"/>
  </Service>
  ...
</REST>
```

Example 4: URL used to invoke service

http://myhost:1234/testSCA?teststring

Example 5: Response

```
HTTP/1.1 200 OK
Content-Type: text/xmlTESTSTRING
```


SALT Web Service Client Programming Tips

This section provides some useful client-side programming tips for developing Web service client programs using the following SALT-tested programming toolkits:

- [Oracle WebLogic Web Service Client Programming Toolkit](#)
- [Apache Axis for Java Web Service Client Programming Toolkit](#)
- [Microsoft .NET Web Service Client Programming Toolkit](#)

For more information, see [Interoperability Considerations](#) in the *SALT Administration Guide*.

Notes: You can use any SOAP toolkit to develop client software.

The sample directories for the listed toolkits can be found *after* SALT is installed.

Oracle WebLogic Web Service Client Programming Toolkit

WebLogic Server provides the `clientgen` utility which is a built-in application server component used to develop Web service client-side java programs. The invocation can be issued from standalone java programs and server instances. For more information, see [Developing JAX-WS Web Services for Oracle WebLogic Server](#).

Besides traditional synchronous message exchange mode, SALT also supports asynchronous and reliable Web service invocation using WebLogic Server. Asynchronous communication is defined by the WS-Addressing specification. Reliable message exchange conforms to the WS-ReliableMessaging specification.

Tip: Use the WebLogic specific WSDL document for HTTP MIME attachment support.

SALT can map Oracle Tuxedo `CARRAY` data to SOAP request `MIME` attachments. This is beneficial when the binary data stream is large since `MIME` binding does not need additional encoding wrapping. This can help save CPU cycles and network bandwidth.

Another consideration, in an enterprise service oriented environment is that binary data might be used to guide high-level data routing and transformation work. Encoded data can be problematic. To enable the `MIME` data binding for Oracle Tuxedo `CARRAY` data, a special flag must be specified in the WSDL document generation options (both for online downloading and using the `tmwsdlgen` command utility).

Online Download:

`http://salt.host:portnumber//wsdl?mappolicy=raw&toolkit=wls`

tmwsdlgen Utility

```
tmwsdlgen -c WSDL_FILE -m raw -t wls
```

Apache Axis for Java Web Service Client Programming Toolkit

SALT supports the AXIS `wsdl2java` utility which generates java stub code from the WSDL document. The AXIS Web service programming model is similar to WebLogic.

Tip: 1. Use the AXIS specific WSDL document for HTTP MIME attachment support.

SALT supports HTTP MIME transportation for Oracle Tuxedo `CARRAY` data. A special option must be specified for WSDL online downloading and the `tmwsdlgen` utility.

Online Download:

```
http://salt.host:portnumber//wsdl?mappolicy=raw&toolkit=axis
```

tmwsdlgen Utility

```
tmwsdlgen -c WSDL_FILE -m raw -t axis
```

Tip: 2. Disable multiple-reference format in AXIS when RPC/encoded style is used.

AXIS may send a multi-reference format SOAP message when RPC/encoded style is specified for the WSDL document. SALT does not support multiple-reference format. You can disable AXIS multiple-reference format as shown in [Listing 3-4](#):

Listing 3-4 Disabling AXIS Multiple-Reference Format

```
TuxedoWebServiceLocator service = new TuxedoWebServiceLocator();
service.getEngine().setOption("sendMultiRefs", false);
```

Tip: 3. Use Apache Sandsha project with SALT for WS-ReliableMessaging communication.

Interoperability has been tested for WS-ReliableMessaging between SALT and the Apache Sandsha project. The Sandsha asynchronous mode and `send offer` must be set in the code.

A sample Apache Sandesha asynchronous mode and send offer code example is shown in [Listing 3-5](#):

Listing 3-5 Sample Apache Sandesha Asynchronous Mode and “send offer” Code Example

```

/* Call the service */
    TuxedoWebService service = new TuxedoWebServiceLocator();

    Call call = (Call) service.createCall();
    SandeshaContext ctx = new SandeshaContext();

    ctx.setAcksToURL("http://127.0.0.1:" + defaultClientPort +
"/axis/services/RMService");
    ctx.setReplyToURL("http://127.0.0.1:" + defaultClientPort +
"/axis/services/RMService");
    ctx.setSendOffer(true);
    ctx.initCall(call, targetURL, "urn:wsm:simpapp",
Constants.ClientProperties.IN_OUT);

    call.setUseSOAPAction(true);
    call.setSOAPActionURI("ToUpperWS");
    call.setOperationName(new
javax.xml.namespace.QName("urn:pack:simpappsimpapp_typedef:salt11",
"ToUpperWS"));
    call.addParameter("inbuf", XMLType.XSD_STRING, ParameterMode.IN);
    call.setReturnType(org.apache.axis.encoding.XMLType.XSD_STRING);

    String input = new String();
    String output = new String();
    int i;
    for (i = 0; i < 3; i++) {
        input = "request" + "_" + String.valueOf(i);

        System.out.println("Request: "+input);

```

```
        output = (String) call.invoke(new Object[]{input});
        System.out.println("Reply:" + output);
    }

ctx.setLastMessage(call);
    input = "request" + "_" + String.valueOf(i);
    System.out.println("Request:"+input);
    output = (String) call.invoke(new Object[]{input});
```

Microsoft .NET Web Service Client Programming Toolkit

Microsoft .Net 1.1/2.0 provides `wsdl.exe` in the .Net SDK package. It is a free development Microsoft toolkit. In the SALT `simpapp` sample, a .Net program is provided in the `simpapp/dnetclient` directory.

.Net Web service programming is easy and straightforward. Use the `wsdl.exe` utility and the SALT WSDL document to generate the stub code, and then reference the .Net object contained in the stub code/binary in business logic implementations.

Tip: 1. Do not use .Net program MIME attachment binding for CARRAY.

Microsoft does not support SOAP communication MIME binding. Avoid using the WSDL document with MIME binding for `CARRAY` in .Net development.

SALT supports `base64Binary` encoding for `CARRAY` data (the default WSDL document generation.)

Tip: 2. Some RPC/encoded style SOAP messages are not understood by the GWWS server.

When the SALT WSDL document is generated using RPC/encoded style, .Net sends out SOAP messages containing `soapenc:arrayType`. SALT does not support `soapenc:arrayType` using RPC/encoded style. A sample RPC/encoded style-generated WSDL document is shown in [Listing 3-6](#).

Listing 3-6 Sample RPC/encoded Style-Generated WSDL Document

```

<wsdl:types>
    <xsd:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
targetNamespace="urn:pack.TuxAll_typedef.salt11">
        <xsd:complexType name="fml_TFML_In">
            <xsd:sequence>
                <xsd:element maxOccurs="60"
minOccurs="60" name="tflong" type="xsd:long"></xsd:element>
                <xsd:element maxOccurs="80"
minOccurs="80" name="tffloat" type="xsd:float"></xsd:element>
            </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="fml_TFML_Out">
            ...
        </xsd:complexType>
    </xsd:schema>
</wsdl:types>

```

Workaround: Use Document/literal encoded style for .Net client as recommended by Microsoft.

Tip: 3. Error message regarding `xsd:base64Binary` in RPC/encoded style.

If `xsd:base64Binary` is used in the SALT WSDL document using RPC/encoded style, `wsdl.exe` can generate stub code; however, the client program might report a runtime error as follows:

```
System.InvalidOperationException: 'base64Binary' is an invalid value for the SoapElementAttribute.DataType property. The property may only be specified for primitive types.
```

Workaround: This is a .Net framework issue.

Use Document/literal encoded style for .Net client as recommended by Microsoft.

Web Service Client Programming References

Online References

- Oracle WebLogic 10.0 Web Service Client Programming References

[Oracle WebLogic 10.0 Documentation](#)

- Apache Axis 1.3 Web Service Client Programming References

[Consuming Web Services with Axis](#)

[Using WSDL with Axis](#)

- Microsoft .NET Web Service Programming References

[Building Web Services](#)

Oracle Tuxedo ATMI Programming for Web Services

This chapter contains the following topics:

- [Overview](#)
- [Converting WSDL Model Into Oracle Tuxedo Model](#)
- [Invoking SALT Proxy Services](#)

Overview

SALT allows you to import external Web Services into Oracle Tuxedo Domains. To import external Web services into Oracle Tuxedo applications, a WSDL file must first be loaded and converted. The SALT WSDL conversion utility, `wsdlcvt`, translates each `wsdl:operation` into a SALT proxy service. The translated SALT proxy service can be invoked directly through standard Oracle Tuxedo ATMI functions.

SALT proxy service calls are sent to the GWWS server. The request is translated from Oracle Tuxedo typed buffers into the SOAP message, and then sent to the corresponding external Web Service. The response from an external Web Service is translated into Oracle Tuxedo typed buffers, and returned to the Oracle Tuxedo application. The GWWS acts as the proxy intermediary.

If an error occurs during the service call, the GWWS server sets the error status using `tperrno` (which can be retrieved by Oracle Tuxedo applications). This enables you to detect and handle the SALT proxy service call error status.

Converting WSDL Model Into Oracle Tuxedo Model

SALT provides a WSDL conversion utility, [wsdlcvt](#), that converts external WSDL files into Oracle Tuxedo specific definition files so that you can develop Oracle Tuxedo ATMI programs to access services defined in the WSDL file.

WSDL-to-Tuxedo Object Mapping

SALT converts WSDL object models into Oracle Tuxedo models using the following rules:

- Only SOAP over HTTP binding are supported. Each binding is defined and saved as a `WSBinding` object in the `WSDF` file.
- Each operation in the SOAP binding is mapped as one Oracle Tuxedo-style service (which is also called a SALT proxy service). The operation name is used as the Oracle Tuxedo service name and indexed in the Oracle Tuxedo Service Metadata Repository.

Note: If the operation name exceeds the Oracle Tuxedo service name length limitation (255 characters), you must manually set a unique short Oracle Tuxedo service name in the metadata repository and set the `<Service> tuxedoRef` attribute in the `WSDF` file.

For more information, see [SALT Web Service Definition File Reference](#) in the *SALT Reference Guide*.

- Other Web service external application protocol information is saved in the generated `WSDF` file (including SOAP protocol version, SOAP message encoding style, accessing endpoints, etc.).
- XML Schema definitions embedded in the `WSDL` file are copied and saved in separate `.xsd` files.
- Each `wsdl:operation` object and its input/output message details are converted as the Oracle Tuxedo service definition conforms to the Oracle Tuxedo Service Metadata Repository input syntax.

[Table 4-1](#) lists detailed mapping relationships between the `WSDL` file and Oracle Tuxedo definition files.

Table 4-1 WSDL Model / Oracle Tuxedo Model Mapping Rules

| WSDL Object | Oracle Tuxedo/SALT Definition File | Oracle Tuxedo/SALT Definition Object |
|----------------------------|---|--------------------------------------|
| /wsdl:binding | SALT Web Service Definition File (WSDF) | /WSBinding |
| /wsdl:portType | | /WSBinding/Servicegroup |
| /wsdl:binding/soap:binding | | /WSBinding/SOAP |
| /wsdl:portType/operation | Metadata Input File (MIF) | /WSBinding/service |
| /wsdl:types/xsd:schema | FML32 Field Definition Table | Field name type |

Invoking SALT Proxy Services

The following sections include information on how to invoke the converted SALT proxy service from an Oracle Tuxedo application:

- [SALT Supported Communication Patterns](#)
- [Oracle Tuxedo Outbound Call Programming: Main Steps](#)
- [Managing Error Code Returned from GWWS](#)
- [Handling Fault Messages in an Oracle Tuxedo Outbound Application](#)

SALT Supported Communication Patterns

SALT only supports the Oracle Tuxedo Request/Response communication patterns for outbound service calls. An Oracle Tuxedo application can request the SALT proxy service using the following communication Oracle Tuxedo ATMI:

- `tpcall(3c)` / `tpacall(3c)` / `tpgetreply(3c)`

These basic ATMI functions can be called with an Oracle Tuxedo typed buffer as the input parameter. The return of the call also carries an Oracle Tuxedo typed buffer. All these buffers conform to the converted outside Web service interface. `tpacall/tpgetreply` are not related to SOAP async communication.

- `tpgetcallinfo(3c)/tpsecallinfo(3c)`

`tpgetcallinfo()` retrieves HTTP headers associated with an application buffer using the GWWS gateway in FML32 format; `tpsetcallinfo()` performs the reverse (i.e., attach FML32 formatted HTTP headers to an application buffer to be sent to a remote HTTP (possibly SOAP) server).

- `tpforward(3c)`

Oracle Tuxedo server applications can use this function to forward an Oracle Tuxedo request to a specified SALT proxy service. The response buffer is sent directly to the client application response queue as if it is a traditional native Oracle Tuxedo service.

- `TMQFORWARD` enabled queue-based communication.

Oracle Tuxedo system server `TMQFORWARD` can accept queued requests, and sends them to SALT proxy services that have the same name as the queue.

For more information, see [Oracle Tuxedo ATMI C Functions and File Formats, Data Descriptions, MIBs, and System Processes Reference](#).

SALT does not support the following Oracle Tuxedo communication patterns:

- Conversational communication
- Event-based communication

Oracle Tuxedo Outbound Call Programming: Main Steps

When the GWWS is booted and SALT proxy services are advertised, you can create an Oracle Tuxedo application to call them. To develop a program to access SALT proxy services, do the following:

1. Check the Oracle Tuxedo Service Metadata Repository definition to see what the SALT proxy service interface is.
2. Locate the generated FML32 field table files. Modify the FML32 field table to eliminate conflicting field names and assign a valid base number for the index.

Note: The `wsdlcvt` generated FML32 field table files are always used by GWWS. You must make sure the field name is unique at the system level. If two or more fields are associated with the same field name, change the field name. Do not forget to change Oracle Tuxedo Service Metadata Repository definition accordingly.

The base number field index in the generated FML32 field table must be changed

from the invalid default value to a correct number to ensure all field indexes in the table are unique at the entire system level.

3. Generate FML32 header files with `mkfldhdr32(1)`.
4. Boot the GWWS with correct FML32 environment variable settings.
5. Write a skeleton C source file for the client to call the outbound service (refer to Oracle Tuxedo documentation and the Oracle Tuxedo Service Metadata Repository generated pseudo-code if necessary). You can use `tpcall(1)` or `tpacall(1)` for synchronous or asynchronous communication, depending on the requirement.
6. For FML32 buffers, you must add each FML32 field (conforming to the corresponding SALT proxy service input buffer details), defined in the Oracle Tuxedo Service Metadata Repository (including FML32 field sequence and occurrence). The client source may include the generated header file to facilitate referencing the field name.
7. Get input buffer ready. You can handle the returned buffer, which should be of the type defined in Metadata.
 - Compile the source to generate executable.
 - Test the executable.

Managing Error Code Returned from GWWS

If the GWWS server encounters an error accessing external Web services, `tperrno` is set accordingly so the Oracle Tuxedo application can diagnose the failure. [Table 4-2](#) lists possible SALT proxy service `tperrno` values.

Table 4-2 Error Code Returned From GWWS/Tuxedo Framework

| TPERRNO | Possible Failure Reason |
|-----------|--|
| TPENOENT | Requested SALT proxy service is not advertised by GWWS. |
| TPESVCERR | The HTTP response message returned from external Web service application is not valid. The SOAP response message returned from external Web service application is not well-formed. |
| TPEPERM | Authentication failure. |

Table 4-2 Error Code Returned From GWWS/Tuxedo Framework

| TPERRNO | Possible Failure Reason |
|-----------|---|
| TPEITYPE | Message conversion failure when converting Oracle Tuxedo request typed buffer into XML payload of the SOAP request message. |
| TPEOTYPE | Message conversion failure when converting XML payload of the SOAP response message into Oracle Tuxedo response typed buffer. |
| TPEOS | Request is rejected because of system resource limitation. |
| TPETIME | Timeout occurred. This timeout can either be a BBL blocktime, or a SALT outbound call timeout. |
| TPSVCFAIL | External Web service returns SOAP fault message |
| TPESYSTEM | GWWS internal errors. Check ULOG for more information. |

Handling Fault Messages in an Oracle Tuxedo Outbound Application

All rules listed in the WSDL file are used to map WSDL input/output message into Oracle Tuxedo Metadata inbuf/outbuf definition. WSDL file default message can also be mapped into Oracle Tuxedo Metadata `errbuf` with some amendments to the rules:

Rules for fault mapping:

There are two modes for mapping Metadata `errbuf` into SOAP Fault messages: Tux Mode and XSD Mode.

- Tux Mode is used to convert Oracle Tuxedo original error buffers returned with `TPFAIL`. The error buffers are converted into the XML payload in the SOAP fault `<detail>` element.
- XSD Mode is used to represent SOAP fault and WSDL file fault messages defined with Oracle Tuxedo buffers. The mapping rule includes:
 - Each service in XSD mode (`servicemode=webservice`), always has an `errbuf` in Metadata with `type=FML32`.
 - `errbuf` is a FML32 buffer. It is a complete description of the `SOAP:Fault` message that may appear in correspondence (which is different for SOAP 1.1 and 1.2). The

`errbuf` definition content is determined by both the SOAP version and WSDL fault message.

- Parameter `detail/Detail` (1.1/1.2) is an FML32 field that represents the `wsdl:part` defined in a `wsdl:fault` message (when `wsdl:fault` is present). Each part is defined as a `param(field)` in the FML32 field. The mapping rules are the same as for input/output buffer. The difference is that each `param` `requiredcount` is 0 (which means it may not appear in the SOAP fault message).
- Other elements that appear in `soap:fault` message are always defined as a file in `errbuf`, with `requiredcount` equal to 1 or 0 (depending on whether the element is required or optional).
- Each part definition in the metadata controls converting a `<detail>` element in the `soap` fault message into a field in the error buffer.

Table 4-3 lists the outbound SOAP fault `errbuf` definitions.

Table 4-3 Outbound SOAP Fault Errbuf Definition

| Meta Parameter | SOAP Version | Type | Required | Memo |
|--------------------------|--------------|---------------------|----------|---|
| <code>faultcode</code> | 1.1 | <code>string</code> | Yes | |
| <code>faultstring</code> | 1.1 | <code>string</code> | Yes | |
| <code>faultactor</code> | 1.1 | <code>string</code> | No | |
| <code>detail</code> | 1.1 | <code>fm132</code> | No | If no <code>wsdl:fault</code> is defined, this field contains an XML field. |
| <code>Code</code> | 1.2 | <code>fm132</code> | Yes | Contains value and optional Subcode |
| <code>Reason</code> | 1.2 | <code>fm132</code> | Yes | Contains multiple text |
| <code>Node</code> | 1.2 | <code>string</code> | No | |

Table 4-3 Outbound SOAP Fault Errbuf Definition

| Meta Parameter | SOAP Version | Type | Required | Memo |
|----------------|--------------|--------|----------|----------------------|
| Role | 1.2 | string | No | |
| Detail | 1.2 | fm132 | No | same as detail field |

See Also

[Oracle Tuxedo ATMI C Functions](#)

[File Formats, Data Descriptions, MIBs, and System Processes Reference](#)

Using SALT Plug-Ins

This chapter contains the following topics:

- [Understanding SALT Plug-Ins](#)
- [Programming Message Conversion Plug-ins](#)
- [Programming Outbound Authentication Plug-Ins](#)

Understanding SALT Plug-Ins

The SALT [GWWS](#) server is a configuration-driven process which, for most basic Web service applications, does not require any programming tasks. However, SALT functionality can be enhanced by developing plug-in interfaces which utilize custom typed buffer data and customized shared libraries to extend the GWWS server.

A plug-in interface is a set of functions exported by a shared library that can be loaded and invoked by GWWS processes to achieve special functionality. SALT provides a plug-in framework as a common interface for defining and implementing a plug-in interface. Plug-in implementation is carried out by a shared library which contains the actual functions. The plug-in implementation library is configured in the [SALT Deployment file](#) and is loaded dynamically during GWWS server startup.

Plug-In Elements

Four plug-in elements are required to define a plug-in interface:

- [Plug-In ID](#)

- [Plug-In Name](#)
- [Plug-In Implementation Functions](#)
- [Plug-In Register Functions](#)

Plug-In ID

The plug-in ID element is a string used to identify a particular plug-in interface function. Multiple plug-in interfaces can be grouped with the same Plug-in ID for a similar function. Plug-in ID values are predefined by SALT. Arbitrary string values are not permitted.

SALT supports the `P_CUSTOM_TYPE` and `P_CREDENMAP` plug-in ID, which is used to define plug-in interfaces for custom typed buffer data handling, and map Oracle Tuxedo user ID and group ID into username/password that HTTP Basic Authentication needs.

Plug-In Name

The plug-in Name differentiates one plug-in implementation from another within the same Plug-in ID category.

For the `P_CUSTOM_TYPE` Plug-in ID, the plug-in name is used to indicate the actual custom buffer type name. When the GWWS server attempts to convert data between Oracle Tuxedo custom typed buffers and an XML document, the plug-in name is the key element that searches for the proper plug-in interface.

Plug-In Implementation Functions

Actual business logic should reflect the necessary functions defined in a plug-in vtable structure. Necessary functions may be different for different plug-in ID categories.

For the `P_CREDENMAP` ID category, one function needs to be implemented:

- `int (* gwws_pi_map_http_basic) (char * domain, char * realm, char * t_userid, char * t_grpid, Cred_UserPass * credential);`

For more information, see [“Programming Outbound Authentication Plug-Ins”](#).

Plug-In Register Functions

Plug-in Register functions are a set of common functions (or rules), that a plug-in interface must implement so that the GWWS server can invoke the plug-in implementation. Each plug-in interface must implement three register functions. These functions are:

- [Information Providing Function](#)
- [Initiating Function](#)
- [Exiting Function](#)
- [vtable Setting Function](#)

Information Providing Function

This function is optional. If it is used, it is first invoked after the plug-in shared library is loaded during GWWS server startup. If you want to implement more than one interface in one plug-in library, you must implement this function and return the counts, IDs, and names of the interfaces in the library.

Returning a 0 value indicates the function has executed successfully. Returning a value other than 0 indicates failure. If this function fails, the plug-in is not loaded, and the GWWS server will not start.

The function uses the following syntax:

```
int _ws_pi_get_Id_and_Names(int * count, char **ids, char **names);
```

You must return the total count of implementation in the library in arguments `count`. The arguments `ids` and `names` should contain all implemented interface `ids` and `names`, separated by a semicolon “;”.

Initiating Function

The initiating function is invoked after all the implemented interfaces in the plug-in shared library are determined. You can initialize data structures and set up global environments that can be used by the plug-ins.

Returning a 0 value indicates the initiating function has executed successfully. Returning a value other than 0 indicates initiation has failed. If plug-in interface initiation fails, the GWWS server will not start.

The initiating function uses the following syntax:

```
int _ws_pi_init_@ID@_@Name@(char * params, void **priv_ptr);
```

@ID@ indicates the actual plug-in ID value. @Name@ indicates the actual plug-in name value. For example, the initiating function of a plug-in with `P_CUSTOM_TYPE` as a plug-in ID and `MyType` as a plug-in name is: `_ws_pi_init_P_CUSTOM_TYPE_MyType (char * params, void **priv_ptr)`.

Exiting Function

The exiting function is called before closing the plug-in shared library when the GWWS server shuts down. You should release all reserved plug-in resources.

The exiting function uses the following syntax:

```
int _ws_pi_exit_@ID@_@Name@(void * priv);
```

@ID@ indicates the actual plug-in ID value. @Name@ indicates the actual plug-in name value. For example, the initiating exiting function name of a plug-in with `P_CUSTOM_TYPE` as a plug-in ID and `MyType` as a plug-in name is: `_ws_pi_exit_P_CUSTOM_TYPE_MyType(void * priv)`.

vtable Setting Function

`vtable` is a particular C structure that stores the necessary function pointers for the actual business logic of a plug-in interface. In other words, a valid plug-in interface must implement all the functions defined by the corresponding `vtable`.

The `vtable` setting function uses the following syntax:

```
int _ws_pi_set_vtbl_@ID@_@Name@(void * priv);
```

@ID@ indicates the actual plug-in ID value. @Name@ indicates the actual plug-in name value. For example, the `vtable` setting function of a plug-in with `P_CUSTOM_TYPE` as a plug-in ID and `MyType` as a plug-in name is: `_ws_pi_set_vtbl_P_CUSTOM_TYPE_MyType(void * priv)`.

The `vtable` structures may be different for different plug-in ID categories. For this SALT release, `P_CUSTOM_TYPE` and `P_CREDENMAP` are the only valid plug-in IDs.

The `vtable` structures for available plug-in interfaces are shown in [Listing 5-1](#).

Listing 5-1 VTable Structure

```
struct credmap_vtable {
    int (* gwws_pi_map_http_basic) (char * domain, char * realm, char *
t_userid, char * t_grpid, Cred_UserPass * credential); /* used for HTTP
Basic Authentication */
    /* for future use */
    void * unused_1;
    void * unused_2;
    void * unused_3;
};
```

`struct credmap_vtable` indicates that one function must be implemented for a `P_CREDENMAP` plug-in interface. For more information, see “[Programming Outbound Authentication Plug-Ins](#)”.

The function input parameter `void * priv` points to a concrete `vtable` instance. You should set the `vtable` structure with the actual functions in the `vtable` setting function.

An example of setting the `vtable` structure with actual functions in the `vtable` setting function is shown in [Listing 5-2](#).

Listing 5-2 Setting the `vtable` Structure with Actual Functions in the `vtable` Setting Function

```
int _DLLEXPORT_ _ws_pi_set_vtbl_P_CREDENMAP_TEST (void * vtbl)
{
    struct credmap_vtable * vtable;
    if ( ! vtbl )
        return -1;

    vtable = (struct credmap_vtable *) vtbl;

    vtable->gws_pi_map_http_basic = Credmap_HTTP_Basic;
    return 0;
}
```

Developing a Plug-In Interface

To develop a comprehensive plug-in interface, do the following steps:

1. Develop a shared library to implement the plug-in interface.
2. Define the plug-in interface in the `SALT` configuration file.

Developing a Plug-In Shared Library

To develop a plug-in shared library, do the following steps:

1. Write C language plug-in implementation functions for the actual business logic. These functions are not required to be exposed from the shared library. For more information, see [“Plug-In Implementation Functions”](#).
2. Write C language plug-in register functions that include: the initiating function, the exiting function, the `vtable` setting function, and the information providing function if necessary. These register functions need to be exported so that they can be invoked from the GWWS server. For more information, see [“Plug-In Register Functions”](#).
3. Compile all the above functions into one shared library.

Defining a Plug-In Interface in the SALT Configuration File

To define a plug-in shared library that is loaded by the GWWS server, the corresponding plug-in library path must be configured in the `SALT` deployment file. For more information, see [Creating the SALT Deployment File](#) in the *SALT Configuration Guide*.

An example of how to define plug-in information in the `SALT` deployment file is shown in [Listing 5-3](#).

Listing 5-3 Defined Plug-In in the SALT Deployment File

```
<?xml version="1.0" encoding="UTF-8"?>
<Deployment xmlns="http://www.bea.com/Tuxedo/SALTDEPLOY/2007">
    . . . . .
    . . . . .
    <System>
        <Plugin>
            <Interface
                id="P_CREDEENMAP"
                name="TEST"
                library="credmap_plugin.dll" />
        </Plugin>
    </System>
</Deployment>
```

Notes: To define multiple plug-in interfaces, multiple `<Interface>` elements must be specified. Each `<Interface>` element indicates one plug-in interface.

Multiple plug-in interfaces can be built into one shared library file.

Programming Message Conversion Plug-ins

SALT defines a complete set of default data type conversion rules to convert between Oracle Tuxedo buffers and SOAP message payloads. However, the default data type conversion rules may not meet all your needs in transforming SOAP messages into Oracle Tuxedo typed buffers or vice versa. To accommodate special application requirements, SALT supports customized message-level conversion plug-in development to extend the default message conversion.

Note: The SALT 12cR2 Message Conversion Plug-in is an enhanced successor to the SALT 1.1 Custom Buffer Type Conversion Plug-in.

The following topics are included in this section:

- [How Message Conversion Plug-ins Work](#)
- [When Do We Need Message Conversion Plug-in](#)
- [Developing a Message Conversion Plug-in Instance](#)
- [SALT 1.1 Custom Buffer Type Conversion Plug-in Compatibility](#)

How Message Conversion Plug-ins Work

Message Conversion Plug-in is a SALT supported Plug-in defined within the SALT plug-in framework. All Message Conversion Plug-in instances have the same [Plug-In ID](#) (“P_CUSTOM_TYPE”). Each particular Message Conversion Plug-in instance may implement two functions, one is used to convert SOAP message payloads to Oracle Tuxedo buffers, and the other is used to convert Oracle Tuxedo buffers to SOAP message payloads. These two function prototypes are defined in [Listing 5-4](#).

Listing 5-4 vtable Structure for SALT Plug-in “P_CUSTOM_TYPE” (C Language)

```
/* custtype_pi_ex.h */
struct custtype_vtable {
    CustomerBuffer * (* soap_in_tuxedo__CUSTBUF) (void * xercesDOMTree,
CustomerBuffer * tuxbuf, CustType_Ext * extinfo)
```

```

    int (* soap_out_tuxedo__CUSTBUF) (void ** xercesDOMTree,
CustomerBuffer * tuxbuf, CustType_Ext * extinfo)
    .....
}

```

The function pointer (`* soap_in_tuxedo__CUSTBUF`), points to the customized function that converts the SOAP message payload to Oracle Tuxedo typed buffer.

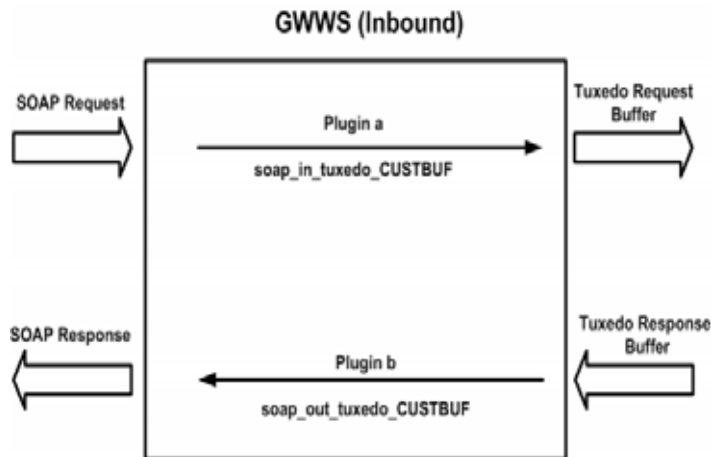
The function pointer (`* soap_out_tuxedo__CUSTBUF`), points to the customized function that converts the Oracle Tuxedo typed buffer to SOAP message payload.

You may implement both functions defined in the message conversion plug-in `vtable` structure if needed. You may also implement one function and set the other function with a `NULL` pointer.

How Message Conversion Plug-in Works in an Inbound Call Scenario

An inbound call scenario is an external Web service program that invokes an Oracle Tuxedo service through the SALT gateway. [Figure 5-1](#) depicts message streaming between a Web service client and an Oracle Tuxedo domain.

Figure 5-1 Message Conversion Plug-in Works in an Inbound Call Scenario



When a SOAP request message is delivered to the GWWS server, GWWS tries to find if there is a message conversion plug-in instance associated with the input message conversion of the target

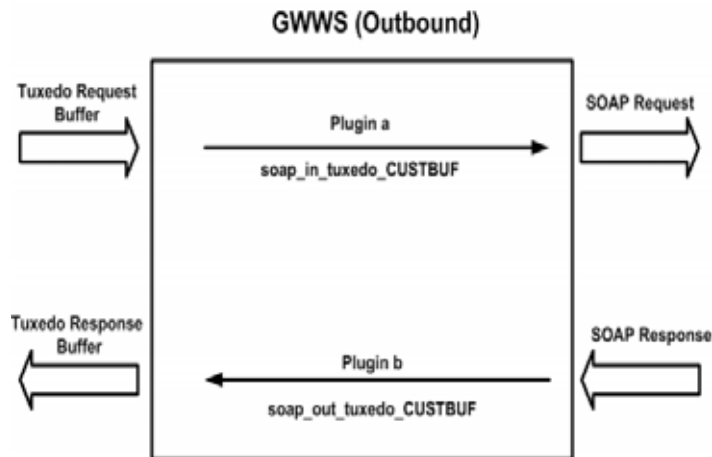
service. If there is an associated instance, the GWWS invokes the customized (`*soap_in_tuxedo__CUSTBUF`) function implemented in the plug-in instance.

When an Oracle Tuxedo response buffer is returned from the Oracle Tuxedo service, GWWS tries to find if there is a message conversion plug-in instance associated with the output message conversion of the target service. If there is an associated instance, GWWS invokes the customized function (`*soap_out_tuxedo__CUSTBUF`), implemented in the plug-in instance.

How Message Conversion Plug-in Works in an Outbound Call Scenario

An outbound call scenario is an Oracle Tuxedo program that invokes an external Web service through the SALT gateway. [Figure 5-2](#) depicts message streaming between an Oracle Tuxedo domain and a Web service application.

Figure 5-2 Message Conversion Plug-in Works in an Outbound Call Scenario



When an Oracle Tuxedo request buffer is delivered to the GWWS server, GWWS tries to find if there is a message conversion plug-in instance associated with the input message conversion of the target service. If there is an associated instance, GWWS invokes the customized function (`*soap_out_tuxedo__CUSTBUF`), implemented in the plug-in instance.

When a SOAP response message is returned from the external Web service application, GWWS tries to find if there is a message conversion plug-in instance associated with the output message conversion of the target service. If there is an associated instance, GWWS invokes the customized function (`*soap_in_tuxedo__CUSTBUF`), implemented in the plug-in instance.

When Do We Need Message Conversion Plug-in

Table 5-1 lists several message conversion plug-in use cases.

Table 5-1 Message Conversion Plug-in Use Cases

| | Scenario Description | soap_in_tuxedo_CUSTBUF | soap_out_tuxedo_CUSTBUF | |
|----------------------------------|--|--|---|--|
| Oracle Tuxedo Originated Service | A SOAP message payload is transformed into a custom typed buffer | Required | N/A | |
| | A custom typed buffer is transformed into a SOAP message payload. | N/A | Required | |
| | An Oracle Tuxedo service input and/or output buffer is associated with a customized XML schema definition when a SOAP message payload is being transformed into this buffer. | Non XML typed buffer: Required XML typed buffer: Optional | N/A | |
| | An Oracle Tuxedo service input and/or output buffer is associated with a customized XML schema definition when this buffer is being transformed into a SOAP message payload. | N/A | Non XML typed buffer: Required XML typed buffer:Optional | |
| | All other general cases when a SOAP message payload is being transformed to an Oracle Tuxedo buffer. | Optional | N/A | |
| | All other general cases when an Oracle Tuxedo buffer is being transformed into a SOAP message payload. | N/A | Optional | |

Table 5-1 Message Conversion Plug-in Use Cases

| | Scenario Description | soap_in_tuxedo_CUSTBUF | soap_out_tuxedo_CUSTBUF |
|--------------------------------|--|------------------------|-------------------------|
| Web Service Originated Service | All cases when an Oracle Tuxedo buffer is transformed into a SOAP message payload. | N/A | Optional |
| | All cases when a SOAP message payload is being transformed into an Oracle Tuxedo buffer. | Optional | N/A |

From [Table 5-1](#), the following message conversion plug-ins general rules are applied.

- If an Oracle Tuxedo originated service consumes custom typed buffers, the message conversion plug-in is required. The Oracle Tuxedo framework does not understand custom typed buffer detailed data structure. Therefore SALT default data type conversion rules cannot be applied.
- If the input and/or output (no matter if returned with `TPSUCCESS` or `TPFAIL`) buffer of an Oracle Tuxedo originated service is associated with an external XML Schema, you should develop message conversion plug-ins to handle the transformation manually (unless you are sure that the SALT default buffer type-based conversion rules can handle it correctly).
 - For example, if you associate your own XML Schema with an Oracle Tuxedo service `FML32` typed buffer, you must provide a message conversion plug-in since SALT default data mapping routines may not understand the SOAP message payload structure when trying to convert into the `FML` typed buffer. Contrarily, the SOAP message payload structure converted from the `FML` typed buffer may be tremendously different from the XML shape defined via your own XML Schema.
 - If you associate your own XML Schema with an Oracle Tuxedo service XML typed buffer, most of time you do not have to provide a message conversion plug-in. This is because SALT passes the XML data as is in both message conversion directions.

For more information, see [Configuring a SALT Application](#).

- You can develop message conversion plug-ins for any message-level conversion to replace SALT default message conversion routines as needed.

Developing a Message Conversion Plug-in Instance

Converting a SOAP Message Payload to an Oracle Tuxedo Buffer

The following function should be implemented in order to convert a SOAP XML payload to an Oracle Tuxedo buffer:

```
CustomerBuffer * (* soap_in_tuxedo__CUSTBUF) (void * xercesDOM,  
CustomerBuffer *a, CustType_Ext * extinfo);
```

Synopsis

```
#include <custtype_pi_ex.h>  
  
CustomerBuffer * myxml2buffer (void * xercesDOM, CustomerBuffer *a,  
CustType_Ext * extinfo);
```

`myxml2buffer` is an arbitrary customized function name.

Description

The implemented function should have the capability to parse the given XML buffer and convert concrete data items to an Oracle Tuxedo custom typed buffer instance.

The input parameter, `char * xmlbuf`, indicates a NULL terminated string with the XML format data stream.

Note: The XML data is the actual XML payload for the custom typed buffer, *not* the whole SOAP envelop document or the whole SOAP body document.

The input parameter, `char * type`, indicates the custom type buffer type name, this parameter is used to verify that the GWWS server expected custom typed buffer handler matches the current plug-in function.

The output parameter, `CustomerBuffer *a`, is used to store the allocated custom typed buffer instance. An Oracle Tuxedo custom typed buffer must be allocated by this plug-in function via the `tpalloc()`. Plug-in code is not responsible to free the allocated custom typed buffer, it is automatically destroyed by the GWWS server if it is not used.

Diagnostics

If successful, this function must return the pointer value of input parameter `CustomerBuffer * a`.

If it fails, this function returns NULL as shown in [Listing 5-5](#).

Listing 5-5 Converting XML Effective Payload to Oracle Tuxedo Custom Typed Buffer Pseudo Code

```

CustomerBuffer * myxml2buffer (void * xercesDOM, CustomerBuffer *a,
CustType_Ext * extinfo)
{
    // casting the input void * xercesDOM to class DOMDocument object
    DOMDocument * DOMTree =

    // allocate custom typed buffer via tmalloc
    a->buf = tmalloc("MYTYPE", "MYSUBTYPE", 1024);
    a->len = 1024;

    // fetch data from DOMTree and set it into custom typed buffer
    DOMTree ==> a->buf;
    if ( error ) {
        release ( DOMTree );
        tpfree(a->buf);
        a->buf = NULL;
        a->len = 0;
        return NULL;
    }

    release ( DOMTree );

    return a;
}

```

Tip: Oracle Tuxedo bundled Xerces library can be used for XML parsing. Tuxedo 12cR2 bundles Xerces 1.7 and Tuxedo 9.1 bundles Xerces 2.5

Converting an Oracle Tuxedo Buffer to a SOAP Message Payload

The following function should be implemented in order to convert a custom typed buffer to SOAP XML payload:

```
int (*soap_out_tuxedo__CUSTBUF)(char ** xmlbuf, CustomerBuffer * a, char *
type);
```

Synopsis

```
#include <custtype_pi_ex.h>
```

```
int * mybuffer2xml (char ** xmlbuf, CustomerBuffer *a, char * type);
```

"mybuffer2xml" is the function name can be specified with any valid string upon your need.

Description

The implemented function has the capability to convert the given custom typed buffer instance to the single root XML document used by the SOAP message.

The input parameter (`CustomerBuffer *a`), is used to store the custom typed buffer response instance. Plug-in code is not responsible to free the allocated custom typed buffer, it is automatically destroyed by the GWWS server if it is not used.

The input parameter (`char * type`), indicates the custom typed buffer type name. This parameter can be used to verify if the SALT GWWS server expected custom typed buffer handler matches the current plug-in function.

The output parameter (`char ** xmlbuf`), is a pointer that indicates the newly converted XML payload. The XML payload buffer must be allocated by this function and uses `malloc ()`. Plug-in code is not responsible to free the allocated XML payload buffer, it is automatically destroyed by the GWWS server if it is not used.

Diagnostics

If successful, this function must return 0.

If it fails, this function must return -1 as shown in [Listing 5-6](#).

Listing 5-6 Converting Oracle Tuxedo Custom Typed Buffer to SOAP XML Pseudo Code

```
int mybuffer2xml (void ** xercesDom, CustomerBuffer *a, CustType_Ext *
extinfo)
{
    // Use DOM implementation to create the xml payload
    DOMTree = CreateDOMTree( );

    if ( error )
        return -1;
```

```

// fetch data from custom typed buffer instance,
// and add data to DOMTree according to the client side needed
// XML format

a->buf ==> DOMTree;

// allocate xmlbuf buffer via malloc
* xmlbuf = malloc( expected_len(DOMTree) );
if ( error ) {
    release ( DOMTree );
    return -1;
}

// casting the DOMDocument to void * pointer and returned
DOMTree >> (* xmlbuf);
if ( error ) {
    release ( DOMTree );
    free ( (* xmlbuf) );
    return -1;
}

return 0;
}

```

WARNING: The GWWS framework is responsible for releasing the `DOMDocument` created inside the plug-in function. To avoid double release, you must pay attention to the following Xerces API usage:

If the `DOMDocument` is constructed from an XML string through `XercesDOMParser::parse()` API. You must use `XercesDOMParser::adoptDocument()` to get the pointer of the `DOMDocument` object. You must not use `XercesDOMParser::getDocument()` to get the pointer of the `DOMDocument` object because the `DOMDocument` object is maintained by the `XercesDOMParser` object and is released when deleting the `XercesDOMParser` object if you do not de-couple the `DOMDocument` from the `XercesDOMParser` via the `XercesDOMParser::getDocument()` function.

SALT 1.1 Custom Buffer Type Conversion Plug-in Compatibility

SALT 1.1 Custom Buffer Type Conversion Plug-in provides a customized message conversion mechanism only for Oracle Tuxedo custom buffer types.

[Table 5-2](#) compares the SALT Message Conversion Plug-in and the SALT 1.1 Custom Buffer Type Conversion Plug-in.

Table 5-2 SALT 12cR2 Message Conversion Plug-in / SALT 1.1 Custom Buffer Type Conversion Plug-in Comparison

| SALT 1.1 Custom Buffer Type Plug-in | SALT 12cR2 Message Conversion Plug-in |
|---|---|
| Plug-in ID is "P_CUSTOM_TYPE" | Plug-in ID is "P_CUSTOM_TYPE" |
| Plug-in Name must be the same as the supported custom buffer type name. | Plug-in Name can be any meaningful value, which is only used to distinguish from other plug-in instances. |
| Only supports message conversion between SOAP message payloads and Oracle Tuxedo custom buffer types. | Supports message conversion between SOAP message payloads and any kind of Oracle Tuxedo buffer type. |
| Buffer type-level association. Each plug-in instance must be named the same as the supported custom buffer type name. Each custom buffer type can only have one plug-in implementation. One custom buffer type can associate with a plug-in instance, and used by all the services. | Message-level association. Each Oracle Tuxedo service can associate plug-in instances with its input and/or output buffers respectively through the plug-in instance name. |
| SOAP message payload is saved as a NULL terminated string for plug-in programming. | SOAP message payload is saved as a Xerces DOM Document for plug-in programming. |

Please note that the SALT 1.1 Custom Buffer Type Plug-in shared library cannot be used directly in SALT 12cR2. You must perform the following tasks to upgrade it to a SALT 12cR2 message conversion plug-in:

1. Re-implement function (`*soap_in_tuxedo__CUSTBUF`) and (`*soap_out_tuxedo__CUSTBUF`) according to the SALT message conversion plug-in

`vtable` function prototype API. The major change is that the SOAP message payload is saved as an Xerces class `DOMDocument` object instead of the old string value.

2. Re-compile your functions as a shared library and configure this shared library in the `SALT Deployment` file so that it can be loaded by GWWS servers.

Tip: You do not have to manually associate the upgraded message conversion plug-ins with service buffers. If a custom typed buffer is involved in the message conversion at runtime, GWWS can automatically search a message conversion plug-in that has the same name as the buffer type name if no explicit message conversion plug-in interface is configured.

Programming Outbound Authentication Plug-Ins

When an Oracle Tuxedo client accesses Web services via SOAP/HTTP, the client may be required to send a username and password to the server to perform HTTP Basic Authentication. The Oracle Tuxedo clients uses `tpinit()` to send a username and password when registering to the Oracle Tuxedo domain. However, this username is used by Oracle Tuxedo and is not the same as the one used by the Web service (the password may be different as well).

To map the usernames, SALT provides a plug-in interface (Credential-Mapping Interface), that allows you to choose which username and password is sent to the Web service.

How Outbound Authentication Plug-Ins Work

When an Oracle Tuxedo client calls a Web service, it actually calls the GWWS server that declares the Web service as an Oracle Tuxedo service. The user id and group id (defined in `tpusr` and `tpgrp` files) are sent to the GWWS. The GWWS then checks whether the Web service has a configuration item `<Realm>`. If it does, the GWWS:

- tries to invoke the `vtable gwws_pi_map_http_basic` function to map the Oracle Tuxedo userid into the username and password for the HTTP Realm of the server.
- for successful calls, encodes the returned username and password with `Base64` and sends it to the HTTP header field “Authorization: Basic”.
- for failed calls, returns a failure to the Oracle Tuxedo Client without invoking the Web service.

Implementing a Credential Mapping Interface Plug-In

Using the following scenario:

- An existing Web service, `myservice`, sited on `http://www.abc.com/webservice`, requires HTTP Basic Authentication. The username is “test”, the password is “1234,” and the realm is “myrealm”.
- After converting the Web service WSDL into the SALT configuration file (using `wsd1cvt`), add the `<Realm>myrealm</Ream>` element to the endpoint definition in the WSDL file.

Perform the following steps to implement a SALT plug-in interface:

1. Write the functions to map the “myrealm” Oracle Tuxedo UID/GID to username/password on `www.abc.com`.

- Use `Credmap_HTTP_Basic()`;

This function is used to return the HTTP username/password. The function prototype defined in `credmap_pi_ex.h`

2. Write the following three plug-in register functions. For more information, see [“Plug-In Register Functions”](#).

- `_ws_pi_init_P_CREDENMAP_TEST(char * params, void ** priv_ptr);`

This function is invoked when the GWWS server attempts to load the plug-in shared library during startup.

- `_ws_pi_exit_P_CREDENMAP_TEST(void * priv);`

This function is invoked when the GWWS server unloads the plug-in shared library during the shutdown phase.

- `_ws_pi_set_vtbl_P_CREDENMAP_TEST(void * vtbl);`

Set the `gwws_pi_map_http_basic` entry in `vtbl` structure `credmap_vtbl` with the `Credmap_HTTP_Basic()` function implemented in step 1.

3. You can also write the optional function:

- `_ws_pi_get_Id_and_Names(int * params, char ** ids, char ** names);`

This function is invoked when the GWWS server attempts to load the plug-in shared library during startup to determine what library interfaces are implemented. For more information, see [“Plug-In Register Functions”](#).

4. Compile the previous four or five functions into one shared library, `credmap_plugin.so`.
 5. Configure the plug-in interface in the SALT deployment file.
- Configure the plug-in interface as shown in [Listing 5-7](#).

Listing 5-7 Custom Typed Buffer Plug-In Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<Deployment xmlns="http://www.bea.com/Tuxedo/SALTDEPLOY/2007">
    . . . . .
    . . . . .
    <System>
        <Plugin>
            <Interface
                id="P_CREDENMAP"
                name="TEST"
                library="credmap_plugin.dll" />
        </Plugin>
    </System>
</Deployment>
```

Mapping the Oracle Tuxedo UID and HTTP Username

The following function should be implemented in order to return username/password for HTTP Basic Authentication:

```
typedef int (* GWWS_PI_CREDMAP_PASSTEXT) (char * domain, char * realm, char
* t_userid, char * t_grpid, Cred_UserPass * credential);
```

Synopsis

```
#include <credmap_pi_ex.h>
typedef struct Cred_UserPass_s {
    char username[UP_USERNAME_LEN];
    char password[UP_PASSWORD_LEN];
} Cred_UserPass;

int gwws_pi_map_http_basic (char * domain, char * realm, char * t_uid, char
* t_gid, Cred_UserPass * credential);
```

The "gwws_pi_map_http_basic" function name can be specified with any valid string as needed.

Description

The implemented function has the capability to determine authorization credentials (usernames and passwords) used for authorizing users with a given Oracle Tuxedo uid and gid for a given domain and realm.

The input parameters, `char * domain` and `char * realm`, represent the domain name and HTTP Realm that the Web service belongs to. The plug-in code must use them to determine the scope to find appropriate credentials.

The input parameters, `char * t_uid` and `char * t_gid`, are strings that contain Oracle Tuxedo user ID and group ID number values respectively. These two parameters may be used to find the username.

The output parameter, `Cred_UserPass * credential`, is a pointer that indicates a pre-allocated buffer storing the returned username/password. The plug-in code is not responsible for allocating the buffer.

Notes: Oracle Tuxedo user ID is available only when `*SECURITY` is set as `USER_AUTH` or higher in the `UBBCONFIG` file. Group ID is available when `*SECURITY` is set as `ACL` or higher. The default is "0".

Diagnostics

If successful, this function returns 0. If it fails, it returns -1 as shown in [Listing 5-8](#).

Listing 5-8 Credential Mapping for HTTP Basic Authentication Pseudo Code

```
int Credmap_HTTP_Basic(char * domain, char * realm, char * t_uid, char *
t_gid, Cred_UserPass * credential)
{
    // Use domain and realm to determine scope
    credentialList = FindAllCredentialForDomainAndRealm(domain, realm);

    if ( error happens )
        return -1;

    // find appropriate credential in the scope
```

```
foreach cred in credentialList {
    if (t_uid and t_gid match) {
        *credential = cred;
        return 0;
    }
}
if ( not found and no default credential) {
    return -1;
}

*credential = default_credential;
return 0;
}
```

Tip: The credentials can be stored in the database with domain and realm as the key or index.
