Sun OpenSSO Enterprise 8.0 Developer's Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–3748
January 2009

Contents

Preface

Sun OpenSSO Enterprise 8.0 provides a comprehensive solution for protecting network resources that integrates authentication and authorization services, policy agents, and identity federation. This Preface to the *OpenSSO Enterprise 8.0 Developer's Guide* provides information about using the OpenSSO Enterprise Java application programming interfaces (API) and service provider interfaces (SPI).

Note – For information about using the C API see *Sun OpenSSO Enterprise 8.0 C API Reference for Application and Web Policy Agent Developers*. Additional information on the Java interfaces can be found in the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

Before You Read This Book

This book is intended for use by IT administrators and software developers who implement a web access platform using Sun servers and software. Readers of this guide should be familiar with the following technologies:

- eXtensible Markup Language (XML)
- Lightweight Directory Access Protocol (LDAP)
- Java™
- JavaServer Pages™ (JSP)
- HyperText Transfer Protocol (HTTP)
- HyperText Markup Language (HTML)

Related Documentation

Related documentation is available as follows:

- "OpenSSO Enterprise 8.0 Core Documentation" on page 12
- "Related Product Documentation" on page 13

OpenSSO Enterprise 8.0 Core Documentation

The OpenSSO Enterprise 8.0 core documentation set contains the following titles:

- The *Sun OpenSSO Enterprise 8.0 Release Notes* will be available online after the product is released. It gathers an assortment of last-minute information, including a description of what is new in this current release, known problems and limitations, installation notes, and how to report issues with the software or the documentation.
- The *Sun OpenSSO Enterprise 8.0 Technical Overview* provides high level explanations of how OpenSSO Enterprise components work together to protect enterprise assets and web-based applications. It also explains basic concepts and terminology.
- The *Sun OpenSSO Enterprise 8.0 Deployment Planning Guide* provides planning and deployment solutions for OpenSSO Enterprise based on the solution life cycle
- The *Deployment Example: Single Sign-On, Load Balancing and Failover Using Sun OpenSSO Enterprise 8.0* provides instructions for building an OpenSSO solution incorporating authentication, authorization and access control. Procedures for load balancing and session failover are also included.
- The *Deployment Example: SAML v2 Using Sun OpenSSO Enterprise 8.0* provides instructions for building an OpenSSO solution incorporating SAML v2 federation. Installation and configuration procedures are included.
- The *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide* provides information for installing and configuring OpenSSO Enterprise.
- The *Sun OpenSSO Enterprise 8.0 Performance Tuning Guide* provides information on how to tune OpenSSO Enterprise and its related components for optimal performance.
- The *Sun OpenSSO Enterprise 8.0 Administration Guide* describes administrative tasks such as *how to create a realm* and *how to configure a policy*. Most of the tasks described can be performed using the administration console as well as the `ssoadm` command line utilities.
- The *Sun OpenSSO Enterprise 8.0 Administration Reference* is a guide containing information about the command line interfaces, configuration attributes, internal files, and error codes. This information is specifically formatted for easy searching.
- The *Sun OpenSSO Enterprise 8.0 Developer's Guide* (this guide) offers information on how to customize OpenSSO Enterprise and integrate its functionality into an organization's current technical infrastructure. It also contains details about the programmatic aspects of the product and its API.

- The *Sun OpenSSO Enterprise 8.0 C API Reference for Application and Web Policy Agent Developers* provides summaries of data types, structures, and functions that make up the public OpenSSO Enterprise C SDK for application and web agent development.
- The *Sun OpenSSO Enterprise 8.0 Java API Reference* provides information about the implementation of Java packages in OpenSSO Enterprise.
- The *Sun OpenSSO Enterprise Policy Agent 3.0 User's Guide for Web Agents* and *Sun OpenSSO Enterprise Policy Agent 3.0 User's Guide for J2EE Agents* provide an overview of the policy functionality and policy agents available for OpenSSO Enterprise.

Updates to the *Release Notes* and links to modifications of the core documentation can be found on the OpenSSO Enterprise page at docs.sun.com. Updated documents will be marked with a revision date.

Related Product Documentation

The following table provides links to documentation for related products.

Product	Link
Sun Java System Directory Server 6.3	`http://docs.sun.com/coll/1224.4`
Sun Java System Web Server 7.0 Update 3	`http://docs.sun.com/coll/1653.3`
Sun Java System Application Server 9.1	`http://docs.sun.com/coll/1343.4`
Sun Java System Message Queue 4.1	`http://docs.sun.com/coll/1307.3`
Sun Java System Web Proxy Server 4.0.6	`http://docs.sun.com/coll/1311.6`
Sun Java System Identity Manager 8.0	`http://docs.sun.com/coll/1514.5`

Searching Sun Product Documentation

Besides searching Sun product documentation from the docs.sun.com^{SM} web site, you can use a search engine by typing the following syntax in the search field:

search-term `site:docs.sun.com`

For example, to search for “broker,” type the following:

```
broker site:docs.sun.com
```

To include other Sun web sites in your search (for example, java.sun.com, www.sun.com, and developers.sun.com), use `sun.com` in place of `docs.sun.com` in the search field.

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. To share your comments, go to `http://docs.sun.com` and click Send Comments. In the online form, provide the full document title and part number. The part number is a 7-digit or 9-digit number that can be found on the book's title page or in the document's URL. For example, the title of this book is *Sun OpenSSO Enterprise 8.0 Technical Overview*, and the part number is 820–3740.

Typographical Conventions

The following table describes the typographic conventions that are used in this deployment example.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`**
		`Password:`

aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Default Paths and Directory Names

The OpenSSO Enterprise documentation uses the following terms to represent default paths and directory names:

TABLE P–2 Default Paths and Directory Names

Term	Description
zip-root	Represents the directory where the `opensso.zip` file is decompressed.

TABLE P–2 Default Paths and Directory Names (Continued)

Term	Description
OpenSSO-Deploy-base	Represents the directory where the web container deploys `opensso.war`. The location varies depending on the web container used. To determine the value of *OpenSSO-Deploy-base*, view the file in the `.openssocfg` directory (located in the home directory of the user who deployed `opensso.war`). For example, consider this scenario with Application Server 9.1 as the web container: ■ Application Server 9.1 is installed in the default directory: `/opt/SUNWappserver`. ■ The `opensso.war` file is deployed by super user (`root`) on Application Server 9.1. The `.openssocfg` directory is in the `root` home directory (`/`), and the file name in `.openssocfg` is `AMConfig_opt_SUNWappserver_domains_domain1_applications_j2ee-modules_opensso_`. Thus, the value for *OpenSSO-Deploy-base* is: `/opt/SUNWappserver/domains/domain1/applications/j2ee-modules/opensso`
ConfigurationDirectory	Represents the name of the directory specified during the initial configuration of OpenSSO Enterprise. The default is `opensso` in the home directory of the user running the Configurator. Thus, if the Configurator is run by `root`, *ConfigurationDirectory* is `/opensso`.

CHAPTER 1

Using the Authentication Service API and SPI

This chapter provides information on the application programming interface (API) and service provider interface (SPI) developed for Sun OpenSSO Enterprise Authentication Service. It contains the following sections:

For information on Authentication Service C APIs, see *Sun OpenSSO Enterprise 8.0 C API Reference for Application and Web Policy Agent Developers*.

For a comprehensive listing of Authentication Service APIs and SPIs, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

Using the Authentication Service API to Initiate Authentication

The `com.sun.identity.authentication` package provides interfaces and classes that can be used by a Java application to access the OpenSSO Enterprise Authentication Service. Through this access the application, running either locally or remotely to OpenSSO Enterprise, can initiate an authentication process, submit required credentials and retrieve the single sign-on (SSO) session token for an application or a user. When implemented, the authentication API starts the authentication process, and the Authentication Service responds with a set of requirements such as user ID and password. The appropriate credentials are returned to the Authentication Service. This back and forth communication between the custom application (with implemented API) and the Authentication Service continues until all requirements have been met. At that point, the client makes one final call to determine if authentication has been successful or has failed.

The first step in the code sequence for the authentication process is to instantiate the `com.sun.identity.authentication.AuthContext` class which will create a new `AuthContext` object for each authentication request. Since OpenSSO Enterprise can handle multiple realms, `AuthContext` should be initialized, at the least, with the name of the realm to which the requestor is authenticating. Once an `AuthContext` object has been created, the `login()` method is called indicating to the server what method of authentication is desired. The `getRquirements` method returns an array of `Callback` objects that correspond to the credentials the user must pass to the Authentication Service. These objects are requested by the authentication plug-ins, and are usually displayed to the user as login requirement screens. For example, if the requested user is authenticating to an organization configured for LDAP authentication only, the server will respond with the LDAP login requirement screen to supply a user name and a password. The code must then loop by calling the `hasMoreRequirements()` method until the required credentials have been entered. Once entered, the credentials are submitted back to the server with the `submitRequirements()` method. The final step is to make a `getStatus()` method call to determine if the authentication was successful. If successful, the caller obtains a session token for the user; if not, a `LoginException` is thrown.

The following code sample illustrates how to authenticate users with user name and password credentials and obtain the session token using `getSSOToken()`.

EXAMPLE 1–1 Authentication Code Sample

```
import com.iplanet.sso.SSOToken;
import com.sun.identity.authentication.AuthContext;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginException;

public class TokenUtils {
    public static SSOToken getSessionToken(String realmName, String userId,
        String password) throws Exception {
        AuthContext ac = null;
        try {
            if (realmName == null || realmName.length() == 0) {
                realmName = "/";
            }
            ac = new AuthContext(realmName);
            ac.login();
        } catch (LoginException le) {
            le.printStackTrace();
            return null;
        }

        try {
```

EXAMPLE 1–1 Authentication Code Sample *(Continued)*

```
            Callback[] callbacks = null;
            // Get the information requested by the plug-ins
            if (ac.hasMoreRequirements()) {
                callbacks = ac.getRequirements();

                if (callbacks != null) {
                    addLoginCallbackMessage(callbacks, userId, password);
                    ac.submitRequirements(callbacks);

                    if (ac.getStatus() == AuthContext.Status.SUCCESS) {
                        System.out.println("Auth success");
                    } else if (ac.getStatus() == AuthContext.Status.FAILED) {
                        System.out.println("Authentication has FAILED");
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
        return ac.getSSOToken();
    }

    static void addLoginCallbackMessage(Callback[] callbacks, String userId,
        String password)
         throws UnsupportedCallbackException
    {
        int i = 0;
        try {
            for (i = 0; i < callbacks.length; i++) {
                if (callbacks[i] instanceof NameCallback) {
                    NameCallback nc = (NameCallback) callbacks[i];
                    nc.setName(userId);
                } else if (callbacks[i] instanceof PasswordCallback) {
                    PasswordCallback pc = (PasswordCallback) callbacks[i];
                    pc.setPassword(password.toCharArray());
                }
            }
        } catch (Exception e) {
             throw new UnsupportedCallbackException(callbacks[i],
                   "Callback exception: " + e);
        }
    }
}
```

Because the Authentication Service is built on the Java Authentication and Authorization Service (JAAS) framework, the Authentication Service API can invoke any authentication modules written with the JAAS API as well as those built specifically for OpenSSO Enterprise.

Using the Authentication Service SPI to Write Authentication Modules

OpenSSO Enterprise provides the `com.sun.identity.authentication.spi` Java package to write Java-based authentication modules and plug them into the Authentication Service framework, allowing proprietary authentication providers to be managed using the administration console. The authentication module is created using the `com.sun.identity.authentication.spi.AMLoginModule` class which implements the Java Authentication and Authorization Service (JAAS) `LoginModule` class.

The JAAS API enables services to authenticate and enforce access controls upon users. It implements a Java version of the standard Pluggable Authentication Module (PAM) framework. Because of this architecture, any custom JAAS authentication module will work with the Authentication Service. For more information on the JAAS API, see the Java Authentication And Authorization Service Reference Guide and `http://java.sun.com/products/jaas/`

The `com.sun.identity.authentication.spi.AMLoginModule` interface provides methods to access the Authentication Service and the authentication module's configuration properties files. This class takes advantage of many built-in features of OpenSSO Enterprise and scales well. Once created, the custom authentication module can be added to the list of authentication modules displayed by the OpenSSO Enterprise console.

▼ To Create an Authentication Module

1. **Create an Authentication Module configuration properties file.**

2. **(Optional) Register the service.**

   Create a `module.xml` and a `service.xml` file for the authentication module.

3. **Write the principal class.**

4. **Create the authentication module.**

5. **Add post processing peatures.**

More Information

Sample Login Module

For a sample Login Module, log into the OpenSSO Enterprise administration console. Then access the following URL: `http://`*openSSO-host*`:3080/opensso/samples/authentication/AuthSampleLoginModule.html`

Creating an Authentication Module Configuration Properties File

The authentication module's configuration properties file is an XML file that defines the module's authentication requirements and login state information. The parameters in this file automatically and dynamically customize the authentication module's user interface, providing the means to initiate, construct and send the credential requests, in the form of login pages, to the Distributed Authentication User Interface. `Auth_Module_Properties.dtd` defines the data structure of the file.

Name the authentication module's configuration properties file using the same name as that of the authentication module's class (no package information) and use the extension `.xml`. Use this naming convention even if no states are required.

When an authentication process is invoked, values nested in the `Callbacks` element of the module's configuration properties file are used to generate login screens. The module controls the login process, and determines each concurring screen. The following configuration properties file for the LDAP authentication module illustrates this concept.

EXAMPLE 1–2 LDAP Authentication Module Configuration Properties File

```
<ModuleProperties moduleName="LDAP" version="1.0" >

    <Callbacks length="2" order="1" timeout="120"
     header="This server uses LDAP Authentication" >
        <NameCallback>
            <Prompt> User Name: </Prompt>
        </NameCallback>
        <PasswordCallback echoPassword="false" >
            <Prompt> Password: </Prompt>
        </PasswordCallback>
    </Callbacks>

    <Callbacks length="4" order="2" timeout="120"
     header="Change Password&lt;BR&gt;&lt;/BR&gt;#REPLACE#&lt;BR&gt;&lt;/BR&gt;" >
        <PasswordCallback echoPassword="false" >
            <Prompt>Old Password </Prompt>
        </PasswordCallback>
        <PasswordCallback echoPassword="false" >
```

EXAMPLE 1–2 LDAP Authentication Module Configuration Properties File *(Continued)*

```
            <Prompt> New Password </Prompt>
        </PasswordCallback>
        <PasswordCallback echoPassword="false" >
            <Prompt> Confirm Password </Prompt>
        </PasswordCallback>
        <ConfirmationCallback>
            <OptionValues>
                <OptionValue>
                    <Value> Submit </Value>
                </OptionValue>
                <OptionValue>
                    <Value> Cancel </Value>
                </OptionValue>
            </OptionValues>
        </ConfirmationCallback>
    </Callbacks>

    <Callbacks length="0" order="3" timeout="120"
     header=" Your password has expired. Please contact service desk to
     reset your password" error="true" />

    <Callbacks length="0" order="4" timeout="120" template="user_inactive.jsp"
     error="true"/>

</ModuleProperties>
```

The initial interface has two `Callback` elements corresponding to requests for the user identifier and password. When the user enters values, the following events occur:

- The values are sent to the module.
- The `process()` routine validates the values.

  If the module writer throws a `LoginException`, an Authentication Failed page will be sent to the user. If no exception is thrown, the user is redirected to his or her default page.
- If the user's password is expiring, the module writer sets the next page state to 2.

  Page state 2 requires the user to change a password. The `process()` routine is again called after the user submits the appropriate values.

Writing the Principal Class

After creating the authentication module's configuration properties file, write a class which implements `java.security.Principal` to represent the entity requesting authentication. For example, the constructor takes the username as an argument. If authentication is successful, the module will return this principal to the Authentication Service which populates the login state and session token with the information representing the user.

Creating the Authentication Module

Custom authentication modules extend the `com.sun.identity.authentication.spi.AMLoginModule` class and must implement the `init()`, `process()` and `getPrincipal()` methods. Other methods that can be implemented include `setLoginFailureURL()` and `setLoginSuccessURL()` which define URLs to which the user is sent based on a failed or successful authentication, respectively. To make use of the account locking feature with custom authentication modules, the `InvalidPasswordException` exception should be thrown when the password is invalid.

If the custom authentication module requires or already uses a service configuration XML file, the file should contain attribute schema for one of the following attributes: `iplanet-am-auth-`*authModuleName*`-auth-level` or `lsunAMAuth`*authModuleName*`AuthLevel`. The module Java file should invoke the `setAuthLevel()` method in the `init()` method implementation.

Three main methods exist for creating a custom authentication module:

- init()
- process()
- getPrincipal()

Implementing the `init()` Method

`init()` is an abstract method that initializes the module with relevant information. This method is called by `AMLoginModule` prior to any other method calls. The method implementation should store the provided arguments for future use. It may peruse the `sharedState` to determine what information it was provided by other modules, and may also traverse through the `options` to determine the configuration parameters that will affect the module's behavior. The data can be ignored if the module being developed does not understand it.

Implementing the `process()` Method

`process()` is called to perform the actual authentication. For example, it may prompt for a user name and password, and then attempt to verify the credentials. If your module requires user interaction (for example, retrieving a user name and password), it should not do so directly. This method should invoke the `handle` method of the `javax.security.auth.callback.CallbackHandler` interface to retrieve and display the appropriate callbacks. The `AMLoginModule` then internally passes the callback values to the Distributed Authentication User Interface which performs the requested authentication.

Consider the following points while writing the `process()` method:

- Perform the authentication and if successful, save the authenticated principal.
- Return `-1` if authentication succeeds.
- Throw an exception, such as `AuthLoginException`, if authentication fails or return the relevant state specified in the module's configuration properties file
- Throw an exception, such as `InvalidPasswordException`, if using the Login Failure Lockout feature
- If multiple states are available to the user, the Callback array from a previous state may be retrieved by using the `getCallback()` method. The underlying login module keeps callback information from previous states until the login process is completed.
- If a module needs to substitute dynamic text (generate challenges, passwords or user identifiers) in the next state, use the `getCallback()` method to retrieve the callback for the next state, modify the text, and call `replaceCallback()` to update the array.
- Each authentication session will create a new instance of your module's Java class. The reference to the class will be released once the authentication session has either succeeded or failed.
- Any static data or reference to any static data in your module must be thread-safe.

Implementing the `getPrincipal()` Method

`getPrincipal()` should be called once at the end of a successful authentication session. This method retrieves the authenticated token string which will refer to the authenticated user in the OpenSSO Enterprise environment. A login session is deemed successful when all pages in the module's configuration properties file have been sent and the module has not thrown an exception.

Adding Post Processing Features

The `com.sun.identity.authentication.spi.AMPostAuthProcessInterface` interface can be implemented for post processing tasks on authentication success, failure and logout using the methods `onLoginSuccess()`, `onLoginFailure()`, and `onLogout()`, respectively. The Authentication Post Processing Classes are defined in the Core Authentication Service and configurable at several levels such as at the realm or role levels. Post processing tasks might include:

- Adding attributes to a user’s session token after successful authentication.
- Sending notification to an administrator after failed authentication.
- General clean up such as clearing cookies after logout, or logging out of other system components.

Using the Sample Client Login Module

OpenSSO Enterprise provides a sample login module to illustrate how to write a client login module. The source `Login.java` is located in the following directory:

OpenSSO-base`/opensso/samples/sdk/source/com/sun/identity/samples/authentication`

▼ To Run the Sample Client Login Module

1. **Download and unzip the `opensso.zip` file.**

```
# unzip opensso.zip
Archive:  opensso.zip
   creating: opensso/
        ...
```

2. **Unzip the Client SDK zip file.**

```
# cd opensso/samples
# unzip opensso-client.zip
```

3. **Set up the Client SDK.**

```
# pwd
/export/home/root/programs/opensso/opensso/samples
# cd sdk
# .Kchmod u+x scripts/*.sh
$ ./scripts/setup.sh
Debug directory (make sure this directory exists): /opensso/client-debugs
Application user (e.g. URLAccessAgent) password: agentadmin
Protocol of the server: http
Host name of the server: host1.example.com
Port of the server: 3080
Server's deployment URI: opensso
Naming URL (hit enter to accept default value,
http://host1.example.com:3080/opensso/namingservice):
```

4. **Compile the OpenSSO Enterprise sample programs.**

```
# cd /export/home/root/programs/opensso/opensso/samples/sdk
# ./coscripts/compile-samples.sh
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

5. **Run the `Login.sh` script.**

```
# cd /export/home/root/programs/opensso/opensso/samples/sdk/
# ./scripts/Login.sh
Realm (e.g. /):
```

```
Login module name (e.g. DataStore or LDAP): DataStore
Login locale (e.g. en_US or fr_FR): en_US
DataStore: Obtained login context
User Name:user1
Password:user1
Login succeeded.
Logged Out!!
```

Communicating Authentication Data as XML

Communication between applications and the Authentication Service is conducted with XML messages sent over `HTTP(s)`. The `remote-auth.dtd` is the template used to format the XML request messages sent to OpenSSO Enterprise and to parse the XML return messages received by the external application. The `remote-auth.dtd` is in the *path-to-context-root*`/opensso/WEB-INF` directory.

- "XML Messages and `remote-auth.dtd`" on page 26
- "XML/HTTP(s) Interface for Other Applications" on page 28

XML Messages and `remote-auth.dtd`

The following sections contain examples of XML messages based on the `remote-auth.dtd`.

Note – The client application writes XML messages based on the `remote-auth.dtd` but, when the messages are sent, the Authentication API adds additional XML code to them. This additional XML is not illustrated in the following examples.

- "Authentication Request Message from Application" on page 26
- "Response Message from OpenSSO Enterprise with Session Identifier and Callbacks" on page 27
- "Response Message from Application with User Credentials" on page 27
- "Authentication Status Message from OpenSSO Enterprise With Session Token" on page 27

Authentication Request Message from Application

This example illustrates the XML message sent to OpenSSO Enterprise requesting authentication. It opens a connection and asks for authentication requirements regarding the `examplerealm` realm to which the user will login.

```
<?xml version="1.0" encoding="UTF-8"?>
<AuthContext version="1.0"><Request authIdentifier="0">
<Login reamName="examplerealm">
<IndexTypeNamePair indexType="moduleInstance"><IndexName>LDAP</IndexName>
</IndexTypeNamePair></Login></Request></AuthContext>
```

Response Message from OpenSSO Enterprise with Session Identifier and Callbacks

This example illustrates an affirmative response from OpenSSO Enterprise that contains the session identifier for the original request (`authIdentifier`) as well as callback details.

```
<?xml version="1.0" encoding="UTF-8"?>
<AuthContext version="1.0"><Response authIdentifier="AQIC5wM2LY4SfczGP8Kp9
cqcaN1uW+C7CMdeR2afoN1ZxwY=@AAJTSQACMDE=#">
<GetRequirements><Callbacks length="3">
<PagePropertiesCallback isErrorState="false"><ModuleName>LDAP</ModuleName>
<HeaderValue>This server uses LDAP Authentication</HeaderValue>
<ImageName></ImageName><PageTimeOutValue>120</PageTimeOutValue>
<TemplateName></TemplateName>
<PageState>1</PageState>
</PagePropertiesCallback>
<NameCallback><Prompt> User Name: </Prompt></NameCallback>
<PasswordCallback echoPassword="false"><Prompt> Password: </Prompt>
</PasswordCallback></Callbacks></GetRequirements></Response></AuthContext>
```

Response Message from Application with User Credentials

This example illustrates the client's response to OpenSSO Enterprise. It contains the credentials input by the user to log in.

```
<?xml version="1.0" encoding="UTF-8"?>
<AuthContext version="1.0"><Request authIdentifier="AQIC5wM2LY4SfczGP8Kp9cqca
N1uW+C7CMdeR2afoN1ZxwY=@AAJTSQACMDE=#">
<SubmitRequirements><Callbacks length="2"><NameCallback><Prompt>User Name:</Prompt>
<Value>amadmin</Value>
</NameCallback>
<PasswordCallback echoPassword="false"><Prompt>Password:</Prompt>
<Value>admin123</Value>
</PasswordCallback></Callbacks></SubmitRequirements></Request></AuthContext>
```

Authentication Status Message from OpenSSO Enterprise With Session Token

This example illustrates the message from OpenSSO Enterprise specifying the user's successful authentication and the session token (`SSOToken`).

```
<?xml version="1.0" encoding="UTF-8"?>
<AuthContext version="1.0"><Response authIdentifier="AQIC5wM2LY4SfczGP8Kp9cqcaN1uW+
C7CMdeR2afoN1ZxwY=@AAJTSQACMDE=#">
<LoginStatus status="success" ssoToken="AQIC5wM2LY4SfczGP8Kp9cqcaN1uW+C7CMdeR2afoN1
ZxwY=@AAJTSQACMDE=#" successURL="http://blitz.red.sun.com/opensso/console">
<Subject>AQICOIy3FdTlJoAiOyyyZRTjOVBVWAb2e5MOAizI7ky3raaKypFE3e+GGZuX6chvLgDO32Zugn
```

```
pijo4xW4wUzyh2OAcdO9r9zhMU2Nhm206IuAmz9m18JWaYJpSHLqtBEcf1GbDrm3VAkERzIqsvkLKHmS1qc
yaT3BJ87wH0YQnPDze4/BroBZ8N5G3mPzPz5RbE07/1/w02yH9w0+UUFwwNBLayywGsr3bJ6emSSYqxos1N
1bo98xqL4FKAzItsfUAMd6v0ylWoqkoyoSdKYNHKbqvLDIeAfhqgoldxt64Or6HMXnOxz/jiVauh2mmwBpH
q1H2mOeF3agfUfuzKxBpLfELLwCH6QWcJmOZl0eNCFkGl7VwfnCJpTx1WcUhPSg0xD26D3dCQNruJpHPgzZ
FThe55M2gQ2qX+I1klmvzghSqiYfyoGg2SFeBeHE7iHuujO0e6UZgKDrOQPjU9aDh1GxxnsMQmaNkjuW+up
ghruWBGy+mDWmPQTme2bQWPIjBgB4wTDXTedeDzDBeulhCH4M0Ak9lvS7EIv6kHX5pRph6d0ND4/RVHka3k
WcQ5e0w2HpPjOxzNrWMfyXTkQJwOrA8yh1eBjG04VwiVqDV4wAV5EsIsIt0TrtAW2VZwV/KtLcGmjaKaT0H
dwRy0M4DHEqDbc6jF5ItVo9NneGFXMswPIoLm2nLuMrteAt7AtK7FGuCHlfYLavKoROtjaSuYTJGFwgz8Oi
vZ2r9boVnWVlz7ehwlyHvdfmpSKVl76Y4qEclX25m+lddAZE92RgSIrg97fp9gBOk2gVJWoQORNRDV2siHr
26 RiPLdvW3foG0hZgpLimJuLdByThRd/tdknDCCNRzelv7khr6nLPVPFVBgEJWlHmuffkdz4OsL0omFWpi
Jq05sQCPs/q6rq9ZJ98a8mcFK10BVPQki/1VfkIbKAdO4eswsIMalYkglBqXT4ARVTWRCWRNMCTDlQitF3g
T51AHn1WioFPm+NZ2KagVjQR6JFxHbdW0bKN7cLQViArJJFRtktR1BJh31/K+dAM2P+KbT1Lq13UUvXCynS
QwVbf7HJP5m3XrIQ6PtgZs4TB026H+iKy5T85YNL03j9sNnALiIKJEgvGLg2jxG+SU10xNLz3P3UVqmAnQI
9FIjmCtJcFtlLYR6BbkTvZVKxWz6+SoxNfDeKhIDwxkTNTLOzK491KzU/XAZTKmvdxTgf+WikbriBhFjsJ4
M6Npsq4p9Ksrjun9FVBTE/EUT5X/bY8zXLm0nw5KspQ7XRHPwrppQMVMMekz5qrNtQ9Cw/TeOhm4jvww/Bz
j4rydi7s7D10s2BWMfcuxmwQEipAWNmraKL37wWskrCdAzO2HXH4iJjWimiJ6J</Subject>
</LoginStatus></Response></AuthContext>
```

XML/HTTP(s) Interface for Other Applications

Applications written in a programming language other than Java or C can also exchange authentication information with OpenSSO Enterprise using the `XML/HTTP(s)` interface and the Authentication Service URL, `http://`*server_name.domain_name*`:`*port*`/`*service_deploy_uri*`/authservice`. An application can open a connection using the HTTP POST method. In order to access the Authentication Service in this manner, the client application must contain the following:

- A means of producing valid XML compliant with the `remote-auth.dtd.`
- HTTP 1.1 compliant client implementation to send XML-configured information to OpenSSO Enterprise.
- HTTP 1.1 compliant server implementation to receive XML-configured information from OpenSSO Enterprise.
- An XML parser to interpret the data received from OpenSSO Enterprise.

Tip – If contacting the Authentication Service directly through its URL, a detailed understanding of `remote-auth.dtd` will be needed for generating and interpreting the messages passed between the client and OpenSSO Enterprise.

Customizing Plug-Ins for the Password Reset User Interface

OpenSSO Enterprise provides customizable plug-ins for the Password Reset service. You must first configure the plug-ins using the Password Reset module in the OpenSSO Enterprise administration console. See Chapter 15, "Password Reset Service," in *Sun OpenSSO Enterprise 8.0 Administration Guide* and "Password Reset" in *Sun OpenSSO Enterprise 8.0 Administration Reference*.

When an application user wants to reset the user password for any reason, the following occurs:

1. The Password Reset service prompts the user for a userID and for the answer to an individualized security question.
2. The Password Reset service calls the `NotifyPassword` plug-in. This plug-in notifies the administrator that a user password is being reset. See the example "Notifying the Administrator that a User Password is Being Reset".
3. The Password Reset service calls the `PasswordGenerator` plug-in. This plug-in generates a new user password based on the developer's specification. See the example "Generating a New User Password" below. If no `PasswordGenerator` plug-in exists, then OpenSSO Enterprise generates a random-string password.

EXAMPLE 1–3 Notifying the Administrator that a User Password is Being Reset

```
import com.iplanet.am.util.Debug;
import com.iplanet.am.sdk.AMUser;

import com.sun.identity.password.plugins.NotifyPassword;
import com.sun.identity.password.ui.model.PWResetException;


/**
 * <code>SampleNotifyPassword</code> is a sample plugin implementation to show
 * how a customer can write their own notification mechanism after the
 * password is reset. It is based on the public interface
 * <code>NotifyPassword</code>.
 * This plugin is called(after configuring it in OpenSSO Enterprise  console by the
 * "Password Reset" module. OpenSSO Enterprise comes with a default plugin
 * implementation as well.
 */
public class SampleNotifyPassword implements NotifyPassword
{

    /**
     * Get a handle of the debug file.
     */
    private static Debug debug = Debug.getInstance("amConsole");
```

EXAMPLE 1–3 Notifying the Administrator that a User Password is Being Reset *(Continued)*

```
    /**
     * Constructs a <code>SampleNotifyPassword</code> object.
     */
    public SampleNotifyPassword() {
    }

    /**
     * Notifies user when password is changed.
     *
     * @param user <code>AMUser</code> object
     * @param password new password
     * @param localeContext <code>ISLocaleContext</code> object
     * @throws PWResetException if password cannot be notified
     */
    public void notifyPassword(
        AMUser user,
        String password,
        java.util.Locale localeContext)
        throws PWResetException
    {
        if (debug.messageEnabled()) {
            debug.message(
                "User password  has changed." +
                " User or Admin will be notified of this change");
        }

        /*
         * Put your own logic here using the
         * info(user, password, localcontext) to notify the user.
         * You can use the Access Manager SDK classes to send the
         *  notification or your own java classes.
         */

        debug.message(
            "User or Admin has been notified about the password change.");
    }
 }
```

EXAMPLE 1–4 Generating a New User Password

```
import com.iplanet.am.util.Debug;
import com.iplanet.am.sdk.AMUser;
import com.sun.identity.password.plugins.PasswordGenerator;
import com.sun.identity.password.ui.model.PWResetException;

/**
```

EXAMPLE 1–4 Generating a New User Password *(Continued)*

```
 * <code>SamplePasswordGenerator</code> is sample plugin implementation to show
 * how a customer can write their own password generation mechanism after the
 * password is reset. It is based on the public interface
 * <code>PasswordGenerator</code>. This plugin is called(after configuring it
 * in OpenSSO Enterprise console by the "Password Reset" module to generate the new
 * password after resetting the old password of the user. OpenSSO Enterprise comes
 * with a default plugin implementation as well. The default plugin generates
 * random passwords.
 */
public class SamplePasswordGenerator implements PasswordGenerator
{

    /**
     * Get a handle of the debug file.
     */
    private static Debug debug = Debug.getInstance("amConsole");



    /**
     * Constructs a random password generator object.
     */
    public SamplePasswordGenerator() {
    }


    /**
     * Generates new password for user.
     *
     * @param user <code>AMUser</code> object
     * @return new password for user
     * @throws PWResetException if password cannot be generated.
     */
    public String generatePassword(AMUser user) throws PWResetException {

        String newpassword = null;

        debug.message(
            "User password has been reset and a new password will be generated."
            );

        /*
         * Put your own logic to create a new password for the user whose
         * password has been reset through the "Password Reset" module.
         * For the sample purpose the new password has been put here as
         * "changeit"
```

EXAMPLE 1–4 Generating a New User Password *(Continued)*

```
         */
        newpassword = "changeit";

        debug.message(" User password  has changed to -" + newpassword);
        return newpassword;
    }
}
```

CHAPTER 2

Using the Policy Service API

OpenSSO Enterprise enables organizations to control the usage of, and access to, their resources. This chapter provides information about how the Policy Service allows you to define, manage, and enforce policies towards that end. It contains the following sections:

- "About the Policy Service Interfaces" on page 33
- "Enabling Authorization Using the Java Authentication and Authorization Service (JAAS)" on page 39
- "Using the Policy Evaluation API" on page 41
- "Using the Policy Evaluation Sample Program" on page 65

OpenSSO Enterprise also provides C APIs for external applications to connect to the Policy Service framework. For information about using the Policy C API, see Chapter 3, "Policy Data Types and Functions," in *Sun OpenSSO Enterprise 8.0 C API Reference for Application and Web Policy Agent Developers*. For a comprehensive listing of all OpenSSO Enterprise APIs and their usage, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

About the Policy Service Interfaces

The Policy Service provides the functionality to control access to web services and applications by providing authorization decisions based on defined and applicable *policies* or rules that define who or what is authorized to access a resource. In a single sign-on (SSO) environment, the Policy Service acts as authorization authority, providing authorization decisions that are enforced by a policy agent. The Policy Service acts as a Policy Administration Point (PAP) and a Policy Decision Point (PDP). As a PAP, it allows privileged users to create, modify, and delete access control policies. As a PDP, it provides access control decisions (after evaluating applicable policies) to a Policy Enforcement Point (PEP) which, in a OpenSSO Enterprise environment, is a policy agent.

For information on how the Policy Service works within a user session, see Chapter 6, "Models of the User Session and Single Sign-On Processes," in *Sun OpenSSO Enterprise 8.0 Technical Overview*. Additional information is in *Chapter 8, "Authorization and the Policy Service," in Sun*

OpenSSO Enterprise 8.0 Technical Overview. More information on policy agents can be found in *Sun OpenSSO Enterprise Policy Agent 3.0 User's Guide for J2EE Agents*.

The Policy Service provides an application programming interface (API) to manage policies and provide authorization decisions. It also provides a service provider interface (SPI) to extend the Policy Service functionality. These interfaces include the following packages:

- "`com.sun.identity.policy`" on page 34
- "`com.sun.identity.policy.client`" on page 37
- "`com.sun.identity.policy.interfaces` " on page 37
- "`com.sun.identity.policy.jaas`" on page 38

`com.sun.identity.policy`

The `com.sun.identity.policy` package contains the following classes for policy management and policy evaluation:

- "Policy Management Classes" on page 34
- "Policy Evaluation Classes" on page 35

Policy Management Classes

Policy management classes are used by privileged system administrators to programmatically add, look up, modify, replace and delete policies, and update the policy data store, if appropriate. Attempts by non-privileged users to manage policies will result in an exception and be logged. A valid session token is required to invoke any method provided by these classes. The key policy management classes are:

- "`PolicyManager`" on page 34
- "`Policy`" on page 35

`PolicyManager`

`com.sun.identity.policy.PolicyManager` is the top-level administrator class for policy management in a specific realm. This class provides methods that enable the administrator to add, look up, modify, replace and delete policies. Only a privileged user with access to the policy data store and a valid session token can create a `PolicyManager` object. Some of the more widely used methods include:

`getPolicyNames()`	Retrieves all named policies created in the realm for which the `PolicyManager` object was instantiated. This method can also take a pattern (filter) as an argument.
`getPolicy()`	Retrieves a policy when given the policy name.
`addPolicy()`	Adds a policy to the realm for which the `PolicyManager` object was instantiated. If a policy with the same name already exists, it will be overwritten.

`removePolicy()`	Removes a policy from the realm for which the `PolicyManager` object was instantiated.
`replacePolicy()`	Overwrites a policy already defined in the realm for which the `PolicyManager` object was instantiated.

`Policy`

`com.sun.identity.policy.Policy` represents a policy definition with all its intended parts, including Rule(s), Subject(s), Condition(s), Referral(s) and Response Provider(s). The `Policy` object can be saved in the policy data store if the `addPolicy()` or `replacePolicy()` methods from the `PolicyManager` class are invoked. This class contains methods for adding, removing, replacing or retrieving any of the parts of a policy definition.

Policy Evaluation Classes

Policy Decision APIs are used to evaluate policy decision when a principal attempts an action on a resource. This section covers some key classes that provide Policy Evaluation APIs. Some classes are also provided to be used only by privileged users to test policy decisions applicable to other users.

Policy evaluation classes are used to evaluate the applicable policy when a principal attempts an action on a resource and send a determination on whether the principal will be allowed or denied access. The key policy evaluation classes are:

- PolicyEvaluator
- ProxyPolicyEvaluator
- PolicyEvent

Caution – Policy evaluation classes from this package require a direct connection to the policy data store. These classes should be used with caution, and only when classes from `com.sun.identity.policy.client` cannot handle your use case. See "`com.sun.identity.policy.client`" on page 37.

`PolicyEvaluator`

`com.sun.identity.policy.PolicyEvaluator` evaluates policy privileges and provides policy decisions. It provides methods to evaluate access to one resource or a hierarchy of resources, and supports both boolean and non-boolean type policies. A valid session token of the principal attempting access is required to invoke any method of this class. A `PolicyEvaluator` class is created by calling the constructor with a service name. Key public methods of this class include:

`isAllowed()`	Evaluates a policy associated with the given resource and returns a boolean-type value indicating an allow or deny decision.

`getPolicyDecision()`	Evaluates policies and returns a decision as to whether the associated principal can perform the specified actions on the specified resource.
`getResourceResults()`	A `ResourceResult` contains policy decisions regarding a particular protected resource and its sub resources. `getResourceResults()` obtains these policy decisions. Possible values for the scope of objects retrieved are `ResourceResult.SELF_SCOPE` (returns an object that contains the policy decision for the specified resource only), `ResourceResult.SUBTREE_SCOPE` (includes policy decisions for the specified resource and its sub-resources), and `ResourceResult.STRICT_SUBTREE_SCOPE` (returns an object that contains one policy decision regarding the `resourceName` only). For example, the `PolicyEvaluator` class can be used to display links for a list of resources to which an authenticated user has access. The `getResourceResults()` method can be used to retrieve a list of resources to which the user has access from a defined `resourceName` parameter — a URL in the form **`http://`***host.domain***`:`***port*. The resources are returned as a `PolicyDecision` object based on the user's policies. If the user is allowed to access resources on different servers, this method needs to be called for each server.

Not all resources that have policy decisions are accessible to the user. Access depends on the `ActionDecision()` value contained in policy decisions.

ProxyPolicyEvaluator

`com.sun.identity.policy.ProxyPolicyEvaluator` allows a privileged user (top-level administrator, organization administrator, policy administrator, or organization policy administrator) to get policy privileges and evaluate policy decisions for any user in their scope of administration. `com.sun.identity.policy.ProxyPolicyEvaluatorFactory` is the singleton class used to get `ProxyPolicyEvaluator` instances. This is supported only within the OpenSSO Enterprise server process.

PolicyEvent

`com.sun.identity.policy.PolicyEvent` represents a policy event that could potentially change the current access status. A policy event is created and passed to registered policy listeners whenever there is a change in a policy rule. This class works with the `PolicyListener` class in the `com.sun.identity.policy.interface` package.

com.sun.identity.policy.client

The `com.sun.identity.policy.client` package contains classes that can be used by remote Java applications to evaluate policies and communicate with the Policy Service to get policy decisions. This package does not communicate with the policy data store therefore, use it when, for example, there is an intervening firewall. The package also maintains a local cache of policy decisions kept current either by a configurable time to live and/or notifications from the Policy Service.

com.sun.identity.policy.interfaces

The `com.sun.identity.policy.interfaces` package contains SPI for writing custom plug-ins to extend the Policy Service. The classes are used by service developers and policy administrators who need to provide additional policy features as well as support for legacy policies.

`Condition`	Provides methods used to constrain a policy to, for example, time-of-day or IP address. This interface allows the pluggable implementation of the conditions.
`PolicyListener`	Defines an interface for registering policy events when a policy is added, removed or changed. `PolicyListener` is used by the Policy Service to send notifications and by listeners to review policy change events.
`Referral`	Provides methods used to delegate the policy definition or evaluation of a selected resource (and its sub-resources) to another realm or policy server.
`ResourceName`	Provides methods to determine the hierarchy of the resource names for a determined service type. For example, these methods can check to see if two resources names are the same or if one is a sub-resource of the other.
`ResponseProvider`	Defines an interface to allow pluggable response providers into the OpenSSO Enterprise framework. Response providers are used to provide policy response attributes which typically provide attribute values from the user profile.
`Subject`	Provides methods to determine if an authenticated user is a member of the given subject.

Policy Service Provider Interfaces and Plug-Ins

OpenSSO Enterprise includes SPIs that work with the Policy Service framework to create and manage policies. You can develop customized plug-ins for creating custom policy subjects, referrals, conditions, and response providers. For information on creating custom policy plug-ins, see "Sample Code for Custom Subjects, Conditions, Referrals, and Response Providers" on page 43. The following table summarizes the Policy Service SPI, and lists the specialized Policy Service plug-ins that come bundled with OpenSSO Enterprise.

TABLE 2–1 Policy Service Service Provider Interfaces

Interface	Description
Subject	Defines a set of authenticated users for whom the policy applies. The following Subject plug-ins come bundled with OpenSSO Enterprise: Access Manager Identity Subject, Access Manager Roles, Authenticated Users, LDAP Groups, LDAP Roles, LDAP Users, Organization Web, and Services Clients.
Referral	Delegates management of policy definitions to another access control realm.
Condition	Specifies applicability of policy based on conditions such as IP address, time of day, authentication level. The following Condition plug-ins come bundled with OpenSSO Enterprise: Authentication Level, Authentication Scheme, IP Address, LE Authentication Level, Session, SessionProperty, and Time.
Resource Name	Allows a pluggable resource.
Response Provider	Gets attributes that are sent along with policy decision to the policy agent, and used by the policy agent to customize the client applications. Custom implementations of this interface are now supported in OpenSSO Enterprise.

`com.sun.identity.policy.jaas`

The `com.sun.identity.policy.jaas` package provides classes for performing policy evaluation against OpenSSO Enterprise using the Java Authentication and Authorization Service (JAAS) framework. JAAS is a set of APIs that enable services to authenticate and enforce access controls upon users. This package provides support for authorization only, making it possible to use JAAS interfaces to access the Policy Service. It contains the following implementations of JAAS classes:

- "`ISPermission`" on page 39
- "`ISPolicy`" on page 39

For more information see "Enabling Authorization Using the Java Authentication and Authorization Service (JAAS)" on page 39.

`ISPermission`

`com.sun.identity.policy.jaas.ISPermission` extends `java.security.Permission`, an abstract class for representing access to a resource. It represents the control of a sensitive operation, such as opening of a socket or accessing a file for a read or write operation. It does not grant permission for that operation, leaving that responsibility to the JAAS `AccessController` class which evaluates OpenSSO Enterprise policy against the Policy Service.

`ISPermission` covers the case when additional policy services are defined and imported provided they only have boolean action values as a JAAS permission only has a boolean result.

`ISPolicy`

`com.sun.identity.policy.jaas.ISPolicy` is an implementation of the JAAS abstract class `java.security.Policy` which represents the system policy for a Java application environment. It performs policy evaluation against the Policy Service instead of against the default file-based `PolicyFile`.

Enabling Authorization Using the Java Authentication and Authorization Service (JAAS)

The Java Authentication and Authorization Service (JAAS) is a set of API that can determine the identity of a user or computer attempting to run Java code, and ensure that the entity has the right to execute the requested functions. After an identity has been determined using authentication, a `Subject` object, representing a grouping of information about the entity, is created. Whenever the `Subject` attempts a restricted operation or access, the Java runtime uses the JAAS `AccessController` class to determine which, if any, `Principal` (representing one piece of information established during authentication) would authorize the request. If the `Subject` in question contains the appropriate `Principal`, the request is allowed. If the appropriate `Principal` is not present, an exception is thrown.

In OpenSSO Enterprise the custom implementation of the JAAS `java.security.Policy`, `com.sun.identity.policy.jaas.ISPolicy`, relies on the policy framework to provide policy evaluation for all Policy Service policies. Policy related to resources not under OpenSSO Enterprise control (for example, system level resources) are evaluated using JAAS.

OpenSSO Enterprise policy does not control access to `com.sun.security.auth.PolicyFile`, the default JAAS policy store.

For more information see the JAAS Java API Reference.

To enable authorization using JAAS in OpenSSO Enterprise use the JAAS `java.security.Policy` API to reset policy during run time. In the sample code, the client application resets the policy to communicate with OpenSSO Enterprise using `ISPolicy`. OpenSSO Enterprise provides the support needed to define policy through `ISPermission`.

EXAMPLE 2–1 Sample JAAS Authorization Code

```
public static void main(String[] args) {
   try {
       // Create an SSOToken

      AuthContext ac = new AuthContext("dc=iplanet,dc=com");
       ac.login();
       Callback[] callbacks = null;
       if (ac.hasMoreRequirements()) {
           callbacks = ac.getRequirements();

           if (callbacks != null) {
               try {
                   addLoginCallbackMessage(callbacks);
                    // this method sets appropriate responses
                    // in the callbacks.
                   ac.submitRequirements(callbacks);
               } catch (Exception e) { }
           }
       }
       if (ac.getStatus() == AuthContext.Status.SUCCESS) {
             Subject subject = ac.getSubject();
                            // get the authenticated subject

               Policy.setPolicy(new ISPolicy());
               // change the policy to  our own Policy

               ISPermission perm = new ("iPlanetAMWebAgentService",

                   "http://www.sun.com:80", "GET");
             Subject.doAs(subject, new PrivilegedExceptionAction() {
                 /* above statement means execute  run() method of the
                             /* Class PrivilegedExceptionAction()
                     as the specified subject */
                 public Object run() throws Exception {
                     AccessController.checkPermission(perm);
                       // the above will return quietly if the Permission
                                  //  has been granted
                       // else will throw access denied
                       // Exception, so if the above highlighed ISPermission
                                  // had not been granted, this return null;
                 }
            });
        }
   }
```

Using the Policy Evaluation API

The OpenSSO Enterprise policy framework defines Subject, Condition, Referral and Response Provider interfaces to enable you to create your own plug- ins to extend the functionality.

▼ To Develop a Custom Policy Plug-In

This information is also included in the OpenSSO Enterprise `/samples` directory. See the following file:

`http://`*openSSO-host*`:3080/opensso/samples/policy/policy-plugins.html`

1 **Write Java source files implementing Subject, Condition, Referral or ResponseProvider interface.**

See "Sample Code for Custom Subjects, Conditions, Referrals, and Response Providers" on page 43.

2 **Compile the source files to create class files.**

Include `opensso.jar` and `opesnsso-sharedlib.jar` in the classpath at compilation time.

3 **Package the compiled classes into a JAR file.**

In this example, the file is named `policy-plugins.jar`.

4 **Explode the `opensso.war` file.**

5 **Add the `policy-plugins.jar` file to `WEB-INF/lib` directory.**

Alternatively, you can copy the custom plug-in classes to the `WEB-INF/classes` directory. Be sure to maintain the directory structure corresponding to the Java package of the plug-in classes.

6 **Update `WEB-INF/classes/amPolicy.properties`.**

Add the globalization (L10N) values for the new internationalization (I18N) keys used by `iPlanetAMPolicyService`.

7 **Update `WEB-INF/classes/amPolicyConfig.properties`.**

Add L10N values for the new I18N keys used by `iPlanetAMPolicyConfigService`.

8 **Recreate the WAR file.**

9 **Redeploy the WAR file.**

Steps 1 through 9 have been already taken care of for the sample plug-ins included in OpenSSO distribution.

10 **Use the `ssoadm` command to register the new plug-ins with the `iPlanetAMPolicyService`.**

In the following example, the `password.txt` file contains the password of `amadmin`:

`ssoadm create-svc -X` *amPolicy_mod.xml* `-u amadmin -f` *password.txt*

See the sample `amPolicy_mod.xml`. The new `i18keys` are referred in the XML file. Add Corresponding L10N values in `amPolicy.properties`.

11 **Register the new plug-ins in one of the following ways:**

- **Use the `ssoadm` command to register the new plug-ins as choice values in the `iPlanetAMPolicyConfigService`.**

```
# ssoadm set-attr-choicevals -s iPlanetAMPolicyConfigService
-t Organization -a iplanet-am-policy-selected-subjects
-k a160=SampleSubject -u amadmin -f password.txt
 # ssoadm set-attr-choicevals -s iPlanetAMPolicyConfigService
-t Organization -a iplanet-am-policy-selected-conditions
-k a161=SampleCondition -u amadmin -f password.txt
# ssoadm set-attr-choicevals -s iPlanetAMPolicyConfigService
-t Organization -a iplanet-am-policy-selected-referrals
-k a162=SampleReferral -u amadmin -f password.txt
#ssoadm set-attr-choicevals -s iPlanetAMPolicyConfigService
-t Organization -a sun-am-policy-selected-responseproviders
-k a163=SampleResponseProvider -u amadmin -f password.txt
```

- **Use the `ssoadm` command to register the new plug-ins as enabled for a selected realm.**

```
# ssoadm add-attr-defs -s iPlanetAMPolicyConfigService -t Organization
-a iplanet-am-policy-selected-subjects=SampleSubject -u amadmin -f password.txt
# ssoadm add-attr-defs -s iPlanetAMPolicyConfigService -t Organization
-a iplanet-am-policy-selected-conditions=SampleCondition -u amadmin -f password.txt
# ssoadm add-attr-defs -s iPlanetAMPolicyConfigService -t Organization
-a iplanet-am-policy-selected-referrals=SampleReferral -u amadmin -f password.txt
# ssoadm add-attr-defs -s iPlanetAMPolicyConfigService -t Organization
-a sun-am-policy-selected-responseproviders=SampleResponseProvider
-u amadmin -f password.txt
```

- **Use the administration console to register the new plug-ins for existing realms.**

  a. **Log in to the administration console as `amadmin` or `administrator`.**

  b. **Navigate to the Realm > Services > Policy Configuration.**

  c. **In the Policy Configuration page, enable or disable the selected plug-in.**

12 **Restart the web application or the container.**

13 **Use either the administration console or the `ssoadm` command to add the instances of the new plug-ins while defining policies.**

The new plug-ins are available as choices in appropriate policy management pages of the administration console.

14 **To disable the custom plug-ins from being added to newly-created policies:**

a. **In the administration console, navigate to Access Control > Realm > Services | Policy Configuration.**

b. **Deselect the appropriate custom plug-ins.**

c. **Save the Policy Configuration properties page for exisiting realms.**

If you navigate to Configuration > Global > Policy Configuration and do this, the custom plug-ins would be deselected for the realms that would be created subsequently.

15 **Copy your custom plug-in classes to `<TOOLS_HOME>/classes`.**

Be sure to maintain the directory structure corresponding to the Java package of the plug-in classes. You can copy the classes of bundled, custom sample plug-ins from the exploded `opensso.war` directory `WEB-INF/classes/com/sun/identity/samples/policy`. This is required if you plan to use `ssoadm` to export or add policies.

Sample Code for Custom Subjects, Conditions, Referrals, and Response Providers

OpenSSO Enterprise provides subject, condition, referral, and response provider interfaces that enable you to develop your own custom subjects, conditions, referrals, and response providers. The following samples illustrate how to implement these custom objects:

- "`SampleSubject.java`" on page 43
- "`SampleCondition.java`" on page 49
- "`SampleReferral.java`" on page 53
- "`SampleResponseProvider.java`" on page 59

`SampleSubject.java`

Implements the `Subject` interface. This subject applies to all the authenticated users who have valid `SSOTokens`.

EXAMPLE 2–2 `SampleSubject.java`

```
package com.sun.identity.samples.policy;
```

EXAMPLE 2–2 `SampleSubject.java` *(Continued)*

```
import java.util.*;
//import java.security.Principal;

import com.iplanet.sso.*;
import com.sun.identity.policy.*;
import com.sun.identity.policy.interfaces.Subject;

/**
 * The class <code>Subject</code> defines a collection
 * of users (or subject) to whom the specified policy is applied.
 * A complete implementation of this interface can have complex
 * boolean operations to determine if the given user identified
 * by the <code>SSOToken</code> belongs to this collection.
 * <p>
 * The interfaces are seperated into administrative
 * interfaces and evaluation interfaces. The administrative interfaces
 * will be used by GUI/CLI component to create a <code>Subject</code>
 * object and the evaluation interfaces will be used by the policy evaluator.
 *
 * This sample inplementation defines the collection of all users who have
 * been authenticated (a user with a valid SSOToken.).
 */
public class SampleSubject implements Subject {

    /**
     * Constructor with no parameter
     */
    public SampleSubject() {
    // do nothing
    }

    /**
     * Initialize (or configure) the <code>Subject</code>
     * object. Usually it will be initialized with the environment
     * paramaters set by the system administrator via SMS.
     * For example in a Role implementation, the configuration
     * parameters could specify the directory server name, port, etc.
     *
     * @param configParams configuration parameters as a map.
     * The values in the map is <code>java.util.Set</code>,
     * which contains one or more configuration paramaters.
     *
     * @exception PolicyException if an error occured during
     * initialization of <code>Subject</code> instance
     */
```

EXAMPLE 2–2 SampleSubject.java *(Continued)*

```
    public void initialize(Map configParams)
        throws PolicyException {
    // do nothing
    }

    /**
     * Returns the syntax of the values the
     * <code>Subject</code> implementation can have.
     * @see com.sun.identity.policy.Syntax
     *
     * @param token the <code>SSOToken</code> that will be used
     * to determine the syntax
     *
     * @return set of of valid names for the user collection.
     *
     * @exception SSOException if SSO token is not valid
     * @exception PolicyException if unable to get the list of valid
     * names.
     *
     * @return syntax of the values for the <code>Subject</code>
     */

    public Syntax getValueSyntax(SSOToken token) {
    return (Syntax.CONSTANT);
    }


    /**
     * Returns the syntax of the values the
     * <code>Subject</code> implementation can have.
     * @see com.sun.identity.policy.Syntax
     *
     * @param token the <code>SSOToken</code> that will be used
     * to determine the syntax
     *
     * @return set of of valid names for the user collection.
     *
     * @exception SSOException if SSO token is not valid
     * @exception PolicyException if unable to get the list of valid
     * names.
     *
     * @return syntax of the values for the <code>Subject</code>
     */

    public ValidValues getValidValues(SSOToken token) {
    return (new ValidValues(ValidValues.SUCCESS,
```

EXAMPLE 2–2 SampleSubject.java *(Continued)*

```
                Collections.EMPTY_SET));
}


/**
 * Returns a list of possible values for the <code>Subject
 * </code>. The implementation must use the <code>SSOToken
 * </code> <i>token</i> provided to determine the possible
 * values. For example, in a Role implementation
 * this method will return all the roles defined
 * in the organization.
 *
 * @param token the <code>SSOToken</code> that will be used
 * to determine the possible values
 *
 * @return <code>ValidValues</code> object
 *
 * @exception SSOException if SSO token is not valid
 * @exception PolicyException if unable to get the list of valid
 * names.
 */

public ValidValues getValidValues(SSOToken token, String pattern) {
return (new ValidValues(ValidValues.SUCCESS,
                Collections.EMPTY_SET));
}


/**
 * Returns the display name for the value for the given locale.
 * For all the valid values obtained through the methods
 * <code>getValidValues</code> this method must be called
 * by GUI and CLI to get the corresponding display name.
 * The <code>locale</code> variable could be used by the
 * plugin to customize
 * the display name for the given locale.
 * The <code>locale</code> variable
 * could be <code>null</code>, in which case the plugin must
 * use the default locale (most probabily en_US).
 * This method returns only the display name and should not
 * be used for the method <code>setValues</code>.
 * Alternatively, if the plugin does not have to localize
 * the value, it can just return the <code>value</code> as is.
 *
 * @param value one of the valid value for the plugin
 * @param locale locale for which the display name must be customized
```

EXAMPLE 2–2 `SampleSubject.java` *(Continued)*

```
     *
     * @exception NameNotFoundException if the given <code>value</code>
     * is not one of the valid values for the plugin
     */

    public String getDisplayNameForValue(String value, Locale locale)
    throws NameNotFoundException {
        return value;
    }


    /**
     * Returns the values that was set using the
     * method <code>setValues</code>.
     *
     * @return values that have been set for the user collection
     */

    public Set getValues() {
    return (Collections.EMPTY_SET);
    }


    /**
     * Sets the names for the instance of the <code>Subject</code>
     * object. The names are obtained from the policy object,
     * usually configured when a policy is created. For example
     * in a Role implementation, this would be name of the role.
     *
     * @param names names selected for the instance of
     * the user collection object.
     *
     * @exception InvalidNameException if the given names are not valid
     */

    public void setValues(Set names) throws InvalidNameException {
    }


    /**
     * Determines if the user belongs to this instance
     * of the <code>Subject</code> object.
     * For example, a Role implemenation
     * would return <code>true</code> if the user belongs
     * the specified role; <code>false</code> otherwise.
     *
```

EXAMPLE 2–2 SampleSubject.java *(Continued)*

```
     * @param token single-sign-on token of the user
     *
     * @return <code>true</code> if the user is memeber of the
     * given subject; <code>false</code> otherwise.
     *
     * @exception SSOException if SSO token is not valid
     * @exception PolicyException if an error occured while
     * checking if the user is a member of this subject
     */

    public boolean isMember(SSOToken token)
    throws SSOException {
    return (SSOTokenManager.getInstance().isValidToken(token));
    }

    /**
     * Indicates whether some other object is "equal to" this one.
     *
     * @param o another object that will be compared with this one
     *
     * @return <code>true</code> if eqaul; <code>false</code>
     * otherwise
     */

    public boolean equals(Object o) {
    if (o instanceof SampleSubject) {
        return (true);
    }
    return (false);
    }


    /**
     * Creates and returns a copy of this object.
     *
     * @return a copy of this object
     */

    public Object clone() {
    return (new SampleSubject());
    }
}
```

SampleCondition.java

Implements the Condition interface. This condition makes the policy applicable to those users whose user name length is greater than or equal to the length specified in the condition.

EXAMPLE 2–3 SampleCondition.java

```
package com.sun.identity.samples.policy;

import java.util.*;

import com.sun.identity.policy.interfaces.Condition;
import com.sun.identity.policy.ConditionDecision;
import com.sun.identity.policy.PolicyException;
import com.sun.identity.policy.PolicyManager;
import com.sun.identity.policy.Syntax;
import com.iplanet.sso.SSOException;
import com.iplanet.sso.SSOToken;
import com.iplanet.sso.SSOTokenManager;


/**
 * The class <code>SampleCondition</code> is a plugin
 * implementation of <code>Condition</code> interface.
 * This condition object provides the policy framework with the
 * condition decision based on the length of the user’s name.
 */

public class SampleCondition implements Condition {

    /** Key that is used to define the minimum of the user name length
     *  for which the policy would apply.  The value should be
     *  a Set with only one element. The element should be a
     *  String, parsable as an integer.
     */

    public static final String USER_NAME_LENGTH = "userNameLength";

    private List propertyNames;
    private Map properties;
    private int nameLength;

    /** No argument constructor
     */
    public SampleCondition() {
         propertyNames = new ArrayList();
         propertyNames.add(USER_NAME_LENGTH);
    }
```

EXAMPLE 2–3 SampleCondition.java *(Continued)*

```
    /**
     * Returns a set of property names for the condition.
     *
     * @return set of property names
     */

    public List getPropertyNames()
    {
        return propertyNames;
    }

    /**
     * Returns the syntax for a property name
     * @see com.sun.identity.policy.Syntax
     *
     * @param String property name
     *
     * @return <code>Syntax<code> for the property name
     */
    public Syntax getPropertySyntax(String property)
    {
        return (Syntax.ANY);
    }

    /**
     * Gets the display name for the property name.
     * The <code>locale</code> variable could be used by the
     * plugin to customize the display name for the given locale.
     * The <code>locale</code> variable could be <code>null</code>, in which
     * case the plugin must use the default locale.
     *
     * @param String property name
     * @param Locale locale for which the property name must be customized
     * @return display name for the property name
     */
    public String getDisplayName(String property, Locale locale)
      throws PolicyException
    {
        return property;
    }

    /**
     * Returns a set of valid values given the property name. This method
     * is called if the property Syntax is either the SINGLE_CHOICE or
     * MULTIPLE_CHOICE.
```

EXAMPLE 2–3 SampleCondition.java *(Continued)*

```
     *
     * @param String property name
     * @return Set of valid values for the property.
     * @exception PolicyException if unable to get the Syntax.
     */
    public Set getValidValues(String property) throws PolicyException
    {
        return (Collections.EMPTY_SET);
    }


   /** Sets the properties of the condition.
    *  Evaluation of ConditionDecision is influenced by these properties.
    *  @param properties the properties of the condition that governs
    *         whether a policy applies. The properties should
    *         define value for the key USER_NAME_LENGTH. The value should
    *         be a Set with only one element. The element should be
    *         a String, parsable as an integer. Please note that
    *         properties is not cloned by the method.
    *
    *  @throws PolicyException if properties is null or does not contain
    *          value for the key USER_NAME_LENGTH or the value of the key is
    *          not a Set with one String element that is parsable as
    *          an integer.
    */

   public void setProperties(Map properties) throws PolicyException {
       this.properties = (Map)((HashMap) properties);
       if ( (properties == null) || ( properties.keySet() == null) ) {
           throw new PolicyException("properties can not be null or empty");
       }

       //Check if the key is valid
       Set keySet = properties.keySet();
       Iterator keys = keySet.iterator();
       String key = (String) keys.next();
       if ( !USER_NAME_LENGTH.equals(key) ) {
           throw new PolicyException(
               "property " + USER_NAME_LENGTH + " is not defined");
       }

       // check if the value is valid
       Set nameLengthSet = (Set) properties.get(USER_NAME_LENGTH);
       if (( nameLengthSet == null ) || nameLengthSet.isEmpty()
           || ( nameLengthSet.size() > 1 )) {
           throw new PolicyException(
```

EXAMPLE 2–3 SampleCondition.java *(Continued)*

```
                "property value is not defined or invalid");
        }

        Iterator nameLengths = nameLengthSet.iterator();
        String nameLengthString = null;
        nameLengthString = (String) nameLengths.next();
        try {
            nameLength = Integer.parseInt(nameLengthString);
        } catch (Exception e) {
            throw new PolicyException("name length value is not an integer");
        }
    }


    /** Get properties of this condition.
     */
    public Map getProperties() {
        return properties;
    }


    /**
     * Gets the decision computed by this condition object.
     *
     * @param token single sign on token of the user
     *
     * @param env request specific environment map of key/value pairs.
     *       SampleCondition doesn't use this parameter.
     *
     * @return the condition decision. The condition decision
     *        encapsulates whether a policy applies for the request.
     *
     * Policy framework continues evaluating a policy only if it
     * applies to the request as indicated by the CondtionDecision.
     * Otherwise, further evaluation of the policy is skipped.
     *
     * @throws SSOException if the token is invalid
     */

    public ConditionDecision getConditionDecision(SSOToken token, Map env)
            throws PolicyException, SSOException {
    boolean allowed = false;

        String userDN = token.getPrincipal().getName();
        // user DN is in the format like "uid=username,ou=people,dc=example,dc=com"
        int beginIndex = userDN.indexOf("=");
```

EXAMPLE 2–3 SampleCondition.java *(Continued)*

```
        int endIndex = userDN.indexOf(",");
        if (beginIndex >= endIndex) {
            throw (new PolicyException("invalid user DN"));
        }

        String userName = userDN.substring(beginIndex+1, endIndex);
        if (userName.length() >= nameLength) {
            allowed = true;
        }

    return new ConditionDecision(allowed);
    }


    public Object clone() {
    Object theClone = null;
    try {
        theClone = super.clone();
    } catch (CloneNotSupportedException e) {
            throw new InternalError();
    }
    return theClone;
    }

}
```

SampleReferral.java

Implements the `Referral` interface. `SampleReferral.java` gets the referral policy decision from a text file `SampleReferral.properties` located in the `/samples` directory.

EXAMPLE 2–4 SampleReferral.java

```
package com.sun.identity.samples.policy;

import java.io.*;
import java.util.*;

import com.sun.identity.policy.*;
import com.sun.identity.policy.interfaces.Referral;
import com.iplanet.sso.SSOToken;
import com.iplanet.sso.SSOException;
import com.iplanet.am.util.SystemProperties;

public class SampleReferral implements Referral {
```

EXAMPLE 2–4 SampleReferral.java *(Continued)*

```
    static final String SEPARATOR = ":";
    static String PROPERTIES = "samples/policy/SampleReferral.properties";
    static String INSTALL_DIR = SystemProperties.get("com.iplanet.am.installdir");
    static Properties properties = new Properties();
    private String _name;
    private Set _values;

    /** No argument constructor */
    public SampleReferral() {
    }

    /**Initializes the referral with a map of Configuration parameters
     * @param configurationMap a map containing configuration
     *        information. Each key of the map is a configuration
     *        parameter. Each value of the key would be a set of values
     *        for the parameter. The map is cloned and a reference to the
     *        clone is stored in the referral
     */
    public void initialize(Map configurationMap) {
    }

    /**Sets the name of this referral
     * @param name name of this referral
     */
    private void setName(String name) {
        _name = name;
    }

    /**Gets the name of this referral
     * @return the name of this referral
     */
    private String getName() {
        return _name;
    }

    /**Sets the values of this referral.
     * @param values a set of values for this referral
     *        Each element of the set has to be a String
     * @throws InvalidNameException if any value passed in the
     * values is invalid
     */
    public void setValues(Set values) throws InvalidNameException {
        _values = values;
    }

    /**Gets the values of this referral
```

EXAMPLE 2–4 `SampleReferral.java` *(Continued)*

```
     * @return the values of this referral
     *                 Each element of the set would be a String
     */
    public Set getValues() {
        return _values;
    }

    /**
     * Returns the display name for the value for the given locale.
     * For all the valid values obtained through the methods
     * <code>getValidValues</code> this method must be called
     * by GUI and CLI to get the corresponding display name.
     * The <code>locale</code> variable could be used by the
     * plugin to customize
     * the display name for the given locale.
     * The <code>locale</code> variable
     * could be <code>null</code>, in which case the plugin must
     * use the default locale (most probabily en_US).
     * This method returns only the display name and should not
     * be used for the method <code>setValues</code>.
     * Alternatively, if the plugin does not have to localize
     * the value, it can just return the <code>value</code> as is.
     *
     * @param value one of the valid value for the plugin
     * @param locale locale for which the display name must be customized
     *
     * @exception NameNotFoundException if the given <code>value</code>
     * is not one of the valid values for the plugin
     */
    public String getDisplayNameForValue(String value, Locale locale)
    throws NameNotFoundException {
    return value;
    }

    /**Gets the valid values for this referral
     * @param token SSOToken
     * @return <code>ValidValues</code> object
     * @throws SSOException, PolicyException
     */
    public ValidValues getValidValues(SSOToken token)
            throws SSOException, PolicyException {
        return getValidValues(token, "*");
    }

    /**Gets the valid values for this referral
     * matching a pattern
```

EXAMPLE 2–4 SampleReferral.java *(Continued)*

```
     * @param token SSOToken
     * @param pattern a pattern to match against the value
     * @return </code>ValidValues</code> object
     * @throws SSOException, PolicyException
     */
    public ValidValues getValidValues(SSOToken token, String pattern)
            throws SSOException, PolicyException {
        Set values = new HashSet();
        values.add(PROPERTIES);
        return (new ValidValues(ValidValues.SUCCESS,
                                values));
    }

    /**Gets the syntax for the value
     * @param token SSOToken
     * @see com.sun.identity.policy.Syntax
     */
    public Syntax getValueSyntax(SSOToken token)
            throws SSOException, PolicyException {
        return (Syntax.SINGLE_CHOICE);
    }

    /**Gets the name of the ReferralType
     * @return name of the ReferralType representing this referral
     */
    public String getReferralTypeName()
    {
        return "SampleReferral";
    }

    /**Gets policy results
     * @param token SSOToken
     * @param resourceType resource type
     * @param resourceName name of the resource
     * @param actionNames a set of action names
     * @param envParameters a map of enivronment parameters.
     *        Each key is an environment parameter name.
     *        Each value is a set of values for the parameter.
     * @return policy decision
     * @throws SSOException
         * @throws PolicyException
     */
    public PolicyDecision getPolicyDecision(SSOToken token, String resourceType,
        String resourceName, Set actionNames, Map envParameters)
            throws SSOException, PolicyException {
```

EXAMPLE 2–4 `SampleReferral.java` *(Continued)*

```
        PolicyDecision pd = new PolicyDecision();
        Iterator elements = _values.iterator();
        if (!elements.hasNext()) {
            return pd;
        }

        String fileName = (String)elements.next();
        fileName = INSTALL_DIR + "/" + fileName;
        try {
            InputStream is = new FileInputStream(fileName);
            if (is == null) {
                return pd;
            }
            properties.load(is);
        } catch (Exception e) {
            return pd;
        }

        String serviceName = getProperty("servicename");
        if (!serviceName.equals(resourceType)) {
            return pd;
        }

        String resName = getProperty("resourcename");
        if (!resName.equals(resourceName)) {
            return pd;
        }

        List actionNameList = getPropertyValues("actionnames");
        List actionValueList = getPropertyValues("actionvalues");

        int numOfActions = actionNameList.size();
        int numOfValues = actionValueList.size();

        if ((numOfActions == 0 || (numOfValues == 0)
                               || numOfActions != numOfValues)) {
            return pd;
        }

        Iterator namesIter = actionNameList.iterator();
        Iterator valuesIter = actionValueList.iterator();

        for (int i = 0; i < numOfActions; i++) {
            String actionName = (String)namesIter.next();
            String actionValue = (String)valuesIter.next();
            if (actionNames.contains(actionName)) {
```

EXAMPLE 2–4 SampleReferral.java *(Continued)*

```
                Set values = new HashSet();
                values.add(actionValue);
                ActionDecision ad = new ActionDecision(
                    actionName, values, null, Long.MAX_VALUE);
                pd.addActionDecision(ad);
            }
        }
        return pd;
    }


    private String getProperty(String key)
    {
        return properties.getProperty(key);
    }


    private List getPropertyValues(String name) {
        List values = new ArrayList();
        String value = getProperty(name);
        if ( value != null ) {
            StringTokenizer st = new StringTokenizer(value, SEPARATOR);
            while ( st.hasMoreTokens() ) {
               values.add(st.nextToken());
            }
        }
        return values;
    }

    /** Gets resource names rooted at the given resource name for the given
     *  serviceType that could be governed by this referral
     * @param token ssoToken sso token
     * @param serviceTypeName service type name
     * @param rsourceName resource name
     * @return names of sub resources for the given resourceName.
     *         The return value also includes the resourceName.
     *
     * @throws PolicyException
     * @throws SSOException
     */
    public Set getResourceNames(SSOToken token, String serviceTypeName,
            String resourceName) throws PolicyException, SSOException {
        return null;
    }

}
```

SampleResponseProvider.java

Implements the ResponseProvider interface. SampleResponseProvider.java takes as input the attribute for which values are retrieved from OpenSSO Enterprise and sent back in the Policy Decision. If the attribute does not exist in the user profile, no value is sent back in the response. SampleResponseProvider.java relies on the underlying Identity Repository service to retrieve the attribute values for the Subject(s) defined in the policy.

EXAMPLE 2–5 SampleResponseProvider.java

```
package com.sun.identity.samples.policy;

import com.sun.identity.policy.PolicyException;
import com.sun.identity.policy.PolicyUtils;
import com.sun.identity.policy.PolicyConfig;
import com.sun.identity.policy.PolicyManager;
import com.sun.identity.policy.interfaces.ResponseProvider;
import com.sun.identity.policy.Syntax;

import com.iplanet.sso.SSOToken;
import com.iplanet.sso.SSOException;

import com.sun.identity.idm.AMIdentity;
import com.sun.identity.idm.IdUtils;
import com.sun.identity.idm.IdRepoException;

import java.util.List;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.Locale;
import java.util.Map;
import java.util.HashSet;
import java.util.HashMap;
import java.util.Set;
import java.util.StringTokenizer;
import java.util.Collections;

/**
 * This class is an implementation of <code>ResponseProvider</code> interface.
 * It takes as input the attribute for which values are to be fetched from
 * the access manager and sent back in the Policy Decision.
 * if the attribute does not exist in the use profile no value is sent
 * back in the response.
 * It relies on underlying Identity repository service to
 * fetch the attribute values for the Subject(s) defined in the policy.
 * It computes a <code>Map</code> of response attributes
 * based on the <code>SSOToken</code>, resource name and  env map passed
 * in the method call <code>getResponseDecision()</code>.
```

EXAMPLE 2–5 SampleResponseProvider.java *(Continued)*

```
 *
 * Policy framework would make a call to the ResponseProvider in a
 * policy only if the policy is applicable to a request as determined by
 * <code>SSOToken</code>, resource name, <code>Subjects</code> and <code>Conditions
 * </code>.
 *
 */
public class SampleResponseProvider implements ResponseProvider {

    public static final String ATTRIBUTE_NAME = "AttributeName";

    private Map properties;
    private static List propertyNames = new ArrayList(1);

    private boolean initialized=false;
    private String orgName = null;

    static {
        propertyNames.add(ATTRIBUTE_NAME);
    }

    /**
     * No argument constructor.
     */
    public SampleResponseProvider () {

    }


    /**
     * Initialize the SampleResponseProvider object by using the configuration
     * information passed by the Policy Framework.
     * @param configParams the configuration information
     * @exception PolicyException if an error occured during
     * initialization of the instance
     */

    public void initialize(Map configParams) throws PolicyException {
        // get the organization name
        Set orgNameSet = (Set) configParams.get(
                                     PolicyManager.ORGANIZATION_NAME);
        if ((orgNameSet != null) && (orgNameSet.size() != 0)) {
            Iterator items = orgNameSet.iterator();
            orgName = (String) items.next();
        }
    /**
```

EXAMPLE 2–5 `SampleResponseProvider.java` *(Continued)*

```
         * Organization name is not used in this sample, but this is code
         * to illustrate how any other custom response provider can get data
         * out from the policy configuration service and use it in
         * getResponseDecision() as necessary.
     */
    initialized = true;
    }


    /**
     * Returns a list of property names for the responseprovider.
     *
     * @return <code>List</code> of property names
     */
    public List getPropertyNames()  {
         return propertyNames;
    }

    /**
     * Returns the syntax for a property name
     * @see com.sun.identity.policy.Syntax
     *
     * @param property property name
     *
     * @return <code>Syntax<code> for the property name
     */
    public Syntax getPropertySyntax(String property) {
        return (Syntax.LIST);
    }

    /**
     * Gets the display name for the property name.
     * The <code>locale</code> variable could be used by the plugin to
     * customize the display name for the given locale.
     * The <code>locale</code> variable could be <code>null</code>, in which
     * case the plugin must use the default locale.
     *
     * @param property property name
     * @param locale locale for which the property name must be customized
     * @return display name for the property name.
     * @throws PolicyException
     */
    public String getDisplayName(String property, Locale locale)
            throws PolicyException {
    return property;
    }
```

EXAMPLE 2–5 `SampleResponseProvider.java` *(Continued)*

```
    /**
     * Returns a set of valid values given the property name.
     *
     * @param property property name
     * from the PolicyConfig Service configured for the specified realm.
     * @return Set of valid values for the property.
     * @exception PolicyException if unable to get the Syntax.
     */
    public Set getValidValues(String property) throws PolicyException {
    if (!initialized) {
        throw (new PolicyException("idrepo response provider not yet "
        +"initialized"));
    }
        return Collections.EMPTY_SET;
    }

    /** Sets the properties of the responseProvider plugin.
     *  This influences the response attribute-value Map that would be
     *  computed by a call to method <code>getResponseDecision(Map)</code>
     *  These attribute-value pairs are encapsulated in
     *  <code>ResponseAttribute</code> element tag which is a child of the
     *  <code>PolicyDecision</code> element in the PolicyResponse xml
     *  if the policy is applicable to the user for the resource, subject and
     *  conditions defined.
     *  @param properties the properties of the responseProvider
     *         Keys of the properties have to be String.
     *         Value corresponding to each key have to be a Set of String
     *         elements. Each implementation of ResponseProvider could add
     *         further restrictions on the keys and values of this map.
     *  @throws PolicyException for any abnormal condition
     */
    public void setProperties(Map properties) throws PolicyException {
        if ( (properties == null) || ( properties.isEmpty()) ) {
            throw new PolicyException("Properties cannot be null or empty");
        }
        this.properties = properties;

        //Check if the keys needed for this provider are present namely
     // ATTRIBUTE_NAME
    if (!properties.containsKey(ATTRIBUTE_NAME)) {
            throw new PolicyException("Missing required property");
    }
    /**
         * Addtional validation on property name and values can be done
         * as per the individual use case
```

EXAMPLE 2–5 `SampleResponseProvider.java` *(Continued)*

```
         */
    }

    /** Gets the properties of the responseprovider
     *  @return properties of the responseprovider
     *  @see #setProperties
     */
    public Map getProperties() {
    return (properties == null)
        ? null : Collections.unmodifiableMap(properties);
    }

    /**
     * Gets the response attributes computed by this ResponseProvider object,
     * based on the sso token and map of environment parameters
     *
     * @param token single-sign-on token of the user
     *
     * @param env specific environment map of key/value pairs
     * @return  a Map of response attributes.
     *          Keys of the Map are attribute names ATTRIBUTE_NAME or
     *          Value is a Set of Strings representing response attribute
     *          values.
     *
     * @throws PolicyException if the decision could not be computed
     * @throws SSOException if SSO token is not valid
     *
     */
    public Map getResponseDecision(SSOToken token,
            Map env) throws PolicyException, SSOException {

    Map respMap = new HashMap();
    Set attrs = (Set)properties.get(ATTRIBUTE_NAME);
    Set values = null;
    if ((attrs != null) && !(attrs.isEmpty())) {
            try {
                if (token.getPrincipal() != null) {
                    AMIdentity id = IdUtils.getIdentity(token);
                    Map idRepoMap = id.getAttributes(attrs);
                    if (idRepoMap != null) {
                        for (Iterator iter = attrs.iterator(); iter.hasNext(); )
            {
                            String attrName = (String)iter.next();
                            values = new HashSet();
                            Set subValues = (Set)idRepoMap.get(attrName);
                            if (subValues != null) {
```

EXAMPLE 2–5 `SampleResponseProvider.java` *(Continued)*

```
                                values.addAll(subValues);
                            }
                    respMap.put(attrName, values);
                        }
                    }
                } else {
                    throw (new PolicyException("SSOToken principal is null"));
                }
            } catch (IdRepoException ide) {
                throw new PolicyException(ide);
          }
        }
    return respMap;
    }


    /**
     * Returns a copy of this object.
     *
     * @return a copy of this object
     */
    public Object clone() {
        SampleResponseProvider theClone = null;
        try {
            theClone = (SampleResponseProvider)super.clone();
        } catch (CloneNotSupportedException e) {
            // this should never happen
            throw new InternalError();
        }

        if (properties != null) {
            theClone.properties = new HashMap();
            Iterator iter = properties.keySet().iterator();
            while (iter.hasNext()) {
                Object obj = iter.next();
                Set values = new HashSet();
                values.addAll((Set) properties.get(obj));
                theClone.properties.put(obj, values);
            }
        }
        return theClone;
    }
}
```

Using the Policy Evaluation Sample Program

OpenSSO Enterprise provides sample code to demonstrate how to write a policy evaluation plug-in. The source file `PolicyEvaluationSample.java` is located in the following directory:

OpenSSO-base`/opensso/samples/sdk/source/samples/policy`.

▼ To Run the Policy Evaluator Sample Program

1 **Download and unzip the `opensso.zip` file.**

```
# unzip opensso.zip
Archive:  opensso.zip
   creating: opensso/
        ...
```

2 **Unzip the Client SDK zip file.**

```
# cd opensso/samples
# unzip opensso-client.zip
```

3 **Set up the Client SDK.**

```
# pwd
/export/home/root/programs/opensso/opensso/samples
# cd sdk
# .Kchmod u+x scripts/*.sh
$ ./scripts/setup.sh
Debug directory (make sure this directory exists): /opensso/client-debugs
Application user (e.g. URLAccessAgent) password: agentadmin
Protocol of the server: http
Host name of the server: host1.example.com
Port of the server: 3080
Server's deployment URI: opensso
Naming URL (hit enter to accept default value,
http://host1.example.com:3080/opensso/namingservice):
```

4 **Compile the OpenSSO Enterprise sample programs.**

```
# cd /export/home/root/programs/opensso/opensso/samples/sdk
# ./coscripts/compile-samples.sh
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

5 **Run the `run-policy-evaluation-sample` script.**

```
# ./scripts/run-policy-evaluation-sample.sh

Using properties file:policyEvaluationSample
```

```
sample properties:
action.name:GET
resource.name:http://www.sample.com:80/banner.html
user.name:amadmin
service.name:iPlanetAMWebAgentService
user.password:adminadmin
------------------------------------------------------------------------------:
Entering getSSOToken():userName=amadmin,password=adminadmin
TokenID:AQIC5wM2LY4SfcwbUOUH0UNrRB2AvKY+XDa94ViserX1OSM=@AAJTSQACMDE=#
returning from getSSOToken()
Entering getPolicyDecision():resourceName=http://www.sample.com:80/banner.html,
serviceName=iPlanetAMWebAgentService,actionName=GET
policyDecision:<PolicyDecision>
<ResponseAttributes>
</ResponseAttributes>
</PolicyDecision>

returning from getPolicyDecision()
```

◆ ◆ ◆ CHAPTER 3

Using the Session Service API

The OpenSSO Enterprise Session Service maintains information about an authenticated user's session across all web applications in a single sign-on environment. This chapter describes the interfaces used to track session data for purposes of single sign-on and related sample code. It includes the following sections:

- "A Simple Single Sign-On Scenario" on page 67
- "Inside a User Session" on page 68
- "About the Session Service Interfaces" on page 70
- "Using the Single Sign-On Sample Programs" on page 74

For a comprehensive listing of all Java interfaces and their usage, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

OpenSSO Enterprise also includes an API for session management in C applications. For information see Chapter 4, "Single Sign-On Data Types and Functions," in *Sun OpenSSO Enterprise 8.0 C API Reference for Application and Web Policy Agent Developers*.

A Simple Single Sign-On Scenario

In a single sign-on scenario, a user logs in to access a protected resource. Once the user has successfully authenticated to OpenSSO Enterprise, a user session is created and stored in OpenSSO Enterprise memory. The user uses browser cookies or URL query parameters to carry a session identifier. Each time the user requests access to another protected resource, the new application must verify the user's identity. It does not ask the user to present credentials. Instead, the application uses the session identifier and the Session Service interfaces to retrieve the user's session information from OpenSSO Enterprise. If it is determined from the session information that the user has already been authenticated and the session is still valid, the new application allows the user access to its data and operations. If the user is not authenticated, or if the session is no longer valid, the requested application prompts the user to present credentials a second time. Until logging out, this scenario is played out every time the user accesses a protected resource in the single sign-on environment. For more detailed information about user sessions and single sign-on, see Chapter 6, “Models of the User Session and Single Sign-On Processes,” in *Sun OpenSSO Enterprise 8.0 Technical Overview*.

Inside a User Session

A user session is, more specifically, a data structure created by the Session Service to store information about a user session. Cookies are used to store a token that uniquely identifies the session data structure. A session data structure contains attributes and properties that define the user's identity and time-dependent behaviors. One example is the maximum time before the session expires.

The values of most of these attributes and properties are set by services other than the Session Service (primarily, the Authentication Service). The Session Service only provides storage for session information and enforces some of the time-dependent behavior. An example of such enforcement is invalidating and destroying sessions which exceed their maximum idle time or maximum session time.

A session data structure may contain the following:

- “Session Attributes” on page 68
- “Protected Properties” on page 69

Session Attributes

The session data structure contains the following fixed attributes:

`sun.am.universalIdentifier`	This universal, unique session identifier is an opaque, global string that programmatically identifies a specific session data structure. With this identifier, a resource is able to retrieve session information.
`Type`	This is specifies the type of client: USER or APPLICATION.
`State`	This is the state of the session: VALID, INVALID, DESTROYED or INACTIVE.
`maxIdleTime`	This is the maximum time in minutes without activity before the session will expire and the user must reauthenticate.
`maxSessionTime`	This is the maximum time in minutes before the session expires and the user must reauthenticate.
`maxCachingTime.`	This is the maximum time in minutes before the client contacts Identity Server to refresh cached session information

`latestAccessTime`	This refers to the last time the user accessed the resource.
`creationTime`	This is the time at which the session token was set to a valid state.

Protected Properties

The session data structure also contains an extensible set of protected (or core) properties. The following protected properties are set by OpenSSO Enterprise and can only be modified by OpenSSO Enterprise (primarily the Authentication Service).

`Organization`	This is the DN of the organization to which the user belongs.
`Principal`	This is the DN of the user.
`Principals`	This is a list of names to which the user has authenticated. (This property may have more then one value defined as a pipe separated list.)
`UserId`	This is the user's DN as returned by the module, or in the case of modules other than LDAP or Membership, the user name. (All Principals must map to the same user. The `UserId` is the user DN to which they map.)
`UserToken`	This is a user name. (All Principals must map to the same user. The UserToken is the user name to which they map.)
`Host`	This is the host name or IP address for the client.
`authLevel`	This is the highest level to which the user has authenticated.
`AuthType`	This is a pipe separated list of authentication modules to which the user has authenticated (for example, `module1|module2|module3`).
`Service`	Applicable for service-based authentication only, this is the service to which the user belongs.
`loginURL`	This is the client's login URL.
`Hostname`	This is the host name of the client.
`cookieSupport`	This attribute contains a value of true if the client browser supports cookies.
`authInstant`	This is a string that specifies the time at which the authentication took place.
`SessionTimedOut`	This attribute contains a value of true if the session has timed out.

About the Session Service Interfaces

All OpenSSO Enterprise services (except for the Authentication Service) require a valid session identifier (programmatically referred to as `SSOToken`) to process an HTTP request. External applications developed using the Session Service interfaces and protected by a policy agent also require an `SSOToken` to determine access privileges. The `SSOToken` is an encrypted, unique string that identifies a specific session data structure stored by OpenSSO Enterprise. If the `SSOToken` is known to a OpenSSO Enterprise service or an external protected resource such as an application, the service or application can access all user information and session data stored in the session data structure it identifies. After successful authentication, the `SSOToken` is transported using cookies or URL parameters, allowing participation in single sign-on.

The Session Service provides Java interfaces to allow OpenSSO Enterprise services and external applications to participate in the single sign-on functionality. The `com.iplanet.sso` package contains the tools for creating, destroying, retrieving, validating and managing session data structures. All external applications designed to participate in the single sign-on solution must be developed using this API. In the case of a remote application, the invocation is forwarded to OpenSSO Enterprise by the client libraries using XML messages over HTTP(S).

The `com.iplanet.sso` package includes the following:

- "`SSOTokenManager`" on page 70
- "`SSOToken`" on page 72
- "`SSOTokenListener`" on page 74

SSOTokenManager

The `SSOTokenManager` class contains the methods needed to get, validate, destroy and refresh the session identifiers that are programmatically referred to as the `SSOToken`. To obtain an instance of `SSOTokenManager`, call the `getInstance()` method. The `SSOTokenManager` instance can be used to create an `SSOToken` object using one of the forms of the `createSSOToken()` method. The `destroyToken()` method is called to invalidate and delete a token to end the session. Either the `isValidToken()` and `validateToken()` methods can be called to verify whether a token is valid (asserting successful authentication). `isValidToken()` returns true or false depending on whether the token is valid or invalid, respectively. `validateToken()` throws an exception only when the token is invalid; nothing happens if the token is valid. The `refreshSession()` method resets the idle time of the session. The following code sample illustrates how to use `SSOTokenManager` to validate a user session.

EXAMPLE 3–1 Code Sample for Validating a User Session

```
try {

        /* get an instance of the SSOTokenManager */
```

EXAMPLE 3–1 Code Sample for Validating a User Session *(Continued)*

```
SSOTokenManager ssoManager = SSOTokenManager.getInstance();

        /* The request here is the HttpServletRequest. Get
        /* SSOToken for session associated with this request.
        /* If the request doe not have a valid session cookie,
        /* a Session Exception would be thrown.*/

SSOToken ssoToken = ssoManager.createSSOToken(request);

        /* use isValid method to check if token is valid or not.
        /* This method returns true for valid token, false otherwise. */

if (ssoManager.isValidToken(ssoToken)) {

        /* If token is valid, this information may be enough for
        /* some applications to grant access to the requested
        /* resource. A valid user represents a user who is
        /* already authenticated. An application can further
        /* utilize user identity information to apply
        /* personalization logic .*/

} else {

        /* Token is not valid, redirect the user login page. */

}

        /* Alternative: use of validateToken method to check
        /* if token is valid */

try {
ssoManager.validateToken(ssoToken);

        /* handle token is valid */

} catch (SSOException e) {

        /* handle token is invalid */

}

        /*refresh session. idle time should be 0 after refresh. */

ssoManager.refreshSession(ssoToken);
```

EXAMPLE 3–1 Code Sample for Validating a User Session *(Continued)*

```
} catch (SSOException e) {

        /* An error has occurred. Do error handling here. */

}
```

SSOToken

The SSOToken interface represents the session identifier returned from the createSSOToken() method, and is used to retrieve session data such as the authenticated principal name, authentication method, and other session information (for example, session idle time and maximum session time). The SSOToken interface has methods to get predefined session information such as:

- getProperty() is used to get any information about the session, predefined or otherwise (for example, information set by the application).
- setProperty() can be used by the application to set application-specific information in the session.
- addSSOTokenListener() can be used to set a listener to be invoked when the session state has become invalid.

Caution – The methods getTimeLeft() and getIdleTime() return values in seconds while the methods getMaxSessionTime() and getMaxIdleTime() return values in minutes.

The following code sample illustrates how to use SSOToken to print session properties.

EXAMPLE 3–2 Using SSOToken to Print Session Properties

```
        /* get http request output stream for output */

PrintWriter out = response.getWriter();

        /* get the sso token from http request */

SSOTokenManager ssoManager = SSOTokenManager.getInstance();
SSOToken ssoToken = ssoManager.createSSOToken(request);

        /* get the sso token ID from the sso token */

SSOTokenID ssoTokenID = ssoToken.getTokenID();
```

EXAMPLE 3–2 Using SSOToken to Print Session Properties *(Continued)*

```
out.println("The SSO Token ID is "+ssoTokenID.toString());

        /* use validate method to check if the token is valid */

try {
ssoManager.validateToken(ssoToken);
out.println("The SSO Token validated.");

} catch (SSOException e) {
out.println("The SSO Token failed to validate.");
}

        /* use isValid method to check if the token is valid */

if (!ssoManager.isValidToken(token)) {
out.println("The SSO Token is not valid.");
} else {

        /* get some values from the SSO Token */

java.security.Principal principal = ssoToken.getPrincipal();
out.println("Principal name is "+principal.getName());

String authType = ssoToken.getAuthType();
out.println("Authentication type is "+authType);

int authLevel = ssoToken.getAuthLevel();
out.println("Authentication level is "+authLevel);

long idleTime = ssoToken.getIdleTime();
out.println("Idle time is "+idleTime);

long maxIdleTime = ssoToken.getMaxIdleTime();
out.println("Max idle time is "+maxIdleTime);

long maxTime = token.getMaxSessionTime();
out.println("Max session time is "+maxTime);

String host = ssoToken.getHostName();
out.println("Host name is "+host);

        /* host name is a predefined information of the session,
        /* and can also be obtained the following way */

String hostProperty = ssoToken.getProperty("HOST");
```

EXAMPLE 3–2 Using SSOToken to Print Session Properties *(Continued)*

```
out.println("Host property is "+hostProperty);

        /* set application specific information in session */

String appPropertyName = "app1propA";
String appPropertyValue = "appValue";
ssoToken.setProperty(appPropertyName, appPropertyValue);

        /* now get the app specific information back */

String appValue = ssoToken.getProperty(appPropertyName);
if (appValue.equals(appPropertyValue)) {
out.println("Property "+appPropertyName+",
 value "+appPropertyValue+" verified to be set.");
} else {
out.println("ALERT: Setting property "+appPropertyName+" failed!");

}

}
```

SSOTokenListener

The `SSOTokenListener` class allows the application to be notified when a `SSOToken` has become invalid — for example, when a session has timed out.

Using the Single Sign-On Sample Programs

OpenSSO Enterprise provides the following code samples that demonstrate how to use the single sign-on APIs. Both code samples are stand-alone Java applications. Sources files for the samples are located in the following directory:

OpenSSO-base`/export/home/root/programs/opensso/opensso/samples/sdk/source/com/sun/identity`

▼ To Use `CommandLineSSO.java` to Return User Profile Attributes

The `CommandLineSSO.java` sample program demonstrates how to retrieve the user profile given the correct user credentials. Required input is an organization name in DN form. The program returns user profile attributes.

1 **Download and unzip the `opensso.zip` file.**

```
# unzip opensso.zip
Archive:  opensso.zip
   creating: opensso/
        ...
```

2 **Unzip the Client SDK zip file.**

```
# cd opensso/samples
# unzip opensso-client.zip
```

3 **Set up the Client SDK.**

```
# pwd
/export/home/root/programs/opensso/opensso/samples
# cd sdk
# .Kchmod u+x scripts/*.sh
$ ./scripts/setup.sh
Debug directory (make sure this directory exists): /opensso/client-debugs
Application user (e.g. URLAccessAgent) password: agentadmin
Protocol of the server: http
Host name of the server: host1.example.com
Port of the server: 3080
Server’s deployment URI: opensso
Naming URL (hit enter to accept default value,
http://host1.example.com:3080/opensso/namingservice):
```

4 **Compile the OpenSSO Enterprise sample programs.**

```
# ./scripts/compile-samples.sh
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

5 **Run the `CommandLineSSO.sh` script.**

```
# ./scripts/Co mmandLineSSO.sh
Organization: /
DataStore: Obtained login context
User Name:user1
Password:user1
Successful authentication ...
```

```
User Name: id=user1,ou=user,dc=opensso,dc=java,dc=net
User Attributes:
sn=[user1]
cn=[user1]
objectclass=[sunFederationManagerDataStore, iplanet-am-managed-person,
iplanet-am-user-service, top, inetadmin, organizationalperson,
iPlanetPreferences, person, inetuser, sunAMAuthAccountLockout,
sunIdentityServerLibertyPPService, inetorgperson, sunFMSAML2NameIdentifier]
userpassword=[{SSHA}8ALDdWR+vBihl1tED1qWuJhrVhJHc5BfKC0oeg==]
uid=[user1]
nsrole=[cn=opensso,dc=opensso,dc=java,dc=net]
givenname=[user1]
inetuserstatus=[Active]
```

▼ To Use the `SSOTokenSample.java` Program to Create an SSOToken

The `SSOTokenSample.java` sample program demonstrates how to create an SSOToken and call various methods from the token. This includes getting or setting the session properties. Required input is a token ID. The program retrieves basic single sign-on token information and session properties.

Before You Begin Copy the token ID so that you can paste it into the appropriate field.

In a browser, log in to the OpenSSO Enterprise administration console. For example, using Firefox 2.0.0.16, in the Browser task bar, in the Edit menu, choose Preferences. In the Preferences page, click Privacy and then Show Cookies. In the Cookies page, search for the cookie for the appropriate domain. Copy the cookie listed in the Content field.

1 Download and unzip the `opensso.zip` file.

```
#
# unzip opensso.zip
Archive:  opensso.zip
   creating: opensso/
        ...
```

2 Unzip the Client SDK zip file.

```
# cd opensso/samples
# unzip opensso-client.zip
```

3 Set up the Client SDK.

```
# pwd
/export/home/root/programs/opensso/opensso/samples
```

```
# cd sdk
# .Kchmod u+x scripts/*.sh
$ ./scripts/setup.sh
Debug directory (make sure this directory exists): /opensso/client-debugs
Application user (e.g. URLAccessAgent) password: agentadmin
Protocol of the server: http
Host name of the server: host1.example.com
Port of the server: 3080
Server’s deployment URI: opensso
Naming URL (hit enter to accept default value,
http://host1.example.com:3080/opensso/namingservice):
```

4 Compile the OpenSSO Enterprise sample programs.

```
# ./scripts/compile-samples.sh
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

5 Run the `SSOTokenSample.sh` script.

```
# ./scripts/SSOTokenSample.sh
```

6 Paste the SSOToken ID you copied from the browser.

```
Enter SSOToken ID: AQIC5wM2LY4Sfcw2g2W3gXhdCf1tg%2FO
gGq2x0lodK3p0uU0%3D%40AAJTSQACMDE%3D%23
```

◆ ◆ ◆ CHAPTER 4

Running OpenSSO Enterprise in Debugging Mode

When you run OpenSSO Enterprise in Debugging Mode, debugging information is written to files in the `ConfigurationDirectory/uri/debug` directory. You can view the debugging files to help you determine where errors or other process problems occur.

To Run OpenSSO Enterprise in Debugging Mode

1. Open the OpenSSO Enterprise Console.
2. Click the Configuration tab.
3. Go to Sites and Servers > `serverName` > General, where `serverName` is the name of the OpenSSO Enterprise server instance you want to debug.
4. Edit the Debug Directory attribute.
5. Specify one of the following debug levels:

Off:	No debugging information is written to the debug files.
Error:	Use this level in production environments. During production, there should be no errors in the debug files.
Warning:	Allows Error and Warning debug messages to be written. Do not use the Warning level in a production environment. This setting can cause severe performance degradation due to excessive debug messages.
Message:	Allows detailed code tracing. Do not use the Message level in a production environment. This level can cause severe performance degradation due to excessive debug messages.

Debugging information is written to files in the `ConfigurationDirectory/uri/debug` directory. By default, debugging information for an OpenSSO enterprise service or major component is written into a file named for the service or component:

- Authentication
- CoreSystem
- amAuthContextLocal
- WebServices
- IDRepo
- Policy
- Configuration
- Session

To Merge Debugging Output into One File

1. In the OpenSSO Enterprise administration console, go to Configuration > Sites and Servers > `serverName` > General.

   In this example, `serverName` is the name of the OpenSSO Enterprise server instance you are debugging.
2. Set the Merge Debug files attribute to ON.

CHAPTER 5

Understanding the Federation Options

Sun OpenSSO Enterprise has a robust framework for implementing a federated identity infrastructure. A federated identity infrastructure allows single sign-on that crosses internet domain boundaries. This chapter contains the following sections.

Understanding Federation

The umbrella term federation encompasses both *identity federation* and *provider federation*. The concept of identity federation begins with the notion of a *virtual identity*. On the internet, one person might have a multitude of accounts set up for access to various business, community and personal service providers. In creating these accounts, the person might have used different names, user identifiers, passwords or preferences to customize, for example, a news portal, a bank, a retailer, and an email provider. A *local identity* refers to the set of attributes that an individual might have with each of these service providers. These attributes uniquely identify the individual for that particular provider and can include a name, phone number, passwords, social security number, address, credit records, bank balances or bill payment information. After implementing a federated identity infrastructure, a user can associate, connect or bind the local identities they have configured with multiple service providers into a *federated identity*. With a federated identity the user can then login at one service provider's site and move to an affiliated (trusted) service provider site without having to reauthenticate or re-establish their identity.

The concept of provider federation as defined in a federation-based environment begins with the notion of a *security domain* (referred to as a *circle of trust* in OpenSSO Enterprise). A circle of trust is a group of service providers (with at least one identity provider) that agree to join together to exchange user authentication information using open standards and technologies.

Once a group of providers has been federated within a circle of trust, authentication accomplished by the identity provider in that circle is honored by all affiliated service providers. Thus, federated single sign-on can be enabled amongst all membered providers as well as identity federation among users. For more information on the federation process in OpenSSO Enterprise, see the *Sun OpenSSO Enterprise 8.0 Technical Overview*.

Understanding Federated Single Sign-on

Federated single sign-on allows authentication among multiple internet domains using multiple authentication authorities — with one authority asserting the identity of the user to the other. OpenSSO Enterprise supports the following federation specifications:

- Liberty Alliance Project Identity Federation Framework (Liberty ID-FF) 1.2 Specifications
- WS-Federation 1.1 Metadata
- Security Assertion Markup Language (SAML)

Here are some general rules to follow when deciding which federation option will work best in your environment.

- Use SAML v2 whenever possible as it supersedes both the Liberty ID-FF and SAML v1.x specifications.
- The Liberty ID-FF and SAML v1.x should only be used when integrating with a partner that is not able to use SAML v2.
- SAML v1.x should suffice for single sign-on basics.
- The Liberty ID-FF can be used for more sophisticated functions and capabilities, such as global sign-out, attribute sharing, web services.
- When deploying OpenSSO Enterprise with Microsoft Active Directory with Federation Services, you must use WS-Federation.

For more information, see Chapter 11, “Choosing a Federation Option,” in *Sun OpenSSO Enterprise 8.0 Technical Overview*.

Note – The proprietary OpenSSO Enterprise single sign-on mechanism, due to its dependency on browser cookies, is limited to single sign-on within a single internet domain only. The proprietary OpenSSO Enterprise cross domain single sign-on (CDSSO) mechanism uses a single authentication authority which means only one user identity can exist in the entire system. If the situation fits, CDSSO may be a solution worthy of further evaluation.

1. Only Sun products (OpenSSO Enterprise and agents) are involved.
2. All policy agents are configured to use the same OpenSSO Enterprise instance where multiple instances are available.
3. Multiple instances of OpenSSO Enterprise, configured for high-availability, must all reside in a single DNS domain.

Only policy agents can reside in different DNS domains. For more information on these proprietary features, see Part II, "Access Control Using OpenSSO Enterprise," in *Sun OpenSSO Enterprise 8.0 Technical Overview*.

Federated Single Sign-on Using OpenSSO Enterprise

In order to communicate identity attributes for the purpose of federated single sign-on, you need, at the least, two instances of OpenSSO Enterprise configured in one circle of trust. Circles of trust configured for real time interactions must have, at the least, one instance of OpenSSO Enterprise acting as the circle's identity provider and one instance of OpenSSO Enterprise acting as a service provider. To prepare your instances of OpenSSO Enterprise, you need to exchange and import the metadata for all participating identity and service providers, and assemble the providers into a circle of trust. The following steps are an overview of the process.

1. Decide whether the instance of OpenSSO Enterprise you are configuring will act as either an identity provider, a service provider, or both.
2. Create standard and extended metadata configuration files containing the appropriate metadata for your organization. See Chapter 1, ?ssoadm Command Line Interface Reference,? in Sun OpenSSO Enterprise 8.0 Administration Reference.
3. Create a circle of trust.
4. Import your organization's provider metadata into the circle of trust.
5. Determine which organizations will be added to the circle of trust as identity providers and service providers and import a standard and an extended metadata configuration file for each.

   Note – The values in these files will come from the providers themselves.

6. Import the provider metadata into the circle of trust

See Chapter 9, "Configuring and Managing Federation," in *Sun OpenSSO Enterprise 8.0 Administration Guide* for more information.

Executing a Multi-Protocol Hub

Because of the federation options available, OpenSSO Enterprise has implemented a new feature: the multi-protocol hub. The multi-protocol hub is an identity provider that supports all federation protocols implemented in OpenSSO Enterprise. It enables seamless single sign-on and single logout with service providers that communicate using the different federation protocols. OpenSSO Enterprise ships with a multi-protocol hub sample that demonstrates single sign-on and single logout within one hub that includes one Liberty ID-FF service provider, one SAML v2 service provider and one WS-Federation service provider. The sample is located in `/`*path-to-context-root*`/opensso/samples/multiprotocol`. Open `index.html` for more information.

CHAPTER 6

Implementing the Liberty Alliance Project Identity-Federation Framework

Sun OpenSSO Enterprise has a robust framework for implementing federated single sign-on infrastructures based on the Liberty Alliance Project Identity-Federation Framework (Liberty ID-FF). It provides interfaces for creating, modifying, and deleting circles of trust, service providers, and identity providers as well as samples to get you started. This chapter covers the following topics:

- "Customizing the Federation Graphical User Interface" on page 85
- "Using the Liberty ID-FF Packages" on page 87
- "Accessing Liberty ID-FF Endpoints" on page 89
- "Executing the Liberty ID-FF Sample" on page 90

Customizing the Federation Graphical User Interface

The Federation Service uses JavaServer Pages™ (JSP™) to define its look and feel. JSP are HTML files that contain additional code to generate dynamic content. More specifically, a JavaServer page contains HTML code to display static text and graphics, as well as application code to generate information. When the page is displayed in a web browser, it contains both the static HTML content and, in the case of the Federation component, dynamic content retrieved through calls to the Federation API. An administrator can customize the look and feel of the interface by changing the HTML tags in the JSP but the invoked APIs must not be changed.

After a default installation, the JSP are located in */path-to-context-root/*`opensso/config/federation/default`. The files in this directory provide the default content to the Liberty ID-FF Federation capability. To customize the pages for a specific organization, this default directory can be copied and renamed to reflect the name of the organization (or any value). This directory would then be placed at the same level as the default directory, and the files within this directory would be modified as needed. The following table lists the JSP including details on what each page is used for and the invoked API that cannot be modified.

TABLE 6–1 Federation JSP and Invoked Interfaces

JSP Name	Description
`CommonLogin.jsp`	Displays a link to the local login page as well as links to the login pages of the trusted identity providers. This page is displayed when a user is not logged in locally or with an identity provider. `com.sun.liberty.LibertyManager` is the invoked interface. The list of identity providers is obtained by using the `getIDPList(hostedProviderID)` method.
`Error.jsp`	Displays an error page when an error has occurred. `com.sun.liberty.LibertyManager` is the invoked interface.
`Federate.jsp`	When a user clicks a federate link on a provider page, this page displays a drop-down list of all providers with which the user is not yet federated. `com.sun.liberty.LibertyManager` is the invoked interface. The list is constructed with the `getProvidersToFederate(realm,providerID,providerRole,userName)` method.
`FederationDone.jsp`	Displays the status of a federation (success or cancelled). `com.sun.liberty.LibertyManager` is the invoked interface. It checks the status with the `isFederationCancelled(request)` method.
`Footer.jsp`	Displays a branded footer that is included on all the pages. No APIs are invoked.
`Header.jsp`	Displays a branded header that is included on all the pages. No APIs are invoked.
`ListOfCOTs.jsp`	Displays a list of circles of trust. When a user is authenticated by an identity provider and the service provider belongs to more than one circle of trust, this page displays and the user is prompted to select a circle of trust as their preferred domain. In the case that the provider belongs to only one domain, this page will not be displayed. `com.sun.liberty.LibertyManager` is the invoked interface. The list is obtained with the `getListOfCOTs(providerID)` method.
`LogoutDone.jsp`	Displays the status of the local logout operation. `com.sun.liberty.LibertyManager` is the invoked interface.
`NameRegistration.jsp`	When a federated user clicks a Name Registration link on a provider page to register a new Name Identifier from one provider to another, this JSP is displayed. `com.sun.liberty.LibertyManager` is the invoked interface.
`NameRegistrationDone.jsp`	Displays the status of `NameRegistration.jsp`. When finished, this page is displayed. `com.sun.liberty.LibertyManager` is the invoked interface.

TABLE 6–1 Federation JSP and Invoked Interfaces *(Continued)*

JSP Name	Description
`Termination.jsp`	When a user clicks a defederate link on a provider page, this page displays a drop-down list of all providers with which the user has federated and from which the user can choose to defederate. `com.sun.liberty.LibertyManager` is the invoked interface. The list is constructed with the `getFederatedProviders(userName)` method which returns all active providers to which the user is already federated.
`TerminationDone.jsp`	Displays the status of federation termination (success or cancelled). `com.sun.liberty.LibertyManager` is the invoked interface. Status is checked using the `isTerminationCancelled(request)` method.

Using the Liberty ID-FF Packages

The following packages form the Federation API. For more detailed information, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

- "`com.sun.identity.federation.accountmgmt`" on page 87
- "`com.sun.identity.federation.common`" on page 87
- "`com.sun.identity.federation.message`" on page 87
- "`com.sun.identity.federation.message.common`" on page 88
- "`com.sun.identity.federation.plugins`" on page 88
- "`com.sun.identity.federation.services`" on page 88
- "`com.sun.liberty`" on page 89

`com.sun.identity.federation.accountmgmt`

The `com.sun.identity.federation.accountmgmt` package contains the `FSAccountFedInfo` class which retrieves the information from the federated user account. After Liberty ID-FF federation is successfully completed, two attributes are set. The `FSAccountFedInfo` class contains the value of one of them: the `iplanet-am-user-federation-info` attribute.

`com.sun.identity.federation.common`

The `com.sun.identity.federation.common` package contains the `IFSConstants` interface which represents common constants used by the federation API.

`com.sun.identity.federation.message`

The `com.sun.identity.federation.message` package contains classes which define the federation protocol messages.

com.sun.identity.federation.message.common

The `com.sun.identity.federation.message.common` package contains classes which can be used by federation protocol messages.

com.sun.identity.federation.plugins

The `com.sun.identity.federation.plugins` package contains the `FederationSPAdapter` interface which can be implemented to allow applications to customize user specific processing before and after invoking the federation protocols. For example, a service provider may want to choose to redirect to a specific location after successful single sign-on. A singleton instance of this `FederationSPAdapter` is used during runtime so make sure the implementation of the methods (except `initialize()`) are thread safe.

com.sun.identity.federation.services

The `com.sun.identity.federation.services` package provides interfaces for writing custom plug-ins that can be used during the federation or single sign-on process. The interfaces are described in the following table.

TABLE 6–2 `com.sun.identity.federation.services` Interfaces

Interface	Description
`FSRealmAttributeMapper`	Plug-in for mapping the attributes passed from the identity provider to local attributes on the service provider side during the single sign-on. `com.sun.identity.federation.services.FSDefaultRealmAttribut` is the default implementation.
`FSRealmAttributePlugin`	Plug-in for an identity provider to add `AttributeStatements` into a SAML assertion during the single sign-on process. `com.sun.identity.federation.services.FSDefaultRealmAttribut` is the default implementation.
`FSRealmIDPProxy`	Interface used to find a preferred identity provider to which an authentication request can be proxied. `com.sun.identity.federation.services.FSRealmIDPProxyImpl` is the default implementation.

`com.sun.liberty`

The `com.sun.liberty` package contains the `LibertyManager` class which must be instantiated by web applications that want to access the Federation framework. It also contains the methods needed for account federation, session termination, log in, log out and other actions. Some of these methods are described in the following table.

TABLE 6–3 `com.sun.liberty` Methods

Method	Description
`getFederatedProviders()`	Returns a specific user's federated providers.
`getIDPFederationStatus()`	Retrieves a user's federation status with a specified identity provider. This method assumes that the user is already federated with the provider.
`getIDPList()`	Returns a list of all trusted identity providers.
`getIDPList()`	Returns a list of all trusted identity providers for the specified hosted provider.
`getProvidersToFederate()`	Returns a list of all trusted identity providers to which the specified user is not already federated.
`getSPList()`	Returns a list of all trusted service providers.
`getSPList()`	Returns a list of all trusted service providers for the specified hosted provider.
`getSPFederationStatus()`	Retrieves a user's federation status with a specified service provider. This method assumes that the user is already federated with the provider.

Accessing Liberty ID-FF Endpoints

For each Liberty ID-FF feature, there are endpoints listening for requests or generating responses. The endpoint URLs are provided in the metadata that is exchanged with other partners in the circle of trust. Following is a list of the Liberty ID-FF endpoints:

- SOAPReceiver is a servlet that listens for SOAP-communicated requests. For example, single logout or requests for artifacts.
- ProcessLogout is a servlet that accepts HTTP-based single logout requests.
- ProcessTermination is a servlet that accepts HTTP-based federation termination requests.
- ProcessRegistration is a servlet that accepts Name Identifier registration requests.
- SingleSignOnService is a servlet on the identity provider side that accepts single sign-on requests.

- ReturnLogout is a servlet that accepts single logout return requests.
- AssertionConsumerService is a servlet on the service provider side that accepts single sign-on responses.

Executing the Liberty ID-FF Sample

OpenSSO Enterprise includes sample code and files that can be used to demonstrate the different Liberty ID-FF protocols such as Account Federation, Single Sign On, Single Logout and Federation Termination. The sample is located in /*path-to-context-root*/`opensso/samples/idff`. Open `index.html` for more information.

CHAPTER 7

Implementing WS-Federation

At one time, federation was implemented using the Liberty Alliance Project Identity Federation Framework (Liberty ID-FF). But federation standards now include SAML v1.x and SAML v2 as well as WS-Federation. Although the protocol are interoperable using OpenSSO Enterprise, they are not related. This chapter contains the following sections on WS-Federation.

Accessing the WS-Federation Java Server Pages

The WS-Federation Service uses JavaServer Pages™ (JSP™) to complete its functionality. After a default installation, the JSP are located in /*path-to-context-root*/`opensso/wsfederation/jsp`. They include:

`logout.jsp`	Page is displayed after a successful logout.
`post.jsp`	The HTML form used to send the WS-Federation single sign-on responses from the identity provider to the service provider.
`realmSelection.jsp`	Page is displayed if no realm is defined.

Using the WS-Federation Packages

The following packages relate to the WS-Federation functionality in OpenSSO Enterprise. For more detailed information, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

com.sun.identity.wsfederation.plugins

This package defines the WS-Federation service provider interfaces (SPI). DefaultIDPAccountMapper.java is an implementation of this SPI.

TABLE 7–1 `com.sun.identity.wsfederation.plugins` Interfaces

Interface	Description
`IDPAccountMapper`	`IDPAccountMapper` is used on the identity provider (SAML v2 provider) side to map the local identities to the SAML v2 protocol objects. It accomplishes the reverse for some of the protocols (for example, `ManageNameIDRequest`). The default implementation, `com.sun.identity.wsfederation.plugins.DefaultIDPAccountMapper`, is used by the SAML v2 framework to retrieve the user's account federation information to construct the SAML protocol objects (for example, an Assertion) and to find out the corresponding user account for the given SAML v2 requests.
`IDPAttributeMapper`	`IDPAttributeMapper` is used to map an authenticated user's attributes to SAML v2 attributes. The SAML v2 framework may then insert the attribute information as an `AttributeStatement` in a SAML v2 assertion. The default implementation, `com.sun.identity.wsfederation.plugins.DefaultIDPAttributeMapper`, reads the configured attributes or attributes that are available through the `SSOToken` and returns the SAML v2 attributes.
`IDPAuthenticationMethodMapper`	`IDPAuthenticationMethodMapper` creates an `IDPAuthenticationTypeInfo` element based on the `RequestAuthnContext` information from the `AuthnRequest` sent by a service provider and the `AuthnContext` configuration om the identity provider side. The default implementation, `com.sun.identity.wsfederation.plugins.DefaultIDPAuthenticationMethodMapper`, will be used by the identity provider to find out the authentication mechanism and set the `AuthnContext` in the assertion.
`SPAccountMapper`	`com.sun.identity.saml.plugins.PartnerAccountMapper` is an interface that is implemented to map a partner account to a user account in OpenSSO Enterprise. Different partners would need to have different implementations of the interface. The mappings between the partner source ID and the implementation class are configured in the Partner URLs field of the SAML service. `com.sun.identity.wsfederation.plugins.DefaultADFSPartnerAccountMapper` is the default implementation.

TABLE 7–1 `com.sun.identity.wsfederation.plugins` Interfaces *(Continued)*

Interface	Description
`SPAttributeMapper`	`SPAttributeMapper` maps SAML v2 attributes to local user attributes. This mapper will be used by the service provider to read the configured map for the corresponding SAML v2 attributes and supply them to the SAML framework. The locally mapped attributes returned by the implementation of this interface will be inserted into the `SSOToken` by the SAML v2 framework. `com.sun.identity.wsfederation.plugins.DefaultSPAttributeMapper` is the default implementation.

`com.sun.identity.wsfederation.common`

This package contains utility methods and constants for WS-Federation implementations.

Executing the Multi-Protocol Hub Sample

OpenSSO Enterprise includes WS-Federation functionality in the multi-protocol hub sample. The sample is located in `/`*path-to-context-root*`/opensso/samples/multiprotocol`. Open `index.html` for more information.

CHAPTER 8

Constructing SAML Messages

Sun OpenSSO Enterprise has implemented two versions of the Security Assertion Markup Language (SAML) in OpenSSO Enterprise. This chapter contains information on these implementations.

- "SAML v2" on page 95
- "Using SAML v2 for Virtual Federation Proxy" on page 113
- "SAML v1.x" on page 124

SAML v2

The following sections include information on the implementation of SAML v2 in OpenSSO Enterprise.

- "Using the SAML v2 SDK" on page 95
- "Service Provider Interfaces" on page 97
- "JavaServer Pages" on page 104
- "SAML v2 Samples" on page 113

Using the SAML v2 SDK

The SAML v2 framework provides interfaces that can be used to construct and process assertions, requests, and responses. The SDK is designed to be pluggable although it can also be run as a standalone application (outside of an instance of OpenSSO Enterprise).

- For information on the packages in the SDK, see "Exploring the SAML v2 Packages" on page 96.
- For ways to set a customized implementation, see "Setting a Customized Class" on page 96.

Exploring the SAML v2 Packages

The SAML v2 SDK includes the following packages:

- "`com.sun.identity.saml2.assertion` Package" on page 96
- "`com.sun.identity.saml2.common` Package" on page 96
- "`com.sun.identity.saml2.plugins` Package" on page 96
- "`com.sun.identity.saml2.protocol` Package" on page 96

For more detailed information, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

`com.sun.identity.saml2.assertion` Package

This package provides interfaces to construct and process SAML v2 assertions. It also contains the `AssertionFactory`, a factory class used to obtain instances of the objects defined in the assertion schema.

`com.sun.identity.saml2.common` Package

This package provides interfaces and classes used to define common SAML v2 utilities and constants.

`com.sun.identity.saml2.plugins` Package

This package provides service provider interfaces to implement for plug-ins.

`com.sun.identity.saml2.protocol` Package

This package provides interfaces used to construct and process the SAML v2 request/response protocol. It also contains the `ProtocolFactory`, a factory class used to obtain object instances for concrete elements in the protocol schema.

Setting a Customized Class

There are two ways you can set a customized implementation class:

1. Add a customized mapper as a value for the Advanced Properties of the appropriate server using the OpenSSO Enterprise console.
   a. Login to the OpenSSO Enterprise console as the administrator.
   b. Click the Configuration tab.
   c. Click Servers & Sites and select the server.
   d. Click the Advanced tab.
   e. Click Add and enter the full interface name as the Property Name and the implemented class name as the Property Value.

      For example, `com.sun.identity.saml2.sdk.mapping.Assertion` and `com.`*ourcompany*`.saml2.AssertionImpl`, respectively.

2. Set an environment variable for the Virtual Machine for the Java™ platform (JVM™). For example, you can add the following environment variable when starting the application:

```
-Dcom.sun.identity.saml2.sdk.mapping.Assertion=com.ourcompany.saml2.AssertionImpl
```

Service Provider Interfaces

The `com.sun.identity.saml2.plugins` package provides pluggable interfaces to extend SAML v2 functionality into your remote application. The classes can be configured per provider entity. Default implementations are provided, but a customized implementation can be plugged in by modifying the corresponding attribute in the provider's extended metadata configuration file. The mappers include:

- "Account Mappers" on page 97
- "Attribute Mappers" on page 98
- "Authentication Context Mappers" on page 99
- "Assertion Query/Request Mappers" on page 102
- "Attribute Authority Mappers" on page 103

For more information, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

Account Mappers

An account mapper is used to associate a local user account with a remote user account based on the Name ID (or another specific attribute value) in the Assertion. A default account mapper has been developed for both sides of the SAML v2 interaction, service providers and identity providers.

- "`IDPAccountMapper`" on page 97
- "`SPAccountMapper`" on page 98

If implementing a custom account mapper, change the value of the provider's Account Mapper property using the OpenSSO Enterprise console.

`IDPAccountMapper`

The `IDPAccountMapper` interface is used on the identity provider side to map user accounts in cases of single sign-on and federation termination. The default implementation is provided in by `com.sun.identity.saml2.plugins.DefaultIDPAccountMapper`. During single sign-on, the `DefaultIDPAccountMapper` returns the Name Identifier to be set in an Assertion based on the entity provider's configuration; for example, the user's profile attributes can be set as the value of the Name ID using the NameID Value Map field in the console.

SPAccountMapper

The SPAccountMapper interface is used on the service provider side to map user accounts in cases of single sign-on and federation termination. The default implementation, com.sun.identity.saml2.plugins.DefaultSPAccountMapper, supports mapping based on the transient and persistent NameID attributes, and attribute federation based on properties defined in the extended metadata configuration file. The user mapping is based on information passed from the identity provider in an <AttributeStatment>.

Attribute Mappers

An attribute mapper is used to associate attribute names passed in the <AttributeStatement> of an assertion. A default attribute mapper has been developed for both participants in the SAML v2 interaction, service providers and identity providers. They are defined in the extended metadata configuration files and explained in the following sections:

- "IDPAttributeMapper" on page 98
- "SPAttributeMapper" on page 99

If implementing a custom attribute mapper, change the value of the provider's Attribute Mapper property using the OpenSSO Enterprise console.

IDPAttributeMapper

The IDPAttributeMapper interface is used by the identity provider to specify which user attributes will be included in an assertion. The default implementation, com.sun.identity.saml2.plugins.DefaultIDPAttributeMapper, retrieves attribute mappings (*SAML v2-attribute=user-attribute*) defined in the attributeMap property in the identity provider's extended metadata configuration file. It reads the value of the user attribute from the identity provider's data store, and sets this value as the <AttributeValue> of the specified SAML v2 attribute. The SAML v2 attributes and values are then included in the <AttributeStatement> of the assertion and sent to the service provider. The value of attributeMap can be changed to modify the mapper's behavior without programming. The default mapper itself can be modified to attach any identity provider user attribute with additional programming.

The identity provider can also send different AttributeStatement elements for different service providers. To support this, define an attribute mapping in the remote service provider's metadata hosted on the identity provider side. This configuration will override the attribute mapping defined on the hosted identity provider itself. (The hosted identity provider configuration serves as the default if no attribute mapping is defined in the service provider metadata.

`SPAttributeMapper`

The `SPAttributeMapper` interface is used by the service provider to map attributes received in an assertion to its local attributes. The default implementation, `com.sun.identity.saml2.plugins.DefaultSPAttributeMapper`, retrieves the attribute mappings defined in the `attributeMap` property in the service provider's extended metadata configuration file. It extracts the value of the SAML v2 attribute from the assertion and returns a key/value mapping which will be set in the user's single sign-on token. The mapper can also be customized to choose user attributes from the local service provider datastore.

Note – `*=*` is a special attribute mapping which can be defined for a service provider hosted on an instance of OpenSSO Enterprise only. (It is not valid for a remote service provider configured on the identity provider side.) It will map all the attribute names as presented in the Assertion. (It will keep the same name as in the `AttributeStatement` element. Enter this as a value of the Attribute Map property under the service provider configuration Assertion Processing tab.

Authentication Context Mappers

Authentication context refers to information added to an assertion regarding details of the technology used for the actual authentication action. For example, a service provider can request that an identity provider comply with a specific authentication method by identifying that method in an authentication request. The authentication context mapper pairs a standard SAML v2 authentication context class reference (`PasswordProtectedTransport`, for example) to a OpenSSO Enterprise authentication scheme (`module=LDAP`, for example) on the identity provider side and sets the appropriate authentication level in the user's SSO token on the service provider side. The identity provider would then deliver (with the assertion) the authentication context information in the form of an authentication context declaration added to the assertion. The process for this is described below.

1. A user accesses `spSSOInit.jsp` using the `AuthnContextClassRef` query parameter.

   For example, **`http://`***SP_host***`:`***SP_port***`/`***uri***`/spSSOInit.jsp?`**
   `metaAlias=`*SP_MetaAlias***`&idpEntityID=`***IDP_EntityID***`&AuthnContextClassRef=PasswordProtec`**
2. The `SPAuthnContextMapper` is invoked to map the value of the query parameter to a `<RequestedAuthnContext>` and an authentication level.
3. The service provider sends the `<AuthRequest>` with the `<RequestedAuthnContext>` to the identity provider.
4. The identity provider processes the `<AuthRequest>` by invoking the `IDPAuthnContextMapper` to map the incoming information to a defined authentication scheme.

Note – If there is no matching authentication scheme, an authentication error page is displayed.

5. The identity provider then redirects the user (including information regarding the authentication scheme) to the Authentication Service for authentication.

   For example, **`http://`***osso_host***`:`***osso_port***`/`***uri***`/UI/Login?module=LDAP`** redirects to the LDAP authentication module.
6. After successful authentication, the user is redirected back to the identity provider for construction of a response based on the mapped authentication class reference.
7. The identity provider then returns the user to the assertion consumer on the service provider side.
8. After validating the response, the service provider creates a single sign-on token carrying the authentication level defined in the previous step.

A default authentication context mapper has been developed for both sides of the SAML v2 interaction. Details about the mappers are in the following sections:

- "`IDPAuthnContextMapper`" on page 100
- "`SPAuthnContextMapper`" on page 101

If implementing a custom authentication context mapper, change the value of the provider's Authentication Context Mapper property using the OpenSSO Enterprise console.

IDPAuthnContextMapper

The `IDPAuthnContextMapper` is configured for the identity provider and maps incoming authentication requests from the service provider to a OpenSSO Enterprise authentication scheme (user, role, module, level or service-based authentication), returning a response containing the authentication status to the service provider. The following attributes in the identity provider extended metadata are used by the `IDPAuthnContextMapper`:

- The `idpAuthncontextMapper` property specifies the mapper implementation.
- The `idpAuthncontextClassrefMapping` property specifies the mapping between a standard SAMLv2 authentication context class reference and an OpenSSO Enterprise authentication scheme. It takes a value in the following format:

```
authnContextClassRef | authlevel | authnType=authnValue | authnType=authnValue | ... [|default]
```

For example,
`urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport|3|module=LDAP|default`
maps the SAMLv2 `PasswordProtectedTransport` class reference to the OpenSSO Enterprise `LDAP` authentication module.

`SPAuthnContextMapper`

The `SPAuthnContextMapper` is configured for the service provider and maps the parameters in incoming HTTP requests to an authentication context. It creates a `<RequestedAuthnContext>` element based on the query parameters and attributes configured in the extended metadata of the service provider. The `<RequestedAuthnContext>` element is then included in the `<AuthnRequest>` element sent from the service provider to the identity provider for authentication. The `SPAuthnContextMapper` also maps the authentication context on the identity provider side to the authentication level set as a property of the user's single sign-on token. The following query parameters can be set in the URL when accessing `spSSOInit.jsp`:

- `AuthnContextClassRef` or `AuthnContextDeclRef`: These properties specify one or more URI references identifying the provider's supported authentication context classes. If a value is not specified, the default is `urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport`.
- *AuthLevel*: This parameter specifies the authentication level of the authentication context being used for authentication.
- *AuthComparison*: This parameter specifies the method of comparison used to evaluate the requested context classes or statements. Accepted values include:
  - *exact* where the authentication context statement in the assertion must be the exact match of, at least, one of the authentication contexts specified.
  - *minimum* where the authentication context statement in the assertion must be, at least, as strong (as deemed by the identity provider) one of the authentication contexts specified.
  - *maximum* where the authentication context statement in the assertion must be no stronger than any of the authentication contexts specified.
  - *better* where the authentication context statement in the assertion must be stronger than any of the authentication contexts specified.

  If the element is not specified, the default value is *exact*.

An example URL might be **`http://`***SP_host***`:`***SP_port***`/`***uri***`/spSSOInit.jsp?`**
`metaAlias=`*SP_MetaAlias***`&idpEntityID=`***IDP_EntityID***`&AuthnContextClassRef=PasswordProtected`**

The following attributes in the service provider extended metadata are used by the `SPAuthnContextMapper`:

- The `spAuthncontextMapper` property specifies the name of the service provider mapper implementation.
- The `spAuthncontextClassrefMapping` property specifies the map of authentication context class reference and authentication level in the following format:

  ```
  authnContextClassRef | authlevel [| default]
  ```

- The `spAuthncontextComparisonType` property is optional and specifies the method of comparison used to evaluate the requested context classes or statements. Accepted values include:
  - *exact* where the authentication context statement in the assertion must be the exact match of, at least, one of the authentication contexts specified.
  - *minimum* where the authentication context statement in the assertion must be, at least, as strong (as deemed by the identity provider) one of the authentication contexts specified.
  - *maximum* where the authentication context statement in the assertion must be no stronger than any of the authentication contexts specified.
  - *better* where the authentication context statement in the assertion must be stronger than any of the authentication contexts specified.

  If the element is not specified, the default value is *exact*.

Assertion Query/Request Mappers

The Assertion Query/Request profile specifies a means for requesting existing assertions using a unique identifier. The requester initiates the profile by sending an assertion request, referenced by an identifier, to a SAML v2 authority. The SAML v2 authority processes the request, checks the assertion cache for the identifier, and issues a response to the requester. An assertion mapper is used by he SAML v2 authority to process assertion ID requests. The `com.sun.identity.saml2.plugins.AssertionIDRequestMapper` class is the default implementation for the `com.sun.identity.saml2.plugins.AssertionIDRequestMapper` SPI. The SPI is used to validate the assertion request on the server side. The Assertion will be returned to the client only after the validation passed.

To define a customized mapper, change the value of the `assertionIDRequestMapper` property in the extended metadata of the provider acting as SAML v2 attribute authority or authentication authority. To send a request for an assertion from a provider, use either of the methods of `com.sun.identity.saml2.profile.AssertionIDRequestUtil` as below.

```
public static Response sendAssertionIDRequest(
AssertionIDRequest assertionIDRequest,
String samlAuthorityEntityID,
String role,
String realm,
String binding)
throws SAML2Exception;

public static Assertion sendAssertionIDRequestURI(
String assertionID,
String samlAuthorityEntityID,
String role,
String realm)
throws SAML2Exception;
```

To construct an assertion request object, use `com.sun.identity.saml2.assertion.*` and `com.sun.identity.saml2.protocol.*`.

Attribute Authority Mappers

The Assertion Query/Request profile specifies a means for requesting attributes (and the corresponding values) from a specific identity profile. A successful response is the return of an assertion containing the requested information. The identity provider acting as the attribute authority uses the `com.sun.identity.saml2.plugins.AttributeAuthorityMapper` to process queries. This default implementation uses the attribute map table configured in the identity provider's extended metadata; this table maps the requested SAML v2 attributes to the user profile attributes in the identity data store. (If an attribute map is not configured, no attributes will be returned.)

To set OpenSSO Enterprise to use a customized attribute mapper implementation, modify the values of the `default_attributeAuthorityMapper` and the `x509Subject_attributeAuthorityMapper` properties in the extended metadata of the provider defined as the attribute authority. The `default_attributeAuthorityMapper` value is used for a standard attribute queries and the `x509Subject_attributeAuthorityMapper` value is used for attribute queries with an X509 subject, mapping the X509 subject to a user by searching the identity data store for a specified attribute. (The specified attribute is defined as the value of the `x509SubjectDataStoreAttrName` property in the identity provider extended metadata of the attribute authority.) If the user has the specified attribute and the attribute's value is the same as that of the X509 subject in the attribute query, the user will be used.

Only SOAP binding is supported and signing is required so make sure the Signing Certificate Alias attribute of the providers acting as the attribute requester and the attribute authority is configured. To send an attribute query from the requester use the method of `com.sun.identity.saml2.profile.AttributeQueryUtil` as follows.

```
public static Response sendAttributeQuery(
AttributeQuery attrQuery,
String attrAuthorityEntityID,
String realm,
String attrQueryProfile,
String attrProfile,
String binding)
throws SAML2Exception;
```

To construct an attribute query object, use `com.sun.identity.saml2.assertion.*` and `com.sun.identity.saml2.protocol.*`.

Service Provider Adapter

A service provider adapter allows the developer to plug-in application specific logic before and/or after single sign-on, single logout, termination and new name identifier process. The `SAML2ServiceProviderAdapter` abstract class provides methods that could be extended to perform user specific logics during SAML v2 protocol processing on the Service Provider side. The implementation class could be configured on a per service provider basis in the extended metadata configuration.

Note – A singleton instance of this `SAML2ServiceProviderAdapter` class will be used per service provider during runtime, so make sure implementation of the methods are thread safe.

JavaServer Pages

JavaServer Pages (JSP) are HTML files that contain additional code to generate dynamic content. More specifically, they contain HTML code to display static text and graphics, as well as application code to generate information. When the page is displayed in a web browser, it will contain both the static HTML content and dynamic content retrieved via the application code. The SAML v2 framework contains JSP that can initiate SAML v2 interactions. After installation, these pages can be accessed using the following URL format:

`http(s)://`*host*`:`*port*`/`*uri*`/`**`saml2/jsp/`***jsp-page-name*`?`**`metaAlias=`***xxx*`&`...

The JSP are collected in the `/`*path-to-context-root*`/`*uri*`/saml2/jsp` directory. The following sections contain descriptions of, and uses for, the different JSP.

- "Default Display Page" on page 104
- "Export Metadata Page" on page 105
- "Fedlet Pages" on page 105
- "Assertion Consumer Page" on page 105
- "Single Sign-on Pages" on page 106
- "Name Identifier Pages" on page 108
- "Single Logout Pages" on page 110

Caution – The following JSP used for the Virtual Federation Proxy cannot be modified:

- `SA_IDP.jsp`
- `SA_SP.jsp`
- `saeerror.jsp`

Default Display Page

`default.jsp` is the default display page for the SAML v2 framework. After a successful SAML v2 operation (single sign-on, single logout, or federation termination), a page is displayed. This page, generally the originally requested resource, is specified in the initiating request using the

`<RelayState>` element. If a `<RelayState>` element is not specified, the value of the `<defaultRelayState>` property in the extended metadata configuration is displayed. If a `<defaultRelayState>` is not specified, this `default.jsp` is used. `default.jsp` can take in a message to display, for example, upon a successful authentication. The page can also be modified to add additional functionality.

Caution – When the value of `<RelayState>` or `<defaultRelayState>` contains special characters (such as &), it must be URL-encoded.

Export Metadata Page

This page is used to export standard entity metadata. The supported query parameters are:

- The role of the entity defined as `sp`, `idp` or `any`.
- The realm to which the entity belongs.
- The identifier of the entity to be exported.

If no query parameter is specified, the page will attempt to export metadata in the following order:

1. The first hosted service provider under the root realm.
2. The first hosted identity provider under root realm.
3. If there is none of the above, an error message will be displayed.

Fedlet Pages

`fedletSSOInit.jsp` initiates single sign-on at the Fedlet side. (It is not designed to be used by a full service provider.) A list of query parameters for use with this page are defined in the page itself. `fedletSampleApp.jsp` is the sample page and should not be modified.

`fedletSSOInit.jsp` initiates single sign-on at the Fedlet side (note:).

Assertion Consumer Page

The `spAssertionConsumer.jsp` processes the responses that a service provider receives from an identity provider. When a service provider wants to authenticate a user, it sends an authentication request to an identity provider. The `AuthnRequest` asks that the identity provider return a `Response` containing one or more assertions. The `spAssertionConsumer.jsp` receives and parses the `Response` (or an artifact representing it). The endpoint for this JSP is *protocol*`://`*host*`:`*port*`/`*service-deploy-uri*`/Consumer`. Some ways in which the `spAssertionConsumer.jsp` can be customized include:

- The `localLoginUrl` parameter in the `spAssertionConsumer.jsp` retrieves the value of the `localAuthUrl` property in the service provider's extended metadata configuration. The value of `localAuthUrl` points to the local login page on the service provider side. If `localAuthUrl` is not defined, the login URL is calculated using the Assertion Consumer

Service URL defined in the service provider's standard metadata configuration. Changing the `localLoginUrl` parameter value in `spAssertionConsumer.jsp` is another way to define the service provider's local login URL.

- After a successful single sign-on and before the final protected resource (defined in the `<RelayState>` element) is accessed, the user may be directed to an intermediate URL, if one is configured as the value of the `intermediateUrl` property in the service provider's extended metadata configuration file. For example, this intermediate URL might be a successful account creation page after the auto-creation of a user account. The `redirectUrl` in `spAssertionConsumer.jsp` can be modified to override the `intermediateUrl` value.

Single Sign-on Pages

The single sign-on JSP are used to initiate single sign-on and, parse authentication requests, and generate responses. These include:

- "`idpSSOFederate.jsp`" on page 106
- "`idpSSOInit.jsp`" on page 106
- "`spSSOInit.jsp`" on page 107

`idpSSOFederate.jsp`

`idpSSOFederate.jsp` works on the identity provider side to receive and parse authentication requests from the service provider and generate a `Response` containing an assertion. The endpoint for this JSP is *protocol*`://`*host*`:`*port*`/`*service-deploy-uri*`/idpSSOFederate`. `idpSSOFederate.jsp` takes the following parameters:

- `SAMLRequest`: This required parameter takes as a value the XML blob that contains the `AuthnRequest`.
- `metaAlias`: This optional parameter takes as a value the `metaAlias` set in the identity provider's extended metadata configuration file.
- `RelayState`: This optional parameter takes as a value the target URL of the request.

`idpSSOInit.jsp`

`idpSSOInit.jsp` initiates single sign-on from the identity provider side (also referred to as *unsolicited response*). For example, a user requests access to a resource. On receiving this request for access, `idpSSOInit.jsp` looks for a cached assertion which, if present, is sent to the service provider in an unsolicited `<Response>`. If no assertion is found, `idpSSOInit.jsp` verifies that the following required parameters are defined:

- `metaAlias`: This parameter takes as a value the `metaAlias` set in the identity provider's extended metadata configuration file. If the `metaAlias` attribute is not present, an error is returned.
- `spEntityID`: The entity identifier of the service provider to which the response is sent.

If defined, the unsolicited `Response` is created and sent to the service provider. If not, an error is returned. The endpoint for this JSP is *protocol*`://`*host*`:`*port*`/`*service-deploy-uri*`/idpssoinit`. The following optional parameters can also be passed to `idpSSOInit.jsp`:

- `RelayState`: The target URL of the request.
- `NameIDFormat`: The currently supported name identifier formats: *persistent* or *transient*.
- `binding`: A URI suffix identifying the protocol binding to use when sending the `Response`. The supported values are:
  - `HTTP-Artifact`
  - `HTTP-POST`

`spSSOInit.jsp`

`spSSOInit.jsp` is used to initiate single sign-on from the service provider side. On receiving a request for access, `spSSOInit.jsp` verifies that the following required parameters are defined:

- `metaAlias`: This parameter takes as a value the `metaAlias` set in the identity provider's extended metadata configuration file. If the `metaAlias` attribute is not present, an error is returned.
- `idpEntityID`: The entity identifier of the identity provider to which the request is sent. If `idpEntityID` is not provided, the request is redirected to the SAML v2 IDP Discovery Service to get the user's preferred identity provider. In the event that more then one identity provider is returned, the last one in the list is chosen. If `idpEntityID` cannot be retrieved using either of these methods, an error is returned.

If defined, the `Request` is created and sent to the identity provider. If not, an error is returned. The endpoint for this JSP is *protocol*`://`*host*`:`*port*`/`*service-deploy-uri*`/spssoinit`. The following optional parameters can also be passed to `spSSOInit.jsp`:

- `RelayState`: The target URL of the request.
- `NameIDFormat`: The currently supported name identifier formats: *persistent* or *transient*.
- `binding`: A URI suffix identifying the protocol binding to use when sending the `Response`. The supported values are:
  - `HTTP-Artifact`
  - `HTTP-POST`
- `AssertionConsumerServiceIndex`: An integer identifying the location to which the `Response` message should be returned to the requester. requester. It applies to profiles in which the requester is different from the presenter, such as the Web Browser SSO profile.
- `AttributeConsumingServiceIndex`: An integer indirectly specifying information (associated with the requester) describing the SAML attributes the requester desires or requires to be supplied.
- `isPassive`: Takes a value of `true` or `false` with `true` indicating the identity provider should authenticate passively.

- ForceAuthN: Takes a value of true indicating that the identity provider must force authentication or false indicating that the identity provider can reuse existing security contexts.
- AllowCreate: Takes a value of true indicating that the identity provider is allowed to created a new identifier for the principal if it does not exist or false.
- Destination: A URI indicating the address to which the request has been sent.
- AuthnContextClassRef: Specifies a URI reference identifying an authentication context class that describes the declaration that follows. Multiple references can be pipe-separated.
- AuthnContextDeclRef: Specifies a URI reference to an authentication context declaration. Multiple references can be pipe-separated.
- AuthComparison: The comparison method used to evaluate the requested context classes or statements. Accepted values include: *minimum*, *maximum* or *better*.
- Consent: Indicates whether or not (and under what conditions) consent has been obtained from a principal in the sending of this request.

Note – Consent is not supported in this release.

To pass parameters to specify RequestedAuthnContext use:

1. AuthLevel
2. AuthnContextClassRef
3. sunamcompositeadvice

Name Identifier Pages

The various *ManageNameID* (MNI) JSP provide a way to change account identifiers or terminate mappings between identity provider accounts and service provider accounts. For example, after establishing a name identifier for use when referring to a principal, the identity provider may want to change its value and/or format. Additionally, an identity provider might want to indicate that a name identifier will no longer be used to refer to the principal. The identity provider will notify service providers of the change by sending them a ManageNameIDRequest. A service provider also uses this message type to register or change the SPProvidedID value (included when the underlying name identifier is used to communicate with it) or to terminate the use of a name identifier between itself and the identity provider.

- "idpMNIPOST.jsp" on page 109
- "idpMNIRequestInit.jsp" on page 109
- "idpMNIRedirect.jsp" on page 109
- "spMNIPOST.jsp" on page 109
- "spMNIRequestInit.jsp" on page 110
- "spMNIRedirect.jsp" on page 110

`idpMNIPOST.jsp`

`idpMNIPOST.jsp` processes the `ManageNameIDRequest` from an identity provider using HTTP Redirect binding. There are no required parameters.

`idpMNIRequestInit.jsp`

`idpMNIRequestInit.jsp` initiates the `ManageNameIDRequest` at the identity provider by user request. The endpoint for this JSP is *protocol*`://`*host*`:`*port*`/`*service-deploy-uri*`/IDPMniInit`. It takes the following required parameters:

- `metaAlias`: The value of the `metaAlias` property set in the identity provider's extended metadata configuration file. If the `metaAlias` attribute is not present, an error is returned.
- `spEntityID`: The entity identifier of the service provider to which the response is sent.
- `requestType`: The type of `ManageNameIDRequest`. Accepted values include `Terminate` and `NewID`.

Some of the other optional parameters are :

- `binding`: A URI specifying the protocol binding to use for the `<Request>`. The supported values are:
  - `urn:oasis:names:tc:SAML:2.0:bindings:SOAP`
  - `urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect`
  - `urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST`
- `RelayState`: The target URL of the request

`idpMNIRedirect.jsp`

`idpMNIRedirect.jsp` processes the `ManageNameIDRequest` and the `ManageNameIDResponse` received from the service provider using HTTP-Redirect. The endpoint for this JSP is *protocol*`://`*host*`:`*port*`/`*service-deploy-uri*`/IDPMniRedirect`. It takes the following required parameters:

- `SAMLRequest`: The `ManageNameIDRequest` from the service provider.
- `SAMLResponse`: The `ManageNameIDResponse` from the service provider.

Optionally, it can also take the `RelayState` parameter which specifies the target URL of the request.

`spMNIPOST.jsp`

`spMNIPOST.jsp` processes the `ManageNameIDRequest` from a service provider using HTTP Redirect binding. There are no required parameters.

spMNIRequestInit.jsp

spMNIRequestInit.jsp initiates the ManageNameIDRequest at the service provider by user request. The endpoint for this JSP is *protocol*://*host*:*port*/*service-deploy-uri*/SPMniInit. It takes the following required parameters:

- metaAlias: This parameter takes as a value the metaAlias set in the identity provider's extended metadata configuration file. If the metaAlias attribute is not present, an error is returned.
- idpEntityID: The entity identifier of the identity provider to which the request is sent.
- requestType: The type of ManageNameIDRequest. Accepted values include Terminate and NewID.

Some of the other optional parameters are :

- binding: A URI specifying the protocol binding to use for the Request. The supported values are:
  - urn:oasis:names:tc:SAML:2.0:bindings:SOAP
  - urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect
  - urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST
- RelayState: The target URL of the request.

spMNIRedirect.jsp

spMNIRedirect.jsp processes the ManageNameIDRequest and the <ManageNameIDResponse> received from the identity provider using HTTP-Redirect. The endpoint for this JSP is *protocol*://*host*:*port*/*service-deploy-uri*/SPMniRedirect. It takes the following required parameters:

- SAMLRequest: The ManageNameIDRequest from the identity provider.
- SAMLResponse: The ManageNameIDResponse from the identity provider.

Optionally, it can also take the RelayState parameter which specifies the target URL of the request.

Single Logout Pages

The single logout JSP provides the means by which all sessions authenticated by a particular identity provider are near-simultaneously terminated. The single logout protocol is used either when a user logs out from a participant service provider or when the principal logs out directly from the identity provider.

- "idpSingleLogoutPOST.jsp" on page 111
- "idpSingleLogoutInit.jsp" on page 111
- "idpSingleLogoutRedirect.jsp" on page 112
- "spSingleLogoutPOST.jsp" on page 112

- "`spSingleLogoutInit.jsp`" on page 112
- "`spSingleLogoutRedirect.jsp`" on page 113

`idpSingleLogoutPOST.jsp`

`idpSingleLogoutPOST.jsp` can do either of the following:

- Receives a Logout Request from an identity provider and sends a Logout Response to a service provider.
- Receives a Logout Response from the service provider.

There are no required parameters.

`idpSingleLogoutInit.jsp`

`idpSingleLogoutInit.jsp` initiates a `LogoutRequest` at the identity provider by user request. The endpoint for this JSP is *protocol*`://`*host*`:`*port*`/`*service-deploy-uri*`/IDPSloInit`. There are no required parameters. Optional parameters include:

- `RelayState`: The target URL after single logout.
- `binding`: A URI specifying the protocol binding to use for the `<Request>`. The supported values are:
  - `urn:oasis:names:tc:SAML:2.0:bindings:SOAP`
  - `urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect`
- `Destination`: A URI indicating the address to which the request has been sent.
- `Consent`: Indicates whether or not (and under what conditions) consent has been obtained from a principal in the sending of this request.

  Note – `Consent` is not supported in this release.

- `Extension`: Specifies permitted extensions as a list of string objects.

  Note – `Extension` is not supported in this release.

- `logoutAll`: Specifies that the identity provider send log out requests to all service providers without a session index. It will logout all sessions belonging to the user.

`idpSingleLogoutRedirect.jsp`

`idpSingleLogoutRedirect.jsp` processes the `LogoutRequest` and the `LogoutResponse` received from the service provider using HTTP-Redirect. The endpoint for this JSP is *protocol*`://`*host*`:`*port*`/`*service-deploy-uri*`/IDPSloRedirect`. It takes the following required parameters:

- `SAMLRequest`: The `LogoutRequest` from the service provider.
- `SAMLResponse`: The `LogoutResponse` from the service provider.

Optionally, it can also take the `RelayState` parameter which specifies the target URL of the request.

`spSingleLogoutPOST.jsp`

`spSingleLogoutPOST.jsp` can do either of the following:

- Receives a Logout Request from a service provider and sends a Logout Response to an identity provider.
- Receives a Logout Response from the identity provider.

Required parameters for the first option are `RelayState` (the target URL for a successful single logout) and `SAMLRequest` (the Logout Request). For the second option it is `SAMLResponse` (the Logout Response).

`spSingleLogoutInit.jsp`

`spSingleLogoutInit.jsp` initiates a `LogoutRequest` at the identity provider by user request. The endpoint for this JSP is *protocol*`://`*host*`:`*port*`/`*service-deploy-uri*`/SPSloInit`. There are no required parameters. Optional parameters include:

- `RelayState`: The target URL after single logout.
- `binding`: A URI specifying the protocol binding to use for the `<Request>`. The supported values are:
  - `urn:oasis:names:tc:SAML:2.0:bindings:SOAP`
  - `urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect`
- `Destination`: A URI indicating the address to which the request has been sent.
- `Consent`: Indicates whether or not (and under what conditions) consent has been obtained from a principal in the sending of this request.

  Note – `Consent` is not supported in this release.

- `Extension`: Specifies permitted extensions as a list of string objects.

Note – `Extension` is not supported in this release.

`spSingleLogoutRedirect.jsp`

`spSingleLogoutRedirect.jsp` processes the `LogoutRequest` and the `LogoutResponse` received from the identity provider using HTTP-Redirect. The endpoint for this JSP is *protocol*`://`*host*`:`*port*`/`*service-deploy-uri*`/SPSloRedirect`. It takes the following required parameters:

- `SAMLRequest`: The `LogoutRequest` from the identity provider.
- `SAMLResponse`: The `LogoutResponse` from the identity provider.

Optionally, it can also take the `RelayState` parameter which specifies the target URL of the request.

SAML v2 Samples

The following SAML v2 samples can be used for testing purposes.

- useCasedemo is a sample that illustrates the following SAML v2 use cases.
  - IDP initiated Single Sign On
  - SP initiated Single Sign On
  - IDP initiated Single Log out
  - SP initiated Single Log out
  - IDP initiated Federation
  - SP initiated Federation
  - IDP initiated Federation Termination
  - SP initiated Federation Termination
- sae is a sample that illustrates the general use cases of the Virtual Federation Proxy (also referred to as Secure Attribute Exchange). See "Using SAML v2 for Virtual Federation Proxy" on page 113 for more information.

Using SAML v2 for Virtual Federation Proxy

Secure Attribute Exchange (also referred to as Virtual Federation Proxy) provides a mechanism for one application to communicate identity information to a second application in a different domain. In essence, Virtual Federation Proxy (VFP) provides a secure gateway that enables legacy applications to communicate user attributes used for authentication without having to deal specifically with federation protocols and processing. A VFP interaction allows:

- Identity provider applications to push user authentication, profile and transaction information to a local instance of OpenSSO Enterprise. OpenSSO Enterprise then passes the data to a remote instance of OpenSSO Enterprise at the service provider using federation protocols.
- Service provider applications to consume the received information.

Note – The scope of the implementation of VFP is currently limited to SAML v2 based single sign-on. It uses the SAMLv2-based protocols (based on the HTTP GET and POST methods as well as URL redirects) to transfer identity data between the communicating entities. The client API (which includes Java and .NET interfaces) run independently of OpenSSO Enterprise and are used to enable existing applications, allowing them to handle SAML v2 interactions.

VFP functionality can be found in three places:

- `deployable-war/opensso.war` on the OpenSSO Enterprise side.
- `libraries/dll/openssosae.dll` for client applications using the OpenSSO Enterprise NET API.
- `libraries/jars/openssoclientsdk.jar` for client applications using the OpenSSO Enterprise Java API.

The following sections contain more information on Virtual Federation Proxy.

How Virtual Federation Proxy Works

The components of a secure attribute exchange are listed and illustrated below.

- Legacy identity provider application (blue IDP)
- Service provider application (blue SP)
- Independent instances of OpenSSO on both the identity provider and the service provider sides (green)
- A user agent

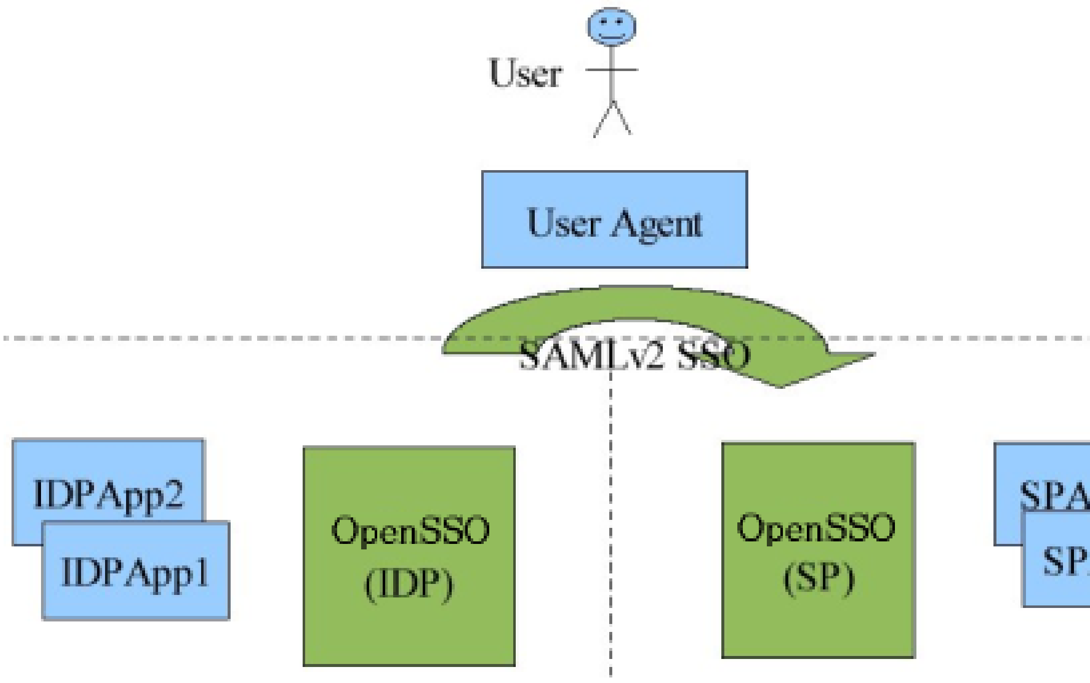
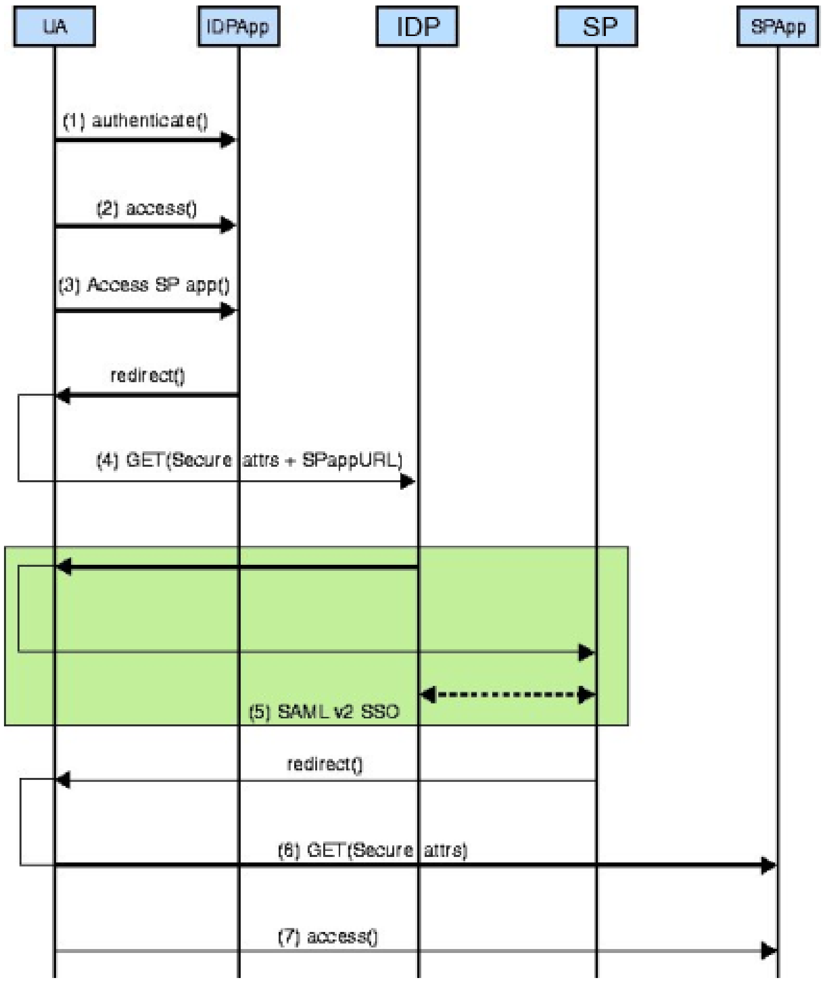


FIGURE 8–1 A Secure Attribute Exchange Using SAML v2

The following graphic illustrates the process behind a secure attribute exchange interaction. Details are below the illustration.



1. **A user authenticates.**

   This may be done by the identity provider application or it may be delegated to an authentication authority.
2. **The authenticated user uses the identity provider application and, at some point, accesses a link representing a service provided by an application in a different domain.**
3. **The identity provider application assembles the appropriate user attributes (authentication and user profile data), encodes and signs it using the API, and posts the secure data to the local instance of OpenSSO Enterprise.**

   The `com.sun.identity.sae.api.SecureAttrs` class is provided by OpenSSO Enterprise and carries the user identifier and the service provider destination.

4. **The SAE authentication module on the instance of OpenSSO Enterprise local to the identity provider verifies the authenticity of the attributes also using the SAE API, and initiates the appropriate SAML v2 single sign-on protocol to send the attributes to the instance of OpenSSO Enterprise local to the service provider being accessed.**
5. **The instance of OpenSSO Enterprise local to the service provider secures the user attributes, and sends them to the service provider application.**

   The service provider application uses interfaces supplied by OpenSSO Enterprise to verify the authenticity of the attributes.
6. **The service provider application provides or denies the service to the user based on the attributes received.**

Note – It is not mandatory for the service provider end of the process to implement VFP. Since the attributes are carried in a SAML v2 assertion, the service provider could choose another way to invoke the requested application. For example, the service provider can use standard SAML v2 protocols to invoke a SAML v2-compliant service provider that does not implement SAE. The `RelayState` element as defined in the SAML v2 specification can be used to redirect to the local service provider application.

Use Cases

The following sections contain information on applicable use cases for SAE.

- "Authentication at Identity Provider" on page 117
- "Secure Attribute Exchange at Identity Provider" on page 117
- "Secure Attribute Exchange at Service Provider" on page 118
- "Global Single Logout" on page 118

Authentication at Identity Provider

When a user is already authenticated in an enterprise, the legacy identity provider application sends a secure HTTP GET/POST message to OpenSSO Enterprise asserting the identity of the user. OpenSSO Enterprise verifies the authenticity of the message and establishes a session for the authenticated user. You can use VFP to transfer the user's authentication information to the local instance of OpenSSO Enterprise in order to create a session.

Secure Attribute Exchange at Identity Provider

When a user is already authenticated by, and attempts access to, a legacy identity provider application, the legacy application sends a secure HTTP POST message to the local instance of OpenSSO Enterprise asserting the user's identity, and containing a set of attribute/value pairs related to the user (for example, data from the persistent store representing certain transactional states in the application). OpenSSO Enterprise verifies the authenticity of the message, establishes a session for the authenticated user, and populates the session with the user attributes.

Secure Attribute Exchange at Service Provider

When a user is already authenticated by the instance of OpenSSO Enterprise at the identity provider and invokes an identity provider application that calls for redirection to a service provider, the identity provider invokes one of the previous use cases and encodes a SAML v2 single sign-on URL as a part of the request. The identity provider instance of OpenSSO Enterprise then initiates SAML v2 single sign-on with the instance of OpenSSO Enterprise at the service provider. The service provider's instance of OpenSSO Enterprise then verifies the SAML v2 assertion and included attributes, and redirects to the service provider application, securely transferring the user attributes via a secure HTTP POST message. The service provider application consumes the attributes, establishes a session, and offers the service to the user.

Global Single Logout

When a user is already authenticated and has established, for example, single sign-on with the instance of OpenSSO Enterprise at the service provider, the user might click on a Global Logout link. The identity provider will then invalidate its local session (if created) and executes SAML v2 single log out by invoking a provided OpenSSO Enterprise URL. The identity provider terminates the session on both provider instances of OpenSSO Enterprise.

Note – An identity provider side application can initiate single logout by sending `sun.cmd=logout` attributes via an SAE interaction to a local instance of OpenSSO Enterprise acting as the identity provider. In turn, this instance will execute SAML v2 single logout based on the current session.

Securing Virtual Federation Proxy

VFP provides two ways to secure identity attributes between an instance of OpenSSO Enterprise and an application:

- Symmetric involves the use of a shared secret key known only to the participants in the communication. The key is agreed upon beforehand and will be used to encrypt and decrypt the message.
- Asymmetric uses two separate keys for encryption and the corresponding decryption - one public and one private. The information is encrypted with a public key known to all and decrypted, by the recipient only, using a private key to which no one else has access. This process is known as a *public key infrastructure*. On the identity provider side, the public key must be added to the OpenSSO Enterprise keystore. The private key must be stored in a protected keystore (such as a Hardware Security Module) for access by the identity provider

application. On the service provider side, the private key must be added to the OpenSSO Enterprise keystore, and the public key stored in a keystore, local to the service provider application.

Both mechanisms result in an encrypted string (referred to as a *cryptostring*) generated for the asserted attributes. The symmetric cryptostring is a SHA-1 hash of the attributes. The asymmetric cryptostring is a digital signature of the attributes.

Note – As each pairing of application to OpenSSO Enterprise instance is independent, different applications involved can use different security methods.

Preparing to Use Virtual Federation Proxy

Before configuring and using the VFP, you will need to make some decisions regarding security, applicable keys, and applications. This section lists what you will need to do before configuring for VFP.

Note – Because OpenSSO Enterprise currently uses SAML v2 for its implementation of SAE, you should familiarize yourself with SAML v2 concepts by running the `useCaseDemo` SAML v2 sample included with OpenSSO Enterprise.

---

1. Establish trust between the application(s) and the instance of OpenSSO Enterprise on the identity provider side.

   Decide the application(s) on the identity provider side that will use SAE to push identity attributes to the local instance of OpenSSO Enterprise. You will need values for the following:

Application Name	This is used for easy identification and can be any string. Use of the application's URL is recommended.
CryptoType	Can be Symmetric or Asymmetric.
Shared Secret or Private and Public Keys	You need the shared secret if using Symmetric, and the private and public keys if using Asymmetric.

   Tip – Multiple applications can share the same application name only if they also share the same shared secret or key.

   ---

2. Establish trust between the application(s) and the instance of OpenSSO Enterprise on the service provider side.

Decide the applications on the service provider side that will receive the identity attributes from the local instance of OpenSSO Enterprise using SAE. You will need the following:

Application Name	This is used for easy identification and can be any string. Use of the application's URL is recommended because the default implementation of the SAE on the service provider side uses a prefix string match from the requested application URL to determine the parameters used to secure the communication.
CryptoType	Can be Symmetric or Asymmetric.
Shared Secret or Private and Public Keys	You need the shared secret if using Symmetric, and the private and public keys if using Asymmetric. If Asymmetric is chosen, use the same keys defined when the SAML v2 service provider was configured as an OpenSSO Enterprise service provider. You can find these keys in the service provider's metadata.

Tip – Multiple applications can share the same application name only if they also share the same shared secret or key.

---

3. **OPTIONAL:** The following steps are specific to using SAML v2 and auto-federation.

   a. Decide which identity attributes you want transferred as part of the SAML v2 single sign-on interaction.

      We choose the `branch` and `mail` attributes.

      Caution – If any attribute needs to be supplied from a local user data store, you must first populate the data store.

      ---

   b. Decide which attribute will be used to identify the user on the service provider side.

      In this instance, we choose the `branch` attribute for user identification.

      Note – The attribute may be one transferred in the SAML v2 assertion or it can be configured statically at the service provider.

      ---

4. Decide which URL on the service provider side will be responsible for handling logout requests from the identity provider.

   The URL will be responsible for terminating the local session state. Only one is allowed per logical service provider configured on the service provider side.

Configuring for Virtual Federation Proxy

Configuring for VFP communication involves modifications on two different installations of OpenSSO Enterprise: one that is local to the identity provider and one that is local to the service provider. The following sections assume that you have downloaded the OpenSSO Enterprise bits and deployed the application to a supported web container. You should also be ready to configure a SAML v2 provider by executing the included SAML v2 sample, by running one of the Common Tasks using the Administration Console, or by importing provider metadata using the Administration Console or `ssoadm` command line interface. The following procedures contain more information.

- "Configure the Instance of OpenSSO Enterprise Local to the Identity Provider" on page 121
- "Configure the Instance of OpenSSO Enterprise Local to the Service Provider" on page 122
- "Configure the Instance of OpenSSO Enterprise Local to the Identity Provider for the Remote Service Provider" on page 123
- "Configure the Instance of OpenSSO Enterprise Local to the Service Provider for the Remote Identity Provider" on page 123

Configure the Instance of OpenSSO Enterprise Local to the Identity Provider

The following procedure illustrates how to configure the instance of OpenSSO Enterprise local to the identity provider.

1. Update the identity provider standard metadata.
   - If you have existing identity provider standard metadata, export it using `ssoadm` and make your modifications. After updating, delete the original file and reload the modified metadata using `ssoadm`.
   - If you have not yet configured identity provider standard metadata, use `ssoadm` to generate an identity provider metadata template. After updating the template, import the modified metadata also using `ssoadm`.
2. Set up the keystore.

   If using the asymmetric cryptotype, add the public and private keys to the application's keystore. Additionally, populate the identity provider's keystore with the application's public key.
3. Update the identity provider configuration.

   a. Setup the application's security configuration as symmetric or asymmetric by defining the Per Application Security Configuration attribute under the Advanced tab of the identity provider configuration.

   Note – Use `ampassword` to encrypt the shared secret used for a symmetric configuration.

   ---

   b. **OPTIONAL:** Modify the IDP URL attribute (if you want to use an alternative or custom SAE landing URL) under the local identity provider's Advanced tab with a value specific to your identity provider instance of OpenSSO Enterprise.

Configure the Instance of OpenSSO Enterprise Local to the Service Provider

The following procedure shows how to configure the instance of OpenSSO Enterprise local to the service provider.

1. Update the service provider standard metadata.
   - If you have existing service provider standard metadata, export it using `ssoadm` and make your modifications. After updating, delete the original file and reload the modified metadata also using `ssoadm`.
   - If you have not yet configured service provider standard metadata, use `ssoadm` to generate a service provider metadata template. After updating the template, import the modified metadata also using `ssoadm`.
2. Set up the keystore.

   If using the asymmetric cryptotype, add the public and private keys to the application's keystore. Additionally, populate the identity provider's keystore with the application's public key.
3. Update the service provider extended metadata.

   a. Enable auto-federation and specify the attribute that will identify the user's identity under the Assertion Processing tab of the service provider configuration.

   b. Specify attributes from the incoming SAML v2 assertion to be used to populate the local OpenSSO Enterprise session under the Assertion Processing tab of the service provider configuration.

   c. Setup the application's security configuration as symmetric or asymmetric by defining the Per Application Security Configuration attribute under the Advanced tab of the service provider configuration.

   Note – Use `ampassword` to encrypt the shared secret used for a symmetric configuration.

   ---

d. **OPTIONAL:** Modify the SP URL attribute (if you want to use an alternative or custom SAE landing URL) under the local service provider's Advanced tab with a value specific to your identity provider instance of OpenSSO Enterprise.

e. Configure the value of the SP Logout URL attribute. The value of this attribute is the URL that will receive global logout requests

---

Note – The configured URL must have a defined symmetric or asymmetric CryptoType with corresponding shared secret and certificates established.

Configure the Instance of OpenSSO Enterprise Local to the Identity Provider for the Remote Service Provider

Both the standard and extended metadata retrieved from the remote service provider will be imported to the instance of OpenSSO Enterprise local to the identity provider.

1. Get both the remote service provider standard metadata and the remote service provider extended metadata used in Configure the Instance of OpenSSO Local to the Service Provider.
2. Modify the remote service provider extended metadata as follows:
   - Remove all shared secrets defined in the actual provider metadata file.
   - Set the hosted attribute to `0` (false) as in `<EntityConfig .. hosted="0" ....>`. This defines the entity as remote and can only be done using the actual provider metadata file.
   - Remove the value for the SP Logout URL attribute under the Advanced tab of the service provider configuration.
   - Add the following attribute and values to the Attribute Map attribute under the Assertion Processing tab.

   ```
   mail=mail
   branch=branch
   ```
3. Import both metadata files to the instance of OpenSSO Enterprise local to the identity provider.

   Use `ssoadm` the command line interface.

Configure the Instance of OpenSSO Enterprise Local to the Service Provider for the Remote Identity Provider

If the SAMLv2 sample has been executed on the instance of OpenSSO Enterprise local to the service provider, nothing else needs to be done. If metadata has been manually configured on the instance of OpenSSO Enterprise local to the service provider, do the following procedure.

1. Get the remote identity provider metadata for import to the instance of OpenSSO Enterprise local to the service provider.

   The standard metadata is the same as the one used in Configure the Instance of OpenSSO Enterprise Local to the Identity Provider.
2. Import the standard metadata to the instance of OpenSSO Enterprise local to the service provider using `ssoadm`.
3. Add the identity provider to the service provider's configured circle of trust.

Note – If using a flat file for a datastore, both the instance of OpenSSO Enterprise at the service provider and the instance at the identity provider must be restarted.

Using the Secure Attribute Exchange Sample

OpenSSO Enterprise includes a sample that can be run for testing your configurations. It is located in *container_context_root*`/opensso/samples/saml2/sae`. In the sample, auto-federation and transient name identifier, two features of SAML v2, are used. If there are no actual users on either the identity provider side or the service provider side, you need to use the following procedure to change the authentication framework to ignore user profiles for these two features to work correctly.

1. Login to OpenSSO Enterprise administration console as administrator.

   By default, this is `amadmin`.
2. Click the name of the realm you are modifying.
3. Click the Authentication tab.
4. Click Advanced Properties.
5. Select the Ignore Profile radio button under User Profile.
6. Click Save.
7. Log out of the console.

SAML v1.x

OpenSSO Enterprise contains SAML v1.x API collected in several Java packages. Administrators can use these packages to integrate the SAML v1.x functionality using XML messages into their applications and services. The API support all types of assertions and operate with OpenSSO Enterprise authorities to process external SAML v1.x requests and generate SAML v1.x responses. The packages include the following:

- “`com.sun.identity.saml` Package” on page 125

- "`com.sun.identity.saml.assertion` Package" on page 125
- "`com.sun.identity.saml.common` Package" on page 126
- "`com.sun.identity.saml.plugins` Package" on page 126
- "`com.sun.identity.saml.protocol` Package" on page 128

For more detailed information, including methods and their syntax and parameters, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

`com.sun.identity.saml` Package

This package contains the following classes.

- "`AssertionManager` Class" on page 125
- "`SAMLClient` Class" on page 125

`AssertionManager` Class

The `AssertionManager` class provides interfaces and methods to create and get assertions, authentication assertions, and assertion artifacts. This class is the connection between the SAML specification and OpenSSO Enterprise. Some of the methods include the following:

- `createAssertion` creates an assertion with an authentication statement based on an OpenSSO Enterprise SSO Token ID.
- `createAssertionArtifact` creates an artifact that references an assertion based on an OpenSSO Enterprise SSO Token ID.
- `getAssertion` returns an assertion based on the given parameter (given artifact, assertion ID, or query).

`SAMLClient` Class

The `SAMLClient` class provides methods to execute either the Web Browser Artifact Profile or the Web Browser POST Profile from within an application as opposed to a web browser. Its methods include the following:

- `getAssertionByArtifact` returns an assertion for a corresponding artifact.
- `doWebPOST` executes the Web Browser POST Profile.
- `doWebArtifact` executes the Web Browser Artifact Profile.

`com.sun.identity.saml.assertion` Package

This package contains the classes needed to create, manage, and integrate an XML assertion into an application. The following code example illustrates how to use the `Attribute` class and `getAttributeValue` method to retrieve the value of an attribute. From an assertion, call the getStatement() method to retrieve a set of statements. If a statement is an attribute statement, call the getAttribute() method to get a list of attributes. From there, call getAttributeValue() to retrieve the attribute value.

EXAMPLE 8–1 Sample Code to Obtain an Attribute Value

```
// get statement in the assertion
Set set = assertion.getStatement();
//assume there is one AttributeStatement
//should check null& instanceof
AttributeStatement statement = (AttributeStatement) set.iterator().next();
List attributes = statement.getAttribute();
// assume there is at least one Attribute
Attribute attribute = (Attribute) attributes.get(0);
List values = attribute.getAttributeValue();
```

com.sun.identity.saml.common Package

This package defines classes common to all SAML elements, including site ID, issuer name, and server host. The package also contains all SAML-related exceptions.

com.sun.identity.saml.plugins Package

The SAML v1.x framework provides service provider interfaces (SPIs), three of which have default implementations. The default implementations of these SPIs can be altered, or brand new ones written, based on the specifications of a particular customized service. The implementations are then used to integrate SAML into the custom service. Currently, the package includes the following.

- "ActionMapper Interface" on page 126
- "AttributeMapper Interface" on page 127
- "NameIdentifierMapper Interface" on page 127
- "PartnerAccountMapper Interface" on page 127
- "PartnerSiteAttributeMapper Interface" on page 127

ActionMapper Interface

ActionMapper is an interface used to obtain single sign-on information and to map partner actions to OpenSSO Enterprise authorization decisions. A default action mapper is provided if no other implementation is defined.

`AttributeMapper` Interface

`AttributeMapper` is an interface used in conjunction with an `AttributeQuery` class. When a site receives an attribute query, this mapper obtains the `SSOToken` or an assertion (containing an authentication statement) from the query. The retrieved information is used to convert the attributes in the query to the corresponding OpenSSO Enterprise attributes. A default attribute mapper is provided if no other implementation is defined.

`NameIdentifierMapper` Interface

`NameIdentifierMapper` is an interface that can be implemented by a site to map a user account to a name identifier in the subject of a SAML assertion. The implementation class is specified when configuring the site's Trusted Partners.

`PartnerAccountMapper` Interface

Caution – The `AccountMapper` interface has been deprecated. Use the `PartnerAccountMapper` interface.

The `PartnerAccountMapper` interface needs to be implemented by each partner site. The implemented class maps the partner site's user accounts to user accounts configured in OpenSSO Enterprise for purposes of single sign-on. For example, if single sign-on is configured from site A to site B, a site-specific account mapper can be developed and defined in the Trusted Partners sub-attribute of site B's Trusted Partners profile. When site B processes the assertion received, it locates the corresponding account mapper by retrieving the source ID of the originating site. The `PartnerAccountMapper` takes the whole assertion as a parameter, enabling the partner to define user account mapping based on attributes inside the assertion. The default implementation is `com.sun.identity.saml.plugin.DefaultAccountMapper`. If a site-specific account mapper is not configured, this default mapper is used.

Note – Turning on the Debug Service in the OpenSSO Enterprise configuration data store logs additional information about the account mapper, for example, the user name and organization to which the mapper has been mapped.

`PartnerSiteAttributeMapper` Interface

Caution – The `SiteAttributeMapper` interface has been deprecated. Use the `PartnerSiteAttributeMapper` interface.

The PartnerSiteAttributeMapper interface needs to be implemented by each partner site. The implemented class defines a list of attributes to be returned as elements of the AttributeStatements in an authentication assertion. By default, when OpenSSO Enterprise creates an assertion and no mapper is specified, the authentication assertion only contains authentication statements. If a partner site wants to include attribute statements, it needs to implement this mapper which would be used to obtain attributes, create the attribute statement, and insert the statement inside the assertion. To set up a PartnerSiteAttributeMapper do the following:

1. Implement a customized class based on the PartnerSiteAttributeMapper interface.

   This class will include user attributes in the SAML authentication assertion.
2. Log in to the OpenSSO Enterprise console to configure the class in the Site Attribute Mapper attribute of the Trusted Partner configuration.

com.sun.identity.saml.protocol **Package**

This package contains classes that parse the request and response XML messages used to exchange assertions and their authentication, attribute, or authorization information.

- "AuthenticationQuery Class" on page 128
- "AttributeQuery Class" on page 128
- "AuthorizationDecisionQuery Class" on page 129

AuthenticationQuery **Class**

The AuthenticationQuery class represents a query for an authentication assertion. When an identity attempts to access a trusted partner web site, a SAML 1.x request with an AuthenticationQuery inside is directed to the authority site.

The Subject of the AuthenticationQuery must contain a SubjectConfirmation element. In this element, ConfirmationMethod needs to be set to urn:com:sun:identity, and SubjectConfirmationData needs to be set to the SSOToken ID of the Subject. If the Subject contains a NameIdentifier, the value of the NameIdentifier should be the same as the one in the SSOToken.

AttributeQuery **Class**

The AttributeQuery class represents a query for an identity's attributes. When an identity attempts to access a trusted partner web site, a SAML 1.x request with an AttributeQuery is directed to the authority site.

You can develop an attribute mapper to obtain an SSOToken, or an assertion that contains an AuthenticationStatement from the query. If no attribute mapper for the querying site is defined, the DefaultAttributeMapper will be used. To use the DefaultAttributeMapper, the query should have either the SSOToken or an assertion that contains an AuthenticationStatement in the SubjectConfirmationData element. If an SSOToken is used, the ConfirmationMethod must be set to urn:com:sun:identity:. If an assertion is used, the assertion should be issued by the OpenSSO Enterprise instance processing the query or a server that is trusted by the OpenSSO Enterprise instance processing the query.

Note – In the DefaultAttributeMapper, a subject's attributes can be queried using another subject's SSOToken if the SSOToken has the privilege to retrieve the attributes.

For a query using the DefaultAttributeMapper, any matching attributes found will be returned. If no AttributeDesignator is specified in the AttributeQuery, all attributes from the services defined under the userServiceNameList in amSAML.properties will be returned. The value of the userServiceNameList property is user service names separated by a comma.

AuthorizationDecisionQuery Class

The AuthorizationDecisionQuery class represents a query about a principal's authority to access protected resources. When an identity attempts to access a trusted partner web site, a SAML request with an AuthorizationDecisionQuery is directed to the authority site.

You can develop an ActionMapper to obtain the SSOToken ID and retrieve the authentication decisions for the actions defined in the query. If no ActionMapper for the querying site is defined, the DefaultActionMapper will be used. To use the DefaultActionMapper, the query should have the SSOToken ID in the SubjectConfirmationData element of the Subject. If the SSOToken ID is used, the ConfirmationMethod must be set to urn:com:sun:identity:. If a NameIdentifier is present, the information in the SSOToken must be the same as the information in the NameIdentifier.

Note – When using web agents, the DefaultActionMapper handles actions in the namespace urn:oasis:names:tc:SAML:1.0:ghpp only. Web agents serve the policy decisions for this action namespace.

The authentication information can also be passed through the Evidence element in the query. Evidence can contain an AssertionIDReference, an assertion containing an AuthenticationStatement issued by the OpenSSO Enterprise instance processing the query, or an assertion issued by a server that is trusted by the OpenSSO Enterprise instance processing the query. The Subject in the AuthenticationStatement of the Evidence element should be the same as the one in the query.

Note – Policy conditions can be passed through `AttributeStatements` of assertion(s) inside the `Evidence` of a query. If the value of an attribute contains a TEXT node only, the condition is set as `attributeName=attributeValueString`. Otherwise, the condition is set as `attributename=attributeValueElement`.

The following example illustrates one of many ways to form an authorization decision query that will return a decision.

EXAMPLE 8–2 `AuthorizationDecisionQuery` Code Sample

```
    // testing getAssertion(authZQuery): no SC, with ni, with
    // evidence(AssertionIDRef, authN, for this ni):
        String nameQualifier = "dc=iplanet,dc=com";
        String pName = "uid=amadmin,ou=people,dc=iplanet,dc=com";
        NameIdentifier ni = new NameIdentifier(pName, nameQualifier);
        Subject subject = new Subject(ni);
        String actionNamespace = "urn:test";
        // policy should be added to this resource with these
        // actions for the subject
        Action action1 = new Action(actionNamespace, "GET");
        Action action2 = new Action(actionNamespace, "POST");
        List actions = new ArrayList();
        actions.add(action1);
        actions.add(action2);
        String resource = "http://www.sun.com:80";
        eviSet = new HashSet();
        // this assertion should contain authentication assertion for
        // this subject and should be created by a trusted server
        eviSet.add(eviAssertionIDRef3);
        evidence = new Evidence(eviSet);
        authzQuery = new AuthorizationDecisionQuery(eviSubject1, actions,
                            evidence, resource);
        try {
            assertion = am.getAssertion(authzQuery, destID);
        } catch (SAMLException e) {
            out.println("--failed. Exception:" + e);
        }
```

CHAPTER 9

Implementing Web Services

OpenSSO Enterprise contains web services that can be used to extend the functionality of your federated environment. Additionally, new web services can be developed. This chapter covers the following topics:

Developing New Web Services

Any web service that is plugged into the OpenSSO Enterprise Liberty ID-WSF framework must register a *key* and an implementation of the `com.sun.identity.liberty.ws.soapbinding.RequestHandler` interface with the SOAP Binding Service. (For example, the Liberty Personal Profile Service is registered with the key `idpp` and the class `com.sun.identity.liberty.ws.idpp.PPRequestHandler`.) The Key value becomes part of the URL for the web service's endpoint (as in *protocol*`://`*host*`:`*port*`/`*deploymenturi*`/`**`Liberty`**`/`*key*). The implemented class allows the web service to retrieve the request (containing the authenticated principal and the authenticated security mechanism along with the entire SOAP message) from the client. The web service processes the request and generates a response. This section contains the process you would use to add a new Liberty ID-WSF web service to the OpenSSO Enterprise framework. Instructions for some of these steps are beyond the scope of this guide. The process has been divided into two tasks:

- "To Host a Custom Service" on page 132

- "To Invoke the Custom Service" on page 138

▼ To Host a Custom Service

Before You Begin The XML Schema Definition (XSD) file written to define the new service is the starting point for developing the service's server-side code.

1. **Write an XML service schema for the new web service and Java classes to parse and process the XML messages.**

   The following sample schema defines a stock quote web service. The `QuoteRequest` and `QuoteResponse` elements define the parameters for the request and response that are inserted in the SOAP Body of the request and response, respectively. You will need to have `QuoteRequest.java` and `QuoteResponse.java` to parse and process the XML messages.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="urn:com:sun:liberty:sample:stockticker"
  targetNamespace="urn:com:sun:liberty:sample:stockticker">
  <xs:annotation>
      <xs:documentation>
         This is a sample stock ticker web service protocol
      </xs:documentation>
  </xs:annotation>

  <xs:element name="QuoteRequest" type="QuoteRequestType"/>
  <xs:complexType name="QuoteRequestType">
    <xs:sequence>
        <xs:element name = "ResourceID" type="xs:string" minOccurs="0"/>
        <xs:element name = "Symbol" type="xs:string" minOccours="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="PriceType">
      <xs:sequence>
          <xs:element name="Last" type="xs:integer"/>
          <xs:element name="Open" type="xs:integer"/>
          <xs:element name="DayRange" type="xs:string"/>
          <xs:element name="Change" type="xs:string"/>
          <xs:element name="PrevClose" type="xs:integer"/>
      </xs:sequence>
  </xs:complexType>

  <xs:element name="QuoteResponse" type="QuoteResponseType"/>
  <xs:complexType name="QuoteResponseType">
    <xs:sequence>
        <xs:element name="Symbol" type="xs:string"/>
```

```
            <xs:element name="Time" type="xs:dateTime"/>
            <xs:element name="Delay" type="xs:dateTime" minOccurs="0"/>
            <xs:element name="Price" type="PriceType"/>
            <xs:element name="Volume" type="xs:integer"/>
        </xs:sequence>
      </xs:complexType>

    </xs:schema>
```

2 Provide an implementation for one of the following interfaces based on the type of web service being developed.

- `com.sun.identity.liberty.ws.soapbinding.RequestHandler` for developing and deploying a general web service.
- `com.sun.identity.liberty.ws.dst.service.DSTRequestHandler` for developing and deploying an identity data service type web service based on the Liberty Alliance Project Identity Service Interface Specifications (Liberty ID-SIS).

In OpenSSO Enterprise, each web service must implement one of these interfaces to accept incoming message requests and return outgoing message responses. The following sample implements the `com.sun.identity.liberty.ws.soapbinding.RequestHandler` interface for the stock quote web service. `com.sun.identity.liberty.ws.soapbinding.Message` is the API used to construct requests and responses.

```
public class StockTickerService implements RequestHandler {
        :
         //implement business logic

         public Message processRequest(Message msg) throws
                      SOAPFaultException, Exception {
                :
                SSOToken token = (SSOToken)msg.getToken();
                List responseBody = processSOAPBody(msg.getBodies());
                :
                Message response = new Message();
                response.setBodies(responseBody);

                return response;
          }
          :
          //more business logic

}
```

3 Compile the Java source code.

Be sure to include `openfedlib.jar` in your classpath.

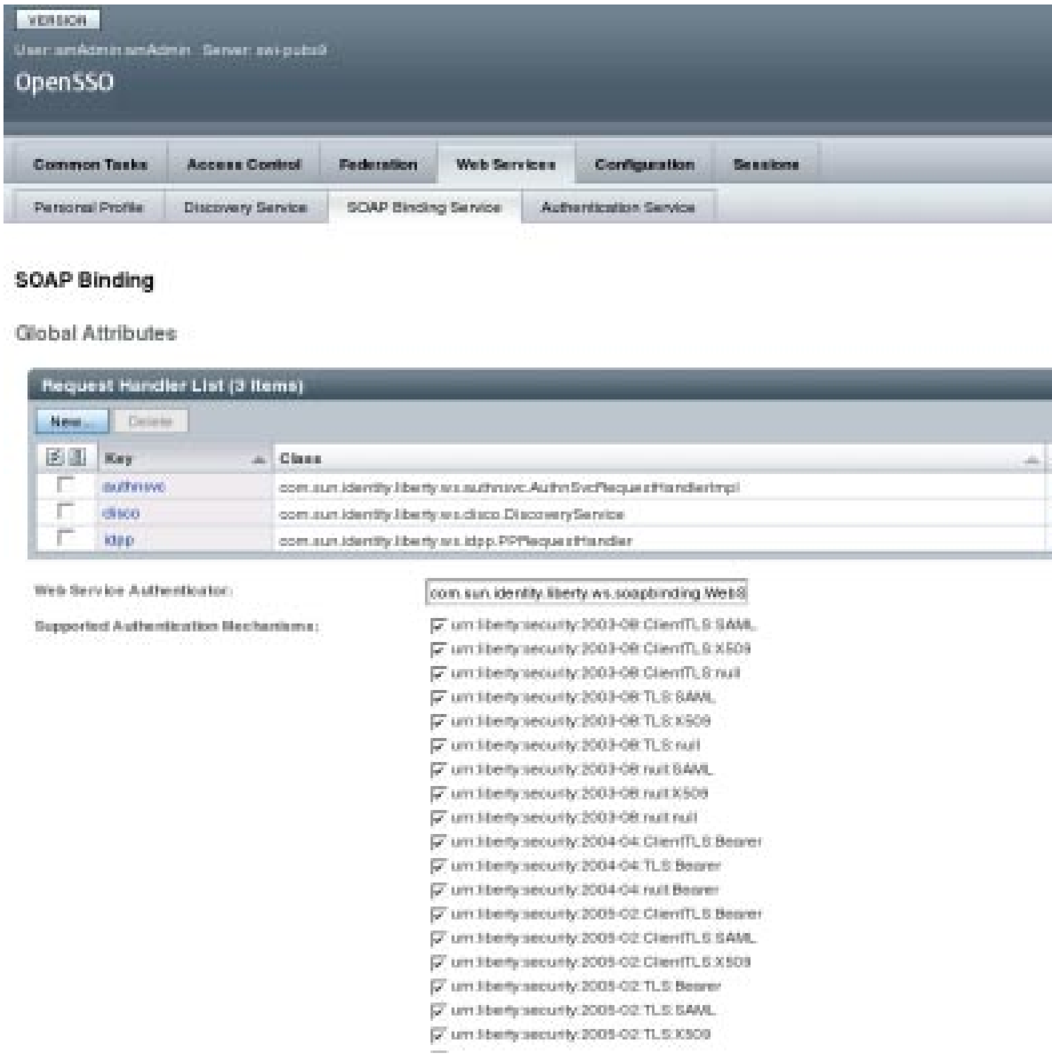
4 **Add the previously created classes to the web container classpath and restart the web container on which OpenSSO Enterprise is deployed.**

5 **Login to the OpenSSO Enterprise console as the top level administrator.**

By default, `amadmin`.

6 **Click the Web Services tab.**

7 **Under Web Services, click the SOAP Binding Service tab to register the new implementation with the SOAP Binding Service.**



VERSION

User: amAdmin amAdmin Server: swi-pub9

OpenSSO

Common Tasks | Access Control | Federation | Web Services | Configuration | Sessions

Personal Profile | Discovery Service | SOAP Binding Service | Authentication Service

SOAP Binding

Global Attributes

Request Handler List (3 Items)

New... Delete

	Key	Class
☐	authnsvc	com.sun.identity.liberty.ws.authnsvc.AuthnSvcRequestHandlerImpl
☐	disco	com.sun.identity.liberty.ws.disco.DiscoveryService
☐	idpp	com.sun.identity.liberty.ws.idpp.PPRequestHandler

Web Service Authenticator: com.sun.identity.liberty.ws.soapbinding.WebS

Supported Authentication Mechanisms:

- ✓ urn:liberty:security:2003-08:ClientTLS:SAML
- ✓ urn:liberty:security:2003-08:ClientTLS:X509
- ✓ urn:liberty:security:2003-08:ClientTLS:null
- ✓ urn:liberty:security:2003-08:TLS:SAML
- ✓ urn:liberty:security:2003-08:TLS:X509
- ✓ urn:liberty:security:2003-08:TLS:null
- ✓ urn:liberty:security:2003-08:null:SAML
- ✓ urn:liberty:security:2003-08:null:X509
- ✓ urn:liberty:security:2003-08:null:null
- ✓ urn:liberty:security:2004-04:ClientTLS:Bearer
- ✓ urn:liberty:security:2004-04:TLS:Bearer
- ✓ urn:liberty:security:2004-04:null:Bearer
- ✓ urn:liberty:security:2005-02:ClientTLS:Bearer
- ✓ urn:liberty:security:2005-02:ClientTLS:SAML
- ✓ urn:liberty:security:2005-02:ClientTLS:X509
- ✓ urn:liberty:security:2005-02:TLS:Bearer
- ✓ urn:liberty:security:2005-02:TLS:SAML
- ✓ urn:liberty:security:2005-02:TLS:X509

8 Click New under the Request Handler List global attribute.

9 Enter a name for the implementation in the Key field.

This value will be used as part of the service endpoint URL for the web service. For example, if the value is *stock*, the endpoint URL to access the stock quote web service will be:

`http://`*host*`:`*port*`/`*deploy_uri*`/Liberty/stock`

10 Enter the name of the implementation class previously created in the Class field.

11 **(Optional) Enter a SOAP Action in the SOAP Action field.**

12 **Click Save to save the configuration.**

The request handler will be displayed under the Request Handler List.

13 **Click on the Access Control tab to begin the process of publishing the web service to the Discovery Service.**

The Discovery Service is a registry of web services. It matches the properties in a request with the properties in its registry and returns the appropriate service location. See "Discovery Service" on page 152 for more information.

14 **Click the name of the realm to which you want to add the web service.**

15 **Click the Services tab to access the realm's services.**

16 **Click Discovery Service to create a new resource offering.**

If the Discovery Service has not yet been added:

a. **Click Add.**

A list of available services is displayed.

b. **Select Discovery Service and click Next to add the service.**

The list of added services is displayed including the link to the Discovery Service.

17 **Click Add on the Discovery Resource Offering screen.**

18 **(Optional) Enter a description of the resource offering in the Description field on the New Resource Offering page.**

19 **Type a URI for the value of the Service Type attribute.**

This URI defines the type of service. It is *recommended* that the value of this attribute be the `targetNamespace` URI defined in the *abstract* WSDL description for the service. An example of a valid URI is `urn:com:sun:liberty:sample:stockticker`.

20 **Type a URI for the value of the Provider ID attribute.**

The value of this attribute contains the URI of the provider of the service instance. This information is useful for resolving trust metadata needed to invoke the service instance. A single physical provider may have multiple provider IDs.

Note – The provider represented by the URI in the Provider ID attribute must also have an entry in the `ResourceIDMapper` attribute. For more information, see "Classes For ResourceIDMapper Plug-in" in *Sun OpenSSO Enterprise 8.0 Administration Guide*.

---

21 Click New Description to define the Service Description.

For each resource offering, at least one service description must be created.

a. Select the values for the Security Mechanism ID attribute to define how a web service client can authenticate to a web service provider.

This field lists the security mechanisms that the service instance supports. Select the security mechanisms that you want to add and click Add. To prioritize the list, select the mechanism and click Move Up or Move Down.

b. Type a value for the End Point URL.

This value is the URL to access the new web service. For this example, it might be:

`http://`*SERVER_HOST*`:`*SERVER_PORT*`/`*SERVER_DEPLOY_URI*`/Liberty/stock`

c. (Optional) Type a value for the SOAP Action.

This value is the equivalent of the `wsdlsoap:soapAction` attribute of the `wsdlsoap:operation` element in the service's concrete WSDL-based description.

d. Click OK to complete the configuration.

22 Check the Options box if there are no options or add a URI to the Options List to specify options for the resource offering.

This field lists the options that are available for the resource offering. Options provide hints to a potential requestor about the availability of certain data or operations to a particular offering. The set of possible URIs are defined by the service type, not the Discovery Service. If no option is specified, the service instance does not display any available options. For a standard set of options, see the *Liberty ID-SIS Personal Profile Service Specification*.

23 Select a directive for the resource offering.

Directives are special entries defined in SOAP headers that can be used to enforce policy-related decisions. You can choose from the following:

- `GenerateBearerToken` specifies that a bearer token be generated.
- `AuthenticateRequester` must be used with any service description that use SAML for message authentication.
- `EncryptResourceID` specifies that the Discovery Service encrypt the resource ID.

- `AuthenticateSessionContext` is specified when a Discovery Service provider includes a SAML assertion containing a `SessionContextStatement` in any future `QueryResponse` messages.
- `AuthorizeRequester` is specified when a Discovery Service provider wants to include a SAML assertion containing a `ResourceAccessStatement` in any future `QueryResponse` messages.

If you want to associate a directive with one or more service descriptions, select the check box for that Description ID. If no service descriptions are selected, the directive is applied to all description elements in the resource offering.

24 **Click OK.**

25 **Logout from the console.**

▼ To Invoke the Custom Service

Web service clients can access the custom web service by discovering the web service's end point and using the required credentials. This information is stored by the OpenSSO Enterprise Discovery Service. There are two ways in which a client can authenticate to OpenSSO Enterprise in order to access the Discovery Service:

- The Liberty ID-FF is generally used if it's a browser-based application and the web service client is a federation enabled service provider.
- The OpenSSO Enterprise Authentication Service (based on the Liberty ID-WSF) is used for remote web services clients with pure SOAP-based authentication capabilities.

In the following procedure, we use the Liberty ID-WSF client API to invoke the web service.

Note – The code in this procedure is used to demonstrate the usage of the Liberty ID-WSF client API. More information can be found in the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

---

1 **Write code to authenticate the WSC to the Liberty ID-WSF Authentication Service of OpenSSO Enterprise.**

The sample code below will allow access to the Discovery Service. It is a client-side program to be run inside the WSC application.

```
public class StockClient {
              :
    public SASLResponse authenticate(
        String userName,
        String password,
        String authurl) throws Exception {
```

```
        SASLRequest saslReq =
                        new SASLRequest(AuthnSvcConstants.MECHANISM_PLAIN);
        saslReq.setAuthzID(userName);

        SASLResponse saslResp = AuthnSvcClient.sendRequest(saslReq, authurl);
        String statusCode = saslResp.getStatusCode();
        if (!statusCode.equals(SASLResponse.CONTINUE)) {
                return null;
        }

        String serverMechanism = saslResp.getServerMechanism();
        saslReq  = new SASLRequest(serverMechanism);
        String dataStr = userName + "\0" + userName + "\0" + password;
        saslReq.setData(dataStr.getBytes("UTF-8"));
        saslReq.setRefToMessageID(saslResp.getMessageID());
        saslResp = AuthnSvcClient.sendRequest(saslReq, authurl);
        statusCode = saslResp.getStatusCode();
        if (!statusCode.equals(SASLResponse.OK)) {
            return null;
        }

        return saslResp;
    }
                       :

           }
```

2. **Add code that will extract the Discovery Service information from the Authentication Response.**

   The following additional code would be added to what was developed in the previous step.

```
ResourceOffering discoro = saslResp.getResourceOffering();
          List credentials = authnResponse.getCredentials();
```

3. **Add code to query the Discovery Service for the web service's resource offering by using the Discovery Service resource offering and the credentials that are required to access it.**

   The following additional code would be added to what was previously developed.

```
RequestedService rs = new RequestedService(null,
                   "urn:com:sun:liberty:sample:stockticker");
              List rss = new ArrayList();
              rss.add(rs);

             Query  discoQuery = new Query(discoro.getResourceID(), rss);

             DiscoveryClient discoClient = null;

             discoClient = new DiscoveryClient(secAssertion, serviceURL, null);

             QueryResponse queryResponse = discoClient.getResourceOffering(discoQuery);
```

4 **The discovery response contains the service's resource offering and the credentials required to access the service.**

`quotes` contains the response body (the stock quote). You would use the OpenSSO Enterprise SOAP API to get the body elements.

```
 List offerings = discoResponse.getResourceOffering();
          ResourceOffering stockro = (ResourceOffering)offerings.get(0);

          List credentials = discoResponse.getCredentials();

          SecurityAssertion secAssertion = null;
          if(credentials != null && !credentials.isEmpty()) {
             secAssertion = (SecurityAssertion)credentials.get(0);
          }

          String serviceURL = ((Description)stockro.getServiceInstance().
                   getDescription().get(0)).getEndpoint();

          QuoteRequest req = new QuoteRequest(symbol,
               stockro.getResourceID().getResourceID());
          Element elem =  XMLUtils.toDOMDocument(
              req.toString(), debug).getDocumentElement();

          List list = new ArrayList();
          list.add(elem);

          Message msg = new Message(null, secAssertion);
          msg.setSOAPBodies(list);

          Message response = Client.sendRequest(msg, serviceURL, null, null);
          List quotes = response.getBodies();
```

Setting Up Liberty ID-WSF 1.1 Profiles

OpenSSO Enterprise automatically detects which version of the Liberty ID-WSF profiles is being used. If OpenSSO Enterprise is the web services provider (WSP), it detects the version from the incoming SOAP message. If OpenSSO Enterprise is the WSC, it uses the version the WSP has registered with the Discovery Service. If the WSP can not detect the version from the incoming SOAP message or the WSC can not communicate with the Discovery Service, the version defined in the `com.sun.identity.liberty.wsf.version` property in the OpenSSO Enterprise configuration data store will be used. Following are the steps to configure OpenSSO Enterprise to use Liberty ID-WSF 1.1 profiles.

- "To Configure OpenSSO Enterprise to Use Liberty ID-WSF 1.1 Profiles" on page 141
- "To Test the Liberty ID-WSF 1.1 Configuration" on page 144

▼ To Configure OpenSSO Enterprise to Use Liberty ID-WSF 1.1 Profiles

- Don't use the Liberty ID-FF sample as it does not configure a signing key.
- If both machines are in the same domain, change the cookie name on one of them to avoid cookie conflict.

1 **Install OpenSSO Enterprise on two different machines.**

Test the installations by logging in to the console at **`http://`***`server`***`:`***`port`**`/opensso/UI/Login`**.

2 **Configure one instance of OpenSSO Enterprise as a Liberty ID-FF identity provider.**

a. **Login to the OpenSSO Enterprise console.**

b. **Click the Federation tab.**

c. **Click New under Entity Providers.**

The Create IDFF Entity Provider page is displayed.

d. **Enter a value for the Entity Identifier attribute on the Create IDFF Entity Provider page.**

e. **Under Identity Provider, enter values for Meta Alias, Signing Certificate Alias, and Encryption Certificate Alias and click Create to create the identity provider metadata.**

f. **Using `ssoadm.jsp`, export the identity provider metadata.**

3 **Configure the second instance of OpenSSO Enterprise as a Liberty ID-FF service provider.**

a. **Login to the OpenSSO Enterprise console.**

b. **Click the Federation tab.**

c. **Click New under Entity Providers.**

The Create IDFF Entity Provider page is displayed.

d. **Enter a value for the Entity Identifier attribute on the Create IDFF Entity Provider page.**

e. **Under Service Provider, enter values for Meta Alias, Signing Certificate Alias, and Encryption Certificate Alias and click Create to create the service provider metadata.**

f. **Using `ssoadm.jsp`, export the service provider metadata.**

4. **Exchange the standard metadata files and import the identity provider metadata onto the service provider machine and the service provider metadata onto the identity provider machine.**

5. **Create a circle of trust that includes the Entity Identifier for both providers on each machine.**

6. **Login to the instance of OpenSSO Enterprise acting as the identity provider.**

   a. **Click the Web Services tab.**

   b. **Click the Discovery Service tab.**

   c. **Scroll down to Resource Offerings for Bootstrapping.**

   d. **Click** `urn:liberty:disco:2003-08`**.**

      The Edit Resource Offerings page is displayed.

   e. **Remove the default value of Service Type.**

   f. **Add** `urn:liberty:security:2005-02:null:X509`**.**

   g. **Change the value of the Provider ID attribute to the entity identifier of the identity provider.**

   h. **Click Save.**

      The Discovery Service page is displayed.

   i. **Scroll down to the Classes for ResourceID Mapper Plug-in attribute.**

   j. **Click the link that is the value of the Provider ID.**

      The Edit Resource ID Mapping page is displayed.

   k. **Change the value of the Provider ID attribute to the entity identifier of the identity provider.**

   l. **Click Save.**

      The Discovery Service page is displayed.

   m. **Click the Configuration tab.**

   n. **Click the Global tab.**

   o. **Click the Liberty ID-WSF Security Service link.**

      The Liberty ID-WSF Security Service page is displayed.

p. **Enter `test` as the value for the following attributes and click Save.**

- Default WSC Certificate alias
- Trusted Authority signing certificate alias
- Trusted CA signing certificate aliases

Note – `test` is the default self-signed certificate shipped with OpenSSO Enterprise. Use your own key and CA name for your customized deployment.

---

q. **Log out of the console and restart the identity provider instance to allow the changes to take effect.**

7 **Login to the instance of OpenSSO Enterprise acting as the service provider.**

a. **Click the Web Services tab.**

b. **Under the Personal Profile tab, change the value of the Provider ID attribute to the entity identifier of the service provider and click Save.**

c. **Click the SOAP Binding Service tab.**

d. **Scroll down, enable 1.1 as the value of the Liberty Identity Web Services Version attribute and click Save.**

e. **Click the Configuration tab.**

f. **Click the Global tab.**

g. **Click the Liberty ID-WSF Security Service link.**

The Liberty ID-WSF Security Service page is displayed.

h. **Enter `test` as the value for the following attributes and click Save.**

- Default WSC Certificate alias
- Trusted Authority signing certificate alias
- Trusted CA signing certificate aliases

Note – `test` is the default self-signed certificate shipped with OpenSSO Enterprise. Use your own key and CA name for your customized deployment.

---

i. **Log out of the console and restart the service provider instance to allow the changes to take effect.**

▼ To Test the Liberty ID-WSF 1.1 Configuration

1. **Deploy the OpenSSO Enterprise client WAR on a third web container.**
   - Use `opensso-client-jdk15.war` for web containers running the Java Development Kit (JDK) 1.5 and above.
   - Use `opensso-client-jdk14.war` for web containers running JDK 1.4.
2. **Configure the client sample and then configure the WSC sample.**
3. **Find `AMConfig.properties` for the Client SDK under the *user_home*/`OpenSSOClient` directory.**

   For example, *path_to_client_sample_deployment*_`AMConfig.properties`
4. **Edit the following properties in `AMConfig.properties`.**
   - `com.sun.identity.liberty.ws.wsc.certalias=test`
   - `com.sun.identity.liberty.ws.ta.certalias=test`
   - `com.sun.identity.liberty.ws.trustedca.certalias=test`

   Note – `test` is the default self-signed certificate shipped with OpenSSO Enterprise. Use your own key and CA name for your customized deployment.

   ---
5. **Restart the Client SDK web container and follow the client SDK sample README to run the sample.**

   All Liberty ID-WSF traffic is using version 1.1 now. You can validate this by looking at the XML message; the name space for the SOAP binding should be `urn:liberty:sb:2004-04` as opposed to `urn:liberty:sb: 2003-08` for version 1.0.

Common Application Programming Interfaces

The following list describes the API common to all Liberty-based OpenSSO Enterprise service components and services.

- "Common Interfaces" on page 144
- "Common Security API" on page 146

Common Interfaces

This section summarizes classes that can be used by all Liberty-based OpenSSO Enterprise web service components, as well as interfaces common to all Liberty-based OpenSSO Enterprise web services. The packages that contain the classes and interfaces are:

- "`com.sun.identity.liberty.ws.common` Package" on page 145
- "`com.sun.identity.liberty.ws.interfaces` Package" on page 145

`com.sun.identity.liberty.ws.common` Package

This package includes the `Status` class common to all Liberty-based OpenSSO Enterprise web service components. It represents a common status object. For more information, including methods and their syntax and parameters, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

`com.sun.identity.liberty.ws.interfaces` Package

This package includes interfaces that can be implemented to add their corresponding functionality to each Liberty-based OpenSSO Enterprise web service.

TABLE 9–1 `com.sun.identity.liberty.ws.interfaces` Interfaces

Interface	Description
`Authorizer`	This interface, once implemented, can be used by each Liberty-based web service component for access control. **Note** – The `com.sun.identity.liberty.ws.disco.plugins.DefaultDiscoAuthorizer` class is the implementation of this interface for the Discovery Service. The `com.sun.identity.liberty.ws.idpp.plugin.IDPPAuthorizer` class is the implementation for the Liberty Personal Profile Service. The `Authorizer` interface enables a web service to check whether a web service consumer (WSC) is allowed to access the requested resource. When a WSC contacts a web service provider (WSP), the WSC conveys a sender identity and an invocation identity. Note that the *invocation identity* is always the subject of the SAML assertion. These conveyances enable the WSP to make an authorization decision based on one or both identities. The OpenSSO Enterprise Policy Service performs the authorization based on defined policies.

TABLE 9–1 `com.sun.identity.liberty.ws.interfaces` Interfaces *(Continued)*

Interface	Description
`ResourceIDMapper`	This interface is used to map a user DN to the resource identifier associated with it. OpenSSO Enterprise provides implementations of this interface. ■ `com.sun.identity.liberty.ws.disco.plugins.Default64ResourceIDMapper` assumes the Resource ID format to be: *providerID* + "/" + *the Base64 encoded userIDs*. ■ `com.sun.identity.liberty.ws.disco.plugins.DefaultHexResourceIDMapper` assumes the Resource ID format to be: *providerID* + "/" + *the hex string of userID*. ■ `com.sun.identity.liberty.ws.idpp.plugin.IDPPResourceIDMapper` assumes the Resource ID format to be: *providerID* + "/" + *the Base64 encoded userIDs*. A different implementation of the interface may be developed. The implementation class should be given to the provider that hosts the Discovery Service. The mapping between the *providerID* and the implementation class can be configured through the Classes For ResourceIDMapper Plugin attribute.
`ServiceInstanceUpdate`	Interface used to include a SOAP header (`ServiceInstanceUpdateHeader`) when sending a SOAP response.

For more information, including methods and their syntax and parameters, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

Common Security API

The Liberty-based security APIs are included in the `com.sun.identity.liberty.ws.security` package and the `com.sun.identity.liberty.ws.common.wsse` package.

`com.sun.identity.liberty.ws.security` Package

The `com.sun.identity.liberty.ws.security` package includes the `SecurityTokenProvider` interface for managing Web Service Security (WSS) type tokens and the `SecurityAttributePlugin` interface for inserting security attributes (using an `AttributeStatement`) into the assertion during the Discovery Service token generation. The following table describes the classes used to manage Liberty-based security mechanisms.

TABLE 9–2 `com.sun.identity.liberty.ws.security` Classes

Class	Description
`ProxySubject`	Represents the identity of a proxy, the confirmation key, and confirmation obligation the proxy must possess and demonstrate for authentication purposes.
`ResourceAccessStatement`	Conveys information regarding the accessing entities and the resource for which access is being attempted.
`SecurityAssertion`	Provides an extension to the Assertion class to support ID-WSF `ResourceAccessStatement` and `SessionContextStatement`.
`SecurityTokenManager`	An entry class for the security package `com.sun.identity.liberty.ws.security`. You can call its methods to generate X.509 and SAML tokens for message authentication or authorization. It is designed as a provider model, so different implementations can be plugged in if the default implementation does not meet your requirements.
`SessionContext`	Represents the session status of an entity to another system entity.
`SessionContextStatement`	Conveys the session status of an entity to another system entity within the body of an `<saml:assertion>` element.
`SessionSubject`	Represents a Liberty subject with its associated session status.

For more information, including methods and their syntax and parameters, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

`com.sun.identity.liberty.ws.common.wsse` Package

This package includes `BinarySecurityToken` which provides an interface to parse and create the X.509 Security Token in accordance with the *Liberty ID-WSF Security Mechanisms*. Both WSS X.509 and SAML tokens are supported. For more information, including methods and their syntax and parameters, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

Authentication Web Service

The SOAP specifications define an XML-based messaging paradigm, but do not specify any particular security mechanisms. Particularly, they do not describe user authentication using SOAP messages. To rectify this, the Liberty-based Authentication Web Service was implemented based on the *Liberty ID-WSF Authentication Service and Single Sign-On Service Specification*. The specification defines a protocol that adds the Simple Authentication and

Security Layer (SASL) authentication functionality to the SOAP binding described in the *Liberty ID-WSF SOAP Binding Specification* and "SOAP Binding Service" on page 159. The Liberty-based Authentication Web Service is for provider-to-provider authentication.

Note – The specification also contains an XML schema that defines the authentication protocol. More information can be found in Schema Files and Service Definition Documents.

- "Authentication Web Service Default Implementation" on page 148
- "Authentication Web Service Packages" on page 149
- "Access the Authentication Web Service" on page 149

Authentication Web Service Default Implementation

The Authentication Web Service attributes are *global*; the value of this attribute is carried across the OpenSSO Enterprise configuration and inherited by every realm. The attributes for the Authentication Web Service are defined in the `amAuthnSvc.xml` service file. The Mechanism Handlers List attribute stores information about the SASL mechanisms that are supported by the Authentication Web Service and contains two parameters.

- "`key` Parameter" on page 148
- "`class` Parameter" on page 148

`key` Parameter

The required `key` defines the SASL mechanism supported by the Authentication Web Service.

`class` Parameter

The required `class` specifies the name of the implemented class for the SASL mechanism. Two authentication mechanisms are supported by the following default implementations:

TABLE 9–3 Default Implementations for Authentication Mechanism

Class	Description
`com.sun.identity.liberty.ws.authnsvc.mechanism.PlainMechanismHandler`	This class is the default implementation for the PLAIN authentication mechanism. It maps user identifiers and passwords in the PLAIN mechanism to the user identifiers and passwords in the LDAP authentication module under the root organization.
`com.sun.identity.liberty.ws.authnsvc.mechanism.CramMD5MechanismHandler`	This class is the default implementation for the CRAM-MD5 authentication mechanism.

The Authentication Web Service layer provides an interface that must be implemented for each SASL mechanism to process the requested message and return a response.

Authentication Web Service Packages

The Authentication Web Service provides programmatic interfaces to allow clients to interact with it. The following sections provide short descriptions of these packages. For more detailed information, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*. The authentication-related packages include:

- "`com.sun.identity.liberty.ws.authnsvc` Package" on page 149
- "`com.sun.identity.liberty.ws.authnsvc.mechanism` Package" on page 149
- "`com.sun.identity.liberty.ws.authnsvc.protocol` Package" on page 149

`com.sun.identity.liberty.ws.authnsvc` Package

This package provides web service clients with a method to request authentication credentials from the Authentication Web Service and receive responses back from it using the Simple Authentication and Security Layer (SASL).

`com.sun.identity.liberty.ws.authnsvc.mechanism` Package

This package provides an interface that must be implemented for each different SASL mechanism to enable authentication using them. Each SASL mechanism will correspond to one implementation that will process incoming SASL requests and generate outgoing SASL responses.

`com.sun.identity.liberty.ws.authnsvc.protocol` Package

This package provides classes that correspond to the request and response elements defined in the Liberty XSD schema that accompanies the *Liberty ID-WSF Authentication Service Specification*.

Access the Authentication Web Service

The URL to gain access to the Authentication Web Service is:

`http://`*SERVER_HOST*`:`*SERVER_PORT*`/`*SERVER_DEPLOY_URI*`/Liberty/authnsvc`

This URL is normally used by the OpenSSO Enterprise client API to access the service. For example, the OpenSSO Enterprise public client, `com.sun.identity.liberty.ws.authnsvc.AuthnSvcClient` uses this URL to authenticate principals with OpenSSO Enterprise.

Data Services

A *data service* is a web service that supports the query and modification of data regarding a principal. An example of a data service is a web service that hosts and exposes a principal's profile information, such as name, address and phone number. A *query* is when a web service consumer (WSC) requests and receives the data (in XML format). A *modify* is when a WSC sends new information to update the data. The Liberty Alliance Project has defined the *Liberty ID-WSF Data Services Template Specification* (Liberty ID-WSF-DST) as the standard protocol for the query and modification of data profiles exposed by a data service. Using this specification, the Liberty Alliance Project has developed additional specifications for other types of data services: personal profile service, geolocation service, contact service, and calendar service). Of these data services, OpenSSO Enterprise has implemented the Liberty Personal Profile Service.

- "Liberty Personal Profile Service" on page 150
- "Data Services Template Packages" on page 150

Liberty Personal Profile Service

The Liberty Personal Profile Service is a default OpenSSO Enterprise identity service. It can be queried for identity data and its attributes can be updated.

For access to occur, the hosting provider of the Liberty Personal Profile Service needs to be registered with the Discovery Service on behalf of each identity principal. To register a service with the Discovery Service, update a resource offering for that service. For more information, see "Discovery Service" on page 152.

The URL to gain access to the Liberty Personal Profile Service is:

http://*SERVER_HOST*:*SERVER_PORT*/*SERVER_DEPLOY_URI*/Liberty/idpp

This URL is normally used by the OpenSSO Enterprise client API to access the service. For example, the OpenSSO Enterprise public Data Service Template client, `com.sun.identity.liberty.ws.dst.DSTClient` uses this URL to query and modify an identity's personal profile attributes stored in OpenSSO Enterprise.

Data Services Template Packages

OpenSSO Enterprise contains two packages based on the Liberty ID-WSF-DST. They are:

- "`com.sun.identity.liberty.ws.dst` Package" on page 151
- "`com.sun.identity.liberty.ws.dst.service` Package" on page 151

For more detailed API documentation, including methods and their syntax and parameters, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

`com.sun.identity.liberty.ws.dst` **Package**

The following table summarizes the classes in the Data Services Template client API that are included in the `com.sun.identity.liberty.ws.dst` package.

TABLE 9–4 Data Service Client APIs

Class	Description
`DSTClient`	Provides common functions for the Data Services Templates query and modify options.
`DSTData`	Provides a wrapper for any data entry.
`DSTModification`	Represents a Data Services Template modification operation.
`DSTModify`	Represents a Data Services Template modify request.
`DSTModifyResponse`	Represents a Data Services Template response to a DST modify request.
`DSTQuery`	Represents a Data Services Template query request.
`DSTQueryItem`	Wrapper for one query item.
`DSTQueryResponse`	Represents a Data Services Template query response.
`DSTUtils`	Provides utility methods used by the DST layer.

`com.sun.identity.liberty.ws.dst.service` **Package**

This package provides a handler class that can be used by any generic identity data service that is built using the *Liberty Alliance ID-SIS Specifications*.

Note – The Liberty Personal Profile Service is built using the *Liberty ID-SIS Personal Profile Service Specification*, based on the *Liberty Alliance ID-SIS Specifications*.

The `DSTRequestHandler` class is used to process query or modify requests sent to an identity data service. It is an implementation of the interface `com.sun.identity.liberty.ws.soapbinding.RequestHandler`. For more detailed API documentation, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

Discovery Service

OpenSSO Enterprise contains a Discovery Service defined by the Liberty Alliance Project specifications. The Discovery Service allows a requesting entity to dynamically determine a principal's registered identity service. It might also function as a security token service, issuing security tokens to the requester that can then be used in the request to the discovered identity service. The following sections contain more information.

- "Generating Security Tokens" on page 152
- "Discovery Service Packages" on page 155
- "Access the Discovery Service" on page 159

Generating Security Tokens

In general, a discovery service and an identity provider are hosted on the same machine. Because the identity provider hosting the Discovery Service might be fulfilling other roles for an identity (such as a Policy Decision Point or an Authentication Authority), it can be configured to provide the requesting entity with security tokens. The Discovery Service can include a security token (inserted into a SOAP message header) in a `DiscoveryLookup` response. The token can then be used as a credential to invoke the service returned with it.

Note – For information regarding the deployment of the Client SDK, see Chapter 13, "Using the Client SDK."

▼ To Configure the Discovery Service to Generate Security Tokens

1 **Generate the keystore and certificate aliases for the machines that are hosting the Discovery Service, the WSP and the WSC.**

OpenSSO Enterprise uses a Java keystore for storing the public and private keys so, if this is a new deployment, you might need to generate one using `keytool`, the key and certificate management utility supplied with the Java Platform, Standard Edition. In short, `keytool` generates key pairs as separate key entries (one for a public key and the other for its associated private key). It wraps the public key into an X.509 self-signed certificate (one for which the issuer/signer is the same as the subject), and stores it as a single-element certificate chain. Additionally, the private key is stored separately, protected by a password, and associated with the certificate chain for the corresponding public key. All public and private keystore entries are accessed via unique aliases.

2 **Update the values of the key-related properties for the appropriate deployed instances of OpenSSO Enterprise.**

Note – The same property might have already been edited depending on the deployment scenario.

---

a. **For the web services provider and web services client deployed on OpenSSO Enterprise:**

   i. **Login to the OpenSSO Enterprise console.**

   ii. **Click the Configuration tab.**

   iii. **Click the Global tab.**

   iv. **Click the Liberty ID-WSF Security Service link.**

   The Liberty ID-WSF Security Service page is displayed.

   v. **Enter `test` as the value for the following attributes and click Save.**

   - Default WSC Certificate alias
   - Trusted Authority signing certificate alias
   - Trusted CA signing certificate aliases

   Note – `test` is the default self-signed certificate shipped with OpenSSO Enterprise. Use your own key and CA name for your customized deployment. If you want to use a different keystore location, under the Configuration tab click Servers and Sites. Click the link of the appropriate server instance. Under the Security tab click Inheritance Settings and do the following:

   - Uncheck the Keystore File box.
   - Optionally, uncheck the Private Key Password File box and the Keystore Password File box.

   Click Save and Back to Server Profile. Click the Keystore link and enter the location of the Keystore File. (If you change the password for the Private Key or Keystore, you need to encode the new password using the `ampassword` command or `encode.jsp` before putting it into the corresponding password file.)

   ---

   vi. **Log out of the console and restart the instance to allow the changes to take effect.**

b. **For the web services provider and web services client deployed on the same machine as the OpenSSO Enterprise Client SDK update the values of the following key-related properties in the `AMConfig.properties`:**

   - `com.sun.identity.saml.xmlsig.keystore` defines the location of the keystore file.

- `com.sun.identity.saml.xmlsig.storepass` defines the location of the file that contains the password used to access the keystore file.
- `com.sun.identity.saml.xmlsig.keypass` defines the location of the file that contains the password used to protect the private key of a generated key pair.
- `com.sun.identity.liberty.ws.wsc.certalias` defines the certificate alias used for signing the WSP protocol responses.
- `com.sun.identity.liberty.ws.trustedca.certaliases` defines the certificate alias and the Provider ID list on which the WSP is trusting.

3. **Configure each identity provider and service provider as an entity using the Federation module.**

   This entails configuring each provider as an entity in a circle of trust.

4. **Establish provider trust between the entities by creating an authentication domain using the Federation module.**

   See Part II, "Federation, Web Services, and SAML Administration," in *Sun OpenSSO Enterprise 8.0 Administration Guide*.

5. **Change the default value of the Provider ID for the Discovery Service on the machine where the Discovery Service is hosted to the value that reflects the previously loaded metadata.**

   a. **Click the Web Services tab from the OpenSSO Enterprise Console.**

   b. **Click the Discovery Service tab under Web Services.**

   c. **Change the default value of the Provider ID from *protocol*`://`*host*`:`*port*`/`*deployuri*`/Liberty/disco` to the Entity ID of the identity provider.**

6. **Change the default value of the Provider ID for the Liberty Personal Profile Service on the machine where the Liberty Personal Profile Service is hosted to the value that reflects the previously loaded metadata.**

   a. **Click the Web Services tab from the OpenSSO Enterprise Console.**

   b. **Click the Liberty Personal Profile Service tab under Web Services.**

   c. **Change the default value of the Provider ID from *protocol*`://`*host*`:`*port*`/`*deployuri*`/Liberty/idpp` to the Entity ID of the identity provider.**

7. **Register a resource offering for the WSP using either of the following methods.**

   Make sure that the appropriate directives are chosen.

   - For SAML Bearer token use `GenerateBearerToken` or `AuthenticateRequester`.
   - For SAML Token (Holder of key) use `AuthenticateRequester` or `AuthorizeRequester`.

Discovery Service Packages

OpenSSO Enterprise contains several Java packages that are used by the Discovery Service. They include:

- `com.sun.identity.liberty.ws.disco` includes a client API that provides interfaces to communicate with the Discovery Service. See "Client APIs in `com.sun.identity.liberty.ws.disco`" on page 155.
- `com.sun.identity.liberty.ws.disco.plugins` includes an interface that can be used to develop plug-ins. The package also contains some default plug-ins. See "`com.sun.identity.liberty.ws.disco.plugins.DiscoEntryHandler` Interface" on page 156.
- `com.sun.identity.liberty.ws.interfaces` includes interfaces that can be used to implement functionality common to all Liberty-enabled identity services. Several implementations of these interfaces have been developed for the Discovery Service. See "`com.sun.identity.liberty.ws.interfaces.Authorizer` Interface" on page 157 and "`com.sun.identity.liberty.ws.interfaces.ResourceIDMapper` Interface" on page 159.

Note – Additional information is in the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

Client APIs in `com.sun.identity.liberty.ws.disco`

The following table summarizes the client APIs in the package `com.sun.identity.liberty.ws.disco`. For more information, including methods and their syntax and parameters, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

TABLE 9–5 Discovery Service Client APIs

Class	Description
`Description`	Represents a DescriptionType element of a service instance.
`Directive`	Represents a discovery service DirectiveType element.
`DiscoveryClient`	Provides methods to send Discovery Service queries and modifications.
`EncryptedResourceID`	Represents an EncryptionResourceID element for the Discovery Service.
`InsertEntry`	Represents an Insert Entry for Discovery Modify request.
`Modify`	Represents a discovery modify request.
`ModifyResponse`	Represents a discovery response to a modify request.
`Query`	Represents a discovery Query object.

TABLE 9–5 Discovery Service Client APIs (Continued)

Class	Description
`QueryResponse`	Represents a response to a discovery query request.
`RemoveEntry`	Represents a remove entry element for the discovery modify request.
`RequestedService`	Enables the requester to specify that all the resource offerings returned must be offered through a service instance that complies with one of the specified service types.
`ResourceID`	Represents a Discovery Service Resource ID.
`ResourceOffering`	Associates a resource with a service instance that provides access to that resource.
`ServiceInstance`	Describes a web service at a distinct protocol endpoint.

`com.sun.identity.liberty.ws.disco.plugins.DiscoEntryHandler` Interface

This interface is used to get and set discovery entries for a user. A number of default implementations are provided, but if you want to handle this function differently, implement this interface and set the implementing class as the value of the Entry Handler Plugin Class attribute as discussed in "Entry Handler Plug-in Class" in *Sun OpenSSO Enterprise 8.0 Administration Guide*. The default implementations of this interface are described in the following table.

TABLE 9–6 Implementations of `com.sun.identity.liberty.ws.disco.plugins.DiscoEntryHandler`

Class	Description
`UserDiscoEntryHandler`	Gets or modifies discovery entries stored in the user's entry as a value of the `sunIdentityServerDiscoEntries` attribute. The `UserDiscoEntryHandler` implementation is used in business-to-consumer scenarios such as the Liberty Personal Profile Service.
`DynamicDiscoEntryHandler`	Gets discovery entries stored as a value of the `sunIdentityServerDynamicDiscoEntries` dynamic attribute in the Discovery Service. Modification of these entries is not supported and always returns `false`. The resource offering is saved in an organization or a role. The `DynamicDiscoEntryHandler` implementation is used in business-to-business scenarios such as the Liberty Employee Profile service.

TABLE 9–6 Implementations of `com.sun.identity.liberty.ws.disco.plugins.DiscoEntryHandler` *(Continued)*

Class	Description
`UserDynamicDiscoEntryHandler`	Gets a union of the discovery entries stored in the user entry `sunIdentityServerDiscoEntries` attribute and discovery entries stored in the Discovery Service `sunIdentityServerDynamicDiscoEntries` attribute. It modifies only discovery entries stored in the user entry. The `UserDynamicDiscoEntryHandler` implementation can be used in both business-to-consumer and business-to-business scenarios.

`com.sun.identity.liberty.ws.interfaces.Authorizer` Interface

This interface is used to enable an identity service to check the authorization of a WSC. The `DefaultDiscoAuthorizer` class is the default implementation of this interface. The class uses the OpenSSO Enterprise Policy Service for creating and applying policy definitions. Policy definitions for the Discovery Service are configured using the OpenSSO Enterprise Console.

Note – The Policy Service looks for an `SSOToken` defined for Authenticated Users or Web Service Clients. For more information on this and the Policy Service in general, see the *Sun OpenSSO Enterprise 8.0 Administration Guide*.

▼ To Configure Discovery Service Policy Definitions

1. **In the OpenSSO Enterprise Console, click the Access Control tab.**
2. **Select the name of the realm in which the policy definitions will be configured.**
3. **Select Policies to access policy configurations.**
4. **Click New Policy to add a new policy definition.**
5. **Type a name for the policy.**
6. **(Optional) Enter a description for the policy.**
7. **(Optional) Select the check box next to Active.**
8. **Click New to add rules to the policy definition.**
9. **Select Discovery Service for the rule type and click Next.**

10 **Type a name for the rule.**

11 **Type a resource on which the rule acts.**

The Resource Name field uses the form *ServiceType* + *RESOURCE_SEPARATOR* + *ProviderID*. For example, `urn:liberty:id-sis-pp:2003-08;http://example.com`.

12 **Select an action and appropriate value for the rule.**

Discovery Service policies can only look up or update data.

13 **Click Finish to configure the rule.**

The `com.sun.identity.liberty.ws.interfaces.Authorizer` interface can be implemented by any web service in OpenSSO Enterprise. For more information, see XXXXXCommon Service Interfaces and the Java API Reference in //*OpenSSO-base*/`SUNWam/docs` or on docs.sun.com.

14 **Click New to add subjects to the policy definition.**

15 **Select the subject type and click Next.**

16 **Type a name for the group of subjects.**

17 **(Optional) Click the check box if this is an exclusive group.**

18 **Select the users and click to add them to the group.**

19 **Click Finish to return to the policy definition screen.**

20 **Click New to add conditions to the policy definition.**

21 **Select the condition type and click Next.**

22 **Type values for the displayed attributes.**

For more information, see the *Sun OpenSSO Enterprise 8.0 Administration Guide*.

23 **Click Finish to return to the policy definition screen.**

24 **Click New to add response providers to the policy definition.**

25 **Type a name for the response provider.**

26 **(Optional) Add values for the StaticAttribute.**

27 **(Optional) Add values for the DynamicAttribute.**

28 Click Finish to return to the policy definition screen.

29 Click Create to finish the policy configuration.

`com.sun.identity.liberty.ws.interfaces.ResourceIDMapper` Interface

This interface is used to map a user ID to the resource identifier associated with it. OpenSSO Enterprise provides two implementations of this interface.

- `com.sun.identity.liberty.ws.disco.plugins.Default64ResourceIDMapper` assumes the format to be *providerID* + "/" + *the Base64 encoded userIDs*
- `com.sun.identity.liberty.ws.disco.plugins.DefaultHexResourceIDMapper` assumes the format to be *providerID* + "/" + *the hex string of userIDs*

A different implementation of the interface may be developed. The implementation class should be given to the provider that hosts the Discovery Service. The mapping between the *providerID* and the implementation class can be configured through the XXXXXClasses For ResourceIDMapper Plug-in attribute.

Note – The `com.sun.identity.liberty.ws.interfaces.ResourceIDMapper` interface is common to all identity services in OpenSSO Enterprise not only the Discovery Service. For more information, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

Access the Discovery Service

The URL to gain access to the Discovery Service is:

`http://`*SERVER_HOST*`:`*SERVER_PORT*`/`*SERVER_DEPLOY_URI*`/Liberty/disco`

This URL is normally used by the OpenSSO Enterprise client API to access the service. For example, the public Discovery Service client, `com.sun.identity.liberty.ws.disco.DiscoveryClient` uses this URL to query and modify the resource offerings of an identity.

SOAP Binding Service

OpenSSO Enterprise contains an implementation of the *Liberty ID-WSF SOAP Binding Specification* from the Liberty Alliance Project. The specification defines a transport layer for sending and receiving SOAP messages.

- “`SOAPReceiver` Servlet” on page 160

- “SOAP Binding Service Package” on page 160

SOAPReceiver Servlet

The SOAPReceiver servlet receives a Message object from a web service client (WSC), verifies the signature, and constructs its own Message object for processing by OpenSSO Enterprise. The SOAPReceiver then invokes the correct request handler class to pass this second Message object on to the appropriate OpenSSO Enterprise service for a response. When the response is generated, the SOAPReceiver returns this Message object back to the WSC. More information can be found in the “SOAP Binding Service” in *Sun OpenSSO Enterprise 8.0 Technical Overview*.

SOAP Binding Service Package

The SOAP Binding Service includes a Java package named com.sun.identity.liberty.ws.soapbinding. This package provides classes to construct SOAP requests and responses and to change the contact point for the SOAP binding. The following table describes some of the available classes. For more detailed information, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

TABLE 9–7 SOAP Binding Service API

Class	Description
Client	Provides a method with which a WSC can send a request to a WSP using a SOAP connection. It also returns the response.
ConsentHeader	Represents the SOAP element named Consent.
CorrelationHeader	Represents the SOAP element named Correlation. By default, CorrelationHeader will always be signed.
ProcessingContextHeader	Represents the SOAP element named ProcessingContext.
ProviderHeader	Represents the SOAP element named Provider.
RequestHandler	Defines an interface that needs to be implemented on the server side by each web service in order to receive a request from a WSC and generate a response. After implementing the class, it must be registered in the SOAP Binding Service so the SOAP framework knows where to forward incoming requests.

TABLE 9–7 SOAP Binding Service API *(Continued)*

Class	Description
`Message`	Represents a SOAP message and is used by both the web service client and server to construct SOAP requests and responses. Each SOAP message has multiple headers and bodies. It may contain a certificate for client authentication, the IP address of a remote endpoint, and a SAML assertion used for signing.
`ServiceInstanceUpdateHeader`	Allows a service to change the endpoint on which requesters will contact it.
`ServiceInstanceUpdateHeader.Credential`	Allows a service to use a different security mechanism and credentials to access the requested resource.
`SOAPFault`	Represents the SOAP element named `SOAP Fault`.
`SOAPFaultDetail`	Represents the SOAP element named `Detail`, a child element of `SOAP Fault`.
`UsageDirectiveHeader`	Defines the SOAP element named `UsageDirective`.

See “PAOS Binding” on page 164 for information on this reverse HTTP binding for SOAP.

Interaction Service

Providers of identity services often need to interact with the owner of a resource to get additional information, or to get their consent to expose data. The Liberty Alliance Project has defined the *Liberty ID-WSF Interaction Service Specification* to specify how these interactions can be carried out. Of the options defined in the specification, OpenSSO Enterprise has implemented the Interaction RequestRedirect Profile. In this profile, the WSP requests the connecting WSC to redirect the user agent (principal) to an interaction resource (URL) at the WSP. When the user agent sends an HTTP request to get the URL, the WSP has the opportunity to present one or more pages to the principal with questions for other information. After the WSP obtains the information it needs to serve the WSC, it redirects the user agent back to the WSC, which can now reissue its original request to the WSP.

- “Configuring the Interaction Service” on page 161
- “Interaction Service API” on page 163

Configuring the Interaction Service

While there is no XML service file for the Interaction Service, this service does have properties. The properties are configured upon installation in the configuration data store and are described in the following table.

TABLE 9–8 Interaction Service Properties

Property	Description
`com.sun.identity.liberty.interaction.wspRedirectHandler`	Points to the URL where the `WSPRedirectHandler` servlet is deployed. The servlet handles the service provider side of interactions for user redirects.
`com.sun.identity.liberty.interaction.wscSpecifiedInteractionChoice`	Indicates the level of interaction in which the WSC will participate if the WSC participates in user redirects. Possible values include `interactIfNeeded`, `doNotInteract`, and `doNotInteractForData`. The affirmative `interactIfNeeded` is the default.
`com.sun.identity.liberty.interaction.wscWillIncludeUserInteractionHeader`	Indicates whether the WSC will include a SOAP header to indicate certain preferences for interaction based on the Liberty specifications. The default value is `yes`.
`com.sun.identity.liberty.interaction.wscWillRedirect`	Indicates whether the WSC will participate in user redirections. The default value is `yes`.
`com.sun.identity.liberty.interaction.wscSpecifiedMaxInteractionTime`	Indicates the maximum length of time (in seconds) the WSC is willing to wait for the WSP to complete its portion of the interaction. The WSP will not initiate an interaction if the interaction is likely to take more time than . For example, the WSP receives a request where this property is set to a maximum 30 seconds. If the WSP property `com.sun.identity.liberty.interaction.wspRedirectTime` is set to 40 seconds, the WSP returns a SOAP fault (`timeNotSufficient`), indicating that the time is insufficient for interaction.
`com.sun.identity.liberty.interaction.wscWillEnforceHttpsCheck`	Indicates whether the WSC will enforce HTTPS in redirected URLs. The Liberty Alliance Project specifications state that, the value of this property is always `yes`, which indicates that the WSP will not redirect the user when the value of `redirectURL` (specified by the WSP) is not an HTTPS URL. The `false` value is primarily meant for ease of deployment in a phased manner.
`com.sun.identity.liberty.interaction.wspWillRedirect`	Initiates an interaction to get user consent for something or to collect additional data. This property indicates whether the WSP will redirect the user for consent. The default value is `yes`.
`com.sun.identity.liberty.interaction.wspWillRedirectForData`	Initiates an interaction to get user consent for something or to collect additional data. This property indicates whether the WSP will redirect the user to collect additional data. The default value is `yes`.

TABLE 9–8 Interaction Service Properties *(Continued)*

Property	Description
`com.sun.identity.liberty.interaction.wspRedirectTime`	Indicates the length of time (in seconds) that the WSP expects to take to complete an interaction and return control back to the WSC. For example, the WSP receives a request indicating that the WSC will wait a maximum 30 seconds (set in `com.sun.identity.liberty.interaction.wscSpecifiedMaxInteractionTime`) for interaction. If the `wspRedirectTime` is set to 40 seconds, the WSP returns a SOAP fault (`timeNotSufficient`), indicating that the time is insufficient for interaction.
`com.sun.identity.liberty.interaction.wspWillEnforceHttpsCheck`	Indicates whether the WSP will enforce a HTTPS `returnToURL` specified by the WSC. The Liberty Alliance Project specifications state that the value of this property is always `yes`. The `false` value is primarily meant for ease of deployment in a phased manner.
`com.sun.identity.liberty.interaction.wspWillEnforceReturnToHostEqualsRequestHost`	Indicates whether the WSP would enforce the address values of `returnToHost` and `requestHost` if they are the same. The Liberty Alliance Project specifications state that the value of this property is always `yes`. The `false` value is primarily meant for ease of deployment in a phased manner.
`com.sun.identity.liberty.interaction.htmlStyleSheetLocation`	Points to the location of the style sheet that is used to render the interaction page in HTML.
`com.sun.identity.liberty.interaction.wmlStyleSheetLocation`	Points to the location of the style sheet that is used to render the interaction page in WML.

Interaction Service API

The OpenSSO Enterprise Interaction Service includes a Java package named `com.sun.identity.liberty.ws.interaction`. WSCs and WSPs use the classes in this package to interact with a resource owner. The following table describes the classes.

TABLE 9–9 Interaction Service Classes

Class	Description
`InteractionManager`	Provides the interface and implementation for resource owner interaction.

TABLE 9–9 Interaction Service Classes *(Continued)*

Class	Description
`InteractionUtils`	Provides some utility methods related to resource owner interaction.

For more information, including methods and their syntax and parameters, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

PAOS Binding

OpenSSO Enterprise has implemented the optional *Liberty Reverse HTTP Binding for SOAP Specification*. This specification defines a message exchange protocol that permits an HTTP client to be a SOAP responder. HTTP clients are no longer necessarily equipped with HTTP servers. For example, mobile terminals and personal computers contain web browsers yet they do not operate HTTP servers. These clients, though, can use their browsers to interact with an identity service, possibly a personal profile service or a calendar service. These identity services could also be beneficial when the client devices interact with an HTTP server. The use of PAOS makes it possible to exchange information between user agent-hosted services and remote servers. This is why the reverse HTTP for SOAP binding is also known as PAOS; the spelling of SOAP is reversed.

- "Comparison of PAOS and SOAP" on page 164
- "PAOS Binding API" on page 164

Comparison of PAOS and SOAP

In a typical SOAP binding, an HTTP client interacts with an identity service through a client request and a server response. For example, a cell phone user (client) can contact the phone service provider (service) to retrieve stock quotes and weather information. The service verifies the user's identity and responds with the requested information.

In a reverse HTTP for SOAP binding, the phone service provider plays the client role, and the cell phone client plays the server role. The initial SOAP request from the server is actually bound to an HTTP response. The subsequent response from the client is bound to a request.

PAOS Binding API

The OpenSSO Enterprise implementation of PAOS binding includes a Java package named `com.sun.identity.liberty.ws.paos`. This package provides classes to parse a PAOS header, make a PAOS request, and receive a PAOS response.

Note – This API is used by PAOS clients on the HTTP server side. An API for PAOS servers on the HTTP client side would be developed by the manufacturers of the HTTP client side products, for example, cell phone manufacturers.

The following table describes the available classes in `com.sun.identity.liberty.ws.paos`. For more detailed API documentation, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

TABLE 9–10 PAOS Binding Classes

Class	Description
`PAOSHeader`	Used by a web application on the HTTP server side to parse a PAOS header in an HTTP request from the user agent side.
`PAOSRequest`	Used by a web application on the HTTP server side to construct a PAOS request message and send it via an HTTP response to the user agent side. **Note –** `PAOSRequest` is made available in `PAOSResponse` to provide correlation, if needed, by API users.
`PAOSResponse`	Used by a web application on the HTTP server side to receive and parse a PAOS response using an HTTP request from the user agent side.
`PAOSException`	Represents an error occurring while processing a SOAP request and response.

◆ ◆ ◆ CHAPTER 10

Securing Web Services

Web services are developed using open standards such as XML, SOAP, WSDL and HTTPS. Sun Java™ System OpenSSO Enterprise provides the functionality to secure web services communications using authentication agents and the Security Token Service. This chapter contains the following sections:

About Web Services Security

A *web service* is an application whose functionality and interfaces are exposed through open technology standards including the eXtensible Markup Language (XML), SOAP, the Web Service Description Language (WSDL) and HTTP(S). A web service client (WSC) sends a SOAP message to the endpoint (identified by a URI) of a web service provider (WSP); after receiving the request, the WSP responds appropriately with a SOAP response. The built-in openness of these technologies though creates security risks. Initially, securing these web services communications was addressed using *transport level security* in which the complete message was encrypted and transmitted using Secure Sockets Layer (SSL) with mutual authentication. But with current enterprise topologies (including proxies, load balancers, data centers, and the like) security must now be addressed when intermediaries are involved. Web services must be prepared to:

- Pass fine-grained security data (for example, identity attributes for authorization).
- Enable one or more trusted authorities to broker trust between communicating entities.
- Maintain security on a per message basis.
- Maintain transport layer independence.

These requirements call for *message level security* (also referred to as *application level security* and *end-to-end security*) in which only the content of the message is encrypted. Message level security embeds all required security information in a message's SOAP header. Additionally, encryption and digital signatures can be applied to the data itself. The advantages of message level security are that:

- Security stays with the message through all intermediaries, across domain boundaries, and after the message arrives at its destination.
- Security can be selectively applied to different portions of the message.
- Security is independent of the application environment and transport protocol.

To address message level security in web services communications, organizations such as the Organization for Advancement of Structured Information Standards (OASIS), the Liberty Alliance Project and the Java Community Process (JCP) have proposed specifications based on open standards and from them OpenSSO Enterprise has implemented "The Security Token Service" on page 172 using the WS-Trust specification and "Security Agents" on page 174.

About Web Services Security with OpenSSO Enterprise

Web services are accessed by sending SOAP messages to service endpoints identified by URIs, and receiving SOAP message responses. Towards this end, OpenSSO Enterprise has implemented a Security Token Service and agents to enforce security. This architecture is illustrated below.

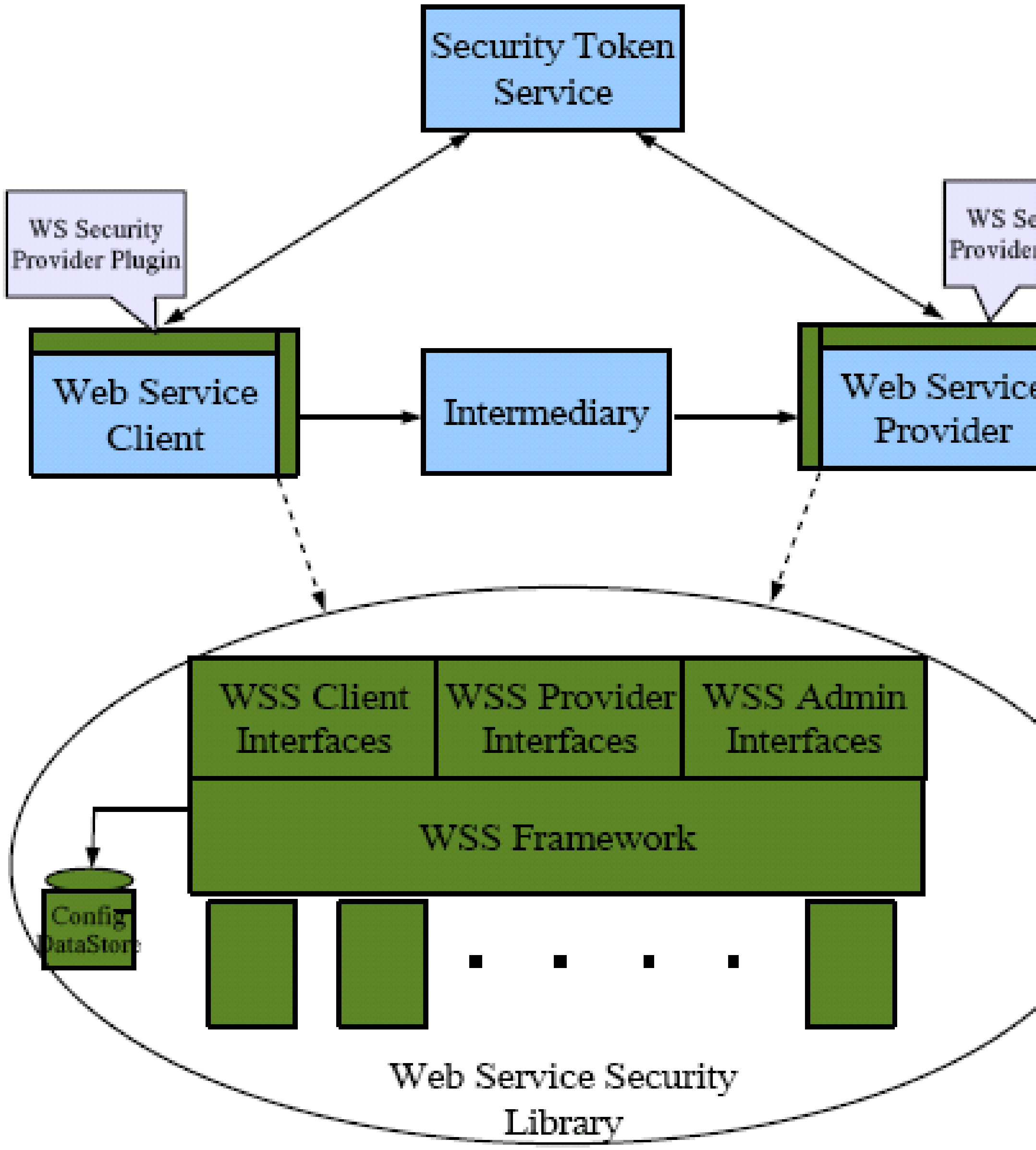


FIGURE 10–1 Web Services Security Architecture in OpenSSO Enterprise

The top half of the diagram illustrates a typical web services communication with the addition of agents (WS-Security Provider Plug-in) to enforce message level security and a security token service to issue security tokens. The bottom half represents OpenSSO Enterprise and the interfaces that are called by the WSC and WSP to accomplish web services security. The agents provide access to OpenSSO Enterprise (using the Client SDK) to secure and validate the SOAP requests and responses.

When using web services security, the outgoing web service messages and the incoming web service calls must be authenticated and authorized. Towards this end, the messages must be modified to add headers containing credentials for those purposes. Additional identity attributes (for example, the roles and memberships) can also be added and used by the web service provider's agent and/or by the web service's business logic to provide appropriate service. The authentication and authorization by the agent at the web service provider would leverage the OpenSSO Enterprise Authentication Service and Policy Service. For authentication, it extracts the authentication credentials from the web service request and calls the appropriate authentication module for validation. For authorization, the web service's endpoint port and the operation being performed is the resource for the defined policy.

Security agents are deployed on both the WSC side and the WSP side of the communication. OpenSSO Enterprise contains interfaces with which the agents (deployed remotely to the server) can communicate. The WSC which makes the web service call provides support for securing outgoing communications and validating incoming responses from the WSP. There are two kinds of interfaces used by the WSC, one for administration and another used at run time for securing and validating requests and responses. The WSP which provides service based on calls from the WSC provides support for validating incoming requests and secure outgoing responses. Similar to the WSC, the WSP has an administration interface and an interface used at run time for securing and validating requests and responses. There are also administrative interfaces to configure (local to OpenSSO Enterprise) the Security Token Service and the respective security mechanisms supported by the WSC and WSP. These configurations are stored in the OpenSSO Enterprise configuration data store.

The following diagram illustrates support for web services security in OpenSSO Enterprise. The Security Token Service is supported with any party that understands the WS-Trust specification on which it is based. On the WSC side, an agent developed using the JSR-196 specification is supported on Glassfish (Sun Application Server). (Currently there are no other WSC supported agents although custom handlers and filters can be developed.) On the WSP, the same JSR-196 agent is supported on the Glassfish (Sun Application Server) while Sun policy agents 3.0 are supported on WebLogic, WebSphere and Tomcat.

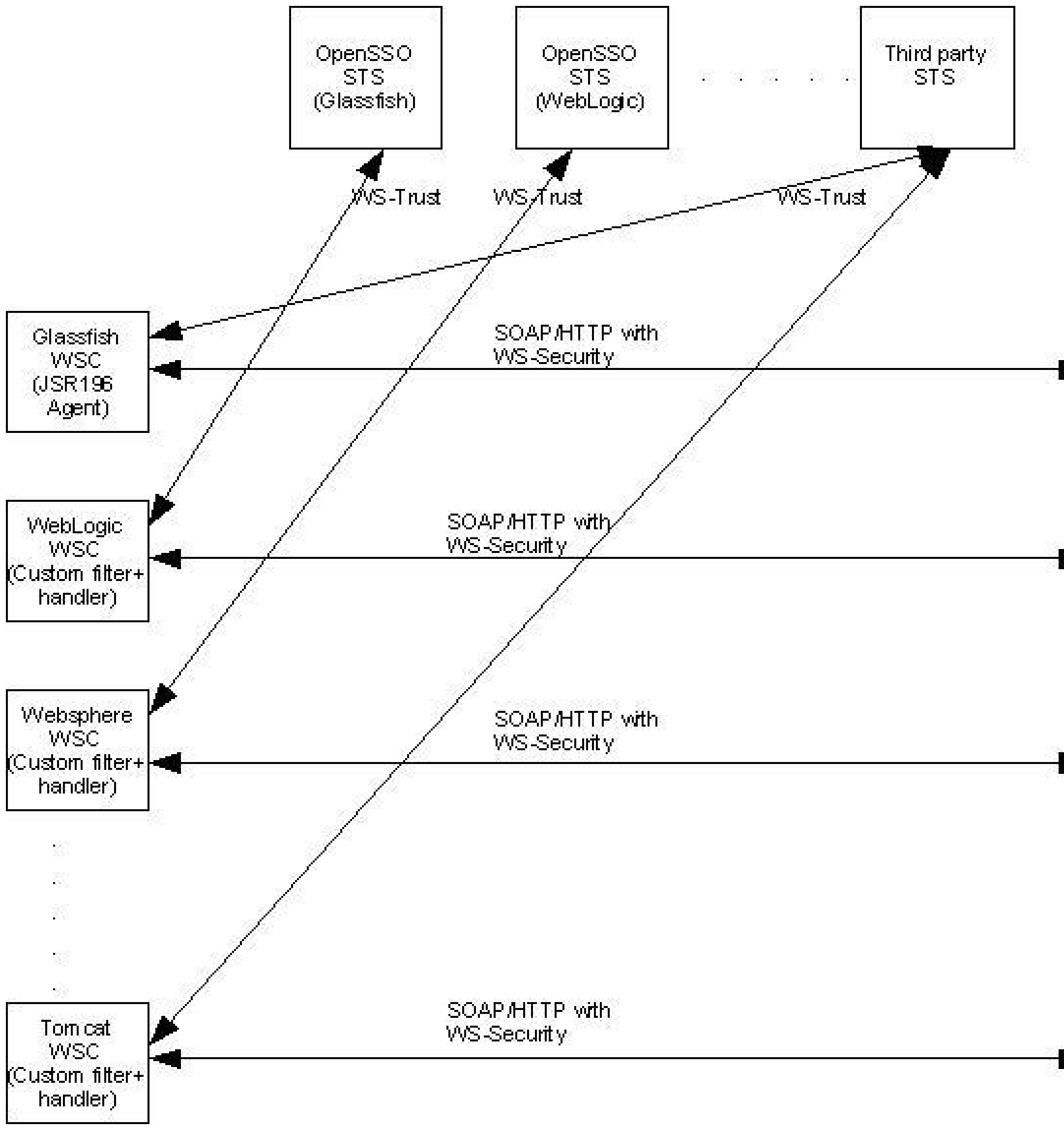

OpenSSO STS (Glassfish)
OpenSSO STS (WebLogic)
Third party STS
WS-Trust
WS-Trust
WS-Trust
Glassfish WSC (JSR196 Agent)
SOAP/HTTP with WS-Security
WebLogic WSC (Custom filter+ handler)
SOAP/HTTP with WS-Security
Websphere WSC (Custom filter+ handler)
SOAP/HTTP with WS-Security
Tom cat WSC (Custom filter+ handler)
SOAP/HTTP with WS-Security

FIGURE 10–2 Web Services Security Support in OpenSSO Enterprise

See "The Security Token Service" on page 172 and "Security Agents" on page 174 for more information.

The Security Token Service

When a WSC communicates with a WSP it must first connect with a trusted authority to determine the security mechanism and, optionally, obtain the security token expected by the WSP. This information is registered with the trusted authority by the WSP. The Security Token Service is a trusted authority that provides issuance and management of security tokens; that is, it makes security statements or claims often, although not required to be, in cryptographically protected sets. The OpenSSO Enterprise trust brokering process is as follows.

1. An authenticated WSC requests a token to access a particular WSP.
2. The Security Token Service verifies the credentials presented by the WSC.
3. In response to an affirmative verification, the Security Token Service issues a security token that provides proof that the client has been authenticated.
4. The WSC presents the security token to the WSP.
5. The WSP verifies that the token was issued by a trusted Security Token Service, affirming authentication and authorizing access.

The Security Token Service communicates using the WS-Trust protocol and serves WS-I BSP security tokens. (Any WSC or WSP can communicate remotely with OpenSSO Enterprise Security Token Service using the WS-Trust protocol.) The Security Token Service also serves as a Discovery Service, able to communicate using the Liberty ID-WSF protocol and serve Liberty Alliance Project security tokens.

- "Web Container Support" on page 172
- "Security Tokens" on page 173
- "Token Conversion" on page 173
- "Configuring the Security Token Service" on page 174

Web Container Support

OpenSSO Enterprise as a Security Token Service is supported on different web containers including:

- Glassfish (Sun Application Server 9.x)
- Sun Web Server 7.x
- WebLogic
- Websphere
- Tomcat
- Oracle Application Server
- JBoss
- Geronimo

With this support, any WSC or WSP can communicate remotely with OpenSSO Enterprise Security Token Service using the WS-Trust protocol.

Security Tokens

The Security Token Service issues, renews, cancels, and validates security tokens that can contain an identifier for either the WSC or the actual end user. It also allows you to write a proprietary token providers using the included service provider interfaces (SPI). Finally, it provides application programming interfaces (API), based on the WS-Trust protocol, that allow applications to access the service. By default, the Security Token Service serves tokens based on the Liberty Alliance Project and WS-Trust specifications. The WS-I BSP specifications and the Liberty Alliance Project developed security profiles for web services security. These security mechanism are implemented for web services security using the provider interfaces. The following list contains the supported mechanisms.

Anonymous	Carries no security information.
User Name Token	Carries basic information (username and, optionally, a password or shared secret) for purposes of authenticating the user identity to the WSP. Communication is done in plain text so SSL over HTTPS transport must be used to protect the credentials.
User Name Token-Plain	Carries basic information (username and a clear text password or shared secret) for purposes of authenticating the user identity to the WSP. Communication is done in plain text so SSL over HTTPS transport must be used to protect the credentials.
Kerberos Token	Carries basic information (username and, optionally, a password or shared secret), in a Kerberos token, for purposes of authenticating the user identity to the WSP.
X.509 Token	Contains an X.509 formatted certificate for authentication using credentials created with a public key infrastructure (PKI). In this case, the WSC and WSP must trust each other's public keys or share a common, trusted certificate authority.
SAML-Holder-Of-Key Token	Uses the SAML *holder-of-key* confirmation method whereby the WSC supplies a SAML assertion with public key information as the means for authenticating the requester to the web service provider. A second signature binds the assertion to the SOAP payload. Can use either SAML v1.x or SAML v2.
SAML-SenderVouches Token	Uses the SAML *sender-vouches* confirmation method whereby the WSC adds a SAML assertion and a digital signature to a SOAP header. A sender certificate or public key is also provided with the signature. Can use either SAML v1.x or SAML v2.

Token Conversion

The Security Token Service is able to convert from one token format to another. For example, an OpenSSO Enterprise `SSOToken` can be converted to a SAML v2 token or a SAML v1.x token to a SAML v2 token. Token conversion plug-ins can be developed using the token conversion interface in the `com.sun.identity.wss.sts` package.

Configuring the Security Token Service

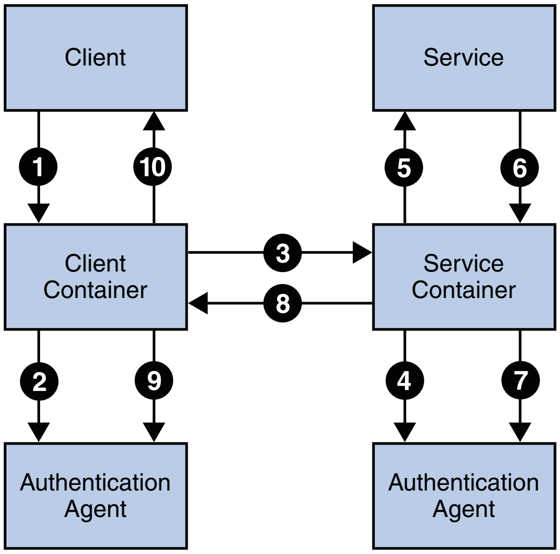
To configure a WSC to communicate with the Security Token Service end point (by default, **`http://`***server***`:`***port***`/opensso/sts`**), download and deploy the Client SDK WAR and see the README and samples. To protect the Security Token Service, login to the console and click the Configuration tab. Following, click the Global tab and the Security Token Service link for security configurations. The Security Token Service WSDL is `fam.sts`.

Security Agents

There are two kinds of security agents developed for web services security. One protects the WSC and the other protects the WSP. The WSC which makes the web service call provides support for securing the outgoing communications and validating the incoming responses from a WSP. The WSP which provides a service from a WSC call provides support for validating the incoming request and securing the outgoing responses. These agents may establish the authenticated identities used by the containers allowing:

- A server side agent to verify security tokens or signatures on incoming requests and extract principal data or assertions before adding them to the client security context.
- A client side agent to add security tokens to outgoing requests, sign messages, and interact with the trusted authority to locate targeted web service providers.

A typical interaction between a WSC and a WSP begins with a request from the WSC. The container on which the WSP is deployed receives the request and dispatches it to perform the requested operation. When the web service completes the operation, it creates a response that is returned back to the client. The following illustration and procedure illustrates a scenario when both client and service web containers employ the Java Authentication SPI.



1. The client browser's attempt to invoke a web service is intercepted by the client's web container.
2. The deployed security agent on the client's web container is invoked to secure the request (based on the security policy of the web service being invoked).
3. The client's web container sends the secured request message to the web service.
4. The web service's web container receives the secured request message and it's deployed security agent is invoked to validate the request and obtain the identity of the caller.
5. Assuming successful authentication, the web service's web container invokes the requested web service.
6. This action (the invocation of the web service) is returned to the web service's web container as a response.
7. The deployed security agent on the web service's web container is invoked to secure the response message.
8. The web service's web container sends the secured response message to the client.
9. The deployed security agent on the client's web container is invoked to validate the secured response message.
10. The invocation of the web service is returned to the client browser.

Security processes can be delegated to a security agent at any of the following interaction points.

- Securing a request on the client side
- Validating a request on the provider side
- Securing a response on the provider side
- Validating a response on the client side

This security agent uses an instance of OpenSSO Enterprise for all authentication decisions. Web services requests and responses are passed to the authentication modules using standard Java representations based on the transmission protocol. Currently, the following security agents are provided.

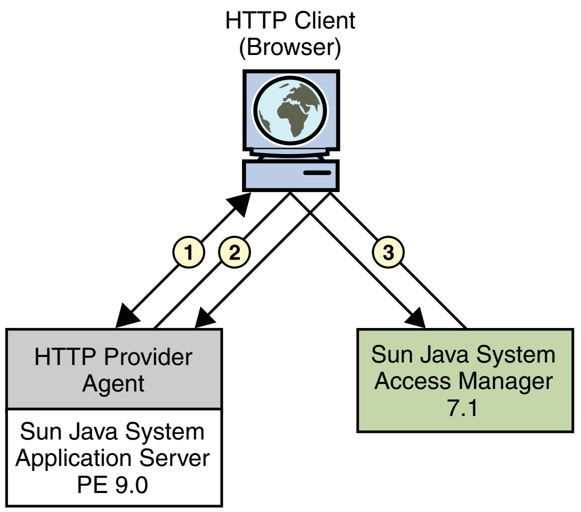
- "WSC Security Agents" on page 176
- "WSP Security Agent" on page 177

WSC Security Agents

The WSC security agent protects the endpoints of a web service that uses HTTP for communication. After the WSC security agent is deployed in a web container on the WSP side, all HTTP requests for access to the web services protected by the agent are redirected to the login and authentication URLs defined in the OpenSSO Enterprise configuration data store on the WSC side.

Note – The available WSC security agent was developed using the Java Specification Request (JSR) 196. JSR 196 is the *Java Authentication Service Provider Interface for Containers*. It defines a standard service provider interface (SPI) with which a security agent can be developed to police Java EE containers on either the client side or the server side. These agents establish the authenticated identities used by the containers. The JSR 196 specifications are available at `http://www.jcp.org/en/jsr/detail?id=196`.

When the WSC makes a request to access a web application (**1** in the illustration below), the agent intercepts the request and redirects it (via the browser) to OpenSSO Enterprise for authentication (**2**). Upon successful authentication, a response is returned to the application, carrying a token as part of the Java EE Subject (**3**). This token is used to bootstrap the appropriate Liberty ID-WSF security profile. If the response is successfully authenticated, the request is granted (**3**).


HTTP Client (Browser)
1
2
3
HTTP Provider Agent
Sun Java System Application Server PE 9.0
Sun Java System Access Manager 7.1

Note – The functionality of the HTTP security agent is similar in to that of the Java EE policy agents when used in SSO ONLY mode. This is a non restrictive mode that uses only the OpenSSO Enterprise Authentication Service to authenticate users attempting access. For more information on Java EE policy agents, see the *Sun Java System Access Manager Policy Agent 2.2 User's Guide*.

Note – Application Server 9 has the ability to configure only one HTTP agent per instance. Therefore, all authentication requests for all web applications hosted in the container will be forwarded to the one configured agent.

WSP Security Agent

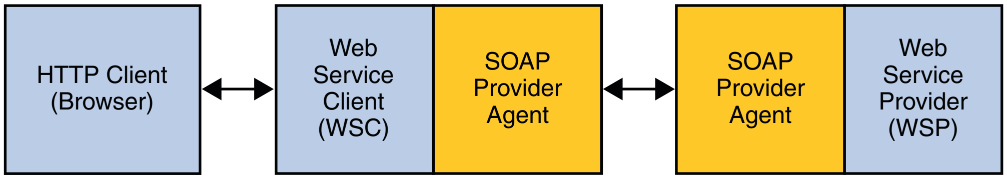
The WSP which provides a service based on calls from a WSC provides support for validating incoming requests and securing outgoing responses. This agent encapsulates the Web Services-Interoperability Basic Security Profile (WS-I BSP) tokens as well as the Liberty Identity Web Services Framework (Liberty ID-WSF) SOAP Binding Specification tokens:

- "Supported Web Services-Interoperability Basic Security Profile Security Tokens" on page 178
- "Supported Liberty Alliance Project Security Tokens" on page 179

Supported Web Services-Interoperability Basic Security Profile Security Tokens

In a scenario where security is enabled using Web Services-Interoperability Basic Security Profile (WS-I BSP) tokens, the client requests access to a service. The configured security agent reads the configuration from the OpenSSO Enterprise configuration data store and redirects the request to the OpenSSO Enterprise Authentication Service for authentication and to determine the security mechanism registered by the WSP and obtain the expected security tokens. After a successful authentication, the WSC provides a SOAP body while the SOAP security agent on the WSC side inserts the security header and a token. The message is then signed before the request is sent to the WSP.

When received by the security agent on the WSP side, the signature and security token in the SOAP request are verified before forwarding the request on to the WSP itself. The WSP then processes it and returns a response, signed by the security agent on the WSP side, back to the WSC. The SOAP security agent on the WSC side then verifies the signature before forwarding the response on to the WSC. The following diagram illustrates the interactions as described.



The following WS-I BSP security tokens are supported in this release.

User Name A secure web service requires a user name, password and, optionally, a signed the request. The web service consumer supplies a username token as the means for identifying the requester and a password, shared secret, or password equivalent to authenticate the identity to the web service provider.

X.509 A secure web service uses a PKI (public key infrastructure) in which the web service consumer supplies a public key as the means for identifying the requester and accomplishing authentication with to the web service provider.

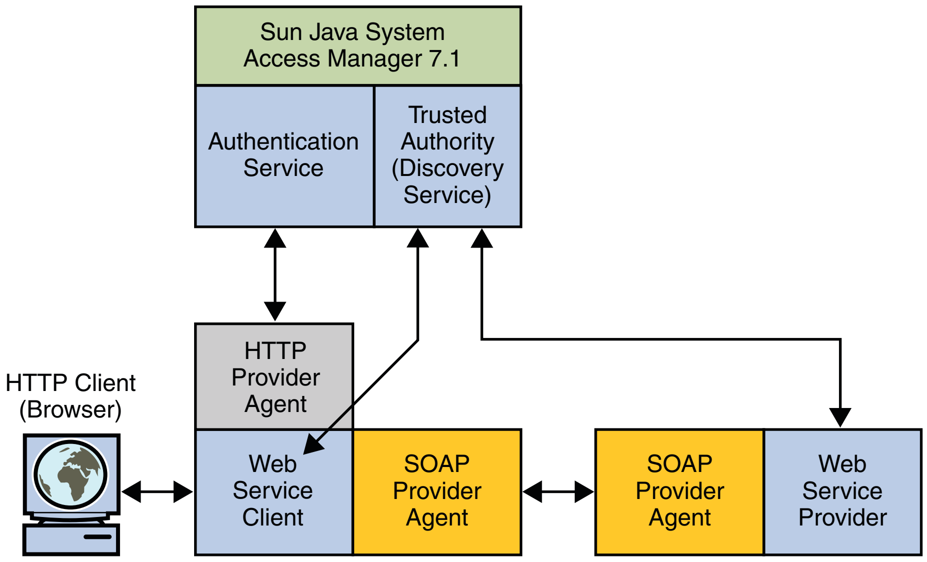
SAML-Holder-Of-Key A secure web service uses the SAML *holder-of-key* confirmation method. The web service consumer supplies a SAML assertion with public key information as the means for authenticating the requester to the web service provider. A second signature binds the assertion to the SOAP payload.

SAML-SenderVouches A secure web service uses the SAML *sender-vouches* confirmation method. The web service consumer adds a SAML assertion and a digital signature to a SOAP header. A sender certificate or public key is also provided with the signature.

Supported Liberty Alliance Project Security Tokens

In a scenario where security is enabled using Liberty Alliance Project tokens, the client requests (via the WSC) access to a service. The security agent redirects the request to the OpenSSO Enterprise Authentication Service for authentication and to determine the security mechanism registered by the WSP and obtain the security tokens expected. After a successful authentication, the WSC provides a SOAP body while the SOAP security agent on the WSC side inserts the security header and a token. The message is then signed before the request is sent to the WSP.

When received by the SOAP security agent on the WSP side, the signature and security token in the SOAP request are verified before forwarding the request on to the WSP itself. The WSP then processes it and returns a response, signed by the SOAP security agent on the WSP side, back to the WSC. The SOAP security agent on the WSC side then verifies the signature before forwarding the response on to the WSC. The following diagram illustrates the interactions as described.



The following Liberty Alliance Project security tokens are supported in this release:

X.509 A secure web service uses a PKI (public key infrastructure) in which the web service consumer supplies a public key as the means for identifying the requester and accomplishing authentication with the web service provider. Authentication with the web service provider using processing rules defined by the Liberty Alliance Project.

BearerToken	A secure web service uses the Security Assertion Markup Language (SAML) *SAML Bearer token* confirmation method. The web service consumer supplies a SAML assertion with public key information as the means for authenticating the requester to the web service provider. A second signature binds the assertion to the SOAP message This is accomplished using processing rules defined by the Liberty Alliance Project
SAMLToken	A secure web service uses the SAML *holder-of-key* confirmation method. The web service consumer adds a SAML assertion and a digital signature to a SOAP header. A sender certificate or public key is also provided with the signature. This is accomplished using processing rules defined by the Liberty Alliance Project.

Testing Web Services Security

OpenSSO Enterprise provides two samples that can be used to test web services security. The Stock Quote Sample and the Loan Service Sample are available in the WSS Agent download available on OpenSSO Downloads. The Stock Quote Sample is for simple web services security. It focuses on building a WSP and a WSC, authenticating the WSC before access to the service is given, and guaranteeing the integrity of the authentication data. The Loan Service Sample is an advanced test case where Security Token Service brokerage is demonstrated.

CHAPTER 11

Creating and Deploying OpenSSO Enterprise WAR Files

Sun™ OpenSSO Enterprise is distributed as a web archive (WAR) file named `opensso.war`. In addition to deploying OpenSSO Enterprise server, you can also use `opensso.war` to create a customized server WAR file and specialized WAR files for an OpenSSO Enterprise Distributed Authentication UI server, the IDP Discovery Service, OpenSSO Enterprise Administration Console only, and OpenSSO Enterprise server without the Administration Console. This chapter describes these sections:

- "Overview of WAR Files in Java EE Software Development" on page 181
- "Deploying the OpenSSO Enterprise WAR File" on page 182
- "Customizing and Redeploying `opensso.war`" on page 185
- "Creating Specialized OpenSSO Enterprise WAR Files" on page 185

Overview of WAR Files in Java EE Software Development

OpenSSO Enterprise is built on the Java EE platform, which uses a component model to create full-scale applications. A component is self-contained functional software code assembled with other components into a Java EE application. The Java EE application components can be deployed separately on different servers. Java EE application components include the following:

- Client components such as including dynamic web pages, applets, and a Web browser that run on the client machine.
- Web components such as servlets and Java Server Pages (JSPs) that run within a web container.
- Business components that meets the needs of a particular enterprise domain such as banking, retail, or finance. Such business components also run within a web container.
- Enterprise infrastructure software that runs on legacy machines.

Web Components

When a web browser executes a Java EE application, it deploys server-side objects known as web components. JSP and corresponding servlets are two such web components.

Servlets	Small Java programs that dynamically process requests and construct responses from a web browser. Servlets run within web containers.
Java Server Pages (JSPs)	Text-based documents that contain static template data such as HTML, Scalable Vector Graphics (SVG), Wireless Markup Language (WML), or eXtensible Markup Language (XML). JSPs also contain elements such as servlets that construct dynamic content.

How Web Components are Packaged

Java EE components are usually packaged separately, and then bundled together into an Enterprise Archive (EAR) file for application deployment. Web components are packaged in WAR files. Each WAR file contains servlets, JSPs, a deployment descriptor, and related resource files.

Static HTML files and JSP are stored at the top level of the WAR directory. The top-level directory contains the `WEB-INF` subdirectory which contains tag library descriptor files in addition to the following:

Server-side classes	Servlets, JavaBean components and related Java class files. These must be stored in the `WEB-INF/classes` directory.
Auxiliary JARs	Tag libraries and any utility libraries called by server-side classes. These must be stored in the `WEB-INF/lib` directory.
`web.xml`	The web component deployment descriptor is stored in the WEB-INF directory

Deploying the OpenSSO Enterprise WAR File

- "OpenSSO Enterprise Deployment Considerations" on page 183
- "To Deploy the OpenSSO Enterprise Server WAR File:" on page 183

OpenSSO Enterprise Deployment Considerations

Before you deploy the OpenSSO Enterprise WAR file, here are a few changes to consider from previous releases of Access Manager and Federation Manager:

- You deploy OpenSSO Enterprise from the `opensso.war` file, using the web container administration console or deployment command. You no longer run the Java Enterprise System installer.
- You initially configure OpenSSO Enterprise using either the GUI or command-line Configurator. Then, to perform additional configuration, you use either the Administration Console or the new `ssoadm` command-line utility. You no longer run the `amconfig` script using variables in the the `amsamplesilent` file as input.
- Configuration data, including policy agent configuration data, is stored in a centralized repository. This repository can be either Sun Java System Directory Server or the OpenSSO data store (which is usually transparent to the user). OpenSSO Enterprise server does not use the `AMConfig.properties` or `serverconfig.xml` files, except for co-existence with previous versions of Access Manager.

▼ To Deploy the OpenSSO Enterprise Server WAR File:

The following procedure summarizes the OpensSSO Enterprise WAR file deployment. Links are provided to the detailed steps in the *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*.

1 **If necessary, install, configure, and start one of the supported web containers listed in Chapter 2, "Deploying the OpenSSO Enterprise Web Container," in *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*.**

2 **Download and unzip the `opensso_enterprise_80.zip` file from one of the following sites:**

- Sun: `http://www.sun.com/software/products/opensso_enterprise`

  or

- OpenSSO site: `http://opensso.dev.java.net/public/use/index.html`

Be sure to check the *Sun OpenSSO Enterprise 8.0 Release Notes*for any current issues.

3 **Deploy the `opensso.war` file to the web container, using the web container administration console or deployment command.**

Detailed steps are in Chapter 3, "Installing OpenSSO Enterprise," in *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*.

4 **Run either the GUI or command-line Configurator.**

To run the GUI Configurator, enter the following URL in your browser:

protocol`://`*host*`.`*domain*`:`*port*`/`*deploy-uri*

For example: `https://opensso.example.com:58080/opensso`

If you are running the GUI Configurator, enter values in the Configurator fields or accept the default value for some fields. The GUI Configurator has two configuration options:

- The **Default Configuration** option requires you to enter only the OpenSSO Enterprise administrator (`amAdmin`) and default policy agent (`UrlAccessAgent`) passwords. The Configurator then uses default values for the other configuration options.

  Use the Default Configuration for development environments or simple demonstration purposes when you just want to evaluate OpenSSO Enterprise features.
- The **Custom Configuration** option allows you to enter specific configuration values for your deployment (or accept the default values).

  Use the Custom Configuration for production and more complex environments. For example, a multi-server installation with several OpenSSO Enterprise instances behind a load balancer.

Detailed steps for configuring OpenSSO Enterprise are in:

- Chapter 4, "Configuring OpenSSO Enterprise Using the GUI Configurator," in *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*

  or
- Chapter 5, "Configuring OpenSSO Enterprise Using the Command-Line Configurator," in *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*

5 **Launch OpenSSO Enterprise using the specific web container console or deployment command, or by specifying the URL from Step 4 in your browser.**

6 **Login to the Console as the OpenSSO Enterprise administrator (`amadmin`) using the password you specified when you ran the Configurator.**

7 **To make additional configuration changes to your deployment, use the OpenSSO Administration Console or the `ssoadm` command-line utility.**

For information, refer to the OpenSSO Administration Console Online Help or the *Sun OpenSSO Enterprise 8.0 Administration Reference*.

Customizing and Redeploying `opensso.war`

The `opensso.war` file contains all OpenSSO Enterprise components. To customize OpenSSO Enterprise, you must update and redeploy this file.

If you have not already done so, download and unzip the `opensso_enterprise_80.zip` file. The `opensso.war` file is then in the *zip-root*`/deployable-war` directory, where *zip-root* is where you unzipped the file.

▼ To Customize and Redeploy `opensso.war`

1 **Make sure that your `JAVA_HOME` environment variable points to a JDK of version 1.5 or later.**

2 **Create a staging directory for your customized WAR file. For example:**

```
# mkdir customized-opensso
```

3 **In the staging directory, extract the files from `opensso.war`:**

```
# cd customized-opensso
# jar xvf zip-root/opensso/deployable-war/opensso.war
```

4 **Customize the files required for your deployment.**

5 **Create the new customized WAR file:**

```
# cd customized-opensso
# jar cvf zip-root/opensso/deployable-war/customized-opensso.war
```

In this example. `customized-opensso.war` is the name of the new customized OpenSSO Enterprise WAR file.

6 **Deploy and configure the new customized OpenSSO WAR file in your specific web container, as described in the *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*.**

Creating Specialized OpenSSO Enterprise WAR Files

You can use the `opensso.war` file to create these specialized WAR files:

- Distributed Authentication UI server
- OpenSSO Administration Console only
- OpenSSO Enterprise server without the Administration Console
- IDP Discovery Service

If you have not already done so, download and unzip the `opensso_enterprise_80.zip` file. You will then need the following files in the *zip-root*`/deployable-war` directory to create a specialized WAR file, where *zip-root* is where you unzipped the `opensso_enterprise_80.zip` file:

- `opensso.war` contains all OpenSSO Enterprise components.
- `fam-distauth.list`, `fam-console.list`, `fam-noconsole.list`, or `fam-idpdiscovery.list` contain a list of files required to create a specialized WAR file.
- `distauth`, `console`, `noconsole`, and `idpdiscovery` directories contains the additional files you will need to create, deploy, and configure a specialized WAR file.

▼ To Create a Specialized OpenSSO Enterprise WAR File

1 Make sure that your `JAVA_HOME` environment variable points to a JDK of version 1.5 or later.

2 Create a staging directory and extract the files from `opensso.war` in this staging directory. For example:

```
# mkdir opensso-staging
# cd opensso-staging
# jar xvf zip-root/opensso/deployable-war/opensso.war
```

3 Create the new specialized WAR file, as follows:

```
# cd opensso-staging
# jar cvf zip-root/opensso/deployable-war/new-war-filename.war \
   @zip-root/opensso/deployable-war/war-file.list
```

- *new-war-filename* is the name of the new WAR file. For example: `opensso-distauth.war`, `opensso-idpdiscovery.war`, `opensso-consoleonly.war`, or `opensso-noconsole.war`.

  Note: Some web containers require the Distributed Authentication UI server WAR file name to use the same name as the deployment URI. Check with your web container documentation for more information.
- *war-file*`.list` specifies the list of files required for the new WAR file, as follows: `fam-distauth.list`, `fam-console.list`, or `fam-noconsole.list`, or `fam-idpdiscovery.list`.

4 Update the WAR file created in previous step with the additional files required for new specialized WAR file. For example:

```
# cd zip-root/opensso/deployable-war/specialized-files-directory
# jar uvf zip-root/opensso/deployable-war/new-war-filename.war *
```

- *specialized-files-directory* specifies the directory where the additional files reside:
  - `distauth`

  - `console`
  - `noconsole`
  - `idpdiscovery`
- *new-war-filename* is the name of the new specialized WAR file.

Next Steps You are now ready to deploy and configure the new specialized WAR file. For the detailed steps, see the following chapters:

- Chapter 8, "Deploying a Distributed Authentication UI Server," in *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*
- Chapter 9, "Deploying the Identity Provider (IDP) Discovery Service," in *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*
- Chapter 10, "Installing the OpenSSO Enterprise Console Only," in *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*
- Chapter 11, "Installing OpenSSO Enterprise Server Only," in *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*

CHAPTER 12

Customizing the Authentication User Interface

The Sun™ OpenSSO Enterprise Authentication Service provides a web-based graphical user interface (GUI) for all default and custom authentication modules installed in a deployment. This interface provides a dynamic and customizable means for gathering authentication credentials by presenting the web-based login pages to a user requesting access.

The Authentication Service GUI is built on top of JATO (J2EE Assisted Take-Off), a Java Enterprise Edition (Jave EE) presentation application framework. This framework is used to help developers build complete functional Web applications. You can customize this user interface per client type, realm, locale, or service.

This chapter includes the following sections:

For more information about the Authentication Service, see Part II, "Access Control Using OpenSSO Enterprise," in *Sun OpenSSO Enterprise 8.0 Technical Overview*.

User Interface Files You Can Modify

The authentication GUI dynamically displays the required credentials information depending upon the authentication module invoked at run time. The following table lists the types of files you can modify to customize the login pages, logout pages, and error messages. Detailed information is provided in subsequent sections.

TABLE 12–1 Authentication User Interface Files and Their Locations at Installation

File Type	Default Location
"Java Server Page (JSP) Files" on page 190	See *OpenSSO-Deploy-base*`/config/auth/default`
"XML Files" on page 193	See *OpenSSO-Deploy-base*`/config/auth/default`
"JavaScript Files" on page 196	See *OpenSSO-Deploy-base*`/js`
"Cascading Style Sheets" on page 196	See *OpenSSO-Deploy-base*`/css`
"Images" on page 197	See *OpenSSO-Deploy-base*`/login_images`
"Localization Files" on page 198	See *OpenSSO-Deploy-base*`/WEB-INF/classes`
OpenSSO-Deploy-base represents the deployment directory where the web container deploys the `opensso.war` file	

Java Server Page (JSP) Files

The authentication GUI pages are `.jsp` files with embedded JATO tags. You do not need to understand JATO to customize the GUI pages. Java server pages handle both the UI elements and the disciplines displayed through peer ViewBeans.

By default, JSP pages are installed and looked up in the following directory:

OpenSSO-Deploy-base`/config/auth/default`

Customizing the Login Page

The login page is a common page used by most authentication modules except for the Membership module. For all other modules, at run time the login page dynamically displays all necessary GUI elements for the user to enter the required credentials. For example, the LDAP authentication module login page dynamically displays the LDAP module header, LDAP user name, and password fields.

To access the default login page, use the following URL:

server-protocol`://`*server-host*`.`*server-domain*`:`*server-port*`/`*service-deploy-uri*`/UI/Login`

To access the default logout page, use the following URL:

server-protocol`://`*server-host*`.`*server-domain*`:`*server-port*`/`*service-deploy-uri*`/UI/Logout`

You can customize the following login page UI elements:

- Module Header text
- User Name label and field

- Password label and field
- Choice value label and field.

  The field is a radio button by default, but can be change to a check box.
- Image (at the module level)
- Login button

Customizing JSP Templates

Use the JSP templates to customize the look and feel presented in the graphical user interface (GUI). "Customizing JSP Templates" on page 191 provides descriptions of templates you can customize. The templates are located in the following directory:

OpenSSO-Deploy-base`/config/auth/default`

TABLE 12–2 Customizable JSP Templates

File Name	Purpose
`account_expired.jsp`	Informs the user that their account has expired and should contact the system administrator.
`auth_error_template.jsp`	Informs the user when an internal authentication error has occurred. This JSP usually indicates an authentication service configuration issue.
`authException.jsp`	Informs the user that an error has occurred during authentication.
`configuration.jsp`	Configuration error page that displays during the Self-Registration process.
`disclaimer.jsp`	Customizable disclaimer page used in the self-registration authentication module.
`Exception.jsp`	Informs the user that an error has occurred.
`invalidAuthlevel.jsp`	Informs the user that the authentication level invoked was invalid.
`invalid_domain.jsp`	Informs the user that no such domain exists.
`invalidPassword.jsp`	Informs the user that the password entered does not contain enough characters.
`invalidPCookieUserid.jsp`	Informs the user that a persistent cookie user name does not exist in the persistent cookie domain.
`Login.jsp`	This is a login and password template.
`login_denied.jsp`	Informs the user that no profile has been found in this domain.
`login_failed_template.jsp`	Informs the user that authentication has failed.
`Logout.jsp`	Informs the user that they have logged out.

TABLE 12–2 Customizable JSP Templates *(Continued)*

File Name	Purpose
`maxSessions.jsp`	Informs the user that the maximum sessions have been reached.
`membership.jsp`	A login page for the self-registration module.
`Message.jsp`	A generic message template for a general error not defined in one of the other error message pages.
`missingReqField.jsp`	Informs the user that a required field has not been completed.
`module_denied.jsp`	Informs the user that the user does not have access to the module.
`module_template.jsp`	Customizable module page.
`new_org.jsp`	Displayed when a user with a valid session in one organization wants to login to another organization.
`noConfig.jsp`	Informs the user that no module configuration has been defined.
`noConfirmation.jsp`	Informs the user that the password confirmation field has not been entered.
`noPassword.jsp`	Informs the user that no password has been entered.
`noUserName.jsp`	Informs the user that no user name has been entered. It links back to the login page.
`noUserProfile.jsp`	Informs the user that no profile has been found. It gives them the option to try again or select New User and links back to the login page.
`org_inactive.jsp`	Informs the user that the organization they are attempting to authenticate to is no longer active.
`passwordMismatch.jsp`	Called when the password and confirming password do not match.
`profileException.jsp`	Informs the user that an error has occurred while storing the user profile.
`Redirect.jsp`	Includes a link to a page that has been moved.
`register.jsp`	User self-registration page.
`session_timeout.jsp`	Informs the user that their current login session has timed out.
`userDenied.jsp`	Informs the user that they do not possess the necessary role (for role-based authentication.)
`userExists.jsp`	Called if a new user is registering with a user name that already exists.
`user_inactive.jsp`	Informs the user that they are not active.
`userPasswordSame.jsp`	Called if a new user is registering with a user name field and password field have the same value.
`wrongPassword.jsp`	Informs the user that the password entered is invalid.

XML Files

XML files describe the authentication module-specific properties based on the Authentication Module properties DTD file:

OpenSSO-Deploy-base`/WEB-INF/Auth_Module_Properties.dtd`

OpenSSO Enterprise defines required credentials and callback information for each of the default authentication modules. By default, authentication XML files are installed in the following directory:

OpenSSO-Deploy-base`/config/auth/default`

The following table provides descriptions of the authentication module configuration files.

TABLE 12–3 Authentication Module Configuration XML Files

File Name	Description
`AD.xml`	Defines a Login screen for use with Active Directory authentication.
`amAuthUnix.xml`	Defines a Login screen for use with Unix authentication
`Anonymous.xml`	For anonymous authentication, although there are no specific credentials required to authenticate.
`Application.xml`	Needed for application authentication.
`Cert.xml`	For certificate-based authentication although there are no specific credentials required to authenticate.
`HTTPBasic.xml`	Defines one screen with a header only as credentials are requested via the user's web browser.
`JDBC.xml`	Defines a Login screen for use with Java Database Connectivity (JDBC) authentication.
`LDAP.xml`	Defines a Login screen, a Change Password screen and two error message screens (Reset Password and User Inactive).
`Membership.xml`	Default data interface which can be used to customize for any domain.
`MSISDN.xml`	Defines a Login screen for use with Mobile Subscriber ISDN (MSISDN).
`NT.xml`	Defines a Login screen.
`RADIUS.xml`	Defines a Login screen and a RADIUS Password Challenge screen.
`SafeWord.xml`	Defines two Login screens: one for User Name and the next for Password.
`SAE.xml`	Defines a Login screen for Virtual Federation Proxy (Secure Attributes Exchange)

TABLE 12–3 Authentication Module Configuration XML Files *(Continued)*

File Name	Description
`SAML.xml`	Defines a Login screen for SAML authentication.
`SecurID.xml`	Defines five Login screens including `UserID` and `Passcode`, PIN mode, and Token Passcode.
`Unix.xml`	Defines a Login screen and an Expired Password screen.
`WindowsDesktopSSO.xml`	Defines a Login screen for Windows Desktop SSO Authentication

Callbacks Elements

Nested Elements

The following table describes nested elements for the `Callbacks` element.

The Callbacks element is used to define the information a module needs to gather from the client requesting authentication. Each Callbacks element signifies a separate screen that can be called during the authentication process.

TABLE 12–4 Nested Elements

Element	Required	Description
`NameCallback`	*	Requests data from the user; for example, a user identification.
`PasswordCallback`	*	Requests password data to be entered by the user.
`ChoiceCallback`	*	Used when the application user must choose from multiple values.
`ConfirmationCallback`	*	Sends button information such as text which needs to be rendered on the module's screen to the authentication interface.
`HttpCallback`	*	Used by the authentication module with HTTP-based handshaking negotiation.
`SAMLCallback`		Used for passing either Web artifact or SAML POST response from SAML service to the SAML authentication module when this module requests for the respective credentials. This authentication module behaves as SAML recipient for both (Web artifact or SAML POST response) and retrieves and validates SAML assertions.

Attributes

The following table describes attributes for the `Callbacks` element.

`length`	Number or length of callbacks.
`order`	Sequence of the group of callbacks.
`timeout`	Number of seconds the user has to enter credentials before the page times out. Default is 60.
`template`	Defines the UI `.jsp` template name to be displayed.
`image`	Defines the UI or page-level image attributes for the UI customization
`header`	Text header information to be displayed on the UI. Default is Authentication.
`error`	Indicates whether authentication framework/module needs to terminate the authentication process. If yes, then the value is true. Default is `false` .

ConfirmationCallback Element

The `ConfirmtationCallback` element is used by the authentication module to send button information for multiple buttons. An example is the button text that must be rendered on the UI page. The `ConfirmationCallback` element also receives the selected button information from the UI.

Nested Element

`ConfirmationCallback` has one nested element named `OptionValues`. The `OptionValues` element provides a list or an array of button text information to be rendered on the UI page.`OptionValues` takes no attributes.

If there is only one button on the UI page, then the module is not required to send this callback. If `ConfirmationCallback` is not provided through the Authentication Module properties XML file, then `anAuthUI.properties` will be used to pick and display the button text or label for the Login button. `anAuthUI.properties` is the global UI properties file for all modules.

Callbacks length value should be adjusted accordingly after addition of the new callback.

Example:

```
<ConfirmationCallback>
    <OptionValues>
        <OptionValue>
            <Value> <required button text> </Value>
        </OptionValue>
    </OptionValues>
</ConfirmationCallback>
```

JavaScript Files

JavaScript files are parsed within the `Login.jsp` file. You can add custom functions to the JavaScript files in the following directory:

OpenSSO-Deploy-base`/js`

The Authentication Service uses the following JavaScript files:

TABLE 12–5 JavaScript Files Used by the Authentication Service

File	Description
`auth.js`	Used by `Login.jsp` for parsing all module files to display login requirement screens.
`browserVersion.js`	Used by `Login.jsp` to detect the client type.
`admincli.js`	Used by the admin CLI.
`opensso.js`	Used to get the context path.

Cascading Style Sheets

To define the look and feel of the UI, modify the cascading style sheets (CSS) files. Characteristics such as fonts and font weights, background colors, and link colors are specified in the CSS files. You must choose the appropriate `.css` file for your browser in order to customize the look and feel on the user interface.

In the appropriate `.css` file, change the background-color attribute. For example:

```
.button-content-enabled { background-color:red; }
button-link:link, a.button-link:visited { color: #000;
background-color: red;
text-decoration: none; }
```

Browser-specific CSS files are installed with OpenSSO Enterprise in the following directory:

OpenSSO-Deploy-base`/css`

The following table describes each CSS file.

TABLE 12–6 OpenSSO Enterprise Cascading Style Sheet (CSS) Files

File Name	Purpose
`css_ie6win.css`	Configured specifically for Microsoft Internet Explorer 6 for Windows.
`css_ie5win.css`	Configured specifically for Microsoft Internet Explorer 5 for Windows.
`css_ns6up.css`	Configured specifically for Netscape Communicator 6.
`css_ns4sol.css`	Configured specifically for Netscape Communicator 4 for Solaris systems.
`css_ns4win.css`	Configured specifically for Netscape Communicator 4 for Windows.
`styles.css`	Used in JSP pages as a default style sheet.

Images

The default authentication GUI is branded with Sun Microsystems, Inc. logos and images. By default, the GIF files are installed in the following directory:

OpenSSO-Deploy-base`/login_images`

These images can be replaced with images relevant to your company or organization. The following table describes each GIF image used for the default GUI.

TABLE 12–7 Sun Microsystems Branded GIF Images

File Name	Purpose
`adminstyle.css`, `master-style.css`, and `CCCSS_Default.css`	Style sheets
`Identity_LogIn.gif`	Sun Java System Access Manager banner
`error_32_sunplex.gif`	Error indicator
`info_32_sunplex.gif`	Information indicator
`spacer.gif`	Spacer graphic
`logo_sun.gif`	Sun Microsystems logo graphic
`Java.gif`	Java graphic
`spacer.gif`	A one pixel clear image used for layout purposes

Localization Files

After you deploy the `opensso.war` file the localized files are located in the following directory:

OpenSSO-Deploy-base`/WEB-INF/classes`

OpenSSO-Deploy-base represents the deployment directory where the web container deployed the `opensso.war` file.

In addition to US English (en_US), OpenSSO Enterprise includes localized properties files for these languages:

- German (de)
- Spanish (es)
- French (fr)
- Japanese (ja)
- Korean (ko)
- Simplified Chinese (zh)
- Traditional Chinese (zh_TW)

A localization properties file, sometimes also referred to as an i18n (internationalization) properties file, specifies the screen text and error messages that an administrator or user sees when directed to the attribute configuration page for an authentication module. The properties files are global to the OpenSSO Enterprise instance.

Each authentication module has its own properties file that follows the naming following format:

`amAuth`*modulename*`.properties`

For example, `amAuthLDAP.properties` is for the default language (US English, ISO-8859-1), `amAuthLDAP_ja.properties` is for Japanese, and so on.

You can adapt Java applications to these various languages without code changes by translating the values in these respective localization properties file.

The following table summarizes the localization properties files for each authentication module.

TABLE 12–8 Localization Properties Files for Authentication Modules

File Name	Description
`amAuth.properties`	Core Authentication Service
`amAuthAD.properties`	Microsoft Active Directory Authentication Module
`amAuthAnonymous.properties`	Anonymous Authentication Module

TABLE 12–8 Localization Properties Files for Authentication Modules *(Continued)*

File Name	Description
`amAuthApplication.properties`	For OpenSSO Enterprise internal use only. Do not remove or modify this file.
`amAuthCert.properties`	Certificate Authentication Module
`amAuthConfig.properties`	Authentication Configuration Module
`amAuthContext.properties`	Localized error messages for the `AuthContext` Java class
`amAuthContextLocal.properties`	For OpenSSO Enterprise internal use only. Do not remove or modify this file.
`amDataStore.properties`	Data Store Authentication Module
`amAuthHTTPBasic.properties`	HTTP Basic Authentication Module
`amAuthJDBC.properties`	Java Database Connectivity (JDBC) Authentication Module
`amAuthLDAP.properties`	LDAP Authentication Module
`amAuthMembership.properties`	Membership Authentication Module
`amAuthMSISDN.properties`	Mobile Subscriber ISDN Authentication Module
`amAuthNT.properties`	Windows NT Authentication Module
`amAuthRadius.properties`	RADIUS Authentication Module
`amAuthSafeWord.properties`	Safeword Authentication Module
`amAuthSAML.properties`	SAML Authentication Module
`amAuthSecurID.properties`	SecurID Authentication Module
`amAuthUI.properties`	Labels used in the authentication user interface
`amAuthUnix.properties`	UNIX Authentication Module
`amAuthWindowsDesktopSSO.properties`	Windows Desktop SSO Authentication Module

Customizing Branding and Functionality

You can modify JSP templates and module configuration properties files to reflect branding or functionality specified for any of the following:

- Organization of the request
- SubOrganization of the request.
- Locale of the request
- Client Path
- Client Type information of the request
- Service Name (serviceName)

▼ To Modify Branding and Functionality

1 **Go to the directory where default JSP templates are stored.**

`cd` *OpenSSO-Deploy-base*`/config/auth`

2 **Create a new directory.**

Use the appropriate customized directory path based on the level of customization. Use the following forms:

```
org_locale/orgPath/filePath
     org/orgPath/filePath
     default_locale/orgPath/filePath
     default/orgPath/filePath
```

In these examples,

`orgPath` represents `subOrg1/subOrg2`

`filePath` represents `clientPath` + `serviceName`

`clientPath` represents `clientType/sub-clientType`

In these paths, SubOrg, Locale, Client Path, Service Name (which represents `orgPath` and `filePath`) are optional. The organization name you specify may match the organization attribute set in the Directory Server. For example, if the organization attribute value is `SunMicrosystems`, then the organization customized directory should also be `SunMicrosystems`. If no organization attribute exists, then use the lowercase value of the organization name (`sunmicrosystems`).

For example, for the following attributes:

`org = SunMicrosystems`

`locale = en`

`subOrg = solaris`

`clientPath = html/` *customerName/*

`serviceName = paycheck`

The customized directory paths would then be:

`SunMicrosystems_en/solaris/html/` *customerName* `/paycheck`

`SunMicrosystems/solaris/html/` *customerName* `/paycheck`

`default_en/solaris/html/` *customerName*`/paycheck`

`default/solaris/html/` *customerName* `/paycheck`

3 **Copy the default templates.**

Copy all the JSP templates (`*.jsp`) and authentication module configuration properties `XML` files (`*.xml`) from the default directory:

OpenSSO-Deploy-base`/config/auth/default`

to the new directory:

OpenSSO-Deploy-base`/config/auth/`*CustomizedDirectoryPath*

4 **Customize the files in the new directory.**

The files in the new directory can be customized if necessary, but not this is not required. See "Customizing the Login Page" on page 190 and "Customizing JSP Templates" on page 191 for information on what you can modify.

5 **Update and redeploy the `opensso.war` file.**

After you have modified the authentication GUI files, in order to see the changes in the actual GUI, you must update and then redeploy the `opensso.war` file. For more information, see Chapter 11, "Creating and Deploying OpenSSO Enterprise WAR Files."

6 **Restart the OpenSSO Enterprise server web container.**

Customizing the Self-Registration Page

You can customize the Self-registration page which is part of Membership authentication module. The default data and interface provided with the Membership authentication module is generic and can work with any domain. You can configure it to reflect custom data and information. You can add custom user profile data or fields to register or to create a new user.

▼ To Modify the Self-Registration Page

1 **Customize the `Membership.xml` file.**

By default, the first three data fields are required in the default Membership Module configuration:

- User name
- User Password
- Confirm User Password

  You can specify which data is requested, which is required, and which is optional. The sample below illustrates how to add a telephone number as requested data.

  You can specify or add data which should be requested from a user as part of the User Profile. By default you can specify or add any attributes from the following `objectClasses`:

- `top`
- `person`
- `organizationalPerson`
- `inetOrgPerson`
- `iplanet-am-user-service`
- `inetuser`

  Administrators can add their own user attributes to the User Profile.

2 **Update and redeploy the `opensso.war` file.**

After you have modified the authentication GUI files, in order to see the changes in the actual GUI, you must update and then redeploy the `opensso.war` file. For more information, see Chapter 11, "Creating and Deploying OpenSSO Enterprise WAR Files."

3 **Restart the OpenSSO Enterprise server web container.**

```
<Callbacks length="9" order="16" timeout="300"
header="Self Registration" template="register.jsp" >

    <NameCallback isRequired="true" attribute="uid" >
    <Prompt> User Name: </Prompt>
    </NameCallback>

    <PasswordCallback echoPassword="false" isRequired="true"
       attribute="userPassword" >
    <Prompt> Password: </Prompt>
    </PasswordCallback>

    <PasswordCallback echoPassword="false" isRequired="true" >
    <Prompt> Confirm Password: </Prompt>
    </PasswordCallback>

    <NameCallback isRequired="true" attribute="givenname" >
    <Prompt> First Name: </Prompt>
    </NameCallback>

    <NameCallback isRequired="true" attribute="sn" >
    <Prompt> Last Name: </Prompt>
    </NameCallback>

    <NameCallback isRequired="true" attribute="cn" >
    <Prompt> Full Name: </Prompt>
    </NameCallback>

    <NameCallback attribute="mail" >
    <Prompt> Email Address: </Prompt>
    </NameCallback>


<NameCallback isRequired="true"attribute="telphonenumber">
<Prompt> Tel:</Prompt>
</NameCallback>

    <ConfirmationCallback>

        <OptionValues>
        <OptionValue>
        <Value> Register </Value>
        </OptionValue>
        <OptionValue>
        <Value> Cancel </Value>
        </OptionValue>
        </OptionValues>

    </ConfirmationCallback>

</Callbacks>
```

Customizing the Distributed Authentication User Server Interface

A Sun OpenSSO Enterprise Distributed Authentication UI server provides for secure, distributed authentication across two firewalls in an OpenSSO Enterprise deployment. You install the Distributed Authentication UI server subcomponent on a web container on one or more servers within the DMZ layer of the OpenSSO Enterprise deployment. This subcomponent acts as an authentication interface between end users and the OpenSSO Enterprise instances behind the second firewall, thus eliminating the exposure of the OpenSSO Enterprise service URLs to the end users.

The remote Distributed Authentication UI server subcomponent uses authentication client APIs and utility classes to authenticate users. The subcomponent uses a customizable JATO presentation framework.

You can modify the JSP templates and module configuration properties files to reflect branding and specific functionality for the following:

Organization/SubOrganization	Organization or sub-organization of the request.
Locale	Locale of the request.
Client Path	Client type information of the request.
Service Name (`serviceName`)	Service name for service-based authentication.

For background information about a Distributed Authentication UI server, see the *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*.

▼ To Customize the Distributed Authentication Server User Interface

In this procedure, you will create a Distributed Authentication Server UI WAR file from `opensso.war` and then customize the new WAR file.

1. **Make sure that your `JAVA_HOME` environment variable points to a JDK of version 1.5 or later.**

2. **If necessary, download and unzip the `opensso_enterprise_80.zip` file.**

   The `opensso.war` file is then in the *zip-root*/`opensso/deployable-war` directory, where *zip-root* is where you unzipped the `opensso_enterprise_80.zip` file.

3. **Create a new staging directory to extract the files from `opensso.war`. For example:**

   ```
   # mkdir opensso-staging
   ```

4. **In the staging directory, extract the files from `opensso.war`. For example:**

   ```
   # cd opensso-staging
   # jar xvf zip-root/opensso/deployable-war/opensso.war
   ```

5. **Create the Distributed Authentication UI server WAR using the files in `fam-distauth.list`:**

   ```
   # cd opensso-staging
   # jar cvf zip-root/opensso/deployable-war/distauth.war \
      @zip-root/opensso/deployable-war/fam-distauth.list
   ```

   where `distauth.war` is the name of the new Distributed Authentication UI server WAR file.

   Note: Some web containers require the Distributed Authentication WAR file name to use the same name as the deployment URI.

6. **Update the WAR file created in previous step with the additional files required for the Distributed Authentication UI server. For example:**

   ```
   # cd zip-root/opensso/deployable-war/distauth
   # jar uvf zip-root/opensso/deployable-war/distauth.war *
   ```

   You are now ready to customize the new `distauth.war`.

7. **Create a new directory to explode your new `distauth.war`. For example:**

   ```
   # mkdir distauth-staging
   ```

8 **Explode the new Distributed Authentication User Interface WAR in the staging directory you created in the previous step. For example:**

```
# cd distauth-staging
# jar xvf zip-root/opensso/deployable-war/distauth.war
```

9 **Create a new directory for your customized files. For example:**

```
# cd distauth-staging/config/auth
# mkdir custdaui
```

Use the following form:

```
org_locale/orgPath/filePath
        org/orgPath/filePath
        default_locale/orgPath/filePath
        default/orgPath/filePath
```

where:

```
orgPath = subOrg1/subOrg2
        filePath = clientPath + serviceName
        clientPath = clientType/sub-clientType
```

The following items are optional: Sub-org, Locale , Client Path , and Service Name . In the following example, `orgPath` and `filePath` are optional.

For example, given the following:

```
org = iplanet
locale = en
subOrg = solaris
clientPath = html/company/
serviceName = paycheck
```

The appropriate directory paths for the above are:

```
iplanet_en/solaris/html/company/paycheck
iplanet/solaris/html/company/paycheck
default_en/solaris/html/company/paycheck
default/solaris/html/company/paycheck
```

10 **Change to the directory where the JSP and XML files are stored, and copy the JSP and authentication module configuration (XML) files from the default directory to the new directory.**

```
#cd distauth-staging/config/auth/default
cp *.jsp distauth-staging/config/auth/custdaui
cp *.xml distauth-staging/config/auth/custdaui
```

11 Customize the following files in the `custdaui` directory, as required for your deployment:

- JSP files: "Java Server Page (JSP) Files" on page 190
- XML configuration files: "XML Files" on page 193

12 Update the WAR file with the customized files:

```
# cd distauth-staging/config/auth/custdaui
# jar uvf zip-root/opensso/deployable-war/distauth.war *
```

You are now ready to deploy the customized `distauth.war` file.

Next Steps To deploy and configure the customized Distributed Authentication User Interface server WAR file, see Chapter 8, "Deploying a Distributed Authentication UI Server," in *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*.

CHAPTER 13

Using the Client SDK

The Sun™ OpenSSO Enterprise Client Software Development Kit (Client SDK) provides the Java libraries for integrating OpenSSO Enterprise functionality in remote standalone or web applications. This chapter contains the following sections:

About the Client SDK

The Client SDK includes the Java packages, classes, and configuration properties that you can use to write remote standalone or web applications that access an OpenSSO Enterprise server. The Client SDK allows an application to use services such as authentication, SSO, authorization, auditing, logging, and the Security Assertion Markup Language (SAML). The Client SDK also includes sample applications that you can deploy to help you understand and write your own custom applications.

The Client SDK is not for use by applications that perform policy management or identity management (which includes the creation and deletion of entries).

From a deployment point of view, the Client SDK offers the following:

- The Client SDK communicates directly with OpenSSO Enterprise server using XML (SOAP) over HTTP or HTTPS. In turn, OpenSSO Enterprise server communicates directly with the data stores.
- The Client SDK does not require administrator credentials.

- An application using the Client SDK can be deployed in a demilitarized zone (DMZ), with a firewall between the application and OpenSSO Enterprise server.
- The Client SDK includes samples to show how it can be used.
- The Client SDK includes these packages:
  - `com.iplanet.am.sdk`
  - `com.iplanet.am.util`
  - `com.iplanet.sso`
  - `com.sun.identity.authentication`
  - `com.sun.identity.federation`
  - `com.sun.identity.idm`
  - `com.sun.identity.liberty.ws`
  - `com.sun.identity.log`
  - `com.sun.identity.policy`
  - `com.sun.identity.policy.client`
  - `com.sun.identity.saml`
  - `com.sun.identity.saml2`
  - `com.sun.identity.smt`
  - `com.sun.identity.xacml`
  - `com.sun.identity.wss`

For a description of these packages, see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

Caution – It is recommended that you do not use the `com.iplanet.am.sdk`, `com.iplanet.am.util`, `com.sun.identity.policy`, and `com.sun.identity.sm` packages directly.

OpenSSO Enterprise Client SDK Requirements

The requirements to use the Client SDK include:

- You will need access to OpenSSO Enterprise running on a remote server. You will need the following information about this remote installation:
  - Protocol (`http` or `https`) used by web container instance on which the OpenSSO Enterprise server is deployed.
  - Fully qualified domain name (FQDN) of the host where the OpenSSO Enterprise server is deployed.
  - Port on which the OpenSSO Enterprise server is running.
  - Deployment URI for the OpenSSO Enterprise server (default is opensso)
  - Default Agent user (`UrlAccessAgent`) password that you entered when you ran the OpenSSO Enterprise Configurator.

- If you are writing a web application, you will need a web container supported by OpenSSO Enterprise. For the list of supported web containers, see the Chapter 2, “Deploying the OpenSSO Enterprise Web Container,” in *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*.

Using the Client SDK

You can use the Client SDK to:

- Build a proprietary application framework in which the Client SDK is a part. The Client SDK features can allow independence from policy agents.
- Access profile data, for purposes of authentication and authorization, beyond the default OpenSSO Enterprise capability.
- Allow authenticated and non-authenticated users access to a login process with a registration option that, if accepted, would create a user account.

Installing the Client SDK

▼ To Install the Client SDK

Before You Begin

- If you have not already done so, download and unzip the `opensso_enterprise_80.zip` file, as described in the *Sun OpenSSO Enterprise 8.0 Installation and Configuration Guide*.

  The Client SDK and samples are then available in the *zip-root*`/opensso/samples/opensso-client.zip` file, where *zip-root* is the directory where you unzipped `opensso.war`.
- If you plan to install the Client SDK in a web container, the web container must be installed on the server where you plan to deploy the Client SDK.

1 **On the server where you plan to deploy the Client SDK, copy the `opensso-client.zip` to a staging directory.**

2 **In the directory from Step 1, unzip the `opensso-client.zip` file.**

The following table describes the layout after you unzip the `opensso-client.zip` file. The directory where you unzip the file is represented by *opensso-client-zip-root*.

opensso-client-zip-root Directory	Description
`/sdk`	Client SDK command—line samples, which you can run in a standalone JVM outside of a web container: ■ `/source` contains the source files that require compilation. ■ `/scripts` contains the scripts to compile and run the samples. ■ `/resources` contains the various properties files required to run the samples, including the `AMConfig.properties.template` file. ■ `/lib` contains the JAR files required by the Client SDK. ■ `/classes` contains the compiled classes from the source files.
`/war`	Client SDK WAR files, which include the web-based client samples: ■ `opensso-client-jdk15.war` is for web containers running JDK 1.5 or later ■ `opensso-client-jdk14.war` is for web containers running JDK 1.4.x Deploy these files using the web container administration console or command-line utility.

Running the Client SDK Samples

OpenSSO Enterprise comes with samples and source code that can help developers understand how the Client SDK classes can be implemented. The samples, acting as standalone applications, can be run on the command-line and in a web browser to see the function being performed. By looking at the provided sample source code you can understand how the Client SDK classes were used to perform the sample function.

The `opensso-client.zip` file is located in the `samples` directory of the inflated OpenSSO Enterprise ZIP file. After unzipping `opensso-client.zip` `opensso-client` directory, you will find two subdirectories:

- `sdk` contains the command-line samples, including the setup and compilation scripts.
- `war` contains deployable WAR files that contain the Client SDK and web-based samples.

To run the Client SDK samples, continue as follows:

- "Running the Client SDK Web-Based Samples" on page 210
- "Running the Client SDK Command-Line Samples" on page 213

Running the Client SDK Web-Based Samples

The following WAR files contain the Client SDK and the web-based samples.

- `opensso-client-jdk15.war` for JDK 1.5 or later
- `opensso-client-jdk14.war` for JDK 1.4.2 or later

You run the Client SDK web-based samples by deploying one of these WAR files, depending on the JDK version you are using.

After you unzip the `opensso-client.zip` file, the Client SDK WAR files are available in the *opensso-client-zip-root*/`war` directory.

The following table describes the web-based sample applications and their corresponding source files on the OpenSSO site: `https://opensso.dev.java.net`.

TABLE 13–1 Client SDK Web-Based Samples

Sample	Function	Source Files
Service Configuration Sample Servlet	Retrieves and displays attributes of the entered service name	ServiceConfigServlet.java
User Profile (Attribute) Sample Servlet	Retrieves and displays the attributes that correspond to the entered user ID	UserProfileServlet.java
Policy Evaluator Client Sample Servlet	Retrieves from the Policy Service a policy decision that would be passed to a web agent for enforcement	PolicyClientServlet.java
Single Sign-on Token Verification Servlet	Validates a session token and then displays the user profile associated with it	SSOTokenSampleServlet.java
Liberty ID-WSF 1.x Web Service Consumer Sample	Query and modify the Discovery Service and the Liberty Personal Profile Service	wsc Directory
Security Token Service (WS-Trust) Client Sample	Obtain security tokens from the Security Token Service	sts Directory

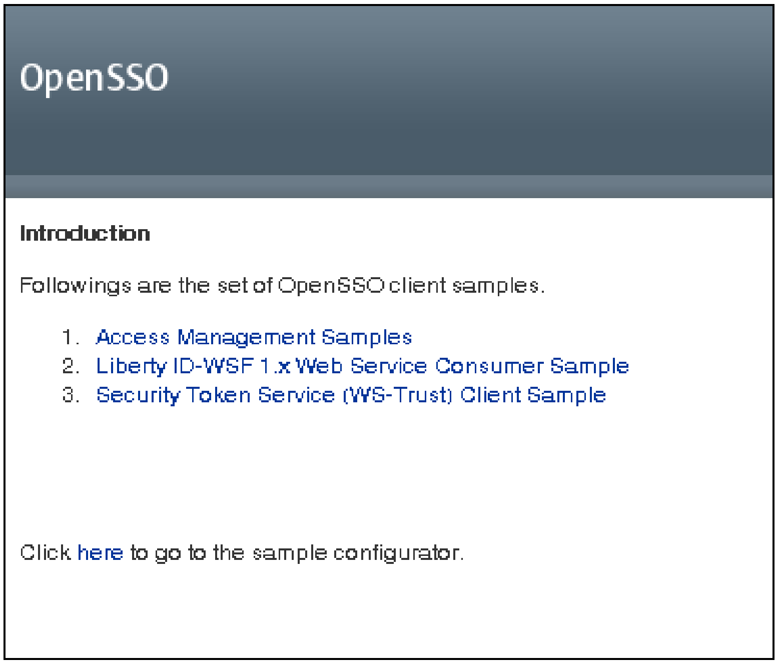
▼ To Run the Client SDK Web-Based Samples

1 **Set your `JAVA_HOME` environment variable to JDK 1.5 or 1.4, depending on the JDK version you are using.**

2 **Deploy either `opensso-client-jdk14.war` or `opensso-client-jdk15.war` to your web container, depending on your JDK version.**

3 **Launch the WAR file you deployed in the previous step, to display the Introduction page for the OpenSSO client samples. For example:**

```
http://host.example.com:8080/opensso-client-jdk15/
```



OpenSSO

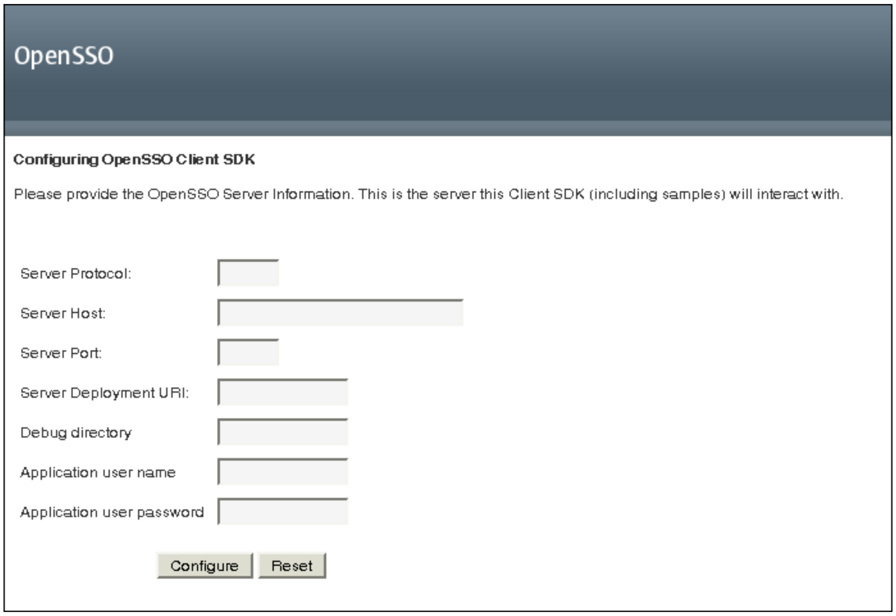
Introduction

Followings are the set of OpenSSO client samples.

1. Access Management Samples
2. Liberty ID-WSF 1.x Web Service Consumer Sample
3. Security Token Service (WS-Trust) Client Sample

Click here to go to the sample configurator.

4 Click the `here` link to view the sample configuration page.



OpenSSO

Configuring OpenSSO Client SDK

Please provide the OpenSSO Server Information. This is the server this Client SDK (including samples) will interact with.

Server Protocol:	
Server Host:	
Server Port:	
Server Deployment URI:	
Debug directory	
Application user name	
Application user password	

5 **Specify the following values for the OpenSSO Enterprise server:**

- **Server Protocol** (`http` or `https`) used by web container instance on which the OpenSSO Enterprise server is deployed.
- **Server Host** fully qualified domain name (FQDN) where the OpenSSO Enterprise server is deployed.
- **Server Port** for OpenSSO Enterprise server.
- **Server Deployment URI** for OpenSSO Enterprise server. The default is `/opensso`.
- **Debug directory**: For example: `/opensso/debug`
- **Application user name:**: `UrlAccessAgent`.
- **Application user password**: `UrlAccessAgent` password.

6 **Click Configure to configure the Client SDK and to create a new `AMConfig.properties` file.**

For information about the location of the `AMConfig.properties` file and setting properties in the file, see "Using the `AMConfig.properties` File With the Client SDK" on page 215.

7 **Click the link to go to the Client SDK samples page.**

The following web-based samples are available for you to run:

- **Client SDK Samples**
- **Liberty ID-WSF 1.x Web Service Consumer Sample** shows how to deploy and run the WSC sample to query and modify the Liberty Discovery Service and ID-SIS Personal Profile Service.
- **Security Token Service (WS-Trust) Client Sample** shows how to use the Security Token Service (STS) Client API to obtain security tokens from a Security Token Service hosted on OpenSSO Enterprise server.

To run the **Liberty ID-WSF 1.x Web Service Consumer Sample** or **Security Token Service (WS-Trust) Client Sample**, click the respective link and follow the instructions.

Running the Client SDK Command-Line Samples

After you unzip the `opensso-client.zip` file, the command-line samples are in the *opensso-client-zip-root*/`sdk` directory. You must setup and compile these samples before you can use them.

The source files in the following table are on the OpenSSO site: `https://opensso.dev.java.net`.

TABLE 13–2 Client SDK Command-Line Samples

Sample	Function	Source Files
`setup.sh`	Creates `AMConfig.properties` and populates it with values based on your deployment.	Main.java
`Login.sh`	Logs in and then logs out the user.	Login.java
`CommandLineSSO.sh`	Demonstrates how to retrieve a user profile.	CommandLineSSO.java
`CommandLineIdrepo.sh`	Perform operations on the Identity Repository. For example, create an identity, delete an identity, and search or select an identity	idrepo Directory
`CommandLineLogging.sh`	Demonstrates the writing to a log a record of a successful authentication. See also Chapter 14, "Reading and Writing Log Records."	logging Directory
`SSOTokenSample.sh`	Verifies a session token from a `SSOTokenID` input.	SSOTokenSample.java
`run-policy-evaluation-sample.sh`	Returns a policy decision based on console created user and configured policy.	policy Directory
`run-xacml-client-sample.sh`	Constructs a XACML request, makes an authorization query, receives the decision, and prints out the response.	XACMLClientSample.java

▼ To Run the Client SDK Command-Line Samples

Before You Begin If you have not already do so, unzip the `opensso-client.zip` file, as described in "Installing the Client SDK" on page 209.

The Client SDK command-line samples are then available in the *opensso-client-zip-root*`/sdk/source` directory, where *opensso-client-zip-root* is the directory where you unzipped `opensso-client-zip-root`.

1 **Set your `JAVA_HOME` environment variable to JDK 1.5 or 1.4, depending on the version of the JDK you are using.**

2 **On Solaris and Linux systems, make all shell scripts in the *opensso-client-zip-root*`/sdk/scripts` directory executable. For example:**

```
# cd opensso-client-zip-root/sdk/scripts
# chmod 755 *.sh
```

3 **Compile the samples by executing the `sdk/scripts/compile-samples.sh` script.**

Note: You can invoke the sample scripts only from the `/sdk` parent directory and not directly from the `/scripts` directory. For example:

```
# cd opensso-client-zip-root/sdk/scripts
# scripts/compile-samples.sh
```

4 **Run the appropriate setup script for the samples:**

- Solaris and Linux systems: `scripts/setup.sh`

  or
- Windows: `scripts/setup.bat`

Run the setup script only once for of the all Client SDK samples. The script will create the `AMConfig.properties` file to point to the OpenSSO Enterprise server in the following directory:

opensso-client-zip-root`/sdk/resources`

5 **Run individual Client SDK samples by executing the shell or bat scripts in the `/scripts` directory. For example:**

```
# scripts/run-xacml-client-sample.sh
```

Note: At run time, a sample might require additional property files to be setup in the `/resources` directory. Check the comments included in each individual script for more information.

Using the `AMConfig.properties` File With the Client SDK

Although the `AMConfig.properties` file has been deprecated as the configuration data store for OpenSSO Enterprise server, this file is still used to store configuration data for the Client SDK.

The `AMConfig.properties` file points to the OpenSSO Enterprise server instance that will be used by the Client SDK.

The location of the `AMConfig.properties` file depends on the samples you are running:

- Web-based samples:

  user.home/*OpenSSO-Deploy-base*`-client-jdk15_AMConfig.properties`

  where `user.home` (JDK system property) is the home directory of the user running the web container, and *OpenSSO-Deploy-base* is determined by the web container.

  For example, if you deployed `opensso-client-jdk15.war` on Sun Java System Application Server 9.1 while running as super user (`root`), the `AMConfig.properties` file is:

  `OpenSSOClient/_opt_SUNWappserver_domains_domain1_applications_j2ee-modules_opensso`
- Command-line sample: *opensso-client-zip-root*`/sdk/resources`

Note – Both openssoclientsdk.jar and servlet.jar are required in the CLASSPATH of the server where the Client SDK is installed.

An AMConfig.properties file with the information needed to point to the remote OpenSSO Enterprise server must be accessible to the Client SDK from the machine on which the client application is hosted. The AMConfig.properties created by the sample WAR can be modified for this purpose, as follows.

- "Setting Properties in the AMConfig.properties File" on page 216
- "OpenSSO Enterprise Properties in the AMConfig.properties File" on page 217

Setting Properties in the AMConfig.properties File

You can set properties in the AMConfig.properties file by editing the file using a text editor, as follows:

property-name=property-value

Note – The AMConfig.properties file must be in the CLASSPATH. If necessary, declare the JVM option as follows:

-Damconfig=*properties-file-name*

You can also set properties as follows:

- "Setting Properties Using the Java API" on page 216
- "Setting Properties at Run Time" on page 217

Setting Properties Using the Java API

You can set Client SDK properties programmatically using the com.iplanet.am.util.SystemProperties class. For example:

EXAMPLE 13–1 Setting Client SDK Properties Programmatically

```
import com.iplanet.am.util.SystemProperties;
import java.util.Properties;
public static void main(String[] args) {
        // To initialize a set of properties
        Properties props = new Properties();
        props.setProperty(”com.iplanet.am.naming.url’,
        ”http://sample.com/opensso/namingservice’);
```

EXAMPLE 13–1 Setting Client SDK Properties Programmatically *(Continued)*

```
        props.setProperty(”com.sun.identity.agents.app.username’, ”amAdmin’);
        props.setProperty(”com.iplanet.am.service.password’, ”11111111’);
        SystemProperties.initializeProperties(props) ;

        // To initialize a single property
        SystemProperties.initializeProperties(“com.iplanet.am.naming.url’,
        ”http://sample.com/opensso/namingservice’);
    // Application specific code ...
}
```

Setting Properties at Run Time

To set a value to a particular property at run time, declare the following JVM option:

`-D`*property-name*=*property-value*

OpenSSO Enterprise Properties in the `AMConfig.properties` File

The Client SDK uses the following OpenSSO Enterprise properties in the `AMConfig.properties` file. This file is generated by the Client SDK installation. You can also add additional properties to `AMConfig.properties`, as required by a client application, For example, an application can register for the notification of changes to session attributes, user attributes, and policy decisions.

- “Naming Properties” on page 218
- “Debug Properties” on page 218
- “Notification URL Property” on page 218
- “Security Credentials Properties” on page 219
- “Encryption Properties” on page 219
- “Cache Update Properties” on page 219
- “Client Services Properties” on page 220
- “Cookie Property” on page 220
- “Session Service Properties” on page 220
- “Certificate Database Properties” on page 220
- “Policy Client Properties” on page 221
- “Remote Client SDK Property” on page 222
- “Federation Properties” on page 222

Naming Properties

`com.iplanet.am.naming.url`

This is a required property. The value of this property is the URI of the Naming Service from which the Client SDK would retrieve the URLs of OpenSSO Enterprise internal services:

```
com.iplanet.am.naming.url=http://opensso-host.domain_name:port/opensso/namingservice
```

`com.iplanet.am.naming.failover.url`

This property can be used by any remote application developed with the Client SDK that wants failover in, for example, session validation or getting the service URLs:

```
com.iplanet.am.naming.failover.url=http://opensso-host.domain_name:port/opensso/failover
```

Note – After you run the setup script, `com.iplanet.am.naming.failover.url` is not in `AMConfig.properties`, so if you want to use this property, you must add it the file.

Debug Properties

`com.iplanet.services.debug.level`

Specifies the debug level:

- **`off`** specifies that no debug information is recorded.
- **`error`** specifies that only debug messages posted as errors should be written to the debug files. This level is recommended for production environments.
- **`warning`** is not a recommended value at this time.
- **`message`** alerts to possible issues using code tracing. Most OpenSSO Enterprise modules use this level to send debug messages.

Caution – Using **`warning`** or **`message`** in production environments is not recommended because they can cause severe performance degradation from excessive of debug messages.

`com.iplanet.services.debug.directory`

The value of this property is the output directory for the debug information. The directory should be writable by the server process. For example:

```
com.iplanet.services.debug.directory=/opensso/debug
```

Notification URL Property

`com.iplanet.am.notification.url`

The value of this property is the URI of the Notification Service running on the host machine where the Client SDK is installed. For example:

```
com.iplanet.am.notification.url=http://sdk-client-host.domain:port
/opensso-URI/notificationservice
```

Security Credentials Properties

`com.sun.identity.agents.app.username`
: User with permission to read the OpenSSO Enterprise configuration data. Default:

```
com.sun.identity.agents.app.username=UrlAccessAgent
```

`com.iplanet.am.service.password`
: Password of user with permission to read OpenSSO Enterprise configuration data.

`com.iplanet.am.service.secret`
: specifies the encrypted password for the user in the `com.sun.identity.agents.app.username` property. For example:

```
com.iplanet.am.service.secret=AQIC24u86rq9RRZGr/HN25OcIuO6w+ne+0lG
```

Encryption Properties

`am.encryption.pwd`
: Encryption key used to decrypt service configuration passwords:

```
am.encryption.pwd=ENCRYPTION_KEY
```

`com.sun.identity.client.encryptionKey`
: Encryption key used to encrypt and decrypt data used locally within the client application. For example:

```
com.sun.identity.client.encryptionKey=ENCRYPTION_KEY_LOCAL
```

`com.iplanet.security.encryptor`
: Property used to set the default encrypting class. Values can be:

- `com.iplanet.services.util.JCEEncryption`
- `com.iplanet.services.util.JSSEncryption`

Cache Update Properties

`com.sun.identity.sm.cacheTime`
: Cache update time (in minutes) for service configuration data if notification URL is not provided. For example:

```
com.sun.identity.sm.cacheTime=1
```

`com.iplanet.am.sdk.remote.pollingTime`
: Cache update time (in minutes) for user management data if notification URL is not provided. For example:

```
com.iplanet.am.sdk.remote.pollingTime=1
```

Client Services Properties

These properties are defined by the Client SDK configuration page.

`com.iplanet.am.server.protocol`
Protocol of machine on which OpenSSO Enterprise is deployed. For example:

```
com.iplanet.am.server.protocol=http
```

`com.iplanet.am.server.host`
Name and domain of machine on which OpenSSO Enterprise is deployed. For example:

```
com.iplanet.am.server.host=OSSO_Host_Machine.domain_name
```

`com.iplanet.am.server.port`
Port of machine on which OpenSSO Enterprise is deployed. For example:

```
com.iplanet.am.server.port=8080
```

`com.iplanet.am.services.deploymentDescriptor`
URI of the deployed instance of OpenSSO Enterprise. For example:

```
com.iplanet.am.server.protocol=opensso
```

Cookie Property

`com.iplanet.am.cookie.name`
The name of the OpenSSO Enterprise cookie. For example:

```
com.iplanet.am.cookie.name=iPlanetDirectoryPro
```

Session Service Properties

`com.iplanet.am.session.client.polling.enable`
A value of `true` or `false` enables or disables, respectively, client-side session polling.

`com.iplanet.am.session.client.polling.period`
Specifies the number of seconds in the polling period. Example

```
com.iplanet.am.session.client.polling.period=180
```

Certificate Database Properties

`com.iplanet.am.admin.cli.certdb.dir`
Identifies the directory path to the certificate database for initializing the JSS Socket Factory when the OpenSSO Enterprise web container is configured for SSL.

`com.iplanet.am.admin.cli.certdb.passfile`
Identifies the certificate database password file for initializing the JSS Socket Factory when the OpenSSO Enterprise web container is configured for SSL. For example:

```
com.iplanet.am.admin.cli.certdb.passfile=/config/.wtpass
```

`com.iplanet.am.admin.cli.certdb.prefix`
Identifies the certificate database prefix for initializing the JSS Socket Factory when the OpenSSO Enterprise web container is configured for SSL.

Policy Client Properties

`com.sun.identity.agents.server.log.file.name`
Specifies name of the client's policy log file. For example:

```
com.sun.identity.agents.server.log.file.name=amRemotePolicyLog
```

`com.sun.identity.agents.logging.level`
Specifies the granularity of logging to the client's policy log file.

- **`NONE`** is the default value. Nothing is logged.
- **`ALLOW`** logs allowed access decisions.
- **`DENY`** logs denied access decisions.
- **`BOTH`** logs allowed and denied access decisions.
- **`DECISION`**

`com.sun.identity.agents.notification.enabled`
A value of `true` or `false` enables or disables, respectively, notifications from OpenSSO Enterprise for updating the client cache.

`com.sun.identity.client.notification.url`
Specifies the URL to which policy, session, and agent configuration notifications from OpenSSO Enterprise are sent.

`com.sun.identity.agents.polling.interval`
Specifies the number of minutes after which an entry is dropped from the Client SDK cache. For example:

```
com.sun.identity.agents.polling.interval=3
```

`com.sun.identity.policy.client.cacheMode`
Specifies the cache mode for the client policy evaluator. Values are:

- **`subtree`** specifies that the policy evaluator obtains policy decisions from the server for all the resources from the root of resource actually requested.
- **`self`** specifies that the policy evaluator obtains policy decisions from the server only for the resource actually requested.

`com.sun.identity.policy.client.usePre22BooleanValues`
is set to `true` by default, specifying to use boolean values.

Note – After you run the setup script, `com.sun.identity.policy.client.usePre22BooleanValues` is not in `AMConfig.properties`. If you do not want to use boolean values, add this property to the file, with a value of `false`.

Remote Client SDK Property

`com.iplanet.am.sdk.package`	specifies the OpenSSO Enterprise SDK package name as: `com.iplanet.am.sdk.package=com.iplanet.am.sdk.remote`

Federation Properties

You must manually add these federation properties to `AMConfig.properties` as needed. They are not automatically placed in the file when generated.

com.sun.identity.wss.provider.plugins.AgentProvider

`com.sun.identity.liberty.ws.soap.supportedActor`
Supported SOAP actors. Each actor must be separated by a pipe (|). For example:

```
com.sun.identity.liberty.ws.soap.supportedActors=http://schemas.xmlsoap.org/soap/actor/next
```

`com.sun.identity.liberty.interaction.wspRedirectHandler`
Indicates the URL for `WSPRedirectHandlerServlet` to handle Liberty the WSF web service provider-resource owner. Interactions are based on user agent redirects. The servlet should be running in the same JVM where the Liberty service provider is running.

`com.sun.identity.liberty.interaction.wscSpecifiedInteractionChoice`
Indicates whether the web service client should participate in an interaction. Valid values are `interactIfNeeded`, `doNotInteract`, and `doNotInteractForData`. Default value is `interactIfNeeded` which is used if an invalid value is specified.

`com.sun.identity.liberty.interaction.wscWillInlcudeUserInteractionHeader`
Indicates whether the web service client should include `userInteractionHeader`. Valid values are `yes` and `no` (case ignored). Default value is `yes`. Default value is used if no value is specified.

`com.sun.identity.liberty.interaction.wscWillRedirect`
Indicates whether the web service client will redirect user for an interaction. Valid values are `yes` and `no`. Default value is `yes`. Default value is used if no value is specified.

`com.sun.identity.liberty.interaction.wscSpecifiedMaxInteractionTime`
Indicates the web service client preference for acceptable duration (in seconds) for an interaction. If the value is not specified or if a non-integer value is specified, the default value is `60`.

`com.sun.identity.liberty.interaction.wscWillEnforceHttpsCheck`
Indicates whether the web service client enforces that redirected to URL is HTTPS. Valid values are `yes` and `no` (case ignored). The Liberty specification requires the value to be `yes`. Default value is `yes`. Default value is used if no value is specified.

`com.sun.identity.liberty.interaction.wspWillRedirect`
Indicates whether the web service provider redirects the user for an interaction. Valid values are `yes` and `no` (case ignored). Default value is `yes`. Default value is if no value is specified.

`com.sun.identity.liberty.interaction.wspWillRedirectForData`
Indicates whether the web service provider redirects the user for an interaction for data. Valid values are `yes` and `no`. Default value is `yes`. If no value is specified, the value is `yes`.

`com.sun.identity.liberty.interaction.wspRedirectTime`
Web service provider expected duration (in seconds) for an interaction. Default value if the value is not specified or is a non-integer value is `30`.

`com.sun.identity.liberty.interaction.wspWillEnforceHttpsCheck`
Indicates whether the web service client enforces that `returnToURL` is HTTP. Valid values are `yes` and `no` (case ignored). Liberty specification requires the value to be `yes`. Default value is `yes`. If no value is specified, then the value used is `yes`.

`com.sun.identity.liberty.interaction.wspWillEnforceReturnToHostEqualsRequestHost`
Indicates whether the web services client enforces that `returnToHost` and `requestHost` are the same. Valid values are `yes` and `no`. Liberty specification requires the value to be `yes`.

`com.sun.identity.liberty.interaction.htmlStyleSheetLocation`
Indicates the path to the style sheet used to render the interaction page in HTML.

`com.sun.identity.liberty.interaction.wmlStyleSheetLocation`
Indicates the path to the style sheet used to render the interaction page in WML.

`com.sun.identity.liberty.ws.interaction.enable`
Default value is false.

`com.sun.identity.wss.provider.config.plugin=`
`com.sun.identity.wss.provider.plugins.AgentProvider`
Used by the web services provider to determine the plug-in that will be used to store the configuration.

For example: `com.sun.identity.wss.provider.config.plugin=`
`com.sun.identity.wss.provider.plugins.AgentProvider`

`com.sun.identity.loginurl`
Used by the web services clients in Client SDK mode. For example:

`com.sun.identity.loginurl=https://`*host*`:`*port*`/`*opensso-uri*`/UI/Login`

`com.sun.identity.liberty.authnsvc.url`
Indicates the Liberty authentication service URL.

`com.sun.identity.liberty.wsf.version`
: Used to determine which version of the Liberty identity web services framework is to be used when the framework can not determine from the inbound message or from the resource offering. This property is used when OpenSSO Enterprise is acting as the web service client. The default version is 1.1. The possible values are 1.0 or 1.1.

`com.sun.identity.liberty.ws.soap.certalias`
: Value is set during installation. Client certificate alias that will be used in SSL connection for Liberty SOAP Binding.

`com.sun.identity.liberty.ws.soap.messageIDCacheCleanupInterval`
: Default value is 60000. Specifies the number of milliseconds to elapse before cache cleanup events begin. Each message is stored in a cache with its `ownmessageID` to avoid duplicate messages. When a message's current time less the received time exceeds `thestaleTimeLimit` value, the message is removed from the cache.

`com.sun.identity.liberty.ws.soap.staleTimeLimit`
: Default value is 300000. Determines if a message is stale and thus no longer trustworthy. If the message timestamp is earlier than the current timestamp by the specified number of milliseconds, the message the considered to be stale.

`com.sun.identity.liberty.ws.wsc.certalias`
: Value is set during installation. Specifies default certificate alias for issuing web service security token for this web service client.

`com.sun.identity.liberty.ws.trustedca.certaliases`
: Value is set during installation. Specifies certificate aliases for trusted CA. SAML or SAML BEARER token of incoming request. Message must be signed by a trusted CA in this list. The syntax is:

```
cert alias 1[:issuer 1]|cert alias 2[:issuer 2]|.....
```

For example: `myalias1:myissuer1|myalias2|myalias3:myissuer3`. The value issuer is used when the token doesn't have a KeyInfo inside the signature. The issuer of the token must be in this list, and the corresponding certificate alias will be used to verify the signature. If KeyInfo exists, the keystore must contain a certificate alias that matches the KeyInfo and the certificate alias must be in this list.

Keeping the Client SDK Cache Up to Date

Notifications enable the synchronization of the client-side cache of remote services or applications. You can use the Notification Service to send session notifications to web containers that are running the OpenSSO Enterprise Client SDK. Session notifications enable real-time updates on the clients. No client application changes are required to support session notifications.

Notifications can be enabled for the Session, Policy, Identity Repository, SMS, and Naming Services. Set up a notification URL that the OpenSSO Enterprise server can use to send change notifications to the clients. Notifications work for web applications and standalone applications that can listen on a port for HTTP(s) traffic. However, a drawback to using this mechanism is that a constant flooding of notification changes, especially for user attribute changes, could overwhelm the clients.

The remote service or application must be running in a web container. The notifications can be received only if the Client SDK is installed on a web container.

▼ To Enable the Notification Service

Before You Begin Copy the encryption value of `am.encryption.pwd` from the OpenSSO Enterprise server to the remote Client SDK. The value of `am.encryption.pwd` is used for encrypting and decrypting passwords.

To access the `am.encryption.pwd`, in OpenSSO Enterprise administration console, click Configuration > Servers and Sites > `serverName` > Security.

1 **Install OpenSSO Enterprise on Host 1.**

2 **Install Sun Java System Web Server on Host 2.**

3 **Install the `ClientSDK` on the same machine as the Web Server.**

4 **Log in to OpenSSO Enterprise as `amadmin`.**

`http://`*OpenSSO-HostName*`:8080/opensso`

5 **Execute the servlet by entering `http://`*ClientSDK_host*`:8080/servlet/SSOTokenSampleServlet` into the browser location field and validating the `SSOToken`.**

`SSOTokenSampleServlet` is used for validating a session token and adding a listener. Executing the servlet will print out the following message:

```
SSOToken host name: 192.18.149.33 SSOToken Principal name:
uid=amAdmin,ou=People,dc=red,dc=iplanet,dc=com Authentication type used: LDAP
IPAddress of the host: 192.18.149.33 The token id is
AQIC5wM2LY4SfcyURnObg7vEgdkb+32T43+RZN30Req/BGE= Property: Company is - Sun
Microsystems Property: Country is - USA SSO Token Validation test Succeeded
```

6 **Set the property `com.iplanet.am.notification.url=` in the machine where the Client SDK is installed:**

`com.iplanet.am.notification.url=http://`*clientSDK_host.domain:port*
`/servlet`

```
com.iplanet.services.comm.client.PLLNotificationServlet
```

The notification URL is where the OpenSSO server can send change notifications to the clients. This works for web application and standalone applications that can listen on port for HTTP(s) traffic.

7 **Restart the Web Server.**

8 **Login into OpenSSO Enterprise as `amadmin`.**

`http://`*OpenSSO-HostName*`:8080/opensso`

9 **Execute the servlet by entering `http://`*ClientSDK_host*`:8080/servlet/SSOTokenSampleServlet` into the browser location field and validating the `SSOToken` again.**

When the machine on which the Client SDK is running receives the notification, it will call the respective listener when the session state is changed. The notifications can be received only if the Client SDK is installed on a web container.

Setting Up a Client SDK Identity

Some OpenSSO Enterprise components (such as SAML, user management, and policy) require an identity to be authenticated before the client application can read configuration data. The client can provide either a username and password that can be authenticated, or an implementation of the `com.sun.identity.security.AppSSOTokenProvider` interface. Either option will return a session token which the client can then use to access OpenSSO Enterprise configuration data.

- "To Set Username and Password Properties" on page 226
- "To Set an SSO Token Provider" on page 227

To Set Username and Password Properties

The following properties in `AMConfig.properties` can be used to set the username and password. The authenticated username should have permission to read the OpenSSO Enterprise configuration data.

- The property to provide the user name is `com.sun.identity.agents.app.username`.
- The property to provide the plain text password is `com.iplanet.am.service.password`.

Note – If a plain text password is a security concern, an encrypted password can be provided as the value of `com.iplanet.am.service.secret`. If an encrypted password is provided, the encryption key must also be provided as the value of `am.encryption.pwd`.

To Set an SSO Token Provider

Provide the implementation of the `com.sun.identity.security.AppSSOTokenProvider` interface as the value of the `com.sun.identity.security.AdminToken` property.

Using the OpenSSO Enterprise Virtual Federation Proxy Client API

OpenSSO Enterprise contains both Java and .Net interfaces to enable applications using Virtual Federation Proxy, which is sometimes referred to as Secure Attribute Exchange (SAE). The scenarios in this section use the following applications.

- An Identity Provider (IDP) application that generates SAE data to be sent to a local OpenSSO Enterprise instance in the IDP role (OpenSSO Enterprise IDP), acting as gateway to a SAMLv2–based circle of trust (COT)
- A Security Provider (SP) application to consume SAE data sent to it by a local OpenSSO Enterprise instance in the SP role (OpenSSO Enterprise SP)

The SAE protocol used to transfer SAE data between these entities is based on a simple HTTP-GET and HTTP-redirect:

- The Java API is provided in `fmsae.jar`.
- The .Net API is provided in `fmsae.dll`.

Two methods of securing SAE data are provided:

- Symmetric method, which requires a shared secret between the application and OpenSSO Enterprise
- Asymmetric digital signing method, which uses X509 certificates

These sections provide an introduction to SAE:

- "IDP Application Typical Usage (Asserting Party)" on page 228
- "SP application Typical Usage (Relying Party)" on page 229
- "Single Logout (SLO): IDP application Initiated Global Logout" on page 229

For more information, see the SAE sample and the OpenSSO Enterprise 8.0 documentation, and specifically the *Java API Reference*:

`http://docs.sun.com/coll/1767.1`

IDP Application Typical Usage (Asserting Party)

1. Perform these steps once:
   a. Register the application name, SAE security type (asymmetric or symmetric), and the corresponding shared secret or public key alias with the local OpenSSO Enterprise IDP.
   b. Store the shared secret in a safe place, preferably encrypted. The private key must be stored in a protected keystore and not revealed to OpenSSO Enterprise. OpenSSO Enterprise needs to know only the public key, which must be added to the OpenSSO Enterprise local keystore.
2. Initialize the `SecureAttrs` class if the Asymmetric method is used. The type `SecureAttrs.SAE_CRYPTO_ASYMMETRIC` must be specified as part of initialization; otherwise, the default is `SecureAttrs.SAE_CRYPTO_SYMMETRIC`.
   - Java: `SecureAttrs` class needs enough information to open a `java.security.KeyStore` instance that contains the keys needed to secure attributes to be sent to the OpenSSO Enterprise IDP. For more information about parameters. see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.
   - .Net: `SecureAttrs` class needs access to an appropriate `System.Security.Cryptography.AsymmetricAlgorithm` instance.
3. Authenticate the user.
4. Construct a Map (Java) or Hashtable (.Net) representing the user attributes to be asserted. The following attribute names are predefined:
   - `sun.userid`: String representing the authenticated user
   - `sun.spappurl`: SP application to be invoked
   - `sun.idpappurl`: Application name
   - `sun.cmd`: used to convey the “logout” message

   You can add any number of custom attributes. Embedded `|`and `=` characters must be escaped.
5. Retrieve shared secret if Symmetric method is used. The Asymmetric method needs the private key alias.
6. Execute `SecureAttrs.getInstance(`*type*`).getEncodedString()` (Java) or `SecureAttrs.getEncodedString(`*type*`)` (.Net), passing the user attributes and shared secret or private key alias.

   where *type* is `SecureAttrs.SAE_CRYPTO_ASYMMETRIC` or `SecureAttrs.SAE_CRYPTO_ASYMMETRIC`.
7. The application can choose to send the SAE data to a local OpenSSO Enterprise instance immediately or to construct a link that the user can click later. An example URL is:

   OpenSSO-instance-base-url`/saml2/jsp/SA+IDP.jsp?sun.data=`*SAE-data*

SP application Typical Usage (Relying Party)

1. Perform these steps once:
   a. Register the application name, SAE security type (symmetric or asymmetric), and the corresponding shared secret.
   b. Store the shared secret in a safe place, preferably encrypted.
   c. Retrieve the OpenSSO Enterprise public key and store it in a local keystore.
2. Initialize the `SecureAttrs` class, if the asymmetric method is used.
   - Java: `SecureAttrs` class requires enough information to open a `java.security.KeyStore` instance that contains the keys needed to verify attributes from local OpenSSO Enterprise SP. For more information about parameters. see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.
   - .Net: `SecureAttrs` class needs access to an appropriate `System.Security.Cryptography.AsymmetricAlgorithm` instance.
3. Be prepared to receive an `HTTP-GET` with the `sun.data` query parameter. The `sun.data` parameter contains `Base64` encoded SAE data.
4. Retrieve the shared secret, if the symmetric method is used. For the asymmetric method be prepared to go with the key alias of the OpenSSO Enterprise public key.
5. Execute `SecureAttrs.getInstance(`*type*`).verifyEncodedString(`*type*`)` (Java) or `SecureAttrs.verifyEncodedString()` (.Net) to verify the SAE data.
6. If the SAE data verification is successful, the returned Map contains user data that can be used to establish a local user session.

Single Logout (SLO): IDP application Initiated Global Logout

1. Register the SP and IDP application, as described as in previous use cases. The SP application additionally configures `saeSPLogoutURL`on the local OpenSSO Enterprise server.
2. The IDP application initiates the single logout by securely sending the `sun.cmd=logout` to the local OpenSSO Enterprise IDP instance.
3. The OpenSSO Enterprise IDP executes the SAMLv2 single logout protocol to the OpenSSO Enterprise SP.
4. The OpenSSO Enterprise SP picks up the saeSPLogoutURL configuration and redirects to it.
5. The `saeSPLogoutURL` logic can verify the `sun.data` parameter sent to it, which contains two SAE parameters: `sun.cmd=logout` and `sun.returnurl`.

After executing the local processing (such as invalidating the local application session and performing any cleanup tasks), the application must redirect back to the URL specified in `sun.returnurl`. Otherwise, the global logout across the SAMLv2 COT will terminate.

Also, ensure that the `saeSPLogoutURL` is up and running all the time; otherwise, the single logout will terminate with an error.

CHAPTER 14

Reading and Writing Log Records

Sun™ OpenSSO Enterprise provides the Logging Service to record information such as user activity, traffic patterns, and authorization violations. This chapter describes how to implement and customize the logging functionality, including:

About the Logging Service

When processing a logging request, the Logging Service extracts information from a user's session data structure and writes it to the configured log format, which can be either a flat file or a relational database. For example, this information can include access denials and approvals, authentication events, and authorization violations.

Administrators can then use the logs to track user actions, analyze traffic patterns, audit system usage, review authorization violations, and troubleshoot. Logged information is recorded in a centralized directory; which by default, is:

ConfigurationDirectory`/`*depoly-uri*`/log`

- *ConfigurationDirectory* is the name of the directory specified during the initial configuration of OpenSSO Enterprise server instance using the Configurator. This directory is created in the home directory of the user who ran the Configurator.
- *deply-uri* is the OpenSSO Enterprise deployment descriptor.

For example: `/opensso/opensso/log`

For more information about user sessions and the session data structure, see Chapter 6, "Models of the User Session and Single Sign-On Processes," in *Sun OpenSSO Enterprise 8.0 Technical Overview*.

For information about how the Logging Service works, see Chapter 15, “Recording Events with the Logging Service,” in *Sun OpenSSO Enterprise 8.0 Technical Overview*.

Using the Logging Interfaces

The Logging Service contains both an application programming interface (API) and service provider interface (SPI). You can use the logging APIs to add logging functionality to a client application and the SPIs to develop custom plug-ins to add functionality to the Logging Service.

Implementing Logging with the Logging Service API

The Logging Service API provides the interfaces for the OpenSSO Enterprise internal services and remote applications running the Client SDK to create and submit log records. Retrieving log records cannot be done using the client SDK. The logging API is in the `com.sun.identity.log` package, which is documented in the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

The Logging Service API extends the core logging APIs in the Java SE. For more information about the Java SE APIs, see `http://java.sun.com/javase/reference/index.jsp`.

The following sections have more information.

- “Writing Log Records” on page 232
- “Reading Log Records” on page 234

For more information see the *Sun OpenSSO Enterprise 8.0 Java API Reference*.

Writing Log Records

When writing log records, the Logging Service verifies that the logging requester has the proper authority to log and then writes the information to the configured location, formatting and completing the columns in the log records.

An application makes logging calls using the `getLogger()` method, which returns a `Logger` object. Each `Logger` keeps track of a log level and discards log requests that are below this level. (There is one `Logger` object per log file.) The applications allocates a `LogRecord`, which is written to the log file using the `log()` method. An `SSOToken`, representing the user's session data, is passed to the `LogRecord` constructor and used to populate the appropriate fields to be logged.

OpenSSO Enterprise contains plug-ins to write log records to:

- The host's flat file system
- The host's flat file system with added signing of the `LogRecord` and periodic verification

- A relational database
- A remote instance of OpenSSO Enterprise

The Logging Service requires two session tokens:

- Creating the `LogRecord` requires an `SSOToken` for the subject about whom the `LogRecord` is being written.
- Writing the `LogRecord` requires an `SSOToken` for the entity requesting the logging of the record.

Note – If your application also invokes utilities that log without using the OpenSSO Logging Service API, then you might also need to include the following:

```
import com.sun.identity.log.Logger;
Logger.token.set(ssoToken);
```

where `ssoToken` is the SSOToken of the entity requesting the logging. Also, once done, the following statement should be executed:

`Logger.token.set(null);` to clear the entity's SSOToken from the Logging Service.

The following parameters can have values logged when the `addLogInfo()` method is invoked. All columns except for *time*, *Data*, and *NameID* can be selected for exclusion from the record written to the log file.

time	The date and time is retrieved from OpenSSO Enterprise and added by the Logging Service.
Data	The event being logged as defined in the `message` string specified in the `LogRecord()` constructor call.
ModuleName	The value specified for the `LogConstants.MODULE_NAME` property in the `addLogInfo()` call. For example, the RADIUS module might be specified in an authentication attempt. **Note –** If no value is specified, this field will be logged as *Not Available*.
MessageID	The value specified for the `LogConstants.MESSAGE_ID` property in an `addLogInfo()` call. **Note –** If no value is specified, this field will be logged as *Not Available*.

Domain	The value for this field is extracted from the `SSOToken` and corresponds to either the subject userID's domain, or organization.
ContextID	The value for this field is extracted from the `SSOToken` and corresponds to the subject userID's session context.
LogLevel	The logging level, passed to the `LogRecord()` constructor, at which this record is being logged.
LoginID	The value for this field is extracted from the `SSOToken` and corresponds to the subject userID's Principal name.
NameID	The value specified for the `LogConstants.NAME_ID` property in an `addLogInfo()` call. It is an alias that maps to the actual userID.

Note – If no value is specified, this field will be logged as *Not Available*.

IPAddr	The value for this field is extracted from the `SSOToken` and corresponds to the originating point of the action being logged.
LoggedBy	The identifier in this field is extracted from the logging requestor's `SSOToken` specified in the `Logger.log()` call.
HostName	The host name corresponding to the originating point of the action being logged is derived from the *IPAddr* in the user's `SSOToken`, if it can be resolved.

Note – Resolving host names is disabled by default; enable this feature by toggling the Log Record Resolve Host Name system configuration attribute under Logging Service. If disabled, the *HostName* value is taken from the user's `SSOToken` and the `IPAddr` value is logged as *Not Available*.

Reading Log Records

When handling log reading requests, a valid `SSOToken` must be provided. The Logging Service verifies that the requester has the proper authority, and then it retrieves the requested records from the configured log location. The `LogReader` class provides the mechanism to read a log file and return the appropriate data to the caller. It provides the authorization check, reads the data, applies the query (if any), and returns the result as a string. The `LogQuery` is constructed using the `getLogQuery()` method.

Note – Reading log records from a remote client program using the client SDK is not supported.

Unless all records from a log file are to be retrieved, at least one `LogQuery` must be constructed. The `LogQuery` objects qualify the search criteria.

A `LogQuery` can specify a list of `QueryElements`, each containing a value for a field (column) and a relationship. The `QueryElement` supports the following relationships:

`QueryElement.GT`	Greater than
`QueryElement.LT`	Less than
`QueryElement.EQ`	Equal to
`QueryElement.NE`	Not equal to
`QueryElement.GE`	Greater than or equal to
`QueryElement.LE`	Less than or equal to
`QueryElement.CN`	Contains
`QueryElement.SW`	Starts with
`QueryElement.EW`	Ends with

Caution – Log files and tables in particular can become very large. If you specify multiple logs in a single query, create queries that are very specific or limited in the number of records to return (or both specific and limited). If a large number of records are returned, the OpenSSO Enterprise resource limits (including those of the host system) can be exceeded.

The following sample code queries for all successful authentications in realm `dc=example,dc=com`, and returns the `time`, `Data`, `MessageID`, `ContextID`, `LoginID`, and `Domain` fields, sorted on the `LoginID` field:

```
ArrayList al = new ArrayList();
al.add (LogConstants.TIME);
al.add (LogConstants.Data);
al.add (LogConstants.MESSAGE_ID);
al.add (LogConstants.CONTEXT_ID);
al.add (LogConstants.LOGIN_ID);
al.add (LogConstants.DOMAIN);
LogQuery lq = new LogQuery(LogQuery.ALL_RECORDS,
    LogQuery.MATCH_ALL_CONDITIONS,
    LogConstants.LOGIN_ID);

QueryElement qe1 = new QueryElement(LogConstants.MESSAGE_ID,
    "AUTHENTICATION-105",
    QueryElement.EQ);
lq.addQuery(qe1);
```

```
QueryElement qe2 = new QueryElement(LogConstants.DOMAIN,
    "dc=example,dc=com",
    QueryElement.EQ);
lq.addQuery(qe2);
```

In this code, assuming that `dc=example,dc=com` is the root realm, changing the `qe2` relationship field to `QueryElement.EW` or `QueryElement.CN` changes the query to include all successful authentications in all realms. To read the example query from the `amAuthentication.access log`, assuming presence of an `SSOToken`, add the following:

```
 String[][] result = new String[1][1];
    result = read("amAuthentication.access", lq, ssoToken);
```

The first record in a log (row 0) contains the field and column names.

Implementing Remote Logging

- "Logging to a Second OpenSSO Enterprise Server Instance" on page 236
- "Logging to OpenSSO Enterprise Server From a Remote Client" on page 237

Logging to a Second OpenSSO Enterprise Server Instance

An OpenSSO Enterprise server instance can use the Logging Service of another OpenSSO Enterprise server instance, if both instances are configured as part of the same site. The remote OpenSSO Enterprise server sets its `Logging Service URL` in the Administration Console (Configuration > System > Naming) to the target OpenSSO Enterprise server instances's Logging Service, by changing the attribute's `protocol`, `host`, `port`, and `uri` values, accordingly. For example:

```
https://ssohost2.example.com:58080/opensso/loggingservice
```

Note – Reading log records remotely from another server or from a client program using the client SDK is not supported.

Logging to OpenSSO Enterprise Server From a Remote Client

A remote client can use the OpenSSO Enterprise client SDK to log to an OpenSSO Enterprise server. In order for the remote client to log to the target OpenSSO Enterprise server, the entity making the logging request must have Log Writing permission on the target OpenSSO Enterprise server. For information, see "Running the Command-Line Logging Sample (`LogSample.java`)" on page 237.

Running the Command-Line Logging Sample (`LogSample.java`)

OpenSSO Enterprise provides a command-line logging sample to show log writing from a client using the OpenSSO client SDK. This sample (and other samples) are in the `opensso-client.zip` file, which is part of the `opensso_enterprise_80.zip` file.

After you unzip `opensso-client.zip`, the command-line logging sample is:

- Solaris and Linux systems: *opensso-client-zip-root*`/sdk/scripts/CommandLineLogging.sh`
- Windows systems: *opensso-client-zip-root*`\sdk\scripts\CommandLineLogging.bat`

opensso-client-zip-root is where you unzipped the `opensso-client.zip` file.

The command-line logging sample runs in a stand alone JVM and does not require a web container.

To run the command-line logging sample, OpenSSO Enterprise server must be running and accessible from the client server. You will also need to know this information:

- Protocol (`http` or `https`) used by the OpenSSO Enterprise server web container instance.
- Fully qualified domain name (FQDN) of the OpenSSO Enterprise server host.
- Port for the OpenSSO Enterprise server.
- Deployment URI for the OpenSSO Enterprise server (default is `opensso`).
- Default agent user (`UrlAccessAgent`) password that you entered when you ran the OpenSSO Enterprise Configurator.
- OpenSSO Enterprise server `amadmin`password, if `amadmin` is the logging requestor in the sample.

▼ To Run the Command-Line Logging Sample

1 **If necessary, unzip** `opensso_enterprise_80.zip` **and then unzip** *zip-root*`/opensso/samples/opensso-client.zip`**.**

2 **Make sure that your** `JAVA_HOME` **environment variable points to a JDK 1.5 or 1.4 installation.**

3 **Change to the** *opensso-client-zip-root*`/sdk` **directory.**

Note: You can invoke the sample scripts only from the `/sdk` parent directory and not directly from the `/scripts` directory.

4 **Follow the instructions in the README file to configure the** `AMConfig.properties` **file and to setup and compile the sample applications.**

Note: You need to setup and compile the sample command-line applications only once. If the sample applications are already compiled, continue with the next step.

5 **Run the sample command-line logging sample script from the** `/sdk` **directory. For example:**

- Solaris and Linux systems: `scripts/CommandLineLogging.sh`
- Windows: `scripts\CommandLineLogging.bat`

6 **The logging sample program prompts you for the subject user's identifier and password, log file to use, message to log, logging user's identifier and password, and the realm (both users must be in the same realm).**

Either accept the default values for the prompts or specify your preferred values. For example:

```
Subject Userid [user1]: accepted default
Subject Userid user1 password [user1password]: user1-password
Log file [TestLog]: accepted default
Log message [Test Log Record]: accepted default
LoggedBy Userid [amadmin]: accepted default
LoggedBy Userid’s password [amadminpswd]: amadmin-password
Realm [/]: accepted default
==>Authentication SUCCESSFUL for user user1
==>Authentication SUCCESSFUL for user amadmin
LogSample: Logging Successful !!!
```

7 **Check the** `TestLog` **created in the OpenSSO Enterprise server log directory.**

The default log directory is *ConfigurationDirectory*/*depoly-uri*/`log`.

For example: `/opensso/opensso/log`

APPENDIX A

Key Management

A public key infrastructure enables users on a public network to securely and privately exchange data through the use of a public and a private key pair that is shared using a trusted authority. For example, the PKI allows the data from a client, such as a web browser, to be encrypted prior to transmission. The private key is used to decrypt text that has been encrypted with the public key. The public key is made publicly available (as part of a digital certificate) in a directory which all parties can access. This appendix contains information on how to create a keystore and generate public and private keys. It includes the following sections:

- "Public Key Infrastructure Basics" on page 239
- "`keytool` Command Line Interface" on page 241
- "Setting Up a Keystore" on page 242

Public Key Infrastructure Basics

Web containers support the use of keystores to manage keys and certificates. The *keystore file* is a database that contains both public and private keys. Public and private keys are created simultaneously using the same algorithm (for example, RSA). A *public key* is used for encrypting or decrypting information. This key is made known to the world with no restrictions, but it cannot be used to decrypt information that the same key has encrypted. A *private key* is never revealed to anyone except it's owner and does not need to be communicated to third parties. The private key might never leave the machine or hardware token that originally generated it. The private key can encrypt information that can later be decrypted by using the public key. Also the private key can be used to decrypt information that was previously encrypted using the public key.

A public key infrastructure (PKI) is a framework for creating a secure method of exchanging information on an unsecure network. This ensures that the information being sent is not open to eavesdropping, tampering, or impersonation. It supports the distribution, management, expiration, rollover, backup, and revoking of the public and private keys used for public key cryptography. *Public key cryptography* is the most common method for encrypting and decrypting a message. It secures the data involved in the communications by using a private key and its public counterpart. Each entity protects its own private key while disseminating its public key for all to use. Public and private keys operate inversely; an operation performed by one key can be reversed, or checked, only by its partner key.

Note – The Internet X.509 Public Key Infrastructure Certificate and CRL Profile is a PKI.

Digital Signatures

So, a private key and a public key can be used for simple message encryption and decryption. This ensures that the message can not be read (as in eavesdropping) but, it does not ensure that the message has not been tampered with. For this, a *one-way hash* (a number of fixed length that is unique for the data to be hashed) is used to generate a digital signature. A *digital signature* is basically data that has been encrypted using a one-way hash and the signer's private key. To validate the integrity of the data, the server receiving the communication uses the signer's public key to decrypt the hash. It then uses the same hashing algorithm that generated the original hash (sent with the digital signature) to generate a new one-way hash of the same data. Finally, the new hash and the received hash are compared. If the two hashes match, the data has not changed since it was signed and the recipient can be certain that the public key used to decrypt the digital signature corresponds to the private key used to create the digital signature. If they don't match, the data may have been tampered with since it was signed, or the signature may have been created with a private key that doesn't correspond to the public key presented by the signer. This interaction ensures that any change in the data, even deleting or altering a single character, results in a different value.

Digital Certificates

A *digital certificate* is an electronic document used to identify an individual, a server, a company, or other entity and to bind that entity to a public key by providing information regarding the entity, the validity of the certificate, and applications and services that can use the certificate. The process of signing the certificate involves tying the private key to the data being signed using a mathematical formula. The widely disseminated public counterpart can then be used to verify that the data is associated with the sender of the data. Digital certificates are issued by a certificate authority (CA) to authenticate the identity of the certificate-holder both before the certificate is issued and when the certificate is used. The CA can be either independent third parties or certificate-issuing server software specific to an enterprise. (Both types issue, verify, revoke and distribute digital certificates.) The methods used to authenticate an identity are dependant on the policies of the specific CA. In general, before issuing a certificate, the CA must use its published verification procedures for that type of certificate to ensure that an entity requesting a certificate is in fact who it claims to be.

Certificates help prevent the use of fake public keys for impersonation. Only the public key certified by the certificate will work with the corresponding private key possessed by the entity identified by the certificate. Digital certificates automate the process of distributing public keys and exchanging secure information. When one is installed on your machine, the public key is freely available. When another computer wants to exchange information with your computer, it accesses your digital certificate, which contains your public key, and uses it to validate your identity and to encrypt the information it wants to share with you. Only your private key can decrypt this information, so it remains secure from interception or tampering while traveling across the Internet.

Note – You can get a digital certificate by sending a request for one to a CA. Certificate requests are generated by the certificate management tool used. In this case, we are using the `keytool` command line interface. When `keytool` generates a certificate request, it also generates a private key.

`keytool` Command Line Interface

`keytool` is a key and certificate management utility used to create the keys. It also manages a `.keystore` file containing private keys and the associated X.509 certificate chains authenticating the corresponding public keys, issues certificate requests (which you send to the appropriate CA), imports certificate replies (obtained from the contacted CA), designates public keys belonging to other parties as trusted, and generates a unique key alias for each *keystore entry*. There are two types of entries in a keystore:

- A keystore entry holds sensitive cryptographic key information, stored in a protected format to prevent unauthorized access. Typically, a key stored in this type of entry is a secret or private key accompanied by a certificate chain for the corresponding public key.
- A trusted certificate entry contains a single public key certificate belonging to another party. It is called a *trusted certificate* because the keystore owner trusts that the public key in the certificate indeed belongs to the identity identified by the *subject* of the certificate. The issuer of the certificate vouches for this, by signing the certificate.

To create a keystore and default key entry in `.keystore`, you must use `keytool`, available from the Java Development Kit (JDK), version 1.3.1 and above. For more details, see keytool — Key and Certificate Management Tool.

Setting Up a Keystore

The following procedure illustrates how to create a keystore file and default key entry using `keytool`.

▼ To Set Up a Keystore

Be sure to use the `keytool` provided with the JDK bundled with OpenSSO Enterprise. It is located in *JAVA_HOME*`/bin/keytool`. When installed using the Java Enterprise System installer, *JAVA_HOME* is */OpenSSO-base*`SUNWam/java`.

Note – The italicized option values in the commands used in this procedure may be changed to reflect your deployment.

---

1 **Generate a certificate using one of the following procedures.**

- **Generate a keystore with a public and private key pair and a self-signed certificate for your server using the following command.**

  ```
  keytool -genkey -keyalg rsa -alias test
  -dname "cn=sun-unix,ou=SUN Java System Access Manager,o=Sun,c=US"
  -keypass 11111111 -keystore .mykeystore
  -storepass 11111111 -validity 180
  ```

  This command will generate a keystore called `.mykeystore` in the directory from which it is run. A private key entry with the alias `test` is created and stored in `.mykeystore`. If you do not specify a path to the keystore, a file named `.keystore` will be generated in your home directory. If you do not specify an alias for the default key entry, `mykey` is created as the default alias. To generate a DSA key, change the value of `-keyalg` to `dsa`. This step generates a self-signed certificate.

- **Create a request and import a signed certificate from a CA (to authenticate your public key) using the following procedure.**

  a. **Create a request to retrieve a signed certificate from a CA (to authenticate your public key) using the following command:**

```
keytool -certreq -alias test -file request.csr -keypass 11111111 -keystore .mykeystore -storepass 11111111 -storetype JKS
```

  `.mykeystore` must also contain a self-signed certificate authenticating the server's generated public key. This step will generate the certificate request file, `request.csr`, under the directory from which the command is run. By submitting `request.csr` to a CA, the requestor will be authenticated and a signed certificate authenticating the public key will be returned. Save this root certificate to a file named `myroot.cer` and save the server certificate generated in the previous step to a file named `mycert.cer`.

b. **Import the certificate returned from the CA using the following command:**

```
keytool -import -alias test -trustcacerts -file mycert.cer -keypass 11111111 -keystore .mykeystore -storepass 111111
```

c. **Import the certificates of any trusted sites (from which you will receive assertions, requests and responses) into your keystore using the following command:**

```
keytool -import -file myroot.cer -keypass 11111111 -keystore .mykeystore -storepass 11111111
```

The data to be imported must be provided either in binary encoding format, or in printable encoding format (also known as *Base64*) as defined by the Internet RFC 1421 standard. In the latter case, the encoding must be bounded at the beginning by a string that starts with `-----BEGIN` and bounded at the end by a string that starts with `-----END`.

2 **Change to the** */OpenSSO-base/*`SUNWam/bin` **directory and run the following command:**

```
ampassword -e original password
```

This encrypts the password. The command will return something like `AQICKuNVNc9WXxiUyd8j9o/BR22szk8u69ME`.

3 **Create a new file named** `.storepass` **and put the encrypted password in it.**

4 **Create a new file named** `.keypass` **and put the encrypted password in it.**

5 **Copy** `.mykeystore` **to the location specified in** `AMConfig.properties`**.**

For example, if `com.sun.identity.saml.xmlsig.keystore=/etc/opt/SUNWam/lib/keystore.jks`, copy `.mykeystore` to `/etc/opt/SUNWam/lib/` and rename the file to `keystore.jks`.

6 **Copy** `.storepass` **and** `.keypass` **to the location specified in** `AMConfig.properties`**.**

For example, if `com.sun.identity.saml.xmlsig.storepass=/etc/opt/SUNWam/config/.storepass` and `com.sun.identity.saml.xmlsig.keypass=/etc/opt/SUNWam/config/.keypass`, copy both files to `/etc/opt/SUNWam/config/`.

7 **Define a value for the** `com.sun.identity.saml.xmlsig.certalias` **property in** `AMConfig.properties`**.**

For this example, the value would be `test`.

8 **(Optional) If the private key was encrypted using the DSA algorithm, change** `xmlsigalgorithm=http://www.w3.org/2000/09/xmldsig#rsa-sha1` **in** */OpenSSO-base/*`locale/amSAML.properties` **to** `xmlsigalgorithm=http://www.w3.org/2000/09/xmldsig#dsa-sha1`**.**

9 **(Optional) Change the canonicalization method for signing or the transform algorithm for signing by modifying `amSAML.properties`, located in** */OpenSSO-base*`/locale/`**.**

a. **Change** `canonicalizationMethod=http://www.w3.org/2001/10/xml-exc-c14n#` **to any valid canonicalization method specified in Apache XML security package Version 1.0.5.**

Note – If this entry is deleted or left empty, we will use `SAMLConstants.ALGO_ID_C14N_OMIT_COMMENTS` (required by the XML Signature specification) will be used.

---

b. **Change** `transformAlgorithm=http://www.w3.org/2001/10/xml-exc-c14n#` **to any valid transform algorithm specified in Apache XML security package Version 1.0.5.**

Note – If this entry is deleted or left empty, the operation will not be performed.

---

10 **Restart OpenSSO Enterprise.**

Index