BEA SALT™

Programming Web Services

Version 1.1
Document Revised: August 16, 2006

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

BEA SALT Client Programming

Using Plug-ins with BEA SALT

CHAPTER 1

BEA SALT Client Programming

This chapter contains the following client programming topics:

- Overview
- BEA SALT Web Service Client Programming Tips
- BEA SALT Sample Applications
- Web Service Client Programming References

Overview

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

To develop a Web service client program, follow these steps:

1. Generate or download the BEA SALT WSDL document/file. See Configuring BEA SALT in the *BEA SALT Administration Guide* for more information on generating the BEA SALT WSDL document.
2. Use a Web service client-side toolkit to parse the SALT WSDL document and generate client stub code. See BEA SALT Web Service Client Programming Tips.

3. Write client-side application code to invoke a BEA SALT Web service using the functions defined in the client-generated stub code.
4. Compile and run your client application.

BEA SALT Web Service Client Programming Tips

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

- BEA WebLogic Web Service Client Programming Toolkit
- Apache Axis for Java Web Service Client Programming Toolkit
- Microsoft .NET Web Service Client Programming Toolkit

See Interoperability Considerations in the *BEA SALT Administration Guide* for more information.

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

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

BEA 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 program and server instances. For more information, see `http://edocs.bea.com/wls/docs91/webserv/client.html#standalone_invoke`.

Besides traditional synchronous message exchange mode, BEA 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. For more information, see "BEA SALT Sample Applications" on page 1-7.

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

BEA SALT can map 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 is, in an enterprise service oriented environment, 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 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 SALT_CONFFILE -m raw -t wls
```

Apache Axis for Java Web Service Client Programming Toolkit

BEA 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. For more information, see "BEA SALT Sample Applications" on page 1-7.

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

BEA SALT supports HTTP MIME transportation for Tuxedo CARRAY data. To achieve this, 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 SALT_CONFFILE -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. BEA SALT does not support multiple-reference format. You can disable AXIS multiple-reference format as shown in Listing 1-1:

Listing 1-1 Disabling AXIS Multiple-Reference Format

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

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

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

A sample Apache Sandensha asynchronous mode and `send offer` code example is shown in Listing 1-2:

Listing 1-2 Sample Apache Sandensha 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:wsrm: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 BEA 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 BEA SALT WSDL document to generate the stub code, and then reference the .Net object contained in the stub code/binary in business logic implementations. For more information, see "BEA SALT Sample Applications" on page 1-7.

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.

BEA 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 BEA SALT WSDL document is generated in RPC/encoded style, .Net will send out SOAP messages containing `soapenc:arrayType`. BEA SALT does not support `soapenc:arrayType` in RPC/encoded style. A sample RPC/encoded style-generated WSDL document is shown in Listing 1-3.

Listing 1-3 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 BEA SALT WSDL document in RPC/encoded style, `wsdl.exe` can generate stub code, but 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.

BEA SALT Sample Applications

BEA SALT bundles six Web service sample applications to demonstrate how to invoke BEA SALT using BEA WebLogic, Apache Axis or Microsoft .NET toolkits.

After BEA SALT is installed, you can find the following sample applications for your reference in the BEA SALT sample directory (each sample contains a detailed `readme` file):

- Basic Sample
- Attachment Sample
- Custom Type Sample
- Data Type Sample
- Reliable Messaging Sample
- Security Sample

Note: UNIX samples: `$TUXDIR/samples/salt`

Windows samples: `%TUXDIR%\samples\salt`

Basic Sample

The Basic Sample demonstrates how to export a simple Tuxedo service as a Web service. The Tuxedo `simpapp` sample is used as an existing application to be exported as a Web service.

This sample contains all needed files to configure and export the `simpserv` server TOUPPER service as a Web service. The Web service accepts a single a string parameter and converts it to uppercase. The client calls the service, and then prints the returned string.

This sample will enable you to learn the basics of running and accessing the GWWS server and the Web Services it provides.

Applicable Client Programs(s): BEA WebLogic, Apache Axis for Java, Microsoft .NET.

Attachment Sample

The Attachment Sample demonstrates how to transport CARRAY buffer types as MIME attachments according to SwA Protocol (SOAP with Attachment) in a SALT Web service. The Tuxedo `simpapp` sample is used as an existing application to be exported as a Web service. This sample contains all needed files to configure and export the `simpserv` server `TOUPPER` service as a Web Service.

Applicable Client Programs(s): BEA WebLogic.

Custom Type Sample

The BEA Salt 1.1 Custom Type Plug-in Sample demonstrates how to use Salt 1.1 plug-in extension mechanisms to implement customized mapping rules between Tuxedo Custom Typed Buffers and XML documents.

Applicable Client Programs(s): BEA WebLogic.

Data Type Sample

The Date Type Sample demonstrates how Tuxedo typed buffer are used in BEA SALT. In this sample shows how the FML and VIEW buffers and their sub-fields are defined in the Tuxedo Service Metadata Repository, and represented in a WSDL document. The WSDL document file generation utility, `tmwsdlgen` is used in this sample.

A WebLogic client program is also provided in this sample to help you get familiar with Web service client programming. The Tuxedo application server is a simple echo service in which the FML/VIEW buffer are checked and return the input data.

Applicable Client Programs(s): BEA WebLogic.

Reliable Messaging Sample

This ReliableMessaging Sample demonstrates how to use BEA SALT WS-Reliable Messaging support and asynchronous communication with WS-Addressing. The Tuxedo `bankapp` sample is used as the Tuxedo application service provider. A WebLogic Server Web service client and standalone java Web service client are also included in this sample.

For more detailed WebLogic Server reliable messaging usage information, see http://e-docs.bea.com/wls/docs91/webserv/advanced.html#reliable_messaging.

Applicable Client Programs(s): BEA WebLogic.

Security Sample

The Security Sample leverages the existing Tuxedo `xmlstockapp` sample in a stock price query scenario. The STOCKQUOTE service is exported as a Web service by the GWWS server.

BEA SALT uses SSL/HTTPS to secure transport and message. It also supports Tuxedo authentication with HTTP Basic Authentication. You will learn how to configure security transport and how to authenticate using two Tuxedo authentication patterns: application password and user authentication.

A client program can be developed from the code in the sample combined with stub codes generated from the WSDL document. The WSDL document file is generated using the SALT configuration file and the `tmwsdlgen` utility.

Applicable Client Programs(s): BEA WebLogic.

Web Service Client Programming References

- BEA WebLogic 9.1 Web Service Client Programming References

  Invoking a Web service from a Stand-alone Client: Main Steps

- 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

CHAPTER 2

Using Plug-ins with BEA SALT

This chapter contains the following topics:

Overview

BEA SALT GWWS server is a configuration-driven process which, for most basic Web service applications, does not require any programming tasks. However, BEA 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.

Understanding BEA SALT Plug-ins

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. BEA 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 configuration 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 BEA SALT. Arbitrary string values are not permitted.

BEA SALT 1.1 only supports the `P_CUSTOM_TYPE` plug-in ID, which is used to define plug-in interfaces for custom typed buffer data handling.

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 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_CUSTOM_TYPE` ID category, two functions need to be implemented:

- `CustomerBuffer *(*soap_in_tuxedo__CUSTBUF)(char *, CustomerBuffer *, char *);`
- `int (*soap_out_tuxedo__CUSTBUF)(char **, CustomerBuffer *,  char *);`

For more information, see "Programming Plug-ins for Custom Typed Buffer Data" on page 2-7.

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 function These functions are:

- Initiating Function
- Exiting Function
- vtable Setting Function

Initiating Function

The initiating function is immediately invoked after the plug-in shared library is loaded during GWWS server startup. Developers 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 businesss 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 the BEA SALT 1.1 release, `P_CUSTOM_TYPE` is the only valid plug-in ID.

The vtable structure for `P_CUSTOM_TYPE` plug-in interfaces is shown in Listing 2-1.

Listing 2-1 custtype_vtable Structure

```
struct custtype_vtable {
       CustomerBuffer *(*soap_in_tuxedo__CUSTBUF)(char *, CustomerBuffer *,
char *);
       int (*soap_out_tuxedo__CUSTBUF)(char **, CustomerBuffer *,  char *);
};
```

`struct custtype_vtable` indicates that two functions need to be implemented for a `P_CUSTOM_TYPE` plug-in interface. For more information, see "Programming Plug-ins for Custom Typed Buffer Data" on page 2-7.

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

An example of setting the vtable structure with the actual functions within the vtable setting function is shown in Listing 2-2.

Listing 2-2 Setting the vtable Structure with Actual functions within the vtable Setting Function

```
int _DLLEXPORT_ _ws_pi_set_vtbl_P_CUSTOM_TYPE_MyType (void * vtbl)
```

```
{
        struct custtype_vtable * vtable;
        if ( ! vtbl )
                return -1;

        vtable = (struct custtype_vtable *) vtbl;

        vtable->soap_in_tuxedo__CUSTBUF = ConvertXML_2_MyType;
        vtable->soap_out_tuxedo__CUSTBUF = ConvertMyType_2_XML;

        userlog(" setup vtable for custom type %s", type_name);

        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 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" on page 2-2.
2. Write C language plug-in register functions that include: the initiating function, the exiting function and the vtable setting function. 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" on page 2-3.
3. Compile all the above functions into one shared library.

Defining a Plug-in interface in SALT configuration file

To define a plug-in shared library that is loaded by the GWWS server, the corresponding plug-in interface information must be configured in the BEA SALT configuration file. For more information, see Configuring BEA SALT in the BEA Salt Administration Guide.

An example of how to define plug-in information in the BEA SALT configuration file is shown in Listing 2-3.

Listing 2-3 Defined Plug-In in the BEA SALT Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration xmlns="http://www.bea.com/Tuxedo/Salt/200606">
       <Servicelist id="sample">
              <Service name="custom_type_svc1"/>
       </Servicelist>
       <Policy/>
       <System>
              <Plugin>
                     <Interface>
                            <ID>P_CUSTOM_TYPE</ID>
                            <Name>MYTYPE</Name>
                            <Library>mytype_plugin.so</Library>
                     </Interface>
              </Plugin>
       </System>
       <WSGateway>
           <GWInstance id="GWWS1">
                     <HTTP address="//my host"/>
           </GWInstance>
       </WSGateway>
</Configuration>
```

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 Plug-ins for Custom Typed Buffer Data

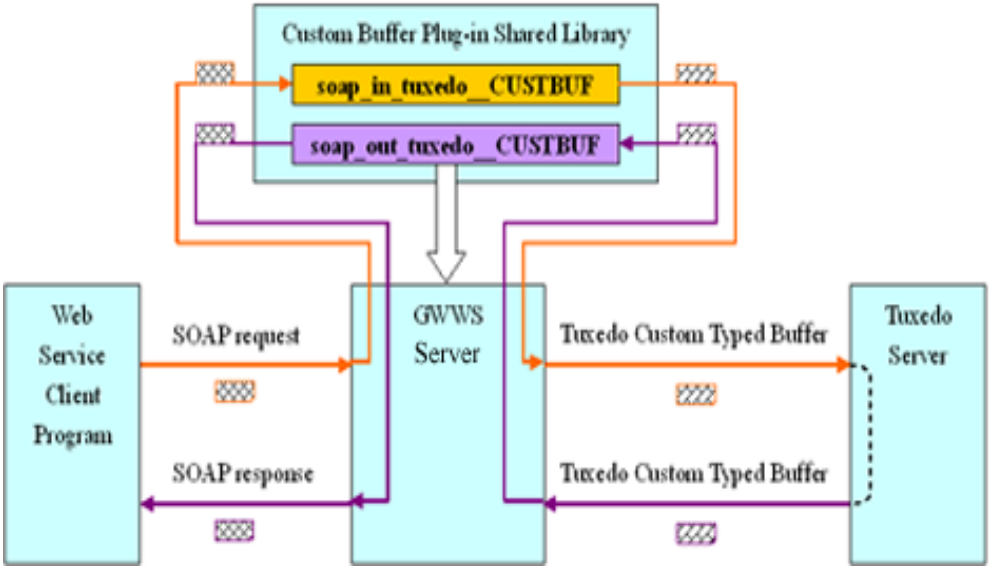
Tuxedo allows developers to customize their own typed buffers. If you are already using your own custom typed buffers, the BEA SALT plug-in mechanism provides a way to convert SOAP XML payloads to and from custom typed buffers.

For more information, see Customizing a Buffer in *Programming a Tuxedo ATMI Application Using C*.

How Tuxedo Custom Typed Buffer Plug-ins Work

Figure 2-1 depicts custom typed buffer data streaming between a Web service client program and a Tuxedo domain.

Figure 2-1 Web Service Client Program and Tuxedo Domain Custom Typed Buffer Data Streaming



When a Tuxedo service requires an input custom typed buffer, the GWWS server automatically looks for the proper custom type name vtable structure:

- The vtable `soap_in_tuxedo__CUSTBUF` function is invoked to convert the SOAP XML payload to a custom typed buffer instance.

- The vtable `soap_out_tuxedo__CUSTBUF` function is invoked to convert a custom typed buffer instance to a SOAP XML payload.

Developing a BEA SALT Plug-in Interface for Tuxedo Custom Typed Buffers

Using the following scenario:

- An existing Tuxedo service, `myservice`, accepts and returns the custom typed buffer `mytype`.
- The typed buffer switch for handling `mytype` has been added to the Tuxedo `libbuft` library.

  Note: For more information, see Programming a BEA Tuxedo ATMI Application Using C.

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

1. Write two functions to convert the SOAP XML payload and `mytype`

- `ConvertXML2MyType();`

  This function is used to convert the SOAP XML payload into a `mytype` instance.

  For more information, see "Converting an XML Effective Payload to a Tuxedo Custom Typed Buffer" on page 2-12

- `ConvertMyType2XML();`

  This function is used to convert a `mytype` typed buffer instance to a SOAP XML payload.

  For more information, see "Converting a Tuxedo Custom Typed Buffer to a SOAP XML Payload" on page 2-14

2. Write three plug-in register functions for `mytype`

- `_ws_pi_init_P_CUSTOM_TYPE_mytype(void * priv);`

  This function is invoked when the GWWS server attempts to load the plug-in shared library during startup. For more information, see "Plug-in Register Functions" on page 2-3.

- `_ws_pi_exit_P_CUSTOM_TYPE_mytype(void * priv);`

  This function is invoked when the GWWS server unloads the plug-in shared library during the shutdown phase. For more information, see "Plug-in Register Functions" on page 2-3.

- `_ws_pi_set_vtbl_P_CUSTOM_TYPE_mytype(void * priv);`

Set the vtable structure `custtype_vtable` with the two functions you implemented in step 1 (`ConvertXML2MyType()` and `ConvertMyType2XML()`). For more information, see "Plug-in Register Functions" on page 2-3.

3. Compile the previous five functions into one shared library, `mytype_plugin.so`.
4. Configure the plug-in interface in the SALT configuration file

Configure the plug-in interface as shown in Listing 2-4.

Listing 2-4 Custom Typed Buffer Plug-in Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration xmlns="http://www.bea.com/Tuxedo/Salt/200606">
       <Servicelist id="sample">
              <Service name="myservice"/>
       </Servicelist>
       <Policy/>
       <System>
              <Plugin>
                     <Interface>
                            <ID>P_CUSTOM_TYPE</ID>
                            <Name>mytype</Name>
                            <Library>mytype_plugin.so</Library>
                     </Interface>
              </Plugin>
       </System>
       <WSGateway>
           <GWInstance id="GW1">
                     <HTTP address="//host:5001"/>
           </GWInstance>
       </WSGateway>
</Configuration>
```

Tuxedo Custom Typed Buffer XML Data Representation

In the BEA SALT generated WSDL document, the XML Schema built-in type `xsd:anyType` is used to represent a Tuxedo custom typed buffer. This allows arbitrary content to be encapsulated within the SOAP message, except for the following format restrictions:

1. Since `xsd:anyType` is the schema type for the SOAP body `tuxtype:inbuf` or `tuxtype:outbuf` element, the effected XML payload for custom typed buffers must be the content encapsulated in the `<inbuf>` or `<outbuf>` SOAP elements.
2. The effective content encapsulated in the SOAP message for Tuxedo custom typed buffers must be a single root XML buffer.
3. The effective content encapsulated in the SOAP message for Tuxedo custom typed buffers must not include an XML prolog.

Listing 2-5 A Valid SOAP Message Carrying Custom Typed Buffer Data (XML)

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Header/>
   <soapenv:Body>
      <m:CALC24 xmlns:m="urn:pack.custtypeapp_typedef.salt11">
         <m:inbuf>
            <poin:point24 xmlns:poin="http://www.example.org/Point24">
               <poin:p>3</poin:p>
               <poin:p>5</poin:p>
               <poin:p>7</poin:p>
               <poin:p>2</poin:p>
            </poin:point24>
         </m:inbuf>
      </m:CALC24>
   </soapenv:Body>
</soapenv:Envelope>
```

Listing 2-6 An Invalid SOAP Message Carrying Custom Typed Buffer Data (Payload is Not a Single Root XML Document)

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Header/>
   <soapenv:Body>
      <m:CALC24 xmlns:m="urn:pack.custtypeapp_typedef.salt11">
         <m:inbuf>
            <poin:point24 xmlns:poin="http://www.example.org/Point24">
               <poin:p>1</poin:p>
               <poin:p>1</poin:p>
               <poin:p>2</poin:p>
               <poin:p>3</poin:p>
            </poin:point24>
            <poin:point24 xmlns:poin="http://www.example.org/Point24">
               <poin:p>2</poin:p>
               <poin:p>3</poin:p>
               <poin:p>5</poin:p>
               <poin:p>8</poin:p>
            </poin:point24>
         </m:inbuf>
      </m:CALC24>
   </soapenv:Body>
</soapenv:Envelope>
```

Converting an XML Effective Payload to a Tuxedo Custom Typed Buffer

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

```
CustomerBuffer * (* soap_in_tuxedo__CUSTBUF) (char *xml, CustomerBuffer *a,
char *type);
```

Synopsis

```
#include <custtype_pi_ex.h>

CustomerBuffer * myxml2buffer (char * xmlbuf, CustomerBuffer *a, char *
type);
```

`myxml2buffer` is the function name of any user-created valid string.

Description

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

The input parameter, `char * xmlbuf`, indicates a NULL terminated string with the XML format data stream. Please note that 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 typed 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. A Tuxedo custom typed buffer must be allocated by this plug-in function via the ATMI function `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.

Listing 2-7 Converting XML Effective Payload to Tuxedo Custom Typed Buffer Pseudo Code

```
CustomerBuffer * myxml2buffer (char * xmlbuf, CustomerBuffer *a, char *
type)
{
       // Use DOM implementation to parse the xml payload
       //SAX can be used as an alternative.
       DOMTree = ParseXML( xmlbuf );

       if ( error )
              return NULL;

       // allocate custom typed buffer via tpalloc
       a->buf = tpalloc("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: Tuxedo bundled Xerces library can be used for XML parsing. Tuxedo 8.1 bundles Xerces 1.7 and Tuxedo 9.1 bundles Xerces 2.5

Converting a Tuxedo Custom Typed Buffer to a SOAP XML 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 use the `malloc ()` system API. 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 returns 0.

If it fails, this function must return -1.

Listing 2-8 Converting Tuxedo Custom Typed Buffer to SOAP XML Pseudo Code

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

```
        // 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;
        }

        // serialize DOMTree as xml string
        DOMTree >> (* xmlbuf);
        if ( error ) {
                release ( DOMTree );
                free ( (* xmlbuf) );
                return -1;
        }

        release ( DOMTree );
        return 0;
}
```

Tip: The Tuxedo bundled Xerces library can be used to create DOM tree and transform to XML data stream.

Tuxedo 8.1 bundles Xerces 1.7 and Tuxedo 9.1 bundles Xerces 2.5.

Using Customized XML Schema to Extend Default SALT WSDL

The default SALT WSDL document uses `xsd:anyType` to represent custom typed buffer XML data. `xsd:anyType` is not well supported by some Web service client-side toolkits. Even for those Web service toolkits that support `xsd:anyType`, client-side programming is still a complex programming task.

Extending the default SALT WSDL document by replacing `xsd:anyType` with a customized XML Schema is highly recommended.

To extend the default SALT WSDL document with a customized XML Schema, do the following:

1. Construct your customized XML Schema file
2. Use Document/literal encoded style to generate the SALT WSDL document.

   Note: BEA SALT only supports extending a Document/literal encoded style WSDL document.

3. Modify the BEA SALT WSDL document by importing the customized XML Schema and replacing `xsd:anyType` with the customized XML Schema type.

Listing 2-9 and Listing 2-10 depict samples of an original SALT WSDL document and an extended WSDL document with a customized XML Schema respectively.

Listing 2-9 Original Default SALT WSDL Document

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ......>
       ......
       <wsdl:types>
              <xsd:schema attributeFormDefault="unqualified"
                                  elementFormDefault="qualified"

targetNamespace="urn:pack.custtypeapp_typedef.salt11">
                     <xsd:element name="CALC24">
                            <xsd:complexType>
                                   <xsd:sequence>
                                          <xsd:element name="inbuf"
type="xsd:anyType"></xsd:element>
```

```
                                    </xsd:sequence>
                              </xsd:complexType>
                        </xsd:element>
                        <xsd:element name="CALC24Response">
                              <xsd:complexType>
                                    <xsd:sequence>
                                          <xsd:element name="outbuf"
type="xsd:anyType"></xsd:element>
                                    </xsd:sequence>
                              </xsd:complexType>
                        </xsd:element>
                  </xsd:schema>
      </wsdl:types>

      <wsdl:message name="CALC24Input">
            <wsdl:part element="tuxtype:CALC24"
name="parameters"></wsdl:part>
      </wsdl:message>
      <wsdl:message name="CALC24Output">
            <wsdl:part element="tuxtype:CALC24Response"
name="parameters"></wsdl:part>
      </wsdl:message>

      <wsdl:portType ...>
            ......
      </wsdl:portType>
      ......
</wsdl:definitions>
```

Listing 2-10 Extended WSDL Document

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ......>
      ......
      <wsdl:import namespace= "http://www.example.org/Point24"
```

```
location="file:///home/user/bea/tuxedo8.1/samples/salt/custtypeapp/Point24
.xsd">
       </wsdl:import>
       <wsdl:types>
              <xsd:schema attributeFormDefault="unqualified"
                                  elementFormDefault="qualified"

targetNamespace="urn:pack.custtypeapp_typedef.salt11">
                     <xsd:element name="CALC24">
                            <xsd:complexType>
                                   <xsd:sequence>
                                          <xsd:element name="inbuf"
type="m:Point24"

xmlns:m="http://www.example.org/Point24">
                                          </xsd:element>
                                   </xsd:sequence>
                            </xsd:complexType>
                     </xsd:element>
                     <xsd:element name="CALC24Response">
                            <xsd:complexType>
                                   <xsd:sequence>
                                          <xsd:element name="outbuf"
type="m:Point24"

xmlns:m="http://www.example.org/Point24">
                                          </xsd:element>
                                   </xsd:sequence>
                            </xsd:complexType>
                     </xsd:element>
              </xsd:schema>
       </wsdl:types>
       <wsdl:message name="CALC24Input">
              <wsdl:part element="tuxtype:CALC24"
name="parameters"></wsdl:part>
       </wsdl:message>
       <wsdl:message name="CALC24Output">
              <wsdl:part element="tuxtype:CALC24Response"
```

```
name="parameters"></wsdl:part>
        </wsdl:message>
        <wsdl:portType ......>
                ......
        </wsdl:portType>
        ......
</wsdl:definitions>
```

BEA SALT Custom Typed Buffer Sample

BEA SALT product distribution bundles a sample application that demonstrates how to write plug-in shared library for custom typed buffer data conversion. This sample also demonstrates how to use a customized XML Schema to extend the WSDL document in order to make client/server-side programming more convenient. For more information, see BEA SALT Sample Applications.