

Oracle® Service Architecture Leveraging Tuxedo (SALT)

Administration Guide

10g Release 3 (10.3)

January 2009

ORACLE®

Copyright © 2006, 2009, Oracle and/or its affiliates. All rights reserved.

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

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

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

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

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

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

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

Contents

Oracle SALT Administration Overview

Basic Concepts for Administering Oracle SALT	1-1
Tuxedo Service Metadata	1-1
Oracle SALT Web Service Deployment Model	1-2
Oracle SALT Web Services Administrative Tasks and Tools.....	1-3
Configuring a SALT Application Using Command-Line Utilities.....	1-4
Administering a SALT Application Using Command-Line Utilities	1-5
Oracle SALT SCA Deployment Model	1-5
Oracle SALT SCA Administrative Tasks and Tools.....	1-9
Configuring a Password for an SCA Client	1-10
Administering a TUXEDO SCA Application	1-10
See Also.....	1-11

Configuring an Oracle SALT Application

Configuring Tuxedo Web Services	2-1
Using Tuxedo Service Metadata Repository for Oracle SALT	2-2
Configuring Native Tuxedo Services	2-8
Configuring External Web Services	2-13
Creating the SALT Deployment File	2-19
Configuring Advanced Web Service Messaging Features	2-26
Configuring Security Features	2-31
Compiling SALT Configuration	2-35

Configuring the UBBCONFIG File for Oracle SALT	2-36
Configuring Oracle SALT In Tuxedo MP Mode	2-41
Migrating from Oracle SALT 1.1.	2-42
Configuring Tuxedo SCA Components	2-45
Configuring an SCA ATMI Client	2-45
Configuring an SCA JATMI Client	2-46
Configuring an SCA Workstation Client	2-47
Configuring an SCA Web Service Client.	2-49
Configuring an SCA ATMI Server	2-51
Configuring an SCA Web Service Server	2-53
Configuring SCA Client Security	2-56
Configuring Service Contract Discovery	2-60
tpforward Support	2-61
Service Contract Text File Output	2-61
See Also.	2-64

Administering Oracle SALT at Run Time

Administering Tuxedo Web Services	3-1
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 Oracle SALT	3-10
Administering Tuxedo SCA Components.	3-12
Tracing the SCA ATMI Server and Client.	3-12
Monitoring SCA ATMI Servers.	3-14
Tracing SCA JATMI Clients	3-15
See Also.	3-19

Appendix A: Oracle Tuxedo SCA Schemas

ATMI and JTMI Binding Schema For C/C++	A-1
Web Service Binding Schema	A-7
See Also	A-9

Oracle SALT Administration Overview

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

- [Basic Concepts for Administering Oracle SALT](#)
- [Oracle SALT Web Services Administrative Tasks and Tools](#)
- [Oracle SALT SCA Administrative Tasks and Tools](#)

Basic Concepts for Administering Oracle SALT

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

- [Tuxedo Service Metadata](#)
- [Oracle SALT Web Service Deployment Model](#)
- [Oracle SALT SCA Deployment Model](#)

Tuxedo Service Metadata

Starting with the Oracle 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 Oracle 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 Oracle SALT, you must define and load your Tuxedo service metadata in the Tuxedo Service Metadata Repository. Oracle SALT can then define the corresponding SOAP message format from the Tuxedo service metadata.

When invoking external Web services from a Tuxedo application, Oracle 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. SALT proxy 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 Oracle SALT gateway `GWWS` server.

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

Oracle SALT Web Service Deployment Model

Deploying Oracle SALT requires two configuration file types:

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

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 Oracle 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 2-8.

- 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

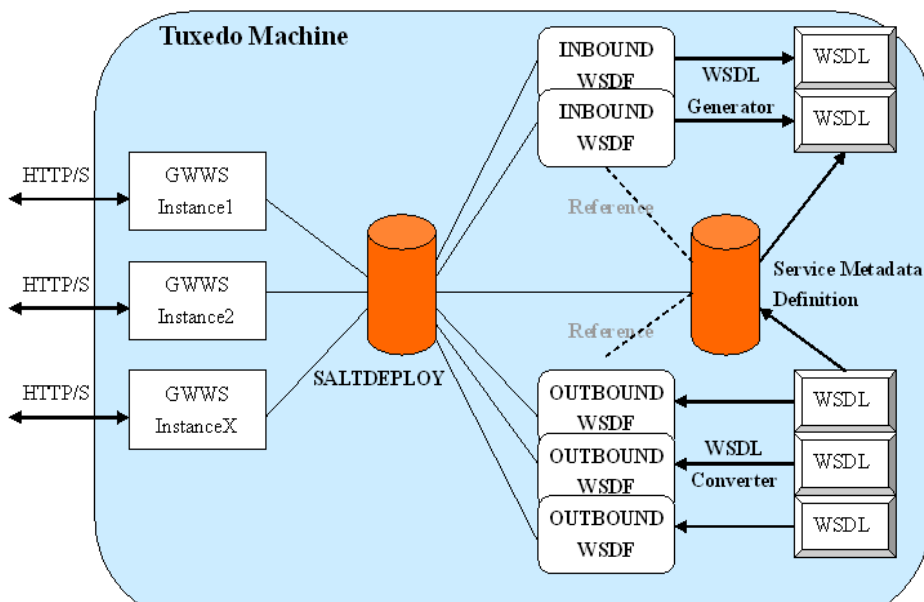
WSDL (except to configure advanced features). For more information, see [“Configuring External Web Services”](#) on page 2-13.

SALT Deployment File

The SALT Deployment File (`SALTDEPLOY`) is an XML-based file used to define Oracle SALT GWWS server deployment information on a *per* Tuxedo machine basis. The `SALTDEPLOY` file lists all necessary WSDL 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 2-19.

Figure 1-1 illustrates the Oracle SALT deployment model.

Figure 1-1 SALT Deployment Model



Oracle SALT Web Services Administrative Tasks and Tools

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

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

Configuring a SALT Application Using Command-Line Utilities

You can configure your Oracle 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 Oracle 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 Oracle SALT built on the Oracle Tuxedo framework, you should also use the following Oracle Tuxedo provided command-line utilities to configure Oracle 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 Oracle 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 a SALT Application Using Command-Line Utilities

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

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

Oracle SALT SCA Deployment Model

An SCA composite is typically described in an associated configuration file, the file name ends with `.composite`. This file uses an XML-based format call the Service Component Definition Language (SCDL) to describe the components this composite contains and specify how they related to one another. Deploying Oracle SALT SCA requires at least one root composite file that is located in `$APPDIR`.

There are two configuration file types:

- [SCA Composite Configuration File](#) (`.composite`)
- [SCA Component Configuration File](#) (`.componentType`)

There can be one or more components configured in the root composite file, and each of these components has its own `.composite` and `.componentType` file residing in its own subdirectory.

SCA Composite Configuration File

There can be zero or more component elements within a composite. The root composite files must be stored in `$APPDIR` in a server environment.

[Listing 1-1](#) shows an example of a root composite which contains two components:

Listing 1-1 Root Composite with Two Components

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="ECHO.app">
  <component name="ECHO">
    <implementation.composite name="ECHO" />
  </component>
  <component name="TOUPPER">
    <implementation.composite name="TOUPPER" />
  </component>
</composite>
```

Based on the configuration in Listing 1-1, [Listing 1-2](#) shows the implied the directory hierarchy.

Listing 1-2 SCA Composite Directory Hierarchy

```
$APPDIR/ECHO.app.composite
$APPDIR/ECHO
$APPDIR/ECHO/ECHO.composite
$APPDIR/ECHO/ECHO.componentType
$APPDIR/TOUPPER
$APPDIR/TOUPPER/TOUPPER.composite
$APPDIR/TOUPPER/TOUPPER.componentType
```

This example is a typical server configuration. The Tuxedo SCA client also has a similar application topology meaning that the client application is located in a subdirectory of the root composite file. [Listing 1-3](#) lists the directory structure for a client named `EchoClient` that uses the `ECHO1` service provided by `ECHO`.

Listing 1-3 Directory Structure

```
$APPDIR/root.composite
$APPDIR/EchoClient/EchoClient.composite
```

```
$APPDIR/EchoClient.composite
$APPDIR/EchoClient/EchoClient.dll
$APPDIR/EchoClient/EchoClient.exe
```

Note: One slight difference between an SCA server environment and an SCA client environment is that there is no need to have a component configuration file in the client environment.

SCA Component Configuration File

Components are the basic elements of business function in an SCA assembly, which are combined into complete business solutions by SCA composites. Components are configured instances of implementations. Components provide and consume services. More than one component can use and configure the same implementation, where each component configures the implementation differently.

Components are declared as sub-elements of a composite in an `xxx.composite` file. A component is represented by a component element that is a child of the composite element. Using the composite from [Listing 1-1](#), the 2 components (ECHO and TOUPPER), contains specific information. For the ECHO service (`$APPDIR/ECHO/ECHO.composite`), the `ECHO.composite` information is shown in [Listing 1-4](#).

Listing 1-4 ECHO.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  name="ECHO">
  <service name="ECHO">
    <interface.cpp header="ECHO.h" />
    <binding.atmi requires="legacy">
      <map target="EchoString1">ECHO1</map>
      <map target="EchoString2">ECHO2</map>
    </binding.atmi>
    <reference>EchoServiceComponent</reference>
  </service>
  <component name="EchoServiceComponent">
    <implementation.cpp library="ECHO" header="ECHOImpl.h" />
```

```
    </component>
</composite>
```

The ECHO service provides two TUXEDO services: ECHO1 and ECHO2. ECHO1 executes CPP function "EchoString1". ECHO2 executes CPP function "EchoString2". The existence of \$APPDIR/ECHO/ECHOImpl.componentType and \$APPDIR/ECHO/ECHO.so. are implied.

[Listing 1-5](#) shows information that may be contained in ECHOImpl.componentType.

Note: On some Unix systems the suffix is .so.71 or .sl.

ECHO.so (or ECHO.dll Windows), is the shared library that contains the actual implementation of EchoString1 and EchoString2 and is loaded into memory when the service is initialized. ECHO1 and ECHO2 are dynamically advertised at server initialization. For example, if EchoServer is the Tuxedo server that provides these two services, the Tuxedo UBBCONFIG file should contain information as shown in [Listing 1-6](#).

Listing 1-5 ECHOImpl.componentType

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.oesa.org/xmlns/sca/1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <service name="ECHO">
    <interface.cpp header="ECHO.h"/>
  </service>
</componentType>
```

Listing 1-6 UBBCONFIG File Example

```
...
*SERVERS
DEFAULT:
    CLOPT="-A"
EchoServer SRVGRP=GROUP1 SRVID=1001
...
```

For the TOUPPER service, the existence of `$APPDIR/TOUPPER/TOUPPER.composite` is also implied by the `ECHO.app.composite` file. Listing 1-7 shows information that may be contained in `TOUPPER.composite` file.

Listing 1-7 TOUPPER.composite file Example

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  name="TOUPPER">
  <service name="TOUPPER">
    <interface.cpp header="TOUPPER.h" />
    <binding.atmi requires="legacy">
      <map target="UpperString1">TOUPPER1</map>
      <map target="UpperString2">TOUPPER2</map>
    </binding.atmi>
    <reference>ToupperServiceComponent</reference>
  </service>

  <component name="ToupperServiceComponent">
    <implementation.cpp library="TOUPPER" header="TOUPPERImpl.h"
  />
  </component>
</composite>
```

This composite file also implies the existence of `$APPDIR/TOUPPER/TOUPPERImpl.componentType` and `$APPDIR/TOUPPER/TOUPPER.so`.

Note: Tuxedo SCA only supports "cpp" implementation types.

Oracle SALT SCA Administrative Tasks and Tools

Oracle SALT SCA provides a set of command line utilities for managing Oracle SALT SCA application built on top of the Tuxedo system. The majority of these utilities are geared toward application development but two of them are for administrative purposes. For more information, see the Oracle SALT Reference Guide.

The administrative tools can be used for the following tasks:

- [Configuring a Password for an SCA Client](#)
- [Administering a TUXEDO SCA Application](#)

Configuring a Password for an SCA Client

You can configure your SALT SCA security by using the `scapasswordtool` command line utility. It manages passwords for Tuxedo authentication in the SCA client and populates the password store file (`password.store`).

When an SCA component makes a reference to a Tuxedo-based service with Tuxedo security set to `APP_PW` or higher, the SCA component searches the `password.store` file to find the matching password and `userID` in the configuration composite file. The password is encrypted with the `userID` in plain text.

Creating a Password

To add/create a user ID and password, do the following:

1. prompt > Enter `scapasswordtool -i userID -a`
2. prompt > Enter password: `password`
3. prompt > Confirm password: `password`

The password is not echoed on the console screen.

Deleting a Password

To delete a `userID` and password, enter: `scapasswordtool -i userID -d`

The `userID` and associated password are deleted.

Administering a TUXEDO SCA Application

Tuxedo SCA components that declare services with ATMI binding are administered as a regular Tuxedo service. The administrator can to boot new instances of these servers, monitor or shut them down using existing Tuxedo commands such as `tmadmin`, `tmboot`, `tmshutdown`.

The Administrator can monitor activity or availability of specific methods by using `tmadmin` and selecting the services declared in the ATMI bindings defined in the `SCDL` file.

Each SCA server built using the `buildscaserver` command will also have administrative functionality that can be invoked using the `scaadmin` command line utility.

See Also

- [scaadmin](#)
- [scapasswordtool](#)
- [buildscaclient](#)
- [buildscacomponent](#)
- [buildscaserver](#)
- [Oracle Salt Programming Guide](#)
- [Oracle Salt Reference Guide](#)

Configuring an Oracle SALT Application

This section contains the following topics:

- [Configuring Service Contract Discovery](#)
- [Configuring Tuxedo SCA Components](#)
- [Configuring Service Contract Discovery](#)

Configuring Tuxedo Web Services

- [Using Tuxedo Service Metadata Repository for Oracle SALT](#)
- [Configuring Native Tuxedo Services](#)
- [Configuring External Web Services](#)
- [Configuring Service Contract Discovery](#)
- [Creating the SALT Deployment File](#)
- [Configuring Advanced Web Service Messaging Features](#)
- [Configuring Security Features](#)
- [Compiling SALT Configuration](#)
- [Configuring the UBBCONFIG File for Oracle SALT](#)

- [Configuring Oracle SALT In Tuxedo MP Mode](#)
- [Migrating from Oracle SALT 1.1](#)

Using Tuxedo Service Metadata Repository for Oracle SALT

Oracle 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 `GWSS` 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 Oracle SALT](#)
- [Defining Service Parameters for Oracle SALT](#)

Defining Service-Level Keywords for Oracle 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 Oracle SALT and are ignored when SALT components load the Tuxedo Service Metadata Repository.

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

Service-Level Keyword	Oracle SALT Usage
service	<p>The unique key value of the service. This value is referenced in the SALT WSDL file.</p> <p>For native Tuxedo services, this value can be the same as the Tuxedo advertised service name or an alias name differ from the actual Tuxedo advertised service name.</p> <p>For SALT proxy services, this value typically is the Web service operation local name.</p>
servicemode	<p>Determines the service mode (i.e., native Tuxedo service or SALT proxy service).</p> <p>The valid values are:</p> <ul style="list-style-type: none"> • tuxedo represents a native Tuxedo service • webservice represents a SALT proxy service, i.e. a service definition converted from a wsdl:operation <p>Do not use “webservice” to define a native Tuxedo service. This value is always used to define services converted from external Web services.</p>
tuxservice	<p>The actual Tuxedo advertised service name. If no value is specified, then the value is the same as the value in the <code>service</code> keyword.</p> <p>For native Tuxedo service, Oracle SALT invokes the Tuxedo service defined using this keyword.</p> <p>For SALT proxy service, GWWS server advertises the service name using this keyword value.</p>
servicetype	<p>Determines the service message exchange pattern for the specified Tuxedo service.</p> <p>The following values specify mapping rules between the Tuxedo service types and Web Service message exchange pattern (MEP):</p> <ul style="list-style-type: none"> • service corresponds to request-response MEP • oneway corresponds to oneway request MEP • queue corresponds to request-response MEP

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

Service-Level Keyword	Oracle SALT Usage
inbuf	<p>Specifies the input buffer (request buffer) type for the service.</p> <p>For native Tuxedo services, the value can be any Tuxedo typed buffer type. The following values are Tuxedo reserved buffer types: STRING, CARRAY, XML, MBSTRING, VIEW, VIEW32, FML, FML32, X_C_TYPE, X_COMMON, X_OCTET, NULL (input buffer is empty)</p> <p>Note: The value is case sensitive, if inbuf specifies any other type other than the previous buffer types, the buffer is treated as a custom buffer type.</p> <p>For SALT proxy services, the value is always FML32.</p>
outbuf	<p>Specifies the output buffer (response buffer with TPSUCCESS) type for the service.</p> <p>For native Tuxedo services, the value can be any Tuxedo typed buffer type. The following values are Tuxedo reserved buffer types: STRING, CARRAY, XML, MBSTRING, VIEW, VIEW32, FML, FML32, X_C_TYPE, X_COMMON, X_OCTET, NULL (input buffer is empty)</p> <p>Note: The value is case sensitive, if outbuf specifies any other type other than the previous buffer types, the buffer is treated as a custom buffer type.</p> <p>For SALT proxy services, the value is always FML32.</p>
errbuf	<p>Specifies the error buffer (response buffer with TPFALL) type for the service.</p> <p>For native Tuxedo services, the value can be any Tuxedo typed buffer type. The following values are Tuxedo reserved buffer types: STRING, CARRAY, XML, MBSTRING, VIEW, VIEW32, FML, FML32, X_C_TYPE, X_COMMON, X_OCTET, NULL (input buffer is empty)</p> <p>Note: The value is case sensitive, if errbuf specifies any other type other than the previous buffer types, the buffer is treated as a custom buffer type.</p> <p>For SALT proxy services, the value is always FML32.</p>

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

Service-Level Keyword	Oracle SALT Usage
inview	<p>Specifies the view name used by the service for the following input buffer types:</p> <p>VIEW, VIEW32, X_C_TYPE, X_COMMON</p> <p>Oracle SALT requires that you specify the view name rather than accept the default <code>inview</code> setting.</p> <p>This keyword is for native Tuxedo services only.</p>
outview	<p>Specifies the view name used by the service for the following output buffer types:</p> <p>VIEW, VIEW32, X_C_TYPE, X_COMMON</p> <p>Oracle SALT requires that you specify the view name rather than accept the default <code>outview</code> setting.</p> <p>This keyword is for native Tuxedo services only.</p>
errview	<p>Specifies the view name used by the service for the following error buffer types:</p> <p>VIEW, VIEW32, X_C_TYPE, X_COMMON</p> <p>Oracle SALT requires that you specify the view name rather than accept the default <code>errview</code> setting.</p> <p>This keyword is for native Tuxedo services only.</p>
inbufschema	<p>Specifies external XML Schema element associated with the service input buffer. If this value is specified, Oracle SALT incorporates the external schema in the generated WSDL to replace the default data type mapping rule for the service input buffer.</p> <p>This keyword is for native Tuxedo services only.</p>

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

Service-Level Keyword	Oracle SALT Usage
outbufschema	<p>Specifies external XML Schema element associated with the service output buffer. If this value is specified, Oracle SALT incorporates the external schema in the generated WSDL to replace the default data type mapping rule for the service output buffer.</p> <p>This keyword is for native Tuxedo services only.</p>
errbufschema	<p>Specifies external XML Schema element associated with the service error buffer. If this value is specified, Oracle SALT incorporates the external schema in the generated WSDL to replace the default data type mapping rule for the service error buffer.</p> <p>This keyword is for native Tuxedo services only.</p>

Defining Service Parameters for Oracle 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 Oracle SALT and are ignored when SALT components load the Tuxedo Service Metadata Repository.

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

Parameter-level Keyword	Oracle SALT Usage
<code>param</code>	<p>Specifies the parameter name.</p> <ul style="list-style-type: none"> VIEW, VIEW32, X_C_TYPE, or X_COMMON Specifies the view structure member name in the <code>param</code> keyword. FML, FML32 Specifies the FML/FML32 field name in the <code>param</code> keyword. STRING, CARRAY, XML, MBSTRING, or X_OCTET Oracle SALT ignores the parameter definitions.
<code>type</code>	<p>Specifies the data type of the parameter.</p> <p>Note: Oracle SALT does not support <code>dec_t</code> and <code>ptr</code> data types.</p>
<code>subtype</code>	<p>Specifies the view structure name if the parameter type is <code>view32</code>. For any other typed parameter, Oracle SALT ignores this value.</p> <p>Note: Oracle SALT requires this value if the parameter type is <code>view32</code>.</p> <p>This keyword is for native Tuxedo service only.</p>
<code>access</code>	<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>
<code>count</code>	<p>The general definition applies for this parameter. For Oracle SALT, the value for the <code>count</code> parameter must be greater than or equal to <code>requiredcount</code>.</p>
<code>requiredcount</code>	<p>The general definition applies for this parameter. The default is 1. For Oracle SALT, the value for the <code>count</code> parameter must be greater than or equal to <code>requiredcount</code>.</p>

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

Parameter-level Keyword	Oracle SALT Usage
size	<p>This optional keyword restricts the maximum byte length of the parameter. It is only valid for the following parameter types: <code>STRING</code>, <code>CARRAY</code>, <code>XML</code>, and <code>MBSTRING</code></p> <p>If this keyword is not set, there is no maximum byte length restriction for this parameter.</p> <p>The value range is [0, 2147483647]</p>
paramschema	<p>Specifies the corresponding XML Schema element name of the parameter. It is generated by SALT WSDL converter.</p> <p>This keyword is for SALT proxy service only. Do not specify this keyword for native Tuxedo services.</p>
primetype	<p>Specifies the corresponding XML primitive data type of the parameter. It is generated by SALT WSDL converter according to SALT pre-defined XML-to-Tuxedo data type mapping rules.</p> <p>This keyword is for SALT proxy service only. Do not specify this keyword for native Tuxedo services.</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 WSDL](#)
- [Using WS-Policy Files](#)
- [Generating a WSDL File from a Native WSDL](#)

Creating a Native WSDL

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

Each native WSDL must be defined with a unique WSDL name. A WSDL 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 WSDL. 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.

[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>
```

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.

[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 Oracle SALT Plug-ins](#) in *Oracle SALT Programming Web Services* and [“Configuring Plug-in Libraries”](#) on page 2-26.

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>, <Output> or <Fault>) to specify which implementation of “P_CUSTOM_TYPE” category plug-in should be used to do the message conversion. The <MsgHandler> element content is the 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 WS-Policy files to use these features. The [Web Service Policy Framework specifications](#) provides a general purpose model and syntax to describe and communicate the policies of a Web Service.

To use WS-Policy files, the `<Policy>` element should be defined in the WSDL to incorporate these separate WS-Policy files. Attribute `location` is used to specify the policy file path, both abstract and relative file path are allowed. Attribute `use` is optionally used by message level assertion policy files to specify the applied messages, request (input) message, response (output) message, fault message, or the combination of the three.

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

- `<Servicegroup>`
 - If a WS-Policy file consists of Web Service Endpoint level Assertions, e.g. Reliable Messaging Assertion, the WS-Policy file applies to all endpoints that serving this `<Servicegroup>`.

- If a WS-Policy file consists of Web Service Operation level Assertions, e.g., Security Identity Assertion, the WS-Policy file applies to all services listed in this <Servicegroup>.
- If a WS-Policy file consists of Web Service Message level Assertions, e.g., Security SignedParts Assertion, the WS-Policy file applies to input, output and/or fault messages of all services listed in this <Servicegroup>.
 - Note: Oracle SALT only supports request message level assertions for the current release. You must only specify use="input" for message level assertion policy files.
- <Service>
 - If a WS-Policy file consists of Web Service Operation level Assertions, e.g. Security Identity Assertion, the WS-Policy file applies to this particular service.
 - If a WS-Policy file consists of Web Service Message level Assertions, e.g. Security SignedParts Assertion, the WS-Policy file applies to input, output and/or fault messages of this particular service.
 - Note: Oracle SALT only supports request message level assertions for the current release. You must only specify use="input" for message level assertion policy files.

Oracle SALT provides some pre-packaged WS-Policy files for most frequently used cases. These WS-Policy files are located under directory \$TUXDIR/udataobj/salt/policy. These files can be referenced using location="salt:<policy_file_name>".

[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="/usr/resc/all_input_msg_policy.xml" use="input" />
      <Service name="toupper">
        <Policy location="service_policy.xml" />
        <Policy location="/usr/resc/input_message_policy.xml"
          use="input" />
      </Service>
    </Servicegroup>
  </WSBinding>
</Definition>
```

```

        <Service name="tolower" />
    </Servicegroup>
    ....
</WSBinding>
</Definition>

```

For more information, see [“Specifying the Reliable Messaging Policy File in the WSDL File”](#) 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 command generates a WSDL file named “`appl.wsdl`” from a given WSDL named “`appl.wsdf`”:

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

Note: 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 WSDL 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. For more information, see [Data Type Mapping and Conversions](#) in the *Oracle SALT Programming Web Services* and [tmwsdlgen](#) in the *Oracle SALT Reference Guide*.

Configuring External Web Services

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

- [Converting a WSDL file into Tuxedo Definitions](#)
- [Post Conversion Tasks](#)

Converting a WSDL file into Tuxedo Definitions

Oracle SALT provides a WSDL conversion command utility to convert external WSDL files into Tuxedo definitions. The WSDL file is converted using Extensible Stylesheet Language Transformations (XSLT) technology. Apache Xalan Java 2.7.0 is bundled in SALT installation package and is used as the default XSLT toolkit.

Oracle SALT WSDL converter is composed of two parts:

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

The following sample command converts an external WSDL file and generates Tuxedo definition files.

```
wslcvt -i http://api.google.com/GoogleSearch.wsdl -o GSearch
```

[Table 2-3](#) lists the Tuxedo definition files generated by Oracle SALT WSDL Converter.

Table 2-3 Tuxedo Definition Files generated by Oracle SALT WSDL Converter

Generated File	Description
Tuxedo Service Metadata Repository input file	<p>Oracle SALT WSDL Converter converts each <code>wsdl:operation</code> to a Tuxedo service metadata syntax compliant service called SALT proxy service. SALT proxy services are advertised by GWWS servers to accept ATMI call from Tuxedo applications.</p>
FML32 field table definition file	<p>Oracle SALT maps each <code>wsdl:message</code> to a Tuxedo FML32 typed buffer. Oracle SALT WSDL Converter decomposes XML Schema of each message and maps each basic XML snippet as an FML32 field. The generated FML32 fields are defined in a definition table file, and the field name equals to the XML element local name by default.</p> <p>To access a SALT proxy service, Tuxedo applications must refer to the generated FML32 fields to handle the request and response message. FML32 environment variables must be set accordingly so that both Tuxedo applications and GWWS servers can map between field names and field identifier values.</p> <p>Note: You may want to re-define the generated field names due to field name conflict or some other reason. In that case, both Tuxedo Service Metadata Definition input file and FML32 field table definition file must be changed accordingly. For more information, see “Resolving Naming Conflict For the Generated SALT Proxy Service Definitions”.</p>
Non-native WSDF file	<p>Oracle SALT WSDL Converter converts the WSDL file into a WSDF file, which can be deployed to GWWS servers in the SALT deployment file for outbound direction. The generated WSDF file is so-called non-native WSDF file.</p> <p>Note: Please do not deploy non-native WSDF files for inbound direction.</p>
XML Schema files	<p>WSDL embedded XML Schema and imported XML Schema (XML Schema content referenced with <code><xsd:import></code>) are saved locally as <code>.xsd</code> files. These files are used by GWWS servers and need to be saved under the same directory.</p> <p>Note: New XML Schema environment variables <code>XSDDIR</code> and <code>XSDFILES</code> must be set accordingly so that GWWS servers can load these <code>.xsd</code> files.</p>

WSDL-to-Tuxedo Service Metadata Keyword Mapping

Table 2-4 lists WSDL Element-to-Tuxedo Service Metadata Definition Keyword mapping rules.

Table 2-4 WSDL Element-to-Tuxedo Service Metadata Definition Mapping

WSDL Element	Corresponding Tuxedo Service Metadata Definition Keyword	Note
/wsdl:definitions /wsdl:portType /wsdl:operation @name	service	SALT proxy service name. The keyword value equals to the operation local name.
	tuxservice	SALT proxy service advertised name in Tuxedo system. If the wsdl operation local name is less than 15 characters, keyword value equals to the operation local name, otherwise the keyword value is the first 15 characters of the operation local name.
/wsdl:definitions /wsdl:portType /wsdl:operation /wsdl:input	inbuf=FML32	WSDL operation messages are always mapped as Tuxedo FML32 buffer type. Please do not change the buffer type any way.
/wsdl:definitions /wsdl:portType /wsdl:operation /wsdl:output	outbuf=FML32	Note: For more information about wsdl message and FML32 buffer mapping, see XML-to-Tuxedo Data Type Mapping for External Web Services in the <i>Oracle SALT Programming Web Services</i> .
/wsdl:definitions /wsdl:portType /wsdl:operation /wsdl:fault	errbuf=FML32	

WSDL-to-WSDF Mapping

Table 2-5 lists WSDL Element-to-WSDF Element mapping rules.

Table 2-5 WSDL Element-to-WSDL Element Mapping

WSDL Element	WSDL Element	Note
/wsdl:definitions @targetNamespace	/Definition @wsdlNamespace	Each wsdl:definition maps to a WSDL Definition.
/wsdl:definitions /wsdl:binding	/Definition /WSBinding	Each wsdl:binding object maps to a WSDL WSBinding element.
/wsdl:definitions /wsdl:binding @type	/Definition /WSBinding /Servicegroup	Each wsdl:binding referenced wsdl:portType object maps to the Servicegroup element of the corresponding WSBinding element.
/wsdl:definitions /wsdl:binding /soap:binding	/Definition /WSBinding /SOAP @version	If namespace prefix “soap” refers to URI “http://schemas.xmlsoap.org/wsdl/soap/”, the SOAP version attribute value is “1.1”; If namespace prefix “soap” refers to URI “http://schemas.xmlsoap.org/wsdl/soap12/”, the SOAP version attribute value is “1.2”.
/wsdl:definitions /wsdl:binding /soap:binding @style	/Definition /WSBinding /SOAP @style	The WSDL WSBinding SOAP message style setting equals to the corresponding WSDL soap binding message style setting (“rpc” or “document”).
/wsdl:definitions /wsdl:binding /wsdl:operation	/Definition /WSBinding /Servicegroup /Service	Each wsdl:operation object maps to a Service element of the corresponding WSBinding element.
/wsdl:definitions /wsdl:port /soap:address	/Definition /WSBinding /SOAP /AccessingPoints /Endpoint	Each soap:address endpoint defined for a wsdl:binding object maps to a Endpoint element of the corresponding WSBinding element.

Post Conversion Tasks

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

- [Resolving Naming Conflict For the Generated SALT Proxy Service Definitions](#)
- [Loading the Generated SALT Proxy Service Metadata Definitions](#)
- [Setting Environment Variables for GWWS Runtime](#)

Resolving Naming Conflict For the Generated SALT Proxy Service Definitions

When converting a WSDL file, unexpected naming conflicts may be found due to truncation or lost context information. Before using the generated Service Metadata Definitions and FML32 field table files, the following potential naming conflicts must be eliminated first.

- Eliminating the duplicated service metadata keyword “`tuxservice`” definitions

The keyword `tuxservice` in the SALT proxy service metadata definition is the truncated value of the original Web Service operation local name if the operation name is more than 15 characters. The truncated `tuxservice` value may be duplicated for multiple SALT proxy service entries. Since GWWS server uses `tuxservice` values as the advertised service names, so you must manually resolve the naming conflict among multiple SALT proxy services to avoid uncertain service request delivery. To resolve the naming conflict, you should assign a unique and meaningful name to `tuxservice`.

- Eliminating the duplicated FML32 field definitions

When converting a external WSDL file into Tuxedo definitions, each `wsdl:message` is parsed and mapped as an FML32 buffer format which containing a set of FML32 fields to represent the basic XML snippets of the `wsdl:message`. By default, The generated FML32 fields are named using the corresponding XML element local names.

The FML32 field definitions in the generated field table file are sorted by field name so that duplicated names can be found easily. In order to achieve a certain SOAP/FML32 mapping, the field name conflicts must be resolved. You should modify the generated duplicated field name with other unique and meaningful FML32 field name values. The corresponding Service Metadata Keyword `param` values in the generated SALT proxy service definition must be modified accordingly. The generated comments of the FML32 fields and Service Metadata Keyword “`param`” definitions are helpful in locating the corresponding `name` and `param`.

Loading the Generated SALT Proxy Service Metadata Definitions

After potential naming conflicts are resolved, you should load the SALT proxy service metadata definitions into the Tuxedo Service Metadata Repository through `tmloadrepos` utility. For more information, see `tmloadrepos`, in the [Oracle Tuxedo Service Metadata Repository Documentation](#).

Setting Environment Variables for GWWS Runtime

Before booting GWWS servers for outbound web services, the following environment variable settings must be performed.

- Update environment variable *FLDTBLDIR32* and *FIELDTBLS32* to add the generated FML32 field table files.
- 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 Oracle SALT Reference Guide.

Importing the WSDF Files

You should import all 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 WSDF file must be 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. Each WSBinding object is referenced using attribute “ref=<wsdl_name>:<WSBinding id>”. 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 Server Level Properties

The GWWS server can be configured with properties that switch feature on/off or set argument to tune the server’s performance.

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. [Table 2-6](#) lists GWWS server level properties.

Table 2-6 GWWS Server Level Properties

Property Name	Description	Value Range	Default
enableMultiEncoding	Switch on/off the SOAP message multiple encoding support	"true" "false"	"false"
max_backlog	Specify socket backlog control value	[1, 255]	20
max_content_length	Specify the maximum allowed incoming HTTP message content length.	[0, 1G](byte) (Can set suffix 'M', 'G', e.g. 1.5M, 0.2G)	0 (means no limit)
thread_pool_size	Specify the GWWS server thread pool size.	[1, 1024]	16

Table 2-6 GWWS Server Level Properties

Property Name	Description	Value Range	Default
timeout	Specify the network timeout in seconds.	[1, 65535] (unit:sec)	300
wsm_acktime	Specify the Reliable Messaging Acknowledgement message reply policy. GWWS servers support replying acknowledgement messages either after receiving the SOAP request from network immediately or after the Tuxedo service returns the response message.	"NETRECV" "RPLYRECV"	"NETRECV"

Note: For more information about GWWS multiple encoding support, see [“Configuring Multiple Encoding Support”](#) on page 2-23.

For more information about Performance tuning properties, see [“Tuning the GWWS Server”](#) on page 3-3.

[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="thread_pool_size" value="20"/>
      <Property name="enableMultiEncoding" value="true"/>
      <Property name="timeout" value="600"/>
    </Properties>
  </GWInstance>
</WSGateway>
...
</ Deployment>

```

Configuring Multiple Encoding Support

SALT supports multiple encoding SOAP messages and the encoding mappings between SOAP message and Tuxedo buffer. SALT supports the following character encodings:

```
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
```

To enable the GWWS multiple encoding support, GWWS server level property “enableMultiEncoding” should be set to “true”.

Note: GWWS internally converts non UTF-8 external messages into UTF-8. However, encoding conversion hurts server performance. By default, encoding conversion is turned off and messages that are not UTF-8 encoded are rejected.

Listing 2-8 Configuring GWWS Server Multiple Encoding Property

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

[Table 2-7](#) explains the detailed SOAP message and Tuxedo buffer encoding mapping rules if the GWWS server level multiple encoding switch is turned on.

Table 2-7 SALT Message Encoding Mapping Rules

Mapping from ...	Mapping to ...	Encoding Mapping Rule
SOAP/XML	Tuxedo Typed Buffer	string/mbstring/xml buffer or field characters' encoding equals to SOAP xml encoding.
STRING Typed Buffer	SOAP/XML	GWWS sets the target SOAP message in UTF-8 encoding, and assumes the original STRING buffer containing only UTF-8 encoding characters. Note: Tuxedo Developers must ensure the STRING characters are in UTF-8 encoding.
MBSTRING/XML Typed Buffer	SOAP/XML	SOAP xml encoding equals to MBSTRING/XML encoding.
FML/32, VIEW/32 Typed Buffer that containing the same encoding setting for multiple FLD_MBSTRING fields	SOAP/XML	SOAP xml encoding is set to FLD_MBSTRING encoding, the original Typed buffer field characters are not changed in the SOAP message. Note: Tuxedo Developers must ensure the FLD_STRING characters in the same buffer are in consistent encoding.
FML/32, VIEW/32 Typed Buffer that containing the different encodings for multiple FLD_MBSTRING fields	SOAP/XML	SOAP xml encoding is set to UTF-8, the original Typed buffer FLD_MBSTRING field characters in other encodings are converted into UTF-8 in the SOAP message. Note: Tuxedo Developers must ensure the FLD_STRING characters in the same buffer are in UTF-8 encoding.

Configuring System Level Resources

Oracle 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 GWSS server to create an SSL listen endpoint, or to use X.509 certificates for authentication and/or message signature. All GWSS 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 GWSS 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 GWSS server. There are two ways to define GWSS server trusted certificates:

1. Include all certificates in one PEM format file and define the file path using the `<<Certificate>/<TrustedCert>` sub-element.
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 GWSS server certificates.

Listing 2-9 Configuring Certificates In the SALTDEPLOY File

```
<Deployment ..>
...
<System>
  <Certificates>
    <PrivateKey>/home/user/gwss_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 GWS server is running. Oracle SALT provides a plug-in framework as a common interface for defining and implementing plug-ins. Plug-in implementation is carried out through a dynamic library that contains the actual function code. The implementation library can be loaded dynamically during GWS server start up. The functions are registered as the implementation of the plug-in interface.

In order for the GWS 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 GWS 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 Oracle SALT](#) in Oracle SALT Programming with Web Services.

Configuring Advanced Web Service Messaging Features

Oracle SALT currently supports the following advanced Web Service Messaging features:

- [Web Service Addressing](#)

Supports both inbound and outbound asynchronous Web service messaging.

- [Web Service Reliable Messaging](#)

Supports inbound Web Service reliable message delivery.

Web Service Addressing

Oracle 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="https://myhost:8801/app1_async_point">
    </WSAddressing>
    <Endpoint use=" simpapp_GWWS1_HTTPPort" />
    <Endpoint use=" simpapp_GWWS1_HTTPSPort" />
  </Binding>
  <Binding ref="app2:app2_binding">
    <WSAddressing>
      <Endpoint address="https://myhost:8802/app2_async_point">
    </WSAddressing>
    <Endpoint use=" simpapp_GWWS1_HTTPPort" />
    <Endpoint use=" simpapp_GWWS1_HTTPSPort" />
  </Binding>
</Outbound>
...
</GWInstance>
</WSGateway>
...
</ Deployment>

```

Notes: In a 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

No matter you create a WS-Addressing endpoint or not in the SALTDEPLOY file, you can explicitly disable the Addressing capability for particular outbound services in the WSDF. To disable the Addressing capability for a particular outbound service, you should use the property name “disableWSAddressing” with a value set to “true” in the corresponding <Service> definition in the WSDF file. This property has no impact to any inbound services.

[Listing 2-12](#) shows WSDF 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

Oracle 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. The policy file must comply with the [WS-ReliableMessaging Policy Assertion Specification \(February 2005\)](#).

Note: A WSDL containing a Reliable Messaging policy definition should be used by the GWWS server for inbound direction only.

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 an 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 about the WS-ReliableMessaging policy file format, see the [Oracle SALT WS-ReliableMessaging Policy Assertion Reference](#) in the *Oracle 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="PLD" />
    <beapolicy:QOS QOS="ExactlyOnce InOrder" />
  </wsm:RMAssertion>
</wsp:Policy>
```

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 location="RMPolicy.xml" />
      <Service ... />
      <Service ... />
      ...
    </Servicegroup ...>
  </WSBinding>
</Definition>
```

Note: Reliable Messaging in Oracle SALT does not support process/system failure scenarios, which means SALT does not store the message in a persistent storage area. Oracle 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

Oracle 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

Oracle 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](#)”.

GWWS server automatically creates SSL secured connection to outbound endpoints that are published with URLs that having prefix “`https://`”.

Configuring Inbound HTTP Basic Authentication

Oracle SALT depends on the Tuxedo security framework for Web Service client authentication. There is no special configuration at Oracle 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 QWxhZGRpbjpvYGVuIHNlc2FtZQ==
```

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 are 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. Oracle SALT does not make group information available for Web service clients.

Configuring Outbound HTTP Basic Authentication

Oracle SALT supports customers to develop authentication plug-in to prepare the user credential for the outbound HTTP Basic Authentication. 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 username and password, 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>` setting, 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 *Oracle SALT Programming Web Services*.

Configuring Message Level Web Service Security

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

- Authentication, by requiring username or X.509 tokens
- Inbound request message integrity, by requiring the soap body signature

Main Use Cases of Web Service Security

Oracle 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.
- Include a token (X.509) and the soap body signature in the SOAP message for integrity.

Using WS-SecurityPolicy Files

Oracle 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 Oracle SALT.

The following table lists the default WS-Security Policy files bundled by Oracle SALT.

Table 2-8 WS-Security Policy Files Provided By Oracle 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-UserNameToken-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 used in `<Service>` level.

Listing 2-16 shows a combination of policy assignment making that the service “TOUPPER” requires client send a UsernameToken (in plain text format) and an X509v3Token in request, and also require the SOAP:Body part of message is signed with the X.509 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 a SALT default bundled policy file is used. User-defined policy file can be used by directly specifying the file path.

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 applied 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 Oracle 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 *Oracle SALT Reference Guide*.

Configuring the UBBCONFIG File for Oracle SALT

After configuring and compiling SALT configuration, Tuxedo `UBBCONFIG` file needs to be updated to apply SALT components in the Tuxedo application. [Table 2-9](#) lists the `UBBCONFIG` file configuration tasks for Oracle SALT.

Table 2-9 UBBCONFIG File Configuration Tasks for Oracle 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

Oracle 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 be present in 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 Oracle 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 generate 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 Oracle 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

Oracle 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 Oracle 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="gwsw_certkey"
    SEC_PRINCIPAL_VAR="gwsw_certkey"
    CLOPT="-A -- -i GW1"
GWWS SRVGRP=GROUP1 SRVID=11
    SEC_PRINCIPAL_NAME="gwsw_certkey"
    SEC_PRINCIPAL_PASSVAR="gwsw_certkey"
    CLOPT="-A -- -i GW2"
.....
```

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

Configuring Tuxedo Authentication for Web Service Clients

Oracle 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 credential 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 Oracle SALT, you must configure Tuxedo authentications in the Tuxedo domain.

For more information about Tuxedo authentication, see “[Administering Authentication](#)” in the *Oracle 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 Oracle 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 WSDL files
- The WS-Policy files
- FML32 field table definition files if Tuxedo Services consume FML32 typed buffers
- XML Schema files excerpted by `wsdlcvt`.

Migrating from Oracle 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 WSDL 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 WSDL files.

The following sample lists the converted `SALTDEPLOY` file and WSDL 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 WSDL file and deployment file are listed below.

Listing 2-22 Converted WSDL File for SALT 1.1 Configuration File (simpapp.xml.wsdl)

```
<Definition name="simpapp" wsdlNamespace="urn:simpapp.wsdl"
  xmlns=" http://www.bea.com/Tuxedo/WSDL/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>

```

Configuring Tuxedo SCA Components

Configuring Tuxedo SCA components comprises the following:

- [Configuring an SCA ATMI Client](#)
- [Configuring an SCA JATMI Client](#)
- [Configuring an SCA Workstation Client](#)
- [Configuring an SCA Web Service Client](#)
- [Configuring an SCA ATMI Server](#)
- [Configuring an SCA Web Service Server](#)
- [Configuring SCA Client Security](#)

Configuring an SCA ATMI Client

The SCA ATMI client is a native Tuxedo client that is written using the SCA paradigm and built using the `buildscaclient` utility. The client executable must be in a subdirectory of a directory that contains the `root.composite` file.

Note: The `APPDIR` environment variable must point to the `root.composite` file directory.

[Listing 2-24](#) shows the client application root composite file `$APPDIR/root.composite`.

Listing 2-24 Client Application Root Composite File

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="testApp">
  <component name="testStringClientComp">
    <implementation.composite name="ECHO"/>
  </component>
</composite>
```

The `$APPDIR/ECHO` directory contains the ECHO application. The directory name, "ECHO", must match the name specified in `<implementation.composite name="ECHO"/>`.

[Listing 2-25](#) shows the client application composite file.

Listing 2-25 Client Application Composite File

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="ECHO">
  <reference name="ECHO">
    <interface.cpp header="ECHO.h"/>
    <binding.atmi requires="legacy">
      <tuxconfig>/tux/application/ECHOServer/tuxconfig</tuxconfig>
      <inputBufferType target="TestString">STRING</inputBufferType>
      <outputBufferType target="TestString">STRING</outputBufferType>
      <errorBufferType target="TestString">STRING</errorBufferType>
    </binding.atmi>
  </reference>
</composite>
```

The client dynamic link library for this client application is also contained in this directory. For example, using the example in [Listing 2-25](#), the `$APPDIR/ECHO/ECHO.so` shared object exists in the ECHO directory. The target "TestStr" is used to group buffer types together.

The client executable also exists in this directory. There is no naming convention associated with a client application. This client ECHO application could very well contain "doEchoClient" in the ECHO application directory. For example: `$APPDIR/ECHO/doEchoClient`.

Note: You must set `SCA_COMPONENT`. See [Listing 2-25](#),
`SCA_COMPONENT=testStringClientComp`.

Configuring an SCA JATMI Client

The JATMI client application configuration composite file is part of the Java `.jar` file. The JATMI client composite file is not part of any package and is located in the base of the `.jar` file. When client application is invoked, SCA Java runtime loads the composite file. No special setup is required.

Note: The client application `.jar` file *must* be included in the `CLASSPATH`. The following `.jar` files should also be part of `CLASSPATH`:

- `binding-jatmi-extension.jar`
- `com.oracle.jatmi.dataxfm_1.0.0.0.jar`

- com.bea.core.jatmi_1.2.0.3.jar
- com.bea.core.i18n_1.4.0.0.jar
- tuscanysca-manifest.jar

[Listing 2-26](#) shows an SCA JATMI client composite file example.

Listing 2-26 SCA JATMI Client Composite File Example

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.oxa.org/xmlns/sca/1.0"
xmlns:f="binding-atmi.xsd"
name="EchoComposite">
  <reference name="ECHO" promote="EchoComponent/ECHO">
    <interface.java class="com.abc.sca.java.Echo" />
    <f:binding.atmi requires="legacy">
      <f:serviceType>RequestResponse</f:serviceType>
      <f:inputBufferType>FML</f:inputBufferType>
      <f:outputBufferType>FML</f:outputBufferType>
      <f:fieldTables>com.abc.sca.java.fml.FMLTABLE
      </f:fieldTables>
      <f:workStationParameters>
        <f:networkAddress>//STRIATUM:15011
        </f:networkAddress>
      </f:workStationParameters>
    </f:binding.atmi>
  </reference>
  <component name="EchoComponent">
    <implementation.java
      class="com.abc.sca.java.EchoComponentImpl" />
  </component>
</composite>
```

Configuring an SCA Workstation Client

Configuring an SCA workstation clients is similar to configuring SCA native clients. One difference is that an SCA workstation client requires using the `<workStationParameters>`

element and its sub-elements in the composite. The SCA runtime automatically detects whether the client is built as an SCA native client or SCA workstation client and loads the correct reference binding library accordingly.

An SCA Tuxedo Workstation client has a similar directory hierarchy to an SCA native client. Both rely on the environment variable `$APPDIR`, which points to where the client application is located.

[Listing 2-27](#) shows a SCA Tuxedo workstation client configuration example.

Listing 2-27 `$APPDIR/root.composite`

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="testApp">
  <component name="testStringClientComp">
    <implementation.composite name="ECHO"/>
  </component>
</composite>
```

Listing 2-28 `$APPDIR/ECHO/ECHO.composite`

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="ECHO">
  <reference name="ECHO">
    <interface.cpp header="ECHO.h"/>
    <binding.atmi requires="legacy">
      <inputBufferType target="TestString">STRING</inputBufferType>
      <outputBufferType target="TestString">STRING</outputBufferType>
      <errorBufferType target="TestString">STRING</errorBufferType>
    <workStationParameters>
      <networkAddress>//STRIATUM:4890</networkAddress>
      <encryptBits>128/128</encryptBits>
    </workStationParameters>
    <remoteAccess>WorkStation</remoteAccess>
  </binding.atmi>
</reference>
</composite>
```

Configuring an SCA Web Service Client

The SCA Web service client is basically the same as SCA native client except that it uses the `<binding.ws>` element instead of `<binding.atmi>`. The SCA runtime automatically detects which binding the client is using, and loads the correct reference binding accordingly.

The SCA Web service client has a similar directory hierarchy as native client. They both rely on the `$APPDIR` environment variable to point to where the client application is located.

[Listing 2-29](#) shows an SCA Web service client configuration example.

Listing 2-29 \$APPDIR/root.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="testApp">
  <component name="calcClient">
    <implementation.composite name="calcClient"/>
  </component>
</composite>
```

Listing 2-30 \$APPDIR/calcClient/calcClient.composite

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="calcClient">
  <reference name="Calculator">
    <interface.cpp header="CalcService.h"/>
    <binding.ws
      endpoint="http://calc.sample#wsdl.endpoint
      (Calculator/CalculatorSOAP11port)"/>
  </reference>
</composite>
```

The `<interface.cpp>` element is required to build the appropriate proxy stub. Also, the client directory should contain the WSDL file where the endpoint specified in `<binding.ws>` is located. In addition, the configuration of the Tuxedo Web Services gateway (GWWS) is necessary and requires the following steps:

1. Make sure the TMMETADATA and GWWS servers are shut down.
2. Run `wsdlcvt` on the WSDL of the service(s) used. This produces a WSDF file, a Tuxedo Metadata Repository interface definitions file, fml32 field tables and XML schemas.
3. Optionally, modify the generated WSDF file to override the actual endpoint address used at runtime. For more information, see WSDF documentation.
4. Load the Tuxedo Metadata Repository interface definitions into the TMMETADATA server repository (e.g.: `$ tmlloadrepos -I calc.mif metadata.repos -y`). For more information, see `tmlloadrepos` documentation.
5. Add a reference to the WSDF in the GWWS configuration input file (named `gwws.dep` for example). [Listing 2-31](#) shows the added elements highlighted in blue.
6. Reload the GWWS binary configuration file to take into account the changes performed in the previous five (e.g.: `$ wslloadcf -y gwws.dep`).
7. Reboot GWWS and TMMETADATA.

Listing 2-31 GWWS Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<saltdep:Deployment
xmlns:saltdep="http://www.bea.com/Tuxedo/SALTDEPLOY/2007"
xmlns="http://www.bea.com/Tuxedo/SALTDEPLOY/2007"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <saltdep:WSDF>
    <saltdep:Import location="calc.wsdf"/>
  </saltdep:WSDF>
  <saltdep:WSGateway>
    <saltdep:GWInstance id="GWWS1">
      <saltdep:Outbound>
        <saltdep:Binding ref="calc:CalculatorSOAP11Binding">
          <saltdep:Endpoint use="CalculatorSOAP11port"/>
        </saltdep:Binding>
      </saltdep:Outbound>
    </saltdep:GWInstance>
  </saltdep:WSGateway>
</saltdep:Deployment>
```

```

        </saltdep:Outbound>
    </saltdep:GWInstance>
</saltdep:WSGateway>
<saltdep:System/>
</saltdep:Deployment>

```

Configuring an SCA ATMI Server

For an SCA ATMI server, the SCA ROOT is the same as \$APPDIR. There should be at least one composite file that describes the SCA application. The SCA runtime searches for this composite file and from there it loads all the `composite` and `componentType` files for SCA server applications that are hosted in a Tuxedo environment.

[Listing 2-32](#) shows a root composite file, named `root.composite` contains two SCA applications hosted in a Tuxedo application domain. The two applications are called `Purchase` and `Finance`. There are at least two subdirectories for these two SCA applications. One is called `Purchase.component` and the other is called `Finance.component`.

Listing 2-32 \$APPDIR/root.composite

```

<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.oso.org/xmlns/sca/1.0" name="root">
    <component name="Purchase.component">
        <implementation.composite name="Purchase" />
    </component>
    <component name="Finance.component">
        <implementation.composite name="Finance" />
    </component>
</composite>

```

[Listing 2-33](#) shows the `Purchase.component` directory contains a composite file for the `Purchase` application named `Purchase.composite`. Similarly, the `Finance.component` directory contains a composite file for the `Finance` application named `Finance.composite`.

Listing 2-33 \$APPDIR/Purchase.component/Purchase.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
  name="Purchase">
  <service name="purchase">
    <interface.cpp header="Purchase.h" />
  <binding.atmi requires="legacy">
    <map target="Order">ORDER</map>
    <map target="TrackOrder">TRACKORDER</map>
  </binding.atmi>
  <reference>PurchaseServiceComponent</reference>
</service>
<component name="PurchaseServiceComponent">
  <implementation.cpp library="Purchase"
header="PurchaseImpl.h" />
</component>
</composite>
```

Listing 2-34 shows `Purchase.composite` contains the `PurchaseImpl.componentType` file in the `$APPDIR/Purchase.component` directory and uses `CPP` as its application implementation. When an SCA server using this configuration is built using the `buildscaserver` utility, it advertises two SCA services automatically at runtime (`ORDER` and `TRACKORDER`). The actual `CPP` implementation of the services is called `Order` and `TrackOrder`.

Listing 2-34 \$APPDIR/Purchase.component/PurchaseImpl.componentType

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.oesa.org/xmlns/sca/1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <service name="purchase">
    <interface.cpp header="Purchase.h" />
  </service>
</componentType>
```

Assume these two SCA applications hosted in Tuxedo and built using `buildscaserver` are called `PurchaseSvr` and `FinanceSvr`. You must add the following lines to the `*SERVERS` section in the `UBBCONFIG` file:

```
PurchaseSvr SRVGRP=PURCHASEGRP SRVID=500
FinanceSvr SRVGRP=FINANCEGRP SRVID=600
```

There is no need to add a service in the `*SERVICES` section. SCA services hosted by Tuxedo are dynamically advertised.

Configuring an SCA Web Service Server

Configuring Web services binding for components (server side) is similar to configuring ATMI binding for hosting SCA components.

[Listing 2-35](#) shows a root composite file named `root.composite`. It contains one SCA component hosted in a Tuxedo application domain. The two applications are called `Purchase` and `Finance`. There are at least two subdirectories for these two SCA applications, one is called `Purchase.component`, and the other is called `Finance.component`.

[Listing 2-36](#) shows the actual component subdirectory. [Listing 2-37](#) shows the `componentType` side file

Listing 2-35 \$APPDIR/root.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="root">
  <component name="account">
    <implementation.composite name="account" />
  </component>
</composite>
```

Listing 2-36 \$APPDIR/account/account.composite

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  name="account">
  <service name="AccountService">
```

```

        <interface.wsdl
interface="http://www.bigbank.com/AccountService#wsdl.interface(AccountSer
vice)"/>
        <binding.ws/>
            <reference>AccountServiceComponent</reference>
        </service>

        <component name="AccountServiceComponent">
            <implementation.cpp library="Account"
header="AccountServiceImpl.h"/>
        </component>
</composite>

```

Listing 2-37 \$APPDIR/account/AccountServiceImpl.componentType

```

<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <service name="AccountService">
        <interface.cpp header="AccountService.h"/>
    </service>
</componentType>

```

The above SCA component will be hosted in a Tuxedo server built using `buildscaserver` with the `-w` option (for Web Services) and named `WSServer`

Then in the Tuxedo `UBBCONFIG` file you need to add the following line in the `*SERVERS` section: `WSServer SRVGRP=ACCTGRP SRVID=500.`

There is no need add a service in the `*SERVICES` section. SCA services hosted by Tuxedo are dynamically advertised.

In addition, configuration of the Tuxedo Web Services gateway (GWWS) is necessary. Do the the following steps:

1. Make sure the `TMMETADATA` and `GWWS` servers are shut down

2. Run `wsdlcvt` on the WSDL of the service(s) used. This produces a WSDL file, a Tuxedo Metadata Repository interface definitions file, fml32 field tables and XML schemas
3. Modify the generated WSDL file to specify the actual endpoint address used at runtime to accept requests. For more information, see WSDL documentation.
4. Load the Tuxedo Metadata Repository interface definitions into the TMMETADATA server repository (e.g.:`$ tmlloadrepos -I AccountService.mif metadata.repos -y`). For more information, see `tmlloadrepos` documentation.
5. Add a reference to the WSDL in the GWWS configuration input file (named `gwws.dep` for example). [Listing 2-38](#) shows the elements added highlighted in blue.
6. Reload the GWWS binary configuration file to take into account the changes performed in the step five (e.g.: `$ wsloadcf -y gwws.dep`).
7. Reboot GWWS and TMMETADATA.

Listing 2-38 gwws.dep File

```
<?xml version="1.0" encoding="UTF-8"?>
<saltdep:Deployment
xmlns:saltdep="http://www.bea.com/Tuxedo/SALTDEPLOY/2007"
xmlns="http://www.bea.com/Tuxedo/SALTDEPLOY/2007"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <saltdep:WSDF>
        <saltdep:Import location="AccountService.wsdf"/>
    </saltdep:WSDF>
    <saltdep:WSGateway>
        <saltdep:GWInstance id="GWWS1">
            <saltdep:Inbound>
                <saltdep:Binding
ref="AccountService:AccountServiceSOAP">
                    <saltdep:Endpoint use="AccountServiceSOAP"/>
                </saltdep:Binding>
            </saltdep:Inbound>
        </saltdep:GWInstance>
    </saltdep:WSGateway>
</saltdep:Deployment>
```

```
        </saltdep:GWInstance>
    </saltdep:WSGateway>
    <saltdep:System/>
</saltdep:Deployment>
```

Configuring SCA Client Security

Tuxedo SCA components support two types of security:

- [Tuxedo Application Domain Security](#)
- [Tuxedo Link-Level Security](#)

Tuxedo Application Domain Security

Tuxedo Application Domain Security is set when the `TUXCONFIG` file for the Tuxedo Application Domain contains the `SECURITY` keyword in the `*RESOURCES` section. There are five levels of application security: `NONE`, `APP_PW`, `USER_PW`, `ACL`, and `MANDATORY_ACL`. All security levels except `NONE` require at least an application password from user to gain access to the Tuxedo application. At the `USER_PW` level and above there is an additional user password to authenticate the user and establish user credentials. In total there are potentially two passwords that need to be configured.

All SCA clients require this password information in order to gain access to Tuxedo application servers. There are two ways for an SCA client to retrieve password information:

- The client application may provide password information to ATMI/JATMI reference binding extensions through a callback mechanism.
- The client application may configure the identification of the password to be retrieved by the ATMI/JATMI reference binding extensions in the appropriate composite file.

Note: For more information, see Password callback methods in the SALT Programmer's Guide.

In order for the SALT administrator to configure password retrieval, the administrator must:

- Maintain the `password.store` file and set this file up correctly for the client application. The administrator must duplicate the `password.store` file across different machines if necessary.

- Add or delete password and identification pairs when necessary.
- Configure the client application composite file with correct information.

[Listing 2-39](#) contains an SCA ATMI client application example.

Listing 2-39 \$APPDIR/password.store \$APPDIR/simple.app.composite

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

<composite xmlns="http://www.oxa.org/xmlns/sca/1.0"
  name="simple.app">

  <component name="simpapp.TuxClient">
    <implementation.composite name="simpapp.client"/>
    <reference name="TOUPPER">tuxToupper</reference>
  </component>

</composite>
```

Listing 2-40 \$APPDIR/simpapp.client/simpapp.client.composite

```
<composite xmlns="http://www.oxa.org/xmlns/sca/1.0"
  name="simpapp.client">

  <reference name="TOUPPER">
    <interface.cpp header="ToupperTuxService.h"/>
    <binding.atmi requires="legacy">
      <tuxconfig>d:\tests\tuxedo\sca\tests\TUXCONFIG</tuxconfig>
      <inputBufferType target="charToup">STRING</inputBufferType>
      <outputBufferType target="charToup">STRING</outputBufferType>
      <authentication
        <passwordIdentifier>aaa</passwordIdentifier>
      </authentication>
    </binding.atmi>
  </reference>

</composite>
```

The above composite defines a Tuxedo application domain password identification "aaa" which will cause the ATMI reference binding to retrieve the password with identification "aaa" from the `password.store` file at the runtime. If you increased Tuxedo application domain security by requiring user authentication. (`SECURITY=USER_PW` or above) you would use the following command: `scapasswordtool -i crusoe -a`.

Then use a text editor or any other tool that can edit the `simpapp.client.composite` file and add the following entry in the `<binding.atmi/authentication>` element:

```
<userPasswordIdentifier>crusoe</userPasswordIdentifier>
```

Anyone using the password "crusoe" can access Tuxedo applications.

Tuxedo Link-Level Security

Tuxedo Link-Level Security has two variations. One is the easily configured Link-Level Encryption (LLE) and the other one is the more commonly used Transport Layer Security (TLS) also known as Secured Socket Layer (SSL). An SCA ATMI client using the native ATMI reference binding does not need link-level security configured at the SCA level since its transport method is native message queues and the Tuxedo BRIDGE.

The SCA JATMI client reference binding does not support link-level security. The only type of SCA client that allows configuration of link-level security is SCA Workstation ATMI client.

The SCA Workstation ATMI client contains a `<workStationParameters>` element configured in the composite file. The SCA runtime automatically loads the correct reference binding for this type of client.

Configuring Link-Level Encryption

Link-level encryption can be configured by adding an `<encryptBits>` element in the composite file. The following elements *should not* be configured for LLE, since they are specific to SSL encryption and imply that SSL encryption is used:

- `secPrincipalName`
- `secPrincipalLocation`
- `secPrincipalPassId`

The `<encryptBits>` element specifies the encryption strength that this client will attempt to negotiate. The syntax for the `<encryptBits>` element is `<minimum encryption`

strength>/<maximum encryption strength>. To configure minimum 56-bit encryption you must add the following to the composite file:

```
<networkAddress>//STRIATUM:8741</networkAddress>
<encryptBits>56/128</encryptBits>
```

Note: `encryptBits` specifies the encryption strength that the client connection attempts to negotiate. The format is `<minencryptbits>/<maxencprytbits>` (for example, 128/128). Values can be 0 (no encryption is used), 40, 56, 128, or 256. Invalid values result in a configuration exception.

This tells SCA Workstation Reference binding to require 56 to 128 bits encryption strength when negotiating with WSH. You must also add the following line to the *SERVERS section in the UBBCONFIG file:

```
WSL   SRVGRP=GROUP1 SRVID=1001 CLOPT="-A -- -n //STRIATUM:8741 -a -z 56 -Z
256
```

Configuring Transport Layer Security

In addition to `<encryptBits>`, to enable Link-Level Security over TLS/SSL you must configure `secPrincipalName`, `secPrincipalLocation`, and `secPrincipalPassId`.

- `secPrincipalName` - the name of the security principal. It is used for searching the client X.509 certification from the LDAP server.
- `secPrincipalLocation` - the client private key file.
- `secPrincipalPassId` - the password identifier that is used to retrieve client password used to encrypt the private key file.

These three parameters specify the parameters needed when a TLS/SSL connection needs to be established by a SCA Workstation ATMI client.

[Listing 2-41](#) contains the lines you must add to the client composite file in `/binding.atmi/workStationParameters` to configure TLS/SSL.

Listing 2-41 Client Composite File

```
<networkAddress>//STRITUM:8742</networkAddress>
<secPrincipalName>crusoe</secPrincipalName>
<secPrincipalLocation>/tux/udataobj/security/keys/crusoe.pem</secPrincipal
Location>
<secPrincipalPassId>crusoe</secPrincipalPassId>
```

In Tuxedo, you must add `-S 8742` to WSL to indicate that TLS/SSL is used if the client connects through port 8742.

```
WSL SRVGRP=GROUP1 SRVID=1001
    CLOPT="-A -- -n //STRIATUM:8741 -S 8742 -z 56 -Z 128"
```

Configuring Service Contract Discovery

When discovery is activated for a service, the server that provides the service collects service contract information and sends the information to an internal service implemented by [TMMETADATA \(5\)](#). The same service contract is only sent once to reduce communication overhead.

The TMMETADATA server summarizes the collected data and generates a service contract. The contract information can either be stored in the metadata repository, or output to a text file (which is then loaded to the metadata repository using `tmloadrepos`). Oracle SALT uses the `tmscd` command to control the service contract runtime collection. For more information, see [tmscd](#) in the *Oracle SALT Command Reference Guide*.

Generated service contract information contains service name, request buffer information, response buffer information, and error buffer information if there is a failure. The collected service contract information is discarded if it fails to send information to the TMMETADATA server. The buffer information includes buffer type and subtype, and field information for FML/FML32 (`name, type, subtype`).

Discovery is supported for any embedded buffer in FML32 buffer. For FML/FML32 field occurrences, the discovery automatically updates the pattern for the `count/requiredcount` in metadata repository. Field occurrence does not impact pattern, but the minimum occurrence is the "requiredcount". The maximum occurrence is the "count" of the entire contract discovery period.

For VIEW/VIEW32/X_C_TYPE/X_COMMON, only the view name is discovered. ORACLE SALT can generate view detail description by view name when using metadata repository.

Note: Patterns flagged with `autodiscovery` (see [Table 2-10](#)) are compared.

If the same autodiscovery pattern already exists in the metadata repository, then the newer pattern is ignored.

Only application ATMI services (local, or imported via /TDOMAIN gateway) are supported. Service contract discovery *does not* support the following services:

- system services (name starts with '.' or '..')
- conversational services
- CORBA services
- /Q and SALT proxy services

Note: Services without a reply are mapped as "oneway" services in the metadata repository.

tpforward Support

If a service issues `tpforward()` instead of `tpreturn()`, its reply buffer information will be same with the reply buffer of the service which it forwards to. For example,

- client calls SVCA with a STRING typed buffer
- SVCA processes the request, and then creates a new FML32 typed buffer as request forwarded to SVCB
- SVCB handles the request and returns a STRING buffer to the client. The SVCA contract is `STRING+STRING`. The SVCB contract is `FML32+STRING`

Service Contract Text File Output

If you want collected service contract discovery information logged to a file instead of directly to the metadata repository, you must use the `TMMETADATA(5) -o<filename>` option and then use `tmloadrepos` to manually load the file to the metadata repository. For more information, see `tmloadrepos` in the Oracle Tuxedo Reference Guide.

The output complies with the format imposed by `tmloadrepos` if a file is used as storage instead of metadata repository. The file contains a service header section and a parameter section for each parameter. Service header contains items listed in [Table 2-10](#). The "service" field format is `<TuxedoServiceName>+'_'+'<SequenceNo>`. The suffix `<SequenceNo>` is used to avoid name conflict when multiple patterns are recognized for a Tuxedo service.

Note: `<SequenceNo>` starts from the last `<SequenceNo>` number already stored in the metadata repository.

Service parameter contains information is listed in [Table 2-11](#).

Table 2-10 Service Level Attributes

Keyword (abbreviation)	Sample Value	Description
<code>service(sv)</code>	<code>TOUPPER_1</code>	<RealServiceName>_<SequenceNo>.
<code>tuxservice(tsv)</code>	<code>TOUPPER</code>	The service name.
<code>servicetype(st)</code>	<code>service oneway</code>	oneway if TPNOREPLY is set.
<code>inbuf(bt)</code>	<code>STRING</code>	FML, FML32, VIEW, VIEW32, STRING, CARRAY, XML, X_OCTET, X_COMMON, X_C_TYPE, MBSTRING or other arbitrary string representing an application defined custom buffer type.
<code>outbuf(BT)</code>	<code>FML32</code>	set to "NULL" if it's an error reply.
<code>errbuf(ebt)</code>	<code>STRING</code>	present only when it's an error reply.
<code>inview</code>		View name. Present only when inbuf is of type VIEW or VIEW32.
<code>outview</code>		View name. Present only when outbuf is of type VIEW or VIEW32.
<code>errview</code>		View name. Present only when errbuf is of type VIEW or VIEW32.
<code>autodiscovery</code>	<code>true</code>	Set to "true".

Table 2-11 Parameter Level Attributes

Keyword(abbreviation)	Sample	Description
param(pn)	USER_INFO	
paramdescription(pd)	service parameter	
access(pa)	in	A combination of {in}{out}{err}.
type(pt)	fml32	byte, short, integer, float, double, string, carray, dec_t, xml, ptr, fml32, view32, mbstring.
subtype(pst)		A view name for a view or view32 typed parameter.
count	100	The maximum occurrence of FML/FML32 field watched during the collection period
requiredcount	1	The minimum occurrence of FML/FML32 field watched during the collection period.

Examples

Example 1:

Input: service=SVC, request=STRING, reply = TPSUCCESS + STRING

Output Pattern: service=SVC_1,tuxservice=SVC,inbuf=STRING,outbuf=STRING

Example 2:

Input: service=SVC, request=STRING, reply = TPFAIL+ STRING

Output Pattern (partial): Service=SVC_1,
tuxservice=SVC,inbuf=STRING,outbuf=NULL,errbuf=STRING

Example 3:

Input:

service=SVC, request=STRING, reply = TPSUCCESS + STRING

```
service=SVC, request=STRING, reply = TPFAIL+ STRING
```

Output Pattern:

```
service=SVC_1,tuxservice=SVC,inbuf=STRING,outbuf=STRING
```

```
Service=SVC_2, tuxservice=SVC,inbuf=STRING,outbuf=NULL,errbuf=STRING
```

Example 4:

Input: `service=FMLS,request=FML32(name,pwd),reply=TPSUCCESS+FML32(id)`

Output Pattern:

```
service=FMLS_1,tuxservice=FMLS,inbuf=FML32,outbuf=FML32
```

```
param: input(name, pwd), output(id)
```

Example 5:

Input:

```
service=FMLS,request=FML32(name,pwd),reply=TPSUCCESS+FML32(id)
```

```
service=FMLS,request=FML32(name,pwd,token),reply=TPSUCCESS+FML32(id)
```

Output Pattern:

```
service=FMLS_1,tuxservice=FMLS,inbuf=FML32,outbuf=FML32
```

```
param: input(name, pwd), output(id)
```

```
service=FMLS_2,tuxservice=FMLS,inbuf=FML32,outbuf=FML32
```

```
param: input(name, pwd,token), output(id)
```

See Also

- [tmadmin](#)
- [tmloadrepos](#)
- [scaadmin](#)
- [buildscaclient](#)
- [buildscaserver](#)
- [ubbconfig](#)
- [WSDF documentation](#)
- [Oracle SALT Programming Guide](#)
- [Oracle SALT Reference Guide](#)

Administering Oracle SALT at Run Time

This section contains the following topics:

- [Administering Tuxedo Web Services](#)
- [Administering Tuxedo SCA Components](#)

Administering Tuxedo Web Services

Tuxedo Web service administration comprises the following:

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

Browsing to the WSDL Document from the GWWS Server

Each `GWWS` server automatically generates a WSDL document for each deployed inbound native WSDL. 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 WSDF name for the WSDL document. The specified native WSDF must be imported via inbound direction by the GWS server. If the option is not specified, the first inbound native WSDF is used.
mappolicy	{ pack raw mtom } Specifies the data mapping policies for certain Tuxedo Typed buffers for the generated WSDL document. Currently, this option impacts CARRY typed buffers only. If the option is not specified, pack is used as the default value.
toolkit	{ wls axis } Use this option only if you have previously defined mappolicy=raw. Specify the client toolkit used so that the proper WSDL document description for a CARRY typed buffer MIME attachment is generated. Oracle SALT supports WebLogic Server and Axis for SOAP with Attachments. The default value is wls.

Note: The WSDL download URL supported by Oracle SALT 2.0 is different from Oracle SALT 1.1. In Oracle SALT 1.1 release, one GWS 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 Oracle SALT 2.0 release, each WSDF file associated with the GWS 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 GWS server are deprecated in this release.

Table 3-2 Deprecated WSDL Download Options

Option	Value Description
--------	-------------------

Table 3-2 Deprecated WSDL Download Options

<code>SOAPversion</code>	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.
<code>encstyle</code>	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 Oracle SALT configuration file. The following topics list deployment considerations based on different scenarios. For more information, see [“Configuring the GWWS Servers” on page 2-20](#).

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`

Oracle 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`

Oracle 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-flood* 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 Tuxedo `TMTRACE` functionality (used to dynamically trace messages). All trace points are logged in the ULOG file. Checking the ULOG file trace

information helps to evaluate GWWS server SOAP message problems. GWWS server message tracing behavior is set using the `TMTRACE` environment variable, or by using the `tmadmin chtr` sub-command.

The reserved trace category, `msg`, is used to trace Oracle SALT messages. It can be used together with other general trace categories. For example, if trace category “`atmi+msg`” is specified, both Oracle SALT and Tuxedo ATMI trace messages are logged.

Notes: Message tracing is recommended for diagnostic treatment only.

The following trigger specifications are not recommended for GWWS servers:

```
abort, system, sleep
```

In any of these trigger specifications are used, GWWS servers may be unexpectedly terminated.

For more `tmtrace` and trace specification information, see [tmtrace\(5\)](#) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.

TMTRACE specification examples for Oracle SALT message tracing are shown below:

- To trace SALT messages only

```
export TMTRACE=msg:ulog:export GWWS_TRACE_LEVEL=100
```
- To trace both Oracle SALT and Tuxedo ATMI messages

```
export TMTRACE=atmi+msg:ulog:export GWWS_TRACE_LEVEL=100
```

[Listing 3-1](#) shows a ULOG file example containing Oracle SALT tracing messages.

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. [Listing 3-2](#) shows a ULOG file example containing WS-ReliableMessaging TMTRACE messages.

Listing 3-2 WS-ReliableMessaging TMTRACE 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-FFFFFFC0FFFFFFA8FFFFFFE010000-1811

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-FFFFFFC0FFFFFFA8FFFFFFE010000-1811

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-FFFFFFC0FFFFFFA8FFFFFFE010000-1811
```

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>	Displays statistics information of GWWS server
<code>configstats(cstat)</code>	Displays configuration information
<code>default(d)</code>	Specifies the default <code>-i</code> option
<code>echo(e)</code>	Switches echo input on/off
<code>paginate(page)</code>	Switches paging output on/off

Table 3-3 wsadmin Sub-Commands

<code>verbose (v)</code>	Switches verbose output on/off
<code>quit (q)</code>	Terminates the program

Troubleshooting Oracle SALT

The following sections explain how to troubleshoot a Oracle 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 Oracle 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` must also 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.

Administering Tuxedo SCA Components

Tuxedo SCA component administration comprises the following:

- [Tracing the SCA ATMI Server and Client](#)
- [Monitoring SCA ATMI Servers](#)
- [Tracing SCA JATMI Clients](#)

Tracing the SCA ATMI Server and Client

Both The SCA ATMI server and client can utilize the existing tracing capability provided by TUXEDO and SCA. The following sections describe how to use them in detail.

Tuxedo TMTRACE

SCA ATMI servers and clients support the Tuxedo `tmtrace(5)` function. All traces generated from TMTRACE are logged in the ULOG file. Checking the ULOG file trace information helps to determine the cause of a failure. The Tuxedo TMTRACE facility is enabled by setting TMTRACE environmental variable, or by using the `tmadmin chtr` sub-command.

Note: To trace Tuxedo ATMI messages enter: `export TMTRACE=atmi:ulog` at the command line.

SCA Runtime, ATMI Service, and Reference Binding Tracing

To trace the SCA runtime, ATMI service binding, and ATMI reference binding you must set `SCACPP_ULOG`. The valid value range of this environmental variable is 10 (limited tracing) to 90 (full tracing). By default, the output is directed to the console. To send information to Tuxedo ULOG, enter "`SCACPP_ULOG=yes`".

Note: This tracing facility is only available for Tuxedo server builds using `buildscaserver` and SCA client builds using `buildscaclient`.

[Listing 3-4](#) shows a ULOG example containing SCA runtime tracing:

Listing 3-4 SCA Runtime Tracing Information ULOG File

```
142059.STRIATUM!?proc.1108.3000.-2:      >>
osoa::sca::CompositeContext::getCurrent
142059.STRIATUM!?proc.1108.3000.-2:      >>
```

```

Tuscany::sca::SCARuntime::getCurrent Runtime
142059.STRIATUM!?proc.1108.3000.-2:    >>
tuscany::sca::util::ThreadLocal::getValu e
142059.STRIATUM!?proc.1108.3000.-2:    <<
tuscany::sca::util::ThreadLocal::getValu e
142059.STRIATUM!?proc.1108.3000.-2:    >>
tuscany::sca::SCARuntime::getShared Runtime
142059.STRIATUM!?proc.1108.3000.-2:    SCARuntime::getSharedRuntime()
142059.STRIATUM!?proc.1108.3000.-2:    >> tuscany::sca::util::Mutex::lock
142059.STRIATUM!?proc.1108.3000.-2:    << tuscany::sca::util::Mutex::lock
142059.STRIATUM!?proc.1108.3000.-2:    >>
tuscany::sca::util::Mutex::unlock
142059.STRIATUM!?proc.1108.3000.-2:    <<
tuscany::sca::util::Mutex::unlock
142059.STRIATUM!?proc.1108.3000.-2:    <<
tuscany::sca::SCARuntime::getSharedR untime
142059.STRIATUM!?proc.1108.3000.-2:    >>
tuscany::sca::util::ThreadLocal::Thread Local
142059.STRIATUM!?proc.1108.3000.-2:    <<
tuscany::sca::util::ThreadLocal::Thread Local
142059.STRIATUM!?proc.1108.3000.-2:    >>
tuscany::sca::SCARuntime::SCARuntime
142059.STRIATUM!?proc.1108.3000.-2:    SCA runtime install root
f:\tuxedo\tux101rp _wsc\udataobj\salt\sca
142059.STRIATUM!?proc.1108.3000.-2:    Default component:
testStringClientComp
142059.STRIATUM!?proc.1108.3000.-2:    >>
tuscany::sca::util::ThreadLocal::getValu e
142059.STRIATUM!?proc.1108.3000.-2:    <<
tuscany::sca::util::ThreadLocal::getValu e

```

Monitoring SCA ATMI Servers

A Tuxedo SCA server built with the `buildscaserver` utility can be monitored using the `scaadmin` utility. This utility shows service statistics information and helps perform maintenance through dynamic shared library loading and unloading.

To reload all components hosted by the `uBikeServer` Tuxedo server previously built using the `buildscaserver` command, do the following:

1. `prompt> scaadmin`
2. `prompt> reload -s uBikeServer`

Enter the following at the command line to display statistics on the services offered by the `uBikeServer` Tuxedo server ([Table 3-4](#) shows the results):

1. `prompt> scaadmin`
2. `prompt> pstats -s uBikeServer`

Table 3-4 pstats Output Service Statics

Service	Method	Status	Requests Processed
SEARCHINVENTORY	searchInventory	A	37

Before `scaadmin` is executed, you must set the `TUXCONFIG` environment variable. [Table 3-5](#) lists `scaadmin` sub-commands.

Table 3-5 scaadmin Sub-Commands

Sub-Command	Abbrev.	Description
<code>default</code>	<code>d</code>	Sets the corresponding argument to default, and it can be machine name, group name, server id, or server name. If the default command is entered without an argument, the current default values is printed.
<code>reload</code>	<code>r</code>	Dynamically reloads the SCA components hosted in a Tuxedo server.
<code>printstats</code>	<code>pstats</code>	Displays the list of services hosted by a Tuxedo server, and the associated method, number of queries, and status (active, idle)

Table 3-5 scaadmin Sub-Commands

Sub-Command	Abbrev.	Description
verbose	v	Produces output in verbose mode.
echo	e	Switches echo input on/off echo.
quit	q	Terminates the session.

Note: Both Windows and HP systems have a limitation using the "reload" sub-command.

When multiple servers share the same component library on Windows and HP systems, the shared component library cannot be reloaded. To reload a component library common to multiple servers, the "scaadmin" reload sub-command must be performed on all affected servers simultaneously.

Tracing SCA JATMI Clients

The TUXEDO SCA Java reference binding and data transformation support output to the console and to a log file. By default there will be at most 10 log files, the maximum size of each file is 100000 bytes, and are located in \$APPDIR with name `jatmi<number>.log` file. The log file names are cycled with the latest one using the number 0, and the one just before latest one uses 1 (for example, `jatmi0.log` is the latest log file, and `jatmi9.log` is the oldest log file). If the APPDIR environment variable is not set and `com.oracle.jatmi.APPDIR` java property is not specified, the log is placed in the current working directory.

By default, the log files are overwritten each time the application starts. Many logger parameters can be fine tuned. [Table 3-6](#) lists tunable Java properties related to logging.

Table 3-6 Logger Tuning Property Table

Function	Properties	Value Range	Default Value
Log File Location	<code>com.oracle.jatmi.APPDIR</code>	valid path name	APPDIR environmental variable, if APPDIR is not set uses current working directory
Log File Size	<code>com.oracle.jatmi.LogFileSize</code>	0 ... maximum file size supported by the system	100,000 bytes

Table 3-6 Logger Tuning Property Table

Function	Properties	Value Range	Default Value
Append File	com.oracle.jatmi.LogFileAppend	{true,false}	false
Number of Log Files	com.oracle.jatmi.LogFileCount	1... maximum number of files in a directory	10
Log Output	com.oracle.jatmi.LogDestination	{file,console,both}	both
Log Format	com.oracle.jatmi.LogFileFormat	{xml,plain}	plain

To have the TUXEDO SCA Java reference binding log in a different language, first check the supported languages that are installed. The default is English. To switch to a different language, add: "-Duser.language=<your preferred language>" to your Java command line when starting the TUXEDO SCA Java client. For example:

```
java -classpath ./apps/classes:$CLASSPATH -Duser.language=ES
-Dcom.oracle.jatmi.LogDestination=console myApplication.
```

This generates an English log in plain text format to the console only.

Table 3-6 shows an example of the log file contents.

Listing 3-5 Log File Contents

```
9/3/08:3:19:14 PM:10:TRACE[TuxedoConversion,processSendBuf]< (10) return
lst args
9/3/08:3:19:14 PM:10:DBG[AtmiBindingInvoker,invoke]ServiceType:
requestresponse
9/3/08:3:19:14 PM:10:DBG[AtmiBindingInvoker,invoke]Return Type Class:
simpapp.View7Rep
9/3/08:3:19:14 PM:10:DBG[AtmiBindingInvoker,invoke]target service name:
RULE7
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]TPURCODE: 0
9/3/08:3:19:15 PM:10:TRACE[TuxedoConversion,processReplyBuffer]> (reply
simpapp.View7Rep@191777e:0:null)
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processReplyBuffer]returnType:
```

```

simpapp.View7Rep
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processReplyBuffer]Reply Buffer
Class: simpapp.View7Rep
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processReplyBuffer]Reply Buffer
Type: X_COMMON
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processReplyBuffer]Reply Buffer
Subtype: View7Rep
9/3/08:3:19:15 PM:10:TRACE[TuxedoConversion,processReplyBuffer]< (30)
return buffer directly
9/3/08:3:19:15 PM:10:DBG[Accessors,getConventionProperty]Convention
Property: CONVENTIONS_TUX
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]networkAddress: host =
STRIATUM, port = 8080
9/3/08:3:19:15
PM:10:TRACE[AtmiBindingInvoker,determineServiceCallParameters]> ( )
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,isLegacy]> ( )
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,isLegacy]< (10) return true
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,isMap]> ( )
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,isMap]< (10) return false
9/3/08:3:19:15
PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]Operation name
= rule7_OVVO
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getServiceType]> (rule7_OVVO)
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getServiceType]< (10) return
null
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getInputBufferType]>
(rule7_OVVO)
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getInputBufferType]< (10)
return null
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getOutputBufferType]>
(rule7_OVVO)
9/3/08:3:19:15 PM:10:TRACE[AtmiBindingImpl,getOutputBufferType]< (10)
return null
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,getErrorBufferType]> (rule7_OVVO)
9/3/08:3:19:15 PM:10:DBG[AtmiBindingImpl,getErrorBufferType]< (10) return
null
9/3/08:3:19:15

```

```

PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]svcName =
RULE7
9/3/08:3:19:15
PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]svcType =
requestresponse
9/3/08:3:19:15
PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]inbuf =
X_COMMON
9/3/08:3:19:15
PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]outbuf =
X_COMMON
9/3/08:3:19:15
PM:10:DBG[AtmiBindingInvoker,determineServiceCallParameters]errbuf = null
9/3/08:3:19:15
PM:10:TRACE[AtmiBindingInvoker,determineServiceCallParameters]< (10)
return
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]Input Buffer Type:
X_COMMON
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]Output Buffer Type:
X_COMMON
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]Error Buffer Type: null
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]inBufType:X_COMMON,
count: 1
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]outBufType:X_COMMON,
count: 1
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]View Classes:
simpapp.View7Req,simpapp.View7Rep
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,getclassList]getClassList:
Getting class for simpapp.View7Req
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,getclassList]getClassList:
Getting class for simpapp.View7Rep
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,setFieldClasses]setFldClasses:
null
9/3/08:3:19:15 PM:10:DBG[AtmiBindingInvoker,invoke]Passing thro invoker...
9/3/08:3:19:15 PM:10:TRACE[TuxedoConversion,processSendBuf]> (args
[Ljava.lang.Object;@ab1b4)
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processSendBuf]args[0] class

```

```

simpapp.Rule7Req
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,needConversion]buftype: X_COMMON
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processSendBuf]Argument Class
Name: simpapp.Rule7Req
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processSendBuf]Input Buffer Id
: XCOMMON
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processSendBuf]Type code      :
10
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,processSendBuf]InputBufferType:
XCOMMON
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,getclassList]getClassList:
Getting class for simpapp.View7Req
9/3/08:3:19:15 PM:10:DBG[TuxedoConversion,getclassList]getClassList:
Getting class for simpapp.View7Rep
9/3/08:3:19:15 PM:10:TRACE[Accessors,determineConvention]>
(simpapp.Rule7Req)
9/3/08:3:19:15 PM:10:DBG[Accessors,determineConvention]Method name: getId
9/3/08:3:19:15 PM:10:DBG[Accessors,determineConvention]Method name: setCmd
9/3/08:3:19:15 PM:10:DBG[Accessors,determineConvention]Method name: setId
9/3/08:3:19:15 PM:10:DBG[Accessors,determineConvention]Method name: getCmd
9/3/08:3:19:15 PM:10:TRACE[Accessors,determineConvention]< (30) return BEAN

```

See Also

- [buildscaclient](#)
- [buildscacomponent](#)
- [buildscaserver](#)
- [scaadmin](#)
- [tmadmin](#)
- [tmtrace](#)
- [Oracle SALT Programming Guide](#)
- [Oracle SALT Reference Guide](#)

Appendix A:

Oracle Tuxedo SCA Schemas

This section contains the following information:

- [ATMI and JTMJ Binding Schema For C/C++](#)
- [Web Service Binding Schema](#)

ATMI and JTMJ Binding Schema For C/C++

Listing A-1 ATMI and JTMJ Binding Schema For C/C++

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

<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        elementFormDefault="qualified">

    <element name="binding.atmi" type="sca:AtmiBinding"
            substitutionGroup="sca:binding"/>
```

Appendix A: Oracle Tuxedo SCA Schemas

```
<complexType name="AtmiBinding">
  <complexContent>
    <extension base="sca:Binding">
      <sequence>
        <element name="tuxconfig" type="string"
          minOccurs="0" />
        <element name="map" type="sca:TargetMapType" minOccurs="0"
          maxOccurs="unbounded" />
        <element name="serviceType" type="sca:SvcType"
          minOccurs="0" maxOccurs="unbounded" />
        <element name="inputBufferType" type="sca:BufferType"
          minOccurs="0" maxOccurs="unbounded" />
        <element name="outputBufferType" type="sca:BufferType"
          minOccurs="0" maxOccurs="unbounded" />
        <element name="errorBufferType" type="sca:BufferType"
          minOccurs="0" maxOccurs="unbounded" />
        <element name="workStationParameters"
          type="sca:WorkStationParameters"
          minOccurs="0" />
        <element name="authentication" type="sca:Authentication"
          minOccurs="0" />
        <element name="fieldTablesLocation" type="string"
          minOccurs="0" />
        <element name="fieldTables" type="string"
          minOccurs="0" />
        <element name="fieldTablesLocation32" type="string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```



```

        minOccurs="0"/>
<element name="fieldTables32" type="string"
        minOccurs="0"/>
<element name="viewFilesLocation" type="string"
        minOccurs="0"/>
<element name="viewFiles" type="string" minOccurs="0"/>
<element name="viewFilesLocation32" type="string"
        minOccurs="0"/>
<element name="viewFiles32" type="string"
        minOccurs="0"/>
<element name="remoteAccess" type="sca:RemoteAccess"
        minOccurs="0"/>
<element name="transaction" type="sca:TransactionType"
        minOccurs="0"/>
</sequence>
<anyAttribute namespace="##any" processContents="lax" />
</extension>
</complexContent>
</complexType>

<complexType name="TargetMapType">
    <simpleContent>
        <extension base="TargetSimple">
            <attribute name="target" type="string" use="optional"/>
        </extension>
    </simpleContent>
</complexType>

```

Appendix A: Oracle Tuxedo SCA Schemas

```
<simpleType name="TargetSimple">
  <restriction base="string" />
</simpleType>

<complexType name="SvcType">
  <simpleContent>
    <extension base="SvcTypeEnum">
      <attribute name="target" type="string" use="optional" />
    </extension>
  </simpleContent>
</complexType>

<simpleType name="SvcTypeEnum">
  <restriction base="string">
    <enumeration value="oneway" />
    <enumeration value="requestresponse" />
  </restriction>
</simpleType>

<complexType name="BufferType">
  <simpleContent>
    <extension base="BufferTypeEnum">
      <attribute name="target" type="string" use="optional" />
    </extension>
  </simpleContent>
</complexType>
```

```

<simpleType name="BufferTypeEnum">
  <restriction base="string">
    <enumeration value="string"/>
    <enumeration value="carray"/>
    <enumeration value="x_octet"/>
    <enumeration value="view"/>
    <enumeration value="x_c_type"/>
    <enumeration value="x_common"/>
    <enumeration value="view32"/>
    <enumeration value="xml"/>
    <enumeration value="fml"/>
    <enumeration value="fml32"/>
    <enumeration value="mbstring"/>
  </restriction>
</simpleType>

<complexType name="WorkStationParameters">
  <sequence>
    <element name="networkAddress" type="string" minOccurs="0"/>
    <element name="secPrincipalName" type="string" minOccurs="0"/>
    <element name="secPrincipalLocation" type="string"
      minOccurs="0"/>
    <element name="secPrincipalPassId" type="string"
      minOccurs="0"/>
    <element name="encryptbits" type="string" minOccurs="0"/>
  </sequence>

```

Appendix A: Oracle Tuxedo SCA Schemas

```
</complexType>

<complexType name="Authentication">
  <sequence>
    <element name="userName" type="string" minOccurs="0"/>
    <element name="clientName" type="string" minOccurs="0"/>
    <element name="groupName" type="string" minOccurs="0"/>
    <element name="passwordIdentifier" type="string"
      minOccurs="0"/>
    <element name="userPasswordIdentifier" type="string"
      minOccurs="0"/>
  </sequence>
</complexType>

<complexType name="RemoteAccess">
  <restriction base="string">
    <enumeration value="native"/>
    <enumeration value="workstation"/>
  </restriction>
</complexType>

<complexType name="TransactionType">
  <attribute name="timeout" type="int" use="optional"/>
</complexType>
</schema>
```

Web Service Binding Schema

Listing A-2 Web Service Binding Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Licensed to the Apache Software Foundation (ASF) under one
or more contributor license agreements. See the NOTICE file
distributed with this work for additional information
regarding copyright ownership. The ASF licenses this file
to you under the Apache License, Version 2.0 (the
"License"); you may not use this file except in compliance
with the License. You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing,
software distributed under the License is distributed on an
"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
KIND, either express or implied. See the License for the
specific language governing permissions and limitations
under the License.
-->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
  elementFormDefault="qualified">
```

Appendix A: Oracle Tuxedo SCA Schemas

```
<element name="binding.ws" type="sca:WebServiceBinding"
substitutionGroup="sca:binding"/>

<complexType name="WebServiceBinding">
  <complexContent>
    <extension base="sca:Binding">
      <sequence>
        <element name="soapbinding" type="sca:SOAPBinding"
minOccurs="0" maxOccurs="unbounded"/>
        <any namespace="##other" processContents="lax"
minOccurs="0" maxOccurs="unbounded" />
      </sequence>
      <attribute name="endpoint" type="anyURI" use="optional" />
      <attribute name="location" type="anyURI" use="optional" />
      <attribute name="conformanceURIs"
type="sca:ConformanceURIList" use="optional" />
      <attribute name="interfaceMapping" type="string"
use="optional" />
      <anyAttribute namespace="##any" processContents="lax" />
    </extension>
  </complexContent>
</complexType>

<complexType name="SOAPBinding">
  <sequence>
    <any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded" />
  </sequence>
  <attribute name="name" type="NCName" use="optional" />

```

```
<attribute name="version" type="string" use="optional" />
    <anyAttribute namespace="##any" processContents="lax" />
</complexType>

<simpleType name="ConformanceURIList">
    <list itemType="anyURI"/>
</simpleType>
</schema>
```

See Also

- [Oracle SALT Administration Guide](#)
- [Oracle SALT Programming Guide](#)

Appendix A: Oracle Tuxedo SCA Schemas

See Also

Appendix A: Oracle Tuxedo SCA Schemas

See Also