



BEA SALT™

Administration Guide

Contents

BEA SALT Administration Overview

Basic Concepts for Administering BEA SALT	1-1
BEA SALT Administrative Tasks and Tools	1-4

Setting Up a BEA SALT Application

Using Tuxedo Service Metadata Repository for BEA SALT	2-1
Configuring Native Tuxedo Services	2-9
Configuring External Web Services	2-14
Creating the SALT Deployment File	2-16
Configuring Advanced Web Service Messaging Features	2-22
Configuring Security Features	2-28
Compiling SALT Configuration	2-32
Configuring the UBBCONFIG File for BEA SALT	2-33
Configuring BEA SALT In Tuxedo MP Mode	2-38
Migrating from BEA SALT 1.1	2-39

Administering BEA SALT at Run Time

Browsing to the WSDL Document from the GWWS Server	3-1
Tuning the GWWS Server	3-3
Tracing the GWWS Server	3-4
Monitoring the GWWS Server	3-7
Troubleshooting BEA SALT	3-9

BEA SALT Administration Overview

The following sections provide an overview to BEA SALT administration topics:

- [Basic Concepts for Administering BEA SALT](#)
- [BEA SALT Administrative Tasks and Tools](#)

Basic Concepts for Administering BEA SALT

This section explains the following basic concepts for administering BEA SALT:

- [Tuxedo Service Metadata](#)
- [BEA SALT Deployment Model](#)

Tuxedo Service Metadata

Starting with the BEA Tuxedo 9.0 release, the Tuxedo Service Metadata Repository was developed to facilitate saving and retrieving Tuxedo service metadata. Tuxedo service metadata is a collection of Tuxedo service attributes that are especially useful in describing the request/response details of a Tuxedo service. The BEA SALT gateway server (GWWS), relies on the Tuxedo Service Metadata Repository for conversions between the Tuxedo request/response format (buffer types) and standard SOAP message format.

When exposing Tuxedo services as Web services using BEA SALT, you must define and load your Tuxedo service metadata in the Tuxedo Service Metadata Repository. BEA SALT can then define the corresponding SOAP message format from the Tuxedo service metadata.

When invoking external Web services from a Tuxedo application, BEA SALT provides a WSDL file converter, `wsdlcvt`. This command utility helps you to define Tuxedo service metadata from each Web service operation. The converted services are called SALT proxy services and can be invoked as normal Tuxedo services.

Note: These services also need to be loaded in the Tuxedo Service Metadata Repository.

To retrieve the Tuxedo service metadata information, you must configure the Tuxedo Service Metadata Repository system server (`TMMETADATA`), to be booted in the Tuxedo application.

Note: `TMMETADATA` must be booted prior to using any BEA SALT gateway `GWWS` server.

For more information, see [“Tuxedo Service Metadata Repository”](#) and [“Using Tuxedo Service Metadata Repository for BEA SALT”](#) on page 2-1.

BEA SALT Deployment Model

Deploying the current BEA SALT version requires two configuration file types:

- SALT Deployment File (`SALTDEPLOY`)
- SALT Web Service Definition File (`WSDF`)

SALT Web Service Definition File

The SALT Web Service Definition File (`WSDF`) is an XML-based file used to define SALT Web service components (Web Service Bindings, Web Service Operations, Web Service Policies, and so on). The `WSDF` is a BEA SALT specific representation of the Web Service Definition Language data model. There are two `WSDF` types: native and non-native.

- Native `WSDF`

A native `WSDF` is created manually. You must define a set of Tuxedo services and how they are exposed as Web services in the `WSDF`. It looks similar to the SALT 1.1 configuration file. The native `WSDF` is the input file for the SALT WSDL generator (`tmwsdlgen`). For more information, see [“Configuring Native Tuxedo Services”](#) on page 2-9.

- Non-native `WSDF`

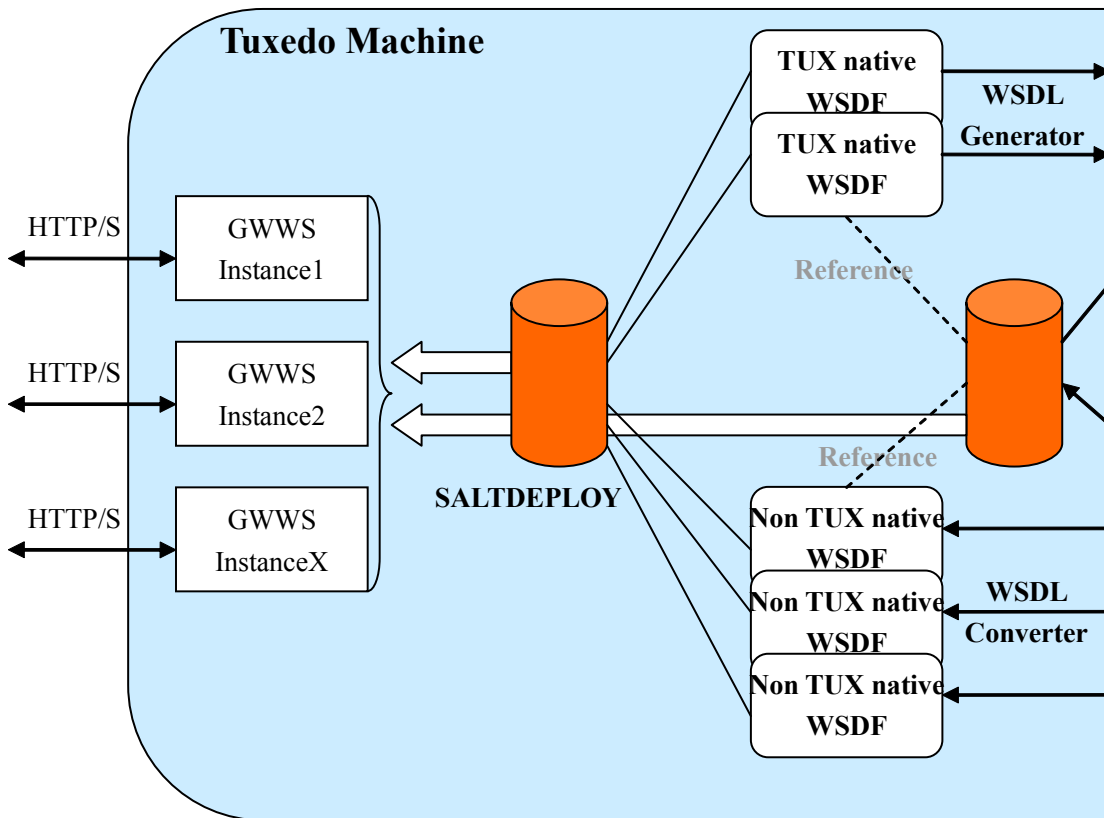
A non-native `WSDF` is generated from an external WSDL file that has been converted using the SALT WSDL converter (`wsdlcvt`). Basically, you do not need to change the generated `WSDF` (except to configure advanced features). For more information, see [“Configuring External Web Services”](#) on page 2-14.

SALT Deployment File

The SALT Deployment File (`SALTDEPLOY`) is an XML-based file used to define BEA SALT GWWS server deployment information on a *per* Tuxedo machine basis. The `SALTDEPLOY` file lists all necessary `WSDF` files. It also specifies how many GWWS servers are deployed on a Tuxedo machine and associates inbound and outbound Web service endpoints for each GWWS server. The `SALTDEPLOY` file contains a system section where global resources are configured (including certificates and plug-in load libraries). For more information, see [“Creating the SALT Deployment File” on page 2-16](#).

Figure 1-1 illustrates the BEA SALT deployment model.

Figure 1-1 SALT Deployment Model



BEA SALT Administrative Tasks and Tools

BEA SALT provides a set of command utilities for managing different parts of a BEA SALT application built on the BEA Tuxedo system. These utilities can be used for the following tasks:

- [Configuring Your SALT Application Using Command-Line Utilities](#)
- [Administering Your SALT Application Using Command-Line Utilities](#)

Configuring Your SALT Application Using Command-Line Utilities

You can configure your BEA SALT application by using command-line utilities. Specifically, you can use an XML editor to create and edit the configuration file (WSDF files and SALTDEPLOY file) for your application, and then use the command-line utility named `wsloadcf` to translate the XML files (SALTDEPLOY file and referenced WSDF files) to a binary file (SALTCONFIG). You are then ready to boot the SALT gateway (GWWS) servers.

The following list identifies BEA SALT command-line utilities that you can use to configure your application:

- `wsloadcf(1)`

A command that is initiated on each Tuxedo machine. It allows you to compile your application SALTDEPLOY file and referenced WSDF files into the binary SALTCONFIG file. The `wsloadcf` command loads the binary file to the location defined by the SALTCONFIG environment variable.

- `wsdlcvt(1)`

A command that converts an external Web Service Description Language (WSDL) file into Tuxedo definition files (WSDF file, Tuxedo Service Metadata definition file, FML32 field table file and XML Schema file). The generated WSDF file is a non-native WSDF file used for SALT outbound calls specifically.

Since BEA SALT built on the BEA Tuxedo framework, you should also use the following BEA Tuxedo provided command-line utilities to configure BEA SALT specific items in a Tuxedo application:

- `tmloadcf(1)`

A command that runs on the master Tuxedo machine. It is used to compile the Tuxedo application UBBCONFIG file into the binary TUXCONFIG file. To boot BEA SALT gateway servers, you must define GWWS servers in the UBBCONFIG file.

- `tmloadrepos (1)`

A command that runs on the machine where Tuxedo Service Metadata Repository System Server (TMMETADATA) is booted. It loads the Tuxedo service metadata definition text files into the binary Tuxedo Service Metadata Repository file. You must load all Tuxedo legacy services that are to be exposed as Web service operations in the Tuxedo Service Metadata Repository. You must also load all `wsdlcvt` generated SALT proxy services in the Tuxedo Service Metadata Repository.

Administering Your SALT Application Using Command-Line Utilities

You can use the command-line utility `wsadmin (1)` to perform administrative functions for BEA SALT gateway servers in your Tuxedo applications. Like the `tmadmin`, `dadmin` and `qadmin` commands, `wsadmin` is an interactive meta-command that enables you to run sub-commands.

In a BEA Tuxedo application, you can run `wsadmin (1)` on any machine to monitor and manage the SALT gateway servers defined in the Tuxedo application.

Setting Up a BEA SALT Application

This section contains the following topics:

- [Using Tuxedo Service Metadata Repository for BEA SALT](#)
- [Configuring Native Tuxedo Services](#)
- [Configuring External Web Services](#)
- [Creating the SALT Deployment File](#)
- [Configuring Advanced Web Service Messaging Features](#)
- [Configuring Security Features](#)
- [Compiling SALT Configuration](#)
- [Configuring the UBBCONFIG File for BEA SALT](#)
- [Configuring BEA SALT In Tuxedo MP Mode](#)
- [Migrating from BEA SALT 1.1](#)

Using Tuxedo Service Metadata Repository for BEA SALT

BEA SALT leverages the [Tuxedo Service Metadata Repository](#) to define service contract information for both Tuxedo legacy services and SALT proxy services. Service contract information for all listed Tuxedo services is obtained by accessing the Tuxedo Service Metadata

Repository system service provided by the local Tuxedo domain. Typically, SALT calls the [TMMETADATA](#) system as follows:

- During `GWWS` server run-time.
It calls the Tuxedo Service Metadata Repository to retrieve necessary Tuxedo service definition at the appropriate time.
- When `tmwsdlgen` generates a WSDL file.
It calls the Tuxedo Service Metadata Repository to retrieve necessary Tuxedo service definitions and converts them to the WSDL description.

The following topics provide SALT-specific usage of Tuxedo Service Metadata Repository keywords and parameters:

- [Defining Service-Level Keywords for BEA SALT](#)
- [Defining Service Parameters for BEA SALT](#)

Defining Service-Level Keywords for BEA SALT

[Table 2-1](#) lists the Tuxedo Service Metadata Repository service-level keywords used and interpreted by SALT.

Note: Metadata Repository service-level keywords that are not listed have no relevance to BEA SALT and are ignored when SALT components load the Tuxedo Service Metadata Repository.

Table 2-1 BEA SALT Usage of Service-Level Keywords in Tuxedo Service Metadata Repository

Service-Level Keyword	Keyword Abbreviation	BEA SALT Usage
<code>service</code>	<code>sv</code>	The unique key value of the service. This value is referenced in the SALT configuration file.
<code>tuxservice</code>	<code>tsv</code>	The actual Tuxedo service name. BEA SALT invokes the Tuxedo service defined using this keyword. If no value is specified, then the value is the same as the value in the <code>service</code> keyword.

Table 2-1 BEA SALT Usage of Service-Level Keywords in Tuxedo Service Metadata Repository

Service-Level Keyword	Keyword Abbreviation	BEA SALT Usage
<code>servicetype</code>	<code>st</code>	<p>Determines the service message exchange pattern for the specified Tuxedo service. BEA SALT maps the service with the Web service message exchange pattern (MEP).</p> <p>The following values specify mapping rules between the Tuxedo service types and Web Service MEP:</p> <ul style="list-style-type: none"> • <code>service</code> corresponds to request-response MEP • <code>oneway</code> corresponds to oneway request MEP • <code>queue</code> corresponds to request-response MEP
<code>servicemode</code>	<code>sm</code>	<p>Determines the service mode (i.e., native Tuxedo service or an external Web Service converted to a Tuxedo service).</p> <p>The valid values are:</p> <ul style="list-style-type: none"> • <code>tuxedo</code> represents a native Tuxedo service • <code>webservice</code> represents a service definition converted from a <code>wsdl:operation</code> <p>Note: Do not use “<code>webservice</code>” to define a native Tuxedo service. This value is always used to define services converted from external Web services.</p>
<code>export</code>	<code>ex</code>	<p>BEA SALT ignores this value. If the service is specified in the SALT configuration <code><Servicelist></code>, it is exposed regardless of the value specified in <code>export</code>.</p>

Table 2-1 BEA SALT Usage of Service-Level Keywords in Tuxedo Service Metadata Repository

Service-Level Keyword	Keyword Abbreviation	BEA SALT Usage
<code>inbuf</code>	<code>bt</code>	<p>Specifies the input buffer (request buffer) type for the Tuxedo service. SALT provides default data representation and conversion between SOAP XML payload and the following Tuxedo buffer types:</p> <ul style="list-style-type: none">• <code>STRING</code> (case sensitive)• <code>CARRAY</code>• <code>XML</code>• <code>MBSTRING</code>• <code>VIEW</code>• <code>VIEW32</code>• <code>FML</code>• <code>FML32</code>• <code>X_C_TYPE</code>• <code>X_COMMON</code>• <code>X_OCTET</code>• <code>NULL</code> (The input buffer is empty) <p>Note: If <code>inbuf</code> specifies any other type other than the previous buffer types, the buffer is treated as a custom buffer type.</p>

Table 2-1 BEA SALT Usage of Service-Level Keywords in Tuxedo Service Metadata Repository

Service-Level Keyword	Keyword Abbreviation	BEA SALT Usage
outbuf	BT	<p>Specifies the output buffer type. SALT provides default data representation and conversion between the SOAP XML payload and the following Tuxedo buffer types:</p> <ul style="list-style-type: none"> • STRING (case sensitive) • CARRAY • XML • MBSTRING • VIEW • VIEW32 • FML • FML32 • X_C_TYPE • X_COMMON • X_OCTET • NULL (The output buffer is empty) <p>Note: If <code>outbuf</code> specifies any type other than the previous buffer types, the buffer is treated as a custom buffer type.</p>

Table 2-1 BEA SALT Usage of Service-Level Keywords in Tuxedo Service Metadata Repository

Service-Level Keyword	Keyword Abbreviation	BEA SALT Usage
<code>errbuf</code>		<p>Specifies the error buffer type returned with TPFail. SALT provides default data representation and conversion between SOAP XML payload and the following Tuxedo buffer types:</p> <ul style="list-style-type: none"> • <code>STRING</code> (case sensitive) • <code>CARRAY</code> • <code>XML</code> • <code>MBSTRING</code> • <code>VIEW</code> • <code>VIEW32</code> • <code>FML</code> • <code>FML32</code> • <code>X_C_TYPE</code> • <code>X_COMMON</code> • <code>X_OCTET</code> • <code>NULL</code> (The error buffer is empty) <p>Note: If <code>errbuf</code> specifies any type other than the above buffer types, the buffer is treated as a custom buffer type.</p>
<code>inview</code>	<code>vn</code>	<p>Specifies the view name used by the service for the following input buffer types:</p> <ul style="list-style-type: none"> • <code>VIEW</code> • <code>VIEW32</code> • <code>X_C_TYPE</code> • <code>X_COMMON</code> <p>BEA SALT requires that you specify the view name rather than accept the default <code>inview</code> setting.</p>

Table 2-1 BEA SALT Usage of Service-Level Keywords in Tuxedo Service Metadata Repository

Service-Level Keyword	Keyword Abbreviation	BEA SALT Usage
<code>outview</code>	VN	<p>Specifies the view name used by the service for the following output buffer types:</p> <ul style="list-style-type: none"> VIEW VIEW32 X_C_TYPE X_COMMON <p>BEA SALT requires that you specify the view name rather than accept the default <code>outview</code> setting.</p>
<code>errview</code>		<p>Specifies the view name used by the service for the following error buffer types:</p> <ul style="list-style-type: none"> VIEW VIEW32 X_C_TYPE X_COMMON <p>BEA SALT requires that you specify the view name rather than accept the default <code>errview</code> setting.</p>

Defining Service Parameters for BEA SALT

The Tuxedo Service Metadata Repository interprets parameters as sub-elements encapsulated in a Tuxedo service typed buffer. Each parameter can have its own data type, occurrences in the buffer, size restrictions, and other Tuxedo-specific restrictions. Please note:

- VIEW, VIEW32, X_C_TYPE, or X_COMMON typed buffers

Each parameter of the buffer should represent a VIEW/VIEW32 structure member.

- FML or FML32 typed buffers

Each parameter of the buffer should represent an FML/FML32 field element that may be present in the buffer.

- STRING, CARRAY, XML, MBSTRING, and X_OCTET typed buffers

Tuxedo treats these buffers holistically. At most, one parameter is permitted for the buffer to define restriction facets (such as buffer size threshold).

- Custom typed buffers

Parameters facilitate describing details about the buffer type.

- FML32 typed buffers that support embedded VIEW32 and FML32 buffers

Embedded parameters provide support.

[Table 2-2](#) lists the Tuxedo Service Metadata Repository parameter-level keywords used and interpreted by SALT.

Note: Metadata Repository parameter-level keywords that are not listed have no relevance to BEA SALT and are ignored when SALT components load the Tuxedo Service Metadata Repository.

Table 2-2 BEA SALT Usage of Parameter-Level Keyword in Tuxedo Service Metadata Repository

Parameter-level Keyword	Abbreviation	BEA SALT Usage
<code>param</code>	<code>pn</code>	<p>Specifies the parameter name.</p> <ul style="list-style-type: none"> • <code>VIEW</code>, <code>VIEW32</code>, <code>X_C_TYPE</code>, or <code>X_COMMON</code> Specifies the view structure member name in the <code>param</code> keyword. • <code>FML</code>, <code>FML32</code> Specifies the FML/FML32 field name in the <code>param</code> keyword. • <code>STRING</code>, <code>CARRAY</code>, <code>XML</code>, <code>MBSTRING</code>, or <code>X_OCTET</code> BEA SALT ignores the parameter definitions.
<code>type</code>	<code>pt</code>	<p>Specifies the data type of the parameter.</p> <p>Note: BEA SALT does not support <code>dec_t</code> and <code>ptr</code> data types.</p>

Table 2-2 BEA SALT Usage of Parameter-Level Keyword in Tuxedo Service Metadata Repository

Parameter-level Keyword	Abbreviation	BEA SALT Usage
subtype	pst	<p>Specifies the view structure name if the parameter type is VIEW32. For any other typed parameter, BEA SALT ignores this value.</p> <p>Note: The Tuxedo Service Metadata Repository allows this value to be empty if the parameter type is VIEW32. In this case the Tuxedo Service Metadata Repository uses a fixed value <viewname> as a default. This default value will not work for BEA SALT. You <i>must</i> specify the view name for BEA SALT to have a valid service definition.</p>
access	pa	<p>The general definition applies for this parameter. To support Tuxedo TPFail scenario, the <code>access</code> attribute value has been enhanced.</p> <p>Original values: <code>in</code>, <code>out</code>, <code>inout</code>, <code>noaccess</code>.</p> <p>New added values: <code>err</code>, <code>inerr</code>, <code>outerr</code>, <code>inouterr</code>.</p>
count	po	The general definition applies for this parameter. For BEA SALT, the value for the <code>count</code> parameter must be greater than or equal to <code>requiredcount</code> .
requiredcount	ro	The general definition applies for this parameter. The default is 1. For BEA SALT, the value for the <code>count</code> parameter must be greater than or equal to <code>requiredcount</code> .
size	pl	<p>This optional keyword restricts the maximum length of the parameter. It is only valid for parameter types: <code>STRING</code>, <code>CARRAY</code>, <code>XML</code>, and <code>MBSTRING</code>. If this keyword is not set, there is no maximum length restriction for this parameter.</p> <p>The value range is <code>[0, 2147483647]</code></p>

Configuring Native Tuxedo Services

This section describes the required and optional configuration tasks for exposing native Tuxedo services as Web Services:

- [Creating a Native WSDF](#)
- [Using WS-Policy Files](#)
- [Generating a WSDL File from a Native WSDF](#)

Creating a Native WSDF

To expose a set of Tuxedo services as Web services through one or more HTTP/S endpoints, a native `WSDF` must be defined.

Each native `WSDF` must be defined with a unique `WSDF` name. A `WSDF` can define one or more `<WSBinding>` elements for more Web service application details (such as SOAP protocol details, the Tuxedo service list to be exposed as web service operations, and so on).

Defining WSBinding Object

Each `WSBinding` object is defined using the `<WSBinding>` element. Each `WSBinding` object must be defined with a unique `WSBinding` id within the `WSDF`. The `WSBinding` id is a required indicator for the `SALTDEPLOY` file reference used by the `GWWS`.

Each `WSBinding` object can be associated with SOAP protocol details by using the `<SOAP>` sub-element. By default, SOAP 1.1, `document/literal` styled SOAP messages are applied to the `WSBinding` object; however, you can change or redefine the default.

[Listing 2-1](#) shows how SOAP protocol details are redefined using the `<SOAP>` sub-element.

Listing 2-1 Defining SOAP Protocol Details for a WSBinding

```
<Definition ...>
  <WSBinding id="simpapp_binding">
    <Servicegroup id="simpapp">
      <Service name="toupper" />
      <Service name="tolower" />
    </Servicegroup>
    <SOAP version="1.2" style="rpc" use="encoded">
      <AccessingPoints>
        ...
      </AccessingPoints>
    </SOAP>
  </WSBinding>
</Definition>
```

```

    </WSBinding>
</Definition>

```

Within the `<SOAP>` element, a set of access endpoints can be specified. The URL value of these access endpoints are used by corresponding `GWWS` servers to create the listen HTTP/S protocol port. It is recommended to specify one HTTP and HTTPS endpoint (at most) for each `GWWS` server for an *inbound* `WSBinding` object.

Each `WSBinding` object must be defined with a group of Tuxedo services using the `<Servicegroup>` sub-element. Each `<Service>` element under `<Servicegroup>` represents a Tuxedo service that can be accessed from a Web service client.

Defining Service Object

Each service object is defined using the `<Service>` element. Each service must be specified with the “name” attribute to indicate which Tuxedo service is exposed. Usually, the “name” value is used as the key value for obtaining Tuxedo service contract information from the Tuxedo Service Metadata Repository.

For inbound applications, you must list all Tuxedo services to be exposed for this `WSBinding` object. [Listing 2-2](#) shows how a group of services are defined for `WSBinding`.

Listing 2-2 Defining a Group of Services for a `WSBinding`

```

<Definition ...>
  <WSBinding id="simpapp_binding">
    <Servicegroup id="simpapp">
      <Service name="toupper" />
      <Service name="tolower" />
    </Servicegroup>
    ...
  </WSBinding>
</Definition>

```

Configuring Message Conversion Handler

You can create your own plug-in functions to customize SOAP XML payload and Tuxedo typed buffer conversion routine. For more information, see [Using BEA SALT Plug-ins](#) in *BEA SALT Programming Web Services* and [“Configuring Plug-in Libraries”](#) on page 2-21.

Once a plug-in is created and configured, it can be referenced using the `<service>` element to specify user-defined data mapping rules for that service. The `<Msghandler>` element can be defined at the message level (`<Input>` and `<Output>`) to specify which implementation of “P_CUSTOM_TYPE” category plug-in should be used to convert buffers for this message. The `<Msghandler>` element content is the “Custom Type Plug-in” name.

[Listing 2-3](#) shows a service that uses the “MBCONV” custom plug-in to convert input and “XMLCONV” custom plug-in to convert output.

Listing 2-3 Configuring Message Conversion Handler for a Service

```
<Definition ...>
  <WSBinding id="simpapp_binding">
    <Servicegroup id="simpapp">
      <Service name="toupper" >
        <Input>
          <Msghandler>MBCONV</Msghandler>
        </Input>
        <Output>
          <Msghandler>XMLCONV</Msghandler>
        </Output>
      </Service>
    </Servicegroup>
    ...
  </WSBinding>
</Definition>
```

Using WS-Policy Files

Advanced Web service features can be enabled by configuring WS-Policy files (for example, Reliable Messaging and Web Service Message-Level Security). You may need to create and save

different WS-Policy files to use these features. To use WS-Policy files, the `<Policy>` element should be defined to incorporate these separate WS-Policy files into a WSDL.

There are two different sub-element levels in the WSDL that reference WS-Policy files:

- `<Servicegroup>` level: the WS-Policy applies to all endpoints serving this `<Servicegroup>`.
- `<Service>` level: the WS-Policy applies to this particular service.

[Listing 2-4](#) shows a sample of using WS-Policy Files in the native WSDL file.

Listing 2-4 A Sample of Defining WS-Policy Files in the WSDL File

```
<Definition ...>
  <WSBinding id="simpapp_binding">
    <Servicegroup id="simpapp">
      <Policy location="endpoint_policy.xml" />
      <Policy location="all_in_msg_policy.xml" use="input" />
      <Service name="toupper">
        <Policy location="service_policy.xml" />
        <Policy location="input_message_policy.xml" use="input" />
      </Service>
      <Service name="tolower" />
    </Servicegroup>
    ....
  </WSBinding>
</Definition>
```

For more information, see [“Web Service Reliable Messaging”](#) and [“Using WS-SecurityPolicy Files”](#).

Generating a WSDL File from a Native WSDL

Once a Tuxedo native WSDL is created, the corresponding WSDL file can be generated using the SALT WSDL generation utility, `tmwsdlgen`. The following example generates a WSDL file named `appl.wsdl` from a given WSDL named `appl.wsdf`:

```
tmwsdlgen -c appl.wsdf -o appl.wsdl
```

Before executing `tmwsdlgen`, the `TUXCONFIG` environment variable must be set correctly and the relevant Tuxedo application using `TMMETADATA` must be booted.

You can optionally specify the output WSDL file name using the `-o` option. Otherwise, `tmwsdlgen` creates a default WSDL file named `"tuxedo.wsdl"`.

If the native WSDF file contains Tuxedo services that use `CARRAY` buffers, you can specify `tmwsdlgen` options to generate different styled WSDL files for `CARRAY` buffer mapping. By default, `CARRAY` buffers are mapped as `xsd:base64Binary` XML data types in the SOAP message. You can alternatively map `CARRAY` buffers as MIME attachments in the WSDL file. For more information, see [Data Type Mapping and Conversions](#) in the *BEA SALT Programming Web Services* and [tmwsdlgen](#) in the *BEA SALT Reference Guide*.

Configuring External Web Services

To invoke an external Web Service, the following configuration tasks need to be performed:

- [Converting a WSDL file into Tuxedo Definitions](#)
- [Resolving Naming Conflict For the Generated Tuxedo Service Contract](#)
- [Post Conversion Tasks](#)

Converting a WSDL file into Tuxedo Definitions

The WSDL file is converted using Extensible Stylesheet Language Transformations (XSLT) 1.0. BEA SALT uses Apache Xalan Java 2.7.0 as the default XSLT toolkit. The WSDL converter is composed of two parts:

- The `xsl` files, which process the WSDL file.
- The Tuxedo utility, `wsdlcvt`, invokes the Xalan toolkit. This wrapper script provides a user friendly WSDL Converter interface.

The following Tuxedo definitions are generated using the SALT WSDL Converter:

- Tuxedo Service Metadata Input File
- FML32 Field Table File
- Non-native WSDF File
- XML Schema Files excerpted from the given WSDL document

Resolving Naming Conflict For the Generated Tuxedo Service Contract

When converting a WSDL file into an MIF (Tuxedo Service Metadata Input File), FML32 Mapping File and WSDF, unexpected naming conflicts may be found in the following items due to truncation or lost context information:

- The MIF “`tuxservice`” keyword

The keyword `tuxservice` in the Tuxedo Service definition is the Tuxedo service that is used for the outbound WSDL ~~operation~~ file. The name of the ~~operation~~ file is truncated to 15 characters during the conversion. To resolve the naming conflict, you can assign any valid Tuxedo service name to `tuxservice`.

- The MIF “`param`” keyword and the FML32 Mapping File field “`name`.”

The FML32 Mapping File field `name` should be unique among all name items in all FML32 Mapping Files. The MIF `param` and FML32 Mapping File field `name` should be coherent. The context of `param` and `name` is not included in the MIF or the FML32 Mapping File. This causes naming conflicts in these files.

The FML32 Mapping File mapping table is sorted by name so that duplicated names can be found easily. To resolve naming conflicts, you can assign any valid Tuxedo FML32 field name to `name`, and then modify the corresponding `param` accordingly. Comments in the FML32 field and MIF `param` are helpful in locating the corresponding `name` and `param`.

Note: Before using the MIF, FML32 Mapping File and WSDF, you should eliminate all the naming conflicts first.

You can also modify the following items in the MIF and FML32 Mapping File:

- The MIF and FML32 Mapping File “`type`” keyword

The `xsd:string` XML data type can be mapped to Tuxedo STRING or MBSTRING buffers. The WSDL converter transforms all `xsd:string` occurrences into to a single instance specified by the `-m` option. If you alter an individual field the type, you must maintain consistency in the MIF and FML32 Mapping File.

- Comment in MIF and FML32 Mapping File:

It is a memo to the field, user are free to put anything they need in this field.

WARNING: All items not mentioned in this section should not be modified.

Post Conversion Tasks

The following post conversion tasks need to be performed for configuring outbound Web service applications:

- Load the Tuxedo Service Metadata Input file into the Tuxedo Service Metadata Repository
- Update environment variable *FLDTBLDIR32* and *FIELDTBLS32* to add the generated FML32 field table file.
- Place all excerpted XML Schema files into one directory, and set environment variable *XSDDIR* and *XSDFILES* accordingly. Environment variable *XSDDIR* and *XSDFILES* are introduced in SALT 2.0 release. They are used by the *GWWS* server to load all external XML Schema files at run time. Multiple XML Schema file names should be delimited with comma ‘,’. For instance, if you placed XML Schema files: *a.xsd*, *b.xsd* and *c.xsd* in directory */home/user/myxsd*, you must set environment variable *XSDDIR* and *XSDFILES* as follows before booting the *GWWS* server:

```
XSDDIR=/home/user/myxsd
XSDFILES=a.xsd,b.xsd,c.xsd
```

Creating the SALT Deployment File

The SALT Deployment file (SALTDEPLOY) defines a SALT Web service application. The SALTDEPLOY file is the major input for Web service application in the binary SALTCONFIG file.

To create a SALTDEPLOY file, do the following steps:

1. [Importing the WSDF Files](#)
2. [Configuring the GWWS Servers](#)
3. [Configuring System Level Resources](#)

For more information, see “[SALT Deployment File Reference](#)” in the BEA SALT Reference Guide.

Importing the WSDF Files

You should import all of your required WSDF files to the SALT deployment file. Each imported WSDF file must have a unique WSDF name which is used by the *GWWS* servers to make

deployment associations. Each imported WSDL file must be ~~locally~~ accessible through the location specified in the SALTDEPLOY file.

[Listing 2-5](#) shows how to import WSDL files in the SALTDEPLOY file.

Listing 2-5 Importing WSDL Files in the SALTDEPLOY File

```
<Deployment ..>
  <WSDL>
    <Import location="/home/user/simpapp_wsdl.xml" />
    <Import location="/home/user/rmapp_wsdl.xml" />
    <Import location="/home/user/google_search.wsdl" />
  </WSDL>
  ...
</Deployment>
```

Configuring the GWWS Servers

Each GWWS server can be deployed with a group of inbound WSBinding objects and a group of outbound WSBinding objects defined in the imported WSDL files. For inbound WSBinding objects, each GWWS server must specify at least one access endpoint as an inbound endpoint from the endpoint list in the WSBinding object. For outbound WSBinding objects, each GWWS server can specify zero or more access endpoints as outbound endpoints from the endpoint list in the WSBinding object.

[Listing 2-6](#) shows how to configure GWWS servers with both inbound and outbound endpoints.

Listing 2-6 GWWS Server Defined In the SALTDEPLOY File

```
<Deployment ..>
  ...
  <WSGateway>
    <GWInstance id="GWWS1">
      <Inbound>
        <Binding ref="appl:appl_binding">
          <Endpoint use="simpapp_GWWS1_HTTPPort" />
```

```

        <Endpoint use="simpapp_GWWS1_HTTPSPort" />
    </Binding>
</Inbound>
<Outbound>
    <Binding ref="app2:app2_binding">
        <Endpoint use=" simpapp_GWWS1_HTTPPort" />
        <Endpoint use=" simpapp_GWWS1_HTTPSPort" />
    </Binding>
    <Binding ref="app3:app3_binding" />
</Outbound>
</GWInstance>
</WSGateway>
...
</ Deployment>

```

Configuring GWWS Instance Level Properties

The GWWS server can be configured with properties that switch feature on/off or set argument to tune the server’s performance. In this chapter, we show the principle to configure the property and discuss feature swithing properties. Performance tuning properties are addressed in [“Tuning the GWWS Server” on page 3-3](#).

Properties are configured in the `<GWInstance>` child element `<Properties>`. Each individual property is defined by using the `<Property>` element which contains a “name” attribute and a “value” attribute). Different “name” attributes represent different property elements that contain a value. [Listing 2-7](#) shows an example of how GWWS properties are configured.

Listing 2-7 Configuring GWWS Server Properties

```

<Deployment ..>
...
<WSGateway>
    <GWInstance id="GWWS1">
        .....
        <Properties>
            <Property name="xxxx" value="aaaa"/>
            <Property name="yyyy" value="bbbb"/>

```

```

        <Property name="zzzz" value="cccc"/>
    </Properties>
</GWInstance>
</WSGateway>
...
</ Deployment>

```

Configuring Multiple Encoding Support

Although the GWWS server uses UTF-8 as internal character encoding, it does have the ability to handle messages with different character encoding. It . Messages with different character encoding are converted into UTF-8 and then processed. However, encoding conversion will impact server performance. By default, encoding conversion is turned off and messages that are not UTF-8 encoded are rejected.

This feature can be switched on per-GWWS server using the property name “enableMultiEncoding” with a value set to “true”:

Listing 2-8 Configuring GWWS Server Properties

```

<Deployment ..>
...
<WSGateway>
    <GWInstance id="GWWS1">
        .....
        <Properties>
            <Property name="enableMultiEncoding" value="true"/>
        </Properties>
    </GWInstance>
</WSGateway>
...
</ Deployment>

```

Note: The supported character encodings includes:

ASCII, BIG5, CP1250, CP1251, CP1252, CP1253, CP1254, CP1255, CP1256, CP1257, CP1258, CP850, CP862, CP866, CP874, EUC-CN, EUC-JP, EUC-KR, GB18030, GB2312, GBK, ISO-2022-JP, ISO-8859-1, ISO-8859-13, ISO-8859-15, ISO-8859-2, ISO-8859-3, ISO-8859-4, ISO-8859-5, ISO-8859-6, ISO-8859-7, ISO-8859-8, ISO-8859-9, JOHAB, KOI8-R, SHIFT_JIS, TIS-620, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, UTF-32LE, UTF-7, UTF-8.

Configuring System Level Resources

BEA SALT defines a set of global resources shared by all `GWWS` servers in the `SALTDEPLOY` file. The following system level resources can be configured in the `SALTDEPLOY` file:

- Certificates
- Plug-in load libraries

Configuring Certificates

Certificate information must be configured in order for the `GWWS` server to create an SSL listen endpoint, or to use X.509 certificates for authentication and/or message signature. All `GWWS` servers defined in the same deployment file shares the same certificate settings, including the private key file, trusted certificate directory, and so on.

The private key file is configured using the `<Certificate>/<PrivateKey>` sub-element. The private key file must be in PEM file format and stored locally.

SSL clients can optionally be verified if the `<Certificate>/<VerifyClient>` sub-element is set to `true`. By default, the `GWWS` server does not verify SSL clients.

If SSL clients are to be verified, and/or the X.509 certificate authentication feature is enabled, a set of trusted certificates must be stored locally and located by the `GWWS` server. There are two ways to define `GWWS` server trusted certificates:

1. Include all certificates in one PEM format file and define the file path using the `<<Certificate>/<TrustedCert>` subelement.
2. Saving separate certificate PEM format files in one directory and define the directory path using the `<<Certificate>/<CertPath>` sub-element.

[Listing 2-9](#) shows a `SALTDEPLOY` file segment configuring `GWWS` server certificates.

Listing 2-9 Configuring Certificates In the SALTDEPLOY File

```

<Deployment ..>
...
<System>
  <Certificates>
    <PrivateKey>/home/user/gwvs_cert.pem</PrivateKey>
    <VerifyClient>true</VerifyClient>
    <CertPath>/home/user/trusted_cert</CertPath>
  </Certificates>
</System>
</Deployment>

```

Configuring Plug-in Libraries

A plug-in is a set of functions that are called when the GWVS server is running. BEA SALT provides a plug-in framework as a common interface for defining and implementing a plug-in. Plug-in implementation is carried out through a dynamic library that contains the actual function code. The implementation library can be loaded dynamically during GWVS server start up. The functions are registered as the implementation of the plug-in interface.

In order for the GWVS server to load the library, the library must be specified using the <Plugin>/<Interface> element in the SALTDEPLOY file.

[Listing 2-10](#) shows a SALTDEPLOY file segment configuring multiple customized plug-in libraries to be loaded by the GWVS servers.

Listing 2-10 Configuring Plug-in Libraries In the SALTDEPLOY File

```

<Deployment ..>
...
<System>
  <Plugin>
    <Interface lib="plugin_1.so" />
    <Interface lib="plugin_2.so" />
  </Plugin>

```

```
</System>  
</Deployment
```

Note: If the plug-in library is developed using the SALT 2.0 plug-in interface, the “id” and “name” attributes for the interface do not need to be specified. These values can be obtained through plugin interfaces.

For more information, see [Using Plug-ins with BEA SALT](#) in BEA SALT Programming with Web Services.

Configuring Advanced Web Service Messaging Features

BEA SALT currently supports the following advanced features:

- [Web Service Addressing](#)

Supports both inbound and outbound Web services.

- [Web Service Reliable Messaging](#)

Supports inbound Web services only.

Web Service Addressing

BEA SALT supports Web service addressing for both inbound and outbound services. The Web service addressing (WS-Addressing) messages used by the `GWWS` server must comply with the [Web Service Addressing standard \(W3C Member Submission 10 August 2004\)](#).

Inbound services do not require specific Web service addressing configuration. The `GWWS` server accepts and responds accordingly to both WS-Addressing request messages and non WS-Addressing request messages.

Outbound services require Web service addressing configuration as described in the following sections:

- [Configuring the Addressing Endpoint for Outbound Services](#)
- [Disabling WS-Addressing](#)

Configuring the Addressing Endpoint for Outbound Services

For outbound services, Web service addressing is configured at the Web service binding level. In the SALTDEPLOY file, each GWWS server can specify a WS-Addressing endpoint by using the `<WSAddressing>` element for any referenced outbound WSBinding object to enable WS-Addressing.

Once the WS-Addressing endpoint is configured, the GWWS server creates a listen endpoint at start up. All services defined in the outbound WSBinding are invoked with WS-Addressing messages.

[Listing 2-11](#) shows a SALTDEPLOY file segment enabling WS-Addressing for a referenced outbound Web service binding.

Listing 2-11 WS-Addressing Endpoint Defined for Outbound Web Service Binding

```
<Deployment ..>
  ...
  <WSGateway>
    <GWInstance id="GWWS1">
      ...
      <Outbound>
        <Binding ref="appl:appl_binding">
          <WSAddressing>
            <Endpoint address="http(s)://<host>:<port>/<context_path>">
          </WSAddressing>
          <Endpoint use=" simpapp_GWWS1_HTTPPort" />
          <Endpoint use=" simpapp_GWWS1_HTTPSPort" />
        </Binding>
        <Binding ref="app2:app2_binding">
          <WSAddressing>
            <Endpoint address="http(s)://<host>:<port>/<context_path>">
          </WSAddressing>
          <Endpoint use=" simpapp_GWWS1_HTTPPort" />
          <Endpoint use=" simpapp_GWWS1_HTTPSPort" />
        </Binding>
      </Outbound>
      ...
    </GWInstance>
  </WSGateway>
```

```
...
</ Deployment>
```

Notes: For each `GWWS` server, each outbound Web Service binding can be associated with a particular WS-Addressing endpoint address. These endpoints can be defined with the same hostname and port number, but the context path portion of the endpoint addresses must be different.

If the external Web service binding does not support WS-Addressing messages, configuring Addressing endpoints may result in run time failure.

Disabling WS-Addressing

Whether you create a WS-Addressing endpoint or not, you may still want to explicitly disable the Addressing capability for particular outbound services. To disable the Addressing capability, you should use the property name “`disableWSAddressing`” with a value set to “`true`” in the `<Service>` definition of the WSDL file. This property has no impact to any inbound services.

[Listing 2-12](#) shows WSDL file segment disabling Addressing capability.

Listing 2-12 Disabling Service Level WS-Addressing

```
<Definition ...>
  <WSBinding id="simpapp_binding">
    <Servicegroup id="simpapp">
      <Service name="toupper">
        <Property name="disableWSAddressing" value="true" />
      </Service>
      <Service name="tolower" />
    </Servicegroup>
    ....
  </WSBinding>
</Definition>
```

Web Service Reliable Messaging

BEA SALT currently supports Reliable Messaging for inbound services only. To enable Reliable Messaging functionality, you must create a Web Service Reliable Messaging policy file and include the policy file in the WSDL file. A WSDL file containing a Reliable Messaging policy definition should be used by the `GWWS` server for inbound direction only. The policy file must comply with the [WS-ReliableMessaging Policy Assertion Specification \(February 2005\)](#).

Creating the Reliable Messaging Policy File

A Reliable Messaging Policy file is a general WS-Policy file containing WS-ReliableMessaging Assertions. The WS-ReliableMessaging Assertion is a XML segment that describes features such as the version of the supported WS-ReliableMessage specification, the source endpoint's retransmission interval, the destination endpoint's acknowledge interval, and so on.

For more information see, the [BEA SALT WS-ReliableMessaging Policy Assertion Reference](#) in the *BEA SALT Reference Guide*.

[Listing 2-13](#) shows a Reliable Messaging policy file example.

Listing 2-13 Reliable Messaging Policy File Example

```
<?xml version="1.0"?>
<wsp:Policy wsp:Name="ReliableSomeServicePolicy"
  xmlns:wsm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:beapolicy="http://www.bea.com/wsm/policy">
  <wsm:RMAssertion>
    <wsm:InactivityTimeout Milliseconds="600000" />
    <wsm:AcknowledgementInterval Milliseconds="2000" />
    <wsm:BaseRetransmissionInterval Milliseconds="500"/>
    <wsm:ExponentialBackoff />
    <beapolicy:Expires Expires="P1D" />
    <beapolicy:QOS QOS="ExactlyOnce InOrder" />
  </wsm:RMAssertion>
</wsp:Policy>
```

All RM assertions are optional. If not specified, the default values are used. The following definitions describe the RM assertion options.

RM Assertion Option	Description
<code><wsrm:InactivityTimeout></code>	<p>Specifies the number of milliseconds, specified with the <code>Milliseconds</code> attribute, which defines an inactivity interval. After time has elapsed, if the destination endpoint has not received a message from the source endpoint, the destination endpoint may terminate current sequence due to inactivity. The source endpoint can also use this parameter.</p> <p>Sequences never time out by default.</p>
<code><wsrm:AcknowledgementInterval></code>	<p>Specifies the maximum interval, in milliseconds, in which the destination endpoint must transmit a stand-alone acknowledgement.</p> <p>There is no time limit by default.</p>
<code><wsrm:BaseRetransmissionInterval></code>	<p>Specifies the interval, in milliseconds, that the source endpoint waits after transmitting a message and before it retransmits the message if it receives no acknowledgment for that message. This value will apply to the GWWS server when it sends a response in an outbound sequence.</p> <p>The default value is 20000 milliseconds.</p>
<code><wsrm:ExponentialBackoff></code>	<p>Specifies that the retransmission interval is adjusted using the exponential backoff algorithm. This value applies to the GWWS server when it sends a response in an outbound sequence.</p>

RM Assertion Option	Description
<code><beapolicy:Expires></code>	<p>Specifies the amount of time after which the reliable Web service expires and does not accept any new sequence messages.</p> <p>This element has a single attribute, Expires, whose data type is an XML Schema duration type. For example, if you want to set the expiration time to one day, use the following:</p> <pre>< beapolicy:Expires Expires="P1D" /></pre> <p>The default value is never expire.</p>
<code><beapolicy:QOS></code>	<p>Specifies the delivery assurance. SALT supports the following assurances:</p> <ul style="list-style-type: none"> • AtMostOnce - Messages are delivered at most once, without duplication. There is possibility that some messages may not be delivered. • AtLeastOnce - Every message is delivered at least once. There is possibility that some messages are delivered more than once. • ExactlyOnce - Each message is delivered exactly once, without duplication. • InOrder - Messages are delivered in the order that they were sent. This delivery assurance can be combined with one of the preceding three assurances. <p>The default value is “ExactlyOnce InOrder”.</p>

Specifying the Reliable Messaging Policy File in the WSDL File

You must reference the WS-ReliableMessaging policy file at the `<Servicegroup>` level in the native WSDL file. The following segment of the WSDL file shows how to reference the WS-ReliableMessaging policy file.

Listing 2-14 Reference the WS-ReliableMessaging Policy At the Endpoint Level

```
<Definition ...>
  <WSBinding ...>
    <Servicegroup ...>
      <Policy refid="RM">
```

```
<Service ... />
<Service ... />
...
</Servicegroup ...>
</WSBinding>
<WSPolicy id="RM" location="RMpolicy.xml" />
</Definition>
```

Note: Reliable Messaging in BEA SALT does not support process/system failure scenarios, which means SALT does not store the message in a persistent storage area. BEA SALT works in a *direct mode* with the SOAP client. Usually, system failure recovery requires business logic synchronization between the client and server.

Configuring Security Features

BEA SALT provides security support at both transport level and SOAP message level. The following topics explain how to configure security features for each level:

- [Configuring Transport Level Security](#)
- [Configuring Message Level Web Service Security](#)

Configuring Transport Level Security

BEA SALT provides point-to-point security using SSL link-level security and supports HTTP basic authentication mechanism for both inbound and outbound service authentication.

Setting Up SSL Link-Level Security

To set up link-level security using SSL at inbound endpoints, you can simply specify the endpoint address with prefix “https://”. The GWWS server who uses this inbound endpoint creates SSL listen port and make SSL secured connections with Web Service Clients. SSL features need to specify certificates settings. For more information about certificate settings, see “[Configuring Certificates](#)”.

Configuring Inbound HTTP Basic Authentication

BEA SALT depends on the Tuxedo security framework for Web Service client authentication. There is no special configuration at BEA SALT side to enable inbound HTTP Basic

Authentication. If Tuxedo system requires user credential, HTTP Basic Authentication is simply an alternative for Web Service client program to carry the user credential.

The `GWWS` gateway supports Tuxedo domain security configuration for the following two authentication patterns:

- Application password (`APP_PW`)
- User-level authentication (`USER_AUTH`)

The `GWWS` server passes the following string from the HTTP header of the client SOAP request for Tuxedo authentication.

```
Authorization: Basic <base64Binary of username:password>
```

The following is an example of a string from the HTTP header:

```
Authorization: Basic QWxhZGRpbjpwVGvUHNlc2FtZQ==
```

In this example, the client sends the Tuxedo username “Aladdin” and the password “open sesame”, and uses this paired value for Tuxedo authentication.

- Using Application Password (`APP_PW`)

If Tuxedo uses `APP_PW`, then the HTTP username value is ignored and the `GWWS` server only uses the password string as the Tuxedo application password to check the authentication.

- Using User-level Authentication (`USER_AUTH`)

If Tuxedo uses `USER_AUTH`, then both the HTTP username and password value is used. In this case, the `GWWS` server does not check the Tuxedo application password.

Note: `ACL` and `MANDATORY_ACL` are not supported for Web service clients, which means the Tuxedo system ignores any ACL-related configuration specifications. BEA SALT does not make group information available for Web service clients.

Configuring Outbound HTTP Basic Authentication

BEA SALT supports customers to develop authentication plug-in to prepare the user profile for the outbound HTTP requests. Outbound HTTP Basic Authentication is configured at Endpoint level. If an outbound Endpoint requires user profile in the HTTP message, you must specify the HTTP Realm for the HTTP endpoint in the WSDF file. The `GWWS` server invokes authentication plug-in library to prepare the user profile information and send them using HTTP Basic Authentication mechanism in the request message.

[Listing 2-15](#) shows how to enable HTTP Basic Authentication for the outbound endpoints.

Listing 2-15 Enabling HTTP Basic Authentication For the Outbound Endpoint

```
<Definition ...>
  <WSBinding id="simpapp_binding">
    <SOAP>
      <AccessingPoints>
        <Endpoint id="..." address="...">
          <Realm>SIMP_REALM</Realm>
        </Endpoint>
      </AccessingPoints>
    </SOAP>
    <Servicegroup id="simpapp">
      ....
    </Servicegroup>
    ....
  </WSBinding>
  .....
</Definition>
```

Once a service request is sending to an outbound endpoint specified with “`Realm`” value, the GWWS server passes the Tuxedo client `uid` and `gid` to the authentication plug-in function, so that the plug-in can determine HTTP Basic Authentication `username/password` according to the Tuxedo client information. To obtain Tuxedo client `uid` / `gid` for HTTP basic authentication `username/password` mapping, Tuxedo security level may also need to be configured in the `UBBCONFIG` file. For more information, see [“Configuring Tuxedo Security Level for Outbound HTTP Basic Authentication”](#).

For more information about how to develop an outbound authentication plug-in, see [Programming Outbound Authentication Plug-ins](#) in the *BEA SALT Programming Web Services*.

Configuring Message Level Web Service Security

BEA SALT supports Web Service Security 1.0 and 1.1 specification for message level security. You can use message-level security in BEA SALT to assure:

- Authentication, by requiring username or X.509 tokens

Main Use Cases of Web Service Security

BEA SALT implementation of the Web Service Security: SOAP Message Security specification supports the following use cases:

- Include a token (username, or X.509) in the SOAP message for authentication.

Using WS-SecurityPolicy Files

BEA SALT includes a number of WS-Security Policy 1.0 and 1.2 files you can use for message level security use cases.

The WS-Policy files can be found at `$TUXDIR/udataobj/salt/policy` once you have successfully installed BEA SALT.

The following table lists the WS-Security Policy files provided by BEA SALT.

Table 2-3 WS-Security Policy Files Provided By BEA SALT

File Name	Purpose
wssp1.0-username-auth.xml	WS-Security Policy 1.0. Plain Text Username Token for Service Authentication
wssp1.0-x509v3-auth.xml	WS-Security Policy 1.0. X.509 V3 Certificate Token for Service Authentication
wssp1.0-signbody.xml	WS-Security Policy 1.0. Signature on SOAP:Body for verification of X.509 Certificate Token
wssp1.2-Wss1.0-Username Token-plain-auth.xml	WS-Security Policy 1.2. Plain Text Username Token for Service Authentication
wssp1.2-Wss1.1-X509V3-auth.xml	WS-Security Policy 1.2. X.509 V3 Certificate Token for Service Authentication
wssp1.2-signbody.xml	WS-Security Policy 1.2. Signature on SOAP:Body for verification of X.509 Certificate Token

The above policy files except WS-Security Policy 1.2 UserToken file can be referenced at <Servicegroup> or <Service> level in the native WSDL file. The WSSP 1.2 UserToken file can only be referenced at <Servicegroup> level. The sample “wsseapp” shows how to clip the WSSP 1.2 UserToken file to be used in <Service> level.

Listing 2-16 shows a combination of policy assignment making that the service “TOUPPER” will require client send a UsernameToken in PlainText format and an X509v3Token in request and also require the SOAP:Body part of message is signed with the X509 token.

Listing 2-16 WS-Security Policy Usage

```
<Definition ...>
  <WSBinding id="simpapp_binding">

    <Servicegroup id="simpapp">
      <Policy location="salt:wsspl.2-Wss1.1-X509V3-auth.xml"/>
      <Service name="TOUPPER" >
        <Policy location="D:/wsseapp/wsspl.2-UsernameToken-Plain.xml"/>
        <Policy location="salt:wsspl.2-signbody.xml" use="input"/>
      </Service>
    </Servicegroup>
    ....
  </WSBinding>
  .....
</Definition>
```

Policy is referred with “location” attribute of the <Policy> element. A prefix “salt:” means the policy is predefined one. User-defined policy file can be used by directly specifying the file name.

Notes: If a policy is referred at <Servicegroup> level, it will apply to all services in this service group.

The “signbody” policy must be used with the attribute “use” set as “input”, which specifies the policy be used only for input message. This is necessary because we do not sign the SOAP:Body of output message.

Compiling SALT Configuration

Compiling a SALT configuration file means generating a binary version of the file (SALTCONFIG) from the XML version SALTDEPLOY file. To compile a configuration file, run the `wsloadcf` command. `wsloadcf` parses a deployment file and loads the binary file.

`wsloadcf` reads a deployment file and all imported WSDL files and WS-Policy files referenced in the deployment file, checks the syntax according to the XML schema of each file format, and optionally loads a binary configuration file called `SALTCONFIG`. The `SALTCONFIG` and (optionally) `SALTOFFSET` environment variables point to the `SALTCONFIG` file and (optional) offset where the information should be stored.

`wsloadcf` validates the given SALT configuration files according to the predefined XML Schema files. XML Schema files needed by BEA SALT can be found at directory:
`$TUXDIR/udataobj/salt.`

`wsloadcf` can execute for validating purpose only without generating the binary version `SALTCONFIG` once option “-n” is specified.

For more information about `wsloadcf`, see [wsloadcf](#) reference in the *BEA SALT Reference Guide*.

Configuring the UBBCONFIG File for BEA SALT

After configuring and compiling SALT configuration, Tuxedo `UBBCONFIG` file needs to be updated to apply SALT components in the Tuxedo application. [Table 2-4](#) lists the `UBBCONFIG` file configuration tasks for BEA SALT.

Table 2-4 UBBCONFIG File Configuration Tasks for BEA SALT

Configuration Tasks	Required	Optional
Configuring the TMMETADATA Server in the *SERVERS Section	X	
Configuring the GWWS Servers in the *SERVERS Section	X	
Updating System Limitations in the UBBCONFIG File	X	
Configuring Certificate Password Phrase For the GWWS Servers		X
Configuring Tuxedo Authentication for Web Service Clients		X
Configuring Tuxedo Security Level for Outbound HTTP Basic Authentication		X

Configuring the TMMETADATA Server in the *SERVERS Section

BEA SALT requires at least one TMMETADATA server defined in the UBBCONFIG file. Multiple TMMETADATA servers are also allowed to increase the throughput of accessing the Tuxedo service definitions.

[Listing 2-17](#) lists a segment of the UBBCONFIG file that shows how to define TMMETADATA servers in a Tuxedo application.

Listing 2-17 TMMETADATA Servers Defined In the UBBCONFIG File *SERVERS Section

```
.....
*SERVERS
TMMETADATA SRVGRP=GROUP1 SRVID=1
           CLOPT="-A -- -f domain_repository_file -r"
TMMETADATA SRVGRP=GROUP1 SRVID=2
           CLOPT="-A -- -f domain_repository_file"
.....
```

Note: Maintaining only one Service Metadata Repository file for the whole Tuxedo domain is highly recommended. To ensure this, multiple TMMETADATA servers running in the Tuxedo domain must point to the same repository file.

For more information, see [“Managing The Tuxedo Service Metadata Repository”](#) in the *Tuxedo 9.1 documentation*.

Configuring the GWWS Servers in the *SERVERS Section

To boot GWWS instances defined in the SALTDEPLOY file, the GWWS servers must be defined in the *SERVERS section of the UBBCONFIG file. You can define one or more GWWS server instances concurrently in the UBBCONFIG file. Each GWWS server must be assigned with a unique instance id with the option “-i” within the Tuxedo domain. The instance id must can be found from the XML version SALTDEPLOY file and the generated binary version SALTCONFIG file.

[Listing 2-18](#) lists a segment of the UBBCONFIG file that shows how to define GWWS servers in a Tuxedo application.

Listing 2-18 GWWS Servers Defined In the UBBCONFIG File *SERVERS Section

```

.....
*SERVERS
GWWS SRVGRP=GROUP1 SRVID=10
    CLOPT="-A -- -i GW1"
GWWS SRVGRP=GROUP1 SRVID=11
    CLOPT="-A -- -i GW2"
GWWS SRVGRP=GROUP2 SRVID=20
    CLOPT="-A -- -c saltconf_2.xml -i GW3"
.....

```

For more information, see “[GWWS](#)” in the BEA SALT Reference Guide.

Note: Be sure that the `TMMETADATA` system server is set up in the `UBBCONFIG` file to start before the `GWWS` server boots. Because the `GWWS` server calls services provided by `TMMETADATA`, it must boot after `TMMETADATA`.

To ensure `TMMETADATA` is started prior to being called by the `GWWS` server, put `TMMETADATA` before `GWWS` in the `UBBCONFIG` file or use `SEQUENCE` parameters in `*SERVERS` definition in the `UBBCONFIG` file.

Note: SALT configuration information is pre-compiled with `wsloadcf` to generated a binary version `SALTCONFIG` file. `GWWS` server reads `SALTCONFIG` file at start up. Environment variable `SALTCONFIG` must be set correctly with the binary version `SALTCONFIG` file entity before booting `GWWS` servers.

Note: Option “`-c`” is deprecated in the current version BEA SALT. In SALT 1.1 release, option “`-c`” is used to specify SALT 1.1 configuration file for the `GWWS` server. In SALT 2.0, `GWWS` server reads `SALTCONFIG` file at start up. `GWWS` server specified with this option can be booted with a warning message to indicate this deprecation. The specified file can be arbitrary and is not read by the `GWWS` server.

Updating System Limitations in the UBBCONFIG File

When configuring the Tuxedo domain with SALT `GWWS` servers, you need to plan and update Tuxedo system limitations defined in the `UBBCONFIG` file according to your SALT application requirements.

Tip: Defining enough `MAXSERVERS` number in the `*RESOURCES` section

BEA SALT requires the following system servers to be started in a Tuxedo domain: `TMMETADATA` and `GWWS`. The number of `TMMETADATA` and `GWWS` server must be accounted for in the `MAXSERVERS` value.

Tip: Defining enough `MAXSERVICES` number in the `*RESOURCES` section

When the `GWWS` server working in the outbound direction, external `wsdl:operations` are mapped with Tuxedo services and advertised via the `GWWS` Servers. The number of the advertised services by all `GWWS` servers must be accounted for in the `MAXSERVICES` value.

Tip: Defining enough `MAXACCESSERS` number in the `*RESOURCES` section

`MAXACCESSERS` value is used to specify the default maximum number of clients and servers that can be simultaneously connected to the Tuxedo bulletin board on any particular machine in this application. The number of `TMMETADATA` and `GWWS` server, maximum concurrent Web Service client requests must be accounted for in the `MAXACCESSERS` value.

Tip: Defining enough `MAXWSCLIENTS` number in the `*MACHINES` section

When the `GWWS` server working in the inbound direction, each Web Service client is deemed a workstation client in Tuxedo system; therefore, `MAXWSCLIENTS` must be configured with a valid number in `UBBCONFIG` for the machine where the `GWWS` server is deployed. The number shares.

Configuring Certificate Password Phrase For the GWWS Servers

Configuring security password phrase is required when setting up certificates for BEA SALT. Certificates setting is desired when the `GWWS` servers enabling SSL link-level encryption and/or Web Service Security X.509 Token and signature features. The certificate private key file needs to be created and encrypted with a password phrase.

When the `GWWS` servers are specified with certificate related features, they are required to read the private key file and decrypt them using the password phrase. To configure password phrase for each `GWWS` server, keyword `SEC_PRINCIPAL_NAME` and `SEC_PRINCIPAL_PASSVAR` must be

specified under each desired GWWS server entry in the `*SERVERS` section. During compiling the UBBCONFIG file with `tmloadcf`, the administrator must type the password phrase, which can be used to decrypt the private key file correctly.

Note: Only one private key file can be specified in the SALT deployment file. All the GWWS servers defined in the SALT deployment file must be provided the same password phrase for the private key file decryption.

[Listing 2-19](#) lists a segment of the UBBCONFIG file that shows how to define security password phrase for the GWWS servers.

Listing 2-19 Security Password Phrase Defined in the UBBCONFIG File For the GWWS Servers

```
.....
*SERVERS
GWWS SRVGRP=GROUP1 SRVID=10
    SEC_PRINCIPAL_NAME="gwws_certkey"
    SEC_PRINCIPAL_VAR="gwws_certkey"
    CLOPT="-A -- -i GW1"
GWWS SRVGRP=GROUP1 SRVID=11
    SEC_PRINCIPAL_NAME="gwws_certkey"
    SEC_PRINCIPAL_PASSVAR="gwws_certkey"
    CLOPT="-A -- -i GW2"
.....
```

For more information, see “[UBBCONFIG \(5\)](#)” in the *Tuxedo 9.1 documentation*.

Configuring Tuxedo Authentication for Web Service Clients

BEA SALT GWWS servers rely on Tuxedo authentication framework to check the validity of the Web Service clients. If your legacy Tuxedo application is already applied with, Web Service clients must send user profile using one of the following approaches:

- HTTP Basic Authentication in the HTTP message header
- Web Service Security Username Token in the SOAP message header

Contrarily, if you want to authenticate Web Service clients for BEA SALT, you must configure Tuxedo authentications in the Tuxedo domain.

For more information about Tuxedo authentication, see “[Administering Authentication](#)” in the *BEA Tuxedo 9.1 Documentation*.

Configuring Tuxedo Security Level for Outbound HTTP Basic Authentication

To obtain Tuxedo client `uid` / `gid` for outbound HTTP Basic Authentication `username` / `password` mapping, you need to configure Tuxedo Security level as `USER_AUTH`, `ACL` or `MANDATORY_ACL` in the `UBBCONFIG` file.

[Listing 2-20](#) lists a segment of the `UBBCONFIG` file that shows how to define security level `ACL` in the `UBBCONFIG` file.

Listing 2-20 Security Level ACL Defined in the UBBCONFIG File For Outbound HTTP Basic Authentication

```
*RESOURCES
IPCKEY ...
.....
SECURITY ACL
.....
```

Configuring BEA SALT In Tuxedo MP Mode

To set up `GWWS` servers running on multiple machines within a MP mode Tuxedo domain, each Tuxedo machine must be defined with a separate `SALTDEPLOY` file and a set of separate other components.

You must propagate the following global resources across different machines:

- Certificates. Private key file and the trusted certificate files must be accessible from each machine according to the settings defined in the `SALTDEPLOY` file.
- Plug-in load libraries. Plug-in shared libraries must be compiled on each machine and must be accessible according to the settings defined in the `SALTDEPLOY` file.

You may define two `GWWS` servers running on different machine with the same functionality by associating the same `WSDF` files. But it requires manual propagation of the following artifacts:

- The WSDF files
- The WS-Policy files
- FML32 field table definition files if Tuxedo Services consume FML32 typed buffers
- XML Schema files excerpted by wsdlcvr.

Migrating from BEA SALT 1.1

This section describes the following two possible migrating approaches for SALT 1.1 customers who plan to upgrade to SALT 2.0 release:

- [Running GWWS servers with SALT 1.1 Configuration File](#)
- [Adopting SALT 2.0 Configuration Style by Converting SALT 1.1 Configuration File](#)

Running GWWS servers with SALT 1.1 Configuration File

After upgrading from SALT 1.1 to SALT 2.0 release, you may still want to run your existing SALT applications with the original SALT 1.1 configuration file. SALT 2.0 definitely supports that.

SALT configuration compiler utility, `wsloadcf`, supports to load the binary version `SALTCONFIG` from one SALT 1.1 format configuration file.

To run SALT 2.0 `GWWS` servers with SALT 1.1 Configuration file, you need to perform the following steps:

1. Load the binary version `SALTCONFIG` from the SALT 1.1 format configuration file via `wsloadcf`.
2. Set environment variable `SALTCONFIG` before booting the `GWWS` servers.
3. Boot the `GWWS` servers associated with this SALT 1.1 configuration file.

Note: If customers have more than one SALT 1.1 configuration files defined in a Tuxedo domain, customers need to follow step 1 to 3 to generate more binary version `SALTCONFIG` files and boot corresponding `GWWS` servers.

Adopting SALT 2.0 Configuration Style by Converting SALT 1.1 Configuration File

When `wsloadcf` loads a binary version `SALTCONFIG` from a SALT 1.1 configuration file, it also convert this SALT 1.1 configuration file into one `WSDF` file and one `SALTDEPLOY` file.

It's highly recommended to start using the SALT 2.0 styled configuration once you get the converted files from SALT 1.1 configuration.

Note: If customers want to incorporate more than one SALT 1.1 configuration files into one SALT 2.0 deployment, customers need to manually edit the `SALTDEPLOY` file for importing the other `WSDF` files.

The following sample lists the converted `SALTDEPLOY` file and `WSDF` file from a given SALT 1.1 configuration file.

Listing 2-21 A Sample of SALT 1.1 Configuration File (simpapp.xml)

```
<Configuration xmlns=" http://www.bea.com/Tuxedo/Salt/200606">
  <Servicelist id="simpapp">
    <Service name="toupper" />
    <Service name="tolower" />
  </Servicelist>
  <Policy />
  <System />
  <WSGateway>
    <GWInstance id="GWWS1">
      <HTTP address="//127.0.0.1:7805" />
      <HTTPS address="127.0.0.1:7806" />
      <Property name="timeout" value="300" />
    </GWInstance>
  </WSGateway>
</Configuration>
```

The converted SALT 2.0 `WSDF` file and deployment file are listed below.

Listing 2-22 Converted WSDF File for SALT 1.1 Configuration File (simpapp.xml.wsdf)

```

<Definition name="simpapp" wsdlNamespace="urn:simpapp.wsdl"
  xmlns=" http://www.bea.com/Tuxedo/WSDF/2007">
  <WSBinding id="simpapp_binding">
    <Servicegroup id="simpapp">
      <Service name="toupper" />
      <Service name="tolower" />
    </Servicegroup>
    <SOAP>
      <AccessingPoints>
        <Endpoint id="simpapp_GWWS1_HTTPPort"
          address=http://127.0.0.1:7805/simpapp />
        <Endpoint id=" simpapp_GWWS1_HTTPSPort"
          address=https://127.0.0.1:7806/simpapp />
      </AccessingPoints>
    </SOAP>
  </WSBinding>
</Definition>

```

Listing 2-23 Converted SALTDEPLOY File for SALT 1.1 Configuration File (simpapp.xml.dep)

```

<Deployment xmlns=" http://www.bea.com/Tuxedo/SALTDEPLOY/2007">
  <WSDF>
    <Import location="/home/myapp/simpapp.wsdf" />
  </ WSDF>
  <WSGateway>
    <GWInstance id="GWWS1">
      <Inbound>
        <Binding ref="simpapp:simpapp_binding">
          <Endpoint use=" simpapp_GWWS1_HTTPPort" />
          <Endpoint use=" simpapp_GWWS1_HTTPSPort" />
        </Binding>
      </Inbound>
      <Properties>
        <Property name="timeout" value="300" />
      </Properties>
    </GWInstance>
  </WSGateway>
</Deployment>

```

```
        </Properties>
    </GWInstance>
</WSGateway>
</ Deployment>
```

Administering BEA SALT at Run Time

This section contains the following topics:

- [Browsing to the WSDL Document from the GWWS Server](#)
- [Tuning the GWWS Server](#)
- [Tracing the GWWS Server](#)
- [Monitoring the GWWS Server](#)
- [Troubleshooting BEA SALT](#)

Browsing to the WSDL Document from the GWWS Server

Each GWWS server automatically generates a WSDL document for each inbound native WSDF for that particular GWWS server. The WSDL document can be downloaded from any of the HTTP/S listening endpoints via HTTP GET.

Use the following URL to browse the WSDL document:

```
"http(s)://<host>:<port>/wsdl[? [id=<wsdf_name>]  
[&mappolicy=<pack|raw|mtom>] [&toolkit=<wls|axis>]]"
```

[Table 3-1](#) lists all WSDL document download options.

Table 3-1 WSDL Download Options

Option	Value Description
id	Specifies the native WSDL name for the WSDL document. The specified native WSDL must be imported via inbound direction by the GWWS server. If the option is not specified, the first inbound native WSDL is used.
mappolicy	<code>{ pack raw mtom }</code> Specifies the data mapping policies for certain Tuxedo Typed buffers for the generated WSDL document. Currently, this option impacts CARRAY typed buffers only. If the option is not specified, <code>pack</code> is used as the default value.
toolkit	<code>{ wls axis }</code> Use this option only if you have previously defined <code>mappolicy=raw</code> . Specify the client toolkit used so that the proper WSDL document description for a CARRAY typed buffer MIME attachment is generated. BEA SALT supports WebLogic Server and Axis for SOAP with Attachments. The default value is <code>wls</code> .

Note: The WSDL download URL supported by BEA SALT 2.0 is different from BEA SALT 1.1. In BEA SALT 1.1 release, one GWWS server adaptively supports both `RPC/encoded` and `document/literal` message style, both SOAP 1.1 and SOAP 1.2 version, from a given configuration file. In BEA SALT 2.0 release, each WSDL file associated with the GWWS server must be pre-combined with a certain SOAP version and a certain SOAP message style. So the following WSDL download options for SALT 1.1 GWWS server are deprecated in this release.

Table 3-2 Deprecated WSDL Download Options

Option	Value Description
SOAPversion	This deprecated option is used to specify the expected SOAP version defined in the generated WSDL document. Now this option is set in the WSDL file.
encstyle	This deprecated option is used to specify the expected SOAP message style defined in the generated WSDL document. Now this option is set in the WSDL file.

Tuning the GWWS Server

The GWWS server is a high performance gateway used between external Web Service application and the Tuxedo application. It uses a thread-pool working model to improve performance in a multi-processor server environment. The GWWS server also provides options to control runtime behavior by setting the <WSGateway> element property values in the BEA SALT configuration file. The following topics list deployment considerations based on different scenarios. For more information, see [“Configuring the GWWS Servers” on page 2-17](#).

Thread Pool Size Tuning

Property: `thread_pool_size`

The default thread pool size is 16, but in some cases this may not be enough to handle high volume loads. It is recommended to conduct a typical usage analysis in order to better estimate the proper size requirement. Usually, if the concurrent client number is large (for example, more than 500), it is suggested that you deploy the GWWS gateway on a server with at least a 4-way processor and set the thread pool size to 64.

Network Timeout Control

Property: `timeout`

BEA SALT provides a network timeout tuning parameter in the configuration file. The default timeout value is 300 seconds. The value can be adjusted to reduce timeout errors.

Max Content Length Control

Property: `max_content_length`

BEA SALT administrators may want to limit the buffer size sent from a client. SALT supports this by using a property value that can be set for particular GWWS instances. By default there is no limit.

Backlog Control

Property: `max_backlog`

The default backlog socket listen value is 20. On some systems, such as Windows, 20 may not meet heavy load requirements. The client connection is rejected during TCP handshake.

The recommended value for Windows is based on the max concurrent TCP connections you may encounter. For example, if 80 is the peak point, you may configure the `max_backlog` property value to 60 in the SALT configuration file.

Note: The default backlog value is adequate for most systems. You do not need to tune it unless you experience client connection problems during heavy loads.

WARNING: A large backlog value may increase *syn-blood* attack risk.

Tuxedo BLOCKTIME

A network receive timeout property is provided in the SALT configuration file. Web service applications are also impacted by the Tuxedo BLOCKTIME parameter. Blocktime accounting begins when a message is transformed from XML to a typed buffer and delivered to the Tuxedo framework.

If no reply is received for a particular Web service client within the BLOCKTIME time frame, the GWWS server sends a SOAP fault message to the client and terminates the connection. If the GWWS server receives a delayed reply, it drops this message because the client has been disconnected.

BLOCKTIME is defined in the [UBBCONFIG](#) file `*RESOURCE` section.

Boost Performance Using Multiple GWWS instances

If one GWWS instance is bottlenecked due to network congestion, low CPU resources and so on, multiple GWWS instances can be deployed with the same Web Service binding on distributed Tuxedo nodes.

Note: Even though multiple GWWS instances can provide the same logic functionality, from a client perspective, they are different Web service endpoints with different HTTP/S listen ports and addresses.

Tracing the GWWS Server

The GWWS server supports the Tuxedo `TMTRACE` utility, which is used to dynamically trace messages. All trace points are logged in the `ULOG` file. The switch is recommended to use only in diagnostic treatment.

Currently, only one `TMTRACE` format is supported: `TMTRACE=msg:ulog`.

The usage is the same as other Tuxedo `TMTRACE` categories, which means it can be set using the environment variable or by the `tadmin` command container through the `chtr` sub-command.

For more information about Tuxedo `TMTRACE`, see .

An example is shown in [Listing 3-1](#).

Listing 3-1 TMTRACE Messages Logged By the GWWS Server

```
183632.BOX1!GWWS.4612.4540.0: TRACE:ms:A HTTP message is received, SCO
index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:A SOAP message is received, SCO
index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:Begin data transformation of
request message, buffer type = STRING, SCO index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:End of data transformation of
request message, buffer type = STRING, SCO index=1023

183632. BOX1!GWWS.4612.840.0: TRACE:ms:Delivering a message to Tuxedo,
service name =TOUPPER, SCO index=1023

183632. BOX1!GWWS.4612.840.0: TRACE:ms:Got a message from Tuxedo, SCO
index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:Begin data transformation of reply
message, buffer type = STRING, SCO index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:End of data transformation of reply
message, buffer type = STRING, SCO index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:Send a http message to net, SCO
index=1023
```

A more complex log is generated by `TMTRACE=msg:uLog`, used in WS-ReliableMessaging communication. All the application and infrastructure messages are sent to ULOG. An example is shown in [Listing 3-2](#).

Listing 3-2 WS-ReliableMessaging TMRACE Messages

184706.BOX1!GWWS.3640.4772.0: TRACE:ms:A HTTP message is received, SCO index=1023

184706.BOX1!GWWS.3640.4772.0: TRACE:ms:A HTTP Get request is received, SCO index=1023

184706.BOX1!GWWS.3640.4772.0: TRACE:ms:Send a http message to net, SCO index=1023

184710.BOX1!GWWS.3640.4772.0: TRACE:ms:A HTTP message is received, SCO index=1022

184710.BOX1!GWWS.3640.4772.0: TRACE:ms:A SOAP message is received, SCO index=1022

184710.BOX1!GWWS.3640.4772.0: TRACE:ms:Create a new inbound sequence, ID=uuid:4F1FEE40-72CB-118C-FFFFFFC0FFFFFFA8FFFFFFFEB010000-1811

184710.BOX1!GWWS.3640.4772.0: TRACE:ms:Create a new outbound sequence, ID=uuid:f7f76200-f612-11da-990d-9f37c3d14ba7

184710.BOX1!GWWS.3640.4772.0: TRACE:ms:Send CreateSequenceResponse message for sequence uuid:4F1FEE40-72CB-118C-FFFFFFC0FFFFFFA8FFFFFFFEB010000-1811

184710.BOX1!GWWS.3640.4772.0: TRACE:ms:Send a http message to net, SCO index=1022

184712.BOX1!GWWS.3640.3260.0: TRACE:ms:A HTTP message is received, SCO index=1022

184712.BOX1!GWWS.3640.3260.0: TRACE:ms:A SOAP message is received, SCO index=1022

184712.BOX1!GWWS.3640.3260.0: TRACE:ms:Begin data transformation of request message, buffer type = STRING, SCO index=1022

```
184712.BOX1!GWWS.3640.3260.0: TRACE:ms:End of data transformation of
request message, buffer type = STRING, SCO index=1022
```

```
184712.BOX1!GWWS.3640.3260.0: TRACE:ms:Received a request message in
sequence uuid:4F1FEE40-72CB-118C-FFFFFFC0FFFFFFA8FFFFFFEB010000-1811
```

Checking the ULOG tracing information helps to evaluate GWWS server SOAP message problem status.

Monitoring the GWWS Server

The GWWS server can be monitored with `wsadmin` utility, which is a command line tool. This tool can show the running status of GWWS.

An example is shown in [Listing 3-3](#).

Listing 3-3 Use `wsadmin` to monitor GWWS

```
$wsadmin
wsadmin - Copyright (c) 2005-2006 BEA Systems, Inc.
Portions * Copyright 1986-1997 RSA Data Security, Inc.
All Rights Reserved.
Distributed under license by BEA Systems, Inc.
SALT is a registered trademark.
```

```
> gwstats -i abcd
GWWS Instance : abcd
```

```
Inbound Statistics :
```

```
-----
Request Response Succ :      74
Request Response Fail :      32
          Oneway Succ :         0
          Oneway Fail :         0

          Total Succ :      74
```

Total Fail : 32

Avg. Processing Time : 210.726 (ms)

Outbound Statistics :

```
-----
Request Response Succ :    0
Request Response Fail :    0
      Oneway Succ :    0
      Oneway Fail :    0

      Total Succ :    0
      Total Fail :    0
```

Avg. Processing Time : 0.000 (ms)

```
-----
Total request Pending :    0
Outbound request Pending :    0
      Active Thread Number :    2
```

> gws -i out -s getTemp

GWWS Instance : out

Service : getTemp

Outbound Statistics :

```
-----
Request Response Succ :    333
Request Response Fail :    139
      Avg. Processing Time : 143.064 (ms)
```

>

Command `gwstats` (abbreviated as `gws`) can display the statistics data of GWWS server with specific instance ID or of certain service of the GWWS server. The data include the amount of successful and failed request, etc.

Before `wsadmin` is executed, both `TUXCONFIG` and `SALTCONFIG` environment variable must be set. `wsadmin` supports both active mode and in-active mode, which means `wsadmin` is able to launch with/without booting the Tuxedo domain.

The following table lists `wsadmin` sub-commands.

Table 3-3 wsadmin sub-commands

Sub-Command	Description
<code>gwstats (gws)</code>	Show statistics information of GWWS server
<code>configstats (cstat)</code>	Show configuration information
<code>default (d)</code>	Specify the default -i option
<code>echo (e)</code>	Switch on/off echo of input
<code>paginate (page)</code>	Switch on/off paging the output
<code>verbose (v)</code>	Switch on/off verbose output
<code>quit (q)</code>	Quit <code>wsadmin</code>

Troubleshooting BEA SALT

The following sections explain how to troubleshoot a BEA SALT run-time failure:

- [GWWS Start Up Failure](#)
- [GWWS Rejects SOAP Request](#)
- [WSDL Document Generated Incorrectly or Rejected by SOAP Client Toolkit](#)

GWWS Start Up Failure

If the GWWS server fails to start, check the following:

- Tuxedo service contract configuration

Check the Tuxedo service contract definition is correct in the Tuxedo Service Metadata Repository and the Tuxedo Service Metadata Repository Server - `TMMETADATA` - is booted successfully.

- GWWS server license

The GWWS server requires an extra license from BEA to enable the functionality. Check to make sure it has been installed properly.

- GWWS server HTTP listen port configuration.

Check the GWWS server listen / WS-Addressing endpoints defined in the SALT configuration files. Avoid port conflicts with other applications.

- GWWS instance ID.

Check the GWWS instance ID to make sure the two names defined in `UBBCONFIG` and `SALTDEPLOY` file are consistent.

- `UBBCONFIG` file `MAXWSCLIENTS` definition.

Make sure that `MAXWSCLIENTS` is defined in the `*MACHINE` section of `UBBCONFIG` file on the computer where GWWS server is deployed.

- `RESTART=Y` and `REPLYQ=Y` parameters.

If the GWWS server is set to `RESTART=Y` in the `UBBCONFIG` file, `REPLYQ=Y` also must be defined.

- `SALTCONFIG` file.

Make sure the binary version `SALTCONFIG` file is compiled successfully and the environment variable `SALTCONFIG` is set correctly for the GWWS server.

GWWS Rejects SOAP Request

In some cases, the GWWS server may reject SOAP requests. The most common causes are:

- The WSDL document is outdated

The WSDL document used by SOAP clients is out of date and some services may not be available.

- The GWWS server environment variables are not set correctly

When exporting a Tuxedo service with FML/VIEW buffers to a Web service, make sure the related GWWS environment variables are set with valid values. The GWWS server needs this information for the data mapping conversion.

- Violated Tuxedo Service Metadata Repository restrictions

Check the SOAP client data and make sure Tuxedo Service Metadata Repository restrictions are not violated.

- Unavailable Tuxedo service

Make sure the Tuxedo service you want exported as a Web service is available.

WSDL Document Generated Incorrectly or Rejected by SOAP Client Toolkit

If the WSDL document is rejected by the Web Service client toolkit, do the following:

- Try to use the `document/literal` message style and SOAP 1.1 to define native Tuxedo WSDL file. This is also the default behavior.
- Use `tmwsdlgen` to generate the WSDL document manually and compare with the one downloaded by the GWWS server. If the `TMMETADATA` server is not started when the GWWS server booted, the GWWS server cannot obtain the correct service contract information. Therefore, the downloaded WSDL document does not contain the correct type definitions.

